Ecole Polytechnique de l'Université François Rabelais de Tours
Département Informatique
64 avenue Jean Portalis
37200 Tours, France
Tél. +33 (0)2 47 36 14 14
polytech.univ-tours.fr

Rapport de stage

2017-2018

Réalisation d'un outil de cartographie sur le progiciel Amplitude

Entreprise
Sopra Banking

Étudiant
Romain ROUSSEAU (DI5)

Tuteur académique
Yannick KERGOSIEN

Tuteur entreprise
Alexandre DURAND (Responsable Cellule Architecture)

31 août 2018

Liste des intervenants

Entreprise

Sopra Banking
47 rue Christiaan Huygens
37073 Tours Cedex 2 - France
www.soprabanking.com

Nom	Email	Qualité
Romain ROUSSEAU	romain.rousseau@etu.univ-tours.fr	Étudiant DI5
Yannick KERGOSIEN	yannick.kergosien@univ-tours.fr	Tuteur académique, Département Informatique
Alexandre DURAND	alexandre.durand@soprabanking.com	Tuteur entreprise, Responsable Cellule Architecture

Avertissement

Ce document a été rédigé par Romain ROUSSEAU susnommé l'auteur.

L'entreprise Sopra Banking est représentée par Alexandre DURAND susnommé le tuteur entreprise.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Yannick KERGOSIEN susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnait assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnait qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.

Pour citer ce document

Romain ROUSSEAU, *Réalisation d'un outil de cartographie sur le progiciel Amplitude*, Rapport de stage, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2017-2018.

```
@mastersthesis{
    author={ROUSSEAU, Romain},
    title={Réalisation d'un outil de cartographie sur le progiciel Amplitude},
    type={Rapport de stage},
    school={Ecole Polytechnique de l'Université François Rabelais de Tours},
    address={Tours, France},
    year={2017-2018}
}
```

Document mis en forme sous LaTeX
grâce à la classe de document polytech.cls V2.2.5 (2017/05/02)

Table des matières

Liste des intervenants a

Avertissement b

Pour citer ce document c

Table des matières i

Table des figures iv

Liste des tableaux vi

Introduction 1

I Présentation de l'entreprise et du projet 3

1 Le groupe *Sopra Steria* et *Sopra Banking Software* 4

1 Histoire des deux groupes : *Sopra* et *Steria* 4

2 Fusion des deux groupes et création de *Sopra Steria* 5

3 La filiale *Sopra Banking Software* 5

2 Le progiciel Amplitude 6

1 Présentation 6

2 Architecture générale 6

3 Présentation du projet 9

1 Contexte 9

2 Expression du besoin 9

3 Description des exigences 10
4 Gestion de projet 10

II Tutoriels et formations **11**

4 Spring Boot **12**
1 Création d'un projet avec *Spring Initializr* 12
2 Utilisation avec *Maven* 13
3 Éléments à ajouter dans le programme 14
4 Exemple d'application web avec *Spring Boot* 15
5 Gestion de base de données 16
6 Déploiement d'une application *Spring Boot* x *Angular* 17
6.1 Déployer les parties séparément 18
6.2 Déploiement sous forme de *.war* avec *Maven* 18
6.3 Déploiement sous forme de *.jar* en utilisant *Spring Boot* 19

5 Spring Data **20**
1 Fonctionnement de base de *Spring Data* et de *Spring Data JPA* 20
2 Fonctionnement de *Spring Data REST* 21
3 Configuration avec *Spring Boot* et *Spring Data JPA* 22

6 Angular **23**
1 Installations nécessaires 23
1.1 *Node.js, npm* et *Angular CLI* 23
1.2 Quelques IDE intéressants pour le développement *Angular* 24
1.2.1 Angular IDE 24
1.2.2 WebStorm 24
1.2.3 Éditeur en ligne : *Stackblitz* 24
2 Structure d'un projet *Angular* 24
3 Gestion des styles : *Bootstrap* et *PrimeNG* 25
3.1 *Bootstrap* 26
3.2 *PrimeNG* 27

III Modélisation de l'outil et chiffrage du projet **28**

7 Modélisation UML **29**
1 Diagramme d'entité 29
2 Diagramme d'activité 30

8 Spécifications de l'outil de cartographie 32
1 Création du modèle 32
2 Alimentation de la base 33
2.1 Création des programmes en base 33
2.2 Création des modules 4GL en base 33
2.3 Création des fonctions techniques en base 33
2.4 Association des modules 4GL et des programmes 34
2.5 Association des fonctions et des programmes 34
2.6 Extraction du dictionnaire des programmes 34
2.7 Récupération des SMG 34

