Hes·so VALAIS WALLIS
Haute Ecole de Gestion
Hochschule für Wirtschaft

Filière Informatique de gestion
Studiengang Wirtschaftsinformatik

Travail de Bachelor 2020

Automatisation du processus de release des SDK smartscan

Étudiant : Ken Jäggi

Professeur : Laurent Sciboz

Déposé le : 7 août 2020

Résumé

Actuellement, le processus de release Android et iOS des décodeurs conçu par l'institut Icare est composé de différentes étapes. La génération, les tests de communication avec la caméra et la publication du SDK sont réalisés manuellement. L'objectif de ce travail est de faciliter ces tâches aux employés de l'institut Icare avec une plateforme web.

Afin de mener à bien cet ouvrage de Bachelor, le développement utilise les outils de la méthodologie Scrum. Le reste du travail est fait parallèlement. Les mockups finaux sont obtenus au bout de quatre itérations. La documentation est écrite durant la conception. Cependant, la majorité est réalisée lors des dernières semaines.

La procédure de test de communication entre le SDK smartscan et la caméra est élaborée. Pour des questions de priorité, le processus n'est pas implémenté. L'architecture 3-tiers de la plateforme web Genica s'appuie sur des conteneurs Docker. Tant bien pour le développement que pour l'environnement d'intégration et de production. Néanmoins, ce ne sont pas les mêmes conteneurs qui sont utilisés. Cette plateforme Genica est composée d'une API réalisée avec Django REST Framework, d'un worker, d'une interface graphique Vue.js et d'une base de données PostgreSQL. La génération des SDK smartscan Android et la publication des librairies sur le dépôt Gitlab se reposent sur la plateforme.

Mots-clés : Docker, Django, REST, Vue.js, Institut Icare, automatisation.

Avant-propos et remerciements

Cet ouvrage rentre dans le cadre de l'obtention du bachelor en informatique de gestion de la HES-SO Valais/Wallis. Il vise à automatiser le processus de release des SDK smartscan que conçoit l'institut Icare. Ce travail est né d'un besoin de simplification de la procédure de génération de ces SDK.

En effet, un client de l'institut Icare a la possibilité de sélectionner une combinaison de décodeurs ayant chacun son propre numéro de version. Ensuite, l'institut Icare fournit un seul SDK par système d'exploitation mobile accompagné de diverses sécurités pour limiter l'accès.

Ce travail de Bachelor a pour objectif la conception et la création d'une architecture capable d'automatiser l'ensemble du processus ainsi que les tests. Étant donné la charge de travail, il a été décidé de concentrer l'étude sur les différentes analyses et la génération de SDK Smartscan par une plateforme web. La méthodologie utilisée pour la gestion de ce projet est basée sur Scrum.

Je tiens personnellement à remercier l'institut Icare qui a offert la possibilité de réaliser certaines parties du développement lors de mes heures de travails. Je suis heureux de savoir que ce projet aura une suite et un grand intérêt pour mes collègues. Merci à Benoit Golay, le product owner, de m'avoir orienté et défini les priorités pour la plateforme. Merci à Christian Pereira et Nouredine Hussain d'avoir pris le temps de répondre à mes questions techniques sur les décodeurs et leur génération. Merci aux relecteurs Flurina Manser et Christian Pereira. Merci à ma famille et à mon entourage de m'avoir soutenue pendant cette période. Finalement, je tiens à remercier mon professeur responsable Laurent Sciboz pour l'aide apporté durant ce travail et de m'avoir incité à écrire le rapport dès le début.

TABLE DES MATIÈRES

LISTE DES TABLEAUX ..... **VIII**

LISTE DES FIGURES ..... **IX**

LISTE DES ABRÉVIATIONS ..... **XI**

INTRODUCTION **1**

1. MÉTHODOLOGIE **2**

1.1. SCRUM 2

1.1.1. L'équipe *2*

1.1.2. Les cérémonies *3*

1.1.3. Les artefacts *3*

1.1.4. Méthodologie appliquée au projet *4*

1.2. MÉTHODOLOGIE POUR LA DOCUMENTATION 4

1.3. MÉTHODOLOGIE POUR LES MOCKUPS 4

1.4. MÉTHODOLOGIE DE L'ÉTAT DE L'ART 5

1.5. MÉTHODOLOGIE POUR LE DÉVELOPPEMENT 5

2. ÉTATS DE L'ART **6**

2.1. CHOIX DU FRAMEWORK BACKEND 6

2.1.1. Express.js *8*

2.1.2. Laravel *9*

2.1.3. Django Rest Framework *10*

2.1.4. Flask *11*

2.1.5. Ruby on Rails *11*

2.1.6. Prise de décision pour le backend *12*

2.2. CHOIX DU FRAMEWORK FRONTEND 15

2.2.1. Angular *15*

2.2.2. React *16*

2.2.3. Vue.js *17*

2.2.4. Prise de décision pour le frontend *18*

2.3. PLATEFORMES DE TEST MOBILE AVEC DES APPAREILS REELS 19

2.3.1. Firebase Test Lab *21*

2.3.2. Kobiton *23*

2.3.3. AWS Device Farm *24*

2.3.4. Prise de position *25*

3. BESOINS DE LA PLATEFORME 27

4. ANALYSES 29

4.1. STRUCTURE DES DÉPÔTS GIT 29

4.2. ANALYSE D'AUTOMATISATION DES TESTS SUR LE CODE SOURCE 30

4.2.1. Existant 30

4.2.2. Proposition 30

4.3. ANALYSE D'AUTOMATISATION DES TESTS D'UN SDK SMARTSCAN 33

4.3.1. Existant 33

4.3.2. Proposition 34

4.4. ANALYSE DE LA COMMUNICATION ENTRE L'API ET LE WORKER 37

4.4.1. Sur le même serveur 37

4.4.2. Communication SSH 37

4.4.3. Communication web sockets 38

4.4.4. Communication message broker 39

4.4.5. Choix de la communication 40

5. LES MOCKUPS 41

5.1. VERSION 1.0 41

5.2. VERSION 1.1 41

5.3. VERSION 1.2 42

5.4. VERSION 1.3 43

6. L'ARCHITECTURE IT 44

6.1. ENVIRONNEMENT LOCAL 45

6.2. ENVIRONNEMENT D'INTÉGRATION 46

6.3. ENVIRONNEMENT PRODUCTION 47

6.4. ARCHITECTURE FINALE 48

7. LE DÉPÔT GIT DE GENICA 50

7.1. LA STRUCTURE DES RÉPERTOIRES 50

7.2. LES BRANCHES 51

7.3. L'INTÉGRATION CONTINUE 52

7.3.1. L'étape test 53

7.3.2. L'étape build 54

7.3.3. L'étape deploy 55

8. L'API ET LE WORKER 56

8.1. STRUCTURE 56

8.1.1. Répertoire genica_backend 57

8.2. LES CONTENEURS 59

8.2.1. Dockerfile de production pour le serveur API 59

8.2.2. Dockerfile de production pour le worker 59

8.3. APPLICATION CLIENT 60

8.4. APPLICATION LIVRAISON 60

8.4.1. La génération des SDK 62

8.4.2. La livraison des SDK 65

8.5. BASE DE DONNÉES 66

8.6. TEST UNITAIRE 68

9. LE FRONTEND 71

9.1. NUXTJS 71

9.1.1. Store 72

9.1.2. Internationalisation 72

9.2. VUETIFY 73

9.3. COMPOSANT MONOFICHIER VUE.JS 73

9.4. STRUCTURE 75

9.5. LE CONTENEUR 76

9.5.1. Dockerfile de développement 76

9.5.2. Dockerfile de production 76

9.6. L'INTERFACE GRAPHIQUE 77

10. RECOMMANDATIONS 78

10.1.1. Backend 78

10.1.2. Frontend 78

11. PRINCIPALES DIFFICULTÉS RENCONTRÉES 79

12. CONCLUSION 80

RÉFÉRENCES 82

ANNEXE 1 – STRUCTURE DES GROUPES ET DES DÉPÔTS GIT SUR GITLAB 85

ANNEXE 2 – DERNIÈRE VERSION DES MOCKUPS 87

ANNEXE 3 - GITLAB-CI.YML **88**

ANNEXE 4 - STRUCTURE DES RÉPERTOIRES DE LA CLÉ USB **91**

ANNEXE 5 – DOCUMENTATION TECHNIQUE **92**

DÉCLARATION DE L'AUTEUR **95**

Liste des tableaux

Tableau 1 : Comparaison des langages backend 8
Tableau 2 : Express.js – Avantages et inconvénients 9
Tableau 3 : Laravel – Avantages et inconvénients 10
Tableau 4 : Django REST Framework – Avantages et inconvénients.....10
Tableau 5 : Flask – Avantages et inconvénients 11
Tableau 6 : Ruby on Rails – Avantages et inconvénients 12
Tableau 7 : Comparatif des frameworks backend.....13
Tableau 8 : Angular - Avantages et inconvénients 16
Tableau 9 : React – Avantages et inconvénients 17
Tableau 10 : Vue.js – Avantages et inconvénients 18
Tableau 11 : Comparatif des frameworks frontend.....18
Tableau 12 : Comparatif de différentes fermes d'appareils mobiles 25
Tableau 13 : Communication SSH – Avantages et inconvénients 38
Tableau 14 : Communication websocket – Avantages et inconvénients.....39

Liste des figures

Figure 1 : Méthodologie Scrum 2
Figure 2 : Exemple d'un burndown chart 4
Figure 3 : Différence entre synchrone et asynchrone 7
Figure 4 : Smartphones dans un datacenter 19
Figure 5 : Boucle DEVOPS 20
Figure 6 : Résultat d'un test sur Firebase Test Lab 22
Figure 7 : Interface web de Kubiton 23
Figure 8 : Caméra lors d'une session de test sur Kubiton 24
Figure 9 : Diagramme de cas d'utilisation de Genica 28
Figure 10: Imbrication des dépôts Git 29
Figure 11: Exemple d'images de test 31
Figure 12: Exemple d'une définition d'un set de données 31
Figure 13 : Exemple d'un résultat d'un benchmark 32
Figure 14: BPMN de test actuel pour le SDK Smartscan 33
Figure 15 : Mockup du nouveau processus de test 34
Figure 16: BPMN pour l'automatisation des tests des SDK Smartscan 36
Figure 17 : Communication message broker – Avantages et inconvénients 39
Figure 18 : Mockup version 1.0 41
Figure 19 : Mockup version 1.1 42
Figure 20 : Mockup version 1.2 42
Figure 21 : Mockup version 1.3 43
Figure 22 : Interface réelle 43
Figure 23 : Différence entre des conteneurs et des machines virtuelles 44
Figure 24 : Schéma basique de l'architecture IT 45
Figure 25 : Schéma de l'architecture finale 48
Figure 26 : Structure du projet 50
Figure 27: Branching Git 51
Figure 28 : Pipelines exécutés 52
Figure 29 : Gitlab-ci – Étape test 53
Figure 30 : Gitlab-ci – Extrait de l'étape build 54
Figure 31 : Gitlab-ci – Extrait de l'étape deploy 55
Figure 32 : Structure du répertoire « backend » 56

Figure 33 : Extrait de code de celery.py 57
Figure 34 : Extrait de code de urls.py 58
Figure 35 : Extrait de code de delivery/serializers.py 61
Figure 36 : Extrait de code de delivery/views.py 61
Figure 37 : Diagramme de séquence de la génération du SDK smartscan Android63
Figure 38 : Diagramme de séquence de livraison d'un SDK smartscan Android...65
Figure 39 : Schéma relationnel de la base de données.....67
Figure 40 : Extrait de code de delivery/tests/utils_tests/test_git_tool. py 68
Figure 41 : Exécution et résultat des tests unitaires 69
Figure 42 : Rapport de couverture 69
Figure 43 : Rapport de couverture HTML.....70
Figure 44 : Store errorNotification.js 72
Figure 45 : Extrait du fichier nuxt.config.js 72
Figure 46 : Exemple d'un composant Vue.js 73
Figure 47 : Structure du dossier « frontend ».....75
Figure 48 : Interface réelle.....77

Liste des abréviations

API	Application Programming Interface
BPMN	Business Process Model and Notation
CD	Déploiement Continu
CI	Intégration Continue
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DB	Base de données
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
JDK	Java Development Kit
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript Syntax eXtension
LAN	Local Area Network
LDAP	Lightweight Directiry Access Protocol
MVC	Model Vue Controller
NAS	Network Attached Storage
NDK	Native Development Kit
ORM	Object Relationnel Model
OS	Système d'exploitation
PHP	Hypertext Preprocessor
PO	Product Owner
REST	Representational State Transfer
SDK	Software Development Kit
SPA	Single Page Application
SSH	Secure Shell
US	User Story
VM	Machine Virtuelle
WSGI	Web Server Gateway Interface

Introduction

À l'heure actuelle, l'institut Icare conçoit un certain nombre de décodeurs fonctionnant principalement sur des smartphones et tablettes Android ou iOS. L'augmentation du nombre de décodeurs et de clients rend le processus de compilation, test, distribution, suivi des licences et mises à jour de plus en plus difficiles. Cette complexité est en partie due au fait qu'un client peut choisir une combinaison quelconque de décodeurs (chaque décodeur ayant son propre numéro de version). Ils sont ensuite compilés dans un seul SDK. La configuration de ces décodeurs, la génération des SDK, le test et la distribution se font manuellement.

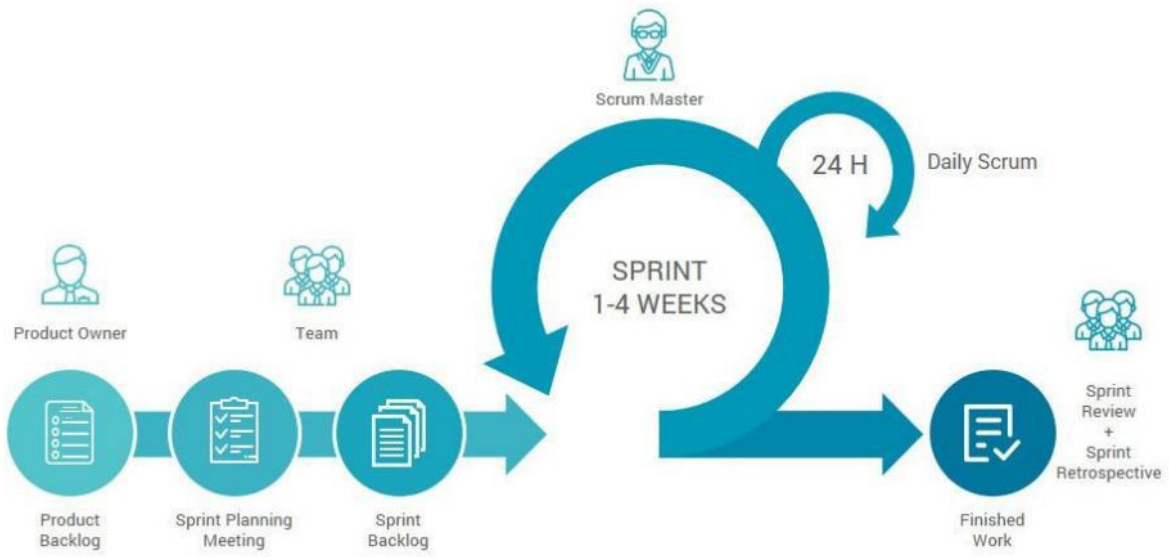
L'objectif de ce sujet est d'analyser l'état actuel, de proposer une architecture IT et un moyen d'automatisation du processus de release des décodeurs. La génération des SDK Android et iOS doit être facilité à l'aide d'une plateforme web. Un accent est mis sur la génération de SDK smartscan Android. Le suivi du développement se base sur les outils de la méthodologie Scrum.

Pour réaliser ce travail, il est pertinent de découper le projet en différents thèmes. Afin de partir sur de bonnes fondations, la méthodologie et l'organisation du travail sont définies dans le chapitre 1. Il est directement suivi par les différents états de l'art portant sur les sujets suivants : framework backend, framework frontend et les plateformes de test mobile avec des appareils réels. Les besoins de la plateforme sont traités dans le chapitre 3 afin de les analyser ainsi que l'existant dans le chapitre 4. Une fois tout passé ces chapitres, le travail s'occupe principalement de la plateforme web. Les mockups sont décrits en 5, l'architecture IT de la plateforme est conçue en 6. Les chapitres 7 et 8 parlent du développement et finalement les conclusions du projet en 9.

1. Méthodologie

1.1. Scrum

FIGURE 1 : MÉTHODOLOGIE SCRUM



Source : https://hsto.org/getpro/habr/post_images/d2a/90d/251/d2a90d2514c3f9649b0c2005730e872c.jpg

Scrum est un processus itératif particulièrement adapté aux projets de programmation. Il met au centre de l'attention le produit et non pas le projet. Le client est intégré durant tout l'avancement du projet. Scrum est principalement basé sur les points suivants :

- L'équipe ;
- Les cérémonies ;
- Les artefacts ;

1.1.1. L'équipe

L'équipe a une place importante et est constituée autour de trois rôles :

- **Product Owner (PO)** : c'est la personne possédant la vision du produit ;
- **Scrum master** : ce n'est pas le chef de projet, mais il aide l'équipe à résoudre les obstacles et veille à l'application de la méthodologie ;
- **L'équipe de développement** : est auto-organisée ;

(Subra, 2019)

1.1.2. Les cérémonies

Selon le manifeste scrum, il existe cinq cérémonies :

- **Le sprint** (ou itération) : c'est une période de courte durée variant de quelques jours à quatre semaines selon les besoins. Ces sprints peuvent être vus comme un petit projet apportant de nouvelles fonctionnalités. Il fixe un but à atteindre dans un court terme ;
- **Le sprint planning** : il s'agit d'une réunion de l'ensemble de l'équipe pour préparer un sprint définissant l'objectif et les tâches à réaliser ;
- **Le daily meeting** : désigne une petite réunion de l'équipe de développement d'une durée de 15 minutes. Elle permet la bonne synchronisation sur les tâches réalisées et celle effectuée ce jour-là ;
- **Le sprint review** : c'est une réunion de l'ensemble de l'équipe autour d'une démonstration du délivrable fourni en fin de sprint ;
- **Le sprint rétrospectif** : il s'agit d'une réunion de l'ensemble de l'équipe permettant d'analyser le déroulement du sprint. Les membres de l'équipe ont la possibilité de s'exprimer sur leur ressenti ;

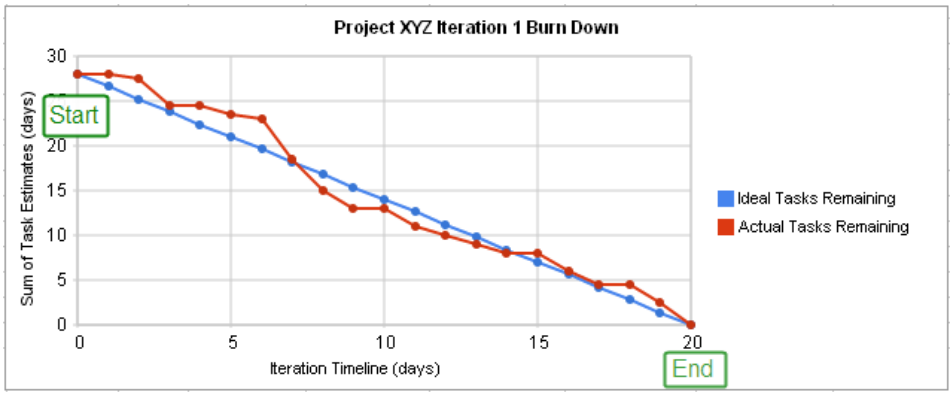
(Subra, 2019)

1.1.3. Les artefacts

Ci-dessous, voici les principaux artefacts de scrum :

- **Le product backlog** : contient la vision du projet sous forme d'User Story (US). Ces US sont définis par un rôle (en tant que), un souhait (je souhaite que) et des critères d'acceptance ;
- **Le sprint backlog** : contient la liste des US à réaliser durant le sprint. L'équipe de développement les divise en tâches pour faciliter le suivi sur le tableau de sprint (kanban) ;
- **Le burndown chart** : c'est un graphique montrant l'avancement du sprint selon une courbe idéale et réelle des tâches restantes. Si la courbe réelle est dans la partie inférieure, cela signifie que l'équipe est en avance. Autrement, l'équipe est en retard ;

FIGURE 2 : EXEMPLE D'UN BURNDOWN CHART



Source : https://fr.wikipedia.org/wiki/Burndown_chart#/media/Fichier:Burn_down_chart.png

(Subra, 2019)

1.1.4. Méthodologie appliquée au projet

Étant donné que ce travail est individuel et que la méthodologie scrum se base sur une équipe, il est nécessaire de faire quelques adaptations. Seul de rôle de PO est retenu et est représenté par Benoît Golay de l'institut Icare. Concernant les cérémonies, la durée des sprints est fixée à trois semaines et les daily meetings ne sont pas effectués.

1.2. Méthodologie pour la documentation

La documentation se fait continuellement et parallèlement aux autres étapes. L'accent est mis spécialement en fin de projet notamment pour la rédaction des parties concernant le développement.

1.3. Méthodologie pour les mockups

Les mockups sont réalisés au début du projet sous forme d'itération. Une fois la première version terminée, elle est présentée au PO. Une deuxième itération est effectuée pour corriger les écrans selon les remarques et souhaits de l'institut Icare. Bien entendu, le design n'est pas gravé dans le marbre et évoluera sans doute.

Cette étape importante permet au mandant de visualiser la future application tout en s'assurant qu'elle réponde aux besoins. Pour le développeur, elle permet d'avoir une vision d'ensemble de chaque écran à réaliser ce qui facilite l'estimation du travail à effectuer.

Lors de ce projet, les mockups sont réalisés à l'aide de Figma. Une application en ligne pour concevoir efficacement des mockups. Il est possible de définir des liens afin de simuler les futurs actions. Ceci permet d'améliorer l'expérience utilisateur.

1.4. Méthodologie de l'état de l'art

Les états de l'art concernant la conception du produit, tel que le choix du framework frontend ou backend, sont réalisés avant le début du sprint 0. Ceci afin de connaitre les technologies utilisées. Ces dernières sont mises en place lors de ce premier sprint. Ces chapitres sont finalisés lors du développement.

À propos de l'état de l'art des différentes plateformes de test mobile utilisant des appareils réels, il est réalisé lors du développement. Cela permet de commencer la conception de la plateforme au plutôt tout en évitant certaines surprises, notamment des délais trop courts.

1.5. Méthodologie pour le développement

Le développement de la plateforme se fait sous forme d'itération de trois semaines. La méthodologie utilisée est similaire à scrum, mais allégée pour répondre au besoin d'un travail individuel (décrit au chapitre 1.1.4).

Les différentes parties de l'application web sont testées. Le backend par des tests unitaires, le but étant d'avoir un taux de couverture proche des 100%. Quant au frontend, il sera uniquement testé de façon manuelle.

