Dossier de projet

Dossier de projet pour le Titre Professionnel de Concepteur-développeur informatique (RNCP6255) réalisé dans le cadre de la formation Bachelor développeur web PHP/Symfony, Studi.

Dossier et travaux réalisés par Sébastien Monterisi, entre février et août 2024.

Dossier réalisé conformément au modèle officiel.

Table des matières

Compétences du référentiel couvertes par le projet....5
Résumé du projet en anglais....6
Cahier des charges ou expression des besoins du projet....6
Document initial....6
Gestion de projet....7
Structuration du déroulement du projet....7
Roadmap....8
Planning....8
Méthodologies de développement....9
Environnement humain, technique et financier....10
Objectifs de qualité....10
Estimations de charge....10
Récapitulatif et limites de la gestion de projet....10
Spécifications fonctionnelles du projet....11
Spécifications techniques du projet....13
Architecture....13
Infrastructure....13
Déploiement....15
Langage serveur....16
Composants....17
Framework....17
Api platform....18
PostgreSQL....18
Nginx....18
Certbot – Let’s encrypt....18
Réalisations du candidat....19
Captures d’écran de l’application....19
Extraits du service de l’API....21
Interrogations de l’API....21
Exemples d’appels pour récupérer des données (méthode GET)....21
Exemple d’appel pour envoyer des données (méthode POST)....22
Code de l’API....23
Extraits d’un *DTO* et utilisation de twig....25
Exemples de DTO....25
Exemple d’utilisation d’un *DTO* dans un *template*....26

Exemple de DTO dans un contrôleur....27
Extrait d’une entité Doctrine/Api Platform....29
Extrait des fichiers Docker....31
Organisation des réseaux....32
Organisation des volumes....33
Env file....34
Extrait d’utilisation des outils de qualité....35
Php-cs-fixer....35
Rector....36
PhpStan....37
Utilisation des outils pendant le développement et l’intégration continue....38
Déploiement et intégration continue....39
Construction d’une application de type Desktop....43
Construction d’une application mobile....44
Composant d’accès d’une interface graphique avec accès à une base de donnée....45
Extrait d’une entité *Doctrine*....47
Présentation du jeu d’essai....49
Jeu de test du client web....49
Tests de non régression....50
Tests sur l’authentification....50
Test sur l’envoi de données....52
Test sur la récupération de données....56
Jeu de test de l’API....56
Tests fonctionnels....57
Test de récupération de données....57
Test d’envoi de données....58
Test de modification de données interdite....60
Tests des entités *Doctrine*....61
Tests des validateurs....61
Veille de sécurité....63
Veille de sécurité back-end, dépendances Composer....63
Veille sur l’exécutable composer....64
Situation de recherche....65
Généralités....65
Recherche Concrète....65
Annexes....68
Énoncé du travail pour le projet en cours de formation (ecf)....68

Note de prédémarrage.....75
Note de cadrage.....79
Roadmap.....86
Planning.....87
Cahier des charges.....88
Document de conception.....101
Document de sécurité.....112
Diagramme MCD.....135
Diagramme cas d’utilisations.....136
Document de réalisation.....137

Compétences du référentiel couvertes par le projet

- Concevoir et développer des composants d’interface utilisateur en intégrant les recommandations de sécurité
  - Maquetter une application
  - Développer une interface utilisateur de type desktop
  - Développer des composants d’accès aux données
  - Développer la partie front-end d’une interface utilisateur web
  - Développer la partie back-end d’une interface utilisateur web
- Concevoir et développer la persistance des données en intégrant les recommandations de sécurité
  - Concevoir une base de données
  - Mettre en place une base de données
  - Développer des composants dans le langage d’une base de données
- Concevoir et développer une application multicouche, répartie, en intégrant les recommandations de sécurité
  - Collaborer à la gestion d’un projet informatique et à l’organisation de l’environnement de développement
  - Concevoir une application
  - Développer des composants métier
  - Construire une application organisée en couches
  - Développer une application mobile
  - Préparer et exécuter les plans de tests d’une application
  - Préparer et exécuter le déploiement d’une application
- Compétences transversales
  - Utiliser l’anglais dans son activité professionnelle en conception et développement d’applications
  - Actualiser et partager ses compétences en conception et développement d’applications

Résumé du projet en anglais

The project goal is to develop a medical appointment scheduling system. It is part of a training program provided by Studi. The client is a fictional hospital that faces issues in managing patient appointments. Currently, scheduling and booking are done manually, and the check-in and check-out processes are slow. Doctors do not have schedules. The hospital aims to solve these problems by implementing an online appointment system.

Develop-Solution is the company responsible for releasing the project. They have created an initial plan. Only one developer is assigned to this project. He is responsible for documentation, technical decisions, development, delivery, and deployment. All necessary documents are included.

To ensure standardization, maintenance, and continuity, the project uses recent and widely adopted technologies. Since hospitals are often targets for hackers, security has been a priority from the beginning.

The client's requirements were clarified through project management documents and wireframes.

The system has three client parts (web, mobile, and desktop) that share data with a server. This design allows each part to evolve independently. The project uses an API-first approach. The mobile and desktop applications are simple versions of the web client. In the future, more clients can connect directly to the API if needed.

Cahier des charges ou expression des besoins du projet

Document initial

Pour ce projet, un document de départ cadrant l’ensemble projet nous a été remis par *Studi* (document disponible en annexe, page 67) .
Il s’agit d’un document qui donne un contexte fictif, des besoins sommaires, quelques *user stories.* Il indique la finalité du projet et impose quelques choix techniques et documentaires. L’objectif est de mettre en œuvre l’ensemble des compétences du référentiel du diplôme afin de démontrer les compétences du développeur.

Concrètement, les détails sont donnés dans le résumé de projet ci-dessus. En résumé, il s’agit de concevoir une ébauche d’un système de prises de réservation de séjours des patients avec des spécialistes médicaux. L’hôpital n’a pas de système de prise de rendez-vous, ce projet amorce ainsi un projet sur le long terme.

Le cahier des charge est un livrable de la phase 3, « spécifications fonctionnelles et planning ». La section Gestion de projet, page 7, détaille à quelle moment du projet il à été écrit. Comme le projet est un début de solution, le cahier des charges est relativement simple. Après une présentation du contexte, il est structuré de cette façon :

- Définition des besoins fonctionnels et non fonctionnels
- Détermination des exigences techniques
- Détermination des livrables

- Planning prévisionnel
- Modalité de gestion de projet
- Identification et mesure d’atténuation des risques

Le cahier des charges initial laisse beaucoup de libertés. Une réflexion et des choix techniques de conception étaient nécessaires. Au-delà du cahier des charges fonctionnel et technique minimal livré, j’ai décidé d’adopter une conduite de projet plus structurée en rédigeant les documents essentiels qui jalonnent le pilotage du projet. J’avais besoin de ce cadrage pour éviter que le projet ne dérive. Le cahier des charges est un document central dans le projet.

J’ai rédigé ce cahier des charges en essayant de ne pas être trop verbeux pour donner au document une visée opérationnelle et contractuelle. J’ai bien gardé à l’esprit que ce document est essentiel en terme d’interface entre la maitrise d’œuvre et la maitrise d’ouvrage. Cependant, je reste développeur et pas chef de projet, donc le document reste assez basique.

Gestion de projet

Structuration du déroulement du projet

En m’inspirant des cours *Studi* et du livre « S'initier à la gestion de projets informatiques[1] » (ISBN : 978-2-409-02506-8), j’ai prévu les phases principales du projet :

1. Pré-démarrage
2. Cadrage
3. Spécifications fonctionnelles, planification & chiffrage
4. Conception technique
5. Réalisation technique
6. Tests et correctifs
7. Mise en production, formation, maintenance
8. Garanties

L’ achèvement de chaque phase est matérialisé par un livrable, un document et/ou du code. En termes de documentation pour y aboutir, j’ai réalisé une note de pré-démarrage (annexe page 74), une note de cadrage (annexe page 78), une roadmap (annexe page 85) et un planning (annexe page 86). J’ai ajusté le planning et le cahier des charges (annexe page 87) pendant leur écriture.

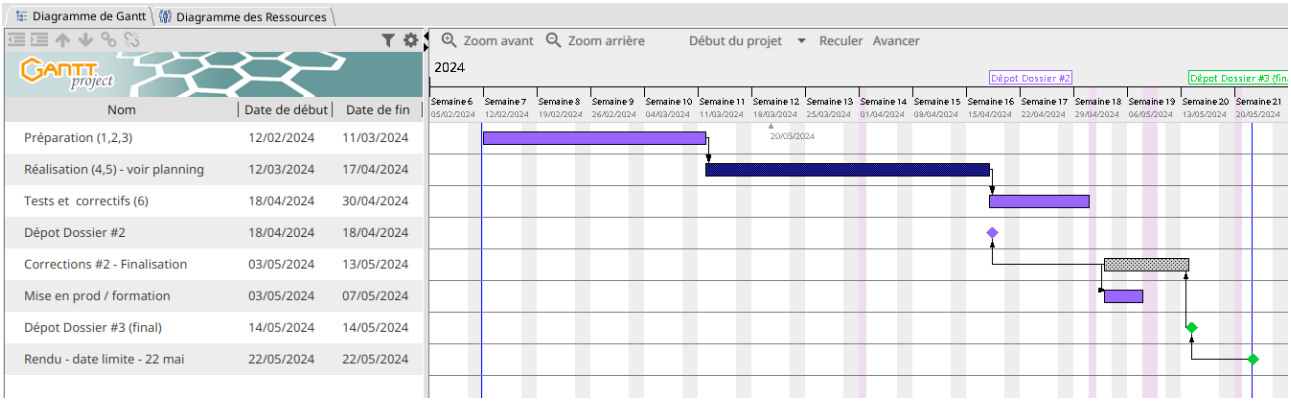
La phase de réalisation technique, la plus longue, lors de laquelle les applications sont développées et les environnement mis en place, j’ai utilisé deux méthodologies de travail. D’abord une méthode en cascade, puis une méthodologie agile. Je détaille ce point dans la section Méthodologies de développement, page 9.

1 https://www.editions-eni.fr/livre/s-initier-a-la-gestion-de-projets-informatiques-9782409025068

Travaillant seul, j'ai pu faire le choix de logiciel qui ne sont pas forcement pensés pour une collaboration en ligne. Pour la planification, j'ai utilisé *Ganttproject* (ganttproject.biz[2]). Pour la gestion des taches, j'ai utilisé un tableau kanban avec le logiciel *Kanboard* (kanboard.org[3]).

Roadmap

Lors de l'étape de cadrage, j'ai réalisé une roadmap, dont voici une capture d'écran :
J'ai donc regroupé les étapes en 3 phases : Préparation, Réalisation et Correctifs.



Dans le cadre d'un projet avec différentes parties prenantes (notamment une maîtrise d'œuvre et un maîtrise d'ouvrage distincte), ce découpage permettrait d'impliquer des acteurs différents et de leur donner une visibilité sur la conduite et l'avancement du projet.

La roadmap a été construite avec un jalon final inamovible, le rendu du projet, et deux jalons intermédiaires (rendus intermédiaires du projet).

Capture du document disponible en annexe page 85.

Planning

Le planning est un développement de la roadmap. Il a été réalisé après le cadrage et lors la rédaction des spécifications dans le cahier des charges.

Les 3 phases structurantes de la roadmap ont donc été déclinées en 7 phases concrètes.

- Préparation
  - 1. Pré-démarrage (*1.prédémarrage.odt, Roadmap.gan*)
  - 2. Cadrage (*2. cadrage.odt*)
  - 3. Spécifications fonctionnelles, planification & chiffrage (*3. cahier des charges, Planning.gan*)
- Développement
  - 4. Conception technique (*4. Conception technique.odt*)
  - 5. Réalisation technique (*5.1. Documentation technique*)
- Finalisation
  - 6. Tests et correctifs (*6.Sécurité.odt*)

2 https://www.ganttproject.biz/

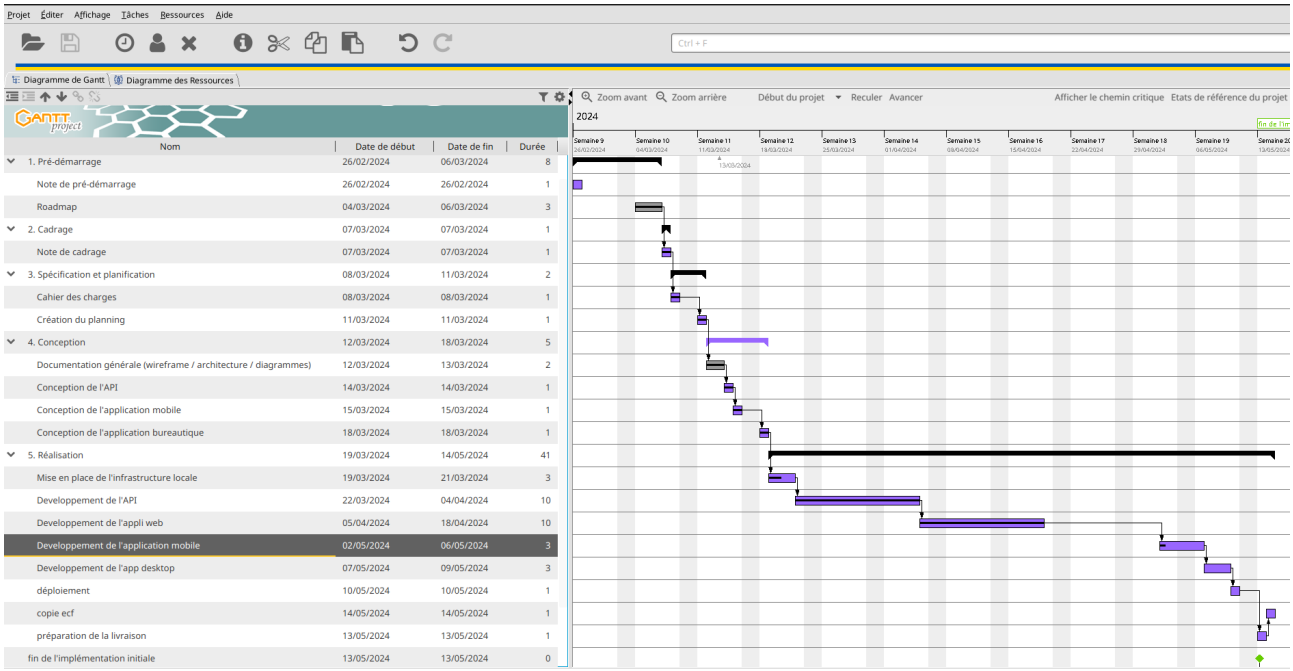
3 https://kanboard.org/

- 7. Mise en production, formation, maintenance (*6.Sécurité.odt et 7. Mise en production et maintenance.odt*)

Chacune des sept phases concrètes se matérialise par la livraison d'un ou plusieurs document (indiqué entre parenthèses plus haut).

Ces documents cadrent le projet, chaque phase s'appuyant sur le document précédemment livré. La réalisation de la note de cadrage et du cahier des charges a permis d'obtenir des informations factuelles pour élaborer un planning. La gestion des risques a révélé un besoin important de marges de sécurité et la nécessité de débuter rapidement le projet pour éviter de revoir les exigences (réduction du périmètre). En référence au triangle des contraintes (Coût / Qualité / Délai), j'ai choisi de prévoir un délai ample pour ne pas sacrifier la qualité. Quant au coût, je l'ai défini comme fixe.

La capture suivante est disponible en annexe, page 86.



La documentation pourra servir pour une analyse à postériori du projet. J'ai en effet limité la conduite du projet pour ne pas y consacrer trop de temps en choisissant de ne pas mettre en place de KPI. Pour compenser ce manque et tout de même réaliser une analyse rétrospective du pilotage et du travail, on pourra s'appuyer sur les documents réalisés. On pourra ainsi vérifier si les estimations, les délais et les spécifications ont été respectés.

Méthodologies de développement

Dans la phase de développement (5), j'ai distingué deux étapes : une étape avec une organisation en cascade traditionnelle, durant laquelle les grandes parties du projet ont été réalisées, et une phase plus agile pour la finalisation.

La gestion en cascade, avec des délais déterminés, est conçue pour assurer le respect des délais et couvrir toutes les composantes du projet, en implémentant toutes les exigences du cahier des charges. De cette façon, j'ai évité de continuer à consacrer du temps à une fonctionnalité déjà conforme alors que d'autres fonctionnalités essentielles n'étaient pas encore développées.

À l’inverse, dans la seconde phase, l’approche agile m’a semblé indiquée pour implémenter les corrections et les fonctionnalités notées pendant la première phase.

Environnement humain, technique et financier

L’environnement humain s’est limité à mon référent de formation pour les rendus et à moi-même en tant que développeur, chef de projet et client fictif. J’ai joué ces trois rôles. La réalisation des documents cités précédemment m’a permis d’endosser ces rôles, tout en les cloisonnant et en évitant les frictions liées à la communication.

L’environnement technique est défini par trois parties : les machines de développement, l’hébergement du code (et intégration continue), et les machines de production. Pour les machines de développement, j’ai utilisé mon ordinateur de bureau et acheté un ordinateur portable (pour le télétravail et la présentation du projet). Les deux machines fonctionnent avec *GNU/Linux* (*Manjaro* et *Fedora*), permettant une utilisation directe de *Docker*. Pour l’hébergement du code et l’intégration continue, je me suis appuyé sur *GitHub*. Pour les machines de production, j’ai profité des crédits étudiants offerts par *Digital Ocean*.

J’ai prévu de consacrer, 2 à 3 jours par semaine au projet et le reste à mes cours. Cela permet d’avoir une marge de manœuvre en changeant la répartition au besoin.

Objectifs de qualité

La durée indicative du projet est estimée à 70 heures dans le document fourni par *Studi*. Cela ne représente pas une quantité suffisante dans mon contexte. Il ne s’agissait pas pour moi de mettre en œuvre des compétences déjà éprouvées, mais au contraire de profiter du projet pour me former (particulièrement à *Symfony*), découvrir des outils, approfondir mes connaissances, m’initier à la conduite de projet. J’ai souhaité mettre en place des fondations solides et réutilisables, quitte à prendre plus de temps.

Les objectifs de qualité sont indiqués dans le cahier des charges, il s’agit d’avoir un système fonctionnel, le plus conforme possible aux spécifications fonctionnelles.

Estimations de charge

Au regard des objectifs de qualité, j’ai planifié environ 210 heures de travail, soit 29 jours complets.

Récapitulatif et limites de la gestion de projet

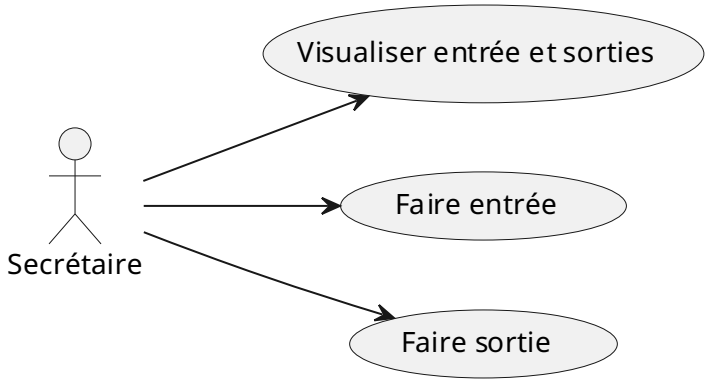
La réalisation s’est étalée sur 3 mois. J’ai appris beaucoup de choses et finalement consacré environ 450 heures au projet, bien au-delà de l’estimation initiale de 210 heures. Comme indiqué plus haut, ce dépassement était possible car les délais de rendu constituaient les impératifs du projet, sans contrainte de charge.

La roadmap initiale a été respectée, avec le rendu du projet final effectué trois jours avant l’échéance. Cependant, il est important de noter que l’intégralité du cahier des charges n’a pas été respectée à la date de rendu du projet. La fonctionnalité d’assignation de rendez-vous par un administrateur et la création de médecin (user story 5, dans le cahier des charges) sont manquantes.

Spécifications fonctionnelles du projet

Les spécifications fonctionnelles du projet on été relativement bien définies dans le document initial (énoncé du projet).

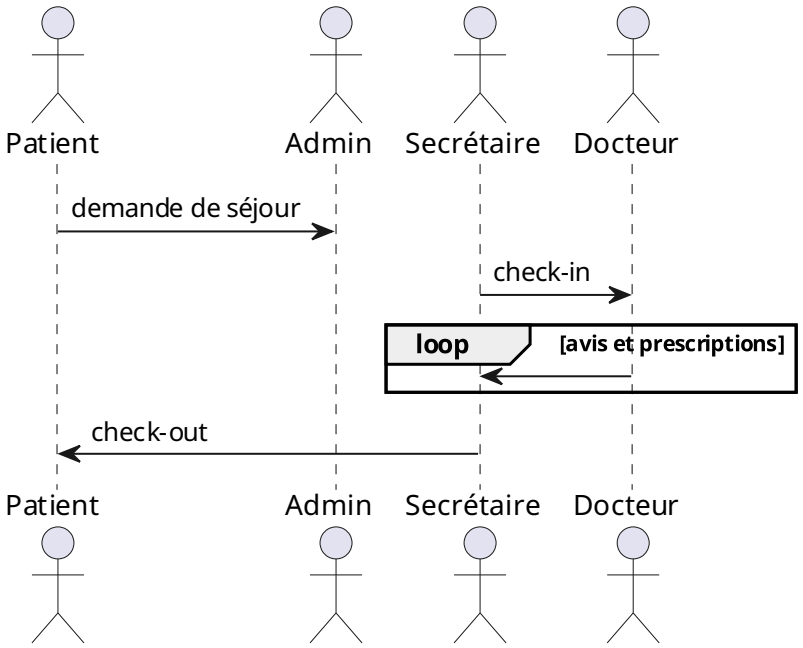
Pour les cas d'utilisation, j'ai réalisé des diagrammes qui reprennent les informations des *User stories*, sous une forme plus lisible et sans ambiguïté possible. Par exemple, ce diagramme des cas d'utilisation montre les actions possibles pour une secrétaire.



Tous les diagrammes sont disponibles en annexes (page 134).

Lors de la rédaction du cahier des charges, j'ai apporté quelques précisions et modifications pour améliorer la clarté et l'efficacité des exigences. À l'origine, la User Story 4 spécifiait que le bouton de confirmation ne devait être cliquable que si l'utilisateur était authentifié. J'ai modifié cette exigence en effectuant un filtrage plus en amont, rendant le formulaire accessible uniquement aux utilisateurs authentifiés en tant que patients. Cette modification répond à l'exigence initiale tout en simplifiant la gestion de l'authentification et en améliorant la sécurité.
De plus, la User Story 4 détaillait les actions à entreprendre et les ordres de saisie sans justification fonctionnelle apparente. J'ai supprimé ces contraintes pour me concentrer sur la fonctionnalité principale, afin d'éviter d'ajouter des contraintes d'UX à ce niveau de définition. Cela permet une plus grande flexibilité dans la conception de l'interface utilisateur, tout en maintenant les exigences fonctionnelles essentielles.

Dans le scénario 5 (*gestion des plannings des médecins*), j'ai précisé que « associer un emploi du temps » consiste à confirmer (et éventuellement à modifier) un rendez-vous. En plus des diagrammes de cas d'utilisation, un diagramme de séquence s'est avéré nécessaire pour clarifier les interactions. C'est le seul diagramme de séquence nécessaire et implémenté. Les diagrammes de cas d'utilisation ont le mérite d'offrir une lecture très rapide et sont simples à vérifier pour le développeur, je les ai donc implémentés aussi.



Utilisateur est utilisé indistinctement pour les visiteurs (non authentifié) et les patients (utilisateur authentifié avec le rôle de patient). J'ai clarifié le mot « utilisateur » en utilisant des rôles plus précis (Patient, Visiteur.)

Spécifications techniques du projet

J’ai élaboré les spécifications techniques en plusieurs étapes. Tout d’abord, en réalisant le document de pré-démarrage, j’ai fixé des lignes directrices. Voici un extrait du document :

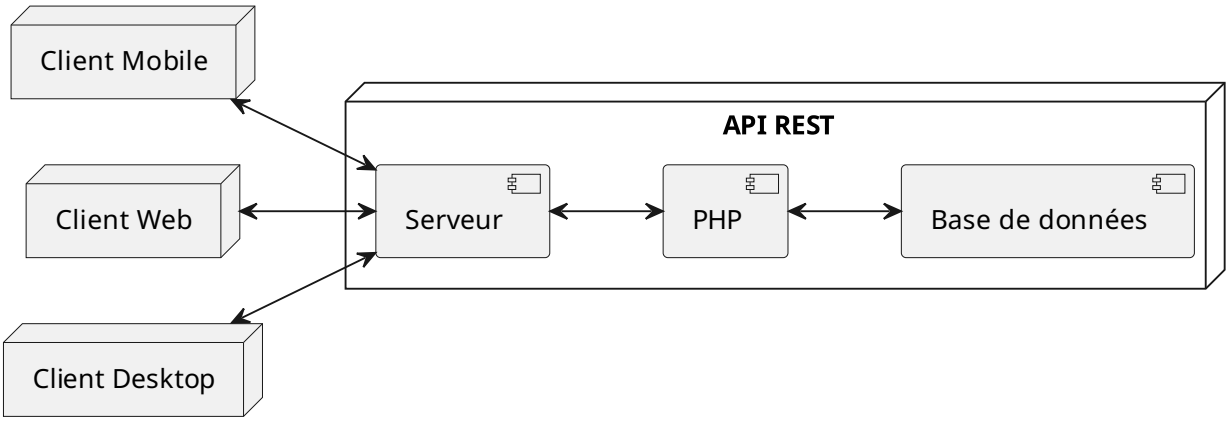
> Les choix techniques seront réalisés de façon à produire une solution robuste, standard et maintenable. Dans un soucis de réduction des temps de développement et de formation, les technologies de référence seront préférées.

Architecture

Cette architecture client/serveur a été décidée au moment de la conception technique. Elle aurait pu l'être au moment de la rédaction du cahier des charges, mais j'ai reporté ces choix au moment où l'ensemble des composants était connu. J'ai un temps envisagé de faire l'API et le client sous forme monolithique pour des raisons de facilité. Cependant, le besoin de disposer de plusieurs clients et de centraliser les données impose une architecture de type client/serveur (avec serveur centralisé). Pour l’évolutivité du projet, la séparation complète entre l’API du client web est aussi très avantageuse.

Pour garantir une implémentation aisée des clients, l'utilisation d'un standard s'est imposée (discuté dans la section Langage serveur), et j'ai choisi la spécification la plus répandue, la spécification *OpenAPI*[4].

J’ai formalisé l’architecture avec ce diagramme de composants *UML*.



Infrastructure

La mise en place des environnements de développement et de production peut-être problématique. Il y a des installations à faire, les versions logicielles et système peuvent différer (et introduire des différences de fonctionnement), les configurations peuvent varier et, marginalement, les systèmes d’exploitation peuvent fonctionner différemment. Les environnements du projet sont multiples : plusieurs machines de développement, une machine d'intégration et de déploiement continu, et les machines en production. Maintenir tout cela cohérent est compliqué, chronophage et constitue une source de risques très importante. Découvrir un dysfonctionnement qui se produit uniquement en production ne sera pas tolérable vis-à-vis des exigences de notre projet.

4 https://www.openapis.org/

Dans ce contexte, nous avons besoin d’environnement facilement reproductibles. La conteneurisation apporte une solution relativement simple à ces problèmes. La mise en place demande un peu de temps, mais celui-ci est largement compensé par l'évitement de problèmes inattendus et la facilité de réplication.

Docker est largement utilisé et constitue un standard dans le domaine ; sa maturité et sa robustesse ne sont plus à démontrer. Ce choix s'est fait rapidement. J'ai cependant effectué une recherche sur les alternatives et n'en ai pas trouvé de suffisamment répandue pour en justifier l'étude.

J'ai également souhaité que le déploiement reste simple. Comme la charge sur le serveur sera faible et que des interruptions de service peuvent être tolérées, orchestrer les conteneurs avec *Docker Compose* a été la solution choisie. Nous n'avons pas besoin d'une solution plus avancée comme *Kubernetes*, que nous pourrions mettre en place à l'avenir sans redéfinir complètement notre infrastructure. De plus, j'ai souhaité éviter d'introduire plus de risques en mettant en place une solution inconnue pour moi, ne présentant pas de bénéfice important. Le rapport bénéfice/risque plaide contre l'implémentation de Kubernetes.

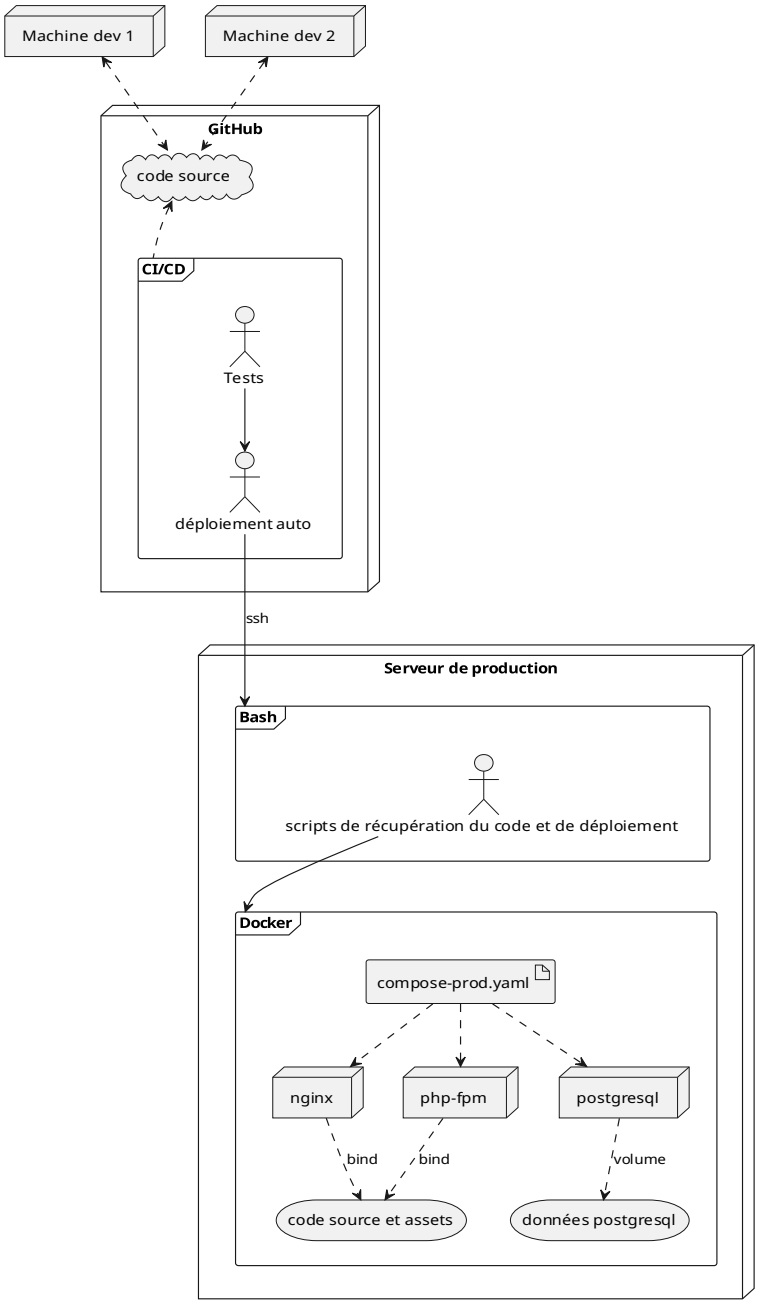
Finalement, la conteneurisation via Docker va permettre :

- une mise en place facile sur les différentes machines de développement
- une intégration simple dans le processus d’intégration et de déploiement continu
- une mise en production simplifiée et testée
- l’utilisation des mêmes versions dans tous les environnements
- un découplage des différents composants (base de données, serveur web, PHP), facilitant ainsi leur maintenance.

Déploiement

Comme expliqué plus haut, utiliser Docker doit permettre le simplifier le déploiement. Dans le document réalisé après le développement, intitulé "7. Mise en production et maintenance.odt", j'explique comment se déroule le processus de déploiement.
Tout est automatisé : dès que du code est poussé dans la branche principale, des tests et vérifications sont lancés via un script *Composer* nommé ci. En cas de réussite, le déploiement est effectué automatiquement, sans nécessiter d'action manuelle.

Au stade de spécification du projet, je me suis contenté de réaliser un diagramme de déploiement complet avec les éléments décrits précédemment. Bien que ce type de diagramme soit statique, on peut y voir le process de déploiement.



Langage serveur

PHP est un langage adapté pour réaliser une API ; il est mature, stable, et je le maîtrise très bien. De plus, c’est le langage le plus utilisé côté serveur. C'est pourquoi je l'ai choisi sans trop hésiter.

Composants

L'API et le client web sont réalisés avec *Symfony*[5], *API Platform*[6], *PostgreSQL*[7] et *Nginx*[8]. Les images présentes sur le *hub Docker*[9], sont supportés officiellement par *Docker*. La mise en production est réalisée sur un VPS (Virtual Private Server) utilisant *Ubuntu*[10].

Framework

Partir sur une solution «*from scratch »* ou de bas niveau n'est pas justifié, l'écosystème PHP dispose de plusieurs solutions fiables, flexibles et éprouvées qui nous facilitent le travaille tout en limitant fortement les problèmes de sécurité.

Plusieurs frameworks (de structure et de composants) sont possibles, Laravel et Symfony répondent aux exigences de maintenance et de popularité que j'ai définies initialement.

J'ai écarté le framework *Laravel*. En France, il y a largement plus de ressources (développeurs et agences) spécialisées en *Symfony* qu'en *Laravel*. J'ai établi ce fait en me basant sur les offres des sites d'emploi.[11]. *Laravel* est axé sur les développements rapides (« The PHP Framework for Web Artisans » est le slogan du site). Je recherche la solidité. La rapidité de mise en place n'est pas un critère déterminant. Je connais très peu cette solution, il m'est difficile d'évaluer la complexité du développement et de l'agencement des composants dans ce cadre. J'identifie un risque important de dérive du temps de développement malgré une mise en place rapide. Il est parfois compliqué d'utiliser Laravel lorsque nos besoins ne sont pas couverts par le framework, c'est ce que révèlent quelques expériences et tests personnels.

À l'inverse, *Symfony* se dédie plus à la création d'applications au niveau entreprise en offrant plus de flexibilité et en disposant d'un composant avancé pour la création d'API (API Platform). Le choix de *Symfony* est moins risqué, plus standard.

Finalement, j'ai choisi *Symfony* pour les raisons suivantes :

- large communauté professionnelle française
- composant dédié aux API reconnu et qualitatif (*Api Platform*)
- reconnu pour son coté professionnel
- flexibilité
- stabilité

La popularité Française du framework est déterminante puisque la continuité du projet, peut être assurée par un structure Française ou Francophone.