9 Chiffrage du projet 35

IV Développement de l'outil 37

10 Outils utilisés et arborescence du projet 38
1 IDE et outils de développement utilisés 38
2 Arborescence du projet 38

11 Exposition REST des données 41
1 Principe du REST 41
2 Implémentation avec *Spring Data* 41

12 Gestion du job d'alimentation 43

13 Gestion du front-end et de la sécurité 45
1 Gestion de la sécurité 45
2 Aperçu de l'affichage 45

Bilan du stage 47

Annexes 48

A Diagramme de classe 49

Table des figures

2 Le progiciel Amplitude

1 Cartographie fonctionnelle d'*Amplitude* 7

4 Spring Boot

1 Aperçu du site *Spring Initializr* 13
2 Extrait d'implémentation de la dépendance parent dans le *pom.xml* 13
3 Extrait d'implémentation d'une dépendance web dans le *pom.xml* 13
4 Implémentation du plug-in *Maven* 14
5 Fonction principale d'un projet *Spring Boot* 15
6 Fichier *pom.xml* en cas de multiples fonctions principales 15
7 Fichier *pom.xml* en cas de multiples fonctions principales 16
8 Exemple de dépendances dans le *pom.xml* pour la gestion d'une base de données . 17
9 Fichier de configuration pour l'initialisation d'une base de données 17
10 Fichier *pom.xml* d'un projet multi-module 18

5 Spring Data

1 Exemple de repository 21
2 Exemple de requête spécifié dans un *repository* 21
3 Exemple de requête modifiée 21
4 Exposition REST d'un *repository* 22

6 Angular

1 Arborescence d'un projet *Angular* 25
2 Sélection des styles en Angular 26

3 Exemple de page formatée avec *Bootstrap* 26
4 Exemple de styles utilisant *PrimeNG* 27
5 Implémentation HTML utilisant *PrimeNG* 27

7 Modélisation UML

1 Extrait de la partie Cartographie Technique du diagramme d'entité 30
2 Diagramme d'activité de l'alimentation des éléments de la cartographie 31

10 Outils utilisés et arborescence du projet

1 Arborescence du *back-end* de l'outil de cartographie 39

11 Exposition REST des données

1 Prise en charge du REST avec *Spring Data* 42

12 Gestion du job d'alimentation

1 Implémentation des méthodes d'alimentation 44

13 Gestion du front-end et de la sécurité

1 Page d'authentification de l'outil de cartographie 46
2 Page d'accueil de l'outil de cartographie 46

Liste des tableaux

9 Chiffrage du projet

1 Chiffrage des phases de développement de l'outil de cartographie 36

Introduction

Ce rapport présente mon stage d'assistant ingénieur réalisé dans l'entreprise *Sopra Banking Software* à Tours. Le sujet du stage était la réalisation d'un outil de cartographie pour le progiciel nommé *Amplitude* de l'entreprise. Le stage a débuté le 9 avril 2018 avant de s'achever le 31 août de la même année.

Le stage de cinquième année a pour but d'appliquer les connaissances acquises lors des précédentes années d'étude. Il doit représenter une synthèse de l'ensemble des caractéristiques qu'un ingénieur doit avoir dans un environnement concret. Alors que le stage de troisième année sert à découvrir le monde de l'entreprise et que celui de quatrième année sert à faire le premier pas vers le métier d'ingénieur, le stage de dernière année permet de travailler dans une équipe installée pour mener à bien un projet, dans son intégralité ou tout du moins sa majorité selon les cas. Ce stage est censé donner une autonomie supérieure aux étudiants, qui doivent prendre des décisions et avoir la possibilité de donner leur avis sur les opérations en cours, ce qui en fait ainsi, une expérience nécessaire pour le métier.

Il se déroule sur une période d'au minimum 18 semaines, ce qui permet de pouvoir réaliser des projets avec une envergure dont les étudiants-ingénieurs n'ont jamais eu l'occasion de faire durant leur cursus. Il s'agit du stage le plus important de la formation car il s'agit du plus complet, du plus formateur et de manière générale, du plus intéressant. Il apporte une expérience non négligeable et représente la dernière marche avant l'arrivée dans la vie active.

Le secteur de l'informatique est très porteur et les offres de stage sont donc très nombreuses et variées. L'opportunité de stage dans l'entreprise *Sopra Banking Software* est venue lors du Forum des Entreprises qui se déroule à Polytech pendant le mois de novembre. J'ai pu y rencontrer de nombreux acteurs du secteur et obtenir des entretiens avec plusieurs d'entre eux, dont notamment deux à Sopra sur le site de Tours. À la suite de ces entretiens, j'ai pu faire mon choix parmi les propositions. J'ai retenu celle de la Cellule Architecture de *Sopra Banking Software* pour plusieurs raisons. Tout d'abord, le domaine de la banque m'a toujours intéressé, j'ai déjà fait mon stage de 4ème année chez Worldline du côté des paiements en ligne et je voulais découvrir un autre aspect du domaine. Ensuite, j'avais envie de faire partie d'un projet que ne soit pas monotone et qui me permette d'acquérir de nouvelles connaissances sur des technologies que je ne maîtrise pas ou très peu. Et c'était le cas avec cette offre qui combinait tous les aspects d'un projet, de l'expression des besoins jusqu'au développement en passant par la modélisation ou encore le chiffrage, le tout en utilisant des technologies récentes, variées et performantes.