2. États de l'art

Afin de mener à bien ce travail, trois états de l'art sont réalisés sur les points suivants :

- Choix du framework frontend ;
- Choix du framework backend ;
- Comparatif des différentes plateformes de test mobile avec des appareils réels ;

2.1. Choix du framework backend

Il existe de nombreux langages de programmation et une multitude de frameworks pour réaliser une application web. Il est donc important de faire un premier tri et de sélectionner des langages/frameworks pouvant répondre aux besoins du projet. Le but principal du backend est d'offrir des services de génération de SDK Android et iOS via une API REST. Le langage doit être capable de :

- Faciliter la programmation web ;
- Interagir avec une base de données (DB) ;
- Servir une API REST ;
- Interpréter des scripts Python ;
- Exécuter des tâches d'automatisation ;
- Exécuter sur divers systèmes d'exploitation (OS) (multiplateforme) ;

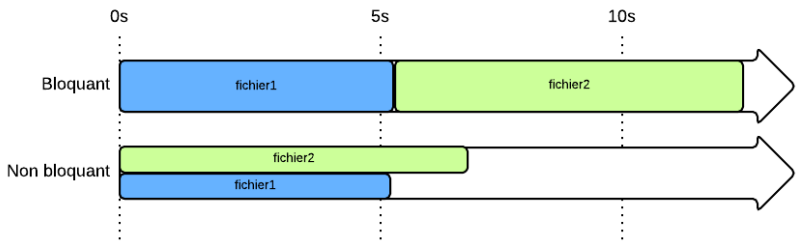
Évidemment, la majorité des langages peuvent correspondre à ces critères. Mais il faut faire un choix. Celui-ci s'est porté sur les langages suivants :

- Node.js ;
- PHP ;
- Python ;
- Ruby ;

Node.js n'est pas à proprement parler un langage, mais une technologie. « La description la plus exacte serait de dire que Node est un interprète JavaScript (JS) associé à un ensemble de bibliothèques permettant de réaliser des actions » (Fontanet & Lambert, 2015). L'une des autres forces de Node.js est son côté

asynchrone qui ne bloque pas l'exécution du programme. Node n'attend pas la fin de l'instruction pour passer à la suivante. Contrairement à un langage synchrone qui exécute qu'une instruction à la fois.

FIGURE 3 : DIFFÉRENCE ENTRE SYNCHRONE ET ASYNCHRONE



Source : https://openclassrooms.com/fr/courses/1056721-des-applications-ultra-rapides-avec-node-js/1056866-node-js-mais-a-quoi-ca-sert

PHP est un langage utilisé pour générer des pages HTML dynamiquement sur un serveur. Le résultat est ensuite envoyé au client qui lui n'a pas connaissance du traitement réalisé. C'est un des langages récurant dans le monde du développement web. « À ce jour, les analystes estiment que PHP est utilisé par plus de 80 % des sites web dans le monde (en nombre de domaines) » (Heurtel, 2019).

« **Python** est un langage créé pour être simple. Il s'oriente autour d'une philosophie qui donne des directions très limpides tout en restant extrêmement souple » (Chazallet, 2019). Python peut être utilisé pour automatiser des tâches système (au même titre que le Bash), écrire un algorithme de machine learning, créer un site web ou développer diverses applications.

Ruby est un langage orienté objet, lisible et facile à apprendre. Ruby est utilisé pour développer des applications web, automatisé des tâches système et divers autres développements. « Ruby is a dynamic programming language with a complex but expressive grammar and a core class library with a rich and powerful API » (Flanagan & Matsumoto, 2008).

TABLEAU 1 : COMPARAISON DES LANGAGES BACKEND

	Node.js	PHP	Python	Ruby
Programmation web	✓	✓	✓	✓
Interagir avec un DB	✓	✓	✓	✓
Servir une API REST	✓	✓	✓	✓
Asynchrone	✓	✗	✗	✗
Utilisé pour automatiser des tâches d'administration système	(✓)	(✓)	✓	✓
Linux, Mac et Windows	✓	✓	✓	✓
WebSockets	✓	✓	✓	✓

Source : Tableau de l'auteur provenant de sources multiples

Ci-dessous se trouve la sélection des frameworks retenus parmi les langages cités précédemment. Bien entendu, ce n'est qu'une sélection tant l'ensemble est grand et en constante évolution.

- Express.js ;
- Laravel ;
- Django REST Framework ;
- Flask ;
- Rails ;

2.1.1. Express.js

« Fast, unopinionated, minimalist web framework for Node.js »
(Express - Node.js web application framework, 2017)

Express.js est basé sur la technologie Node.js, elle-même basée sur le JavaScript. C'est un langage généralement utilisé dans les librairies ou frameworks frontend. Depuis l'arrivée de Node.js, il peut s'exécuter sur un serveur. Il est maintenant possible d'avoir le même langage pour le backend et le frontend. (Express/Node introduction, 2020)

Ce microframework créé par la fondation Node.js est utilisé par de nombreux développeurs. Express.js est rapide, car JS est un langage asynchrone. Il peut exécuter plusieurs instructions simultanément, mais il n'est pas possible de créer des threads supplémentaires. C'est un framework offrant beaucoup de liberté, il est simple de mal concevoir son application ce qui complique sa maintenabilité. (Express/Node introduction, 2020)

Le JavaScript n'est pas utilisé pour automatiser des tâches d'administration système, mais il est possible de lancer des actions sur le système. (Tesanovic, 2018)

TABLEAU 2 : EXPRESS.JS – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• JavaScript ; • Asynchrone ; • Rapide ;	• Beaucoup de liberté ; • Pas de multithreading ; • JavaScript n'est pas utilisé pour l'automatisation de tâches d'administration système ;

Source : Tableau de l'auteur provenant de sources multiples

2.1.2. Laravel

« The PHP Framework for Web Artisans » *(Laravel - The PHP Framework for Web Artisans, s.d.)*

Laravel est l'un des frameworks PHP les plus populaires. Avec l'une des communautés les plus actives. Il dispose d'un écosystème puissant avec une ligne de commande, un gestionnaire de paquets, un Object Relationnel Model (ORM) performant et une librairie pour les tests unitaires native. (Laravel - The PHP Framework for Web Artisans, s.d.) Toutefois, Laravel a beaucoup de dépendance et est de ce fait un framework volumineux.

C'est d'ailleurs pour cela que le créateur de Laravel a créé Lumen. Lumen est un microframework, un Laravel amaigri pour ne laisser que le cœur. Ce framework offre énormément de liberté et le choix des librairies (notamment pour l'accès DB). Ce n'est pas ce qui est recherché pour ce projet. (Lumen - PHP Micro-Framework By Laravel, s.d.)

Ces deux frameworks sont faits avec PHP, un langage utilisé dans de nombreux sites web. Néanmoins PHP était un langage avec énormément de failles de sécurités, une conception hasardeuse et peu performant. En revanche, ces dernières années, avec PHP 7.0, le langage s'est nettement amélioré. (Brent, 2019)

TABLEAU 3 : LARAVEL – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Bon écosystème ; • Documentation ; • Communauté active ;	• Performance ; • Apprentissage lent ; • Beaucoup de dépendances ;

Source : Tableau de l'auteur provenant de sources multiples

2.1.3. Django Rest Framework

> « Django REST framework is a powerful and flexible toolkit for building Web APIs » *(Home - Django REST framework, s.d.)*

Comme son nom l'indique, ce framework Python se base sur Django ce qui apporte un bon nombre de fonctionnalités. Le site de Django REST Framework fournit un certain nombre de tutoriels ainsi qu'une bonne documentation. Les fonctionnalités suivantes sont intégrées :

- Intégration de l'authentification facilitée ;
- Sérialisation pour stocker des données dans une DB ;
- Interface web d'administration ;
- API web navigable en mode développement ;

(Home - Django REST framework, s.d.)

TABLEAU 4 : DJANGO REST FRAMEWORK – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Python ; • API navigable ; • Documentation ; • Communauté active ;	• Nécessite Django (dépendance) ;

Source : Tableau de l'auteur provenant de sources multiples

2.1.4. Flask

> « Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications » *(Flask | The Pallets Project, s.d.)*

Flask est un microframework web Python. Pour une première utilisation, ce framework peut être risqué, car il n'offre pas de structure concrète. Les erreurs de conception arrivent facilement. Toutefois, il existe un bon nombre de tutoriels et la documentation est bien construite. (Flask | The Pallets Project, s.d.)

TABLEAU 5 : FLASK – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Python ; • Léger ; • Documentation ;	• Trop de liberté ;

Source : Tableau de l'auteur provenant de sources multiples

2.1.5. Ruby on Rails

> « Ruby on Rails makes it much easier and more fun. It includes everything you need to build fantastic application » *(Ruby on Rails | A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern., s.d.).*

Rails est le framework web pour le langage Ruby. C'est probablement un des frameworks web les plus aboutis. Il intègre un grand nombre de fonctionnalités. Si ce n'est pas le cas, une Gem le fait. Une Gem est le nom d'une librairie Ruby. En contrepartie, ce framework peut paraître magique, car un certain nombre de tâches sont faites à votre place. L'on considère ce framework comme une boite noire. Mais dès que Rails a été dompté, les tâches deviennent limpides. Cependant, le déploiement est compliqué. La communauté autour de Ruby on Rails est grande en dépit de la baisse de popularité ces dernières années. (Ruby on Rails | A web-

application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern., s.d.)

TABLEAU 6 : RUBY ON RAILS – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Ruby ; • Structure ; • Unit test intégré de base ; • Communauté active ;	• Considéré comme une boite noire ; • Performance ; • Popularité décroissante ;

Source : Tableau de l'auteur provenant de sources multiples

2.1.6. Prise de décision pour le backend

Ces frameworks sont différents et chacun d'eux est optimisé pour certaines tâches. Prenons Ruby on Rails, c'est un framework très efficace pour réaliser une application web rapidement avec une bonne architecture. En revanche, certaines tâches se font par enchantement, car Ruby on Rails le fait pour le développeur. Laravel, lui, offre un bon environnement de développement (ligne de commande, gestion de paquet PHP …). Cependant, il se base sur PHP qui est un langage moyennement lisible.

La décision du framework est prise selon la matrice ci-dessous évaluée par différents critères de 1 à 6 :

- **6** : correspond en tout point ;
- **4** : Remplit la demande ;
- **2** : Inadapté ;
- **1** : Hors sujet ;

TABLEAU 7 : COMPARATIF DES FRAMEWORKS BACKEND

	Express.js	Laravel	Django REST Framework	Flask	Ruby on Rails
Structure	3	5	5	3	6
Lisibilité	5	3	6	6	5
Automatisation de tâches système	2	4	6	6	5
API REST	6	5	6	6	5
Liberté	6	4	4	6	4
Test unitaire possible	5	6	5	5	6
Accès à la base de données via une ORM	5	6	5	5	6
Performance	6	3	5	5	4
Moyenne	***4.9***	***4.5***	***5.3***	***5.3***	***5.1***

Source : Tableau de l'auteur provenant de sources multiples

Ce comparatif montre que les frameworks Python sont les solutions les plus adaptées au projet. Django REST Framework et Flask obtiennent la meilleure note. Python est un langage couramment utilisé pour l'automatisation de tâches système. En outre, les scripts existants pour la génération des SDK sont écrits en Python. Ce sont également les frameworks avec le moins de points négatifs. D'ailleurs Python est natif (installé de base) sur la majorité des distributions Linux ainsi que sur macOS. En revanche, il est difficile de les départager, car ces deux frameworks obtiennent une note identique.

Ces frameworks sont différenciés sur la création d'une API de test permettant d'afficher des tâches. Cette démonstration permet de mieux comprendre les différences entre ces deux frameworks.

L'on trouve la démonstration Flask sur Github, les deux tutoriels suivants sont utilisés :

- https://dev.to/paurakhsharma/flask-rest-api-part-2-better-structure-with-blueprint-and-flask-restful-2n93
- https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask

Et voici l'application développée avec Django Rest Framework que l'on retrouve ici. Ci-dessous les documentations utilisées :

- https://www.django-rest-framework.org/tutorial/quickstart/
- https://www.django-rest-framework.org/tutorial/1-serialization/#creating-a-model-to-work-with

Les deux démonstrations se trouvent également dans le répertoire `/backend_evaluation` de la clé USB. Pour conclure, Flask offre beaucoup de liberté et peut par conséquent être source d'erreurs lors de la conception. Il est possible de structurer l'application de différentes façons. Ce n’est pas le cas avec Django REST. Ce framework met en place une structure dès le début. **Django REST est donc retenu**. Ce framework offre différents avantages lors de la conception, notamment la gestion d'une DB simple, l'authentification et des API navigables en mode développement.

2.2. Choix du framework frontend

Cette analyse permet de choisir l'outil le plus adapté pour créer l'interface web de la plateforme et non sur l'utilisation de ces frameworks pour la création d'application pour smartphone. Les trois principaux frameworks qui sont analysés sont :

- Angular ;
- React ;
- Vue.js ;

2.2.1. Angular

Angular est un framework complet développé et maintenu par Google. Il est utilisé pour la création d'application par composant. Il est difficile de l'intégrer à un projet web existant. Mais, la création de nouvelles applications reste relativement simple, car Angular fournit nativement le nécessaire pour :

- La configuration de build ;
- Le routage ;
- La gestion des états « state manager » ;
- La validation de formulaire ;
- Le débogage ;

(Angular, s.d.)

Angular est basé sur une architecture par composants. La création de ceux-ci se fait avec du code HTML, du CSS et du TypeScript. Ce dernier est un langage compilé et open source semblable au JavaScript. Il a été créé par Microsoft et permet un typage des variables ou des fonctions afin d'améliorer et de sécuriser la production du code. C'est un des points qui poussent les entreprises à utiliser Angular. (AltexSoft, 2020). Il est également possible de créer des applications natives pour smartphone. (Angular, s.d.)

TABLEAU 8 : ANGULAR - AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Approche MVC ; • Architecture basée sur des composants ; • Structuré ; • Écosystème puissant ;	• Courbe d'apprentissage difficile ; • Complexe ;

Source : Tableau de l'auteur provenant de sources multiples

2.2.2. React

React est une librairie open source pour la création et le rendu de composants. Cette dernière est principalement développée et maintenue par Facebook. Cette librairie peut être utilisée dans une application web existante ou pour créer une Single Page Application (SPA) moyennant l'utilisation d'autres librairies non développées par le React core team. Elles doivent être rajoutées manuellement au projet :

- Un « router » (React-router) ;
- Un « state manager » (Redux) ;

React est basé sur une architecture par composants. Ils sont entièrement développés en JavaScript. Par ailleurs, la création du HTML se fait soit en JavaScript ou en JSX. JSX est une extension permettant de garder la puissance du JavaScript pour créer l'interface.

React est très populaire auprès des développeurs et des entreprises. Ceci étant donné qu'il s'intègre facilement au site existant et offre une énorme flexibilité (parfois trop grande) pour la création de SPA. Il est également possible de créer des applications natives pour smartphone à l'aide de React Native.

(AltexSoft, 2018)

TABLEAU 9 : REACT – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Architecture basée sur des composants ; • DOM virtuel ; • Compact ; • Flexible ; • Évolutif ;	• Courbe d'apprentissage (JSX) ; • C'est une librairie ; • Flexibilité parfois trop grande ;

Source : Tableau de l'auteur provenant de sources multiples

2.2.3. Vue.js

Vue.js est un framework progressif mélangeant les styles de React et d'Angular. Ce Framework est développé par Evan You (un ancien ingénieur de Google) en 2013. D'autres membres se sont vite rajoutés au projet pour créer la Vue.js core team.

Contrairement à Angular, Vue.js s'utilise dans des applications web existantes tout comme React. Cependant, il reste un framework et ne dispose pas de toutes les fonctionnalités d'Angular. De ce fait, Vue.js reste petit (en taille). Voici deux librairies fréquemment utilisées que le core team développe et maintient :

- Un « router » (Vue Router) ;
- Un « state manager » (Vuex) ;

Les composants Vue.js sont développés dans un seul fichier avec une extension `.vue`. Dans ces fichiers, l'on retrouve du code HTML, du CSS et du JavaScript. Ces dernières années, Vue.js gagne de plus en plus de partisans. Ce sont principalement des développeurs. Les entreprises quant à elles restent pour le moment sur ses deux concurrents. Comme pour Angular et React, il est également possible de créer des applications natives pour smartphone.

(AltexSoft, 2019)

TABLEAU 10 : VUE.JS – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Architecture basée sur des composants ; • Composant dans un fichier `.vue` unique ; • Compact ; • Lisibilité ; • Facile à apprendre ; • Documentation ;	• Framework jeune ; • Complexe pour de grande SPA ;

Source : Tableau de l'auteur provenant de sources multiples

2.2.4. Prise de décision pour le frontend

TABLEAU 11 : COMPARATIF DES FRAMEWORKS FRONTEND

	Angular	React	Vue
Lisibilité	5	5	5
Documentation	6	5	6
Écosystème	6	4	5
Flexibilité	4	6	5
Performance	5	6	5
Courbe d'apprentissage	4	4	5
Idéal pour petite SPA	4	5	5
Moyenne	***4.9***	***5.0***	***5.1***

Source : Tableau de l'auteur provenant de sources multiples

Ces trois frameworks/librairies sont de très bons choix et sont équivalents. Il n'y a pas de grand gagnant. Le choix dépend essentiellement du type de projet et des préférences des développeurs. L'institut Icare a déjà employé React dans certains projets. Mais au début du mois de mars 2020, l'institut Icare est en pleine réflexion sur les technologies qu'elle souhaite utiliser à l'avenir. De plus, Benoît Golay, le Product Owner (PO) laisse le choix de la technologie.

Selon le comparatif ci-dessus, **Vue.js semble être le framework le plus adapté à la situation.** Le temps d'apprentissage d'Angular et la trop grande flexibilité de React ont accentué ce choix. De plus, le focus doit être mis sur les processus de génération des décodeurs qui se situent au niveau du backend. En outre, Vue.js a certains avantages :

- Facile à apprendre ;
- Lisible (un fichier unique pour un composant) ;
- À une excellente documentation ;

2.3. Plateformes de test mobile avec des appareils réels

Les plateformes de test mobile avec des appareils réels fournissent un accès à des smartphones ou à des tablettes via une API ou un partage d'écran. Les appareils proposés sont en règle générale des smartphones réels et physiques. Mais cela peut, dans une moindre mesure, être un smartphone virtuel.

> *« Device farms are collections of various devices that can be accessed through a web interface. These devices can be accessed and tested over the web in a similar way how one would do testing on an actual device. » (Ravulavaru, 2017)*

FIGURE 4 : SMARTPHONES DANS UN DATACENTER

Source : https://firebase.google.com/docs/test-lab/

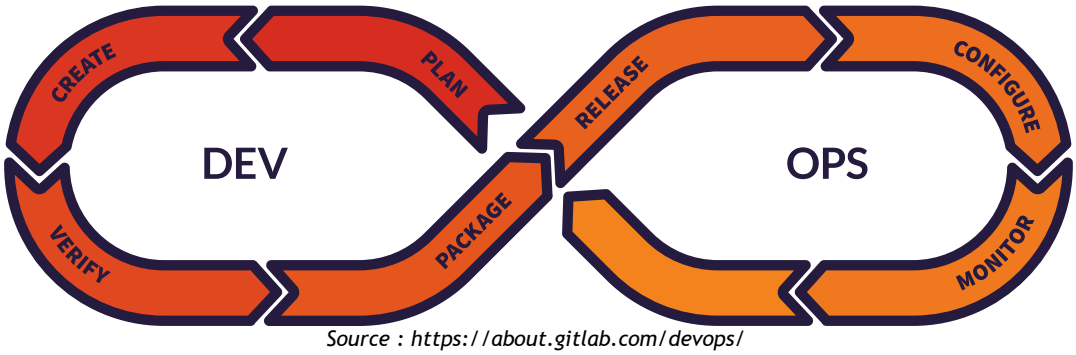
La demande de telles plateformes n'est pas nouvelle. Par le passé, Nokia a mis en place une plateforme similaire permettant de tester des applications sur leurs téléphones mobiles. Ces dernières années, le besoin de plateformes modernes a fortement augmenté. Ce n'est pas un constat étonnant vu la diversité des modèles, des surcouches Android des constructeurs et des différentes versions des systèmes d'exploitation (iOS et Android).

Il existe d'autres facteurs à cet engouement. Notamment le DevOps qui est de plus en plus présent dans le monde des entreprises. « L'approche DevOps est un vaste domaine qui est en pleine ébullition » (LEITE, 2018). Cette méthodologie tend à automatiser et faciliter les étapes entre la programmation et le déploiement via des :

- Intégration Continue (CI) ;
- Déploiement Continu (CD) ;
- Infrastructure as Code (IaC) ;

> *« DevOps culture is a set of practices that reduce the barriers between developers, who want to innovate and deliver faster, on the one side and, on the other side, operations, who want to guarantee the stability of production systems and the quality of the system changes they make. » (Krief, 2019)*

FIGURE 5 : BOUCLE DEVOPS



Source : https://about.gitlab.com/devops/

Dans une CI, il est intéressant d'avoir une étape exécutant des tests graphiques pour valider le plus grand nombre de smartphones. Sans ce type de plateforme, cette étape est difficile à mettre en œuvre.

Les fermes de smartphones évitent aux entreprises le maintien d'un parc de smartphones coûteux qui se renouvelle pratiquement tous les six mois. Elles permettent d'accéder facilement à différents smartphones depuis n'importe où et de façon automatisée. C'est pourquoi l'institut Icare cherche à s'informer sur ces plateformes. Ce type de plateforme a-t-elle une plus-value pour le test des décodeurs ou pour de futures applications Android ou iOS ? Les plateformes suivantes vont être analysées et comparées :

1. Firebase Test Lab (https://firebase.google.com/docs/test-lab/) ;
2. Kobiton (https://kobiton.com/) ;
3. AWS Device Farm (https://aws.amazon.com/ru/device-farm/) ;

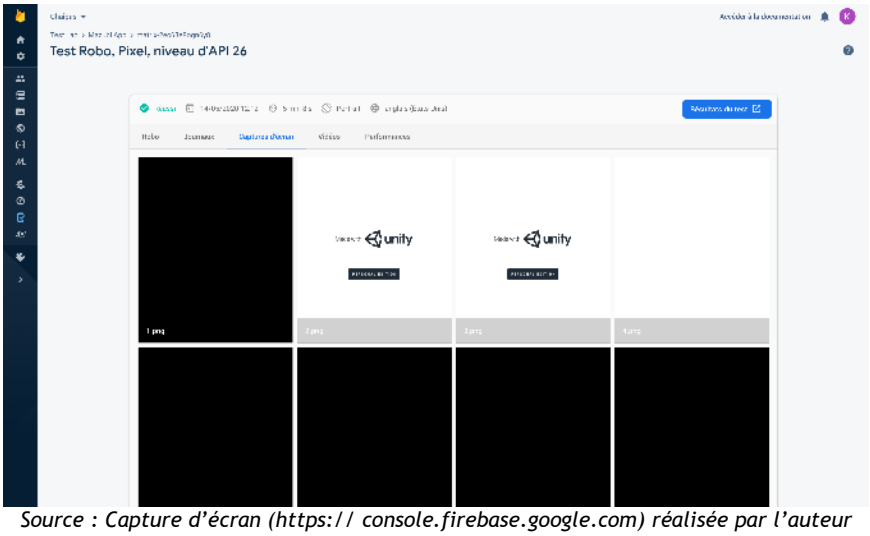
2.3.1. Firebase Test Lab

Firebase Test Lab est une infrastructure cloud permettant le test d'application mobile créée par Google. Cette plateforme utilise des smartphones ou des tablettes (Android et iOS) se trouvant dans des Datacenters de Google ainsi que quelques appareils virtualisés. Au total, ce sont 150 appareils réels qui sont fournis par ce service. Cet outil est disponible via :

- La console web de Firebase ;
- Android studio ;
- La ligne de commande Gcloud ;

La console web n'offre pas de partage d'écran, mais il est possible de télécharger une application pour lancer une session de test avec « Robo Tools ». Cet outil intégré dans Firebase permet d'exécuter des tests exploratoires. Le résultat de la session de test est disponible dans un rapport contenant les journaux, la vidéo et les captures d'écran prises à chaque action.

FIGURE 6 : RÉSULTAT D'UN TEST SUR FIREBASE TEST LAB



Source : Capture d'écran (https:// console.firebase.google.com) réalisée par l'auteur

Pour tester des cas d'utilisation spécifiques, il est possible de créer des scénarios avec les outils ci-dessous puis de les lancer via la console web ou en ligne de commande Gcloud.

- Espresso (Android) ;
- UI Automator 2.0 (Android) ;
- XCTest (iOS) ;

(Google, 2020)

Ce sont les tarifs de Firebase qui s'appliquent, car le Test Lab y est intégré. Il existe deux plans, un gratuit et un à l'utilisation.