---

5 https://symfony.com/at-a-glance

6 https://api-platform.com/

7 https://www.postgresql.org/

8 https://nginx.org/

9 https://hub.docker.com/

10 https://ubuntu.com/server

11 https://fr.indeed.com/q-d%C3%A9veloppeur-php-laravel-emplois.html?vjk=cb4a39eeb58f14ae
https://fr.indeed.com/q-developpeur-symfony-emplois.html?vjk=7c47a408cf874bc0

Api platform

Api platform est un composant déterminant dans le choix du framework comme évoqué dans la section Framework, page 16 .

Ce composant répond à nos exigences initiales. Ce composant est construit sur *Symfony*, est mature et très bien documenté. Je me suis également assuré de la qualité du projet (car la popularité n'est pas toujours un gage de qualité) en effectuant quelques recherches sur les auteurs et les problèmes rencontrés par les développeurs. J'ai aussi étudié les alternatives avant de décider qu'API Platform était un bon choix.

PostgreSQL

Symfony, via *Doctrine*, peut fonctionner avec différents serveurs de bases de données. La base de données n'est pas utilisée de façon avancée ; l'accès est géré par *Doctrine*. Par défaut, *Symfony* est configuré pour utiliser *PostgreSQL*, qui est également le SGBD (Système de gestion de bases de données) utilisé dans les exemples du livre *The Fast Track*[12]. Comme évoqué précédemment, je m'efforce de me conformer aux standards et aux configurations initiales, ce qui incite donc à utiliser ce SGBD. Au-delà de cet argument, pas très convaincant, *PostgreSQL* est également plus performant, plus conforme aux standards SQL, et respecte mieux les règles ACID[13]. *PostgreSQL* est open source, c'est aussi un point décisif dans mon choix.

Étant donné que nous n'utiliserons pas de fonctionnalités avancées de PostgreSQL et que nous utilisons une couche d'abstraction (*Doctrine*) qui prend également en charge *MySQL* (entre autres), la transition, si elle devait être nécessaire, ne serait probablement pas très complexe. Le choix du SGBD n'est donc pas très contraignant.

Nginx

Le cœur de notre API et du client web fonctionne avec PHP-FPM (php-fpm.org[14]). PHP-FPM ne doit pas être exposé au web pour des raisons de sécurité ; nous avons donc besoin d'un serveur web intermédiaire. De plus, la sécurisation des communications par TLS (Transport Layer Security) nécessitera également un serveur.

Nginx et *Apache* sont les deux alternatives standards qui nous sont offertes. *Symfony* fonctionne très bien avec ces deux serveurs. J'ai choisi *Nginx*, qui est plus simplement configurable. Ce choix n'est pas très contraignant ; le changement de serveur n'entraînera pas de modification dans les autres couches de l'application.

Certbot – Let's encrypt

Pour la mise en place du TLS sur le serveur *Nginx*, nous avons besoin de certificats SSL. Let's Encrypt[15] s'est imposé comme service de génération des certificats pour plusieurs raisons. La mise en place est simple, l'image est sponsorisée par Docker officielle (certbot)[16], elle est bien documentée, et aucune ressource financière n'est demandée.

12 https://symfony.com/book

13 https://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s_ACID

14 https://php-fpm.org/

15 https://letsencrypt.org/about/

16 https://hub.docker.com/r/certbot/certbot

Réalisations du candidat

Captures d’écran de l’application

Figure 1: Accueil version desktop

Figure 2: Accueil version mobile

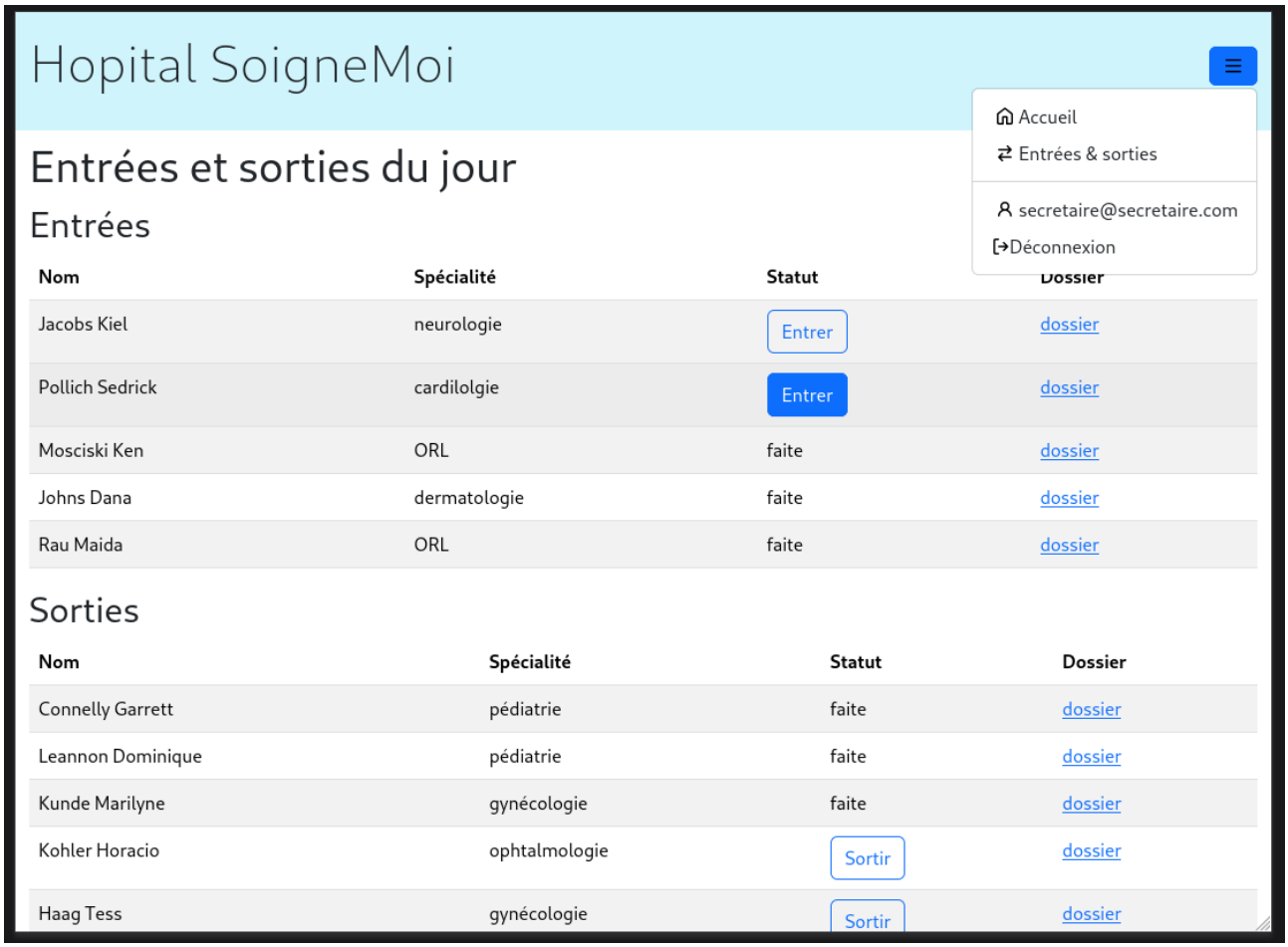


Figure 3: entrées et sortie pour un(e) secrétaire

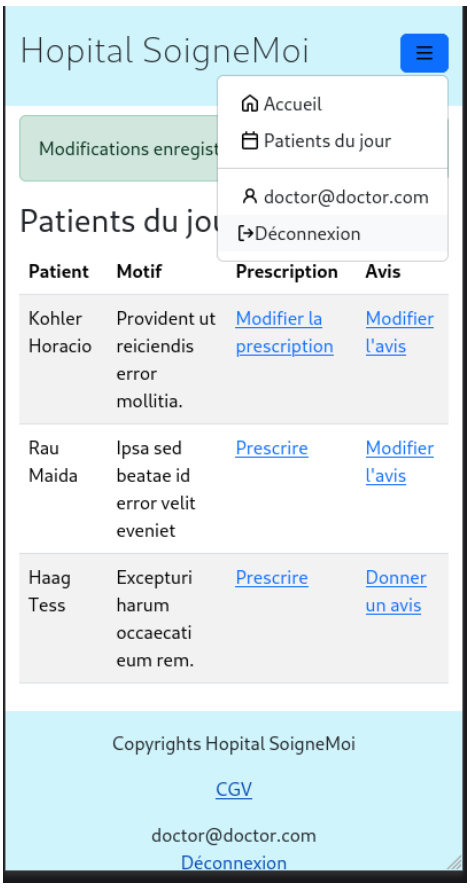


Figure 5: Patients à voir pour un docteur

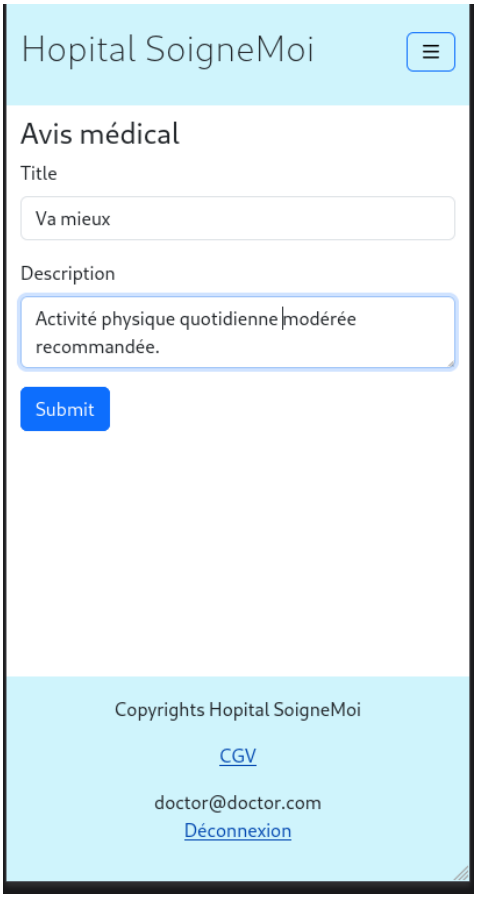


Figure 4: Création / Modification d'un avis par un docteur

Extraits du service de l'API

Interrogations de l'API

Le client web interroge l'API REST[17]. L'interrogation et la réception de la réponse ont lieu dans le contrôleur du client. L'interrogation nécessite de connaître le endpoint à atteindre, quels paramètres y passer, comment réaliser l'authentification, quels types de réponses sont possibles et comment elles sont formatées. Sans ajouter de couche d'abstraction, chaque appel nécessiterait de connaître les détails d'utilisation de l'API, rendant l'utilisation complexe, sujette à erreur, et finissant par faire grossir les contrôleurs. Il faut éviter cela et rendre l'utilisation de l'API la plus simple possible, et la moins sujette à une mauvaise utilisation. J'ai donc choisi d'encapsuler les appels à l'API et le traitement des réponses dans un composant. En procédant de la sorte, on obtient une unité de code isolée et testable. Si nous souhaitons réaliser une API dans le cadre d'un client mobile, dans un autre langage, nous pourrons nous référer à l'interface de ce composant. Je n'ai pas actuellement extrait les fonctions publiques du service dans une interface, pour éviter d'anticiper inutilement et de complexifier gratuitement mon travail. On peut voir que les appels dans un contrôleur sont ainsi simplifiés au maximum.

Exemples d'appels pour récupérer des données (méthode GET)

```
// src/Controller/SecretaryHomeController.php
class SecretaryHomeController extends AbstractController
{
    #[Route('/secretary/', name: 'app_secretary_home')]
    public function index(): Response
    {
        return $this->render('secretary/home.html.twig', [
            'entries' => $this->apiService->getEntriesToday(),
            'exits' => $this->apiService->getExitsToday(),
        ]);
    }
```

17 https://fr.wikipedia.org/wiki/Representational_state_transfer

Exemple d'appel pour envoyer des données (méthode POST)

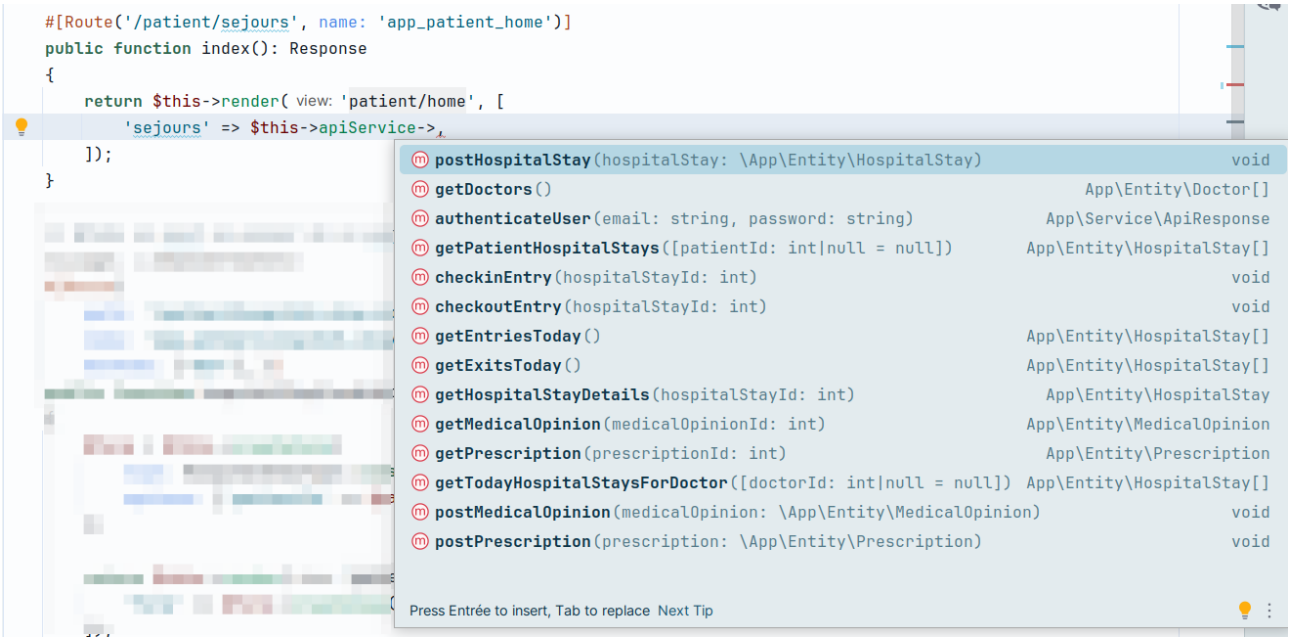
```php
// src/Controller/DoctorPatientsTodayController.php
#[Route(
    path: '/doctor/patients/today/prescription',
    name: 'app_doctor_patients_today_prescription_submit',
    methods: ['POST']
)]
public function prescriptionFormSubmit(Request $request): Response
{
    $form = $this->createForm(PrescriptionType::class);
    try {
        $form->handleRequest($request);
        /** @var Prescription $prescription */
        $prescription = $form->getData();

        $this->apiService->postPrescription($prescription);

        $this->addFlash(
            'success',
            'Modifications enregistrées.'
        );

        return $this->redirectToRoute('app_doctor_patients_today');
    } catch (InvalidContentFailure $validationException) {
        ...
    }
}
```

On peut voir ici que les appels à l'API sont réduits au minimum.

L'API renvoie des objets du domaine, simples et faciles à utiliser (voir la section Extraits d'un DTO et utilisation de twig, page 24).

Code de l’API

L’API expose des méthodes explicites et simples d’utilisation qui utilisent des DTO pour éviter les ambiguïtés et les paramètres trop nombreux :



Les fonctions exposées publiquement utilisent des fonctions privées qui gèrent concrètement les appels à l’API. On voit dans le code suivant comment cela se produit dans le service :

```php
// src/Service/SoigneMoiApiService.php
/** @return HospitalStay[] */
public function getTodayHospitalStaysForDoctor(?int $doctorId = null): array
{
   $doctorId ??= $this->getUserId();
   return $this->getRequest(
      self::API_DOCTORS_HOSPITAL_STAYS_GET_IRI,
      $doctorId,
      HospitalStay::class.'[]');
}
```

La fonction *getTodayHospitalStaysForDoctor*() demande les visites qu’un médecin doit faire ce jour. Si l’utilisateur courant est un médecin, l’identifiant n’a pas besoin d’être passé à la fonction car celui-ci est dans enregistré dans les données de session de l'utilisateur (composant de sécurité injecté dans le service). Dans cette fonction, l'appel à une fonction privée *getRequest*() permet d’abstraire l’ensemble des requêtes de type GET effectuées par le service. L’URL appelée est définie dans une constante de classe ; si nous devons changer les URLs, nous le ferons en modifiant cette constante, sans toucher au code des méthodes. Le dernier paramètre *HospitalStay[]* permet de définir vers quel type de contenu la réponse doit être sérialisée. J’ai fait la même chose pour les requêtes POST et PATCH. La modification et la revue de code seront alors simples et peu risquées.

Du point de vue sécurité et exposition des données, dans l’API on contrôle que le *token* est bien celui d’un docteur et précisément celui indiqué en paramètre.

La méthode privée *getRequest*() interroge directement l’API avec l’authentification, gère les erreurs et renvoi un objet DTO (Data Transfert Object) de notre domaine.

```php
/**
 * @template T
 *
 * @param class-string<T> $type
 *
 * @return T
 *
 * @throws JsonException
 * @throws ClientExceptionInterface
 * @throws RedirectionExceptionInterface
 * @throws ServerExceptionInterface
 * @throws TransportExceptionInterface
 */
private function getRequest(string $url, int $id, string $type): mixed
{
    $response = $this->httpClient->request(
        'GET',
        $this->apiUrl.sprintf($url, $id),
        [
            'auth_bearer' => $this->getToken(),
        ]);

    $this->abortOnNonOkResponse($response, 'GET', $this->apiUrl.sprintf($url, $id));

    try {
        return $this->serializer->deserialize($response->getContent(), $type, 'json');
    } catch (Exception $exception) {
        throw new UnexpectedApiFailure($exception->getMessage(), $exception->getCode(), $exception);
    }
}
```

Dans le détails, on utilise le *client http* injecté pour faire les requêtes. On ajoute des entêtes à la requête pour indiquer le *token* d'autorisation. Puis on lève éventuellement une exception selon la réponse reçue. Et finalement, on désérialise les contenus pour avoir un objet (ou array d'objets) du type demandé.

La requête possède également une en-tête *Accept* pour indiquer que le client attend une réponse de type JSON. Pour ne pas répéter cette information dans les différents types de requêtes (POST, PATCH), cette information est définie par défaut dans la configuration du client HTTP injecté.

```yaml
# fichier config/packages/framework.yaml
# see https://symfony.com/doc/current/reference/configuration/framework.html
framework:

    http_client:
        default_options:
            headers:
                Accept: 'application/json'
```

La fonction *abortOnNonOkResponse*() gère les exceptions pour distinguer les réponses de l'API : soit ce sont des erreurs attendues (contenu invalide, autorisation manquante, etc.), soit ce sont des erreurs inattendues (erreur réseau, erreur serveur interne, etc.). Ce traitement est encapsulé dans une fonction utilisée pour les différents types de requêtes à l'API (PATCH, etc.).

Extraits d'un *DTO* et utilisation de twig

Les avantages d'un DTO (Data Transfer Object) sont de permettre de représenter explicitement les objets du domaine et d'éviter l'utilisation de données sous forme de tableau (moins consistantes, plus difficiles à utiliser, plus propices aux erreurs). On obtient alors un code plus explicite et clair. Nous utilisons ces DTO à tous les niveaux de notre application : dans le service d'API, les contrôleurs et les templates Twig.

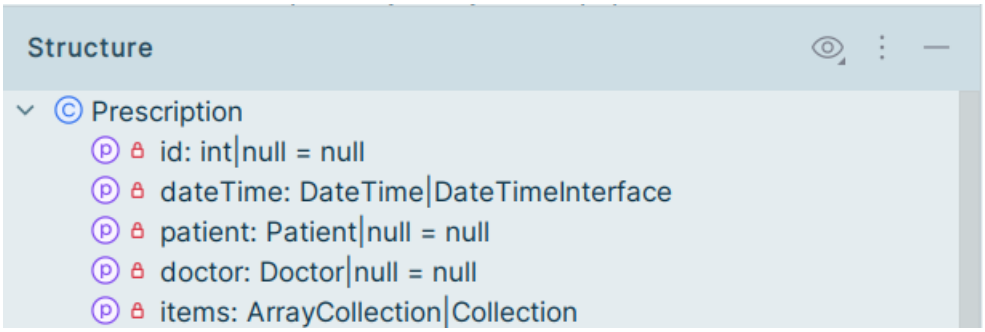
Les DTO nous offrent aussi une façon simple d'assurer la cohérence des objets présents côté serveur (API) et côté client.

Exemples de DTO

```
class Prescription
{
    /**
     * @param PrescriptionItem[] $items
     */
    public function __construct(
        public ?int $id = null,
        public ?Doctor $doctor = null,
        public ?Patient $patient = null,
        public array $items = [],
        public ?DateTime $dateTime = null,
    ) {
    }
}
```

Cet objet est le plus simple possible. L'utilisation des formulaires Symfony oblige à utiliser des valeurs nullables et empêche l'utilisation des modificateurs de propriété readonly. L'objet reste cependant très pratique.

L'entité *Prescription*, coté serveur, reflète l'objet coté client de façon transparente. Comme on peut le voir sur cette capture montrant la structure de l'entité Prescription coté serveur :



Exemple d’utilisation d’un *DTO* dans un *template*

```
{% block body %}
<h1>Dossier du séjour</h1>

  <h2>Séjour</h2>

  <table class="table table-striped table-hover">
    <tr>
      <td>Date d'entrée prévue</td>
      <td>{{ stay.startDate|date('Y-m-d') }}</td>
    </tr>
    <tr>
      <td>Entrée</td>
      <td>{{ stay.checkin|date('Y-m-d H:m') }}</td>
    </tr>
    <tr>
      <td>Motif</td>
      <td>{{ stay.reason }}</td>
    </tr>
    <tr>
      <td>Spécialité</td>
      <td>{{ stay.medicalSpeciality }}</td>
    </tr>
  </table>

  <h3>Préscriptions du séjour</h3>
  {% if stay.prescriptions is not empty %}
    {% for prescription in stay.prescriptions %}
      <h4>Prescription du {{ prescription.dateTime|date('Y-m-d') }}</h4>
      <table class="table table-striped table-hover">
        <thead>
        <tr>
          <th>Medicament</th>
          <th>Posologie</th>
        </tr>
        </thead>
        {% for item in prescription.items %}
          <tr>
            <td>{{ item.drug }}</td>
            <td>{{ item.dosage }}</td>
          </tr>
        {% endfor %}
      </table>
    {% endfor %}
  {% else %}
    <p>Pas de prescription</p>
  {% endif %}
```

Exemple de DTO dans un contrôleur

L'objet de type *MedicalOpinion* dans la variable *$medicalOpinion* est initialisé grâce au retour de l'API (*$this→apiService→getMedicalOpinion($medicalOpinionId))* pour construire le formulaire (`$form = $this->createForm(MedicalOpinionType::class, $medicalOpinion, ['patientId' => $patientId]);`).

```
// src/Controller/DoctorPatientsTodayController.php
#[Route(
    path: '/doctor/patients/today/medical_opinion/{patientId}/{medicalOpinionId?}',
    name: 'app_doctor_patients_today_medical_opinion',
    methods: ['GET']
)]
public function medicalOpinionFormEdit(int $patientId, ?int $medicalOpinionId = null): Response
{
    $medicalOpinion = null;
    if (!is_null($medicalOpinionId)) {
        $medicalOpinion = $this->apiService->getMedicalOpinion($medicalOpinionId);
    }

    $form = $this->createForm(MedicalOpinionType::class, $medicalOpinion, ['patientId' => $patientId]);

    return $this->render('doctor/patients/medical_opinion.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

Pour passer la variable au formulaire, j'ai développé un *FormType* à partir des informations de la documentation officielle[18] et en particulier la documentation sur les composants personnalisés (Custom form types[19]). Utiliser ces composants *form* permet d'obtenir un code objet propre et de réaliser des tests unitaires simplement. J'ai suivi la documentation dédiée[20] à la création de test unitaires pour les *forms* pour le faire. J'obtiens alors un test unitaire comme celui-ci qui remplace un test fonctionnel sur la soumission d'un formulaire dans un navigateur.

18 https://symfony.com/doc/current/forms.html

19 https://symfony.com/doc/current/form/create_custom_field_type.html

20 https://symfony.com/doc/current/form/unit_testing.html

```php
class HospitalStayTypeTest extends TypeTestCase
{
    public function testSubmitValidData(): void
    {
        // Arrange
        $doctorId = 11;

        $hospitalStay = new HospitalStay();
        $hospitalStay->patient = new Patient();
        $hospitalStay->doctor = new Doctor($doctorId);
        $data = [
            'reason' => 'le motif',
            'medicalSpeciality' => 'la spé',
            'patient' => 5,
            'startDate' => '2024-10-05',
            'doctor' => $doctorId
        ];

        $form = $this->factory->create(HospitalStayType::class, $hospitalStay, ['patientId' => 7]);

        $expectedHospitalStay = new HospitalStay(
            startDate: new DateTime('2024-10-05'),
            patient: new Patient(5),
            doctor: new Doctor($doctorId),
            reason: 'le motif',
            medicalSpeciality: 'la spé'
        );

        // Act
        $form->submit($data);

        // Assert
        // check transformation failures
        $this->assertTrue($form->isSynchronized());

        $this->assertEquals($expectedHospitalStay, $hospitalStay);
    }
}
```

Extraits d'une entité Doctrine/Api Platform

Côté API, les entités sont les points clés d'entrée et de sortie. Elles constituent un composant d'accès à la base de données. Ces entités sont utilisées à la fois pour configurer l'ORM (Object Relational Mapping) Doctrine et le composant gérant l'API, ApiPlatform. La configuration d'ApiPlatform est essentiellement définie avec l'attribut de classe *ApiResource*.

```
// fichier src/Entity/HospitalStay.php
#[ORM\Entity(repositoryClass: HospitalStayRepository::class)]
#[ApiResource(
    operations: [
        new GetCollection(
            security: "is_granted('ROLE_DOCTOR') or is_granted('ROLE_PATIENT') or
is_granted('ROLE_ADMIN')",
        ),
        new GetCollection(
            uriTemplate: '/hospital_stays/today_entries',
            security: "is_granted('ROLE_SECRETARY') or is_granted('ROLE_DOCTOR')",
            normalizationContext: ['groups' => 'hospital_stay:read'],
            provider: HospitalStayTodayEntries::class
        ),
        new GetCollection(
            uriTemplate: '/hospital_stays/today_exits',
            security: "is_granted('ROLE_SECRETARY')",
            normalizationContext: ['groups' => 'hospital_stay:read'],
            provider: HospitalStayTodayExits::class
        ),
        new GetCollection(
            uriTemplate: '/doctors/{doctor_id}/hospital_stays/today',
            uriVariables: [
                'doctor_id' => new Link(fromClass: Doctor::class),
            ],
            security: "is_granted('ROLE_DOCTOR')",
            normalizationContext: ['groups' => 'hospital_stay:read'],
            provider: HospitalStayDoctorToday::class
        ),
        new GetCollection(
            uriTemplate: '/patients/hospital_stays',
            normalizationContext: ['groups' => 'hospital_stay:read'],
            security: "is_granted('ROLE_PATIENT')",
            provider: PatientHospitalStaysProvider::class
        ),
        new Get(
            security: "is_granted('ROLE_SECRETARY')",
            normalizationContext: ['groups' => 'hospital_stay:details'],
        ),
        new Post(
            security: "is_granted('ROLE_PATIENT')",
        ),
        new Patch(
            security: "is_granted('ROLE_SECRETARY') or is_granted('ROLE_ADMIN')",
        ),
    ],
    security: "is_granted('')",
    //    paginationItemsPerPage: 5,
)]
class HospitalStay
{
```

On définit ici les différentes opérations (ou endpoints) de l'API, une par entrée de l'array *operation*. Le paramètre *security* permet de restreindre l'accès en fonction des rôles de l'utilisateur identifié. La première opération, de type *Collection*, permet de récupérer l'ensemble des séjours (*HospitalStay*). Elle n'a pas d'*uriTemplate* défini, donc le chemin */doctors/* sera utilisé. Pour les

opérations suivantes de type *Collection*, il faut définir des chemins (IRI), comme */hospital_stays/today_entries* pour la seconde.

Concernant normalizationContext, ce paramètre permet de spécifier, via le champ *groups*, quelles seront les données renvoyées par l'API. On peut ainsi préciser quelles données nous souhaitons renvoyer et également les données des ressources référencées (par exemple, dans cette entité Séjour, on peut choisir de renvoyer le nom du patient et pas simplement son identifiant (IRI)). En interne, ce sont les groupes de sérialisation qui sont utilisés[21].

Un autre paramètre intéressant dans cet extrait est le *provider*[22]. Ce *provider* remplace celui nativement utilisé avec *Doctrine*. On peut ainsi renvoyer un ensemble de données issues d'une requête *Doctrine* écrite par mes soins (par exemple, limiter les séjours à ceux débutant ce jour). Le *provider* est une classe toute simple.

```
// fichier src/ApiResource/StateProvider/HospitalStayTodayEntries.php
readonly class HospitalStayTodayEntries implements ProviderInterface
{
    public function __construct(private HospitalStayRepository $hospitalStayRepository)
    {
    }

    /**
     * @return HospitalStay[]
     */
    #[Override]
    public function provide(Operation $operation, array $uriVariables = [], array
$context = []): array
    {
        return $this->hospitalStayRepository->findBy(['startDate' => new DateTime()]);
    }
}
```

Dans ce provider, il n'y a pas d'enjeux de sécurité, aucun paramètre externe n'est passé. Par contre, dans cet autre provider, il y a un enjeu de sécurité car on utilise un paramètre en provenance de la requête.

```
readonly class HospitalStayDoctorToday implements ProviderInterface
{
    public function __construct(
          private HospitalStayRepository $hospitalStayRepository)
    {}

    /**
     * @return HospitalStay[]
     */
    #[Override]
    public function provide(
          Operation $operation,
           array $uriVariables = [], array $context = []): array
    {
        if (!isset($uriVariables['doctor_id'])) {
            throw new RuntimeException('la variable doctor_id devrait être définie.');
        }

        assert(is_int($uriVariables['doctor_id']));

        return
$this->hospitalStayRepository->findByDoctorForToday($uriVariables['doctor_id']);
    }
}
```

21 https://symfony.com/doc/current/serializer.html#using-serialization-groups-attributes

22 https://api-platform.com/docs/core/state-providers/

On s'assure dans le *provider* de bien recevoir un paramètre de type *integer*. D'autre part, il faut sécuriser la variable le plus proche possible de son utilisation par l'ORM, on a donc le code suivant pour la méthode du *repository :*

```
class HospitalStayRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $managerRegistry)
    {
        parent::__construct($managerRegistry, HospitalStay::class);
    }

    /**
     * @return array<HospitalStay>
     */
    public function findByDoctorForToday(int $doctor_id): array
    {
        return $this->createQueryBuilder('h')
            ->where('h.doctor = :doctor_id')
            ->andWhere('h.checkin IS NOT NULL')
            ->andWhere('h.checkout IS NULL')
            ->setParameter('doctor_id', $doctor_id)
            ->getQuery()
            ->getResult();
    }
}
```

On utilise la méthode *setParameter*() de *Doctrine* qui prémunit des injections SQL et XSS. Concernant les injections SQL, c'est indiqué clairement dans la documentation[23]. Concernant les injections XSS, *Doctrine* a un mécanisme d'inférence du type de variable qui fait que cette variable ne peut accepter que des paramètres de type entier (car le champ de la base de données et la propriété de l'entité sont de type entier), comme l'indique également la documentation[24].
D'autre part, le code utilise un typage strict (`declare(strict_types=1);`) et le type du paramètre est déclaré dans la signature de la fonction. Donc, même en cas de vulnérabilité dans *Doctrine*, le code PHP ne s'exécuterait tout simplement pas. À noter également que PhpStan est configuré pour nous obliger à typer cette variable.

Extraits des fichiers Docker

J'ai entièrement réalisé le développement, les tâches d'intégration et de déploiement continu, ainsi que la mise en production avec *Docker*. J'ai orchestré les conteneurs avec *Docker Compose*. Il y a un fichier *Docker Compose* pour la production et un autre pour le développement, pour des raisons pratiques, notamment en ce qui concerne les certificats SSL.

23 https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/security.html#user-input-and-doctrine-orm

24 https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/query-builder.html#binding-parameters-to-your-query

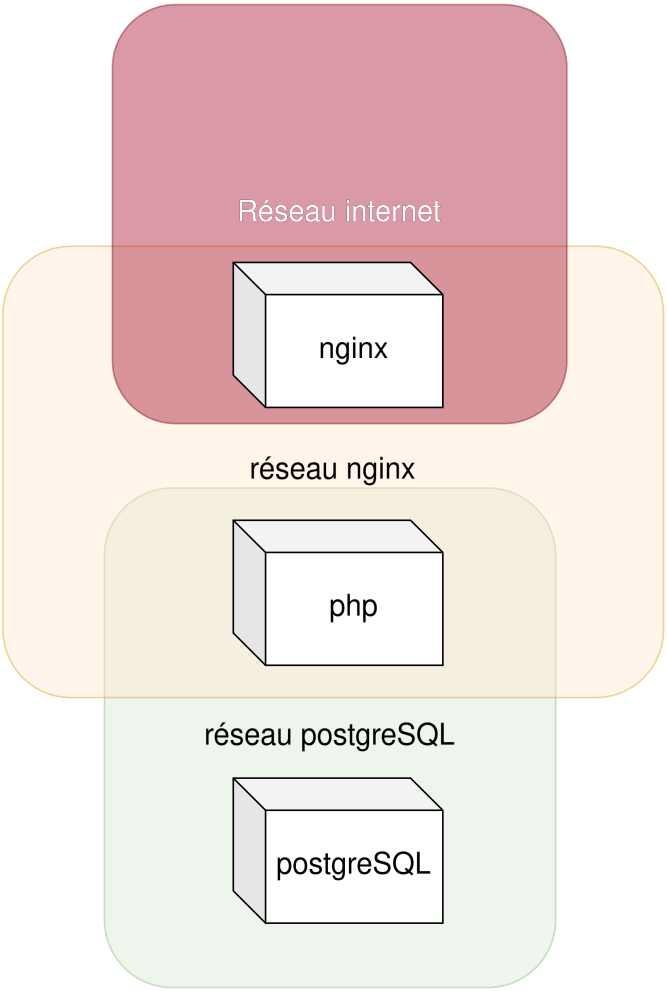
Organisation des réseaux

Dans ces fichiers *compose-prod.yaml* et *compose-dev.yaml*, l'organisation des réseaux est faite en couches pour isoler les composants des applications (api et client web).