Les chapitres qui vont suivre exposeront les principales phases de mon stage, en commençant

d'abord par une présentation de l'entreprise et du projet en lui-même. La suite sera consacrée à mon parcours au sein de l'équipe, avec dans un premier temps, une phase de formation sur les différents éléments de l'entreprise ainsi que sur les technologies qui seront utilisées ; puis, à la modélisation de l'outil, avec l'élaboration des diagrammes de Gantt, et le chiffrage du projet ; et enfin le développement en lui-même.

Première partie

Présentation de l'entreprise et du projet

Cette partie sera consacrée à la présentation du cadre général de mon stage, c'est-à-dire le groupe *Sopra Steria*, sa filiale *Sopra Banking Software*, le progiciel *Amplitude* et la présentation générale du sujet.

1 Le groupe *Sopra Steria* et *Sopra Banking Software*

1 Histoire des deux groupes : *Sopra* et *Steria*

Sopra Steria est une entreprise de services du numérique qui propose des prestations de conseils, des services technologiques et qui édite des logiciels métiers dans trois principaux domaines : les ressources humaines, l'immobilier et la banque. Définie comme leader européen de la transformation numérique, l'entreprise, de par sa diversité, propose l'un des portefeuilles d'offres les plus complets du marché. L'une des principales motivations du groupe est d'accompagner ses clients dans leur transformation numérique et les aider à faire le meilleur usage de leurs outils. Parmi les actuels clients de l'entreprise, on peut citer des grands noms comme Airbus, La Banque Postale, le Ministère de la Défense, EDF, le Ministère des Finances, la SNCF, Easyjet, et bien d'autres.

Comme beaucoup de SSII à grandes envergures, *Sopra Steria* est le fruit de nombreuses acquisitions et fusions au fil du temps. Tout d'abord, *Sopra* et *Steria* composaient deux entreprises distinctes. Elles ont toutes les deux été créées à la fin des années 70 (en 1968 pour *Sopra* et en *1968* pour *Steria*). Les deux entreprises se développent dans les années qui suivent. *Sopra* (acronyme pour **SO**ciété de **PR**ogrammation et d'**A**nalyses) investit dans le développement de logiciels, notamment dans le domaine bancaire et les ressources humaines. Le groupe *Sopra* fera son entrée à la bourse de Paris en 1990 après avoir travaillé sur un projet avec Ministère de l'Intérieur. De son côté, *Steria* (acronyme de Société d'étude et de réalisation en informatique et automatisme) réalise de grandes signatures du côté de la sphère publique, en informatisant l'AFP en 1975 ou encore en participant au développement du Minitel en 1981 par exemple.

Le milieu des années 90 marque le début d'une série d'acquisitions de la part des deux groupes. *Sopra* s'implante au Royaume-Uni, en Espagne, en Italie et en Allemagne en 1999 tandis que *Steria* rentre à la bourse de Paris cette même année. Par la suite, le groupe double de taille en intégrant les activités européennes de *Bull* en 2001 et se renforce dans le conseil en acquérant l'entreprise allemande *Mummert Consulting* en 2005. *Steria* développera considérablement

ses parts de marché dans le secteur public au Royaume-Uni en acquérant la société *Xansa*, expert dans le *Business Output Processing*. Le groupe *Sopra* quant à lui, consolide son expansion européen en créant la filiale *Axway Software* en 2001 qui deviendra indépendante en 2011. En parallèle, *Sopra* acquiert 100% du groupe *Delta Informatique*, société indépendante éditrice d'une offre de solutions « Global Banking » destinée aux banques de détail en France et à l'international. Suite à ce rachat et à celui d'autres sociétés et filiales, en 2012, le groupe *Sopra*, reconnu pour son expertise dans les services financiers, crée la filiale *Sopra Banking Software*. Les solutions dédiées aux ressources humaines feront parties d'une autre filiale appelée, *Sopra HR Software*.