- Gratuit :
  - Appareil virtuel : 10 tests par jour ;
  - Appareil physique : 5 tests par jour ;
- Paiement à l'utilisation :
  - Appareil virtuel : $1 l'heure par appareil ;
  - Appareil physique : $5 l'heure par appareil ;

(Google, s.d.)

2.3.2. Kobiton

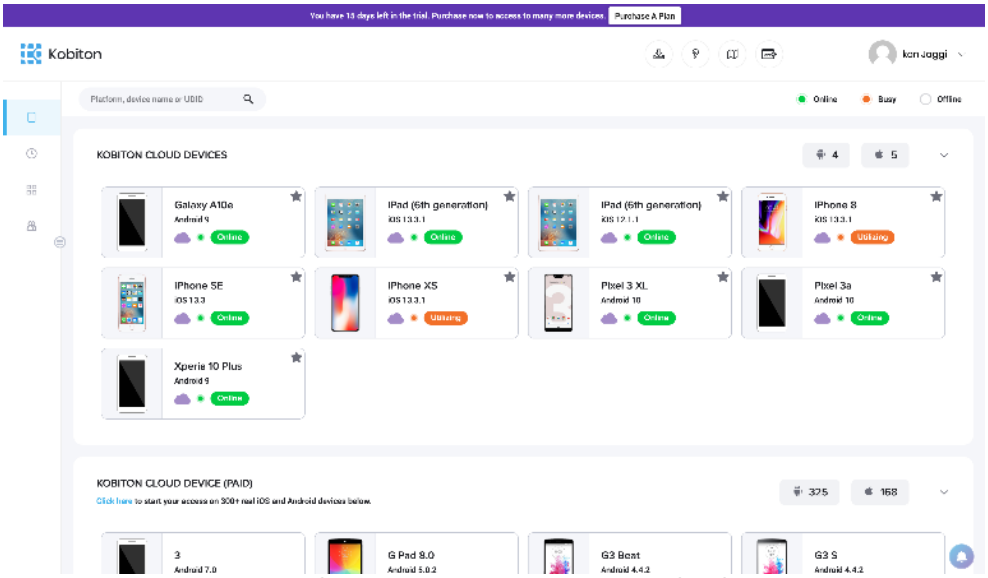
Kobiton est une plateforme dédiée au test mobile. Elle met à disposition plus de 350 smartphones et tablettes Android ou iOS sur le web. Elle a pour but d'automatiser les tests afin de limiter le temps pour la réalisation d'une version et le nombre d'appareils testés. Les smartphones et tablettes sont accessibles via :

- Un partage d'écran depuis la plateforme web ;
- Une API ;

Pour créer les scénarios afin d'automatiser les tests, il est possible d'utiliser Appium. C'est un outil pour automatiser les tests sur Android, iOS et Windows 10.

Il existe différents types de téléphones, la majorité est publique donc utilisable par tout le monde. Kobiton offre la possibilité d'avoir des smartphones dédiés (privés) ou d'ajouter des smartphones locaux à la plateforme. Bien entendu, ces appareils sont accessibles uniquement par le propriétaire.

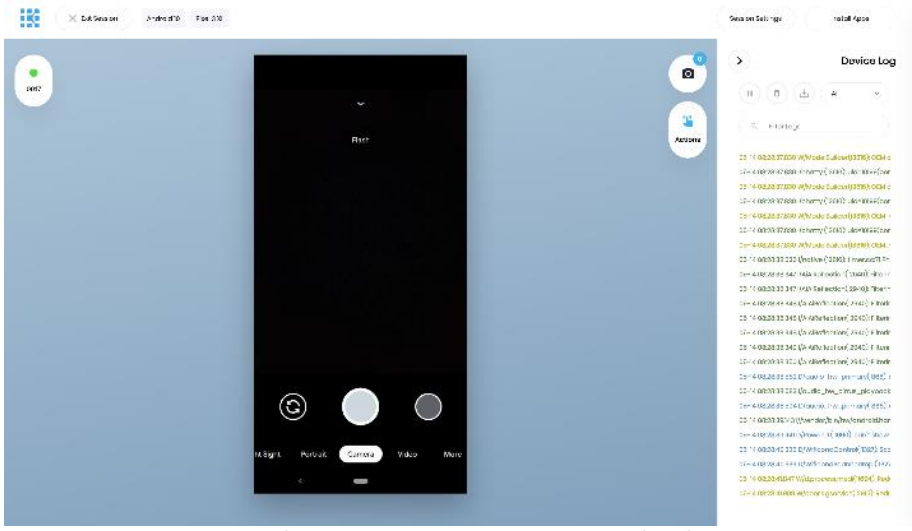
FIGURE 7 : INTERFACE WEB DE KUBITON



Source : Capture d'écran (https://portal.kobiton.com) réalisée par l'auteur

L'interface graphique est appréciable et intuitive. Il est simple de lancer une session de test manuel. Pendant ce test, l'on prend possession du téléphone par un partage d'écran. La caméra fonctionne parfaitement, néanmoins, l'objectif est recouvert. Un autre inconvénient est qu'il est impossible d'afficher un flux vidéo au capteur.

FIGURE 8 : CAMÉRA LORS D'UNE SESSION DE TEST SUR KUBITON



Source : Capture d'écran (https://portal.kobiton.com) réalisée par l'auteur

Il existe deux abonnements mensuels :

- « Team » au **prix de $40** permet d'obtenir 500 minutes de test ;
- « Organization » au **prix de $415** permet d'obtenir 5000 minutes de test ;

(Kobiton, s.d.)

2.3.3. AWS Device Farm

Amazon a également créé un service mettant à disposition des smartphones et des tablettes physiques Android et iOS. C'est la plateforme proposant le plus grand nombre d'appareils avec plus de 2500 appareils physiques. Ce service est accessible via :

- Le navigateur ;
- Une API ;
- Des plugins AWS ;

Depuis la plateforme web, il est possible de manipuler l'appareil via un partage d'écran et d'exécuter des tests manuels. Les scénarios de test automatisé peuvent être créés avec des logiciels comme Appium, Calabash et Espresso. Ces tests peuvent être exécutés via une CI. Autrement, il est possible d'effectuer un « Monkey test » qui est un test exploratoire. (Ravulavaru, 2017)

Les tarifs d'AWS Device Farm sont selon l'utilisation. Une minute de test sur un appareil coûte en moyenne $0,17. Il existe un abonnement illimité pour $250 par mois et la possibilité d'avoir des appareils privés pour un coût de $200 par mois. Cependant, AWS ne propose pas son service gratuitement et pour la version d'essai, il est obligatoire de renseigner une carte de crédit. (Amazon, s.d.)

2.3.4. Prise de position

TABLEAU 12 : COMPARATIF DE DIFFÉRENTES FERMES D'APPAREILS MOBILES

	Poids	Firebase Test Lab	Kobiton	AWS Device Farm
Test manuel	*	✗	✓	✓
Partage d'écran du mobile	*	✗	✓	✓
Accessible via une API	**	✓	✓	✓
Accessible via une ligne de commande	***	✓	✗	✓
Test exploratoire	**	✓	✗	✓
Possibilité de créer des scénarios	**	✓	✓	✓
Version gratuite	***	✓	✗	✗
Paiement à l'utilisation	**	✓	✗	✓
Test illimité	*	✓	✗	✓
Nombre d'appareils	*	150	350	2500
Android	***	✓	✓	✓
iOS	***	✓	✓	✓
Appareil privé	*	✗	✓	✓

Intégration à un CI	***	✓	✓	✓
Résultat		**24**	**18**	**26**

Source : Tableau de l'auteur provenant de sources multiples

Ce type de plateforme ne convient pas pour tester la communication entre la caméra et la librairie Smartscan. Car il est primordial de maitriser le flux vidéo de la caméra. Les objectifs de ces téléphones sont obstrués. Comme il est impossible de scanner des objets, les fermes d'appareils mobiles n'apportent pas de plus-value.

Cependant, ces plateformes ont des avantages pour d'autres types d'applications mobiles. En effet, il est possible d'automatiser les tests de l'interface graphique. Ce procédé permet de limiter les coûts tout en augmentant le nombre de smartphones testés. Deux plateformes se distinguent. Ce sont les plateformes d'Amazon et de Google.

Firebase Test Lab s'intègre facilement à l'environnement existant. Il n'est pas indispensable de créer des scénarios, mais il est possible d'utiliser Robo Tool qui réalise un test exploratoire. La version gratuite et la payante disposent des mêmes appareils. Les tarifs sont abordables et la version gratuite permet de tester la plateforme de Firebase.

AWS Device Farm n'a pas d'accès gratuit. Mais les tarifs sont abordables, permettant de tester une multitude de smartphones et offrant les fonctionnalités nécessaires. **C'est pour cela qu'AWS Device Farm prend la main par rapport aux autres plateformes**.

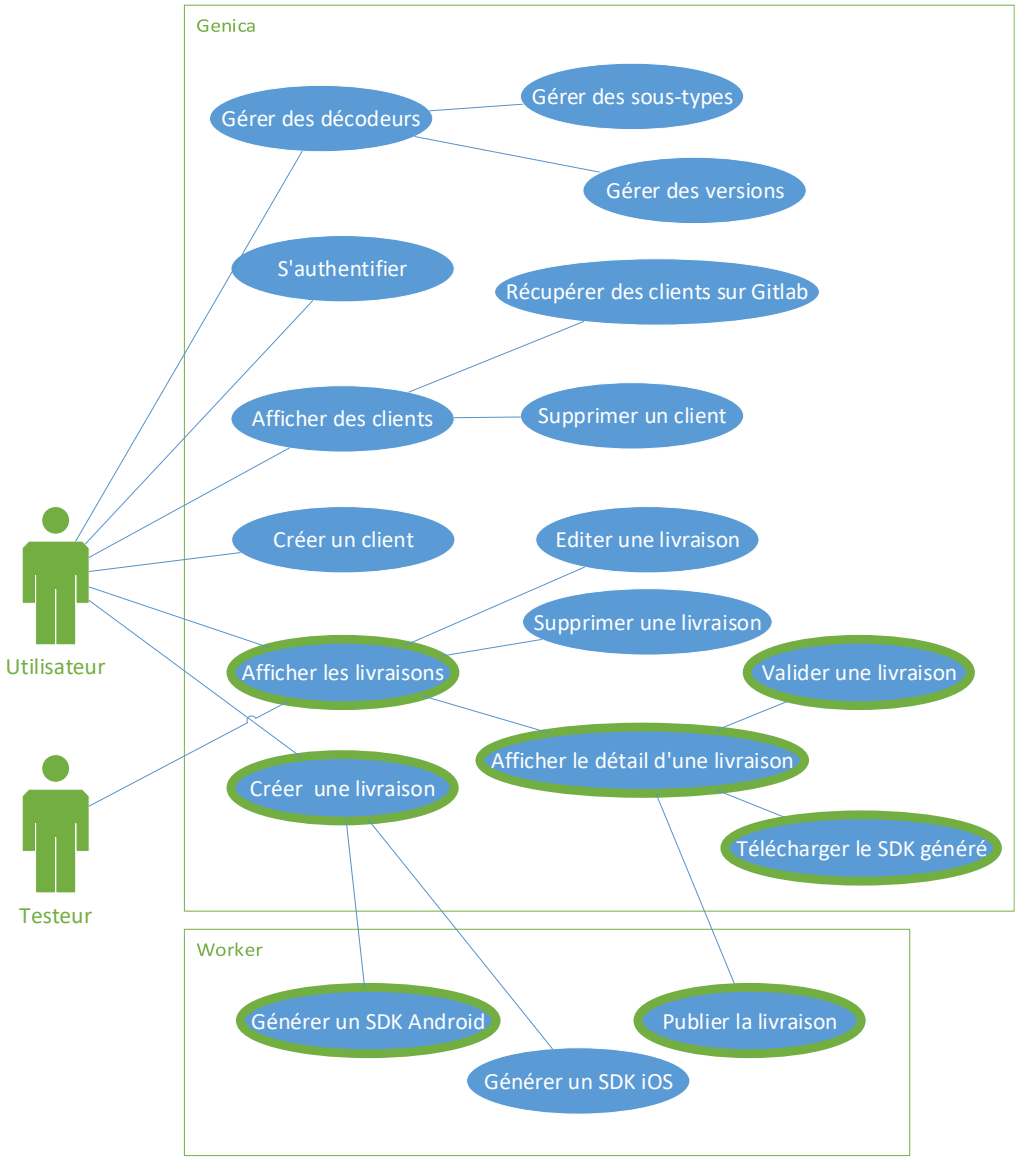
3. Besoins de la plateforme

La plateforme a pour but de faciliter la création ainsi que la livraison des décodeurs. Pour ce faire, l'interface web finale doit répondre à un certain nombre de critères permettant de limiter l'accès aux employés de l'institut Icare et d'utiliser les services existants de l'entreprise. Le site web reste uniquement accessible depuis le réseau interne (LAN) et l'authentification se fait avec le serveur Lightweight Directory Access Protocol (LDAP) présent. Les besoins de la plateforme sont de :

- Créer de nouvelles livraisons Android ou iOS et de suivre automatiquement le processus. Il passe de la création à la publication du SDK sur le serveur Gitlab. La procédure comprend la création, la génération du SDK, la validation, la gestion et la publication d'une livraison ;
- Afficher les détails d'une livraison et le téléchargement du SDK généré ;
- Se connecter avec les identifiants LDAP ;
- Mettre à jour la liste de clients de la plateforme selon Gitlab ;
- Gérer les décodeurs avec leurs sous-types et leurs versions ;
- Visualiser les actions effectuées ;
- Faciliter la validation des tests ;

Le diagramme de cas d'utilisation ci-dessous décrit au mieux les exigences de ce système de génération/suivi de livraison de décodeur ainsi que les rôles différenciés par leur interface. L'usager a le choix de changer sa fonction, en passant de la vue standard à la vue testeur.

FIGURE 9 : DIAGRAMME DE CAS D'UTILISATION DE GENICA



Source : Schéma réalisé par l'auteur

Cet ouvrage est vaste pour un projet individuel. Toutefois, ce projet analyse l'ensemble des processus. Cependant, tout ne sera pas implémenté, seulement une version minimale définie avec le PO. Elle comprend la procédure complète de livraison pour Android. Ce sont les cas d'utilisation avec la bordure verte. Les autres fonctionnalités, telles que le processus iOS, l'automatisation des tests des décodeurs et l'authentification, seront réalisées après le travail de bachelor.

4. Analyses

4.1. Structure des dépôts Git

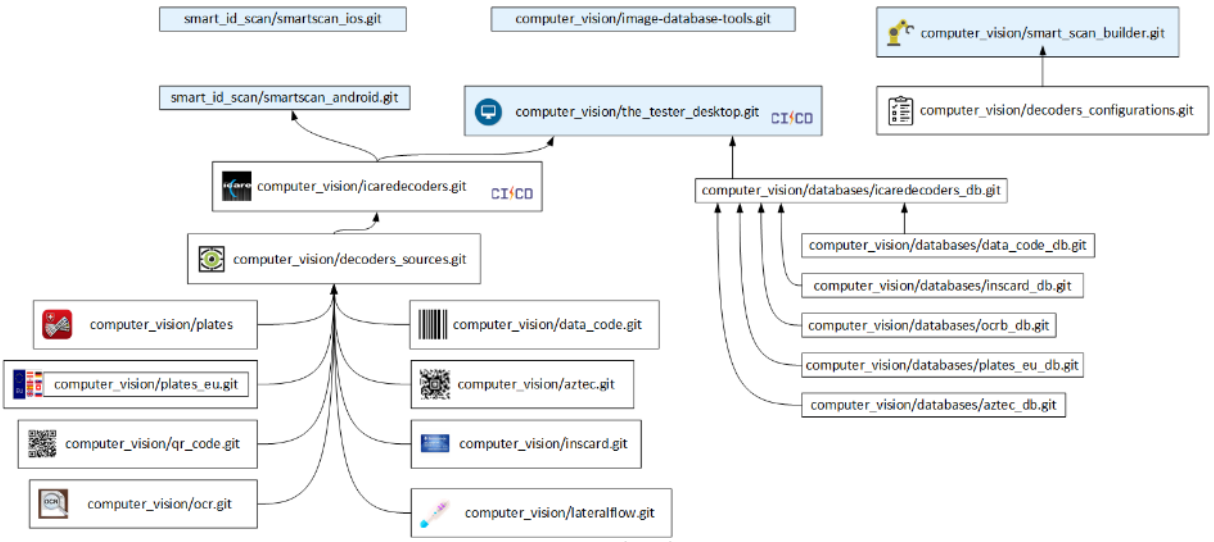
Les décodeurs et les utilitaires de génération ou de benchmarking sont regroupés dans différents groupes.

Le principal, « Computer vision » est composé de 35 dépôts et d'un sous-groupe « Databases ». Ce dernier contient plusieurs dépôts Git LFS contenant des images. Ils ont la vocation d'être utilisés comme des sets de données, mais leurs définitions sont manquantes. Les sets de données actuelles sont volumineux, utilisés en local et sauvegardés sur le Network Attached Storage (NAS) de l'institut Icare.

Le second groupe est « SmartID Scan », moins volumineux que le précédent. C'est ici que se trouvent les projets de construction des librairies (« smartscann_android » pour Android et « smartscann_ios » pour iOS) et les démonstrations pour les diverses plateformes.

Finalement, le dernier groupe « SmartscanReleases » contient les groupes pour les clients. C'est ici que les livraisons s'effectuent.

FIGURE 10: IMBRICATION DES DÉPÔTS GIT



Source : Infographie réalisée par l'auteur

Les projets sont fortement imbriqués. En bleu, ce sont les projets utilisés pour les tests ou pour la génération de la librairie Smart Scan. Tous les liens entre les dépôts représentent l'inclusion des sous-modules Git. La structure des groupes « Computer vision » et « SmarID Scan » sont plats et comportent énormément de dépôts visibles à l'annexe 1.

4.2. Analyse d'automatisation des tests sur le code source

4.2.1. Existant

Actuellement, il n'y a pas de tests unitaires pour le code source. Ceci est dû à une raison toute simple. Le code est difficilement testable de façon unitaire. Les décodeurs sont une suite de filtres pour améliorer la lisibilité, de traitements pour le découpage des caractères et une partie de Machine Learning pour certains décodeurs. Les développeurs jonglent en quelque sorte entre les différents paramètres. Certains fonctionnent mieux pour un certain type d'image alors que d'autres ne tournent pas.

C'est pour cela que le projet « The tester desktop » a été mis en place. C'est un dépôt central contenant divers sous-modules Git. C'est par ce projet que l'institut Icare évalue les nouvelles versions des décodeurs. Ce projet permet de réaliser un test de performance des décodeurs (QR Code, OCR, Inscard, Aztec, Data Code, Plates et Plates EU). Ensuite, il est analysé à l'aide du projet « Image database tools ».

(Hussain, 2020)

4.2.2. Proposition

Étant donné que les tests unitaires ne sont pas possibles sur ce type de projet, il est nécessaire d'avoir une bonne suite de régression pour « The testeur Desktop ». Cette suite doit être lancée par une CI à chaque nouveau commit la nuit. De ce fait, la CI se déclenchera rarement. De plus, la majorité du développement s'effectue dans les sous-modules.

Les sets de données utilisés dans cette suite de régression doivent encore être définis pour chaque décodeur. Une partie a déjà été mise en place par Christopher Finelli qui a regroupé des images par décodeur dans des dépôts Git LFS (dans le sous-

groupe databases). Mais un set de données doit également contenir le résultat attendu. Ces données sont stockées sous forme d'un fichier `.json` comme ci-dessous.

FIGURE 11: EXEMPLE D'IMAGES DE TEST

Source : https://gitlab.icare.ch

FIGURE 12: EXEMPLE D'UNE DÉFINITION D'UN SET DE DONNÉES

```
[
    {
        "image": "/path/to/image1",
        "result": "125997"
    },
    {
        "image": "/path/to/image2",
        "result": "896223"
    }
    ...
]
```

Source : fichier JSON réalisé par l'auteur

Ces images et le fichier de définition sont utilisés pour réaliser un benchmark. Lors de cette phase, les images sont décodées les unes après les autres. Le résultat obtenu est confronté à celui attendu dans le set de données. Cela permet de sortir un fichier comprenant un résumé et le résultat de chaque image. La dernière étape permet d'analyser le fichier de résultat et de le comparer aux précédents.

FIGURE 13 : EXEMPLE D'UN RÉSULTAT D'UN BENCHMARK


```
{
    "resume": {
        "total": 271,
        "success": 236,
        "fail": 25,
        "execution_time": "23.258s"
    },
    "result": [
        {
            "image": "/path/to/image1",
            "sucess": true,
            "result": "125997",
            "execution_time": "0.02348s"
        },
        {
            "image": "/path/to/image2",
            "sucess": false,
            "dataset_result": "896223",
            "result": "899283",
            "execution_time": "0.08348s"
        },
        ...
    ]
}
```


Source : fichier JSON provenant de l'auteur

Toutes ces phases doivent être lancées par la CI. Pour récapituler :

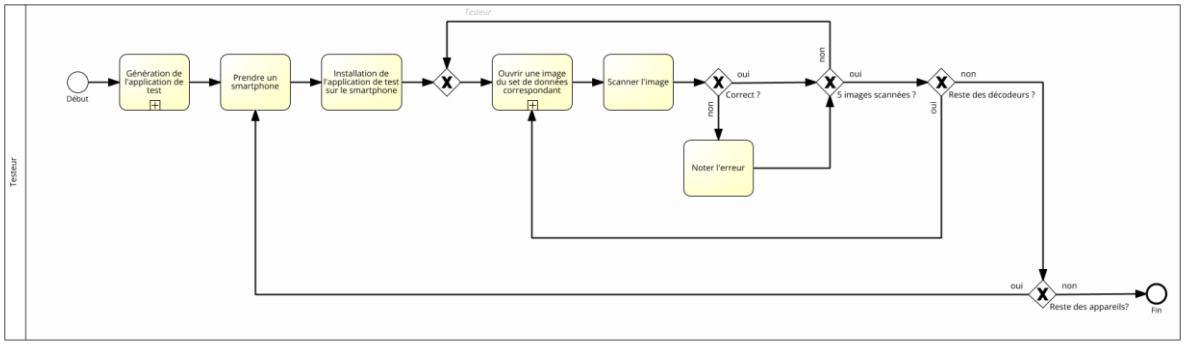
- L'étape « Setup » prépare les conteneurs Docker utilisés par la CI (aucune modification) ;
- L'étape « Build » compile le code C des décodeurs (aucune modification) :
- L'étape « Benchmark » exécute le décodage suivant le set de données (nouvelle étape) ;
- L'étape « Analyse » compare les résultats obtenus aux sessions précédentes (nouvelle étape) ;

4.3. Analyse d'automatisation des tests d'un SDK smartscan

4.3.1. Existant

Les tests effectués sur un SDK smartscan ne sont pas optimaux. Ils sont réalisés manuellement, les sets de données sont petits et les appareils restreints. Malgré tout, ce processus fonctionne bien et permet d'éliminer certains bugs. Cependant, la procédure est longue et n'est pas très intéressante pour les développeurs. C'est pour ces raisons que l'institut Icare souhaite les automatiser. Le Business Process Model and Notation (BPMN) des tests opérés par le développeur se trouve ci-dessous. Il est également disponible au format `.pdf` sur la clé USB (voir annexe 4).