```yaml
services:
  nginx:
    expose:
      - "90"
    image: nginx:1.26.0
    networks:
      nginx: null
    ports:
      - target: 90
        published: "${NGINX_PORT}"
  php:
    image: ghcr.io/sebsept/docker-php-symfony-starter:dev
    networks:
      - nginx
      - partage
networks:
  nginx:
  partage:
    external: true
```

Seul le service *nginx*, le serveur web, a des ports ouverts vers l'extérieur (web).

Les autres conteneurs ne sont pas ouverts à l'extérieur, ce qui constitue une mesure de sécurité. Ces conteneurs communiquent cependant entre eux. Pour éviter que tous les services ne communiquent entre eux et ainsi respecter la répartition en couches, j'ai créé différents réseaux, dont la répartition est illustrée ci-dessous :



On voit bien que seul le réseau conteneur (et réseau) est exposé à Internet, que nginx peut communiquer uniquement avec le conteneur php. Le conteneur php peut également communiquer avec PostgreSQL. Et le conteneur PostgreSQL ne communique qu'avec le conteneur php.

Organisation des volumes

Les volumes servent trois objectifs : persister des données, partager des données entre les conteneurs, configurer les applications des conteneurs.

Les conteneurs Docker sont conçus pour être volatiles, ils doivent pouvoir être détruits et reconstruits en toute transparence, afin d'être déployés le plus simplement possible. Il est donc nécessaire d'organiser la persistance en dehors des conteneurs. Pour cette persistance, on utilise des volumes de type *volume* qui persistent indépendamment des conteneurs. Il y en a un pour les données de la base de données (*postgres_data_prod2*), trois pour les certificats SSL et le conteneur qui les génère (*certbot-www, certbot-etc, nginx_data_ssl, certbot-var*). Il y a des volumes de type bind qui permettent aux conteneurs PHP (*./ :app/*) et Nginx (*./public/ :app/public*) d'accéder aux assets et au code source qui sont versionnés. Le dernier volume permet de configurer le serveur Nginx sans avoir à construire une image à chaque modification, tout en versionnant clairement la configuration (./docker/nginx_prod.conf:/etc/nginx/conf.d/default.conf).

Env file

Le serveur de base de données utilisé par *Symfony* est créé par *Docker*. Un utilisateur de la base de données est également créé par *Docker*. Cet utilisateur, le nom de la base de données et le mot de passe doivent être connus de *Symfony*. Nous souhaitons éviter de dupliquer ces données pour prévenir des problèmes de cohérence et de maintenance qui pourraient mettre notre application hors service.

Docker Compose peut utiliser des informations présentes dans un fichier texte, un fichier d'environnement. Symfony, via le composant *symfony/dotenv*, utilise la même fonctionnalité. J'ai donc choisi d'utiliser ce mécanisme pour synchroniser la configuration de la base de données.

Pour des raisons de sécurité, il ne faut pas versionner le fichier qui contient ces données. On voit cette interdiction dans le fichier *.gitignore* :

```
# fichier .gitignore
###> symfony/framework-bundle ###
/.env.local
/.env.local.php
/.env.*.local
```

On doit donc créer ce fichier (*.env.local* en l'occurrence) en production avant le premier déploiement (le déploiement échoue si le fichier n'est pas présent, on ne peut donc pas oublier de le mettre en place).

```
POSTGRES_USER="postgres_prod_user"
POSTGRES_PASSWORD="xxx"
POSTGRES_DB="soignemoi_prod"
# uri de la base de donnée composées des variables précédentes
DATABASE_URL="postgresql://${POSTGRES_USER}:${POSTGRES_PASSWORD}@postgres:5432/${POSTGRES_DB}?
serverVersion=16&charset=utf8"
```

Ce fichier de configuration est référencé dans le fichier *compose-prod.yaml* qui utilise les variables *POSTGRES_** pour créer la base de données.

```
services:
 postgres:
  image: postgres:16.3-alpine3.20
  env_file:
   - .env.local
```

Concernant la sécurité des images Docker, Docker Scout (service d'analyse de sécurité des images Docker) est intégré dans la chaine d'intégration continue.

Extraits d’utilisation des outils de qualité

Ces outils sont utilisés en local, pendant le développement, pour guider et corriger l’implémentation. Ils sont également employés dans la chaîne d’intégration continue pour garantir la qualité du code livré.

Php-cs-fixer

PHP CS Fixer est un outil qui vérifie et formate le code pour qu'il corresponde à des règles standards. J’ai utilisé les standards proposés par Symfony et ajouté des règles spécifiques, telles que l’obligation du typage strict et l’ajout d'une en-tête dans les fichiers PHP.

Cet outil garanti un formatage cohérent et stable, notamment pour s'assurer que seules les lignes modifiées apparaissent dans les analyses des différences lors de revue de code. Il contribue également à la qualité du code, par exemple en imposant l'utilisation de fonctions et non de leurs alias (règle *no_alias_functions*). La configuration s’effectue en PHP, pour le client web, j’ai ce fichier :

```php
// fichier .php-cs-fixer.php
$header = <<<'EOF'
   SoigneMoi Webcli - Projet ECF

   @author Sébastien Monterisi <sebastienmonterisi@gmail.com>
   2024
   EOF;

$finder = (new Finder())
   ->ignoreDotFiles(true)
   ->ignoreVCSIgnored(true)
   ->exclude(['public', 'assets', 'tests', 'config', 'src/Factory/', 'src/DataFixtures/',])
   ->in(__DIR__);

return (new Config())
   ->setRiskyAllowed(true)
   ->setParallelConfig(ParallelConfigFactory::detect())
   ->setRules([
      '@Symfony' => true,
      'global_namespace_import' => ['import_classes' => true, 'import_constants' => true,
'import_functions' => true],
      'declare_strict_types' => true,
      'header_comment' => ['header' => $header],
      'no_alias_functions' => true,
      'no_unneeded_final_method' => true,
      'non_printable_character' => ['use_escape_sequences_in_strings' => false],
      'no_break_comment' => true,
   ])
   ->setFinder($finder);
```

Rector

Rector[25] s'occupe de recommander et de modifier le code pour s'assurer qu'il utilise les constructions et les fonctions les plus récentes de notre version de PHP, entre autres. Cet outil aide à limiter les bugs inattendus, améliore les performances et standardise le code. Rector complète *Php-cs-fixer* en apportant des améliorations spécifiques au code.

Dans l'extrait de code, on peut voir que nous nous conformons aux standards de Symfony et de Doctrine, que nous exigeons le typage pour tout ce qui peut l'être, que nous supprimons le code mort, qu'on utilise la syntaxe et les constructions tel que le permet PHP 8.3, etc.

```php
// fichier rector.php
return RectorConfig::configure()
    ->withPaths([
        __DIR__.'/config',
        __DIR__.'/public',
        __DIR__.'/src',
        __DIR__.'/tests',
        __DIR__.'/rector.php',
    ])
    ->withSkipPath(__DIR__.'/config/bundles.php')
    ->withSkip([__DIR__.'/src/DataFixtures', __DIR__.'/src/Factory'])
    ->withImportNames(removeUnusedImports: true)
    ->withPhpSets(php83: true)
    ->withSets([
        DoctrineSetList::DOCTRINE_CODE_QUALITY,
        SymfonySetList::SYMFONY_CODE_QUALITY,
        PHPUnitSetList::PHPUNIT_CODE_QUALITY,
    ])
    ->withRules([])
    ->withPreparedSets(
        deadCode: true,
        codeQuality: true,
        codingStyle: true,
        typeDeclarations: true,
        privatization: true,
        instanceOf: true,
        earlyReturn: true,
        strictBooleans: true
    )
    ->withSkip([AddMethodCallBasedStrictParamTypeRector::class])
    ->withParallel()
    ->withPHPStanConfigs([__DIR__.'/phpstan.neon'])
;
```

25 https://getrector.com/about

PhpStan

PhpStan[26] est un analyseur de code statique qui inspecte le code sans l'exécuter pour détecter les problèmes, les problèmes potentiels, et mettre en place des bonnes pratiques pour la qualité du code. Il aide à éliminer les ambiguïtés et à assurer une utilisation cohérente du code par les développeurs. PhpStan propose plusieurs niveaux d'analyse, allant du moins strict (1) au plus strict (9). J'ai utilisé le niveau 9, qui permet d'être le plus strict possible. Avec ces règles, on peut prévenir de nombreux bugs, même sans écrire de tests.

```
; fichier phpstan.neon
parameters:
    level: 9
    paths:
        - src
includes:
    - phpstan-baseline.neon
```

26 https://phpstan.org/

Utilisation des outils pendant le développement et l'intégration continue

Ces outils sont actionnés par des scripts Composer. On veille à les lancer régulièrement pendant le développement et l'intégration continue. En local, comme pour l'intégration continue, j'utilise exactement la même commande. Par exemple, la commande *composer ci* lance les scripts *lint-passive*, *security*, etc qui vérifient le formatage du code, des templates, fichiers de composer, etc.
Le script lint-active est assez similaire, mais il effectue des changements de mise en forme et des réécritures via *Rector*, entre autres.Le script *security* est un ensemble de commandes destiné à vérifier que les packages Composer n'ont pas de vulnérabilités connues. Ce script vérifie également les dépendances JavaScript incluses par le composant *AssetMapper*. Il est lancé à la fois dans le script *ci* et dans le script *pre-commit*, ce qui permet d'assurer la détection des failles de sécurité tant lors du développement que lors de la mise en ligne.

```
// fichier composer.json
{ ...
"scripts": {
  "ci": [
    "@lint-passive",
    "@security",
    "@phpstan",
    "@tests"
  ],
  "lint-active": [
    "twig-cs-fixer lint --fix templates/",
    "composer validate --strict",
    "composer normalize",
    "php-cs-fixer fix",
    "@rector"
  ],
  "lint-passive": [
    "twig-cs-fixer lint templates/",
    "composer validate --strict",
    "composer normalize --dry-run --diff",
    "php-cs-fixer check --show-progress none --diff",
    "@rector --dry-run --no-progress-bar"
  ],
  "phpstan": "phpstan analyse",
  "pre-commit": [
    "@lint-active",
    "@security",
    "@phpstan",
    "@tests"
  ],
  "rector": "php vendor/bin/rector",
  "security": [
    "composer audit",
    "./bin/console importmap:audit"
  ],
  "tests": "php vendor/bin/paratest --runner WrapperRunner"
}
```

Déploiement et intégration continue

Pour éviter les taches rébarbatives, s'assurer continuellement de la qualité de l'application et rendre la mise en production la plus simple possible, j'ai mis en place un process d'intégration continue et de déploiement automatisé.
Cette démarche *devops* à été prévue dès la phase de pré-démarrage et a orienter toute la démarche de mise en place des environnements.
J'ai basé la démarche sur les *workflows*[27] et *actions* GitHub. Les actions de vérification du code et de sécurité s'exécutent dès que du code est poussé, le déploiement quand du code est poussé dans la branche *main*.

Le job *integration* est référencé et non défini dans ce workflow. On a donc une *factorisation* qui permet de ne maintenir qu'un seul job de vérification du code exécuté en intégration continue, comme lors du déploiement. Ce job lance les tests, les vérifications de code, la vérification des vulnérabilités des composants *Composer* et des composants de *l'AssetMapper*[28] (dépendances JavaScript).

Le déploiement (job *deploy*) est réalisé si le job d'intégration (integration) se termine avec succès (exit code à 0). Il est effectué via une action (*appleboy/ssh-action*[29]) qui permet de lancer une suite de commandes sur la machine de production par SSH.
Concrètement, le déploiement se fait en exécutant une série de commandes SSH sur la machine de production. Ces commandes effectuent une mise à jour des images *Docker* et du code, puis relancent les conteneurs Docker. Les actions suivantes sont réalisées : création éventuelle du dossier qui va contenir le code (et les assets).

- récupération du code source de *GitHub*
- récupération des nouvelles images *Docker*
- destruction et reconstruction des contenaires
- installation des dépendances *Composer*
- lancement des migrations *Doctrine*
- mise en place des droits d'écriture pour php (création des caches)

27 https://docs.github.com/en/actions/using-workflows/about-workflows#about-workflows

28 https://symfony.com/doc/current/frontend/asset_mapper.html

29 https://github.com/marketplace/actions/ssh-remote-commands

```
// fichier .github/workflows/deploy.yaml
name: Deployment

on:
  push:
    branches:
      - main

jobs:
  integration:
    uses: ./.github/workflows/integration.yaml
    secrets: inherit
  deploy:
    needs: integration
    runs-on: ubuntu-latest

    steps:
      - name: Deployment
        uses: appleboy/ssh-action@f9163462563f649b27272d32e585525a5fe68d76
        with:
          host: api.ecf.seb7.fr
          username: root
          key: ${{ secrets.SERVER_SSH_KEY }}
          script: |
            set -e
            if [ ! -d "/app" ]; then
              git clone --depth=1 --branch main git@github.com:SebSept/soignemoi-api.git /app
            fi
            cd /app
            git fetch origin main --depth=1
            git reset --hard origin/main --
            docker compose -f compose-prod.yaml pull
            docker compose -f compose-prod.yaml down
            docker compose -f compose-prod.yaml up -d --build
            docker compose -f compose-prod.yaml exec -u root php composer install --no-dev --working-dir=/app
            docker compose -f compose-prod.yaml exec php ./bin/console doctrine:migrations:migrate --no-interaction --allow-no-migration
            docker compose -f compose-prod.yaml exec -u root php chown www-data:www-data /app/var /app/public

      - name: Check API is running
        run: curl -s --fail https://api.ecf.seb7.fr
```

L'action *ssh-action*, effectue une connexion sur la machine de production : c'est un point critique de sécurité. C'est pourquoi, je me suis référé à la documentation[30] pour pouvoir utiliser une clé ssh (à garder secrète) dans un fichier de workflow publique et partagé.

Pour revenir à ce qui concerne l'intégration, on a un fichier de workflow très simple, intégrant deux autres workflows : l'un destiné à l'analyse des images Docker, et l'autre aux tests sur le code.

30 https://docs.github.com/en/actions/security-guides/using-secrets-in-github-actions#using-secrets-in-a-workflow

```yaml
# fichier .github/workflows/integration.yaml
name: integration continue

on:
  workflow_call:
  push:
    branches:
      - main
      - dev
  pull_request:
    branches:
      - main
  # chaque jour du lundi au vendredi à 8h
  schedule:
    - cron: '0 8 * * 1-5'

jobs:
  docker-scout:
    uses: ./.github/workflows/_docker_scout.yaml
    permissions:
      contents: read
      pull-requests: write
    secrets: inherit
  composer:
    permissions:
      contents: read
      pull-requests: write
    uses: ./.github/workflows/_tests.yaml
```

Le workflow *_tests.yaml* lance simplement une commande PHP exécutant un script composer, le même que je lance sur mes machines de développement. De cette façon, je peux anticiper le résultat des *workflows* sur GitHub.

```yaml
# fichier .github/workflows/_tests.yaml
name: Composer check & CI

on:
  workflow_call:

jobs:
  quality:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: docker compose
        run: |
          set -e
          docker network create partage
          docker compose --progress quiet -f compose-dev.yaml up -d --build
          docker compose -f compose-dev.yaml exec php composer install --quiet

      - name: check composer vulnerabilities
        run: docker compose -f compose-dev.yaml exec -it php /app/check_composer.sh

      - name: run composer ci script
        run: docker compose -f compose-dev.yaml exec php composer run-script ci
```

Construction d’une application de type Desktop

Construire une application bureautique représente un travail important, cumulant la mise en place d'un environnement, le développement, les tests et le déploiement. Lorsque un client web est déjà disponible, il semble plus pertinent de créer une application minimale qui se contente d'une vue web. C'est pourquoi j'ai réalisé une application avec ElectronJS[31], qui offre cette possibilité. On obtient ainsi une application distribuable sous forme d'exécutable indépendant, comme demandé par nos spécifications, tout en bénéficiant des avantages d'un client web (mises à jour instantanées, code presque entièrement déporté dans le client web, tests et environnement web déjà implémentés, etc.).

Comme il s'agit spécifiquement d'une application desktop et que le client web est implémenté avec une approche *mobile-first*, j'ai testé le rendu et l'utilisation des pages sur un écran large et moyen. Le code produit se résume finalement à cet extrait :

```
const { app, BrowserWindow } = require('electron')

const createWindow = () => {
 const win = new BrowserWindow({
 width: 800,
 height: 600,
 webPreferences: {
 nodeIntegration: true,
 }
 })
 win.setMenu(null)
 win.loadURL('https://cli.ecf.seb7.fr/')
}

app.whenReady().then(() => {
 createWindow()

 app.on('activate', function () {
 if (BrowserWindow.getAllWindows().length === 0) createWindow()
 })
})
```

Le code et le fichier distribuable sont disponibles sur GitHub : SebSept/soignemoi-desktopcli/[32]

31 https://www.electronjs.org/

32 https://github.com/SebSept/soignemoi-desktopcli/

Construction d’une application mobile

Pour les même raisons que l’application bureautique (économie de code, fiabilité et maintenance), l’application mobile est une simple *vue web* vers le client web.
Pour partir sur une base standard, j’ai choisi le framework Flutter[33], le code que j’ai écrit se limite au bloc suivant :

```
import 'package:flutter/material.dart';
import 'package:webview_flutter/webview_flutter.dart';

void main() {
  runApp(
    const MaterialApp(
      home: WebViewApp(),
      debugShowCheckedModeBanner: false,
    ),
  );
}

class WebViewApp extends StatefulWidget {
  const WebViewApp({super.key});

  @override
  State<WebViewApp> createState() => _WebViewAppState();
}

class _WebViewAppState extends State<WebViewApp> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: WebViewWidget(
        controller: _MyWebViewController(),
      ),
    );
  }
}

class _MyWebViewController extends WebViewController {
  _MyWebViewController() {
    super.setJavaScriptMode(JavaScriptMode.unrestricted);
    super.setBackgroundColor(const Color(0x00000000));
    super.loadRequest(Uri.parse('https://cli.ecf.seb7.fr/'));
  }
}
```

J’ai testé le rendu simplement dans mon navigateur web pendant le développement et j’ai finalement vérifié le rendu dans l’application elle même en installant le fichier *apk* produit.

Le code et le fichier distribuable sont disponible sur GitHub : SebSept/soignemoi-mobilecli[34]

Comme pour le client bureautique, la création des exécutables est possible de plusieurs façons, que je ne connais pas de mémoire. Pour garantir l’aboutissement du projet, j’ai documenté ces processus de *release* dans les fichiers *README* des dépôts de code.

33 https://flutter.dev/

34 https://github.com/SebSept/soignemoi-mobilecli

Composant d’accès d’une interface graphique avec accès à une base de donnée

L’API ne présente aucune interface graphique ; les composants graphiques sont uniquement du côté du client web (et indirectement dans les clients mobile et bureautique).

Dans le client web, j’ai utilisé le composant Form de *Symfony*. Côté template, le code est relativement simple, car j’ai créé des composants de formulaires (FormType) qui encapsulent le traitement des données pour l’affichage et la sérialisation. On obtient ainsi du code qui ne nécessitera de modification que dans un seul fichier si l'on devait ajouter des données au formulaire. Le code dans le template est finalement très simple, par exemple :

```
// fichier templates/doctor/patients/prescription.html.twig
{% block body %}
    <div class="row">
        <h2>Prescription</h2>
        {{ form_start(
           form,
           {action: path('app_doctor_patients_today_prescription_submit')})
        }}

        <button type="button" class="btn btn-primary add_item_link">
             Ajouter une prescription
        </button>
        {{ form_rest(form) }}
    </div>
```

Dans le contrôleur, pour afficher le formulaire avec les données on se contente de :

```
#[Route(
    path: '/doctor/patients/today/prescription/{patientId}/{prescriptionId?}',
    name: 'app_doctor_patients_today_prescription',
    methods: ['GET']
)]
public function prescriptionFormEdit(int $patientId, ?int $prescriptionId = null):
Response
{
    if (!is_null($prescriptionId)) {
        $prescription = $this->apiService->getPrescription($prescriptionId);
    } else {
        $prescription = new Prescription(null, new Doctor(), new Patient($patientId),
[]);
    }

    $form = $this->createForm(PrescriptionType::class, $prescription);

    return $this->render('doctor/patients/prescription.html.twig', [
        'form' => $form->createView(),
    ]);
}
```

Et pour traiter les données du formulaire, on a simplement :

```php
#[Route(
    path: '/doctor/patients/today/prescription',
    name: 'app_doctor_patients_today_prescription_submit',
    methods: ['POST']
)]
public function prescriptionFormSubmit(Request $request): Response
{
    try {
        $form = $this->createForm(PrescriptionType::class);
        $form->handleRequest($request);
        /** @var Prescription $prescription */
        $prescription = $form->getData();

        $this->apiService->postPrescription($prescription);

        $this->addFlash(
            'success',
            'Modifications enregistrées.'
        );

        return $this->redirectToRoute('app_doctor_patients_today');
    } catch (InvalidContentFailure $validationException) {
        $this->addFlash(
            'danger',
            $validationException->getMessage()
        );

        return $this->render('doctor/patients/prescription.html.twig', [
            'form' => $form->createView(),
        ]);
    } catch (Exception $e) {
        $this->addFlash(
            'danger',
            'Erreur interne.'
        );

        return $this->redirectToRoute('app_doctor_patients_today');
    }
}
```

L’accès à la base de données se fait par les appels à l’API qui fait la validation des données. La validation des données coté client est réduite au minimum pour laisser la validation du coté de l’API. On évite une double validation et les risques d’incohérence de cette façon.

Extrait d'une entité *Doctrine*

Grâce à l'utilisation de Doctrine, je n'ai pas eu à écrire directement les schémas SQL. Par contre, j'ai pu vérifier le schéma généré à partir des entités que j'ai codées.
J'ai défini ces entités en me basant sur le diagramme MCD (annexe p 133), lui-même produit à partir des classes de conception.

J'ai utilisé la documentation sur les colonnes[35] *Doctrine* et la documentation sur les contraintes de validation[36] *Symfony* pour écrire les définitions des colonnes.

Voici un exemple (tronqué) qui reprend quelques points clés :

```php
#[ORM\Entity(repositoryClass: HospitalStayRepository::class)]
class HospitalStay
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(type: Types::DATE_MUTABLE)]
    private ?DateTimeInterface $startDate = null;

    #[ORM\Column(type: Types::DATETIME_MUTABLE, nullable: true)]
    private ?DateTimeInterface $checkin = null;

    #[ORM\Column(length: 255)]
    #[Assert\NotBlank]
    private ?string $reason = null;

    #[ORM\ManyToOne(inversedBy: 'hospitalStays')]
    #[ORM\JoinColumn(nullable: false)]
    #[Groups(['hospital_stay:read', 'hospital_stay:details'])]
    private ?Patient $patient = null;

    #[ORM\ManyToOne(inversedBy: 'hospitalStays')]
    #[ORM\JoinColumn(nullable: false)]
    #[Groups(['hospital_stay:read', 'hospital_stay:details'])]
    private ?Doctor $doctor = null;
```

On peut voir que la classe est définie comme une entité Doctrine (`ORM\Entity`).
La propriété id est utilisée comme clé primaire[37].
L'attribut *GeneratedValue* est laissé par défaut pour ne pas compromettre la portabilité vers un SGBD différent. On utilise *PostgreSQL*, on a donc une stratégie du type *Sequence*, comme indiqué dans la documentation[38]. En utilisant un autre SGBD, on aura une valeur effective différente. Si on avait explicitement défini la stratégie à Sequence, on aurait alors eu un problème avec les SGBD qui ne possèdent pas cette fonctionnalité d'indexage.
Pour la propriété *startDate*, j'ai défini le type à *DATE_MUTABLE*, *Doctrine* définira le type concret lui même en fonction de notre SGBD.
Pour la propriété *checkin*, j'ai défini le type à *DATETIME_MUTABLE*, c'est-à-dire que je stockerai la date et l'heure (contrairement à un champ *DATE_MUTABLE*).
Pour le champ *reason*, je définis sa longueur maximale en base de données, son type est inféré à

35 https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/attributes-reference.html#column

36 https://symfony.com/doc/current/reference/constraints.html

37 https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/attributes-reference.html#id

38 https://www.doctrine-project.org/projects/doctrine-orm/en/3.2/reference/basic-mapping.html#identifier-generation-strategies

partir du type PHP. J’ajoute la contrainte *NotBlank* à notre validation, mais cela ne concerne pas la base de données.
Pour la propriété *patient*, j’utilise une autre entité et établis donc une relation avec une autre table de la base de données. Il s’agit de la table définie pour l’entité *Patient*, car le type utilisé est *Patient*. Dans cette entité *Patient*, le champ qui *inverse* la relation est *hospitalStays*. La relation est de type *ManyToOne*, ce qui signifie qu’à un *Patient* peuvent être associés plusieurs *Séjours*. Je définis également que la propriété doit toujours être définie. En conséquence de cette relation, Doctrine ajoutera une colonne *patient_id* dans la table *hospital_stay.* Le modèle est alors un modèle normalisé.

Présentation du jeu d’essai

Techniquement, pour la mise en place des tests, comme pour l’ensemble de l’application, je reste le plus proche possible des standards d’usage. J’ai donc utilisé phpunit[39].

Jeu de test du client web

Concernant ce client web, les tests se centrent sur le service d’API. C’est le point le plus important de l’application cliente et c’est un composant entièrement élaboré par mes soins. Il faut couvrir l’envoi de requêtes (l’authentification et les autorisations) et la réception des réponses aux requêtes. Il faut également s’assurer que les pages s’affichent correctement. Pour cet aspect, il s’agit de tests de non-régression.

Ces tests font intervenir plusieurs composants ; réaliser des tests unitaires (simples) n’est pas possible. Réaliser des tests fonctionnels (end-to-end) n’est pas non plus souhaitable car cela nécessite une API fonctionnelle dont on contrôle les données, cette mise en place n’est pas pratique, pas rapide, peut être compliquée et sujette à des défaillances. J’ai donc implémenté des tests d’intégration (qui sollicitent plusieurs composants conjointement). Ces tests nécessitent de contrôler les composants liés au service testé, notamment le client HTTP qui doit faire les requêtes à l’API. Plusieurs classes de tests sont disponibles dans l’intégration de *phpunit* à *Symfony*, j’ai utilisé la classe *KernelTestCase*, qui permet de modifier le contenaire Symfony (*stubbing*) et de profiter de l’injection de dépendances (*autowiring*). J’ai utilisé la documentation officielle pour m’aider[40].

J’ai finalement les tests suivants :

- tests de non régression
- tests d’authentification et d’autorisation
- tests d’envoi de données
- tests de réception des réponses

39 https://phpunit.de

40 https://symfony.com/doc/current/http_client.html#testing

Tests de non régression

Ces tests sont relativement basiques. Dans une approche TDD[41], j'ai écrit ces tests dès le début du développement. Ils m'ont servi à m'assurer de ne pas casser l'application pendant le développement, sans pour autant tester continuellement l'application manuellement dans un navigateur.
Ces tests m'ont également servi pour m'assurer que le système d'autorisation était en place et fonctionnel.

```
// tests/e2e/HomePageTest.php
class HomePageTest extends WebTestCase
{
    use HasBrowser;

    public function testViewHomePage(): void
    {
        $this->browser()
            ->visit('/')
            ->assertSuccessful();
    }

    public function testRedirectedToLoginPageIfNotAuthenticated(): void
    {
        $this->browser()->interceptRedirects()
            ->visit('/patient/sejours')
            ->assertNotAuthenticated()
            ->assertRedirectedTo('/login');
    }

    public function testViewLoginFormIfNotLoggedIn(): void
    {
        $this->browser()->visit('/login')
            ->assertSuccessful()
            ->assertSee('Connexion')
            ->assertSeeElement('form');
    }

}
```

Tests sur l'authentification

L'authentification consiste à envoyer à l'API une paire de chaine de caractères *identifiant/mot de passe*, au format JSON, afin de recevoir un token qui est stocké côté client dans les données de session. Ce token sert à authentifier l'utilisateur dans les requêtes à l'API faites par la suite.

Dans la classe PHP *SoigneMoiApiServiceAuthenticationTest*, j'ai d'abord testé le comportement lors de la réception de réponses non conformes aux attentes pour m'assurer que l'application gère bien ces cas et propose une réponse adéquate à l'utilisateur. J'ai donc testé les cas suivants :

- réponse http d'autorisation refusée (401) (*testAuthenticationFailsIfUnAuthorized)*
- le contenu reçu n'est pas du json (*testAuthenticationFailsIfNoJsonResponse*)

41 Test driven development

- le contenu reçu ne contient pas de token (*testAuthenticationFailsIfNoTokenReceived*)
- le contenu reçu n'est ne contient pas de champ *rôle* (*testAuthenticationFailsIfNoRoleReceived*)
- le contenu reçu a un rôle non reconnu / non autorisé (*testAuthenticationFailsRoleIsNotAllowed*)

On peut remarquer que je n'ai pas testé les réponses 403 (autorisation manquante). En effet, j'ai déterminé la pertinence des tests selon une approche en boîte blanche, en ayant connaissance des réponses possibles, sans traiter les réponses que l'API ne peut pas émettre. L'approche en boîte blanche permet de gagner du temps en n'implémentant que les tests nécessaires. Étant donné que le client et l'API ont été développés (presque) en même temps et que ces parties sont entièrement sous notre contrôle, c'est une méthode adaptée. Cette approche introduit cependant un risque si on change les réponses de l'API et qu'on oublie de modifier le client et les tests en conséquence. Ceci dit, un travail rigoureux ne devrait pas permettre cette incohérence. En particulier, au moment de l'authentification, demander au client d'être déjà authentifié n'a pas de sens et n'est effectivement pas une réponse possible de mon contrôleur côté API. Finalement, cette approche en boîte blanche est un bon compromis entre fiabilité et économie.

Les extraits de code suivants illustrent la structure et les mécanismes de *stubbing*[42] des dépendances mis en place. Dans le code métier, le service d'API, un client HTTP est injecté dans le service lors de la construction du service par le conteneur de Symfony. Ce client HTTP injecté effectue les requêtes concrètes à l'API. Pour le test, il faut créer un *stub* du client et définir la réponse de l'API ; on simule ainsi une réponse déterminée de l'API :

```
public function __construct(
    private readonly Security $security,
    private readonly HttpClientInterface $httpClient,
    private readonly string $apiUrl,
    #[Autowire(service: 'monolog.logger.api_errors')]
    private readonly LoggerInterface $apiErrorsLogger,
) {
...
}
```

Dans le test, il faut injecter le client avec le stubbing adéquat avant de faire l'appel au code testé dans le service.

42 Ici, le stubbing consiste à prédéterminer les réponses d'une méthode de classe.

```php
// tests/Service/SoigneMoiApiServiceAuthenticationTest.php
public function testAuthenticationFailsIfUnAuthorized(): void
{
    // Arrange
    self::bootKernel();

    $mockResponse = new JsonMockResponse(
        ['role' => 'ROLE_ADMIN', 'accessToken' => '123', 'id' => 1],
        ['http_code' => Response::HTTP_UNAUTHORIZED]
    );
    $httpClient = new MockHttpClient();
    $httpClient->setResponseFactory(static fn(): JsonMockResponse => $mockResponse);
    static::getContainer()->set(HttpClientInterface::class, $httpClient);

    // Act
    /** @var SoigneMoiApiService $api */
    $api = static::getContainer()->get(SoigneMoiApiService::class);
    $response = $api->authenticateUser('nomatter@nomatter.com', 'nomatter');

    // Assert
    $this->assertFalse($response->ok);
}
```

Finalement, dans ce test on vérifie que la méthode *authenticateUser*() renvoi un *Response* avec la propriété *ok* valant *false* quand l'API renvoi une réponse avec un code HTTP 401.

Test sur l'envoi de données

Ces tests ont une portée plus limitée, ils permettent de s'assurer que :

- le service n'a pas de défaillance interne grave
- le token d'authentification est bien passé dans la requête
- les objets passés au service pour envoi sont transformées en données *json* exploitables par l'API
- les réponses http attendues sont cohérentes avec la validité des données passées

PhpStan vient compléter ces tests, il s'assure qu'on ne passe jamais autre chose que les objets du domaine attendus (Prescription, Avis médical, etc) grâce au typage fort.

Pour exemple d'envoi de données, voici le test qui vérifie que le token d'authentification est bien envoyé dans les requêtes à destination du serveur.

```php
// tests/Service/SoigneMoiApiServiceAuthenticationTest.php
public function testAuthTokenIsSent(): void
{
    // les tests sont réalisés au moment de la création des requetes,
    // dans les callbacks définis ici
    $testExpectedApiCalls = [
        function ($method, $url, array $options): JsonMockResponse {
            $headers = $options['normalized_headers'];
            // Assert
            $this->assertSame('GET', $method);
            $this->assertContains('Authorization: Bearer 123', $headers['authorization']);

            return new JsonMockResponse(
                ['rien' => 'sans aucune importance, non testé'],
                ['http_code' => Response::HTTP_OK] // important sinon, exception est levée
            );
        }
    ];

    $httpClient = new MockHttpClient($testExpectedApiCalls);

    static::getContainer()->set(HttpClientInterface::class, $httpClient);
    static::getContainer()->set(Security::class, $this->getMockedSecurity());

    // Act
    /** @var SoigneMoiApiService $api */
    $api = static::getContainer()->get(SoigneMoiApiService::class);
    $api->getDoctors();
}
```

On note l'utilisation de la fonction `$this->getMockedSecurity()` qui *stub* le *user* :

```php
protected function getMockedSecurity(): MockObject
{
    $user = new User('nop@nop.com');
    $user->setToken('123');
    $user->setId(7);

    $mockedSecurity = $this->createMock(Security::class);
    $mockedSecurity->method('getUser')->willReturn($user);

    return $mockedSecurity;
}
```

Le test montre bien que le token défini au niveau de l'utilisateur est utilisé pour construire les en-têtes de la requête envoyée à l'API.

Dans le test suivant, je vérifie que les données envoyées lors de la création d'un séjour sont formatées comme attendu.

```php
// tests/Service/PostPatientHospitalStaysTest.php
public function testPostPatientHospitalStay(): void
{
    // Arrange
    $hospitalStay = new HospitalStay(
        id:null,
        startDate: new DateTime('2025-12-07'),
        endDate: new DateTime('2025-12-14'),
        medicalSpeciality: 'la specialite',
        reason: 'une raison valable',
        doctor: new Doctor(4)
    );

    // les tests sont réalisés au moment de la création des requetes,
    // dans les callbacks définis ici
    $testExpectedApiCalls = [
        function ($method, $url, array $options): JsonMockResponse {
            $body = $options['body'];
            // Assert
            // tests basiques
            $this->assertSame('POST', $method);
            $this->assertJson($body);
            // tests sur les contenus
            $this->assertStringContainsString('"startDate":"2025-12-07', $body);
            $this->assertStringContainsString('"endDate":"2025-12-14', $body);
            $this->assertStringContainsString('"doctor":"\/api\/doctors\/4"', $body);
            $this->assertStringContainsString('"patient":"\/api\/patients\/7"', $body);
            $this->assertStringContainsString('"medicalSpeciality":"la specialite"', $body);
            $this->assertStringContainsString('"reason":"une raison valable"', $body);

            return new JsonMockResponse(
                ['rien' => 'sans aucune importance, non testé'],
                ['http_code' => Response::HTTP_OK]
            );
        }
    ];

    $httpClient = new MockHttpClient($testExpectedApiCalls);

    static::getContainer()->set(HttpClientInterface::class, $httpClient);
    static::getContainer()->set(Security::class, $this->getMockedSecurity());

    // Act
    /** @var SoigneMoiApiService $api */
    $api = static::getContainer()->get(SoigneMoiApiService::class);
    $api->postHospitalStay($hospitalStay);
}
```

Les questions liées à la logique métier sont implémentées du côté de l'API. Je m'assure du respect des contraintes métier à ce niveau pour éviter que cette logique puisse être contournée ou oubliée et pour qu'elle reste stable.

J’ai également implémenté des tests fonctionnels, mais ils ne sont pas référencés dans le fichier *phpunit.xml* pour ne pas être lancés automatiquement. En effet, ils sont sujets aux problèmes décrits plus haut car ils reposent sur les réponses de l’API. Ils m’ont tout de même été utiles lors de l’écriture du code.

Le service que j’ai implémenté n’est pas complètement couvert par les tests. Dans une optique de tests en boîte blanche et pour des raisons d’économie, j’ai déterminé que leur implémentation n’était pas utile. Certaines fonctions utilisent en interne les mêmes fonctions privées qui implémentent le cœur de la logique. Les fonctions exposées (publiques) sont assez rudimentaires et ne font qu’appeler les fonctions internes en passant les paramètres adéquats ; les tester n’apporte rien de concret.

Par exemple, la méthode pour obtenir les détails d’un séjour se limite à :

```
public function getHospitalStayDetails(int $hospitalStayId): HospitalStay
{
    return $this->getRequest(self::API_HOSPITAL_STAY_DETAILS, $hospitalStayId, HospitalStay::class);
}
```

De la même façon, la méthode pour réaliser une entrée (*checkin*), de type *PATCH*, est minimale :

```
public function checkinEntry(int $hospitalStayId): void
{
    $this->patchRequest(
        self::API_HOSPITAL_STAYS_PATCH_IRI,
        $hospitalStayId,
        ['checkin' => (new DateTime())->format('c')]
    );
}
```

J’ai fais un test sur une méthode GET, POST et PATCH pour couvrir chaque type de méthode interne. C’est suffisant pour s’assurer du bon fonctionnement du service.

Test sur la récupération de données

Pour la récupération de données, j'utilise le même principe, sauf qu'on doit cette fois traiter concrètement les données reçues. Il faut donc des données réalistes, conformes aux données réellement renvoyées par l'API. Pour des raisons d'organisation du code et parce que la quantité de données est plus conséquente, j'ai séparé les tests et les données ; les données sont stockées dans un fichier à part, *tests/Service/stubs/response_hospital_stays.json*.

```
// tests/Service/CommonTest.php
public function testGetRequestReturnsArrayOfEntities(): void
{
    $this->prepareHttpResponse('response_hospital_stays.json');
    static::getContainer()->set(Security::class, $this->getMockedSecurity());

    /** @var SoigneMoiApiService $api */
    $api = static::getContainer()->get(SoigneMoiApiService::class);

    $hospitalStays = $api->getPatientHospitalStays();

    $this->assertIsArray($hospitalStays);
    $this->assertContainsOnlyInstancesOf(HospitalStay::class, $hospitalStays);
}
```

On peut voir que j'utilise le contenu d'un fichier avec la réponse attendue (au format JSON) et que j'ai dû *stubber* le service de sécurité.

Jeu de test de l'API

Du coté de l'API, il n'y a pas besoin d'un service externe, la création de tests est facilité. Pour ces tests, j'ai procédé de façon plus exhaustive et avec une approche TDD, en implémentant les tests avant les fonctionnalités.

J'ai élaboré un plan de test relativement simple. Ce plan adopte une approche en trois temps : un premier temps pour m'assister dans l'écriture du code et faire les vérifications basiques de non régression; un deuxième temps pour compléter les tests et vérifier la conformité aux exigences du cahier des charges et enfin j'ai implémenté quelques tests uniquement dédiés à la sécurité avec *sqlmap*[43] (outil automatisé de test d'injections SQL et XSS).

J'ai réalisé deux types de tests. D'un côté, des tests fonctionnels, puisque la création de l'API repose essentiellement sur la configuration et l'agencement de composants tiers : Api-Platform, Doctrine, les composants *Security* et *Validation* de *Symfony*. Avec des tests d'intégration, je m'assure que l'API répond comme il se doit, donc que les composants sont bien mis en place conformément aux attentes.

J'ai également réalisé des tests d'intégration (*Doctrine* et interrogation de la base de données) pour les méthodes que j'ai implémentées dans les entités. Une partie de ces tests couvre deux fois des parties du code, mais dans l'ordre de création, ces tests avaient du sens.

43 https://sqlmap.org/

Pour faciliter l'écriture des tests et la mise en place des *fixtures*, j'ai utilisé des packages non fournis par Symfony : *Zenstruck/Foundry*[44], pour créer des fixtures relativement simplement et *zenstruck/browser*[45] pour simuler simplement des requêtes à l'API.

Tests fonctionnels

Test de récupération de données

Le test sur l'URL *hospital_stays/today_entries* permet de récupérer les séjours dont l'entrée se fait aujourd'hui.

```
// tests/e2e/SecretaryTest.php
public function testCountTodayEntries(): void
{
    // Arrange
    PatientFactory::new()->many(10)->create();
    HospitalStayFactory::new()->withExistingPatient()->entryBeforeToday()->many(3)->create();
    HospitalStayFactory::new()->withExistingPatient()->entryToday()->many(5)->create();
    HospitalStayFactory::new()->withExistingPatient()->exitToday()->many(2)->create();
    HospitalStayFactory::new()->withExistingPatient()->entryAfterToday()->many(3)->create();

    // Act
    $secretary = UserFactory::new()->secretary()->create();
    $this->browser()->actingAs($secretary->object())
        ->get('/api/hospital_stays/today_entries', HttpOptions::json())
    // Assert
        ->assertSuccessful()
        ->use(static function (Json $json) : void {
            $json->hasCount(5);
        });
}
```

Dans la partie de mise en place (*arrange*), je vais créer des données pour nos tests. Foundry nous permet d'être très expressifs : on comprend facilement qu'on commence par créer 10 patients, puis 3 séjours affectés à des patients existants, dont la date d'entrée est avant aujourd'hui, etc. Dans la partie *act*, on crée un client HTTP (navigateur simulé) utilisé par un utilisateur authentifié avec un rôle de secrétaire. On fait un appel sur l'URL à tester, on vérifie qu'on a bien une réponse valide (code HTTP de type 200) et avec 5 résultats, soit 5 séjours dont l'entrée doit être faite ce jour.

La base de données est remise à zéro automatiquement grâce au trait *ResetDatabase*.

44 https://symfony.com/bundles/ZenstruckFoundryBundle/current/index.html

45 https://github.com/zenstruck/browser

Test d'envoi de données

Dans ce test, on effectue la démarche inverse : on appelle l'API et vérifie ensuite que la base de données a été impactée comme attendu.

```
// tests/e2e/PatientTest.php
public function testCreatePatientSuccess(): void
{
    $this->browser()
        ->request('POST', '/api/patients', [
            'headers' => [
                'Content-Type' => 'application/json',
                'Accept' => 'application/json',
            ],
            'json' => $this->validPayload()
        ]
    )
        ->assertSuccessful();

    UserFactory::assert()->count(1);
    PatientFactory::assert()->count(1);

    $user = UserFactory::repository()->first();
    $this->assertTrue($user->object()->isPatient());
}
```

Ici, on crée un patient, sans restriction d'accès, il n'y a donc pas besoin de s'identifier. On fait une requête sur l'URL */api/patients* avec une charge en JSON. On vérifie que la requête a bien un retour positif, qu'un nouveau patient a été créé, qu'un nouvel utilisateur système a été généré et qu'il a bien le rôle de patient. On utilise à nouveau les fonctionnalités de *Foundry* pour faire les vérifications sur la base de données.

J'ai également implémenté les cas d'échec attendu (mot de passe trop faible, prénom vide, etc) . Par exemple pour la force du mot de passe, j'ai fais ce test :

```
public function testCreatePatientFailsOnTooWeakPassword(): void
{
    $this->testCreationFails($this->validPayload(['userCreationPassword' => '1234567']));
}
```

Pour tester l'injection XSS sur le second champs d'adresse, on a le test :

```
/**
 * @dataProvider dataProviderXSS
 */
public function testCreatePatientFailsOnAddress2XSSAttack($payload): void
{
    $this->testCreationFails($this->validPayload(['address2' => $payload]));
}
```

Pour ce test, j'utilise un *dataprovider* qui va lancer le test plusieurs fois avec un jeu de données permettant une attaque XSS[46].

46 A noter que la protection contre les attaques XSS est assurée par l'échappement des caractères réalisé par Twig. La présence de données dangereuses dans la base de données et les formulaires n'est pas problématique dans ce contexte.

```
private function dataProviderXSS(): array
{
    return [
        ["&lt;script&gt;alert('XSS')&lt;/script&gt;"],
        ["http://example.com/search?term=%3Cscript%3Ealert('XSS')%3C%2Fscript%3E"],
        ["#<img src=x onerror=alert('XSS')>"],
        ["<script>alert('XSS')</script>"],
    ];
}
```

Les tests précédents reposent sur une fonction utilisée pour toutes les requêtes qui doivent avoir un retour en erreur :

```
private function testCreationFails(array $payload): void
{
    $usersCountsBefore = UserFactory::repository()->count();
    $patientsCountsBefore = PatientFactory::repository()->count();

    $this->browser()
        ->request('POST', '/api/patients', [
            'headers' => [
                'Content-Type' => 'application/json',
                'Accept' => 'application/json',
            ],
            'json' => $payload
        ]
        )
        ->assertStatus(422);

    // aucune entité crée
    UserFactory::assert()->count($usersCountsBefore);
    PatientFactory::assert()->count($patientsCountsBefore);
}
```

J’ai également réalisé un test contre les injections SQL, mais ce sont surtout les tests de sécurité avec Sqlmap qui permettront de tester la protection contre ce type d’attaques.

Test de modification de données interdite

Nos tests vérifient le fonctionnement correct de l'API et également que les restrictions métier sont bien respectées. Par exemple, on peut vérifier qu'un docteur ne peut pas modifier un avis médical écrit par un autre docteur.

```
public function testPatchExistingMedicalOpinionWithAnotherDoctor(): void
{
    // Arrange
    $patient = PatientFactory::new()->create();
    $doctorUser = $this->makeDoctorUser();
    DoctorFactory::repository()->first()->object();
    $otherDoctor = DoctorFactory::new()->create();
    $medicalOpinion = MedicalOpinionFactory::new()->create([
        'patient' => $patient,
        'doctor' => $otherDoctor,
    ]);
    $medicalOpinionId = $medicalOpinion->getId();

    // Act
    $payload = [
        'patient' => '/api/patients/'.$patient->getId(),
        'doctor' => '/api/doctors/'.$otherDoctor->getId(),
    ];

    $this->browser()->actingAs($doctorUser->object())
        ->patch(
            '/api/medical_opinions/'.$medicalOpinionId,
            HttpOptions::create(['headers' =>
                [
                'Content-Type' => 'application/merge-patch+json',
                'Accept' => 'application/json'
                ],
                'json' => $payload])
        )
        // Assert
        ->assertStatus(Response::HTTP_UNPROCESSABLE_ENTITY);
}
```

Dans la partie *arrange*, on crée un avis médical attribué à un premier docteur. Dans la partie *act*, on fait une requête de modification de l'avis médical et, finalement, on fait une assertion sur le code de retour HTTP qui correspond à un rejet de l'API.

J'ai également implémenté des tests pour vérifier la création de prescription.

Tests des entités *Doctrine*

Les tests sur les entités recouvrent en partie les tests fonctionnels. Cependant, ils ont été utiles pendant l'implémentation et restent utiles si un test fonctionnel dysfonctionne, permettant ainsi de déterminer plus facilement à quel niveau le dysfonctionnement fonctionnel se produit.

Par exemple, dans le test suivant, on s'assure que la fonction de l'entité *Patient* *getTodayMedicalOpinionByDoctor*() renvoie bien l'objet attendu ou null.

```
// tests/integration/PatientTest.php
public function testGetTodayMedicalOpinionByDoctorReturnsAMedicalOpinion(): void
{
    // Arrange
    $patient = PatientFactory::new()->create();
    $doctor = DoctorFactory::new()->create();
    HospitalStayFactory::new()
        ->exitAfterToday()
        ->entryBeforeToday()
        ->create([
            'patient' => $patient,
            'doctor' => $doctor,
        ]
        );

    MedicalOpinionFactory::new()->create([
            'patient' => $patient,
            'doctor' => $doctor,
        ]
    );

    // Act
    $foundMedicalOpinion = $patient->getTodayMedicalOpinionByDoctor($doctor->object());

    // Assert
    $this->assertInstanceOf(MedicalOpinion::class, $foundMedicalOpinion);
}
```

Ce test est complété par le test *testGetTodayPrescriptionByDoctorReturnsNull()* qui vérifie l'absence de retour (*null*) si le séjour n'a pas prescription. Et le test *testGetTodayPrescriptionByDoctorReturnsNullIfPrescriptionIsNotMadeToday()* qui vérifie également le retour *null*, avec la présence uniquement d'une prescription faite la veille.
On teste ainsi les trois états importants de la base de données identifiés : absence de prescription, présence de prescription faite ce jour, présence de prescription d'une autre date.

Tests des validateurs

Comme pour les entités *Doctrine*, ces tests sont en partie redondants avec les tests fonctionnels mais permettent de s'assurer de la fiabilité des composants de manière indépendante, ce qui est essentiel pour la confiance et le débogage.

J'ai écrit les tests des contraintes en me référant au modèle proposé officiellement[47].

47 https://symfony.com/doc/current/validation/custom_constraint.html#testing-custom-constraints

J’ai créé une classe pour chaque test, car chaque test nécessite un validateur différent (méthode *createValidator*()). On teste simplement la réussite et l’échec du validateur.

Nous avons besoin d’un accès à la base de données pour vérifier la présence d’un avis médical déjà enregistré pour les mêmes critères patient/docteur/jour. Nous utilisons un stub de PHPUnit pour simuler les réponses du SGBD.

```
// tests/Validator/MedicalOpinionDateUnchangedValidator2Test.php
class MedicalOpinionDateUnchangedValidator2Test extends ConstraintValidatorTestCase
{
    #[Override]
    protected function createValidator(): ConstraintValidatorInterface
    {
        $existingMedicalOpinion = new MedicalOpinion();
        $mockRepository = $this->createMock(MedicalOpinionRepository::class);
        $mockRepository->method('findOneBy')->willReturn($existingMedicalOpinion);

        return new MedicalOpinionDateUnchangedValidator($mockRepository);
    }

    public function testMedicalOpinionExists(): void
    {
        $testedMedicalOpinion = new MedicalOpinion();
        $testedMedicalOpinion->setPatient(new Patient());
        $testedMedicalOpinion->setDoctor(new Doctor());

        $this->validator->validate($testedMedicalOpinion, new MedicalOpinionDateUnchanged());
        $this->buildViolation('La création de cet objet est limitée à 1 par jour par patient et par docteur')
            ->assertRaised();
    }
}
```

Veille de sécurité

Pour l'écriture du code j'ai respecté le plus possible les bonnes pratiques de sécurité de *Doctrine* (utiliser les requêtes préparées, les méthodes *find*, etc), de *Symfony*[48] et j'ai adopté une approche défensive. Malgré une utilisation correcte, cela n'empêche pas les composants logiciels, le framework, les composants d'infrastructure et les machines de présenter des vulnérabilités.

En fin de projet, j'ai effectué un audit de sécurité complet pour déterminer si le projet présentait des vulnérabilités et établir comment faire la veille de sécurité. Ce rapport de sécurité est disponible dans les documents annexes, page 110.

Veille de sécurité back-end, dépendances Composer

Le code PHP constitue le cœur applicatif du projet, avec de nombreuses dépendances directes et indirectes. Le conteneur PHP n'est pas exposé directement au web, mais la couche d'entrée (serveur Nginx) laisse passer toutes les requêtes qui ne correspondent pas à des fichiers existants. Une veille de sécurité est donc très importante à ce niveau.

Le script de sécurité est simplement :

```
"security": [
  "composer audit",
  "./bin/console importmap:audit"
],
```

Ce script lance la commande composer audit, qui interroge la base de vulnérabilités de Packagist.org pour vérifier si une version d'un composant présente une vulnérabilité. La seconde commande, *importmap:audit*[49], est fournie par le package Symfony AssetMapper, qui permet l'installation de bibliothèques JavaScript, notamment Bootstrap utilisé dans ce projet. Sur le même principe, cette commande interroge le service GitHub[50] pour indiquer si un composant JavaScript utilisé est vulnérable.
Ce script de sécurité est lancé avant la réalisation de chaque commit ; la création du commit échoue si la commande ne se termine pas positivement, c'est-à-dire si une vulnérabilité est détectée par la commande *Composer*. La même procédure est appliquée dans le processus d'intégration continue : il devient impossible de déployer l'application si des vulnérabilités sont présentes, ce qui force ainsi la correction des dépendances.
En dehors du cadre des modifications de code, il est de notre ressort de s'assurer que notre application ne devient pas vulnérable. Pour cela, il suffit de lancer la commande *docker compose -f compose-dev.yaml exec php composer run-script ci* sur une machine de développement. Avec le task runner Just, il suffit de lancer *just composer security*.
Pour compléter l'automatisation, j'ai programmé le job *intégration*, qui s'exécute quotidiennement et me donne une alerte *GitHub* en cas d'échec. Cela permet de continuer la veille de sécurité même lorsque nous ne faisons plus de modifications de code ou de nouveaux déploiements.

48 https://symfony.com/doc/current/best_practices.html#security

49 https://symfony.com/doc/current/frontend/asset_mapper.html#run-security-audits-on-your-dependencies

50 https://symfony.com/blog/new-in-symfony-6-4-assetmapper-improvements

Cette veille de sécurité repose sur l'exécution de scripts de *Composer*. Il convient donc également de réaliser une veille sur *Composer* lui-même, car *Composer* peut vérifier s'il peut être mis à jour, mais il ne peut pas vérifier s'il présente des vulnérabilités.

Veille sur l’exécutable Composer

Pour mettre en place cette veille, je me suis appuyé sur les services de osv.dev . Une commande shell suffit pour vérifier si notre version de Composer est vulnérable, via l'API de osv.dev. J'ai écrit un petit script qui effectue cette vérification ; il renvoie 0 s'il n'y a pas de vulnérabilité et une autre valeur dans le cas contraire. Avec ce fonctionnement, on peut inclure le script dans la chaîne d'intégration continue et être alerté par GitHub en cas de vulnérabilité. Le script est à la racine du projet, non accessible par le web, dans le fichier check_composer.sh[51].

Le script est à la racine du projet, non accessible par le web, dans le fichier .