FIGURE 14: BPMN DE TEST ACTUEL POUR LE SDK SMARTSCAN



Source : BPMN réalisé par l'auteur

1. Compiler l'application de test avec le décodeur généré ;
2. Installer l'application sur les smartphones/tablettes pouvant poser des problèmes ;
    a. Samsung S10 5G ;
    b. Nokia 7+ ;
    c. Samsung Galaxy Tab S5E ;
    d. Sony Xperia Z2 ;
3. Scanner au minimum cinq images par sous-type de décodeurs ;

(Pereira, 2020)

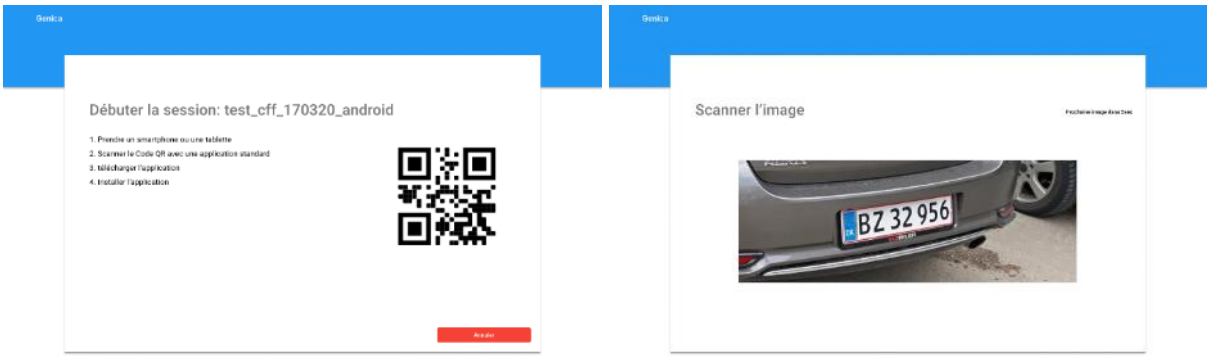
4.3.2. Proposition

Cette partie est difficilement automatisable. Néanmoins, il existe des idées pour faciliter ce test. Ici, aucun test unitaire n'est ajouté, car il ne testerait pas efficacement la communication entre la caméra et la librairie. Le but principal de ce test est de vérifier la communication entre la caméra et le SDK smartscan.

Ce test doit à tout prix utiliser la caméra de l'appareil. Sans elle, le test revient à réaliser la suite de non-régression effectuée par la CI. Ce test automatisé a besoin d'être suivi par un testeur, mais son travail sera simplifié et la qualité des tests supérieurs. Voici les améliorations apportées par le nouveau processus :

- Utilise la caméra du smartphone ou de la tablette ;
- Un set de données élargi ;
- De nombreuses tâches automatisées :
    - Choix de l'image ;
    - Passage à l'image suivante ;
    - Changement au prochain sous-type de décodeur ;
- Génération d'un rapport ;

FIGURE 15 : MOCKUP DU NOUVEAU PROCESSUS DE TEST



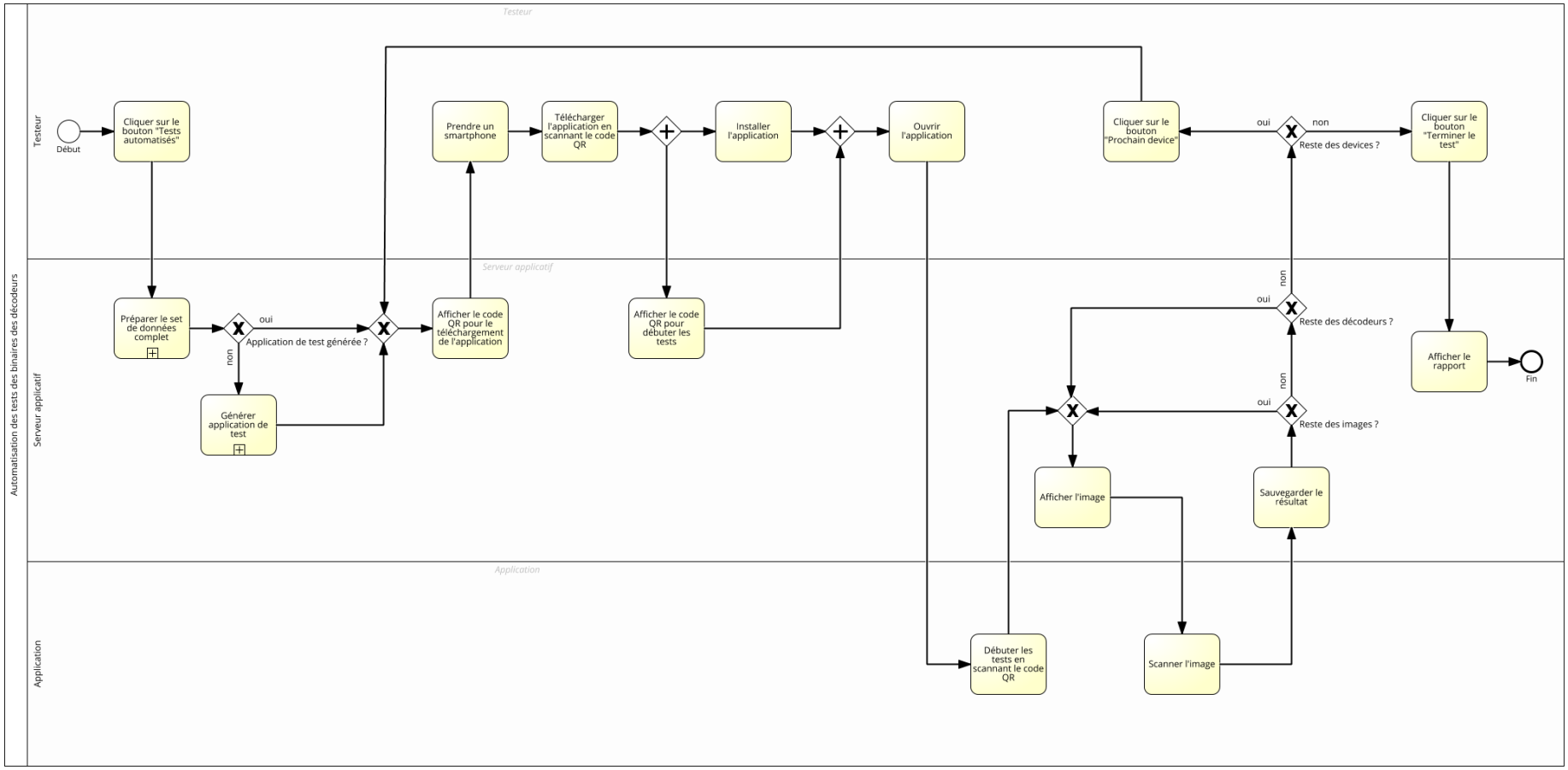
Source : Mockup réalisé par l'auteur

Le nouveau BPMN « automatisé » est le suivant :

1. Le testeur clique sur le bouton « Tests automatisés » de la page détail d'une livraison ;
2. La demande est transmise au serveur. Il commence à récupérer et à préparer le set de données complet ;
3. Vérification aidant à connaitre si l'application de test a déjà été générée. Si elle n'existe pas, le serveur la construit ;
4. Génération de la séance de test et affichage d'un code QR permettant de télécharger l'application de test ;
5. Le testeur scanne le code QR pour installer l'application de test sur l'appareil qu'il aura précédemment sélectionné ;
6. Après la lecture du code QR, le serveur génère un autre code QR. Celui-ci est utilisé pour démarrer la session de test ;
7. Le testeur ouvre l'application et numérise le code QR. La session de test débute ;
8. L'application scanne l'image puis envoie le résultat au serveur ;
9. Le serveur affiche l'image suivante jusqu'à la dernière ;
10. La plateforme demande si le testeur souhaite tester un autre téléphone (retour à l'étape n° 4) ou générer le rapport ;
11. Le serveur affiche le rapport ;

Le BPMN est visible à la page suivante et est disponible au format `.pdf` sur la clé USB (voir annexe 4).

FIGURE 16: BPMN POUR L'AUTOMATISATION DES TESTS DES SDK SMARTSCAN



Source : BPMN réalisé par l'auteur

4.4. Analyse de la communication entre l'API et le worker

Comment se passe la communication entre l'API et le worker lors la génération des SDK ? Cette connexion doit pouvoir se faire sur un serveur Linux ou un MacOs. En effet, le processus iOS a besoin d'être réalisé sur un MacOs alors que les SDK Android peuvent être créés sur n'importe quel système d'exploitation. Il existe différentes solutions afin d'établir un couplage entre l'API et un Worker, les voici :

- Exécuter la génération au même endroit que l'API ;
- Établir une connexion Secure Shell (SSH) permettant l'exécution d'un script ;
- Établir une connexion via des web sockets et un agent installé sur le serveur, conteneur ou machine ;

4.4.1. Sur le même serveur

Le code pour la génération est inclus dans celle du serveur API. Cette communication s'effectue comme n'importe quel appel à une classe ou une méthode. Ce processus pose quelques problèmes :

- Possible surcharge des ressources lors de la génération des SDK ;
- Blocage dans l'exécution courante du serveur, la réponse est rendue à la fin de la génération des SDK, car Python ne peut pas opérer des fonctions asynchrones nativement. Les nouvelles connexions sont toujours acceptées, puisque Django Rest Framework le supporte ;
- Irréalisable de générer les SDK iOS sauf si le serveur API est exécuté sur un MacOs ;

Cette solution inadéquate est abandonnée au vu de ces nombreux désavantages.

4.4.2. Communication SSH

Ce processus utilise plusieurs connexions SSH. Elle permet d'établir un lien sécurisé et crypté entre deux machines. Ce protocole est utilisé pour le transfert de données ou de commandes. Pour l'instauration de cette connexion, le serveur API doit connaitre le mot de passe ou avoir installé sa clé SSH public sur le worker et inversement. La façon la plus efficace et la moins complexe est de passer par ces clés SSH. Toutefois, cela peut être délicat à maintenir et plus particulièrement dans

des conteneurs Docker. À chaque nouveau worker, il faut veiller à bien rajouter les clés SSH public du serveur API. Cette solution est sûrement la solution la plus rapide à être mise en place.

1. Établissement d'une connexion SSH vers le worker pour transmettre le fichier de configuration via la commande `scp` ;
2. Établissement d'une connexion SSH vers le worker qui lance le script de génération puis ferme la connexion ;
3. Le script génère le SDK smartscan ;
4. Les scripts établissent une connexion vers l'API pour transmettre le SDK en HTTP ;
5. Traitement pour rendre la librairie téléchargeable. Changement du propriétaire, permission et emplacement si nécessaire ;

TABLEAU 13 : COMMUNICATION SSH – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Rapide à être mis en place ;	• Gestion des clés SSH ; • Ouverture de plusieurs connexions ; • Aucune file de messages ;

Source : Tableau de l'auteur provenant de sources multiples

4.4.3. Communication web sockets

Dans cette solution, un worker Python est créé pour établir une connexion avec le backend via des web sockets. Cette connexion permet à l'API de communiquer avec les workers. Une fois que l'agent Python est rattaché à l'API, il attend la demande de génération de SDK de la part de l'API en envoyant la configuration du SDK souhaitée. Une fois la génération terminée par l'agent, ce dernier envoie le SDK fraîchement généré à travers l'API.

1. Le conteneur API envoie la configuration à l'agent ;
2. Le worker génère la librairie ;
3. Le worker transmet le SDK au conteneur API ;

4. Traitement pour rendre le SDK téléchargeable. Changement du propriétaire, permission et emplacement si nécessaire ;

TABLEAU 14 : COMMUNICATION WEBSOCKET – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Communication ; • Facilite la transmission de la configuration ou du fichier ;	• Mise à jour du worker ; • Nécessite plus de travail ; • Aucune file de messages ;

Source : Tableau de l'auteur provenant de sources multiples

4.4.4. Communication message broker

Cette section fait suite à une deuxième analyse. Ici, la communication entre les deux parties se fait à l'aide d'un intermédiaire. En réalité, c'est une DB spécifique qui comporte un mécanisme publish-subscribe. C'est-à-dire que la DB émet un événement aux différents clients connectés. L'intérêt d'un tel système est la possibilité de créer des files afin de s'assurer que la tâche soit traitée et historisée. Dans ce cas, il faut également instaurer un worker pour le traitement des tâches de génération.

1. L'API envoie la configuration à la DB Redis (publish-subscribe) ;
2. La DB Redis avertit les workers connectés ;
3. Le premier worker à réagir génère le SDK ;
4. Le worker transmet le SDK au conteneur API ;
5. Traitement pour rendre le SDK téléchargeable. Changement du propriétaire, permission et emplacement si nécessaire ;

FIGURE 17 : COMMUNICATION MESSAGE BROKER – AVANTAGES ET INCONVÉNIENTS

Avantages	Inconvénients
• Communication ; • Connexion maitrisée ; • Facilite la transmission de la configuration ou du fichier ; • File de messages, historisation ;	• Mise à jour du worker ;

Source : Tableau de l'auteur provenant de sources multiples

4.4.5. Choix de la communication

Le premier choix s'est porté sur une communication via des web sockets. Toutefois lors de la mise en pratique, ce n'était pas la bonne solution. Cela vu que cette technologie a pour but de notifier tout client d'un changement d'état du serveur. De plus, cette solution ne permet pas d'assurer un transport correct de la tâche, car ces dernières ne sont pas stockées.

Finalement, après une deuxième réflexion, **c'est la communication message broker qui a été retenu**. Elle a l'avantage de :

- Maitriser la communication afin de garder les tâches si aucun worker n'est connecté grâce à une file ;
- Redistribuer les tâches au bon worker. Le worker iOS ne va pas tout d'un coup recevoir des demandes Android ;
- Connaitre le statut d'une tâche et son erreur si elle échoue ;

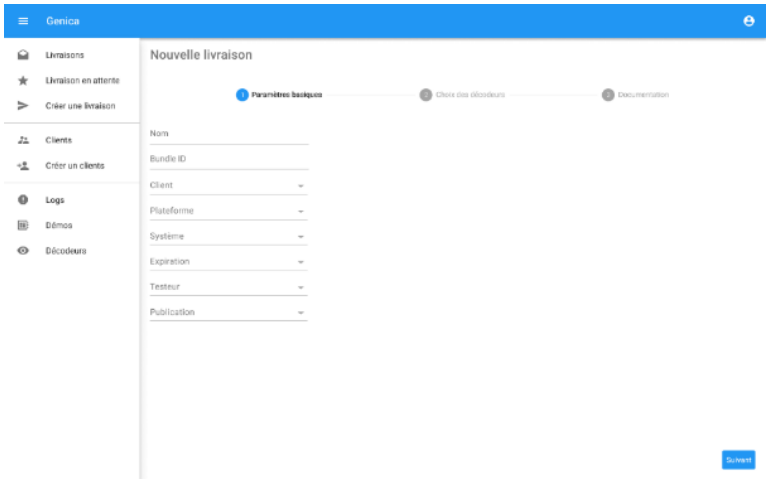
5. Les mockups

L'interface graphique de l'application se base sur les règles du « material design » de Google. Ce guide permet de créer rapidement des programmes simples et intuitifs. Ces mockups présentent une interface professionnelle et facile à utiliser. La dernière version est visible à l'annexe 2. Les mockups ont été réalisés en quatre itérations.

5.1. Version 1.0

La première a consisté à créer les premiers mockups de l'application selon les besoins définis au chapitre 3. Le but de version 1 est d'amener des discussions sur une interface plus ou moins concrète pour cerner au mieux le projet.

FIGURE 18 : MOCKUP VERSION 1.0



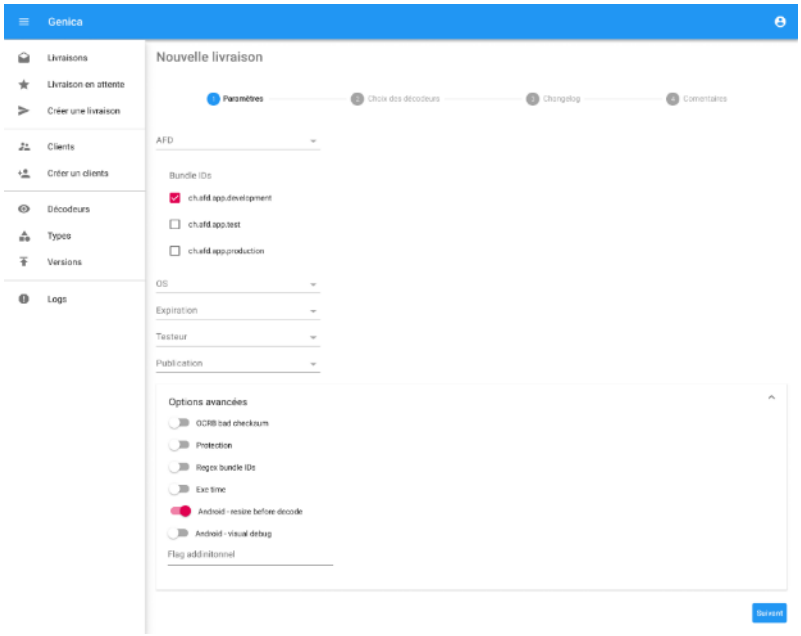
Source : Mockup réalisé par l'auteur

5.2. Version 1.1

La seconde version prend en compte les remarques ainsi que la réévaluation des besoins. Le formulaire de création de livraison voit sa dernière étape disparaitre au profit de deux autres. Une pour les changelogs et la prochaine pour des commentaires. D'autres nouvelles interfaces sont également ajoutées comme l'ensemble de pages détails de livraison différencié par leur statut et d'autres pages moins importantes :

- Détail des décodeurs ;
- La liste des types de décodeurs ;
- La liste des versions ;

FIGURE 19 : MOCKUP VERSION 1.1

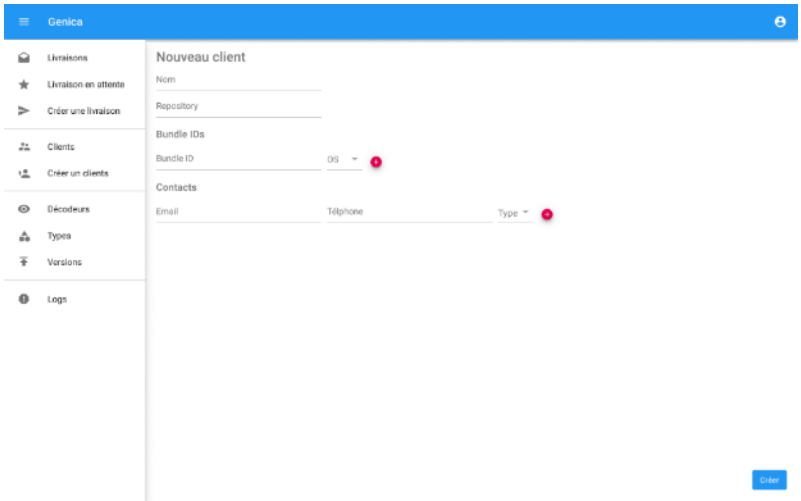


Source : Mockup réalisé par l'auteur

5.3. Version 1.2

Cette troisième itération règle certains détails de la version précédente. Une section avancée voit le jour dans le formulaire de création de livraison. Elle permet d'y déplacer le choix de la version. Par défaut, la dernière version est toujours sélectionnée. Lors de la création d'un client, les bundle IDs doivent être liés à un OS. Concernant les contacts, il est bien de préciser un numéro de téléphone ainsi qu'un type afin de savoir si cette personne est un développeur ou une personne du business. Cette version 3 représente l'interface de base finale.

FIGURE 20 : MOCKUP VERSION 1.2

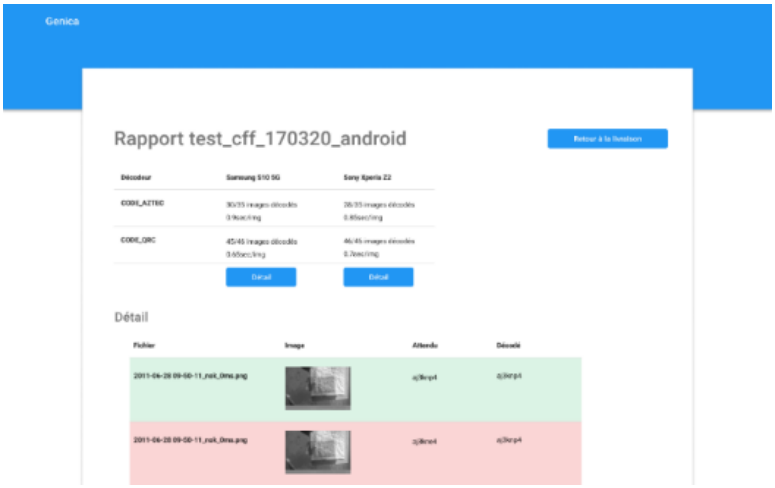


Source : Mockup réalisé par l'auteur

5.4. Version 1.3

La version finale fait suite à l'analyse des moyens d'automatisation des tests du chapitre 4.3. Elle ajoute six mockups pour les sessions des tests.

FIGURE 21 : MOCKUP VERSION 1.3

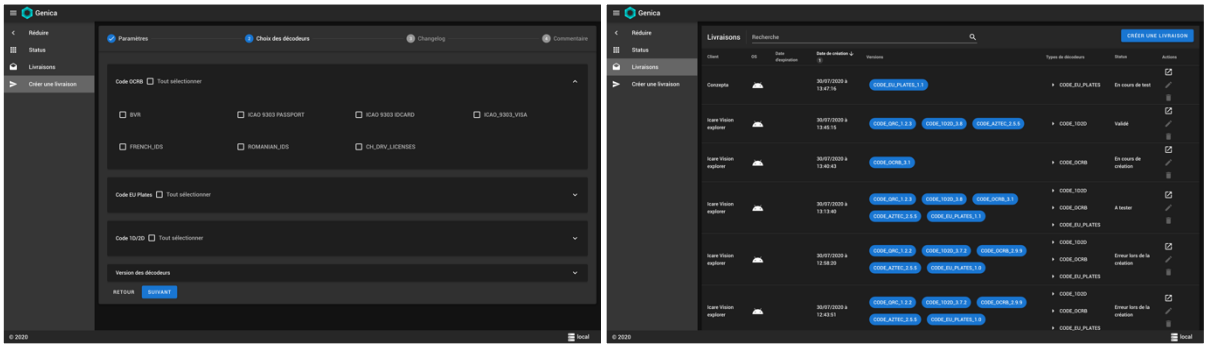


Source : Mockup réalisé par l'auteur

Finalement, l'application ressemble aux mockups. Les pages et les catégories sont respectées. Bien entendu, certaines divergences existent entre le logiciel réel et le mockup. Par exemple :

- Le thème blanc prédéfini est devenu foncé ;
- Certaines modifications sur le formulaire de création de livraison ;
- Quelques détails sur la liste des livraisons

FIGURE 22 : INTERFACE RÉELLE

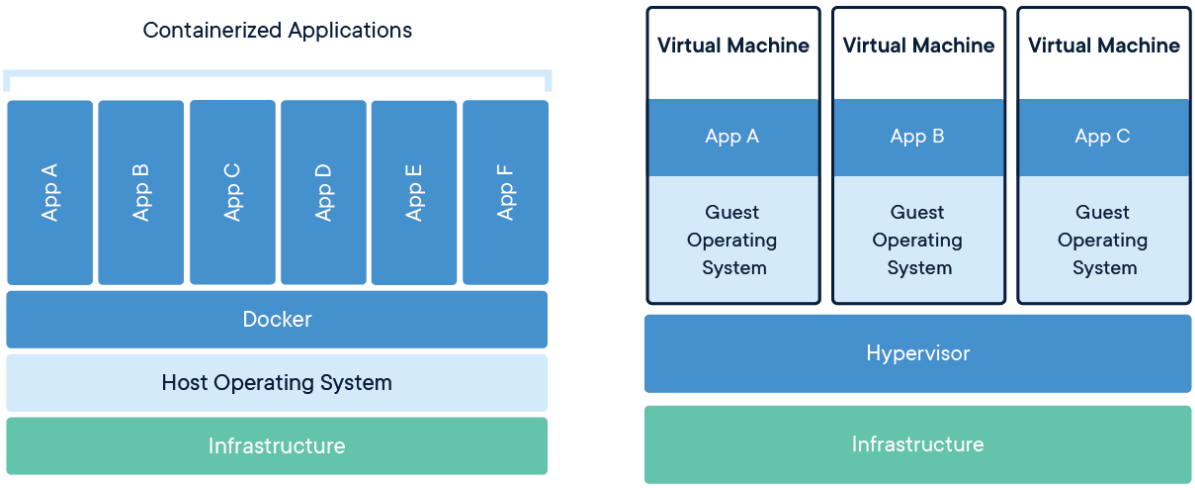


Source : capture d'écran réalisée par l'auteur

6. L'architecture IT

La plateforme se base sur la technologie de conteneurisation. Aucun serveur n'est utilisé, mais des conteneurs Docker. L'application et ses dépendances sont regroupées dans cette unité. L'avantage de cette technologie est que chaque programme est séparé, indépendant et facilement déployable. Les conteneurs sont légers, autonomes et fonctionnent toujours de la même façon contrairement à des VMs plus volumineuses, demandent plus de ressources et l'OS doit être maintenu à jour (Docker, s.d.). Le fait d'avoir différentes facettes du projet (une API, une interface graphique web et un worker) a poussé le choix d'utiliser des conteneurs.

FIGURE 23 : DIFFÉRENCE ENTRE DES CONTENEURS ET DES MACHINES VIRTUELLES



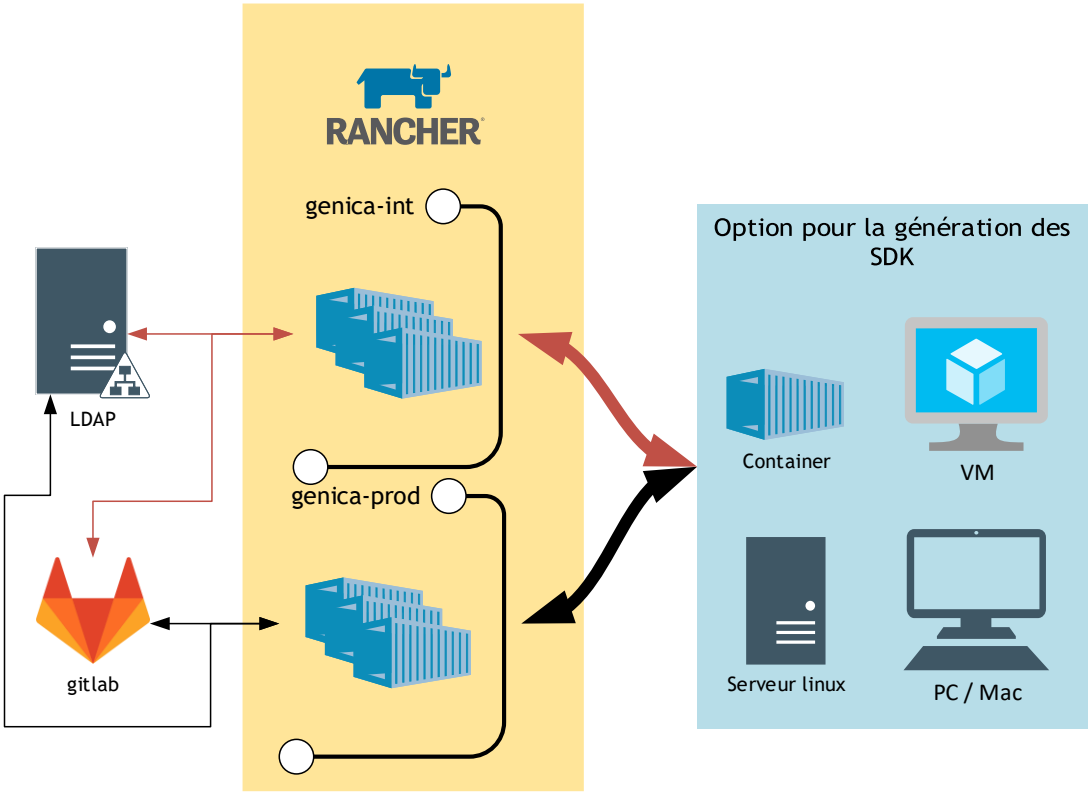
Source : https://www.docker.com/resources/what-container

Comme l'on peut le voir sur les schémas ci-dessus, Docker s'exécute directement sur le système hôte, puis l'application est conteneurisée. Tandis que l'autre système a plus de niveaux. Un hyperviseur virtualise les composants d'une machine pour installer un OS et finalement exécuter le logiciel.

Docker permet de faire de l'Infrastructure as Code (IaC). Le système se base sur des fichiers de définition et non pas sur des configurations physiques ou sur des outils de configuration. Avec Docker, un conteneur se définit à l'aide d'un fichier nommé `Dockerfile`. Concernant les applications composées de plusieurs services, la définition s'écrit dans un fichier `docker-compose.yml`.

La plateforme Genica est déployée dans deux écosystèmes distincts. Le premier est un environnement d'intégration avec les dernières modifications tandis que le deuxième correspond à la production, qui s'appuie sur la dernière release. Concernant le worker pour la génération des SDK, aucune porte n'est fermée. Si le worker a la capacité de s'adapter à divers supports, il sera autant plus simple de réaliser la construction des SDK iOS.

FIGURE 24 : SCHÉMA BASIQUE DE L'ARCHITECTURE IT



Source : Schéma réalisé par l'auteur

6.1. Environnement local

L'environnement de développement utilise des conteneurs docker spécifiques, c'est-à-dire qu'ils ne sont pas forcément prêts à être employés en production. Ils ont en revanche l'avantage de transmettre chaque modification, car le projet est monté dans le conteneur. De plus, le serveur API, l'interface graphique et le worker se mettent à jour automatiquement lors d'un changement.

C'est une nouvelle approche de développement et de gestion de déploiement qui est proposée à l'institut Icare. Pour le moment, les conteneurs ne sont pas démocratisés tant bien du côté du développement que du côté infrastructure. Cette méthodologie offre les avantages suivants :

- Unifie les environnements de développement entre les différents OS et développeurs ;
- Le worker est facilement créé alors que la configuration nécessaire est fastidieuse, les nombreux dépôts des décodeurs, les librairies ou les paquets à installer ;
- Déploiement plus rapide ;
- IaC ;
- Environnement reproductible en quelques clics ;

Bien entendu, des désavantages existent :

- Certains Dockerfiles peuvent être compliqués et longs à écrire, notamment le container pour le worker ;
- Le lancement des tests unitaires est plus complexe, car il faut passer par le docker. Par exemple : `docker exec -it <container_id> coverage run -m pytest`;
- Attention à la sécurité ;

La génération des conteneurs peut prendre du temps, en particulier celui du worker. Ce dernier a besoin de télécharger et d'installer de nombreux paquets. Une amélioration est sûrement possible sur ce conteneur.

Pour lancer un groupe de conteneurs définis dans un fichier `docker-compose.yml`, la commande suivante est exécutée `docker-compose up --build -d`. Bien entendu, Docker doit être installé et configuré. Si Docker Desktop ne se trouve pas sur votre ordinateur, la documentation officielle explique bien la procédure d'installation.

6.2. Environnement d'intégration

Cet environnement intègre les dernières modifications du dépôt Git et plus précisément de la branche « develop ». La plateforme d'intégration est uniquement accessible depuis le Local Area Network (LAN) de l'institut Icare via l'URL https://genica-int.icare.ch.

Les conteneurs utilisés sont exécutés sur un serveur situé sur le cloud de l'institut Icare. Ces derniers fonctionnent en mode « debug », c'est-à-dire que les fichiers journaux sont détaillés afin d'analyser correctement les erreurs. L'option des API navigables de Django REST Framework est activée.

Un conteneur worker a besoin d'un serveur de source et le plus simple est d'employer le serveur Gitlab déjà mis en place. Il suffit de créer un sous-groupe dans le groupe Smartscan release et d'y ajouter un dépôt.

Concernant l'authentification, cet environnement utilise le serveur LDAP déjà mis en place à l'institut. De toute façon, l'accès se fait seulement en lecture, donc sans modification de l'arbre.

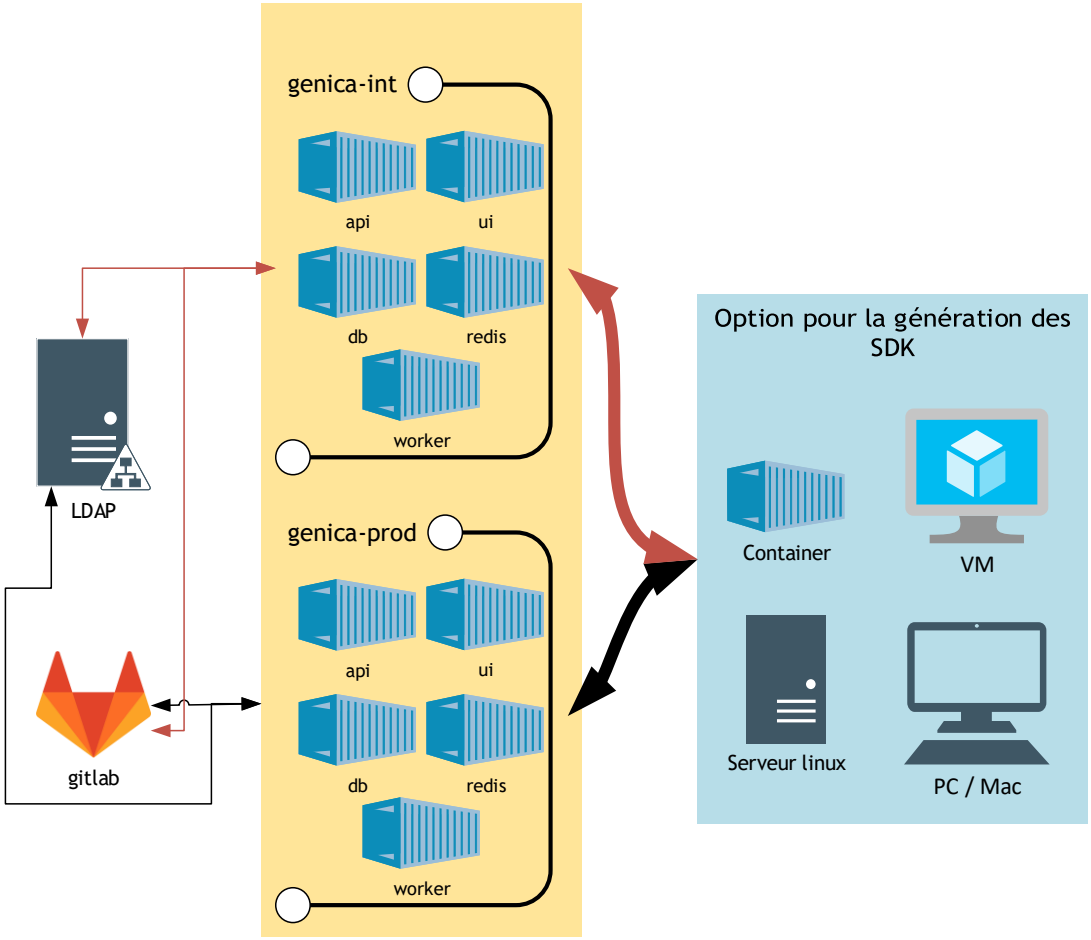
6.3. Environnement production

Cet environnement de production rend accessible la dernière release du projet via l'URL https://genica.icare.ch. Comme pour l'environnement d'intégration, la plateforme est uniquement disponible depuis le réseau LAN et les conteneurs sont exécutés sur un serveur CentOS situé sur le cloud de l'institut Icare.

Le serveur source et LDAP sont bien évidemment ceux de l'institut Icare. Mais les dépôts utilisés sont ceux des clients contrairement à l'intégration.

6.4. Architecture finale

FIGURE 25 : SCHÉMA DE L'ARCHITECTURE FINALE



Source : Schéma réalisé par l'auteur

Au départ, le but était d'utiliser Rancher pour faciliter la gestion des conteneurs. Inopportunément, cette plateforme open source de gestion de cluster de conteneurisation n'a pas été mis en place assez rapidement à l'institut Icare. C'est pour cela que les conteneurs se trouvent sur une machine CentOS. Les conteneurs sont gérés avec la ligne de commande au lieu de l'interface de Rancher.

L'environnement de la plateforme Genica se base sur une architecture trois tiers. En effet, la logique business est séparé de l'interface graphique et de la DB. Pour cela, voici les conteneurs obligatoires :

- **Un conteneur API** où se situe le backend. Le code est réalisé avec le framework Django REST et servi par Gunicorn qui est un Web Server Gateway Interface (WSGI) pour Python. Les SDK générés et les fichiers journaux sont persistants ;

- **Un conteneur UI** expose le frontend développé avec Vue.js ;
- **Un conteneur DB** (PostgreSQL) pour sauvegarder les données dans un volume monté sur le conteneur, ceci dans le but de persister les informations ;
- **Un conteneur Redis** utilisé par l'API pour transmettre les tâches aux différents workers via une file de messages ;
- **Un ou plusieurs conteneurs worker** similaires à celui de développement avec quelques adaptations. Ceci afin de faciliter son déploiement ;

7. Le dépôt Git de Genica

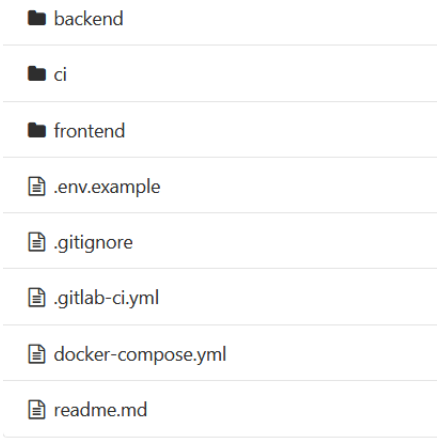
Un groupe genica et un dépôt du même nom sont créés sur le serveur Gitlab de l'institut Icare. Le groupe est accessible à :

- Benoit Golay (Product Owner) ;
- Christian Pereira ;
- Christophe Abbet ;
- Ken Jäggi ;

7.1. La structure des répertoires

Ce seul dépôt réunit les trois sous projets (API, worker et frontend). Mais attention, l'API et le worker partagent le même code base. Les sous-modules Git ne sont pas utilisés, car ils nécessitent trois projets Git. Un pour le frontend, un pour l'API/worker et le dernier pour regrouper le tout et revenir à la même structure qu'avec un dépôt. Les sous-modules seront éventuellement mis en place pour la continuation de ce projet, car d'autres personnes travailleront peut-être sur ce projet. Et là, les sous-modules peuvent être avantageux en découpant les projets.

FIGURE 26 : STRUCTURE DU PROJET



Source : capture d'écran réalisée par l'auteur

Le projet est séparé en trois répertoires principaux. Le dossier `backend`, contient tout le code pour le serveur API et le worker réalisé en Python avec Django REST Framework. C'est également ici que se trouvent les fichiers `Dockerfile` de l'API et du worker pour l'environnement local. Le second dossier `ci` contient l'ensemble des définitions des conteneurs Docker (pour le frontend, l'API et le worker) ainsi que le

`docker-compose.yml`. Ces fichiers définissent les environnements. Le troisième dossier `frontend` contient l'interface graphique réalisée avec Vue.js ainsi que le `Dockerfile` pour l'environnement local. Ces dossiers sont vus plus en détail dans les chapitres suivants.

Ce dossier contient également d'autres fichiers importants comme le fichier `gitlab-ci.yml` qui définit les actions de la CI/CD. Le `.env. exemple` pour créer le fichier `.env` spécifie les variables et les configurations de l'environnement. Le fichier `readme.md` contient des informations sur le projet et la documentation technique (lancement, configuration et explication).

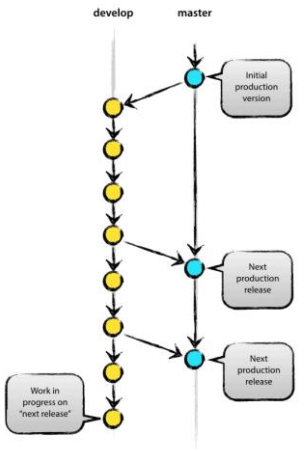
7.2. Les branches

Afin d'organiser le travail sur le dépôt, une stratégie Git élaborée prend en compte :

- La gestion de deux environnements (intégration et production) ;
- Un développeur, mais l'application sera maintenue par plusieurs développeurs dans un futur proche ;

La stratégie choisie pour ce projet est « Environment branches ». Cette stratégie offre la possibilité de gérer plusieurs environnements. La branche « `develop` » est utilisée pour développer et correspond à l'environnement d'intégration. La branche « `master` » est protégée et n'accepte que les merges request ou des pulls request. C'est la branche correspondant à l'environnement de production.

FIGURE 27: BRANCHING GIT



Source : *https://medium.com/@sairamkrish/git-branching-strategy-for-true-continuous-delivery-eade4435b57e*

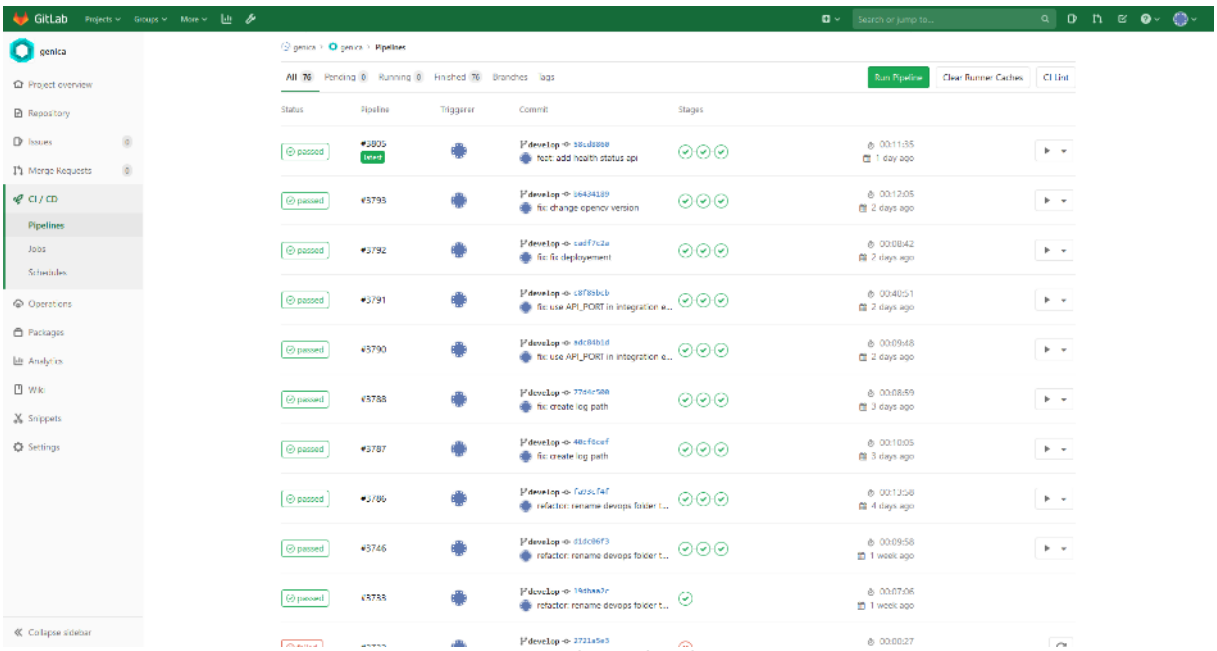
7.3. L'intégration continue

> *« L'intégration en continu est une pièce centrale de tout système CI-CD. Elle a pour but de traiter chaque changement effectué sur le dépôt source et d'enclencher les mécaniques de vérification, de compilation et de tests. » (Leite, 2019)*

Dans ce projet, une intégration continue CI/CD de Gitlab a été mise en place. Les actions exécutées sont définies dans le fichier `gitlab-ci.yml` écrit en yaml visible à l'annexe 3. Ce fichier décrit l'image utilisée, les étapes et les tâches de chaque pipeline, une par branche. Si une tâche échoue, le pipeline est arrêté et un e-mail est envoyé au développeur.

Le pipeline d'intégration réagit automatiquement à chaque modification sur la branche `develop` du dépôt source. Si aucune étape n'échoue, le changement est automatiquement déployé. Le second pipeline, relative à l'environnement de production, est quant à lui exécuté manuellement. Il comporte les mêmes tâches, mais concerne la branche `master`.

FIGURE 28 : PIPELINES EXÉCUTÉS



Source : capture d'écran réalisée par l'auteur

7.3.1. L'étape test

L'étape `test` comporte une tâche décrite par deux paramètres. La première partie `before_script` permet de créer un fichier `.env` défini par une variable dans Gitlab, de configurer l'accès au serveur source, d'installer les dépendances systèmes et Python dans le conteneur utilisé par la CI/CD. La deuxième étape `script` lance les tests unitaires et calcule le taux de couverture du code python. Cette étape se lance sur toutes les branches.

FIGURE 29 : GITLAB-CI - ÉTAPE TEST

```
test:api:
  stage: test
  before_script:
    - apt-get update -y && apt-get install git -y
    - cp $ENV_TEST ./.env
    - pip install -r requirements.txt
    - mkdir -p /root/.ssh
    - chmod 700 /root/.ssh
    - touch /root/.ssh/known_hosts
    - chmod 600 /root/.ssh/known_hosts
    - ssh-keyscan gitlab.icare.ch > /root/.ssh/known_hosts
  script:
    - coverage run -m pytest
    - coverage report -m
```

Source : code provenant de l'auteur

7.3.2. L'étape build

L'étape `build` est composée de six tâches, une pour chaque conteneur. Chaque tâche lance la construction des conteneurs docker pour les déposer dans le registery docker de Gitlab.

FIGURE 30 : GITLAB-CI - EXTRAIT DE L'ÉTAPE BUILD

```
.build: &build
  stage: build
  image: docker:19.03.8
  services:
    - docker:19.03.8-dind
  before_script:
    - docker login -u root -p ${REGISTERY_PWD} gitlab.icare.ch:4567

build:api:dev:
  <<: *build
  script:
    - docker build --pull --no-cache -t ${CI_REGISTRY_IMAGE}/api:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-dev -t ${CI_REGISTRY_IMAGE}/api:dev -f
./devops/docker/backend/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/api:dev
  only:
    - develop
```

Source : code provenant de l'auteur

Le paramètre `.build` définit une partie générique de la construction. Plus précisément, cette partie définit l'image, le service et la connexion au registery docker de gitlab.icare.ch. La tâche de construction du conteneur `api:dev` utilise la partie générique (`.build`). Lors de cette tâche, le conteneur se construit avec les tags suivants :

- SHA du commit (identifiant alphanumérique unique d'un commit) ;
- ID de la tâche ;
- Tag unique « dev » pour la branche `develop` ou « latest » pour la branche `master` ;

Ces tags permettent de différencier les conteneurs entre eux et de savoir à quelle branche ils appartiennent. Le dernier paramètre `only` permet de limiter la branche où la tâche est exécutée.