```
#!/usr/bin/env sh
version=$(composer --version |  awk '{print $3}')
echo "Composer version extracted : $version"
echo "Querying OSV API for Composer vulnerabilities... (exit code is the result)"
# Ckecking osv response is {}
curl -sd '{"version": "'"$version"'",  "package": {"name": "composer/composer", "ecosystem":
"Packagist"}}' "https://api.osv.dev/v1/query" \
| grep "^{}$" -q
```

51 https://github.com/SebSept/soignemoi-api/blob/main/check_composer.sh

Situation de recherche

Généralités

Ayant choisi de réaliser le projet sur la base de *Symfony* que je ne connais pas complètement, j'ai souvent été amené à réaliser des recherches. La plupart du temps, j'ai trouvé mes réponses dans la documentation officielle. J'ai aussi acquis des connaissances manquantes à la réalisation des applications avec le site SymfonyCasts[52].
De manière générale, je me réfère aux documentations officielles. Quand elles sont un peu trop verbeuses et que je manque de temps, je fais des recherches par un moteur de recherches (détaillé dans la recherche concrète).
J'ai également utilisé *GitHub Copilot*[53], qui est utile pour résoudre des problèmes lorsqu'on lui fournit des messages d'erreur peu compréhensibles ou pour amorcer une recherche quand je n'ai pas suffisamment de connaissances pour commencer une recherche pertinente.

Finalement, en excluant les solutions trouvées dans les documentations officielles, les problèmes simples résolus avec GitHub Copilot, et les informations fournies par SymfonyCasts, j'ai fait très peu de recherches en utilisant un moteur de recherche.

Recherche concrète

Pour mettre en place les échanges entre l'API et les clients, la solution répandue est d'utiliser des tokens d'authentification. Ma recherche a consisté à déterminer comment générer un token fiable.

J'ai commencé par faire une recherche *Google* avec les mots clés « generate api token php »[54]

J'ai ouvert les premiers résultats dans de nouveaux onglets :

1. https://www.codexworld.com/how-to/generate-unique-and-secure-api-keys-with-php/
2. https://medium.com/winkhosting/create-a-basic-php-api-with-token-authentication-96111eada51
3. https://stackoverflow.com/questions/1448455/php-api-key-generator
4. https://stackoverflow.com/questions/38546905/php-generating-tokens-for-transactions
5. https://stackoverflow.com/questions/42046387/generate-php-access-token
6. https://stackoverflow.com/questions/21378988/how-can-i-generate-strong-unique-api-keys-with-php
   J'ai ignoré les 3 résultats suivants qui ne semblaient pas répondre à ma problématique.
   J'ai ouvert les 2 liens suivants :
7. https://www.cloudways.com/blog/symfony-api-token-authentication/
8. https://symfonycasts.com/screencast/api-platform-security/token-string-fixtures

Une fois ces résultats collectés, j'ai regardé leurs contenus et fais les analyses suivantes :

52 https://symfonycasts.com/

53 https://github.com/features/copilot/

54 https://www.google.com/search?q=generate+api+token+php

1. J’ai regardé la solution proposée par le premier site (https://www.codexworld.com/how-to/generate-unique-and-secure-api-keys-with-php/) , je l’ai gardé en mémoire et j’ai fermé l’onglet. J’ai accordé peu de crédit à cette solution, car je ne connais pas le site et il n’y a aucune justification de son utilisation.
2. J’ai parcouru rapidement le second lien (https://medium.com/winkhosting/create-a-basic-php-api-with-token-authentication-96111eada51) et constaté que le sujet était plus vaste que le mien. J’ai effectué une recherche du mot "token" dans la page et constaté l’absence de réponse, puis j’ai fermé l’onglet.
3. Pour le premier lien de stackoverflow.com (https://stackoverflow.com/questions/1448455/php-api-key-generator) J’ai vérifié la date, un critère important pour déterminer si la solution est potentiellement périmée. Le résultat date de 14 ans, ce qui me semble problématique. Malgré cela, j’ai parcouru le contenu et noté les solutions proposées uniquement pour accumuler de l’information. Toutefois, la solution est trop ancienne pour que je la considère fiable, car PHP a beaucoup évolué en 14 ans.
4. Pour ce résultat, (https://stackoverflow.com/questions/38546905/php-generating-tokens-for-transactions), la question n’est pas exactement la mienne, mais la problématique similaire et la réponse à été validée 15 fois, donc assez fiable à mon avis. Je note le résultat cohérent avec ce que j’ai déjà obtenu.
5. Le résultat suivant (https://stackoverflow.com/questions/42046387/generate-php-access-token) fait référence à une implémentation personnalisée. Je rejette la question (et la réponse) car je souhaite utiliser une fonctionnalité native, la plus simple possible, car elle sera plus fiable.
6. Le résultat suivant (https://stackoverflow.com/questions/21378988/how-can-i-generate-strong-unique-api-keys-with-php), correspond exactement à ma demande. Bien que la question soit assez ancienne, la réponse obtenue a déjà été confirmée précédemment, ce qui renforce sa validité. Je la retiens.
7. Le résultat ne correspond pas à ce que je recherche, je l’ignore. https://www.cloudways.com/blog/symfony-api-token-authentication/
8. Le dernier onglet ouvert, https://symfonycasts.com/screencast/api-platform-security/token-string-fixtures est une source que je juge très fiable. J’ai déjà consulté beaucoup de contenu sur ce site, qui est directement référencé sur le site officiel de *Symfony* et dont le contenu est l’œuvre d’un contributeur reconnu de Symfony, notamment en ce qui concerne la sécurité. Je note que la solution proposée correspond à des solutions déjà trouvées plus haut.

J’ai trouvé ce qu’il me faut ; SymfonyCasts fait autorité et j’ai pu confirmer le choix effectué. Cependant, je lance une nouvelle recherche car la solution trouvée manque de justification. Pour un enjeu lié à la sécurité, je ne peux pas me contenter d’un simple copié-collé de solution sans étudier moi-même le problème de manière plus approfondie.
J’ai donc lancé une nouvelle recherche sur duckduckgo.com avec « random token generation php » [55], et le premier résultat est cette fois très satisfaisant en termes de justification : https://stackoverflow.com/questions/18910814/best-practice-to-generate-random-token-for-forgot-password#18910943. La réponse a fait l’objet de discussions, de références sérieuses, et elle est

55 https://duckduckgo.com/?t=ffab&q=random+token+generation+php&ia=web&iax=qa

validée par des personnes très compétentes (contributeurs PHP), obtenant ainsi 156 votes positifs. Pour finaliser ma recherche, j'ai effectué une dernière vérification sur la longueur du token. J'ai ouvert une dizaine de résultats et les ai jugés en fonction du site, de la crédibilité des répondants, de la qualité et de la quantité des discussions, ainsi que des références. J'en ai conclu que 64 caractères était une longueur de chaîne assurant la sécurité. J'ai noté cette page en référence https://security.stackexchange.com/questions/94630/token-based-authentication-whats-a-good-token-length

Annexes

Énoncé du travail pour le projet en cours de formation (ecf)

TP Concepteur développeur d'applications

Énoncé

Durée indicative : 70 h
Documents autorisés : annexes
Matériel autorisé : calculatrice, tableur, etc.

Votre examen comporte :

- ✓ Cet **énoncé** qui vous présente le sujet de l'épreuve
- ✓ Une **copie à rendre** (Excel ou Word) que vous devez télécharger, remplir informatiquement et déposer dans l'espace de dépôt prévu à cet effet.

Renommer votre copie à rendre Word ou Excel comme suit :
TP_CDA_déc2022_copiearendre_NOM_Prenom

Objectifs de l'évaluation :
L'évaluation en cours de formation que vous allez réaliser a pour vocation de figurer dans votre **livret d'évaluation**. Il sera donc remis à votre jury le jour des épreuves du titre professionnel accompagné de votre évaluation et du sujet initial.

Nous vous demandons de vous mettre en situation d'examen. Prenez le temps de lire le sujet et de manipuler les annexes afin de répondre en situation professionnelle aux questions et problématiques évoquées dans le sujet

À vous de jouer !

Présentation de l'entreprise

SoigneMoi, est un hôpital de la région lilloise (dans le nord de la France), cet hôpital n'a pas beaucoup de clients par manque d'efficacité. Ils sont assez lents dans l'accueil des patients, mais également dans la prise en charge. Les praticiens, qui doivent effectuer des opérations, n'ont pas d'agenda informatisé, de ce fait, toute la gestion des plannings est faite à la main.

L'hôpital ne pouvant plus continuer en ce sens, ils ont décidé d'investir afin de proposer des outils permettant de corriger toutes les faiblesses actuelles. Pour répondre à ce besoin, ils ont mandaté une entreprise, Develop-Solution, dont vous faites partie pour développer les applications nécessaires.

Après consultation avec les équipes de SoigneMoi, il a été défini les besoins suivants :

- Application Web :
  - Possibilité de création de comptes utilisateur
  - Un utilisateur peut sélectionner une date de séjour ainsi qu'un motif afin que l'hôpital puisse préparer au mieux son accueil
  - L'utilisateur possède sur son espace, un historique des séjours que l'utilisateur a effectué.

- Application mobile :
  - Application à destination des médecins
  - Il est possible qu'un patient ait besoin de prescription
    - Un médecin peut saisir sur son application mobile des prescriptions pour un patient, il donne également un avis sur le statut du patient

- Application bureautique :
  - Application à destination des secrétaires de l'hôpital
  - Une / Un secrétaire a accès à toutes les admissions à l'hôpital du jour.
    - Gestion des entrées et sortie de l'hôpital

Description du projet

1. Application Web

US 1 : Page d'accueil

Utilisateur concerné : Visiteur

La page d'accueil doit comporter obligatoirement une description de l'hôpital ainsi qu'une liste des prestations qu'il propose.

US 2 : Création d'un compte utilisateur

Utilisateur concerné : Visiteur

Un Visiteur doit être capable de créer un compte en saisissant, un mot de passe, suivi d'un mail. Il doit préciser obligatoirement, un nom ainsi qu'un prénom avec son adresse postale.

US 3 : Espace Utilisateur

Utilisateur concerné : Utilisateur

Un Utilisateur, depuis son espace, doit être capable de visualiser tout son historique de séjour : séjour effectué, en cours ou encore à venir. Les séjours ne sont pas modifiables.

US 4 : Création d'un séjour

Utilisateur concerné : Utilisateur

Depuis le site web, il est possible de créer un séjour. Pour ce faire, le visiteur doit cliquer sur le bouton présent dans le menu principal « Séjour » et doit choisir une date de début ainsi que de fin sur un calendrier.

Une fois la date sélectionnée, il doit préciser :

- Le motif de son séjour
- La spécialité nécessaire (Chirurgien, etc.)
- Le médecin qu'il souhaite

À la fin de cette saisie, un bouton « confirmation » doit être présent, mais cliquable, uniquement si le visiteur est authentifié.

US 5 : Gestions des plannings des médecins

Utilisateur concerné : Administrateur

Un Administrateur, peut depuis son espace, créer des médecins et leur associer un emploi du temps. Un médecin peut prendre en charge 5 patients par jours au maximum.

Un médecin est caractérisé par :

- Un nom ainsi qu'un prénom
- Une spécialité (chirurgie, etc.)
- Un matricule

2. **Application mobile**

US 6 : Prescriptions

Utilisateur concerné : Médecin

Un médecin doit être capable de saisir depuis son mobile, la prescription qu'il donne à un patient. Chaque jour, à 10h, il visite chaque patient et donne un avis ainsi qu'une prescription qui sera ajoutée dans le dossier du patient.

Un avis est caractérisé par :

- Un libelle : titre de l'avis
- Une date
- Une description
- Le nom et prénom du médecin

Une prescription est caractérisée par :

- Une liste de médicament
- Une posologie pour chacun d'entre eux
- Une date de début de traitement ainsi qu'une date de fin
- La date de fin peut être modifiable, si le médecin juge que le patient est soigné

3. **Application bureautique**

US 7 : Gestion des entrées et sortie

Utilisateur concerné : secrétaire

Pour le jour courant, Un / Une secrétaire peut visualiser les patients effectuant une entrée ou une sortie le jour de la consultation du logiciel. Il devra être possible d'accéder au détail du dossier du patient.

US 8 : Visualisation du détail des entrées et sorties

Utilisateur concerné : secrétaire

Au clic sur un patient depuis l'US précédente, il doit être possible de consulter tout le détail du dossier patient. L'ensemble de ses informations personnelles, ainsi que les dates de son séjour suivi du motif, et enfin, les prescriptions et avis donnés par le médecin durant le séjour.

4. **Jeu de test**

Au minimum une de vos applications doit être pourvue d'un environnement de test complet. Une des fonctionnalités sera alors obligatoirement vérifiée en intégralité, et ce à l'aide :

- Tests unitaires
- Tests fonctionnels
- etc.

Objectif & Livrables

L'objectif de cet ECF est que vous passiez sur chaque étape nécessaire à l'élaboration d'une solution de qualité : analyse des besoins, maquettage, intégration, développement des règles de gestion, ainsi que le déploiement.

Vous devrez détailler dans votre copie les dispositions que vous avez prises notamment sur la sécurité aussi bien front, back que l'intégrité de vos données en base. De plus, vous devrez justifier tous vos choix techniques : il est très important de pouvoir les justifier et pouvoir exposer votre point de vue sur différentes technologies.

Les livrables de ce sujet sont les suivants :

- ➔ Le lien du (ou des) dépôt(s) github PUBLIC où sera présent le code de vos applications
- ➔ Le (ou les) lien(s) de votre (vos) application(s) déployée(s)
- ➔ Le lien vers votre logiciel de gestion de projet (Jira, Notion, Trello , etc.)

Votre git, devra comporter :

- Un fichier README.md contenant la démarche à suivre afin de déployer votre application en local
- Les bonnes pratiques de git doivent être appliqué
  - Une branche principale
  - Une branche de développement
    - Chaque fonctionnalité sera une branche issue de la branche développement, après test, le merge sera effectué vers la branche développement
    - Une fois que la branche développement est correctement testée, il faudra effectuer un merge vers la branche principale
- Les fichiers de création de base de données, mais également d'intégration de données (cela peut être un fichier sql, une fixture ou encore une migration)
- Transaction SQL dans un fichier .sql
  - Vous devez proposer un document avec comme extensions '.sql' présentant une transaction SQL sur ce que vous souhaitez
    - Lors de votre explication de cette transaction dans votre document technique, vous devez présenter quelle est le but de cette requête
    - N'oubliez pas qu'une transaction SQL est un ensemble de requêtes SQL : tout est effectué ou rien.
- Manuel d'utilisation en format PDF

- Il doit présenter l’application et donner des identifiants afin de réaliser les différents parcours possibles.

- Une charte graphique en format PDF
  - Palette de couleurs utilisée ainsi que la police
  - L’export des maquettes attendues (wireframes & mockup pour mobile / web / bureautique) => 2 pour chaque
- Une documentation traitant de votre gestion de projet
  - Explication de votre gestion de projet
- Une documentation technique de votre application
  - Réflexions initiales technologiques sur le sujet
  - Configuration de votre environnement de travail
  - Modèle conceptuel de données (MCD)
  - Diagramme d’utilisation, diagramme de séquence
  - Explication de votre plan de test

Votre gestion de projet devra être sous la forme d’un kanban partagé afin d’avoir une vue globale de votre réalisation. Il devra comprendre :

- Une colonne recensant toutes les fonctionnalités prévues et ordonnées par priorité
- Une colonne recensant les fonctionnalités que vous comptez faire, prévues de développement (ou dans le sprint si vous êtes en agile)
- Une colonne qui recensant les fonctionnalités que vous faites actuellement
- Une colonne qui énumère les fonctionnalités terminées (sur la branche de développement)
- Enfin, vous pouvez effectuer une dernière colonne stipulant les fonctionnalités qui ont été merge dans la branche principale

Stack technique

Aucune technologie n’est obligatoire pour cet ECF, à l’exception de la base de données qui doit être relationnelle.

Voici un exemple de stack technique possible :

Application web :

- Front : HTML 5, CSS (Bootstrap), JS
- Back-end : PhP avec utilisation de PDO
- Base de données : MySQL ou MariaDB
- Déploiement : fly.io

Application mobile :

- Front : flutter
- Le front interroge le back-end de l’application web

Application bureautique :

- Python avec la librairie tkinter
- L’application interroge le back-end de l’application web

Note de prédémarrage

Pré-démarrage

Ce document résume la phase de pré-démarrage.

Rappel des phases

1. **Pré-démarrage**
2. Cadrage
3. Spécifications fonctionnelles, planification & chiffrage
4. Conception technique
5. Réalisation technique
6. Tests et correctifs
7. Mise en production, formation, maintenance
8. Garanties

La phase de pré-démarrage est la phase initial du projet. Son objet est de recueillir les besoins et contraintes du projet afin de permettre d’ébaucher le cadrage de projet.

Table des matières

Table des matières

Contexte....78
Faisabilité....78
Besoins fonctionnels....78
Besoin en ressources....78
Besoins techniques pour la réalisation du projet....78
- Planifier....78
- Documenter le projet....78
- Développer et déployer les applicatifs....79
  - Organiser le développement....79
  - Réaliser, coder....79
  - Stocker....79
  - Déployer....79
  - Application bureautique et application mobile....79
- Choix Techniques....79

Contexte

Le projet est réalisé dans le cadre de l’évaluation en cours de formation (ECF) demandée par l’école *Studi* pour le *Bachelor Développeur PHP/Symfony* préparant à la certification du Titre Professionnel “Concepteur développeur d'applications” niveau 6, enregistré au RNCP sous le numéro 31678.

C’est un projet individuel réalisé en autonomie avec des évaluations et corrections intermédiaires par un formateur de l’école *Studi*.

Faisabilité

Dans le contexte de ce projet, la faisabilité à une réponse imposée, la date de livraison est imposée également. En référence au cadre (Cout / Délai / Qualité), le cout sera la variable d’ajustement principale de gestion du projet. Le cout sera modulé par l’implémentation de fonctionnalités secondaires, non explicitement déterminée mais qui apporteraient beaucoup de valeur à la solution.

Besoins fonctionnels

Les besoins fonctionnels sont exprimés dans le document fourni par l’école (*documents/1.prédémarrage/ECF_Finale_ TPDREETS été 24 .pdf*) et seront récapitulés dans le cahier des charges.

Besoin en ressources

La durée indicative de réalisation du projet est de 70 heures. Je vais me référer à cette indication pour prendre les décisions de gestion de projet.

Je suis la seule personne impliquée sur le projet, mon temps disponible est réduit, je prévois de consacrer de 2,5 jours par semaine à la réalisation de ce projet.

Il n’y aura pas de sous traitance sur le projet.

Besoins techniques pour la réalisation du projet

J’ai ma machine de développement (Fedora Linux) et de ma connexion interne. Les besoins sont relatifs aux différents domaines de travail sur le projet.

Planifier

J’utilise le logiciel ganttproject pour la réalisation des plannings, roadmap et l’inscription des jalons.

Documenter le projet

Les documents relatifs au projet sont stockés sur ma machine dans un dossier dédié.

Les documents sont produits au format *Libre Office.* La suite Libre Office est donc requise. Les documents pourrons être transformés en pdf avec ma machine en fin de projet.

Étant donné l'absence de besoin de partage des documents (pas de développement collaboratif) et la contrainte de ressource forte, cette forme sera suffisante.

L'ensemble des documents sera ajouté au dépôt GitHub livré en fin de projet.

Développer et déployer les applicatifs

Organiser le développement

Les grandes phases du développement seront planifiées avec le logiciel *GanttProject*.

Les taches de développement seront concrètement gérées avec un logiciel comme Trello, Jira ou Kanboard (décision non actée).

Réaliser, coder

Le développement logiciel se fera à l'aide des outils de JetBrains (Phpstorm, IntelliJ IDEA) dont je dispose gratuitement dans le cadre de l'offre GitHub Student Developer Pack.

La lecture et la modification du code pourront se faire ultérieurement avec n'importe quel outil.

Stocker

Le stockage et le versionnage du code se feront avec *Git*.

Le code, son historique sera partagé sur GitHub.

Déployer

Le déploiement se fera idéalement automatiquement via les actions de GitHub.
Éventuellement, pour des raisons de rapidité il sera lancé à partir de la machine du développeur en utilisant un outil de déploiement pour veiller à ce que le process soit atomique, et intégré au projet.

Les applicatifs en ligne seront déployés sur des machines *Digital Ocean* (offre du *GitHub Student Pack*)

Application bureautique et application mobile

Les contraintes et choix techniques liés à l'application bureautique et mobile ne sont pas déterminés, une étude est nécessaire.

Choix Techniques

Les choix techniques seront réalisés de façon à produire une solution robuste, standard et maintenable. Dans un soucis de réduction des temps de développement et de formation, les technologies de référence seront préférées.

Besoins en formation

Il y aura un besoin en formation pour le développement mobile et bureautique. Ce temps sera intégré dans la phase de conception et la phase de réalisations des chaque partie.

Essentiel

- La contrainte forte est la date de livraison au 22 mai 2024
- La durée cible est de 70 heures
- Une seule personne réalise l’ensemble du travail
- Les besoins fonctionnels sont dans le document *documents/1.prédémarrage/ECF_Finale_ TPDREETS été 24 .pdf*
- Les logiciels *JetBrains* seront utilisés pour le développement
- Code versionné avec *Git*
- Code partagé sur *GitHub*
- Code déployé via *GitHub* (ou via outil local)
- Application déployée sur une machine *Digital Ocean*.
- Planification avec *GantProject*
- Outils de gestion des taches de développement à déterminer entre Kanboard, Trello et Jira.
- La réalisation de l’application bureautique et mobile devront être déterminés

Note de cadrage

Rappel des phases

1. Pré-démarrage
2. **Cadrage**
3. Spécifications fonctionnelles, planification & chiffrage
4. Conception technique
5. Réalisation technique
6. Tests et correctifs
7. Mise en production, formation, maintenance
8. Garanties

Cadrage

La phase de pré-démarrage est terminée et apporte les premiers éléments pour réaliser le cadrage du projet.

On va s’attacher à déterminer les besoins client, identifier les risques, déterminer les outils de gestion de projet et préparer la planification détaillée en identifiant les étapes de réalisation du projet.

Table des matières

Table des matières

Besoin client..... 84
Spécifications générales.....84
Périmètre fonctionnel.....84
Lot 1 : Application Web & et stockage des données.....84
Lot 2 : Application mobile..... 84
Lot 3 : Application bureautique.....84

Périmètre opérationnel....84
Gestion des risques....85
Identification et évaluations des risques....85
Applications mobiles et bureautique....85
Sécurité....85
Gestion de projet....86
Risques stratégiques....86
Outils de gestion de projet....86
Étapes clés de réalisation du projet....86
Parties prenantes....87
Livrables....87

Besoin client

Spécifications générales

Après consultation avec les équipes de *SoigneMoi*, nous avons définis les besoins.

A l'hôpital *SoigneMoi*, la gestion des planning est manuelle, ce qui entraine un manque d'efficacité.

Pour optimiser la planification, une solution informatisée de gestion des rendez-vous va être mise en place.

Le temps de développement étant très réduit, une solution simple, définie comme un socle de départ va être mise en place.

Une application web sera disponible pour les patients, une application mobile pour les médecins et une application bureautique pour les secrétaires.

Le critère de succès du projet consiste à vérifier que le nombre de rendez-vous quotidien moyen à augmenter (on donnerai des chiffres dans un vrai projet). On pourrait également vérifier que l'investissement dans le développement de l'application est amorti en quelques mois (on donnerai également un chiffre exacte dans la réalité et cela permet de validité l'utilité du projet).

Périmètre fonctionnel

Lot 1 : Application Web & et stockage des données

- Possibilité de création de comptes utilisateur
- Un utilisateur peut sélectionner une date de séjour ainsi qu'un motif afin que l'hôpital puisse préparer au mieux son accueil
- L'utilisateur possède sur son espace, un historique des séjours que l'utilisateur a effectué.

Lot 2 : Application mobile

- Application à destination des médecins
- Il est possible qu'un patient ait besoin de prescription
    - Un médecin peut saisir sur son application mobile des prescriptions
- pour un patient, il donne également un avis sur le statut du patient

Lot 3 : Application bureautique

- Application à destination des secrétaires de l'hôpital
- Une / Un secrétaire a accès à toutes les admissions à l'hôpital du jour.
    - Gestion des entrées et sortie de l'hôpital

Périmètre opérationnel

- Lot 1

    - api : Déploiement et mise en production d'un serveur *Linux* intégrant *Docker.*
      Le serveur stocke l'ensemble des données et sert de référence pour l'application web, l'application mobile et l'application bureautique.
    - Application web : l'application web est mise en place sur la même machine.
- Lot 2 : L'application mobile est livrée sous forme de fichier *apk* sans prestation de mise en place sur les terminaux des médecins.
- Lot 3 : L'application est livrée sous forme d'exécutable sans prestation de mise en place sur les ordinateurs.
- Versionning avec *Git*
- Partage du code sur GitHub
- Serveur web hébergé sur un VPS *Digital Ocean*.

Gestion des risques

Identification et évaluations des risques

Applications mobiles et bureautique

Les temps de réalisation et la qualité des livrables pour l'application mobile et bureautique sont sujets à des dépassements car ces technologies ne sont pas bien maitrisées par le développeur. Il y a un risque de dépassement de temps important.

Ce risque est jugé très important, sa probabilité de survenue élevée et la criticité élevée.

→ Pour limiter ce risque, il à été convenu d'utiliser les technologies les plus communément utilisées et notamment celle présentées dans les cours Studi. Pour s'assurer des bons choix, une étape de la conception sera spécifiquement dédiée à cette tâche.

→ La réalisation le plus tôt possible pour garder de la marge de manœuvre est à mettre en place

Le suivi du temps sera assuré en dédiant une phase de développement spécifiquement à la réalisation des applications mobiles et bureautique et en la découpant si cela est possible.

Le suivi du temps sera assuré en dédiant une phase de développement spécifiquement à la réalisation des applications mobiles et bureautique et en la découpant si cela est possible.

Sécurité

Les hôpitaux sont des cibles particulièrement prisées des hackeurs.

Ce risque est jugé important (sa probabilité de survenue est faible mais son impact majeur).

→ Pour limiter ce risque :

- la prise en compte de la sécurité sera explicitement intégrée à toutes les phases du projet.
- Le développeur intègrera les contraintes de sécurité dans chacune de ses réalisations sans délai, dès l'écriture du code et la configuration des outils.

Une partie de la phases de tests sera dédiée aux tests de sécurité.

Gestion de projet

Le gestionnaire de projet n'ayant jamais réalisé de gestion complète de projet, les risques d'évaluation et de planification peuvent exister.

Pour limiter ce risque, des marges de délais seront intégrées aux estimation. On se donnera la possibilité de revoir les choix technique si ils sont trop ambitieux ; tout en respectant les impératifs du projet.

Risques stratégiques

Les risques stratégiques sont faibles car les investissements financiers et temporels sont relativement faibles.

Outils de gestion de projet

L'étape de pré-démarrage nous a permis de faire quelques choix qui restaient a compléter.

- La planification est réalisée avec GantProject - https://www.ganttproject.biz/
  Les fichiers GantProject seront intégrés in fine au dépôt GitHub avec le code source. Ce choix est possible car il n'y a pas de collaboration sur le projet.
- La gestion des taches est réalisée avec Kanboard - https://kanboard.org/
  Éventuellement, une instance du logiciel sera accessible en ligne en consultation. Les fichiers seront également livrés dans le dépôt du projet.

Étapes clés de réalisation du projet

La planification vient détailler la roadmap et intégrer les phases demandées par la gestion des risques.

- Préparation du projet
  - 1. pré-démarrage
  - 2. Cadrage
  - 3. Planification, chiffrage et cahier des charges
- Réalisation
  - 4. Conception technique
    - API et application web
    - Application mobile
    - Application bureautique
  - 5. Réalisation technique
    - Mise en place du serveur (api + application web)

        - Développement de l’API
        - Développement de l’application web
        - Développement de l’application mobile
        - Développement de l’application bureautique
- 6. Tests et correctifs
    - Tests d’acceptance
    - Tests de sécurité
- 7. Mise en production, formation, maintenance

La phase de garantie n’est plus incluse dans le projet car il s’agit le livrer une solution minimale réalisé dans le cadre d’une évaluation en cours de formation.

Parties prenantes

Les parties prenantes sont le développeur/chef de projet et le client (fictif). Il n’y a pas d’échange entre les parties, nous n’avons pas de rôle à déterminer, le développeur/chef de projet fera les choix les plus susceptibles d’être réalistes dans le contexte du projet.

Livrables

- Le code source intégrale du projet
- les documentations et l’ensemble des fichiers et ressources utilisés pendant le développement
- les documents de projets
    - note de pré-démarrage
    - note de cadrage (présent document)
    - cahier des charges
    - plannings et roadmap
    - documents techniques

Roadmap

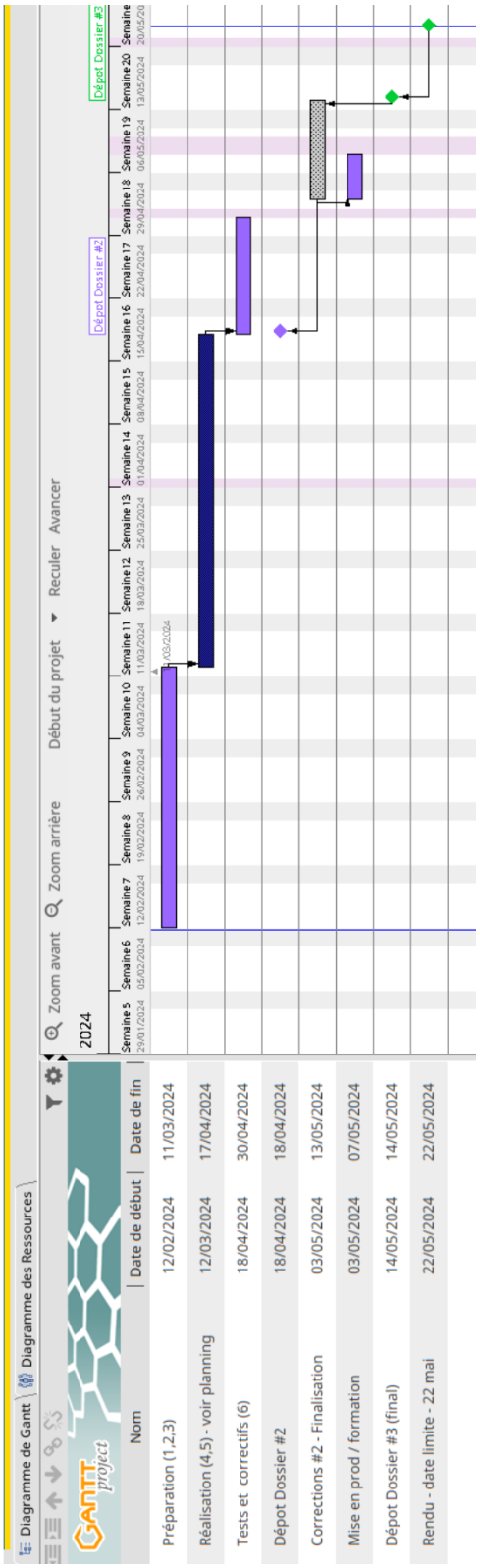


Diagramme de Gantt
Diagramme des Ressources
Zoom avant
Zoom arrière
Début du projet
Reculer
Avancer
2024
Dépot Dossier #2
Dépot Dossier #3
Nom
Date de début
Date de fin
Préparation (1,2,3)
12/02/2024
11/03/2024
Réalisation (4,5) - voir planning
12/03/2024
17/04/2024
Tests et correctifs (6)
18/04/2024
30/04/2024
Dépot Dossier #2
18/04/2024
18/04/2024
Corrections #2 - Finalisation
03/05/2024
13/05/2024
Mise en prod / formation
03/05/2024
07/05/2024
Dépot Dossier #3 (final)
14/05/2024
14/05/2024
Rendu - date limite - 22 mai
22/05/2024
22/05/2024

Planning

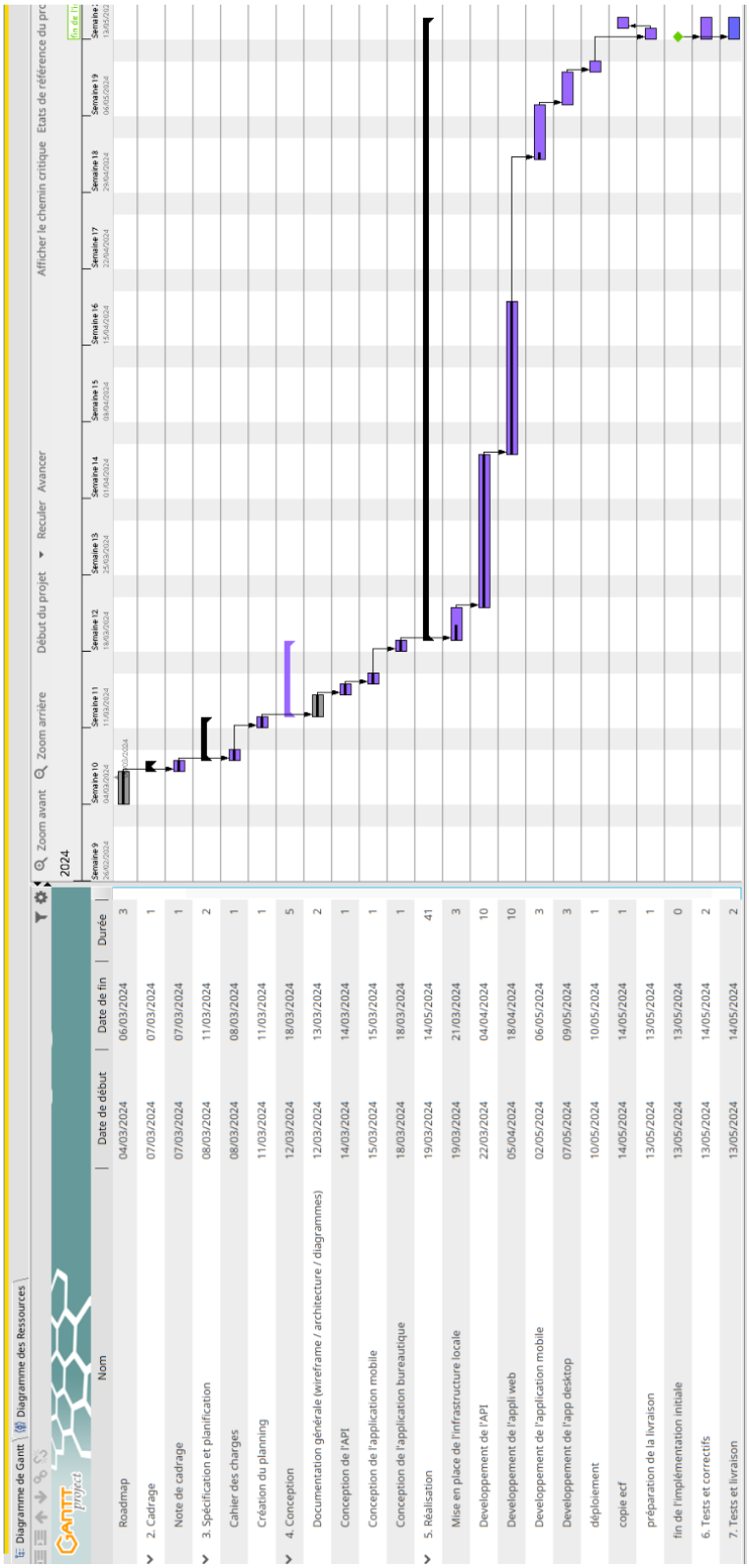


Diagramme de Gantt
Diagramme des Ressources
Zoom avant
Zoom arrière
Début du projet
Reculer
Avancer
Afficher le chemin critique
2024
Nom
Date de début
Date de fin
Durée
Roadmap 04/03/2024 06/03/2024 3
2. Cadrage 07/03/2024 07/03/2024 1
Note de cadrage 07/03/2024 07/03/2024 1
3. Spécification et planification 08/03/2024 11/03/2024 2
Cahier des charges 08/03/2024 08/03/2024 1
Création du planning 11/03/2024 11/03/2024 1
4. Conception 12/03/2024 18/03/2024 5
Documentation générale (wireframe / architecture / diagrammes) 12/03/2024 13/03/2024 2
Conception de l'API 14/03/2024 14/03/2024 1
Conception de l'application mobile 15/03/2024 15/03/2024 1
Conception de l'application bureautique 18/03/2024 18/03/2024 1
5. Réalisation 19/03/2024 14/05/2024 41
Mise en place de l'infrastructure locale 19/03/2024 21/03/2024 3
Développement de l'API 22/03/2024 04/04/2024 10
Développement de l'appli web 05/04/2024 18/04/2024 10
Développement de l'application mobile 02/05/2024 06/05/2024 3
Développement de l'app desktop 07/05/2024 09/05/2024 3
déploiement 10/05/2024 10/05/2024 1
copie ecf 14/05/2024 14/05/2024 1
préparation de la livraison 13/05/2024 13/05/2024 1
fin de l'implémentation initiale 13/05/2024 13/05/2024 0
6. Tests et correctifs 13/05/2024 14/05/2024 2
7. Tests et livraison 13/05/2024 14/05/2024 2

Cahier des charges

Document

Version	1.1
Date de modification	22/03/24 07:47:18

Rappel des phases

1. Pré-démarrage
2. Cadrage
3. **Spécifications fonctionnelles, planification & chiffrage**
4. Conception technique
5. Réalisation technique
6. Tests et correctifs
7. Mise en production, formation, maintenance

Ce cahier des charge est destiné à compléter la note de cadrage et à fournir un document contractuel pour définir les besoins, les spécifications fonctionnelles, les délais, planning et livrables du projet.

Table des matières

Préambule....93
Présentation....93
Besoins....93
Besoins fonctionnels....93
Application Web....93
US 1 : Page d’accueil....93
US 2 : Création d’un compte patient....94
US 3 : Espace patient....94
US 4 : Création d’une demande de séjour....94
US 5 (optionnelle) : Gestions des plannings des médecins....95
Application mobile....96
US 6 : Prescriptions....96
Application bureautique....96
US 7 : Gestion des entrées et sortie....96
US 8 : Visualisation du détail des entrées et sorties....98
Serveur API....98
Besoins non fonctionnels....98
Exigences techniques....98
API....98
Diagramme MCD....98
Application web....99
Serveur....99
Application mobile....100
Application bureautique....100
Livrables....100
Planning prévisionnel....100
Gestion du projet....101
Processus de suivi et de reporting....101
Risques et mesures d'atténuation....101
Identification et évaluations des risques....101
Applications mobiles et bureautique....101
Sécurité....101
Gestion de projet....102
Risques stratégiques....102
Notes complémentaires....102

Préambule

Le projet est réalisé dans le cadre de l'évaluation en cours de formation (ECF) demandée par l'école *Studi* pour le *Bachelor Développeur PHP/Symfony* préparant à la certification du Titre Professionnel "Concepteur développeur d'applications" niveau 6, enregistré au RNCP sous le numéro 31678.

C'est un projet individuel réalisé en autonomie avec des évaluations et corrections intermédiaires par un formateur de l'école *Studi*.

Présentation

Après consultation avec les équipes de SoigneMoi, nous avons définis les besoins.

A l'hôpital SoigneMoi, la gestion des planning est manuelle, ce qui entraine un manque d'efficacité.

Pour optimiser la planification, une solution informatisée de gestion des rendez-vous va être mise en place.

Le temps de développement étant très réduit, une solution simple, définie comme un socle de départ va être mise en place.

Une application web sera disponible pour les patients, une application mobile pour les médecins et une application bureautique pour les secrétaires.

Le critère de succès du projet consiste à vérifier que le nombre de rendez-vous quotidien moyen à augmenter (on donnerai des chiffres dans un vrai projet). On pourrai également vérifier que l'investissement dans le développement de l'application est amorti en quelques mois (on donnerai également un chiffre exacte dans la réalité et cela permet de validité l'utilité du projet).

Besoins

Besoins fonctionnels

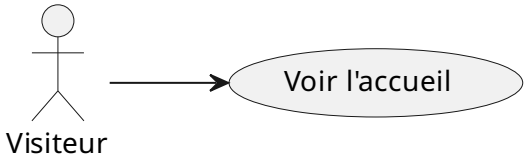
La solution se décompose en 4 parties : 3 applications et une api qui centralise les données et actions.

Application Web

US 1 : Page d'accueil

Utilisateur concerné : Visiteur (non authentifié).

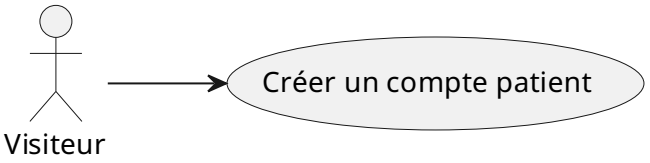
La page d'accueil doit comporter obligatoirement une description de l'hôpital ainsi qu'une liste des prestations qu'il propose.



US 2 : Création d'un compte patient

Utilisateur concerné : Visiteur (non authentifié)

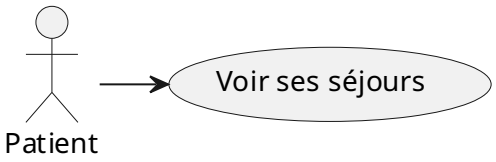
Un visiteur doit être capable de créer un compte en saisissant, un mot de passe, suivi d'un mail. Il doit préciser obligatoirement un nom ainsi qu'un prénom avec son adresse postale. Le compte utilisateur crée est assigné au rôle de patient.



US 3 : Espace patient

Utilisateur concerné : Patient

Un patient, depuis son espace, doit être capable de visualiser tout son historique de séjour : séjours effectués, en cours ou à venir. Les séjours ne sont pas modifiables par le patient.



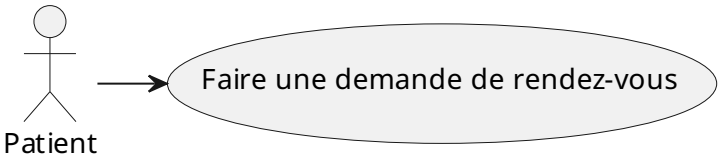
US 4 : Création d'une demande de séjour

Utilisateur concerné : Patient

Depuis le site web, il est possible de créer un séjour. Pour ce faire, l'utilisateur doit être authentifié avec un rôle de patient.
Le patient sélectionne :

- Une date de début de séjour
- Une date de fin de séjour
- Le motif de son séjour
- La spécialité nécessaire (Chirurgie, etc.)
- Le médecin qu'il souhaite

La demande de séjour est effectuée.
Le séjour sera confirmé par un administrateur. (US 5)



US 5 (optionnelle) : Gestions des plannings des médecins

Utilisateur concerné : Administrateur

Un Administrateur peut, depuis son espace :

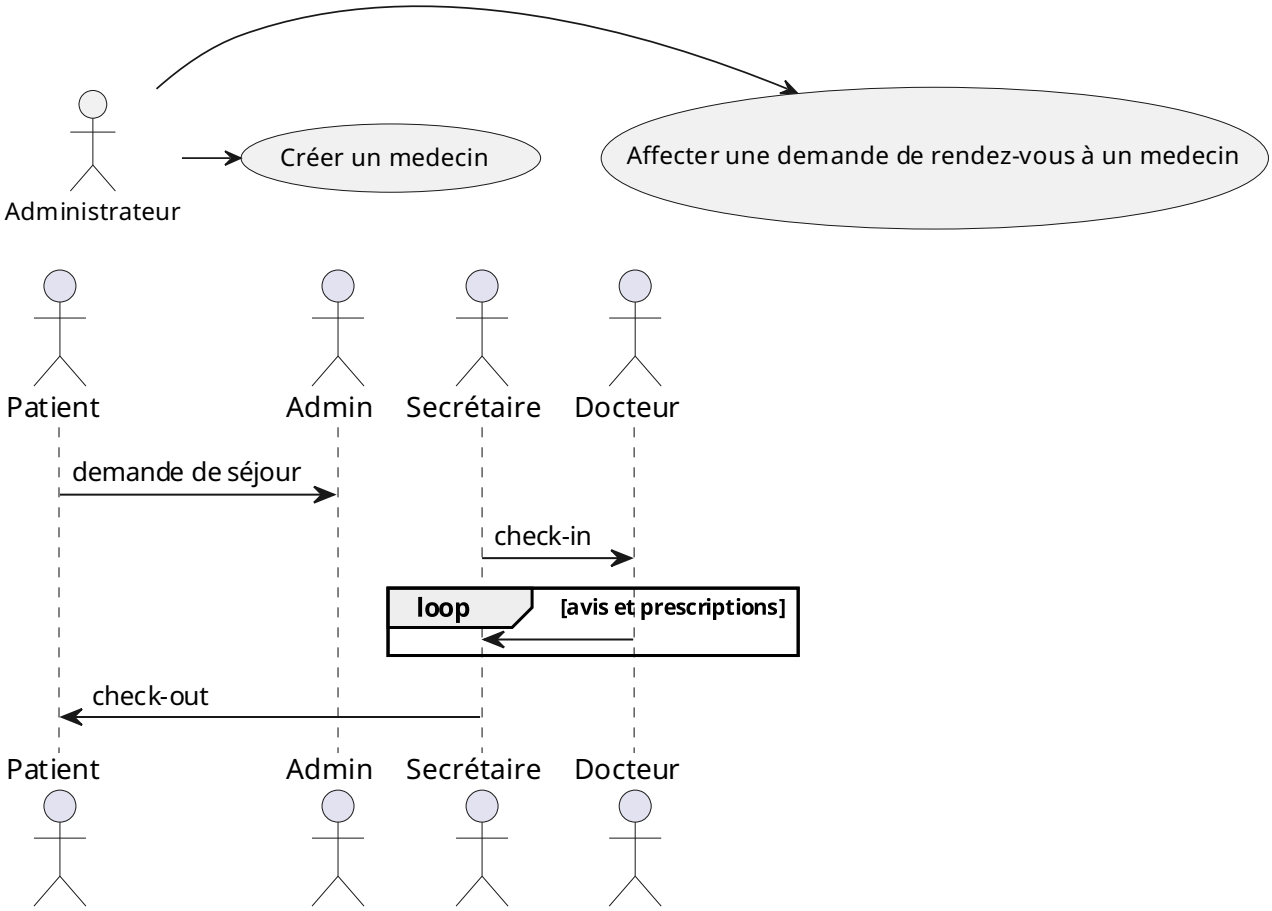
- créer des médecins
- leur associer un emploi du temps.

Associer un emploi du temps consiste à confirmer/modifier une demande de séjour. Le rendez-vous est alors validé et peut apparaitre dans les entrées à faire (et par conséquent les patients à traiter pour le médecin).

Un médecin peut prendre en charge 5 patients par jours au maximum.

Un médecin est caractérisé par :

- un nom
- un prénom
- une spécialité (chirurgie, etc.)
- un matricule



Application mobile

US 6 : Prescriptions

Utilisateur concerné : Médecin

Chaque jour, à 10h, un médecin visite chaque patient.
Depuis son mobile, un médecin doit être capable de :

- faire une prescription qui sera ajoutée dans le dossier du patient.
- donner un avis

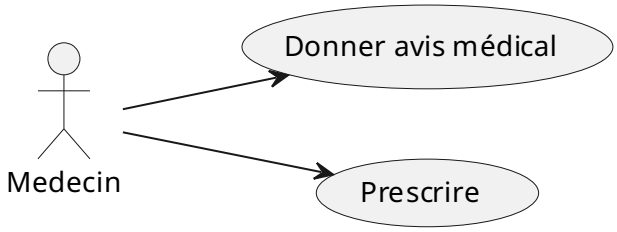
Un seul avis et une seule prescription sont possibles sur une journée./+
Les avis et prescriptions sont modifiables le jour de leur création uniquement.

Un avis est caractérisé par :

- Un libelle : titre de l'avis
- Une date
- Une description
- Le nom et prénom du médecin

Une prescription est caractérisée par :

- Une liste de médicament
- Une posologie pour chacun d'entre eux
- Une date de début de traitement
- Une date de fin de traitement
- La date de fin peut être modifiable, si le médecin juge que le patient est soigné



Application bureautique

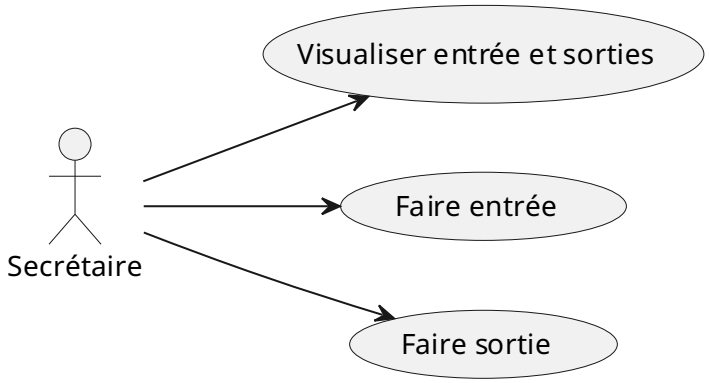
US 7 : Gestion des entrées et sortie

Utilisateur concerné : secrétaire

Pour le jour courant, Un / Une secrétaire peut

- visualiser les patients effectuant une entrée ou une sortie le jour de la consultation du logiciel.
- Il devra être possible d'accéder au détail du dossier du patient.

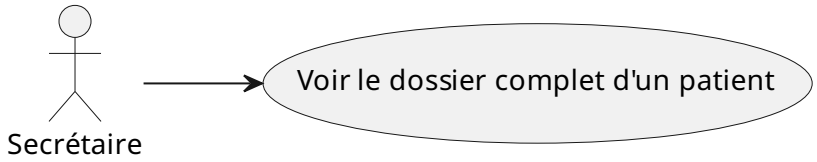
- indiquer qu’un patient effectue son entrée, l’heure de cette entrée est enregistrée dans les informations du séjour.
- indiquer qu’un patient effectue sa sortie, l’heure de cette entrée est enregistrée dans les informations du séjour.



US 8 : Visualisation du détail des entrées et sorties

Utilisateur concerné : secrétaire

Au clic sur un patient depuis l'*US* précédente, il doit être possible de consulter tout le détail du dossier patient. L'ensemble de ses informations personnelles, ainsi que les dates de son séjour suivi du motif, et enfin, les prescriptions et avis donnés par le médecin durant le séjour.



Serveur API

Le serveur d'api répond aux demandes de consultation,création, modification et suppression des différents objets du système : médecins, rendez-vous et patients.

Besoins non fonctionnels

La solution, c'est à dire l'ensemble des livrables ne comporteront pas de failles de sécurité ou de vulnérabilité.

Le serveur d'api est le nœud du système, il stocke les informations et répond aux demandes de création/modifications des objets du système : médecins, rendez-vous et patients.

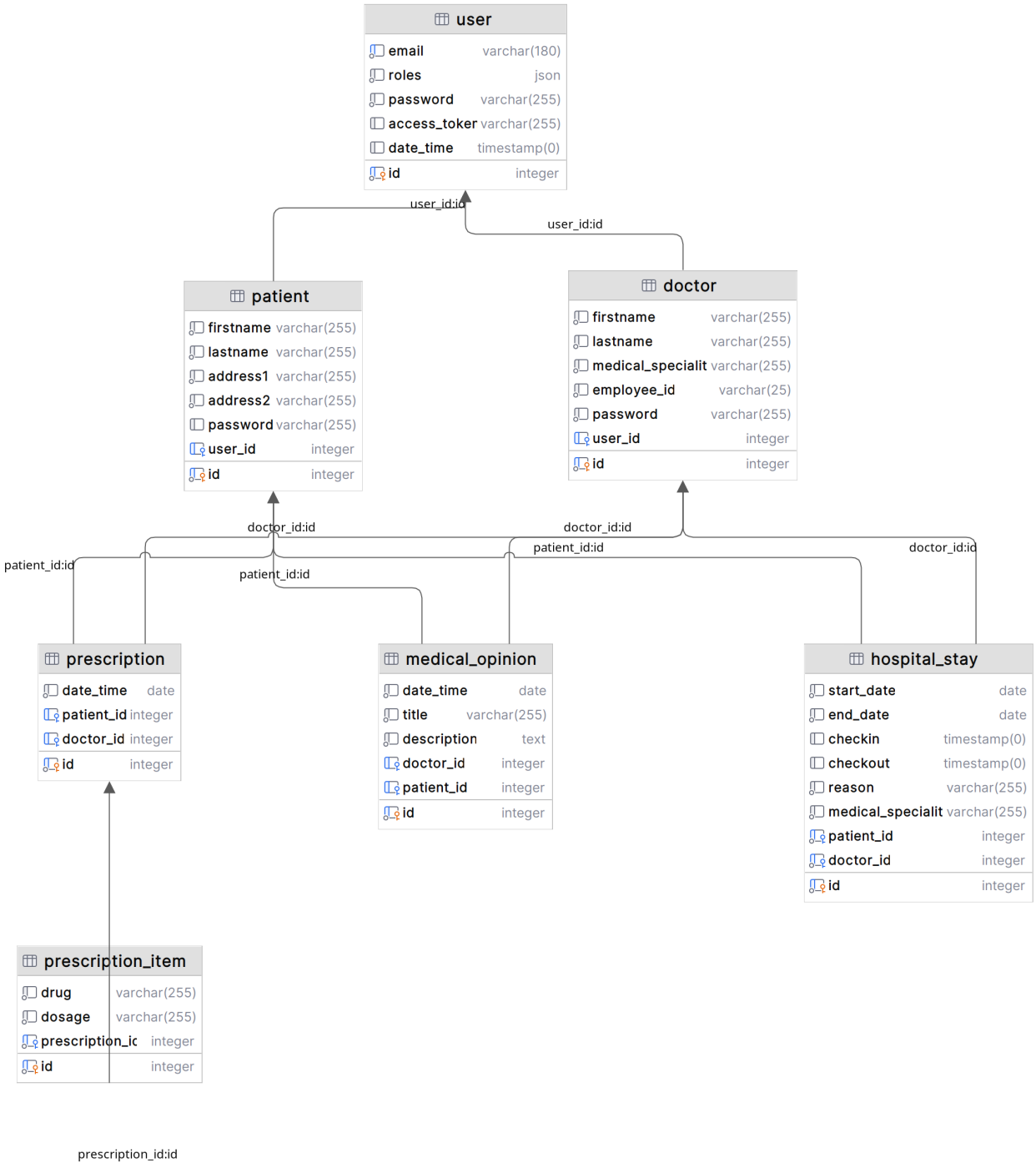
Exigences techniques

API

- L'API est développée en PHP avec Symfony 7
- L'API est conforme au standards open api - https://www.openapis.org/
- L'API permet les opérations CRUD en respectant les droits définis dans la documentation technique.

Diagramme MCD

Diagramme du Modèle Conceptuel de Données.



Application web

- L'application est développée avec Symfony 7
- est utilisable sur mobiles et ordinateurs sans dégradation de l'expérience

Serveur

- L'API et l'application web sont hébergées sur la même machine.
- Le serveur est une instance *Digital Ocean* opérationnelle pour l'utilisation de *Docker*

- L’API et l’application web fonctionnent par *Docker*
- Le serveur est accessible par internet

Application mobile

- Est une application mobile utilisable sur les systèmes Android
- Les technologies utilisées pour coder l’application seront définies lors de la conception, à la discrétion du développeur

Application bureautique

- Est un programme utilisable sur Linux à minima
- Les technologies utilisées pour coder l’application seront définies lors de la conception, à la discrétion du développeur

Livrables

- Le code source intégral du projet
- les documentations
- Les fichiers et ressources utilisés pendant le développement
- les documents de projets
  - notes de pré-démarrage
  - note de cadrage (présent document)
  - cahier des charges
  - plannings et roadmaps
- Une application mobile sous forme de fichier *apk* (Android Package).
- Une application mobile sous une forme déterminée par le développeur.
- Un serveur hébergeant l’api et l’application web en état de fonctionnement.

Planning prévisionnel

Le fichier livrables/documents/RoadMap.gan contient la roadmap du projet.

Le fichier livrables/documents/Planning.gan contient le planning détaillé de la réalisation.

- 12/02 : démarrage de la préparation du projet (pré-démarrage, cadrage, spécifications et planification)
- 12/03 : démarrage de la réalisation (conception, développement)
- 18/04 : livraison d’une première version (premier dépôt ECF)
- 22/05 : livraisons des applicatifs et mise en production

Gestion du projet

La conduite de projet est réalisée de façon « classique », en cascade.

Nous avons une livraison unique et pas de retours de la part du client (fictif). Une méthodologie agile ne peut pas être envisagée de ce fait. Une méthodologie en V demande beaucoup de ressource et permet d'aboutir à un produit de haute qualité. Nous n'avons pas d'impératif fort de qualité (en dehors des critères de sécurité) puisque le projet consiste à réaliser une solution de départ, minimaliste.

Méthodologie en cascade est adaptée car elle nous permet de définir les temps de travail tout en restant focalisé sur des dates de livraison inamovibles.

Processus de suivi et de reporting

Le suivi et le reporting seront notés dans le Kanboard mis en place et dans le fichier de planning GantProject.

La planification est réalisée avec GantProject - https://www.ganttproject.biz/
Les fichiers GantProject seront intégrés in fine au dépôt GitHub avec le code source. Ce choix est possible car il n'y a pas de collaboration sur le projet.

La gestion des taches est réalisée avec Kanboard - https://kanboard.org/
Éventuellement, une instance du logiciel sera accessible en ligne en consultation. Les fichiers seront également livrés dans le dépôt du projet.

Risques et mesures d'atténuation

Identification et évaluations des risques

Applications mobiles et bureautique

Les temps de réalisation et la qualité des livrables pour l'application mobile et bureautique sont sujet à des dépassements car ces technologies ne sont pas bien maitrisées par le développeur. Il y a un risque de dépassement de temps important.

Ce risque est jugé très important, sa probabilité de survenue élevée et la criticité élevée.

→ Pour limiter ce risque, il à été convenu d'utiliser les technologies les plus communément utilisées et notamment celle présentées dans les cours Studi. Pour s'assurer des bons choix, une étape de la conception sera spécifiquement dédiée à cette tâche.

→ La réalisation le plus tôt possible pour garder de la marge de manœuvre est à mettre en place

Le suivi du temps sera assuré en dédiant une phase de développement spécifiquement à la réalisation des applications mobiles et bureautique et en la découpant si cela est possible.

Sécurité

Les hôpitaux sont des cibles particulièrement prisées des hackeurs.

Ce risque est jugé important (sa probabilité de survenue est faible mais son impact majeur).

→ Pour limiter ce risque :

- la prise en compte de la sécurité sera explicitement intégrée à toutes les phases du projet.
- Le développeur intègrera les contraintes de sécurité dans chacune de ses réalisations sans délai, dès l'écriture du code et la configuration des outils.

Une partie de la phases de tests sera dédiée aux tests de sécurité.

Gestion de projet

Le gestionnaire de projet n'ayant jamais réalisé de gestion complète de projet, les risques d'évaluation et de planification peuvent exister.

Pour limiter ce risque, des marges de délais seront intégrées aux estimation. On se donnera la possibilité de revoir les choix technique si ils sont trop ambitieux ; tout en respectant les impératifs du projet.

Risques stratégiques

Les risques stratégiques sont faibles car les investissements financier et temporels sont relativement faibles.

Notes complémentaires

La définition des ressources, parties prenantes et rôles ne nécessite pas de détails, le client est fictif et seul le développeur/chef est impliqué.

La budgétisation n'a pas été effectué, le nombre de jours/homme sera indiqué après réalisation du planning.

Le cahier des charge est un document contractuel.

Il a été validé par les parties prenantes (développeur/chef de projet et l'hôpital client).

Conception technique

Document

Version	1.0
Date de modification	22/03/24

Rappel des phases

1. Pré-démarrage
2. Cadrage
3. Spécifications fonctionnelles, planification & chiffrage
4. **Conception technique**
5. Réalisation technique
6. Tests et correctifs
7. Mise en production, formation, maintenance

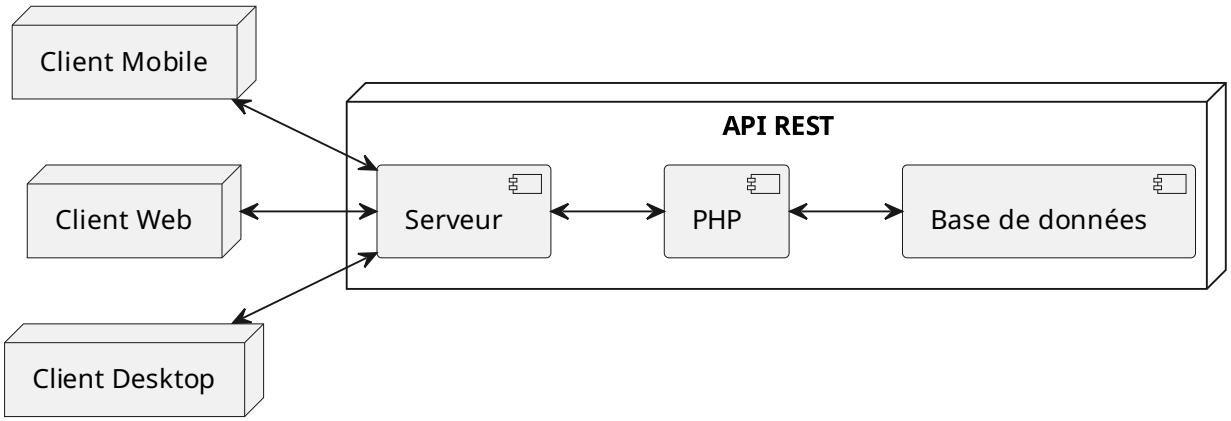
Table des matières

Architecture..... 106
Composants..... 106
API REST..... 106
Client mobile..... 106
Client web..... 106
Client desktop..... 106
API..... 107
Modèle conceptuel de données..... 107
Classes..... 107
Diagramme de classes..... 107
Endpoints..... 109
Droits..... 109
Rôle Docteur..... 109
Patient..... 109
Secrétaire..... 109
Admin..... 109
Tests de l'API..... 110
Design..... 110
Wireframes..... 110
Application bureautique..... 110
Identification des risques de sécurité..... 111
Environnement de développement..... 111
Machine de développement..... 111
Scripts de validation / hook pré-commit..... 112
Mise en production..... 113
Diagramme de déploiement..... 113

Architecture

Le système étant réparti sur plusieurs dispositifs, une architecture avec différents services organisés autour du service de stockage des données est adapté.

On aura donc l'architecture suivante.



Composants

API REST

La communication des clients avec l'API REST est assurée par un serveur web (technologie à déterminer)

L'API sera construite avec API Platfom en utilisant Symfony en dernière version disponible (7).

Client mobile

Le client mobile sera réalisé avec Flutter qui doit permettre un développement à la fois rapide et de bonne qualité.

Le client sera une simple webview (navigateur internet interne) qui se utilisera le client web.

Client web

Le client web sera un site réalisé avec Symfony, de façon indépendante du composant API pour permettre une évolutivité découplée des deux composants.

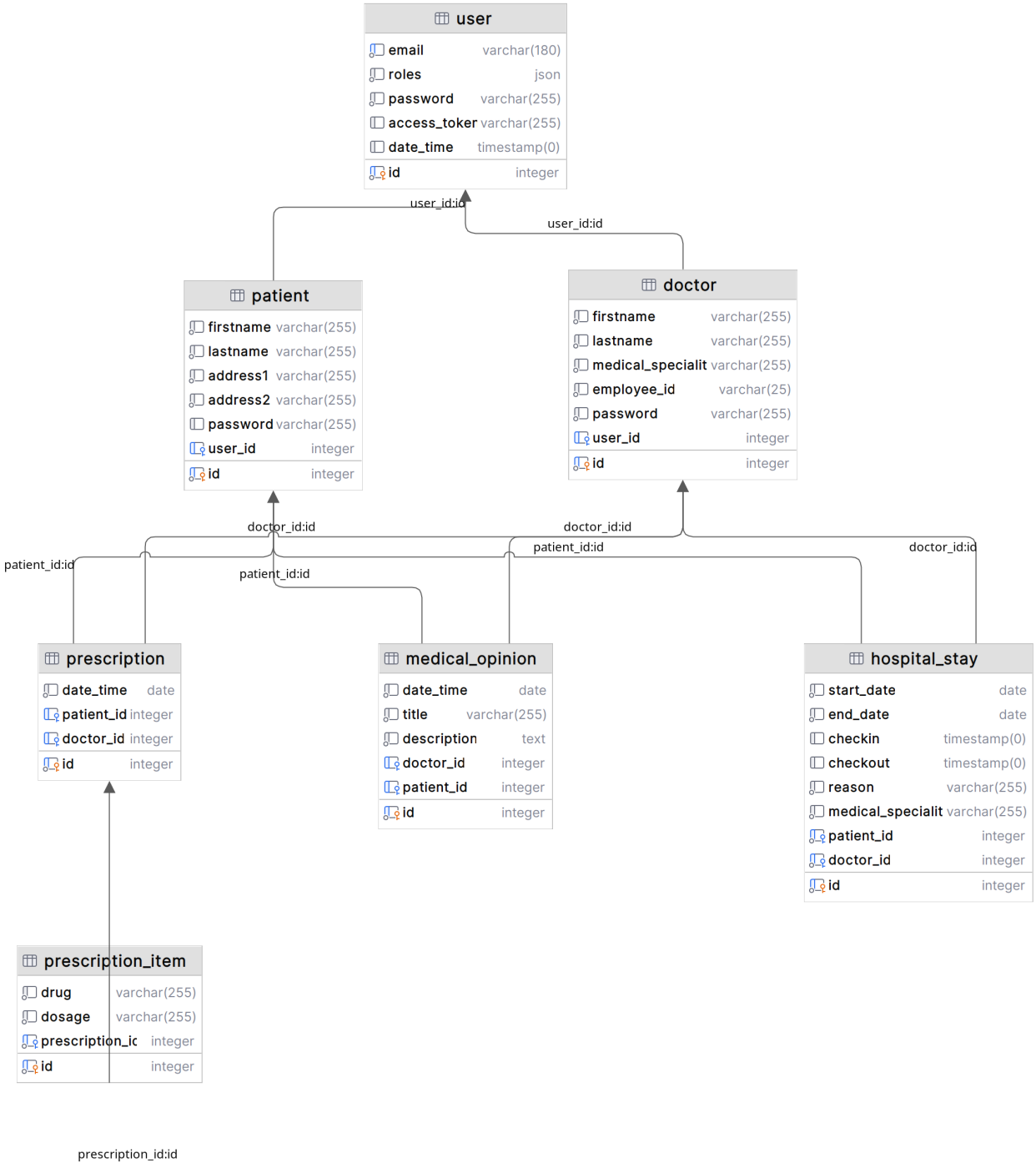
Le coté front sera réalisé avec Bootstrap pour rester dans les standards solides et répandus.

Client desktop

Le client desktop sera développé avec electronjs. Comme le client mobile, il s'agira d'une simple webview.

API

Modèle conceptuel de données



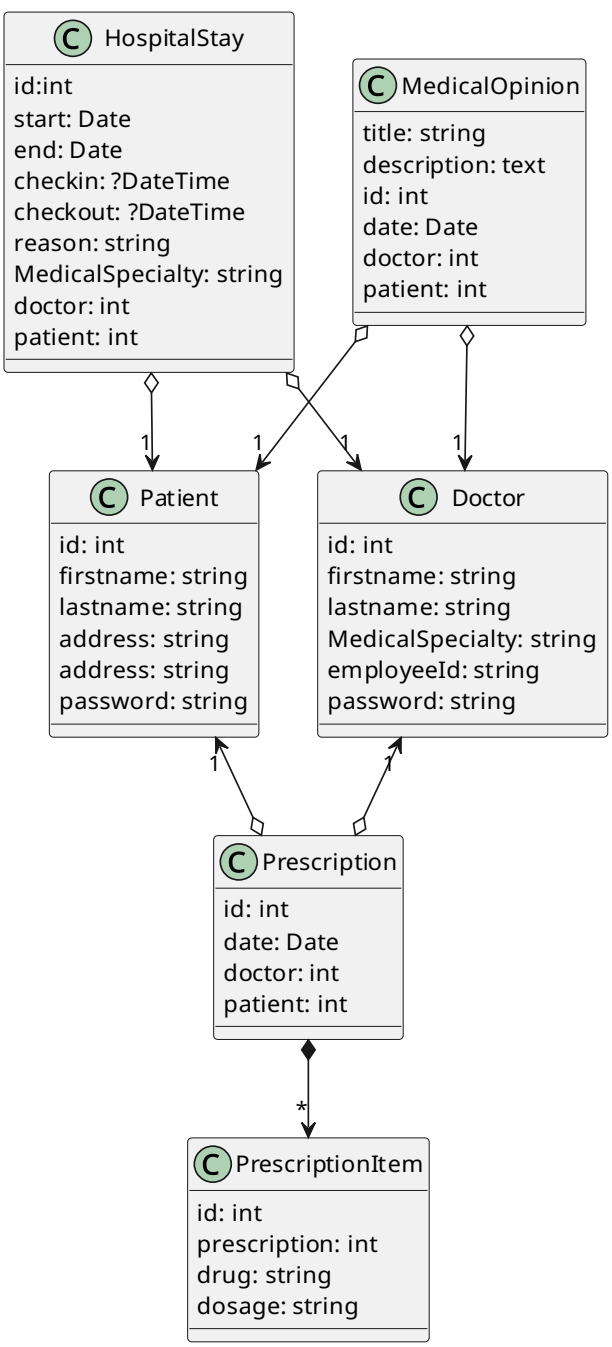
Classes

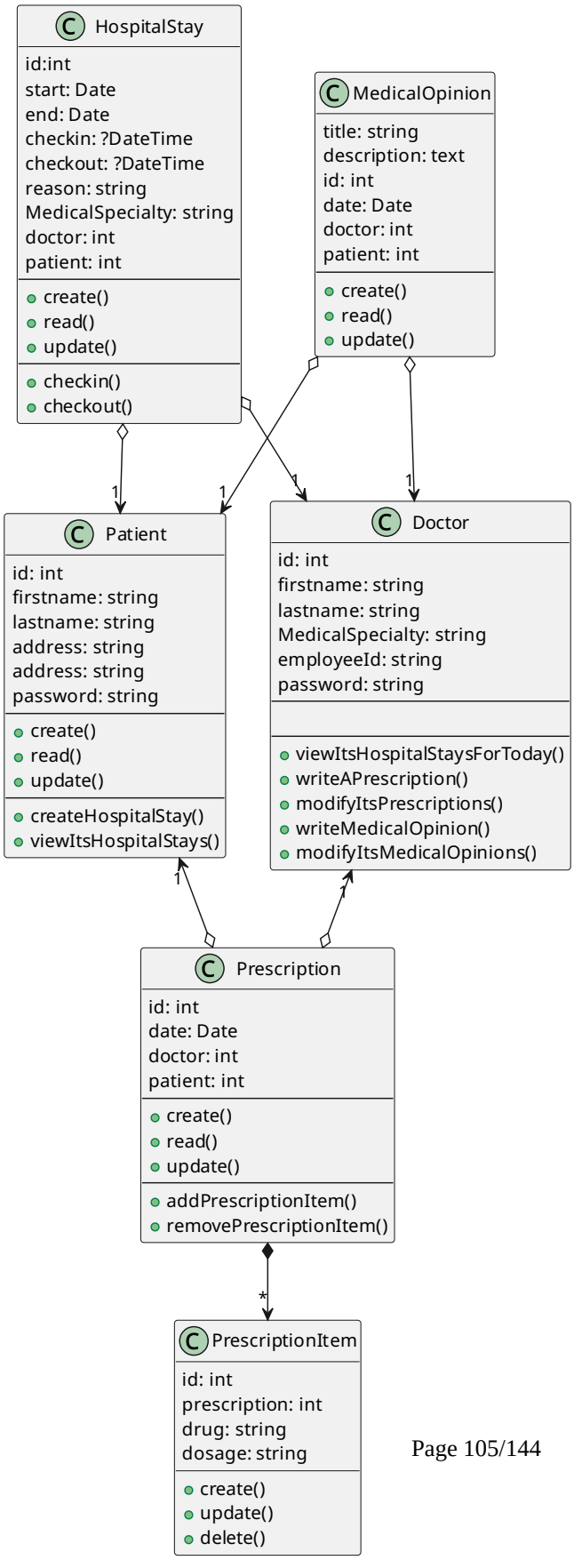
Diagramme de classes

Les deux diagrammes représentent strictement la même chose, une vue avec les agrégations/compositions, une vue avec les cardinalités.

La visibilité des propriétés n’est pas représentée : toutes les propriétés sont privées.
Toutes les propriétés possèdent des getters et setters.
Ce design répond aux besoins de l’ORM Doctrine.

Pour la compréhension, on peut faire une diagramme avec les méthodes correspondantes aux cas d’utilisation. Cependant, on n’implémentera pas ces méthodes dans les objets car c’est Api Platform qui va se charger des opérations de crud. On aura par contre des endpoints qui reflètent ces opérations.





Endpoints

L’ensemble des endpoints est disponible à l’URL de l’API, on livrera un document json et/ou html qui listera les points d’entrées.

Cette documentation est générée par api platform en temps réel, elle est donc toujours à jour.

L’ensemble des endpoints se retrouve dans les droits décrits ci-dessous.

Droits

Ces droits d’accès aux api dépendent des cas d’usages décrits dans les spécifications fonctionnelles.

Rôle Docteur

- Prescription
  - création (une par jour, par patient)
  - modification (prescription du jour)
- Prescription Items
  - création
- Avis
  - création (une par jour, par patient)
  - modification (le jour même)
  - suppression (non implémenté)
- Séjours
  - lister (ceux du jour, pour ses patients)

Patient

- Utilisateur
  - création (création compte + entité)
- Séjour
  - lister (ses séjours uniquement)
  - création

Secrétaire

- Séjours
  - lister entrées
  - lister sorties
  - modifier
- Patient
  - voir détails
- Prescription
  - lister
  - voir détails
- Avis
  - lister
  - voir détails

Admin

- Docteur
  - lister
  - création
  - modification
- Séjour
  - lister

- modification

Tests de l’API

L’API est testable par l’admin offerte par api plateform directement sur le serveur de d’api.

Éventuellement, l’utilisation du client desktop httpie est possible, quelques requêtes d’exemples sont livrées avec le projet.

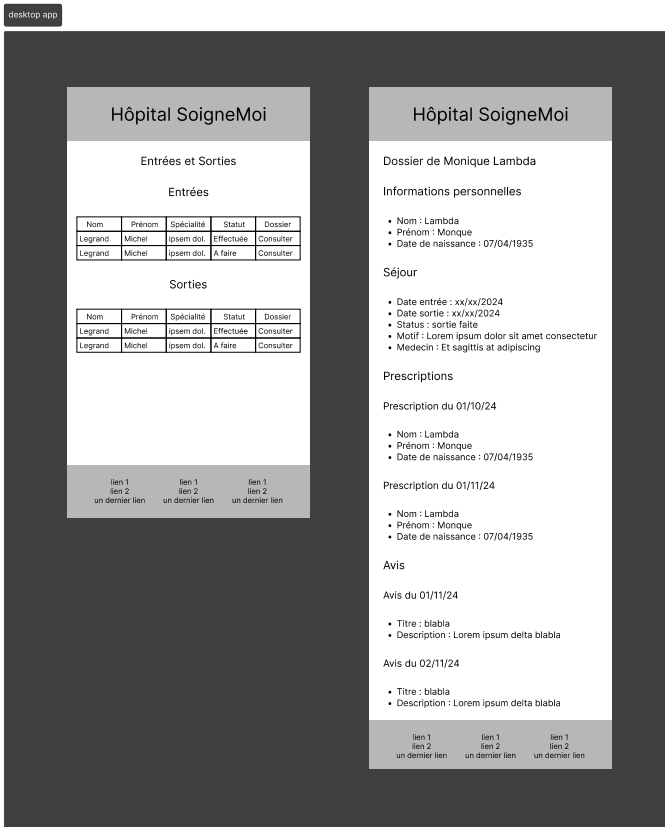
Design

Wireframes

Les fichiers sont disponibles dans wireframes sont dans *livrables/documents/design.*

Application bureautique

Les rendus proposés ici sont au format mobile, les informations seront strictement les même sur un affichage plus large de type ordinateur.



Cette vue concerne les utilisateurs connecté avec le rôle de secrétaire. Le premier écran liste les entrées et le sorties que le/la secrétaire devra traiter à la date actuelle.
Le libellé « A faire » sera un lien permettant d’enregistrer une entrée (*checkin)* pour les patients qui n’ont pas encore effectué leur entrée.
Le libellé « A faire » sera un lien permettant d’enregistrer une sortie (*checkout)* pour les patients qui

ont pas encore effectué leur entrée et dont la sortie est prévue ce jour.

La seconde vue est accessible à partir du lien « consulter » de la première vue.
On trouve sur cette page :

- le prénom et le nom du patient en titre
- une section informations personnelles avec le nom, le prénom et la date de naissance du patient au format *jour/mois/année*.
- Les données du séjour, date d'entrée, etc
- La liste des prescriptions, par ordre chronologique, pour chaque prescription, la date et la liste des médicaments prescrits avec leur posologie.
- La liste des avis médicaux, par ordre chronologique, avec le titre et la description.

A compléter

Identification des risques de sécurité

Avec les éléments techniques précédents on note plusieurs niveaux auxquels les risques peuvent survenir.

- Fuite de données sur les outils et services de gestion de code et de déploiement (code source, service de déploiement, hébergement, gestion de bugs, etc)
- Vulnérabilités sur les machines d'hébergement, si on administre le serveur, il faudra le mettre à jour
- Vulnérabilités des composants logiciels, contenaires *Docker* (contenaires et leurs configurations)
- Vulnérabilités des composants de code (packages composer, librairies JavaScript, etc)
- Vulnérabilités liées aux mauvaises pratiques de développement, code vulnérables
- Vulnérabilités liées à une mauvaise conception, architecture.

Il faudra adresser des réponses et mesures pour chaque points.
On pourra réaliser un document de sécurité qui traites l'état et les mesures à mettre en place pour livrer et maintenir un système applicatif sans vulnérabilités.

Environnement de développement

Machine de développement

Le développement est réalisé via *Docker*. On assure ainsi un environnement de développement et de déploiement identique (même versions, même configurations de serveur (nginx, PHP, PostgreSQL, …).

Les pré-requis sur la machine de développement sont donc *Docker* et optionnellement *Just task runner* pour simplifier les commandes dans le container.

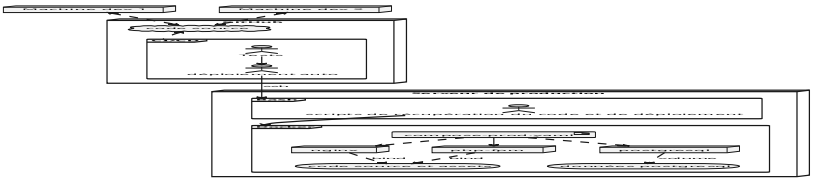
Scripts de validation / hook pré-commit

La démarche qualité est assurée via un ensemble de scripts qui lancent les tests, la validation et le style du code. Ces scripts sont référencés dans les scripts *composer*.

Ces scripts doivent être lancé avant tout commit, manuellement ou via un hook pré-commit. L'installation de ce hook pré-commit est possible par la tache *install-pre-commit-hook* (lancer *just install-pre-commit-hook* dans un terminal, dans le dossier du projet).

Mise en production

Diagramme de déploiement



Document de sécurité

Sécurité

Rapport rédigé avant la mise en production de l’application.

Ce document présente une analyse des actifs, leurs états, les solutions, les mesures prises et comment assurer la maintenance.

Il doit servir de référence pour le suivi et l’élaboration de rapports.

Table des matières

Vue d’ensemble....118
Objectifs....118
Méthodologie....118
Structure du document....118
Liste des actifs....118
Vulnérabilités OWASP....119
Contrôle d’accès incorrect....119
Divulgation de données / faiblesse cryptographiques....119
Injections....119
Construction non sécurisée....119
Mauvaise configuration de sécurité....119
Composants vulnérables et périmés....120
Contrôle d’authentification incorrect....120
Intégrité logiciel....120
Erreurs et faiblesses d’enregistrement des logs....120
Tableau actifs / vulnérabilités et faiblesses....121
Récapitulatif de la veille....122
Détails des actifs....123
Machines du développeur....123
Vulnérabilités....123
Faiblesse cryptographique / accès aux données....123
Intégrité logiciel....123
Enregistrement des logs....123
Veille....123
Hébergement....123
Infrastructure et machines virtuelle....123
Conservation d’une clé ssh publique....124
VPS Ubuntu....124
Contrôle d’accès incorrect....124
Faiblesse cryptographique....124
Mauvaise configuration de sécurité....124
Intégrité logiciel....124
Logs....124
Composants vulnérables....124
Docker....125

Faiblesse cryptographique....125
Mauvaise configuration de sécurité....125
Intégrité logiciel....125
Logs....125
Composants vulnérables....125
Conteneurs Docker....126
Faiblesse cryptographique....126
Contenaire php....126
Contenaire nginx....126
Contenaire certbot....126
Contenaire postgreSQL....126
Mauvaise configuration de sécurité....126
Configuration php....126
Configuration nginx....126
Configuration postgres....127
Composants vulnérables....127
Politique....127
Solution....127
Application....127
Mesure de suivie....127
Intégrité logiciel....128
Logs....128
Composer....128
Dépendances Composer et Symfony....128
Dépendances Javascript....129
Code....129
Certificats TLS....130
Contexte....130
Exigences....130
Solutions....130
Mesures....130
Noms de domaine....130
Dépôt de code....130
Action GitHub *ssh-remote-commands* (déploiement)....131
Autres actions GitHub....132
Accès données privées (identifiants)....132
Accès par le web / nginx....132

Données de sessions et cookies....132
Configurations....133
Application Symfony....133
Configuration Docker....133
Surveillances à appliquer....134
Améliorations....134
Annexes....135
1. Vérification du type de connexion ssh aux Droplet....135
2. Mise en place des mises à jour Ubuntu....135
3. Analyse des images docker....135
Image php-fpm personnalisée....135
Image nginx (serveur web)....136
PostgreSQL (base de données)....136
certbot/certbot (outils de création des certificats TLS)....136
4. Renouvellement des certificats TLS....136

Vue d’ensemble

Objectifs

Ce document présente une analyse de sécurité des applications, serveur api et client web. Il est établi avec les objectifs suivants :

- faire un état de la sécurité de l’application
- corriger les vulnérabilités trouvées
- indiquer comment réaliser la veille et l’automatiser
- servir de référence pour établir un prochain rapport

Les grandes lignes des risques des sécurité ont été définies dans le document *4. Conception technique.odt*.

Méthodologie

J’ai cadré les points à étudier en me référant d’une part aux actifs et d’autre part au top 10 de l’owasp, organisme qui fait référence en la matière.

J’ai réalisé ce document en plusieurs étapes :

- lister les actifs des applications
- déterminer quelles vulnérabilités du top 10 de l’owasp concernent les actifs
- étudier les vulnérabilités pour chaque actif

Structure du document

Pour faciliter l’exploitation du document, pour une maintenance efficace et simplifiée, le document est structuré autour des actifs. Les vulnérabilités owasp sont présentées dans un premier temps. Pendant ce temps, on détermine, pour chaque actif, les vulnérabilités concernés.

Les vulnérabilités de type Composants vulnérables et périmés (*Vulnerable_and_Outdated_Components*) constituent l’essentiel du travail réalisé.

Liste des actifs

- Machines du développeur
- Hébergement
- VPS Ubuntu
- Docker
- Conteneurs Docker
- Composer
- Dépendances Composer et Symfony
- Dépendances Javascript

- Code
- Certificats TLS
- Noms de domaine
- Dépôt de code
- Action GitHub ssh-remote-commands (déploiement)
- Autres actions GitHub

Vulnérabilités OWASP

Contrôle d’accès incorrect

Dans nos applications, les vulnérabilités de type *contrôle d’accès incorrect* (*Broken Access Control*) peuvent être liés à une mauvaise configuration ou à des défaillances internes au composant. Concernant la configuration, elle est stable, évolue avec le code et fait l’objet de tests intégrés dans la chaine d’intégration continue, il n’y a pas de veille à faire. Concernant les vulnérabilités liées aux composants Symfony et ApiPlateform, ils y a une veille à faire, elle est faite par la veille sur les dépendances Composer et Symfony.

Divulgation de données / faiblesse cryptographiques

Cette vulnérabilité, *Cryptographic Failures*, est traitée dans deux sections de mon rapport, *données privées (identifiants)* et *Certificats TLS*.

Injections

Les injections (*SQL* et *XSS*), (*Injection*) sont traitées par le respect des bonnes pratiques de codage et ne font pas l’objet d’une veille. Les injections pourraient éventuellement survenir via des vulnérabililtés directement dans les packages, ce point est traitée par la surveillances sur les packages composer et les packages javascript.

Construction non sécurisée

Les vulnérabilités du groupe *Insecure_Design* font référence à l’architecture du code. J’ai utilisé des patterns répandus, les applications sont conçues en couches séparées, connectées au minimum. Les couches sont connectées et uniquement aux couches adjacentes. Au niveau des liens entre les applications, j’ai utilisé une architecture client-serveur tout à fait standard avec une authentification par *token* par des échanges chiffrés (TLS) uniquement. Je note cependant une amélioration, il aurait été possible d’utiliser le composant (*bundle*) *LexikJWTAuthenticationBundle* pour se reposer sur un composant publique et recommandé par *Symfony* (donc jugé fiable).

Ce type de vulnérabilité concerne l’architecture de infrastructure et celle du code, il n’est pas utile de mentionner ce point dans les tableaux récapitulatifs

Mauvaise configuration de sécurité

Les vulnérabilités Security misconfiguration concerne les contenaires Docker, l’application Symfony et les bundles Docker. L’étude à permis de voir qu’à l’exception du contenaire PHP-fpm,

tous sont sécurisés par défaut. J'ai cependant consulter les documentations et vérifier les configurations.

Composants vulnérables et périmés

Ce groupe de vulnérabilité, Vulnerable and Outdated Components, est celui sur lequel j'ai porté la plus partie de mon attention. Les applications réalisées font finalement appel à peu de code réalisé par mes soins, a beaucoup de composants, directement et indirectement, pour le code, pour les services, le déploiement et les serveurs.

J'ai listé l'ensemble des actifs, vérifié leur sécurité, procédé à des mises à jour, documenter comment déterminer les besoins de mise à jour, mise en place des vérification de sécurité automatisée, priorisé les niveaux de vigilance et défini des pistes d'amélioration.

Contrôle d'authentification incorrect

Cette catégorie, *Identification_and_Authentication_Failures* ne nécessite pas de veille mais une mise en place correcte.

Seul l'actif Code est concerné, ce groupe de vulnérabilités ne figurera pas dans les tableaux.

Dans la liste des vulnérabilités, deux points ne sont pas satisfaits, la protection contre les attaques *brute force* (cette amélioration est suggérée) et la vérification que les mots de passe ne sont pas dans le top 10 000 des mots de passe les plus utilisés.

Cependant au niveau logiciel, seul les mots de passe forts sont autorisés (contrainte PasswordStrength sur le champ $userCreationPassword dans l'entité *Patient*).

Intégrité logiciel

Cette catégorie fait référence à l'utilisation non contrôlée de logiciel et composants (services Docker, packages composer, actions GitHub). Les vulnérabilités Software_and_Data_Integrity_Failures sont traitées dans nos applications de deux façon, en faisant une veille sur ces composants et utilisant des versions strictement définies.

L'intégrité des composants logiciel est mise en place partout par l'utilisation de références précises uniquement.

Avec composer, le fichier composer.json référence les versions exactes de tous les packages utilisé, on s'assure d'utiliser strictement la même version que lors du développement et on évite d'obtenir une version inattendue (non contrôlée ou éventuellement corrompue).

Pour les actions GitHub, on a aussi utilisé les *hash* pour obtenir les versions exactes et éviter les problèmes évoqués juste avant.

Erreurs et faiblesses d'enregistrement des logs

Security_Logging_and_Monitoring_Failures fait référence à 4 faiblesses. Il s'agit de loguer assez d'informations pour débusquer des attaques tout en ne divulguant pas d'informations sensibles. Je n'ai pas traité ce point peut important dans notre contexte. J'ai cependant ajout ces faiblesses à la liste des améliorations à faire.

Le dernier item du top 10 concerne la récupération de contenus distants. Une url est passé à notre application et des données récupérées à cette adresse. Notre application ne réalise pas ce type de taches, nous ne sommes pas concernés.

Tableau actifs / vulnérabilités et faiblesses

Ce tableau indique si il y a une analyse à faire de vulnérabilité/faiblesse (colonne) à faire pour les actifs (lignes). *Oui* indique qu'un travail est à faire *Non* indique qu'on va ignorer ce cas après l'avoir justifié. *n/a* indique l'actif n'est pas concerné.

	Contrôle d'accès incorrect	faiblesse cryptographiques	Injections	Mauvaise configuration de sécurité	Composants vulnérables et périmés	Intégrité logiciel	enregistrement des logs
Machines du développeur	n/a	Non	n/a	n/a	n/a	Non	Non
Hébergement	n/a	Non	n/a	n/a	n/a	n/a	n/a
VPS Ubuntu	Oui	Non	n/a	Non	Oui	Non	Non
Docker	n/a	Oui	n/a	Non	Oui	Non	Oui
Conteneurs Docker	n/a	n/a	n/a	Oui	Oui	Oui	Oui
Composer	n/a	Non	n/a	n/a	Oui	Non	n/a
Dépendances Composer et Symfony	n/a	Oui	n/a	Oui	Oui	Non	n/a
Dépendances JavaScript	n/a	Non	n/a	Non	Oui	Oui	n/a
Code	n/a	Non	Oui	n/a	n/a	Non	n/a
Certificats TLS	n/a	Oui	n/a	Oui	n/a	n/a	n/a
Noms de domaine	n/a	n/a	n/a	Non	n/a	n/a	n/a
Dépôt de code	Oui	Non	n/a	Oui	Non	Non	n/a
Action GitHub ssh-remote-commands (déploiement)	n/a	n/a	n/a	n/a	Oui	Oui	n/a

Récapitulatif de la veille

Ce tableau récapitule les actifs, le mécanisme de mise à jour, l'état de sécurité, le mode déclenchement des mises à jour et finalement la vigilance à porter sur l'actif et ses processus.

Le déclenchement automatique indique que la mise à jour se fait de façon autonome, aucune action n'est requise. Le déclenchement semi automatique indique que le processus est partiellement automatisé et qu'une intervention est donc nécessaire.

Les états de vigilance reflètent en partie le statut de déclenchement et d'autre part le besoin d'intervention. 3 niveaux de vigilances sont définis : basse, moyenne forte.

Actif	Mécanisme	État	Déclenchement	Vigilance
Machines du développeur	Mise à jour de l'OS	Actif	Semi auto	basse
Hébergement	Identification via GitHub	Actif	Manuel	basse
VPS Ubuntu	Mise à jour automatiques Ubuntu Connexion par clé ssh uniquement. (*unattended upgrades*)	Actif	Auto	basse
Docker	Mise à jour automatiques Ubuntu (*unattended upgrades*)	Actif	Auto	basse
Conteneurs Docker	Analyse par *Docker Scout* via l'intégration continue sur GitHub	Actif	Semi auto	forte
Certificats TLS	Mise à jour régulière par cron	Actif	Auto	basse
Noms de domaine	Alerte email de du *registrar*	Actif	Semi auto	basse
Composer	Intégration continue GitHub et hook sur modification de code local	Actif	Semi auto	forte
Dépendances Composer et Symfony	Intégration continue GitHub et hook sur modification de code local	Actif	Semi auto	moyenne
Dépendances JavaScript	Intégration continue GitHub et hook sur modification de code local	Actif	Semi auto	forte
Code	Intégration continue GitHub, modification de code local et vérifications manuelles	Actif	Semi auto	moyenne
Dépôt de code	Identification double facteur GitHub et identification par clé ssh	Actif	Manuel	basse
Action GitHub ssh-remote-commands (déploiement)	Identification par clé ssh	Actif	Manuel	moyenne
Autres actions GitHub	Utilisation sécurisée, surveillance déléguée à GitHub	Actif	Manuel	moyenne
Accès données privées (identifiants)	Mise en place initiale sécurisée	Actif	Manuel	basse
Configurations	Mise en place initiale sécurisée	Actif	Manuel	basse

Détails des actifs

Machines du développeur

Le développeur dispose de deux machines.

L'ordinateur portable sous *Manjaro Linux* est mis à jour à chaque utilisation et l'ordinateur de bureau, sous *Fedora,* est également maintenu à jour à chaque utilisation. Ces machines n'utilisent aucun package ou programme ne provenant pas de source sure directement.

Vulnérabilités

Faiblesse cryptographique / accès aux données

Ces machines sont importantes car elles contiennent des clés ssh permettant :

- la publication du code sur GitHub (possibilité de compromettre le code)
- le lancement des déploiements (via la publication de code)
- l'accès aux instances hébergeant l'api et le client web.

Étant donnée qu'aucun partage n'est mis en place, qu'aucun package non sur est installé et que les machines sont mise à jour on considère que l'accès aux données est sécurisée.

Intégrité logiciel

Les machines sont à jour via les systèmes natifs (dnf et pacman) qui assurent l'intégrité des logiciels.

Enregistrement des logs

Les machines journalises les actions système, il n'y a pas de travail à effectuer à ce niveau, je fait confiance aux OS.

Veille

La résolution des problèmes de sécurité est assuré par les mises à jour du système et des logiciel. Il faut mettre à jour les machines à chaque utilisation.

Hébergement

L'api et le client web sont déployés chez l'hebergeur *Digital Ocean.*

L'hébergeur est réputé et jugé fiable, nous lui attribuons notre confiance.

Infrastructure et machines virtuelle

Le superviseur qui génère notre serveur virtuel est entièrement contrôlés par *Digital Ocean*. Aucune mesure de sécurité n'est à prendre de notre coté.

Conservation d’une clé ssh publique

Nous avons ajouté une clé ssh publique à notre compte utilisateur. Il s’agit d’une clé publique, sa fuite n’est pas problématique. La modification de cette clé n’est pas possible.

L’accès au compte client digital ocean est assuré par un mot de passe fort, non partagé et effectué via une autorisation de GitHub (réalisée par une authentification à double facteur).

Il n’y a pas de travail particulier à réaliser pour s’assurer de valider les faiblesses de type **faiblesse cryptographique / accès aux données.**

VPS Ubuntu

L’api et le client web sont déployés via une action *GitHub* sur des *Droplet Digital Ocean* en utilisant *Docker*. La distribution utilisée est Ubuntu 22.04 LTS (Jammy Jellyfish)

Contrôle d’accès incorrect

Cette clé ssh publique a été ajoutée aux instances Ubuntu pour nous permettre une connexion ssh (création du vps par Digital Ocean). Nous utilisons ainsi une connexion uniquement par ssh, sans de mot de passe ce qui limite fortement les possibilités de connexion distante frauduleuses.

Il a été vérifié le 24/05/2024 que la connexion ssh via mot de passe n’est pas possible (voir annexe 1. Vérification du type de connexion ssh aux Droplet).

Ce point est donc validé.

Faiblesse cryptographique

Aucune données de la machine n’est accessible directement, aucun partage n’a été mise en place.

Ce point est validé.

Mauvaise configuration de sécurité

Aucune modification ou reparamétrage de sécurité n’a été effectué, on considère ce point valide car on peut faire confiance à Digital Ocean et Canonical (éditeur d’Ubuntu) pour livrer des OS sécurisé par défaut.

Intégrité logiciel

L’installation et la mise des logiciels est assuré par le gestionnaire de package *apt*. Aucune modification de configuration n’a été effectuée (pas d’ajout de dépôt ou autre modification). A nouveau, on délègue ces vérifications à Ubuntu/Canonical.

Logs

Comme précédemment, nous déléguons la responsabilité de la bonne configuration.

Composants vulnérables

Le système d’exploitation est composé de nombreux paquets qui peuvent présenter des vulnérabilités. Il est important que ce système soit maintenu à jour. La version d’*Ubuntu* utilisée est en cours de maintenance, elle est couverte par le support *LTS* jusqu’en avril 2027 (source).

Le système de mise à jour de sécurité automatique est activé et son fonctionnement à été attesté. (source)

La configuration et la vérification des mises à jour de sécurités de l'OS et des paquets (logiciels) à été effectuée (voir 2. Mise en place des mises à jour Ubuntu)

Nous pouvons faire confiance à ce système de mise à jour de l'OS.

On devra vérifier que le système de mise à jour automatisé est toujours opérationnel de temps en temps.

Nous n'avons pas mis en place l'envoi de mail lié aux actions de ces mises à jour afin de simplifier l'administration du serveur. C'est une amélioration possible.

Sur cet OS, en dehors des composants liés aux mises à jour, nous utilisons une tache cron pour générer les certificats TLS (Voir Conteneurs Docker) et Docker (section suivante).

Docker

Docker est installé sur la machine Ubuntu, c'est la fondation de nos applications.

Faiblesse cryptographique

Les seules données sensibles de docker sont le fichier .env.local et le fichier de socket du *daemon*. Le fichier de socket est celui par défaut, il ne présente donc pas de risque d'accès (on délègue cette responsabilité au système Ubuntu). Le fichier d'environnement est placé dans le dossier /app. Docker n'offre pas d'accès à ce fichier. Une conteneur pourrait le faire, c'est discuté dans la section des contenaires (Voir Conteneurs Docker)).

Mauvaise configuration de sécurité

Aucune modification n'a été effectué, aucun privilège supplémentaire n'est exigé. Docker en lui même ne présente pas de possibilités à ce niveau.

Intégrité logiciel

Les paquets liés à Docker sont gérés par le l'OS (Voir VPS Ubuntu).

Logs

La configuration par défaut à été laissée, Docker enregistre les logs par défaut. On note qu'il faudra cependant contrôler précisément les logs des contenaires.

Composants vulnérables

Aucune extensions ou plugin n'a été ajouté, il n'y a pas de risque à ce niveau.

Pour simplifier la maintenance et clairement délimiter les responsabilités, je n'ai pas considéré les containers comme des composants. Chaque contenaire est considéré comme un actif et étudié dans ce document (voir Conteneurs Docker))

Conteneurs Docker

Nous avons plusieurs containers déployés dans le serveur api et le client web.

- Contenaire php personnalisé
- contenaire nginx
- contenaire cerbot
- contenaire postgresql (api uniquement)

Faiblesse cryptographique

Contenaire php

Le contenaire php est solicité via le contenaire nginx. La configuration nginx dirige les requetes vers le fichier index.php uniquement, aucune autre entrée n'est possible. La configuration de ce contenaire est donc sécurisé. On note cependant la nécessité de vérifier les faiblesses d'accès au niveaux de *Symfony* qui traite ces requêtes.

Contenaire nginx

Ce contenaire redirige les requêtes de fichiers non existants vers php (traité au dessus) et les fichiers existants sont servi. La racine web est configuré sur */app/public*, on a donc pas de vulnérabilité directe, aucun fichier système, de configuration docker, certficats, etc n'est accessible. Il faudra vérifier au niveau de l'application Symfony qu'il n'y a pas de vulnérabilté à ce niveau.

Contenaire certbot

Le contenaire cerbot, destiné à la génération des certificats SSL/TLS est en fonctionnement que lors de la génération des certificats, une fois tous les 90 jours. Il n'expose aucune données publiquement.

Contenaire postgreSQL

Ce contenaire n'expose aucune données, il est accessible uniquement par le contenaire php, ses accès sont donc controllés par ce contenaire via Symfony (voir la section dédiée). Les données sont stockées dans le système hote Ubuntu, la sécurité de ces données est gérée par l'OS (voir VPS Ubuntu)

Mauvaise configuration de sécurité

Configuration php

Nous utilisons l'image Docker officielle de *php-fpm* pour construire notre propre image. Nous avons tenu compte des recommandations officielles de la page docker en activant la configuration de production fournie lors de la construction de l'image Docker :