7.3.3. L'étape deploy

L'étape `deploy` est composée de deux tâches similaires, une pour l'environnement d'intégration et la deuxième pour la production. La tâche concernant l'intégration :

- Transfert de `/ci/environment/docker-compose.yml` ;
- Création du fichier `.env` selon la variable `$ENV_INT` créée dans Gitlab ;
- Transfert de la clé privée `id_rsa` ;
- Connexion au registery Gitlab
- Démarrage des conteneurs dockers

FIGURE 31 : GITLAB-CI - EXTRAIT DE L'ÉTAPE DEPLOY

```
deploy:int:
  <<: *deploy
  variables:
    user: icare
    host: 10.0.5.11
    port: 22
    destination_path: /home/icare/genica-int
    docker_compose_path: ./ci/environment/docker-compose.yml
  script:
    - scp -o StrictHostKeyChecking=no $docker_compose_path
$user@$host:$destination_path/docker-compose.yml
    - scp -o StrictHostKeyChecking=no $ENV_INT $user@$host:$destina-
tion_path/.env
    - ssh -t -o StrictHostKeyChecking=no $user@$host 'mkdir -p
/home/icare/genica-int/.ssh'
    - scp -o StrictHostKeyChecking=no ~/.ssh/id_rsa $user@$host:$destina-
tion_path/.ssh/id_rsa
    - ssh -t -o StrictHostKeyChecking=no $user@$host "docker login -u root
-p ${PASSWORD} ${CI_REGISTRY}"
    - ssh -t -o StrictHostKeyChecking=no $user@$host 'cd /home/icare/gen-
ica-int && docker-compose pull && docker-compose up -d'
  only:
    - develop
```

Source : code provenant de l'auteur

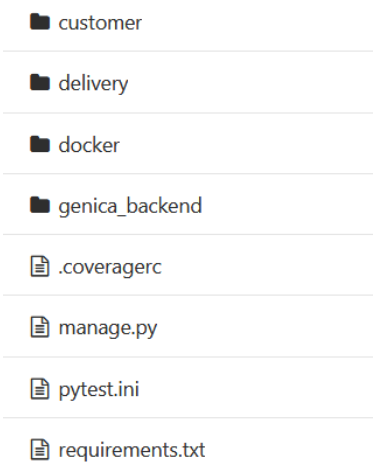
8. L'API et le worker

Le code réalisé avec Django REST Framework et Celery se situe dans le dossier `backend`. Le serveur API et le worker partagent le même code de base, c'est-à-dire que les tâches du worker sont intégrées au code du serveur API. Django a besoin de connaitre et d'accéder à la méthode qui doit être exécutée de façon asynchrone. Cependant, lors du lancement du serveur API, le worker n'est pas exécuté. Le worker doit être lancé séparément avec une ligne de commande spécifique `` `celery worker -A genica_backend -Q android,publish -l info` ``.

Django n'est pas capable d'effectuer des fonctions asynchrones, c'est pour cela que Celery est utilisé. Ce module python externe permet de réaliser des tâches asynchrones en employant une file de messages que le worker peut également lire. Dans ce projet, c'est Redis, une DB clé-valeur performante et open source qui est choisie. L'intégralité des messages est stockée en mémoire vive.

8.1. Structure

FIGURE 32 : STRUCTURE DU RÉPERTOIRE « BACKEND »



Source : capture d'écran réalisée par l'auteur

Ci-dessus, la structure du répertoire `backend` est composée de deux applications (`customer` et `delivery`) permettant de regrouper des fonctionnalités similaires. « Le minimum est d'avoir une application. Pour un site très basique, on peut s'en contenter et y loger tout le code. Mais le plus souvent, la conception d'une architecture modulaire est préférable, en cloisonnant les fonctionnalités par grandes

familles, chacune donnant lieu à une application. » (Samson, 2020). Les autres répertoires sont décrits dans les chapitres suivants et concernant les fichiers :

- Le fichier `manage.py` est le point d'entrée des commandes destinées à ce projet. Par exemple, pour créer une application, exécuter une migration, etc. ;
- Le fichier `requirements.txt` contient les dépendances Python du projet ;
- Les fichiers `pytest.ini` et `.coveragerc` sont utilisés pour les tests unitaires ;

Le code source est disponible dans le répertoire `/source_code/genica` de la clé USB (voir annexe 4).

8.1.1. Répertoire genica_backend

C'est le dossier central du projet. Il est créé lors de la génération de la plateforme avec la ligne de commande : `python manage.py startproject genica-backend`. Cette commande fonde le projet et une partie de la structure vue précédemment. Ce dossier contient :

- `init.py` définit le répertoire comme étant un module Python ;
- `celery.py` définit l'instance Celery ;
- `settings.py` décrit les paramètres de configurations du worker et du serveur API ;
- `urls.py` déclare les routes du projet ;
- `wsgy.py` est le point d'entrée pour les serveurs WSGI comme Gunicorn ;

FIGURE 33 : EXTRAIT DE CODE DE CELERY.PY

```
def get_customers():
    url = urljoin(settings.API_ENDPOINT, f"/api/v1/customers/")
    response = requests.get(url)
    if response.status_code != 200:
        logger.error(f"Expected status ... with url {url}")
        raise requests.ConnectionError(f"Expected status ... url {url}")
    return json.loads(response.text)

@celeryd_init.connect
def configure_workers(sender=None, conf=None, **kwargs):
    for customer in get_customers():
        repo = GitTool(customer['repository'], '/usr/src/')
        repo.fetch()
```

Source : code source provenant de l'auteur

`Celery.py` est utilisé lors du démarrage d'un worker. Il configure et charge les tâches des différents modules et initialise le worker. La méthode `configure_workers(…)` appelle `get_customers()` pour interroger le serveur API et récupérer les clients. Ces derniers sont employés afin de récupérer les changements ou cloner le dépôt de publication de chaque client. Le décorateur `@celeryd_init.connect` permet de spécifier cette méthode comme étant la tâche d'initialisation du worker.

`settings.py` définit les configurations du serveur API ou du worker selon le fichier `.env` ou les variables d'environnement de l'OS. Grâce à la variable `MODE`, l'application est configurable en serveur API ou en worker. De base ce fichier offre la possibilité de :

- Créer le dossier et le fichier journal ;
- Définir les applications utilisées ;
- Configurer l'accès à la DB PostgreSQL ;
- Configurer l'accès à Redis ;

FIGURE 34 : EXTRAIT DE CODE DE URLS.PY

```
urlpatterns = [
    path('api/v1/', include([
        path('', include('delivery.urls')),
        path('', include('customer.urls'))
    ])),
]

urlpatterns += static(
                          settings.RELEASE_URL,
                          document_root=settings.RELEASE_ROOT
                      )
if settings.DEBUG:
    urlpatterns += static(
                              settings.STATIC_URL,
                              document_root=settings.STATIC_ROOT
                          )
```

Source : code source provenant de l'auteur

`urls.py` permet de définir les routes de la plateforme. Les routes des différentes applications sont insérées sous `api/v1/`. Alors que les fichiers statiques comme les releases sont accessibles directement. Si le mode debug est activé, le lien vers les fichiers statiques de l'interface des API est ajouté aux URLs.

8.2. Les conteneurs

Les `Dockerfile` utilisés lors du développement du serveur API et du worker se situent dans le répertoire `/backend/docker/` du projet. Pour rappel, le code source est disponible dans la clé USB (voir annexe 4). Ces conteneurs sont uniquement employés pour la programmation et aucunement sur un environnement, même celui d'intégration. Cela peut représenter un risque de sécurité, le code est exécuté par l'utilisateur `root`. De plus, les applications sont automatiquement rafraîchies à chaque modification d'un fichier Python. C'est pour cela que d'autres `Dockerfile` existent dans le répertoire `/ci/docker/` du projet.

8.2.1. Dockerfile de production pour le serveur API

Ce conteneur se base sur l'image officielle `python:3.8-slim` qui utilise une distribution Debian allégée. La construction de ce docker se fait en deux étapes. La première étape `build` permet de générer, de télécharger et de packager toutes les dépendances Python. Cette étape déleste le conteneur final. La deuxième phase reprend le package contenant les dépendances, copie le code et crée un utilisateur spécifique pour exécuter Django. L'API n'est pas lancée avec l'utilitaire `manage.py` de Django, mais avec un serveur WSGY Gunicorn. Le `Dockerfile` est disponible dans le code source dans le répertoire `/ci/docker/api/`.

8.2.2. Dockerfile de production pour le worker

Ce conteneur emploie également l'image `python:3.8-slim` et non pas une image Java. Car l'installation de Java Development Kit (JDK) s'est avérée plus simple que la bonne version de Python. Ce conteneur est utilisé pour les files de : générations des SDK Android et de publication.

Ce `Dockerfile` est le plus complexe. Il exécute des tâches Python, notamment celles de génération. Ce conteneur a recours aux outils de construction Android (JDK, gradel, SDK et Native Development Kit (NDK)), les outils de génération C/C++ (Cmake et ninja) et le code source des décodeurs. C'est pour cela que ce conteneur est volumineux. Le `Dockerfile` est disponible dans le code source dans le répertoire `/ci/docker/worker/`.

8.3. Application client

Cette application s'occupe de la gestion des clients. Actuellement, le modèle du client est simple, il comporte un nom et le chemin du dépôt de publication. Le serializer permet de transformer le modèle en JavaScript Object Notation (JSON) pour l'envoi et la réception de données. Cette API est exploitée par le worker pour récupérer la liste des clients. Pour le moment, aucune interface utilisateur ne l'utilise. Le dossier fixtures contient un fichier `.json` qui remplit la DB. Le dossier migrations renferme les fichiers de construction de la DB.

8.4. Application livraison

Cette application s'occupe de la gestion des livraisons. Actuellement, il a trois modèles :

- Le modèle `delivery` établit les données d'une livraison ainsi que les liaisons avec les modèles `configuration` et `customer` de l'application `customer` ;
- Le modèle `configuration` détient tous les paramètres utiles à la génération des SDK smartscan ;
- Le modèle `release` établit une relation entre les fichiers `.aar` et `.zip` téléchargés sur le serveur API ;

Dans les serializer `DeliverySerializer` et `ReleaseSerializer`, la fonction de création est redéfinie, car la méthode héritée de `ModelSerializer` ne convient pas. En effet, lors de la création d'une livraison, le code va :

1. Recevoir les données et extraire la configuration ;
2. Sauvegarder la configuration liée à la livraison dans la DB ;
3. Enregistrer la livraison en transmettant la configuration créée à l'étape 2 ;

Concernant l'ajout d'une nouvelle release, l'identifiant de la livraison est extirpé pour être spécifié lors de la sauvegarde de la release.

FIGURE 35 : EXTRAIT DE CODE DE DELIVERY/SERIALIZERS.PY

```
class DeliverySerializer(serializers.ModelSerializer):
    ...

    def create(self, validated_data):
        config_data = validated_data.pop('config')
        configuration = Configuration.objects.create(**config_data)
        delivery = Delivery.objects.create(
                        config=configuration,
                        **validated_data
                    )

        return delivery


class ReleaseSerializer(serializers.ModelSerializer):
    ...

    def create(self, validated_data):
        release = Release.objects.create(
                        delivery=validated_data.pop('delivery'),
                        **validated_data
                    )

        return release
```

Source : code source provenant de l'auteur

La logique métier de l'API est implémentée dans le fichier nommé `views.py`. L'héritage de `ModelViewsSet` sur les views facilite la création d'API. Car elle hérite des fonctions Create, Read, Update, Delete (CRUD). Ces fonctions n'ont pas besoin d'être spécifiées. Cependant, certaines vues sont plus complexes.

FIGURE 36 : EXTRAIT DE CODE DE DELIVERY/VIEWS.PY

```
class DeliveryViewSet(NestedViewSetMixin, viewsets.ModelViewSet):
    ...

def create(self, request):
    serializer = DeliverySerializer(data=request.data)

    if serializer.is_valid():
        serializer.save()
        logger.debug('Delivery created')
        task = generate_library.delay(serializer.data)
        logger.debug('generate library queued')
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    logger.error('Delivery not created')
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Source : code source provenant de l'auteur

C'est pour cela que la `DeliveryViewSet` implémente une méthode `create`. Elle permet d'enregistrer la nouvelle livraison puis de lancer la tâche de génération Android. Quand la génération iOS existera, une condition sera ajoutée afin

d'exécuter la bonne tâche de génération. Cette tâche n'est pas effectuée, elle est transmise à un worker via la DB Redis.

`DeliveryReleaseFileViewSet` passe outre la méthode de création afin d'adjoindre la livraison liée à cette release aux données. Deux actions sont également ajoutées. Quant à la méthode `publish`, elle lance la tâche asynchrone de publication de la release. La fonction `last` a pour objectif d'obtenir la dernière release pour une livraison donnée.

Les routes de cette application sont définies dans le fichier urls.py.

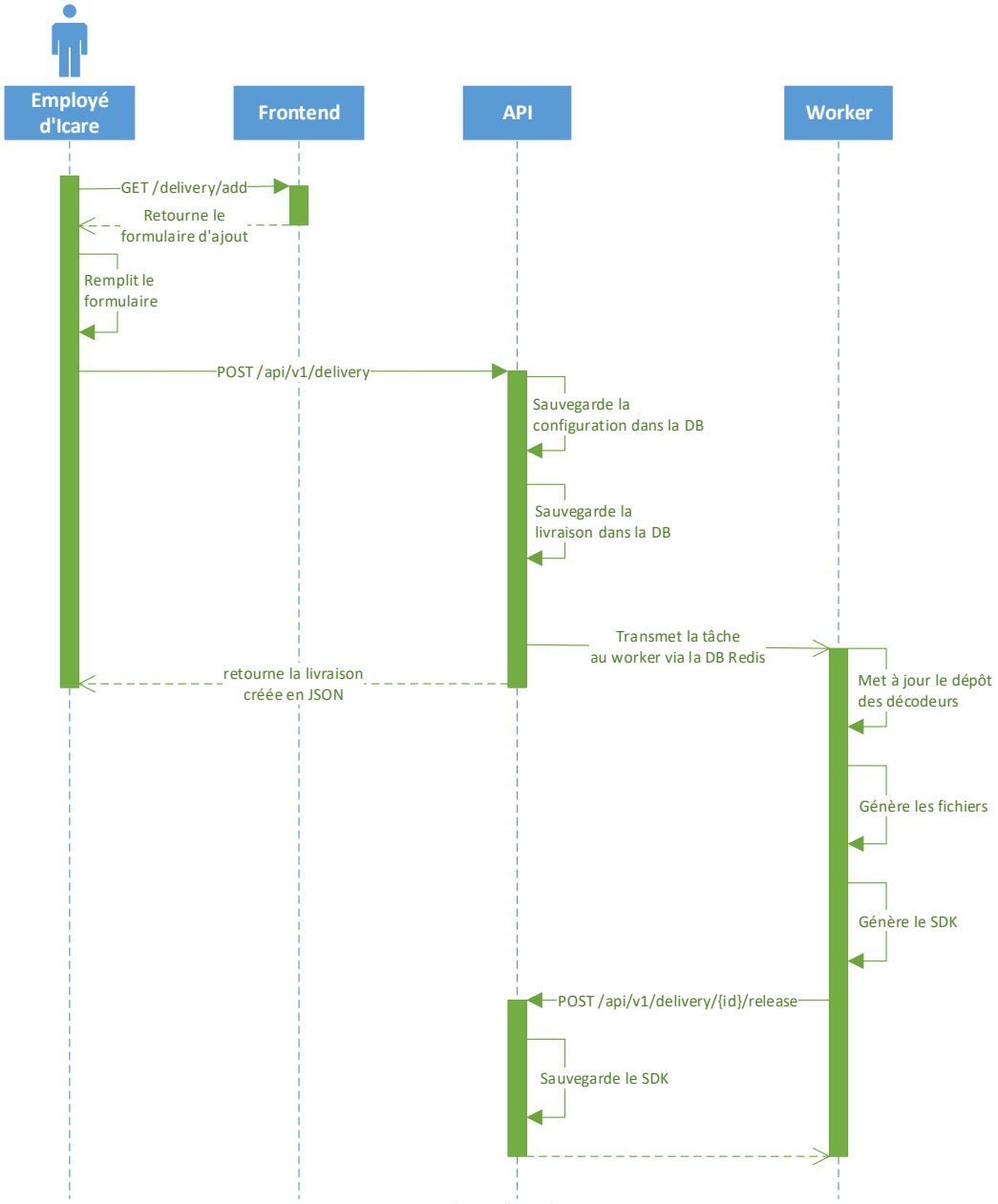
- Pour les livraisons, les API sont disponibles sous `/api/v1/delivery/` ;
- Pour les releases, les API sont disponibles sous `/api/v1/delivery/{id}/release/` ;

8.4.1. La génération des SDK

La génération des SDK est un processus long qui ne doit absolument pas bloquer l'exécution du serveur API. C'est pour cela qu'une tâche asynchrone Celery est exécutée sur un worker.

La méthode `generate_library()` du fichier `tasks.py` définit cette tâche. Actuellement, une seule tâche de génération existe. C'est celle pour Android. La tâche de génération iOS sera similaire.

FIGURE 37 : DIAGRAMME DE SÉQUENCE DE LA GÉNÉRATION DU SDK SMARTSCAN ANDROID



Source : Schéma réalisé par l'auteur

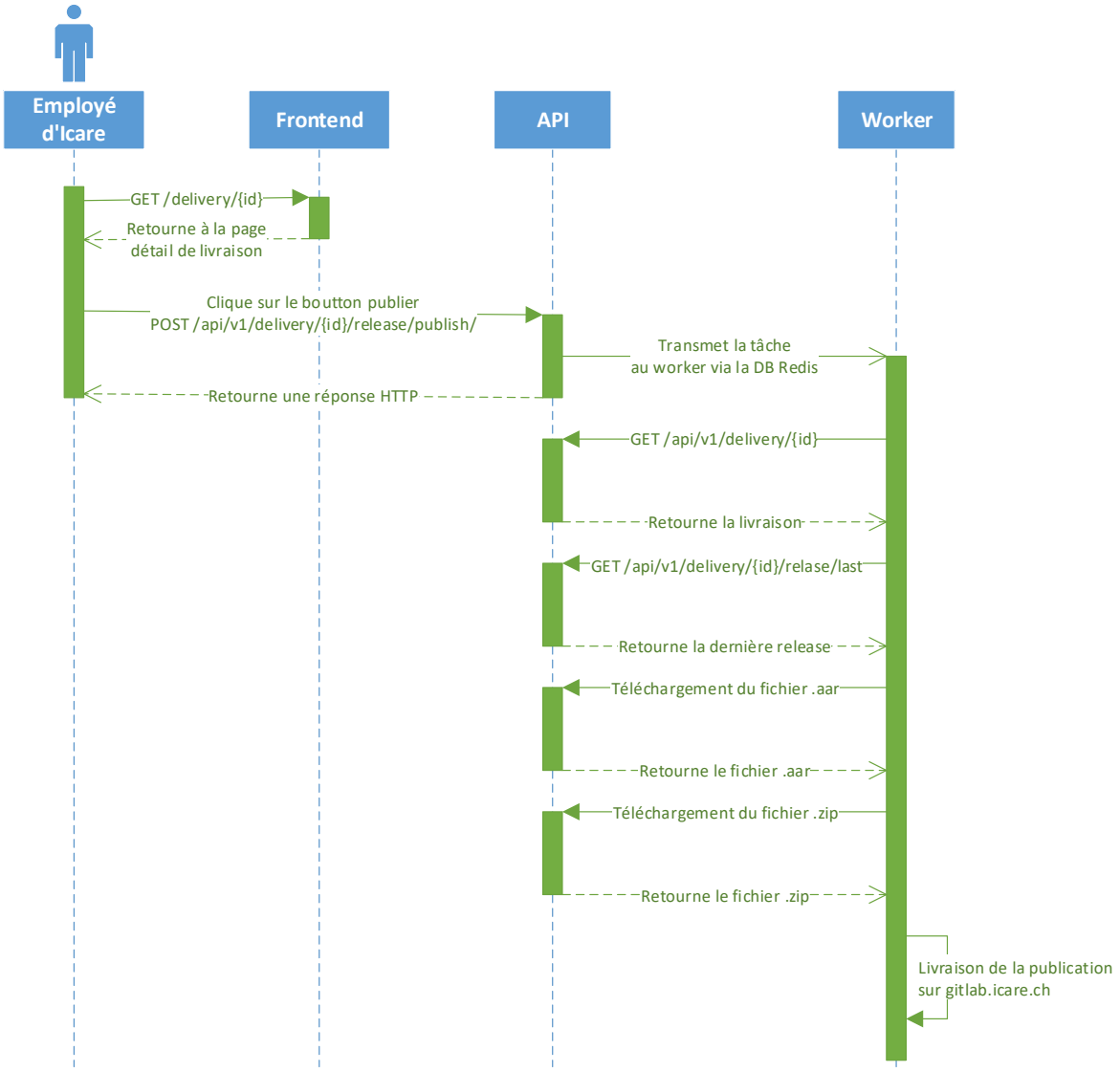
Ce processus débute lors de l'appel `POST` de l'API `/api/v1/delivery/`. Comme vu précédemment, la tâche est transmise à la DB Redis après la sauvegarde d'une livraison. Le worker écoute les messages transmis à la file « `android` » dès son lancement. Une fois le message reçu au worker, la génération peut commencer. La tâche va :

1. Mettre à jour le dépôt `smartscan_decoders` ;
2. Nettoyer le répertoire de construction Android ;
3. Générer quatre fichiers qui sont utilisés pour compiler les bons décodeurs et les versions adéquates :
    a. `DecodersConf.cmake` ;
    b. `Icare_decoders_private_codes.hpp` ;
    c. `IcareDecoders.java` ;
    d. `IcareDecodersCodes.hpp` ;
4. Générer le SDK :
    a. Construction de l'assemble externe pour smartscan (fichier `.aar`) ;
    b. Construction de l'assemble icare_decoders (fichier `.zip`) ;
5. Téléchargement du SDK smartscan vers l'API ;

8.4.2. La livraison des SDK

La livraison des SDK n'est pas très longue, mais elle nécessite d'avoir tous les dépôts de livraison clonés et téléchargés des SDK vers le serveur de source. C'est pourquoi cette tâche est exécutée de façon asynchrone. En effet, si un problème survient, l'application peut relancer facilement et automatiquement une tâche échouée ce qui permet de garantir que la livraison est effective.

FIGURE 38 : DIAGRAMME DE SÉQUENCE DE LIVRAISON D'UN SDK SMARTSCAN ANDROID



Source : Schéma réalisé par l'auteur

Ce processus démarre lorsque qu'un utilisateur clique sur le bouton « publier », ce qui envoie une requête API POST `/api/v1/delivery/{id}/release/publish/`. La tâche est transmise à la DB Redis. Le worker écoute les messages transférés à la file

« `publish` » dès son lancement. Une fois le message reçu au worker, la publication peut commencer. La tâche va :

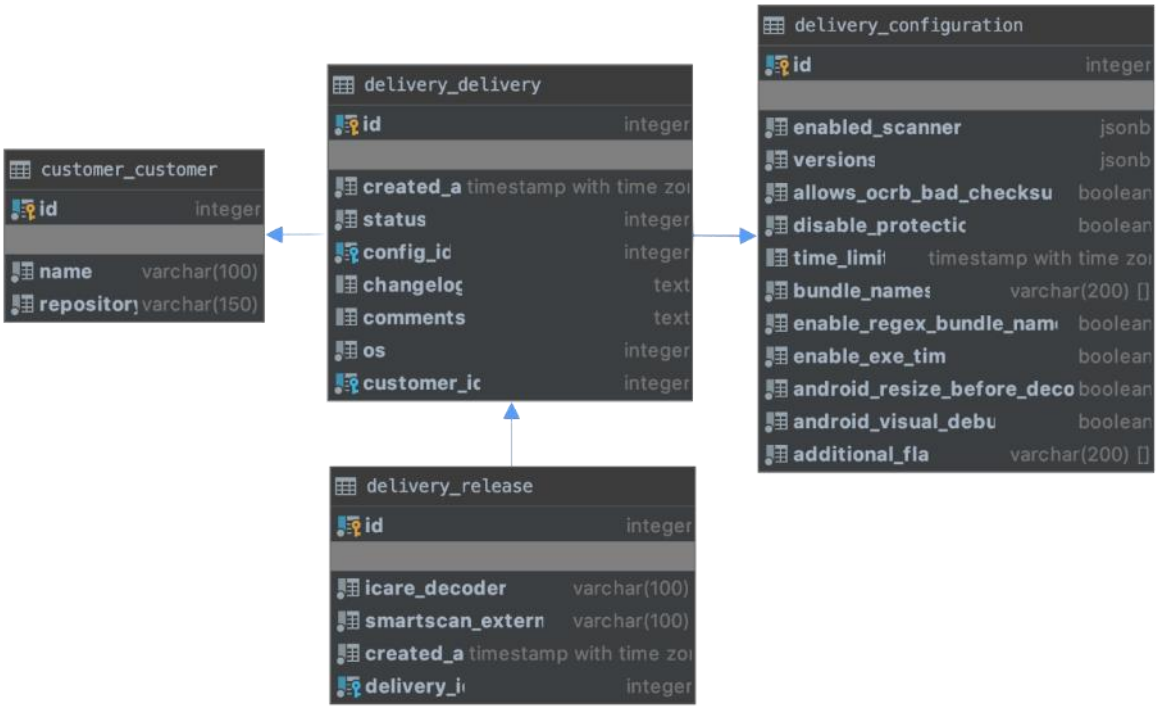
1. Récupérer le détail de la livraison pour obtenir l'utilisateur ;
2. Récupérer la dernière release liée à la livraison ;
3. Télécharger le fichier `.arr` ;
4. Télécharger le fichier `.zip` ;
5. Transmettre les fichiers téléchargés dans le répertoire du dépôt Git pour ensuite se synchroniser et publier le serveur Gitlab ;

8.5. Base de données

La DB de Genica est composée de quatre tables. Elles séparent les configurations, les livraisons, les releases (fichier `.aar` et fichier `.zip` du SDK Android) et les clients (support minimal). Toutes les tables sont nommées de la même façon, à savoir `{app_name}_{model_name}`. Pour parfaire ce schéma relationnel, des clés étrangères sont utilisées afin de relier les informations :

- Relation `1:0` entre `delivery_delivery` et `delivery_configuration` dissocie les éléments de configuration dans des tables distinctes et des livraisons sont réutilisables dans d'autres livraisons ;
- Relation 1:n entre delivery_delivery et customer_customer évite les duplications de données des clients, car seul l'identifiant est utilisé ;
- Relation 1:n entre delivery_relase et delivery_delivery existe, car il est possible d'avoir plusieurs générations de SDK pour une seule livraison ;

FIGURE 39 : SCHÉMA RELATIONNEL DE LA BASE DE DONNÉES



Source : Schéma réalisé par l'auteur

La DB se crée avec la commande `python manage.py migrate`. Cette instruction exécute les migrations situées dans les répertoires `migrations` de chaque application. Ces dernières sont générées selon les modèles avec la commande `python manage.py makemigrations <app_name>`.

8.6. Test unitaire

Les tests unitaires sont effectués avec pytest. Cette librairie permet de facilement réaliser des assertions, de plus l'utilisation de mock permet de simuler un objet et de reproduire un aspect réel de manière contrôlée.

FIGURE 40 : EXTRAIT DE CODE DE DELIVERY/TESTS/UTILS_TESTS/TEST_GIT_TOOL. PY