```
RUN mv  «$PHP_INI_DIR/php.ini-production »  «$PHP_INI_DIR/php.ini »
```

Configuration nginx

La configuration par défaut de *nginx* est sécurisée. Aucune action, autre que documentaire, n'a été entreprise.

On surveillera les logs d'erreurs pour y détecter des attaques et prendre les mesures en temps voulu.

Configuration postgres

La configuration de *postgres* ne demande pas d'attention particulière.

Il existe une option de restriction des accès par adresse ip mais le contenaire *postgres* n'est pas accessible de l'extérieur (web), uniquement par le contenaire php (architecture en couche, classique), cette mesure ne présente pas d'intérêt.

Il n'y a pas d'accès avec mot de passe par défaut. Le mot de passe de l'utilisateur est défini à la construction du contenaire par une variable définie dans le fichier *.env.local* qui n'est ni versionné ni accessible par le web, (ni en variable d'environnement).

Composants vulnérables

Politique

Les images employées peuvent être sujettes à des problèmes de sécurité. Certaines images employées ne sont pas complètement exemptes de vulnérabilités, cependant, après contrôle, on peut en tolérer certaines. On ne tolèrera pas de vulnérabilité critique sur les contenaires. Les vulnérabilités sévères feront l'objet d'un traitement le plus rapidement possible. Les vulnérabilités moyennes seront étudiées pour voir si nos machines sont concernées. Les vulnérabilités inférieures seront généralement ignorées.

Solution

La solution retenue est la solution la plus directe, fournie par le créateur de Docker : le service d'analyse de vulnérabilités *Docker Scout*.

La solution *Trivy* à l'avantage de pouvoir être utilisée plusieurs contexte et indépendamment de GitHub mais n'a pas été retenue car elle remonte trop de faux positifs.

Nous utilisons des images recommandées et vérifiées par *Docker* et les distributeurs (*php*, *nginx*, *postgres*, *certbot*).

Notre image php-fpm de développement est agrémentée d'utilitaires d'analyse et de formatage qui ne sont pas présents sur l'image utilisée en production (https://github.com/SebSept/docker-php-symfony-starter/blob/prod/Dockerfile)

Application

Lors de la mise en place du dispositif, des vulnérabilités ont été trouvées et corrigées (voir 3. Analyse des images Docker)

Mesure de suivie

L'analyse des vulnérabilités est automatisée par une action GitHub lancée quotidiennement, sur les actions de *pull request* et sur l'envoi de code dans la branche principale (*main*).

L'action *GitHub* présente la limitation de ne pas renvoyer de code d'échec (exit code différent de 0), il convient donc de vérifier manuellement le résultat des actions GitHub. Ces résultats sont indiqués dans des commentaires réalisés par le *bot* de l'action. À noter également que ces commentaires

peuvent être cachés (marqués *outdated*), il faut bien chercher chaque résultats de chaque image analysée.

En résumé, les analyses sont lancées lors des modifications de code, et il faut surveiller très régulièrement l'état des images utilisées (à minima une fois par semaine).

Intégrité logiciel

Le contenu des images distribuées n'est pas sous notre contrôle. Les distributeurs sont des acteurs reconnus à qui on peut faire confiance. Cependant la prise de contrôle de la distribution des images ne peut être complètement exclue. Nous pouvons prendre une mesure de sécurité à ce niveau en définissant des versions exactes des images employées. Avec cette politique on s'assure de ne pas récupérer d'images empoisonnée ou malveillante sans le savoir (comme cela se produirait en utilisant des versions sous la forme *:latest*).

On a en même l'assurance d'utiliser strictement les mêmes versions dans tous les contextes (developpement, CI, production).

Par contre, on devra être attentif à la découverte de nouvelles vulnérabilités. Les corrections donnent lieux à de nouvelle versions, il faut donc mettre en place une veille pour mettre à jour nos images : voir la section précédente (Voir Composants vulnérables)

Logs

Aucune modification n'a été faite à ce niveau, les logs des contenaires sont envoyés et gérér par le daemon Docker. (voir la section Logs de Docker)

Composer

Composer est un élément central de notre base de code, du déploiement et des scripts qui vérifient la sécurité des composants.

Aucun problème de sécurité de doit être toléré a ce niveau. Cependant, il doit être noté que l'exploitation de failles par ce vecteur est très limitée, pour plusieurs raisons. Les packages et plugins ont été choisis soigneusement et sont donc peu susceptibles d'altérer composer. Les packages font l'objet d'une vérification de sécurité avant mise en place (voir Dépendances Composer et Symfony). Les scripts sont rédigés par nos soins. Les problèmes de sécurités de composer sont extrêmement rares (liste publiée).

La solution est de garder une version de composer à jour.

Les mesures prisent sont une activation des notifications de sécurité sur GitHub et la mise en place d'un vérification auprès d'osl.dev dans la chaine d'intégration continue.

Dépendances Composer et Symfony

Symfony, bien que constituant le framework de structure, demeure un composant de *Composer*, et à ce titre, est analysé comme toute dépendance. Pour faciliter le maintient de la sécurité dans le temps, nous avons opté pour une version de *Symfony* soutenue sur le long terme (LTS).

La sécurité des composants (packages et plugins) est assurée par 2 mécanismes. La commande `composer audit` analyse les packages et nous informe de leurs vulnérabilités. Cette commande est lancée automatiquement lors de l'installation et la mise à jour de packages. Elle est également intégrée dans le process d'intégration continue. – https://getcomposer.org/doc/03-cli.md#audit

Le second mécanisme vise à interdire l'installation de packages vulnérables, c'est la dépendance composer *roave/security-advisories* qui assure cette tache.

Une amélioration possible serait de lancer régulièrement l'*audit* composer pour s'assurer que notre système n'est pas concerné par une vulnérabilité découverte après la publication de notre code.

Dépendances Javascript

Les dépendances javascript sont incluses via le composant *composer* asset-mapper. Ce composant possède une commande permettant la vérification des vulnérabilités dans les dépendances javascript (`console importmap:audit`)- https://symfony.com/doc/current/frontend/asset_mapper.html#run-security-audits-on-your-dependencies

Il ne doit y avoir aucune vulnérabilité à ce niveau, le javascript étant directement exposé au web.

Il faut faire une forte surveillance de ces dépendances.

La solution est d'automatiser cette vérification.

La mesure prise est de lancer le script de vérification avant les commits et dans le process d'intégration continue.

Pour amélioration, cette vérification mérite d'être faite régulièrement, en dehors des opérations sur le code.

Code

Le code est la partie la plus difficilement contrôlable par des mécanismes pré établis. Il engage la responsabilité du développeur, essentiellement au moment de l'écriture du code et des tests.

Il convient de respecter les pratiques recommandées et d'adopter une approche défensive du code. Nous avons adopté cette approche défensive qui consiste à considérer par défaut que les entrées sont sujet à malveillance.

Pour nous aider dans l'écriture de code non vulnérable, nous nous reposons sur l'utilisation de tests, l'utilisation de typage strict, et l'analyse de code par *phpstan*, configuré au niveau maximum. On s'est aussi attaché à utiliser les fonctionnalités et constructions récentes de php, pour une meilleure fiabilité (lisibilité, comportement plus stricts, dépréciation, etc). Pour cela nous avons utilisé *rector*. Pour maintenir du code formaté de façon standard, lisible (et donc plus maintenable) nous avons utilisé *php-cs-fixer*. Pour chacun de ces outils nous avons utilisé des extensions Doctrine, Symfony, etc quand elles sont existantes.

Une relecture du code, à froid, par une tierce personne serait une bonne chose en complément.

Le travail à réaliser se fait en continue lors du codage. Les outils cités précédemment sont utilisés dans le process d'intégration continue et lors des commits.

Certificats TLS

Contexte

Les certificats TLS sont essentiels pour le fonctionnement du client web et de l'api. Leur expiration rendrait l'api et le client inopérants.

Exigences

Les certificats doivent être maintenus à jour et générés par une autorité reconnue. L'échange des données sans TLS ne doit pas être possible pour empecher une utilisation non sécurisée.

Solutions

- Mettre en place une tâche de renouvellement automatique de ces certificats.
- Utiliser les services de l'autorité *letsencrypt* qui génère des certificats simplement et gratuitement.
- Rediriger les communications réseaux non sécurisées (port 80) vers le canal supportant le TLS (SSL)

Mesures

- Une tache cron est installée sur les 2 serveurs, elle lance cette régénération et le redémarrage (requis) du server *nginx*.
- Il faut vérifier que le cron est bien en place sur les machines (voir 4. Renouvellement des certificats TLS)

Ce processus peut être amélioré en déléguant la gestion du cron à Symfony (voir notes dans le script référencé au dessus).

Noms de domaine

Les noms de domaine *api.ecf.seb7.fr* et *cli.ecf.seb7.fr* sont enregistrés chez ovh.

La sécurité de ces domaines consiste à vérifier qu'ils sont protégés contre le transfert (c'est bien le cas) et qu'ils n'arrivent pas à expiration (c'est bien le cas, un renouvellement automatique est en place).

Dans ces circonstances, la seule action requise est de vérifier de temps à autre que le renouvellement automatique n'est pas suspendu et de vérifier les e-mails en provenance d'OVH.

On peut envisager d'automatiser le processus de surveillance en déclenchant l'envoi d'une alerte en inspectant la date d'expiration renvoyée par le service *whois* quand cette date approche.

Dépôt de code

Le dépôt de code est accessible publiquement en lecture. Ce dépôt de code est lié à des actions qui déclenchent l'analyse du code et le lancent le déploiement.
L'envoi de code se fait par ssh via l'utilisation d'une clé ssh.

Les actions importantes sur le dépôt (via l'administration en ligne) demandent une identification qui est réalisée par une double authentification (mot de passe (fort et stocké dans une application locale dédiée) et saisie de code dans l'application mobile installée sur le téléphone du développeur).

On peut considérer que la sécurité est assurée par ces ménanismes. Il est donc à noter que la clé ssh privée du développeur joue un rôle important.

Les mesures sur ce point sont indiquées dans la section consacrée au stockage de la clé ssh, (Voir Machines du développeur) .

Il faut également veiller à ce que le code ne contiennent pas directement de *secrets* (identifiants, mots de passe, token, etc). Cette sécurité est assurée par le respect des bonnes pratiques, celle de Symfony en particulier. (https://symfony.com/doc/current/best_practices.html#security , https://symfony.com/doc/current/best_practices.html#use-environment-variables-for-infrastructure-configuration). Une exception à été faite et est détaillé dans les améliorations (voir Application Symfony)

Action GitHub *ssh-remote-commands* (déploiement)

Le déploiement est assuré par l'action github *ssh-remote-commands (* *https://github.com/marketplace/actions/ssh-remote-commands* *).* L'adoption de cette action, inconnue de notre part, a été murement réfléchie. L'action est très utilisée (donc très surveillée) et depuis un temps assez long, elle est référencée sur la *marketplace*, ces points sont des critères de qualité mineurs mais non négligeables.

Aucun problème de sécurité n'est référencé sur osv.dev (source). L'action repose sur une librairie tierce qui a également été contrôlée (aucun vulnérabilité connu, code inspecté rapidement)

L'utilisation de l'action requière une clé ssh privée pour lancer accéder au *droplets* de production. Une clé ssh a été créée spécialement pour cet usage. L'utilisation de la clé est faite via un *secret* GitHub pour empêcher son affichage ou sa diffusion.

Une vulnérabilité dans l'action pourrait entrainer la fuite de la clé qui permettrait à son tour l'accès au serveur hébergeant l'application. Un soin marqué doit être apporté au suivi de sécurité de cette action.

Un suivi manuel doit être assuré. Il n'y a actuellement pas de suivi automatisé de notre part. Une part importante de la veille de sécurité de ce composant repose sur GitHub qui ne manquera pas de désactiver l'action et prévenir les utilisateurs si elle posait problème.

Une surveillance automatisée apporterai peu de chose pas de sécurité supplémentaire car l'action serait désactivée. Pour éviter l'utilisation d'une version malveillante publiée illégitimement, nous avons fixé la version des actions avec un hash qui assure que c'est toujours la même version qui est utilisée.

Aucun droit d'écriture n'est accordé à cette action.

Pour le suivi, nous avons souscrit aux notifications liées à la sécurité et aux *releases*.

Autres actions GitHub

Les autres actions GitHub sont très répandues, éditées par *GitHub* (actions/checkout) et *Docker*. Nous pouvons leur déléguer la sécurité en toute confiance.

Nous appliquons cependant toutes les mesures énumérée dans la section Action GitHub ssh-remote-commands. (utilisation de hash, souscription aux alertes, droits minimisés)

On notera que l'action d'analyse des images Docker, repose sur une identification au service *Docker Scout* et que cette identification se fait en utilisant des *secrets* pour garantir la non diffusion de ces informations dans les résultats et détails des *workflows*.

Accès données privées (identifiants)

L'accès aux données de la base de données est couvert par la section PostgreSQL. Il n'y a pas de données utilisateurs hors base de données.

Accès par le web / nginx

Il faut surveiller l'accès aux données de configuration stockées sur les serveurs.

L'ensemble des fichiers de configuration et du code source sont hors d'accès web. L'accès racine du serveur est *public/*. Elles ne sont donc pas accessibles par le web. La vérification à été faite. La surveillance de contenaire *nginx* est impliquée dans ces accès puisque c'est ce serveur qui assure l'accès aux ressources (images, etc). (Voir Image nginx et Configuration nginx).

Il ne faut pas y placer de fichiers sensibles. Il appartient au développeur de continuer à respecter cette règle évidente.

Données de sessions et cookies

L'api n'utilise pas de cookies ou de données de session. Pour le client web, les données liées à l'utilisateur connecté ne sont pas conservées directement dans un cookie mais dans la session php (avec un cookie qui identifie la session).

Il ne doit pas être possible d'usurper une identité.

La solution consiste à limiter au maximum le risque de fuite de l'identifiant de session qui doit être échangé entre le client et le serveur.

Ici, il n'y a pas de risque de fuite de données par vol de cookie. Il pourrait subsister un risque de vol de l'identifiant de session par le vol du cookie. Mais Symfony est configuré pour indiquer que le cookie est uniquement disponible par http et pas créé en javascript, ce qui empêche tout vol de données par javascript (`framework.cookie_httponly : true`). Les échanges client/serveur passent par un canal TLS, une attaque de type *man in the middle* n'est donc pas possible.

Il n'y a pas de mesure à prendre autre que maintenir la configuration dans cet état.

Configurations

Application Symfony

Les configurations de l'application sont placées dans les fichiers *.env.local* (identifiants à la base de données et la variable *APP_SECRET* pour la génération des tokens crsf entre autre). Ça n'est pas la pratique recommandée. Cette pratique à été utilisée car elle permet de synchroniser ces identifiants entre *Docker compose* et l'application *Symfony*. Le fichier *.env.local* n'est pas versionné, interdit au versionnage (ligne présente dans le fichier *.gitignore*), et non accessible par le web, ce qui assure la sécurité des données.

Il convient d'utiliser les secrets *Symfony*.

Bien qu'en pratique, il n'y ai pas de problème de sécurité, il est préférable de stocker ces informations de façon sécurisée (cryptée) Référence : https://symfony.com/doc/current/configuration/secrets.html

Configuration Docker

Les identifiants utilisés par docker sont destinés au contenaire P*ostgreSQL*, se référer à la section précédente.

Surveillances à appliquer

En plus des mesures décrites précédemment, il convient d'inspecter les logs d'erreur *nginx*, *PHP*, *Symfony* et système pour détecter d'éventuelles tentatives de piratage.

Améliorations

Actions à entreprendre pour améliorer la sécurité.

- Ubuntu : Mettre en place un outil d'envoi de mail pour recevoir les notifications des *unattended upgrades*. (référence : https://www.digitalocean.com/community/tutorials/send-email-linux-command-line)
- Recevoir des alertes quand nos images Docker présentent de nouvelles vulnérabilités critiques ou sévères.
- Le cron de renouvellement des certificats : ce processus peut être amélioré en déléguant la gestion du cron à *Symfony* (voir notes dans le script référencé) et profiter de la gestion d'erreur de *Symfony*.
- Nom de domaine : On peut envisager d'automatiser le processus de surveillance en déclenchant l'envoi d'une alerte en inspectant la date d'expiration renvoyée par le service *whois* quand cette date approche.
- Vérification de l'exécutable composer : la vérification se fait lors de l'envoi de code, elle serait à déclencher régulièrement (avec une alerte).
- Vérification régulière des vulnérabilités des packages composer. Composer :audit. Cette vérification pourrait aussi être facilement lancée quotidiennement par le process d'intégration continue.
- Vérifier les composants JavaScript quotidiennement, voir item précédent.
- Stocker les identifiants de base de données de façon plus sécurisée : Référence : https://symfony.com/doc/current/configuration/secrets.html
- empêcher la publication directe de code sur la branche main qui déclenche le déploiement (protection de branche).
- Vérifier que les mots de passe saisi ne sont pas dans le top 10 000 des mots de passe les plus utilisés. Référence : https://haveibeenpwned.com & api disponible ici : https://api.pwnedpasswords.com/range/.

Annexes

1. Vérification du type de connexion ssh aux Droplet

Cette vérification c'est fait en recherchant dans les fichiers de configuration ssh.

Cd /etc/ssh/ grep PasswordAuthentication * ssh_config :# PasswordAuthentication yes grep : ssh_config.d : Is a directory **sshd_config :PasswordAuthentication no** sshd_config :# PasswordAuthentication. Depending on your PAM configuration, sshd_config :# PAM authentication, then enable this but set PasswordAuthentication grep : sshd_config.d : Is a directory **sshd_config.ucf-dist :PasswordAuthentication no** sshd_config.ucf-dist :# PasswordAuthentication. Depending on your PAM configuration, sshd_config.ucf-dist :# PAM authentication, then enable this but set PasswordAuthentication

2. Mise en place des mises à jour Ubuntu

Vérification que le service *unttended upgrades* est en fonctionnement :

```
systemctl status unattended-upgrades.service
```

Indique que le service est actif et configuré pour démarrer au lancement de la machine.

La configuration du service est vérifiée de cette façon :

```
~# apt-config dump unattended-upgrade
Unattended-Upgrade  « » ; Unattended-Upgrade ::Allowed-Origins  « » ;
Unattended-Upgrade ::Allowed-Origins ::  «${distro_id}:${distro_codename} » ;
Unattended-Upgrade ::Allowed-Origins ::  «${distro_id}:${distro_codename}-
security » ; Unattended-Upgrade ::Allowed-Origins ::  «${distro_id}ESMApps :$
{distro_codename}-apps-security » ; Unattended-Upgrade ::Allowed-Origins ::  «$
{distro_id}ESM :${distro_codename}-infra-security » ; Unattended-
Upgrade ::DevRelease  «auto » ; Unattended-Upgrade ::Automatic-Reboot  «true » ;
Unattended-Upgrade ::Automatic-Reboot-Time  «02 :00 » ;
```

La configuration est liée aux fichiers */etc/apt/apt.conf.d/50unattended-upgrades* et */etc/apt/apt.conf.d/52unattended-upgrades*

Bien que cela ne soit pas nécessaire, pour limiter la maintenance du système, un reboot à été rendu possible (si nécessaire) (à 2h du matin).

Il a également été contrôlé qu'un reboot de la machine relance automatiquement le service docker et que les contenaires sont relancés. L'application est donc relancée après mise à jour du système.

3. Analyse des images docker

Image php-fpm personnalisée

Image : ghcr.io/sebsept/docker-php-symfony-starter:prod Cette image est basée sur l'image php-fpm alpine php:8.3.4-fpm-alpine3.18 – https://hub.docker.com/layers/library/php/8.3.4-fpm-alpine3.18/images/sha256-d468b558e1d22bcd4fc861c62d07f2ee0fb7b412e8baaf8b294e41eace8c9b47?context=explore Cette image présente 2 vulnérabilités medium et 3 vulnérabilités non spécifiées Une mise à jour vers la version 8.3.7-fpm-alpine3.20 à été effectuée et la nouvelle image ne présente aucune vulnérabilité.

Image nginx (serveur web)

L'image nginx :1.21.3 : contient des vulnérabilités. La mise à jour 1.26.0 contient une vulnérabilité moyenne (pas de vulnérabilité critique ou sevère). Après analyse, il apparaît que la vulnérabilité, dans l'exécutable nghttp2, permet de faire une consommation excessive du cpu si on fait une requête sur une url qui renvoi continuellement des frames *continuation*. L'exploit n'est pas grave et ne surviendra pas puisque les url appelées par le client et l'api sont strictement sous notre contrôle. Les vulnérabilités de niveau bas, que nous tolérons, au regard de notre politique, sont ignorées après une analyse rapide.

Cette nouvelle image peut donc être utilisée.

PostgreSQL (base de données)

La version employée contient des vulnérabilités (2 critiques et 36 sévères). Une mise à jour vers la version 16.3-alpine3.20 à été effectuée, elle ne contient aucune vulnérabilité.

Notez que nous sommes passé d'une version basée sur Debian à une version basée sur Alpine linux.

certbot/certbot (outils de création des certificats TLS)

Le service *certbot* présente 2 vulnérabilités sévères. Cependant cet outil n'est utilisé que pour générer les certificats SSL/TLS, il ne fonctionne qu'a ce moment, n'est pas accessible en ligne (aucun port ouvert), pas utilisé par d'autres contenaires et n'est lié à aucun volume de stockage. On peut le considérer sûr malgrés les vulnérabilités indiquées (non affichées sur hub.docker)

4. Renouvellement des certificats TLS

Sur chaque chaque machine concernées :

vérifier que la tache cron est installée et programée au moins tous les 90 jours :

```
~# crontab -l 0 0 */90 * * /app/cron/certificates.sh >/dev/null 2>&1
```

On peut voir si une ligne référençant le script */app/cron/certificates.sh*

À la mise en place on peut vérifier que le script est fonctionnel en le lançant

```
/app/cron/certificates.sh
En plus du résultat affiché on pourra vérifier dans un navigateur que le
certificat date bien du jour de régénération.
```

Diagramme MCD

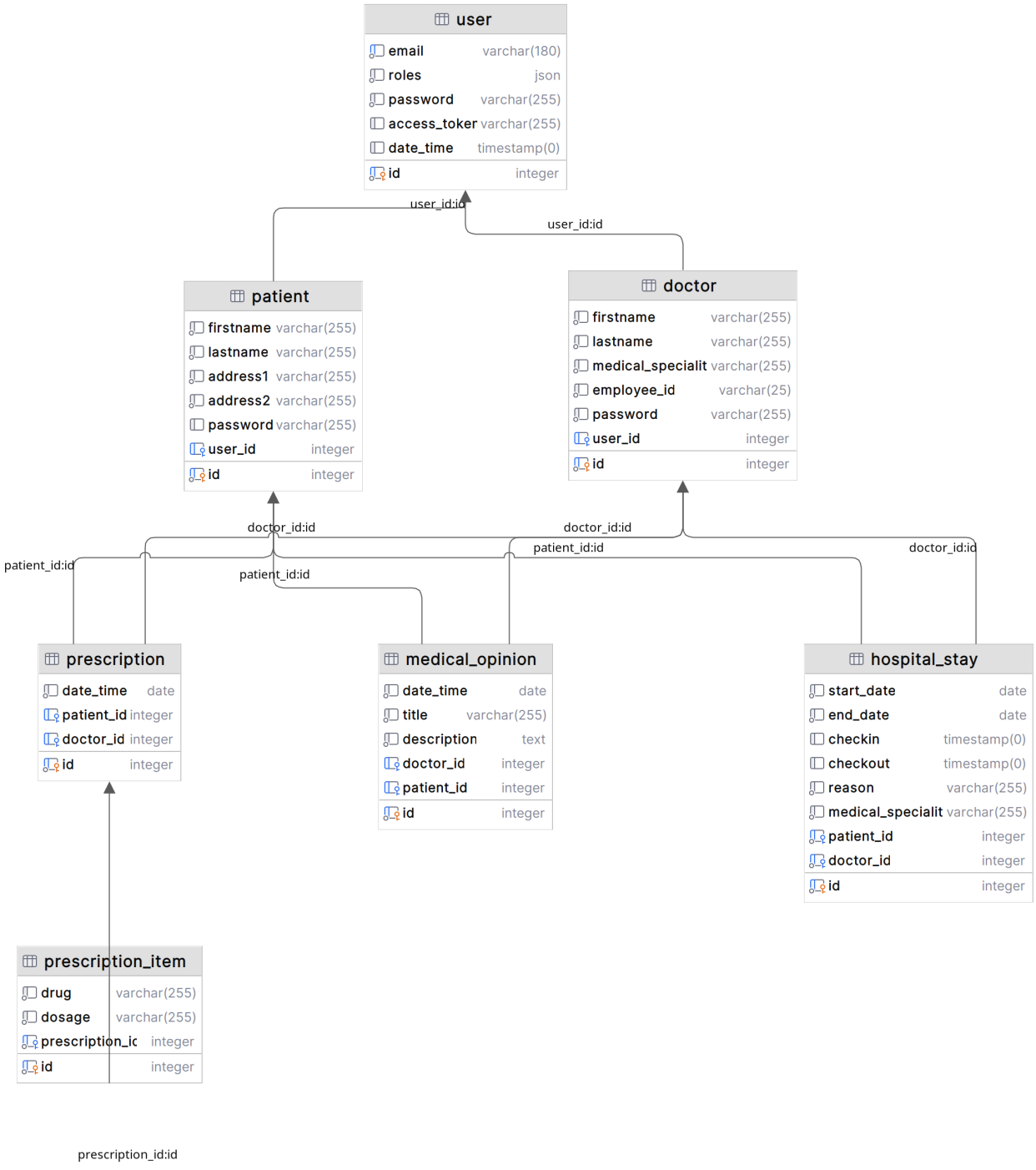
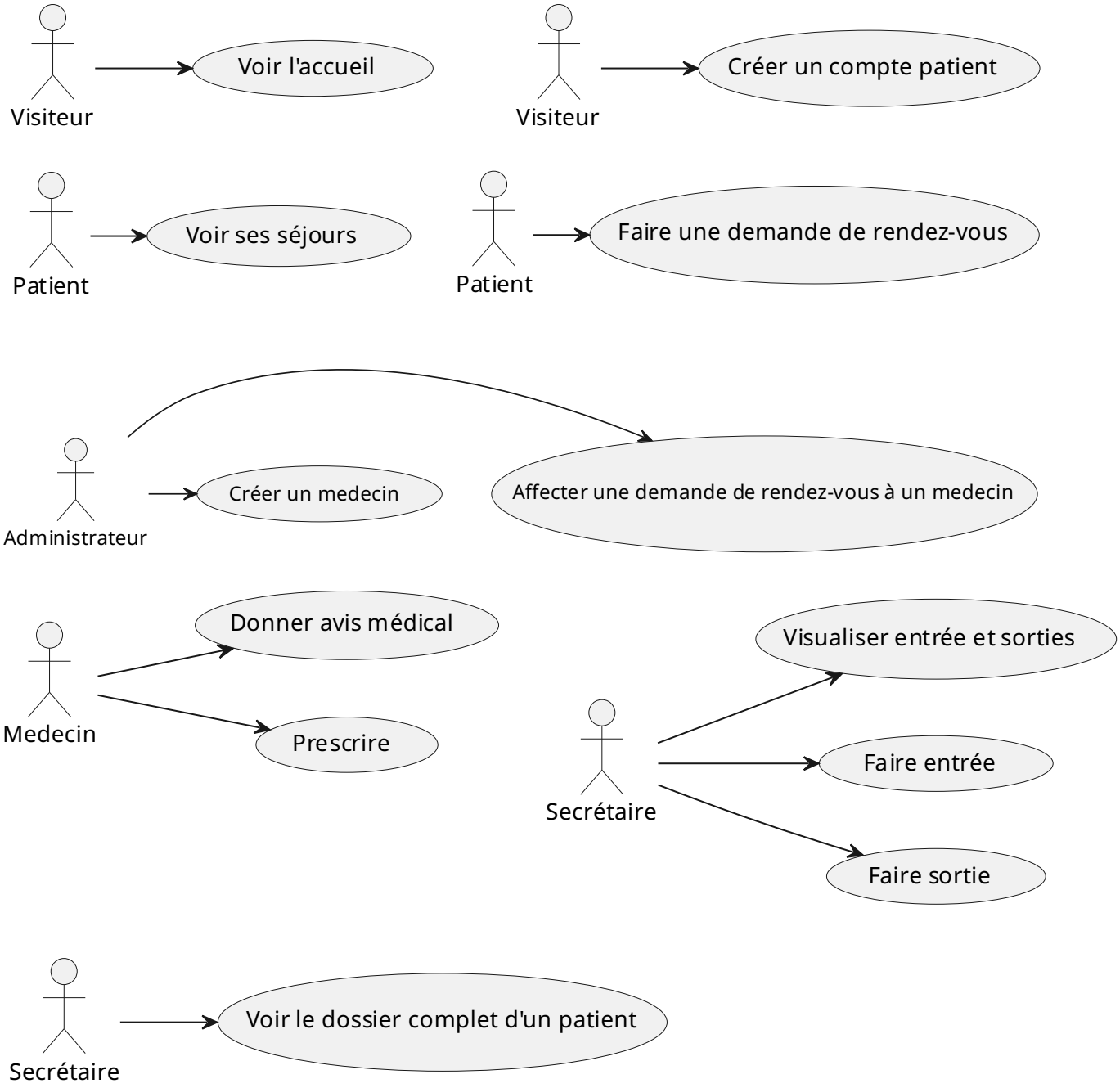


Diagramme cas d’utilisations



Documentation technique

Ce document présente succinctement les aspects techniques du projet.

Prérequis logiciel

Logiciels requis

- *Docker 27.0.3*
- *Docker compose* 2.17.2
- Git
- IDE (*PhpStorm* avec le plugin *Symfony* sont recommandés) - https://www.jetbrains.com/phpstorm/
- Un navigateur web récent (*Firefox* par exemple)
- LibreOffice (pour consulter et mettre à jour les documentations).

Les version de *Docker* et *Docker compose* en production sont les versions proposées par Ubuntu 22.04.4 LTS, elles seront mises à jour. Il convient d'avoir sur les machines de développement et

d’intégration continue, les versions les plus proches possibles. Cependant, n’exploitant pas de fonctionnalités avancée ou très récentes, des versions différentes peuvent convenir.

Logiciels recommandés

- Lanceur de tâche *Just* - https://just.systems/ (auto-complétion native avec *Fish*)
- un terminal *Fish* (3.7.0) - https://fishshell.com/
- Un outil pour tester les requêtes (HTTPie, PostMan ou autre)

Les serveurs, outils de tests, linting, etc sont lancé dans les contenaires *Docker*, il n’y a pas d’autres prérequis, il n’est pas nécessaire d’avoir PHP ou un SGBD sur la machine de développement.

Le lanceur de taches *Just* est destiné à le quotidien du développeur. Il permet de lancer des commandes simples (et rapidement avec l’auto-complétion (*Fish*)). Par exemple pour récupérer le code dans le dépôt, récupérer les images Docker, stopper et redémarrer les contenaires puis lancer *composer install* dans le contenaire PHP, il suffit de lancer `*just update*` et l’ensemble des opérations s’effectue. L’ensemble des taches est listé et documenté dans le fichier *.justfile*.

Environnements

Environnement de développement

L’environnement de développement est le plus simple possible et est complètement reproductible. Il est basé sur Docker.
Cet environnement est défini et orchestré selon les spécifications du fichier *compose-dev.yaml* .

Environnement de production

L’environnement de production est presque similaire à l’environnement de développement avec un contenaire en plus, celui qui à pour mission de produire les certificats SSL/TLS des applications.

Cet environnement est défini dans le fichier *compose-prod.yaml*

Le système d’exploitation hôte est Ubuntu 22.04 LTS

L’environnement de production se déploie automatiquement par le processus d’intégration continue.

Environnement d’intégration continue

L’environnement d’intégration continue est déployée dans un *runner* GitHub, *Ubuntu latest*.

Contenaires

Les détails de ses contenaires sont définis dans les fichiers *compose-prod.yaml* et *compose-dev.yaml*.

Contenaire nginx

C’est le serveur web, toutes les requêtes entrantes et sortantes passe par ce contenaire.
Il transmet sert directement des fichiers existants (assets publiques) ou passe les requetes au contenaire php-fpm.

Contenaire php-fpm

C’est un contenaire personnalisé, défini ici : https://github.com/SebSept/docker-php-symfony-starter. Il héberge une application Symfony/ApiPlatform qui effectue les traitements, c’est le cœur de nos applications.

Contenaire PostgreSQL

Contenaire avec la base de données. N’est accessible par le contenaire php-fpm.

Ce container n’existe que sur l’application API et pas sur l’application de client web.

Container Certbot

Ce contenaire est destiné à la création des certificats SSL/TLS, il n’est pas actif en permanence mais uniquement lorsqu’une tache *cron* de renouvellement des certificats est lancée.

Processus d’intégration continue

Ce processus est également décrit dans le document *7. Mise en production et maintenance*

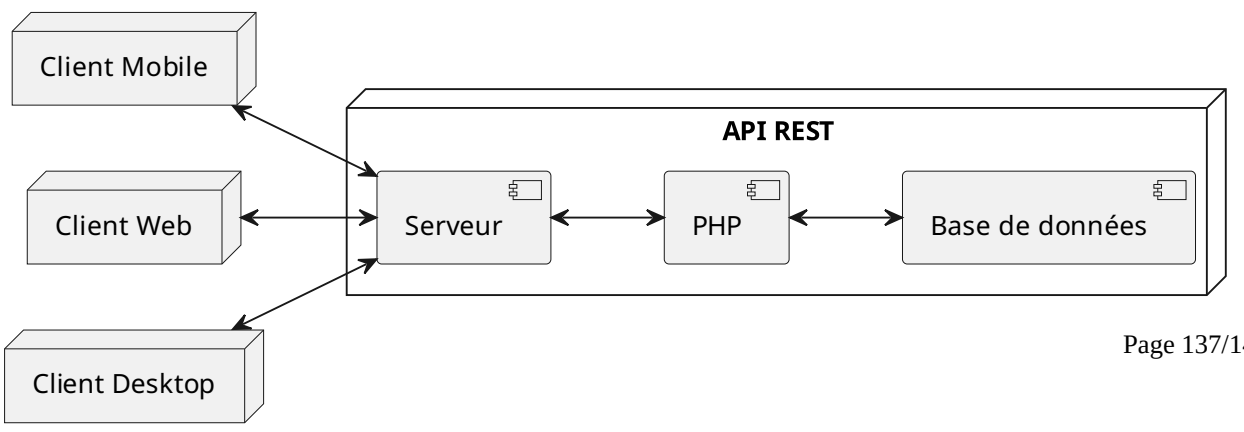
Mise en place du processus d’intégration continue

Le processus nécessite plusieurs autorisations :

- Exécuter les commandes ssh sur la machine de production. Cette mise en place est documentée dans l’action GitHub dédiée : https://github.com/appleboy/ssh-action.
- Récupérer le code des dépôt GitHub à partir des machines de production. C’est documenté officiellement ici : https://github.com/SebSept/soignemoi-api/settings/keys et https://docs.github.com/fr/authentication/connecting-to-github-with-ssh/managing-deploy-keys#deploy-keys

Architecture applicative

On a une architecture de type client/serveur organisée de cette façon :



Applications clientes

Client web

Le client web est réalisé avec *Symfony*, de façon indépendante du composant API pour permettre une évolutivité découplée des deux composants.

Le coté front est réalisé avec *Bootstrap* de manière responsive.
L'application est destinée aux patients, toutes les pages patient s'affiche parfaitement sur toutes les résolution d'écran et tout les types de dispositifs.

Le code source est disponible sur GitHub : https://github.com/SebSept/soignemoi-webcli
L'accès web se fait à cette adresse : https://cli.ecf.seb7.fr/

Client mobile

Le client mobile est réalisé avec Flutter par l'intégration d'une simple *webview* (navigateur internet interne).

Cette application est destinée aux médecins.
Les pages de l'application web (décrite au dessus) destinées aux médecins s'affichent parfaitement sur les écrans de type téléphone mobile.

Le code et une application prête à l'installation sont disponibles sur GitHub :
https://github.com/SebSept/soignemoi-mobilecli

Client desktop

Le client desktop est développé avec electronjs. Ce client embarque simplement une vue web affichant le client web.

Cette application est destinée aux secrétaires.
Les pages de l'application web (décrite au dessus) destinées aux secretaires s'affichent parfaitement sur les écrans de type ordinateur.

Le code et une application prête à l'installation sont disponibles sur GitHub :
https://github.com/SebSept/soignemoi-desktopcli

Application API REST

L'API sera construite avec *API Platfom* en utilisant *Symfony*

L'ensemble de la documentation de l'API est généré automatiquement lors de chaque commit et est disponible au format openapi via une interface swagger : dans le fichier *docs/openapi/index.html* (a ouvrir avec un serveur web local, par exemple avec *php -S localhost:8077 -t ./*) ou plus simplement en mode développement en accédant au serveur d'api local.

Diagrammes

Schéma réseau

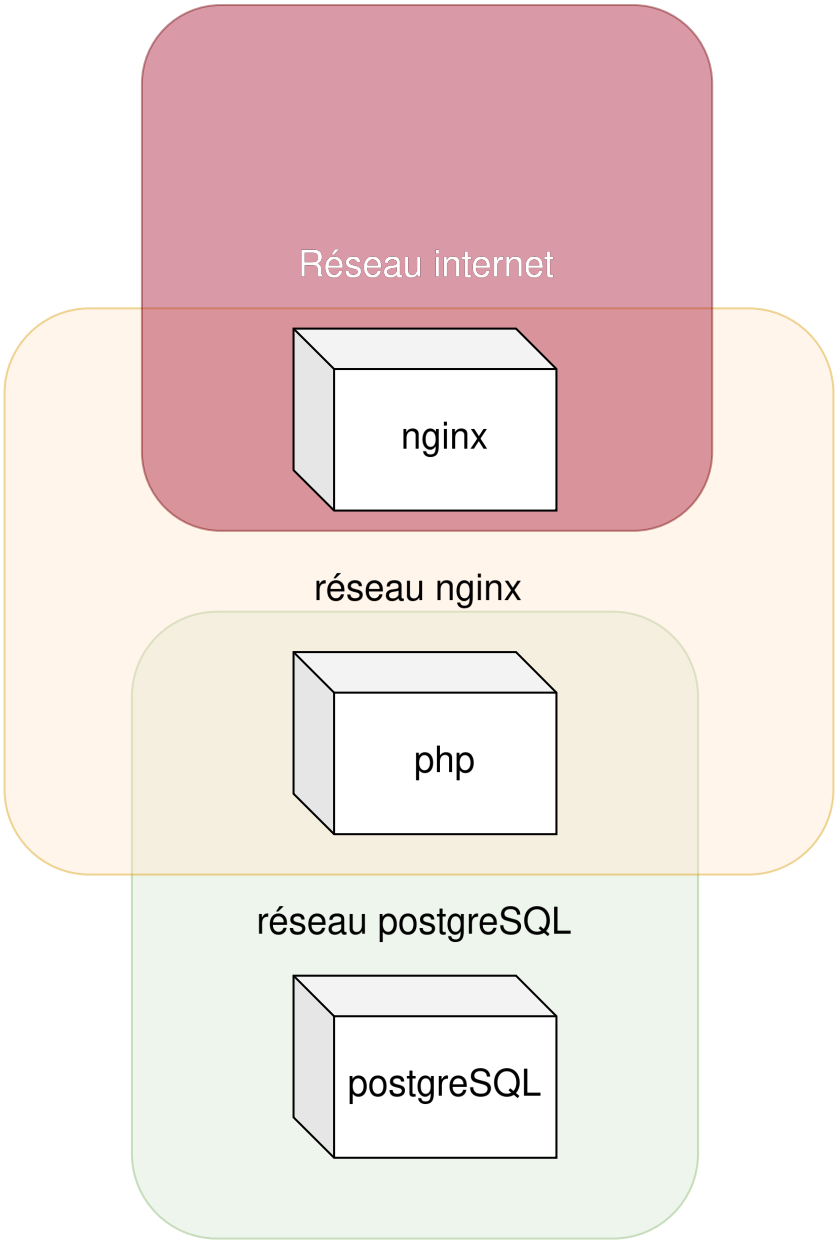


diagramme MCD

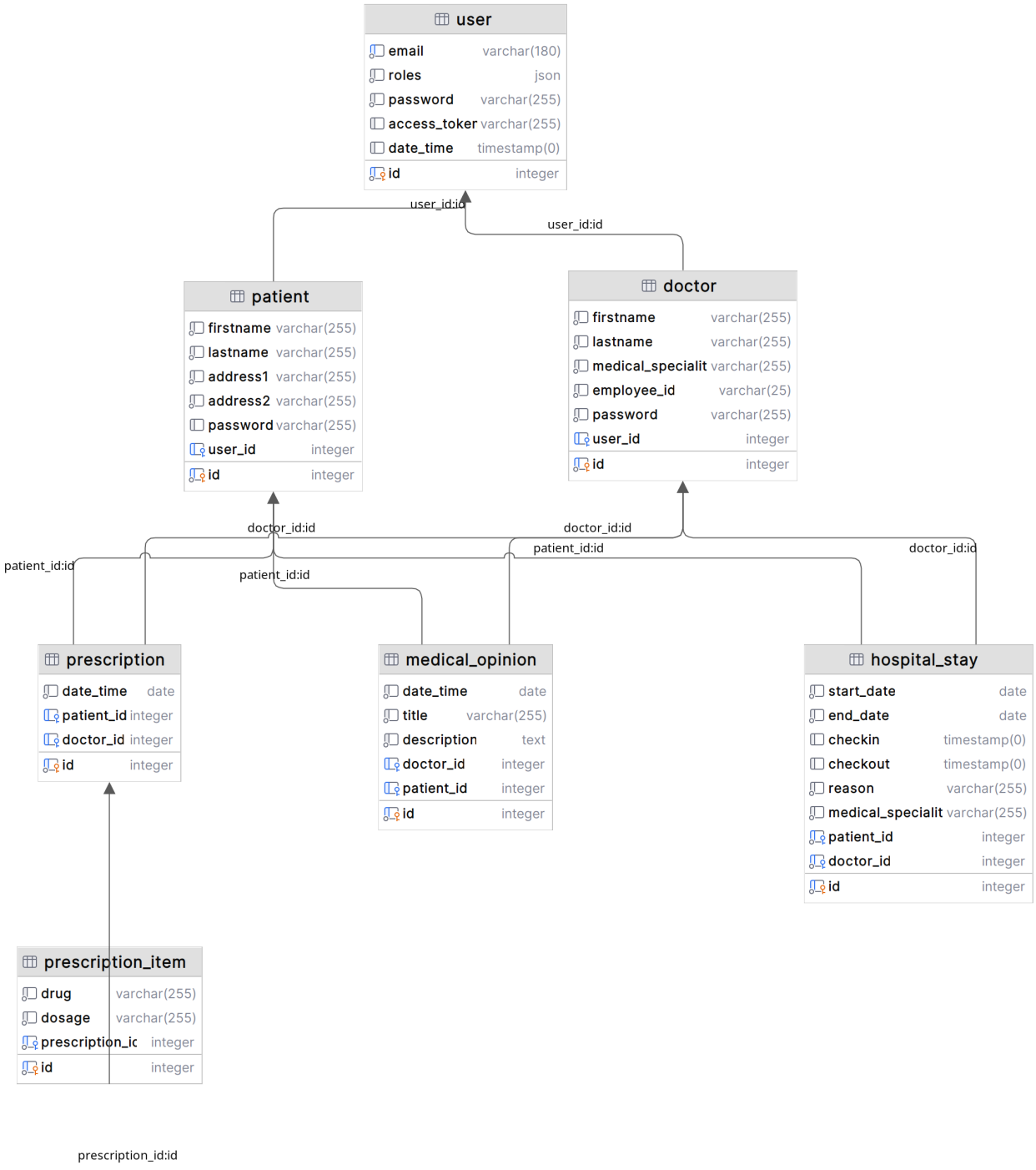


Diagramme de déploiement

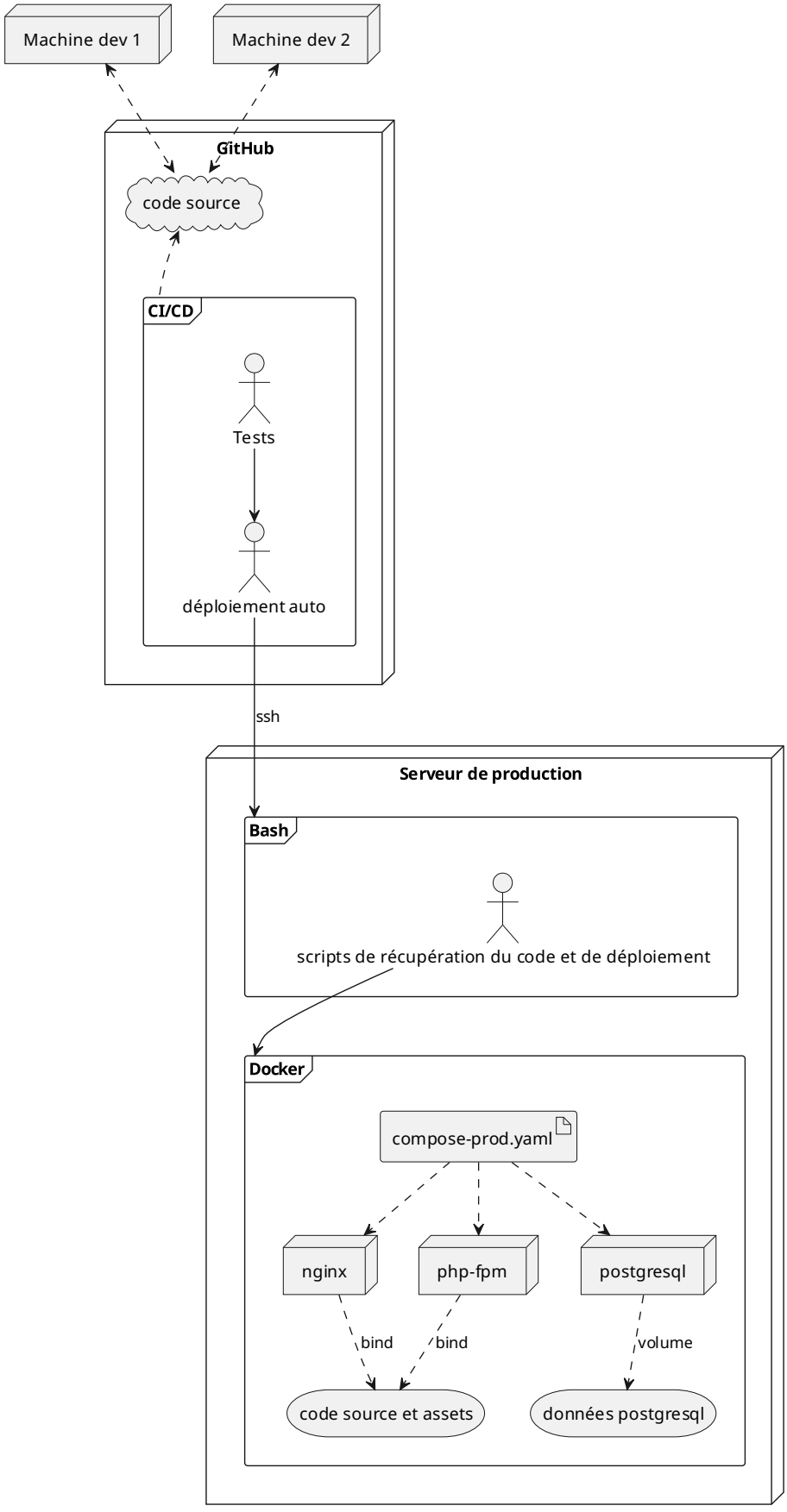


Diagramme de séquence pour un séjour

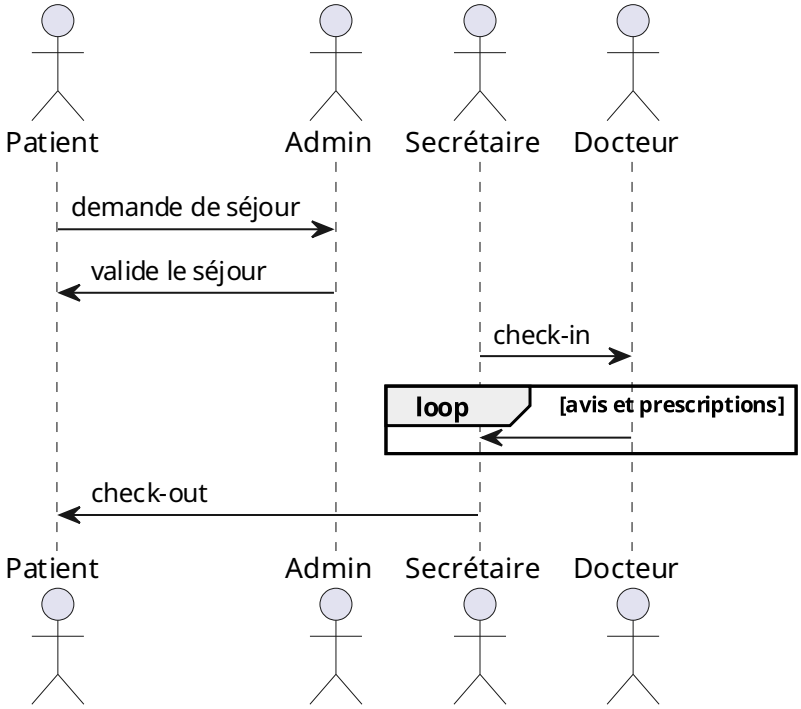


Diagramme d’activité pour un séjour

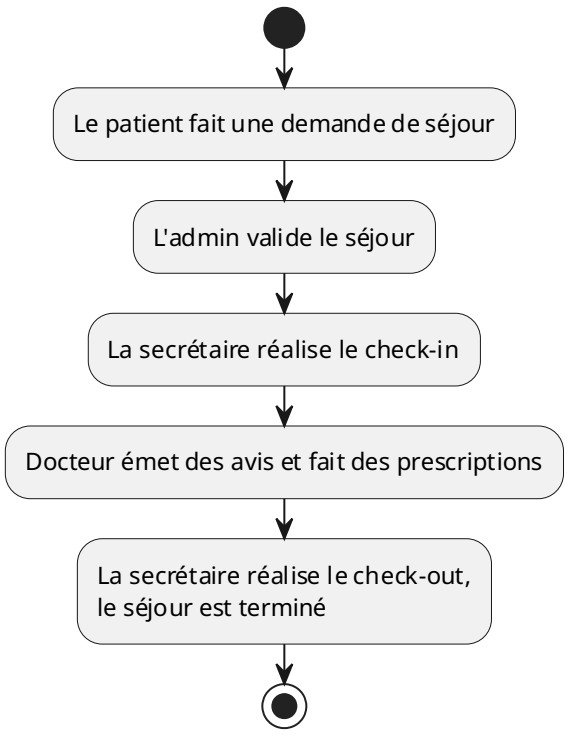


Diagramme de Classe

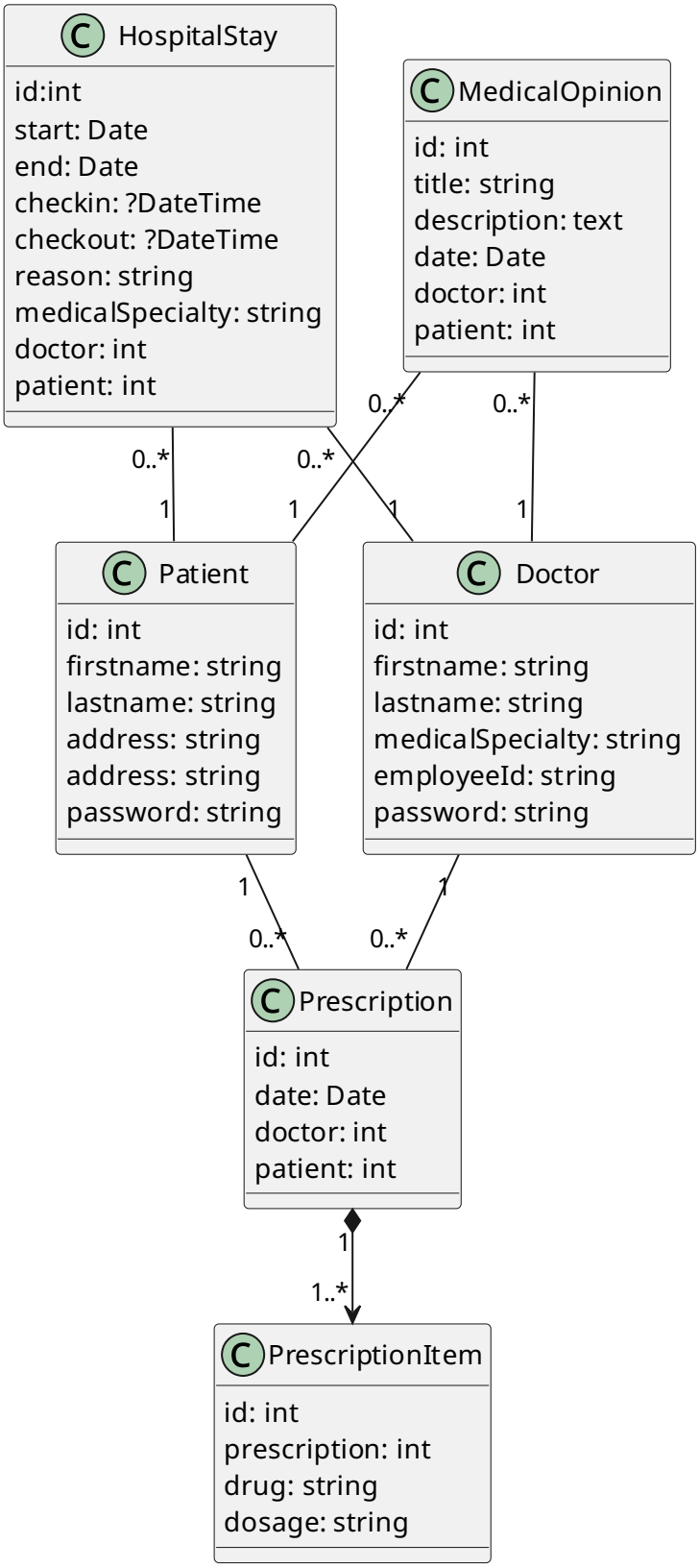


Diagramme de cas d’utilisations