```
class TestGitTool:
    @pytest.fixture(autouse=True)
    def setup(self, mocker, tmpdir):
        mocker.patch('git.Repo.clone_from')
        self.base_path = tmpdir.mkdir("git")
        self.repo_url = 'git@gitlab.icare.ch:open-source/monitoring.git'
        self.project_path = os.path.join(self.base_path, 'monitoring')
        self.git_tool = GitTool(self.repo_url, self.base_path)

    def test_init(self, mocker):
        mock_clone_from = mocker.patch('git.Repo.clone_from')
        GitTool(self.repo_url, self.base_path)
        assert os.path.isdir(self.base_path)
        mock_clone_from.called_once_with(
            url=self.repo_url,
            to_path=self.project_path,
            origin='origin'
        )

    def test_init_base_path_not_exist(self, mocker, tmpdir):
        mock_clone_from = mocker.patch('git.Repo.clone_from')
        assert not os.path.isdir(os.path.join(tmpdir, 'base_path'))
        GitTool(self.repo_url, os.path.join(tmpdir, 'base_path'))
        assert os.path.isdir(os.path.join(tmpdir, 'base_path'))
        mock_clone_from.called_once_with(
            url=self.repo_url,
            to_path='base_path',
            origin='origin'
        )

...
```

Source : code source provenant de l'auteur

Pour connaitre le taux de couverture du code Python, la librairie coveragec est utilisée. Ce module a également la possibilité de générer un rapport HTML. Ces pages affichent les lignes testées ou non.

FIGURE 41 : EXÉCUTION ET RÉSULTAT DES TESTS UNITAIRES

```
~/G/g/genica > on develop *3 !7 ?1  docker exec -it dcc coverage run -m pytest
========================================= test session starts =========================================
platform linux -- Python 3.8.3, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
django: settings: genica_backend.settings (from ini)
rootdir: /usr/src/api, configfile: pytest.ini
plugins: mock-3.1.0, django-3.9.0, celery-4.4.2
collected 26 items

delivery/tests/test_android_utils.py ........                                          [ 30%]
delivery/tests/generator_test/test_cmake.py ....                                       [ 46%]
delivery/tests/generator_test/test_header.py .........                                 [ 80%]
delivery/tests/utils_tests/test_git_tool.py .....                                      [100%]

========================================== 26 passed in 3.24s ==========================================
Coverage.py warning: No data was collected. (no-data-collected)
```

Source : capture d'écran réalisée par l'auteur

FIGURE 42 : RAPPORT DE COUVERTURE

```
~/Git/genica/genica > on develop *3 !8 ?1  docker exec -it dcc coverage report -m
Name                                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------------------------
customer/__init__.py                              0      0   100%
customer/apps.py                                  3      3     0%   1-5
customer/models/__init__.py                       2      0   100%
customer/models/customer.py                       4      0   100%
customer/serializers/__init__.py                  2      2     0%   1-3
customer/serializers/customer_serializer.py       6      6     0%   1-8
customer/urls.py                                  9      9     0%   1-16
customer/views.py                                 7      7     0%   1-9
delivery/__init__.py                              0      0   100%
delivery/admin.py                                 1      1     0%   1
delivery/android_utils.py                        54     12    78%   33-43, 70-71, 88-91, 95-98
delivery/apps.py                                  3      3     0%   1-5
delivery/checks.py                               17     17     0%   1-28
delivery/generator/__init__.py                    0      0   100%
delivery/generator/cmake.py                      62      2    97%   19-20
delivery/generator/cmake_config.py                4      0   100%
delivery/generator/header_file.py                55      6    89%   14-15, 22-23, 34, 63
delivery/generator/java_file.py                   8      0   100%
delivery/generator/private_hpp_file.py           23      0   100%
delivery/generator/public_hpp_file.py             6      0   100%
delivery/models/__init__.py                       4      0   100%
delivery/models/configuration.py                 14      0   100%
delivery/models/delivery.py                      11      0   100%
delivery/models/release.py                        7      0   100%
delivery/serializers.py                          32     32     0%   1-47
delivery/tasks.py                               130    130     0%   1-220
delivery/urls.py                                 12     12     0%   1-23
delivery/utils/__init__.py                        2      0   100%
delivery/utils/git_tool.py                       59     19    68%   26-40, 56, 59-61, 77, 86, 89-90
delivery/views.py                                63     63     0%   1-85
genica_backend/__init__.py                        3      0   100%
genica_backend/asgi.py                            4      4     0%   10-16
genica_backend/celery.py                         28      9    68%   36-38, 42-49
genica_backend/settings.py                       58     10    83%   24, 28, 49, 90-98, 153
genica_backend/urls.py                            8      8     0%   16-33
genica_backend/wsgi.py                            4      4     0%   10-16
-----------------------------------------------------------------------------
TOTAL                                           705    359    49%
```

Source : capture d'écran réalisée par l'auteur

FIGURE 43 : RAPPORT DE COUVERTURE HTML

Coverage for **genica_backend/celery.py** : 68%

28 statements | 19 run | 9 missing | 0 excluded

```
1  from __future__ import absolute_import, unicode_literals
2
3  import json
4  import os
5  import logging
6  import requests
7  from celery import Celery
8  from celery.signals import celeryd_init
9  from urllib.parse import urljoin
10 from delivery.utils import GitTool
11
12 from django.conf import settings
13
14 logger = logging.getLogger(__name__)
15
16 # set the default Django settings module for the 'celery' program.
17
18
19 os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'genica_backend.settings')
20 os.environ.setdefault('FORKED_BY_MULTIPROCESSING', '1')
21
22 app = Celery('task')
23
24 # Using a string here means the worker doesn't have to serialize
25 # the configuration object to child processes.
26 # - namespace='CELERY' means all celery-related configuration keys
27 #   should have a `CELERY_` prefix.
28 app.config_from_object('django.conf:settings', namespace='CELERY')
29
30 # Load task modules from all registered Django app configs.
31 app.autodiscover_tasks()
32
33
34 @celeryd_init.connect
35 def configure_workers(sender=None, conf=None, **kwargs):
36     for customer in get_customers():
37         repo = GitTool(customer['repository'], '/usr/src/')
38         repo.fetch()
39
40
41 def get_customers():
42     url = urljoin(settings.API_ENDPOINT, f"/api/v1/customers/")
43
44     response = requests.get(url)
45     if response.status_code != 200:
46         logger.error(f"Expected status code 200, but got {response.status_code} with url {url}")
47         raise requests.ConnectionError(f"Expected status code 200, but got {response.status_code} with url {url}")
48
49     return json.loads(response.text)
```

Source : capture d'écran réalisée par l'auteur

9. Le frontend

L'interface graphique est réalisée avec Vue.js, comme décidé au chapitre 2.2. Cependant, d'autres librairies sont utilisées afin d'accélérer la programmation :

- NuxtJS ;
- Vuetify ;

9.1. NuxtJS

NuxtJS est un framework basé sur Vue.js. Il donne une structure aux projets tout en restant modulaire. Il propose des outils de déploiement afin de s'affranchir des questions de distribution. Ce framework est employé pour réaliser des applications SPA comme ici ou procéder à des rendus serveurs.

C'est un framework léger d'à peine 60 Ko une fois minifié. D'ailleurs, l'empaquetage et la minification du code sont opérés à l'aide de webpack et de vue-loader. D'autres librairies sont incluses de base :

- Vue2 ;
- Vue router ;
- Vuex ;
- Vue meta ;

Les fonctionnalités principales sont :

- Routage automatique ;
- Serveur de fichiers statiques ;
- Empaquetage et minification des fichiers JS et CSS ;
- Rechargement à chaud pendant le développement ;
- Structure modulaire ;

Dans ce projet NuxtJS offre un gain de temps pour le développement et la mise en production ainsi qu'une structure ordonnée et intuitive.

(NuxtJS, s.d.)

9.1.1. Store

Un store est un modèle de gestion d'états utilisant la librairie vuex. Ce module a pour but de centraliser des données pour tous les composants de l'application tout en garantissant l'état. De tels modèles sont pratiques dans de grands programmes.

FIGURE 44 : STORE ERRORNOTIFICATION.JS

```
export const state = () => ({
  message: ''
})

export const mutations = {
  setMessage (state, message) {
    state.message = message
  }
}
```

Source : code source provenant de l'auteur

Actuellement, la plateforme utilise un seul store qui stocke le dernier message d'erreur afin que tous les composants aient accès dans l'intention de l'afficher sous forme de popup.

9.1.2. Internationalisation

L'internationalisation adapte la langue de l'interface selon l'utilisateur ou le navigateur. Pour cela, le module `nuxt-i18n` est employé. Dans le cas de la plateforme Genica, le fichier de langue établi sépare les chaines de caractères des composants. De ce fait, une autre langue est plus facilement ajoutée.

FIGURE 45 : EXTRAIT DU FICHIER NUXT.CONFIG.JS

```
i18n: {
  locales: [
    {code: 'fr', file: 'fr_CH.js'}
  ],
  strategy: 'no_prefix',
  langDir: 'locales/',
  defaultLocale: 'fr',
  detectBrowserLanguage: false,
  vueI18n: {
    fallbackLocale: 'fr',
    messages: {fr}
  },
},
```

Source : code source provenant de l'auteur

Dans le fichier `nuxt.config.js`, l'on ajoute la configuration ci-dessus. Les paramètres principaux sont :

- Le paramètre `locales` configure la liste des langues disponibles ;
- Le paramètre `langDir` est le répertoire où sont stockés les fichiers de langue ;
- Le paramètre `defaultLocale` spécifie la langue de défaut ;

9.2. Vuetify

« Vuetify est la bibliothèque de composant #1 pour Vue.js » (Vuetify, s.d.). C'est-à-dire que cette bibliothèque procure tous les éléments graphiques pour construire une application web. Ces éléments sont basés sur les spécifications Material Design dictées par Google. Elle offre toutes les pièces utilisées dans les mockups de Genica.

9.3. Composant monofichier Vue.js

Un composant Vue.js est généralement décrit dans un seul et unique fichier `.vue`. Cependant, la séparation des composants en différents fichiers est possible, mais cela n'est pas recommandé et apporte les désavantages suivants :

- Les templates HTML se trouvent être des chaines de caractère ;
- Les templates HTML ne bénéficient plus de la coloration syntaxique ;
- Perte du support CSS ;
- L'utilisation des préprocesseurs est restreinte ;

FIGURE 46 : EXEMPLE D'UN COMPOSANT VUE.JS

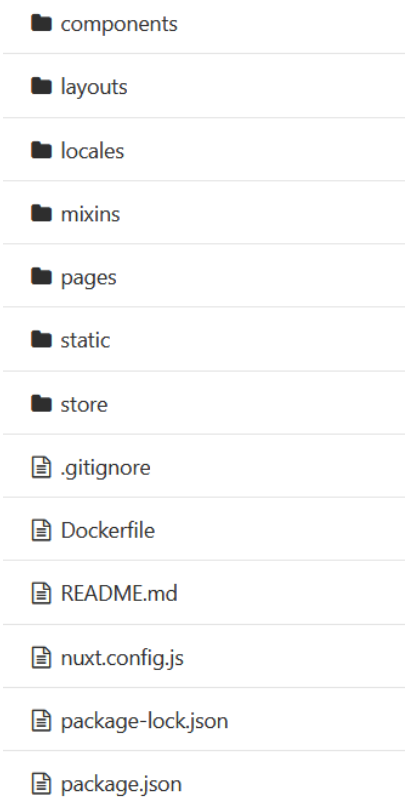
```
<template>
  <div v-if="os === 0">
    <v-icon large>mdi-android</v-icon>
  </div>
  <div v-else>
    <v-icon large>mdi-apple</v-icon>
  </div>
</template>

<script>
  export default {
    name: "OsIcon",
    props: ["os"]
  }
</script>
```

Source : code source provenant de l'auteur

« Dans le développement des interfaces utilisateur modernes, nous avons constaté que plutôt que de diviser tout notre code en trois grosses couches distinctes interdépendantes, il était plus intuitif de le scinder en petits composants faiblement couplés, et de les combiner. Au sein d'un composant, son gabarit, sa logique et ses styles sont intrinsèquement jumelés, et les réunir rend en réalité le composant plus cohérent et facile à maintenir. » (Vue.js, s.d.)

9.4. Structure

FIGURE 47 : STRUCTURE DU DOSSIER « FRONTEND »



Source : capture d'écran réalisée par l'auteur

La structure du répertoire `frontend` est composée de différents dossiers :

- Le répertoire `components` contient les composants Vue.js créés pour la plateforme ;
- Le répertoire `layouts` contient les templates de l'application ;
- Le répertoire `locales` n'est pas un dossier de base de NuxtJS. Il a été rajouté pour les fichiers de langues ;
- Le répertoire `mixins` n'est pas un dossier de base de NuxtJS. C'est ici que sont placées les fonctions utilisables par n'importe quels composants ;
- Le répertoire `pages` contient les pages de la plateforme et la structure interne de ce répertoire détermine la navigation. Ce sont tous des fichiers `.vue` ;
- Le répertoire `static` contient les fichiers statiques comme le logo ou le favicon. Pour le moment, c'est ici que se trouve la liste des décodeurs ;
- Le répertoire `store` contient les fichiers Vuex Store permettant de gérer certaines données ;

9.5. Le conteneur

9.5.1. Dockerfile de développement

Le `Dockerfile` utilisé lors du développement se situe dans le répertoire `/frontend/` du projet. Pour rappel, le code source est disponible dans la clé USB (voir annexe 4). Ce conteneur est uniquement employé dans l'environnement local et aucunement sur un autre écosystème. Cela peut représenter un risque de sécurité, car le code est exécuté par l'utilisateur `root`. De plus, l'interface est automatiquement rafraîchie à chaque modification d'un fichier JS.

Ce conteneur se base sur l'image officielle `node:lts-alpine`. Elle reprend la dernière version de Node.js et est basée sur la distribution alpine ultra légère. Ce `Dockerfile` :

- Copie les fichiers `package*. json` ;
- Installe les dépendances JS ;
- Copie le code ;
- Exécute le serveur avec la commande `` `npm run dev` `` ;

9.5.2. Dockerfile de production

Ce conteneur se base également sur l'image officielle `node:lts-alpine`. Le `Dockerfile` est accessible dans le dossier `/ci/docker/frontend/` du projet. Pour rappel, le code source est disponible dans la clé USB (voir annexe 4). À la différence du conteneur de développement, la première étape consiste à créer un utilisateur spécifique pour l'application. La deuxième se trouve au démarrage du programme. La commande de construction des composants et l'exécution du serveur se trouve dans un script `entrypoint.sh`. Ce script exploite les commandes suivantes :

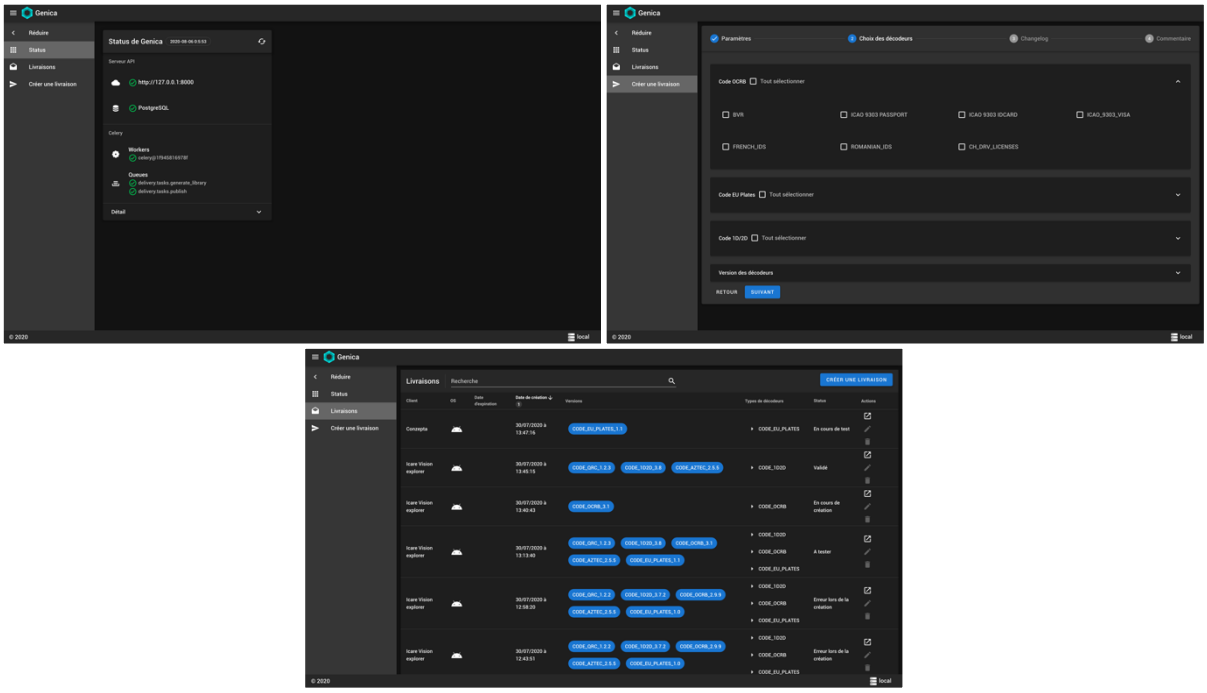
- Construction des composants `` `npm run build` `` ;
- Lancement du serveur web `` `npm start` `` ;

9.6. L'interface graphique

L'interface graphique est composée de quatre pages :

- Page d'accueil `/` ;
- Page listant toutes les livraisons `/deliveries` ;
- Page affichant les détails d'une livraison selon son identifiant `/deliveries/{id}/` ;
- Formulaire d'ajout de livraison séparé en quatre sections `/deliveries/add/` ;

FIGURE 48 : INTERFACE RÉELLE



Source : capture d'écran réalisée par l'auteur

10. Recommandations

10.1.1. Backend

Le conteneur worker est à revoir, car il est trop volumineux. Il y a sûrement une possibilité de réduire son volume et de le simplifier. La version d'openCV est fixe dans le conteneur. Elle est utilisée pour compiler le code source des décodeurs. Néanmoins, la plateforme devrait donner la possibilité de spécifier la version souhaitée.

Le taux de couverture actuelle de 52% est faible, car les tests unitaires ont pris plus de temps que prévu. Le test de certaines fonctionnalités a nécessité l'utilisation de mock et il a d'abord fallu comprendre son usage. Le taux de couverture devrait s'approcher des 100% afin de garantir un code sûr.

Pour le moment, les statuts des livraisons ne sont pas gérés de façon optimale, ils sont hard codé. C'est pour cela qu'un tuple ou une table spécifique doit être utilisé pour les statuts d'une livraison.

Les URLs ne respectent pas tout à fait les bonnes pratiques REST. En effet, la barre oblique de fin lors de l'appel API devrait être supprimée. Les URLs doivent être mises au pluriel, même celles retournant le détail d'un modèle comme `/api/v1/deliveries/{id}/`.

10.1.2. Frontend

Les appels API se font directement depuis les composants des pages. Il est possible que certains appels soient appelés dans différentes pages. C'est pour cela que ces requêtes sont abstraites pour éviter la duplication du code.

La page du formulaire d'ajout d'une livraison est composée de plusieurs étapes qu'il faudrait séparer en plusieurs composants.

Actuellement, aucun Store Vuex n'est utilisé pour les livraisons, mais cela pourrait présenter un certain nombre d'avantages d'en créer un. Notamment, la centralisation de la logique business des livraisons.

11. Principales difficultés rencontrées

La première problématique rencontrée était la mise en place de la CI/CD de Gitlab. Plusieurs essais ont été nécessaires pour réussir la construction des différents conteneurs et leur publication. Concernant le déploiement des environnements, un serveur virtuel CentOS a dû être créé et configuré. Cela car Rancher n'a pas été mis en place comme planifiée. Les déploiements sur l'environnement de production ont été effectués rarement.

La deuxième problématique est décrite au chapitre 4.4 relatif à la transmission des tâches de génération. Lors de l'implémentation, la première analyse s'est avérée incorrecte. Les websockets ne sont pas prévus pour cette fonctionnalité. Pour finir, après une deuxième réflexion, c'est la communication passant par une DB publish-subscrib qui a été retenue.

Une autre problématique fut le changement de version d'openCV en fin juillet pour la compilation des décodeurs. Ce changement a cassé toutes les générations des SDK. Le problème a été résolu en modifiant la version d'openCV du worker et des décodeurs.

Le dernier problème important concerne les tests unitaires. Certaines méthodes se sont révélées difficilement testables, car ces méthodes réalisaient des appels à ressource externe. C'est pour cela que ces appels ont dû être simulés à l'aide de la librairie mock. Toutefois son utilisation s'est avérée compliquée et le temps d'apprentissage long. En conséquence, le taux de couverture est bas.

12. Conclusion

Ce travail est un succès et beaucoup d'éléments sont mis en place pour le futur. Toutefois, ce n'est pas la totalité du processus de release des SDK smartscan qui est automatisée. Mais les concepts ainsi que les analyses permettent de poursuivre ce projet. De plus, la génération Android des SDK smartscan a ôté les doutes sur la faisabilité du projet.

L'utilisation de Django REST Framework a été le bon choix, bien que Python ne soit pas un langage asynchrone. Cela a poussé à chercher une solution pour ne pas bloquer le serveur. Finalement, la solution trouvée se révèle meilleure. Cela étant donné que les tâches sont gardées en mémoire et si aucun worker n'est actif, le message est traité plus tard. Ce framework a facilité l'intégration d'une partie des scripts exploités auparavant pour la génération.

Malheureusement, il s'est avéré que l'utilisation de plateforme de test mobile profitant des appareils réels n'est pas appropriée à ce genre de projet. Mais de telles plateformes peuvent se révéler intéressantes pour d'autres programmes au sein de l'institut Icare.

L'architecture de la plateforme est modulable et évolutive grâce à l'utilisation de conteneurs docker. Cela m'a permis d'apprendre énormément sur l'usage de Docker et d'une CI/CD.

Concernant la programmation de la plateforme Genica, toutes les user stories du product backlog ne sont pas terminées. Mais tous les musts sont réalisés. L'on peut donc dire que le but de l'application est atteint. Les prochaines livraisons Android aux clients se feront via la plateforme Genica. Les autres analyses faciliteront la continuation du projet au sein de l'institut Icare, notamment celle pour l'automatisation des tests du SDK smartscan.

Il reste du travail. Certaines parties ont été mises de côté lors du développement. Ces fonctionnalités sont peu abordées dans ce document. Surtout :

- La génération des SDK smartscan iOS ;
- L'authentification via le serveur LDAP de l'institut Icare ;

- La gestion des clients coordonnée avec le serveur Gitlab ;
- La gestion des décodeurs ;
- La section concernant les tests du bon fonctionnement du SDK smartscan ;

Cette plateforme est déployée dans deux environnements distincts accessibles aux URLs genica-int.icare.ch et genica.icare.ch. Le processus de génération des SDK Android est utilisable par les employeurs de l'institut Icare. Le processus iOS suivra.

Références

AltexSoft. (2018, Septembre 10). *Pros and Cons of ReactJS and React Native | AltexSoft*. Récupéré sur AltexSoft: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-reactjs-and-react-native/

AltexSoft. (2019, Septembre 11). *The Pros and Cons of Vue.js | AltexSoft*. Récupéré sur AltexSoft: https://www.altexsoft.com/blog/engineering/pros-and-cons-of-vue-js/

AltexSoft. (2020, Mars 25). *Pros and Cons of Angular Development | AltexSoft*. Récupéré sur AltexSoft: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/

Amazon. (s.d.). *AWS Device Farm - Amazon Web Service*. Récupéré sur Amazon: https://aws.amazon.com/fr/device-farm/

Angular. (s.d.). Récupéré sur Angular: https://angular.io/

Brent. (2019, Mai 10). *PHP in 2019*. Récupéré sur Stitcher.io: https://stitcher.io/blog/php-in-2019

Chazallet, S. (2019). *Python 3 - Les fondamentaux du langage (3e édition)*. ENI.

Docker. (s.d.). *What is a Container? | App Containerization | Docker*. Récupéré sur docker: https://www.docker.com/resources/what-container

Express - Node.js web application framework. (2017). Récupéré sur expressjs: https://expressjs.com/

Express/Node introduction. (2020, Mars 12). Récupéré sur MDN web docs: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction

Flanagan, D., & Matsumoto, Y. (2008). *The Ruby Programming Language*. O'Reilly Media.

Flask | The Pallets Project. (s.d.). Récupéré sur The Pallets Project: https://palletsprojects.com/p/flask/

Fontanet, J., & Lambert, O. (2015). *Node.js - Exploitez la puissance de JavaScript côté serveur*. ENI.

Google. (2020, Avril 8). *Firebase Test Lab*. Récupéré sur Firebase: https://firebase.google.com/docs/test-lab

Google. (s.d.). *Firebase Pricing*. Récupéré sur Firebase: https://firebase.google.com/pricing

Heurtel, O. (2019). *PHP et MySQL - Maîtrisez le développement d'un site web dynamique et interactif (4e édition)*. ENI.

Home - Django REST framework. (s.d.). Récupéré sur Djans REST framework: https://www.django-rest-framework.org/

Hussain, N. (2020, Avril 2). Communication personnelle. (K. Jäggi, Intervieweur)

Kobiton. (s.d.). *Kobiton - Mobile Device Testing Platform*. Récupéré sur Kobiton: https://kobiton.com/

Krief, M. (2019). *Learning DevOps*. Packt Publishing.

Laravel - The PHP Framework for Web Artisans. (s.d.). Récupéré sur Laravel: https://laravel.com/

LEITE, E. (2018). *DevOps - Intégrez et déployez en continu (avec exemple clés en main)*. ENI.

Leite, E. (2019). *DevOps - Intégrez et déployez en continu (avec exemple clés en main)*. Edition ENI.

Lumen - PHP Micro-Framework By Laravel. (s.d.). Récupéré sur Lumen: https://lumen.laravel.com/

NuxtJS. (s.d.). *Introduction - NuxtJS*. Récupéré sur NuxtJS: https://fr.nuxtjs.org/guide

Pereira, C. (2020, Avril 2). Communication personnelle. (K. Jäggi, Intervieweur)

Rancher. (s.d.). *Managed Kubernetes Platform | Rancher*. Récupéré sur Rancher: https://rancher.com/products/rancher/

Ravulavaru, A. (2017). *Learning Ionic*. Packt Publishing.

Ruby on Rails | A web-application framework that includes everything needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. (s.d.). Récupéré sur Ruby on Rails: https://rubyonrails.org/

Samson, P. (2020). *Django - Développez vos applications web en Python (fonctionnalités essentielles et bonnes pratiques)*. Edition ENI.

Subra, J.-P. (2019). *Scrum - Une méthode agile pour vos projets (3e édition)*. Editions ENI.

Tesanovic, V. (2018, Avril 12). *Multi threading and multiple process in Node.js*. Récupéré sur itnext: https://itnext.io/multi-threading-and-multi-process-in-node-js-ffa5bb5cde98

Vue.js. (s.d.). *Composants monofichiers*. Récupéré sur Vue.js: https://fr.vuejs.org/v2/guide/single-file-components.html

Vuetify. (s.d.). *Pourquoi Vuetify*. Récupéré sur Vuetify.js: https://vuetifyjs.com/fr-FR/introduction/why-vuetify/

Annexe 1 - Structure des groupes et des dépôts Git sur Gitlab

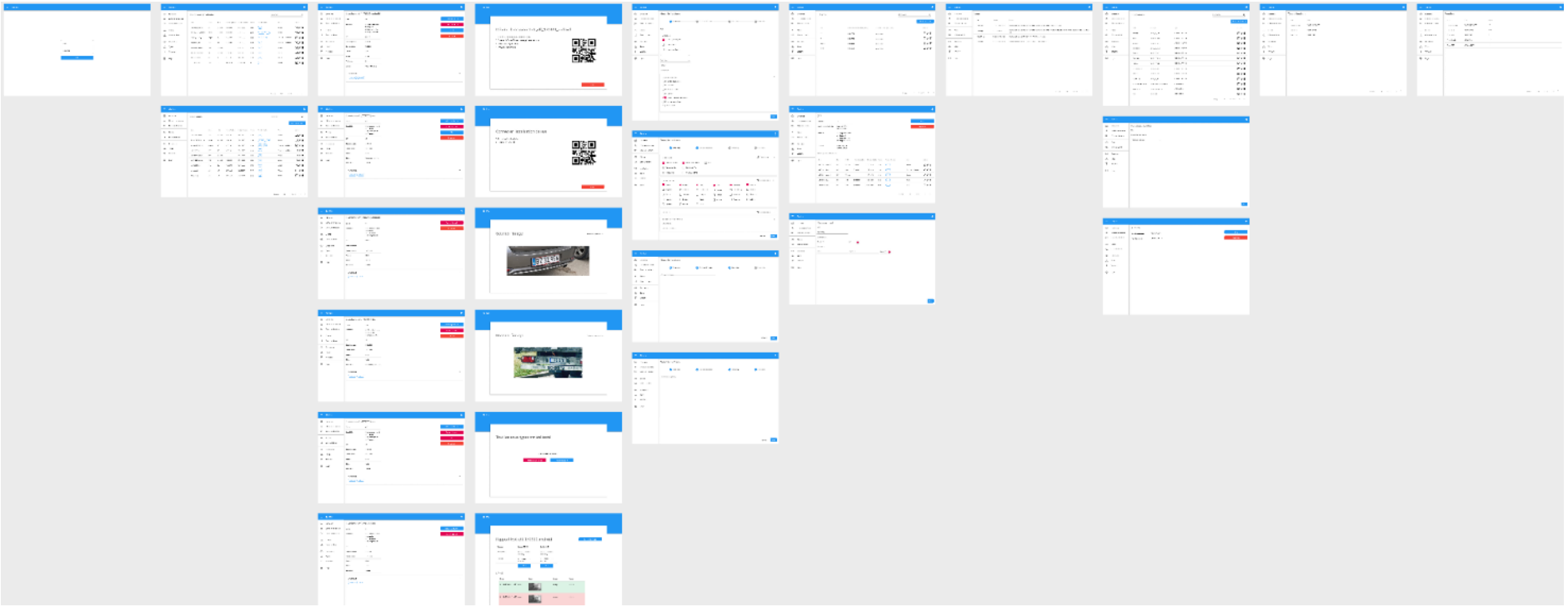
```
Computer vision
|-- Databases
|-- Icare decoders databases
|     |-- Data code database
|     |-- OCRB database
|     |-- Inscard database
|     |-- Aztec database
|     +-- Plates EU database
|-- The tester desktop
|-- The tester mac
|-- The tester ios
|-- The tester android
|-- Lateralflow tester
|-- Image database tools
|-- Icare decoders
|-- Icare decoders ios
|-- Icare decoders android
|-- Decoder configuration
|-- Smart scann builder
|-- Build Opencv for Android
|-- Decoder sources
|-- Qr code
|-- Qr_scripts
|-- Ocr
|-- Lateralflow
|-- Inscard
|-- Aztec
|-- Data code
|-- Plates
|-- Plate eu
|-- Plates EU annotation
|-- Plates EU script
|-- Ann xml2hpp
|-- Ann training
|-- Anntranning country
|-- Ndk aztec
|-- Ndk qr code
|-- Code kit exemple
|-- Vein tree
|-- Checksum checker
|-- Gazecontrol eplorationscript
+-- Test git lfs
```

SmartID Scan

```
|-- smartidscan_flutter_demo
|-- smartidscan_reactnative_demo
|-- augurix_proto
|-- SmartID NFS Passport IOS
|-- smartidscan_xamarin_android_demo
|-- smartidscan_ios_demo
|-- smartidscan_xamarin_android
|-- smartidscan_android_demo
|-- smartidscan_android
|-- smart_vision_explorer_ios
|-- smartidscan_ios
+-- smartscan_windows
```

Annexe 2 – Dernière version des mockups

Un fichier PDF plus lisible se trouve sur la clé USB sous `annexes/mockup_genica_v.1.3.pdf`.



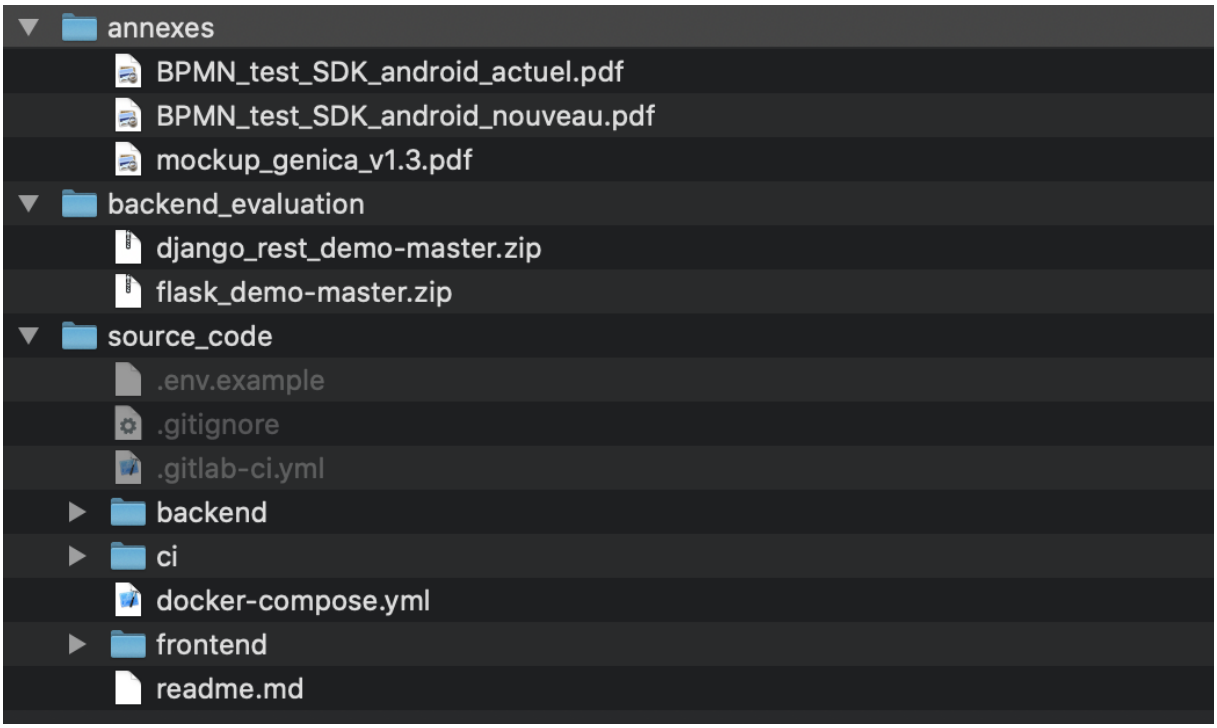
Annexe 3 - gitlab-ci.yml

```yaml
image: python:3.8-slim

stages:
  - test
  - build
  - deploy

##############
# Test stage #
##############

test:api:
  stage: test
  before_script:
    - apt-get update -y && apt-get install git -y
    - cp ${ENV_TEST} backend/.env
    - cat backend/.env
    - pip install -r backend/requirements.txt
    - mkdir -p /root/.ssh
    - chmod 700 /root/.ssh
    - touch /root/.ssh/known_hosts
    - chmod 600 /root/.ssh/known_hosts
    - ssh-keyscan gitlab.icare.ch > /root/.ssh/known_hosts
    - cd backend
  script:
    - coverage run -m pytest
    - coverage report -m

###############
# Build stage #
###############

.build: &build
  stage: build
  image: docker:19.03.8
  services:
    - docker:19.03.8-dind
  before_script:
    - echo ${PASSWORD} | docker login -u root --password-stdin ${CI_REGIS-
TRY}

build:api:dev:
  <<: *build
  script:
    - docker build --pull --no-cache -t ${CI_REGISTRY_IMAGE}/api:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-dev -t ${CI_REGISTRY_IMAGE}/api:dev -f
./ci/docker/backend/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/api:dev
only:
    - develop
```

```
build:api:latest:
  <<: *build
  script:
    - docker build --pull --no-cache -t ${CI_REGISTRY_IMAGE}/api:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-stable -t ${CI_REGISTRY_IMAGE}/api:latest -f
./ci/docker/backend/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/api:latest
  only:
    - master

build:worker:dev:
  <<: *build
  script:
    - docker build --pull -t ${CI_REGISTRY_IMAGE}/worker:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-dev -t ${CI_REGISTRY_IMAGE}/worker:dev -f
./ci/docker/worker/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/worker:dev
  when: manual
  only:
    - develop

build:worker:latest:
  <<: *build
  script:
    - docker build --pull --no-cache --build-arg SSH_PRIVATE_KEY="$(echo
$DEPLOY_KEY)" -t ${CI_REGISTRY_IMAGE}/worker:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-stable -t ${CI_REGISTRY_IMAGE}/worker:latest -f
./ci/docker/worker/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/worker:latest
  when: manual
  only:
    - master

build:ui:dev:
  <<: *build
  script:
    - echo "${CI_COMMIT_SHORT_SHA}" >> frontend/version
    - docker build --pull --no-cache -t ${CI_REGISTRY_IMAGE}/ui:${CI_COM-
MIT_SHORT_SHA}.${CI_JOB_ID}-dev -t ${CI_REGISTRY_IMAGE}/ui:dev -f
./ci/docker/frontend/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/ui:dev
  only:
    - develop

build:ui:latest:
  <<: *build
  script:
    - echo "${CI_COMMIT_SHORT_SHA}" >> frontend/version
    - docker build --pull --no-cache -t
${CI_REGISTRY_IMAGE}/ui:${CI_COMMIT_SHORT_SHA}.${CI_JOB_ID}-stable -t
${CI_REGISTRY_IMAGE}/ui:latest -f ./ci/docker/frontend/Dockerfile .
    - docker push ${CI_REGISTRY_IMAGE}/ui:latest
  only:
    - master
```

```
################
# Deploy stage #
################

.deploy: &deploy
  stage: deploy
  before_script:
    - apt-get update -y && apt-get install openssh-client -y
    - mkdir -p ~/.ssh
    - echo "$DEPLOY_KEY" | tr -d '\r' > ~/.ssh/id_rsa
    - chmod 700 ~/.ssh/id_rsa
    - eval "$(ssh-agent -s)"
    - ssh-add ~/.ssh/id_rsa
  script:
    - scp -o StrictHostKeyChecking=no $docker_compose_path
$user@$host:$destination_path/docker-compose.yml
    - scp -o StrictHostKeyChecking=no $ENV_INT $user@$host:$destina-
tion_path/.env
    - ssh -t -o StrictHostKeyChecking=no $user@$host “mkdir -p ${destina-
tion_pass}/.ssh”
    - scp -o StrictHostKeyChecking=no ~/.ssh/id_rsa $user@$host:$destina-
tion_path/.ssh/id_rsa
    - ssh -t -o StrictHostKeyChecking=no $user@$host "docker login -u root
-p ${PASSWORD} ${CI_REGISTRY}"
    - ssh -t -o StrictHostKeyChecking=no $user@$host “cd ${destina-
tion_pass} && docker-compose pull && docker-compose up -d”


deploy:int:
  <<: *deploy
  variables:
    user: icare
    host: 10.0.5.11
    port: 22
    destination_path: /home/icare/genica-int
    docker_compose_path: ./ci/environment/docker-compose.yml
  only:
    - develop

deploy:prod:
  <<: *deploy
  variables:
    user: icare
    host: 10.0.5.11
    port: 22
    destination_path: /home/icare/genica-prod
    docker_compose_path: ./ci/environment/docker-compose.yml
  when: manual
  only:
    - master
```

Annexe 4 - Structure des répertoires de la clé USB



Annexe 5 - Documentation technique

Requirements

Make sure you're familiar with these technologies:

- Docker
- Python (Django, Celery)
- Vue.js
- Nuxt

On your computer, it is necessary to have installed `docker` and `docker-compose`.

Make sure your SSH key is in `~/.ssh/id_rsa` and that you have access to the genica group on gitlab.icare.ch.

Folder organization

This project is composed of an API, a Frontend and one or several workers.

It is composed of 3 main folders:

- `./backend/`: contains the API and worker files (made with Django and Celery)
- `./frontend/`: contains the files related to the frontend (made with Vue.js and Nuxt)
- `./ci/`: contains the files relating to the gitlab CI/CD

Run the project on your computer

> Make sure you fill out the requirements!

1. Copy `.env.example` to `.env`
2. Update the environment variables in the `.env`
3. Build images: `docker-compose build --forece-rm`
4. Run containers: `docker-compose up -d`

> Steps 3 & 4 can be passed with a one line command (but generate `none` images): `docker-compose up --build -d`

Test API: http://localhost:8000. Test UI: http://localhost:3000.

Code changes apply automatically.

If you see a lot of `<none>` images when you run `docker image ls` You can delete it with: `docker rmi $(docker images | grep "^<none>" | awk "{print $3}")`

Without Docker

> It is also possible to run the project on your computer without docker. However, the configuration is more complex and undocumented in this file

`.env` for API

```
MODE=api
DEBUG=true|false
SECRET_KEY=my-secret-key
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1,hostname

# Database
SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=my-db
```

```
SQL_USER=my-user
SQL_PASSWORD=my-secret
SQL_HOST=127.0.0.1
SQL_PORT=5432

# Log
DJANGO_LOG_LEVEL=DEBUG
DJANGO_LOG_PATH=./log
DJANGO_LOG_FILENAME=genica.log

# Redis (for celery task)
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_CELERY_DB=0
```

`.env` for worker

```
MODE=worker
SECRET_KEY=my-secret-key
API_ENDPOINT=https://my-endpoint

# Log
DJANGO_LOG_LEVEL=DEBUG
DJANGO_LOG_PATH=./log
DJANGO_LOG_FILENAME=genica.log

# Redis (for celery task)
REDIS_HOST=redis
REDIS_PORT=6379
REDIS_CELERY_DB=0

# Smartscann android git folder
SMARTSCAN_ANDROID_PATH=/usr/src/smartscan_android
```

Unit test

1. Open a terminal
2. Search your API container with `docker ps`
3. Copy container ID
4. Commands

- run test `docker exec -it <id> coverage run -m pytest`
- view report `docker exec -it <id> coverage report -m`
- generate HTML report `docker exec -it <id> coverage html`

On environment

Integration

Integration https://genica-int.icare.ch:

- API: 10.0.5.11:8001
- UI: 10.0.5.11:3001

Change on develop branch will deploy automatically to integration environment.

Production

Production https://genica.icare.ch:

- API: 10.0.5.11:8080
- UI: 10.0.5.11:3000

Production will not deploy change automatically, you need:

1. Login into gitlab.icare.ch
2. Go to CI/CD job
3. Click execute deploy:prod

Add new environment

1. Create a server with `docker` and `docker-compose`
2. Add this public key to `.ssh/authorized_keys`


```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDRHv7VH6csAruNK8AC7FxXDdkEvC/6TcBO08JA/OYzsDegbW+QJaE4OUeeQ2assXm2kyrGnJXQ
PaqUcGOEPrySHtoWT05Fyq0ljNEMCd4O8LyQBk93K6O541qK3Bi062Pp92M4RgqqWl6CzlJbLj9aQ4MCduH59JrCYxCR9Zwik7NPeljUgpv4PHGb
tPxQ03SVBuUz6qOgJMMS2MXkV/2xBTP4w1O3o5io7NFSxbh9m4Rn0OyEw7rY4TNY31/BzWkR5AjX+F+YxcG+SwSRfIhNkedgF/7QQLMDts9Dztne
rQjSe3Ov2zIGKXgixO2WR61nXSs6A6wI0mg/MTRajmvCv2g4cSoaUutiSJEvTAE2+6vyC6Lm3/OpkiBEiLmFqHltmLezwNtA2EbRLz+QbPiJzbn0
/8IIlR0UfeD9/dTMmGNkH00yuD+FeX1Yp7IHhlxydV2f2ksL8Ydxss1qWZq2L4vl5/jE9qTpWQO4fRbK6VfD47YDuFBzCz+W1EDCSBc=
ken@MacBook-Pro-de-Jaggi.localken@MacBook-Pro-de-Jaggi.local
```


3. Create folder for docker environment `mkdir ~/<env_name>`
4. Add necessary variable in CI `settings -> CI -> Variables`
5. Add environment deploy step into `gitlab-ci.yml`
6. Configure the reverse proxy of Icare

Déclaration de l'auteur

Je déclare, par ce document, que j'ai effectué le travail de Bachelor ci-annexé seul, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de Bachelor, y compris au partenaire de recherche appliquée avec lequel j'ai collaboré, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail et que je cite ci-après :

- Benoit Golay ;
- Christian Pereira ;
- Nouredine Hussain ;
- Ainsi que les autres employés de l'institut Icare ;

Ken Jäggi