2 Fusion des deux groupes et création de *Sopra Steria*

L'année 2014 marquera le rapprochement amical des deux entités. L'OPE du groupe *Sopra* visant la totalité des actions de *Steria* sera un succès. *Sopra Group* devient ainsi *Sopra Steria Group*. Le changement sera effectif le 31 décembre 2014. Suite au plan d'intégration construit conjointement par les équipes des deux entités précédentes, le groupe continue de mener sa politique basée l'acquisition de nouvelles entreprises comme *Cassiopae*, éditeur spécialisé dans les solutions de crédits à l'entreprise et la gestion immobilière locative, *Kentor*, société scandinave spécialisée dans le conseil, l'intégration de systèmes et la maintenance applicative, ou encore *Galitt*, éditeur de solutions sur le marché des systèmes de paiement et des transactions sécurisées.

Le développement continuel du groupe lui permet d'atteindre une place importante parmi les entreprises de services numériques mondiales. Aujourd'hui, le groupe compte plus de 42 000 collaborateurs dans plus de 20 pays à travers le monde et un chiffre d'affaires de 3,8 milliards d'euros en 2017.

3 La filiale *Sopra Banking Software*

La filiale *Sopra Banking Software* est un fournisseur de solutions globales comprenant, outre sa gamme de progiciels, les services d'intégration, de support et de conseil associés. Elle a été initiée suite au rachat de plusieurs entreprises du secteur bancaire dont notamment *Delta Informatique* en 2011. Par ailleurs, le site *Sopra Banking Software* de Tours où j'ai réalisé mon stage était auparavant un site *Delta Informatique*.

Ses solutions accompagnent près de 800 banques dans 70 pays. Son objectif est d'accompagner les établissements dans leur développement et dans leur stratégie internationale, par une approche de partenariat à long terme. La société s'appuie pour cela sur l'engagement et l'expertise de plus de 3 500 personnes. Les principales zones d'activités de *Sopra Banking* sont en Europe, en Afrique et au Moyen-Orient. La filiale compte un panel d'offres variées pour les clients. Parmi ces offres, on trouve notamment le progiciel *Amplitude* sur lequel mon stage va se baser.

2 Le progiciel Amplitude

1 Présentation

Amplitude, de son nom complet *Sopra Banking Amplitude*, est la solution de *core banking* proposée par *Sopra Banking Software* pour traiter de manière intégrée toutes les problématiques bancaires. Le progiciel a été développé au sein du groupe *Delta Informatique* avant son acquisition par *Sopra*. *Amplitude* est adoptée par 200 banques dans 50 pays, principalement en Afrique et en Europe, et s'adresse à tous types d'institutions financières, de la banque en création aux grands groupes.

Parmi les avantages listées sur le site de *Sopra Banking*, on retrouve :

— Une large couverture avec plus de 80 modules métiers
— Un système entièrement digital ready et sécurisé
— Une solution évolutive et agile
— Une architecture orientée client et process

Les premières mises en production d'*Amplitude* datent du début des années 90. Le progiciel est basé sur le langage de programmation *GENERO 4GL*. Il s'agit d'un langage propriétaire dérivé du *Informix 4GL* développé par la société *Informix* au milieu des années 80. Ce langage appartient aujourd'hui à *IBM* depuis le rachat d'*Informix* en 2005. De son côté, le *Genero 4GL* est maintenu par *4Js* et intègre de nombreux éléments graphiques *Windows*. La particularité du *4GL* réside dans le fait qu'il a été conçu pour communiquer avec des bases de données. De ce fait, il est similaire au langage SQL. Par ailleurs, on appelle **module** un fichier source de 4GL (avec une extension *.4gl*).

Amplitude a beaucoup évolué depuis ses premières versions, avec l'ajout de nouvelles fonctionnalités et des évolutions marquantes. La dernière version stable, la v.11 nommée *Amplitude Up*, ajoute par exemple de nombreuses fonctionnalités dans le domaine du digital.

2 Architecture générale

L'architecture d'*Amplitude* repose sur de nombreux programmes et modules qui communiquent entre eux. La complexité de la structure réside principalement dans le fait que chaque client ne dispose pas forcément des mêmes fonctionnalités car les demandes et les ajustements diffèrent

pour chaque établissement. Par exemple, un banque A disposant d'une version 9.3.0 d'*Amplitude* aura des ajustements différents d'une banque B avec une version 9.3.0 également. Néanmoins, depuis la version 11 d'*Amplitude*, le progiciel s'appuie sur une base de programmes commune à chaque branche.

D'un point de vue conceptuel, les différentes fonctions présentes dans le progiciel ont été cartographiées et répertoriées sous différentes entités. La Figure 1 représente un aperçu de la cartographie fonctionnelle d'*Amplitude*. Pour comprendre un peu mieux cette cartographie, nous pouvons prendre un exemple. La fonction "Chèques" fait partie du domaine "Opérations". Et la fonction "Chèques" contient elle-même des sous-fonctions, et ainsi de suite. L'ensemble des données cartographiées est visible sur le dictionnaire des données, qui sera présenté plus tard dans le rapport.

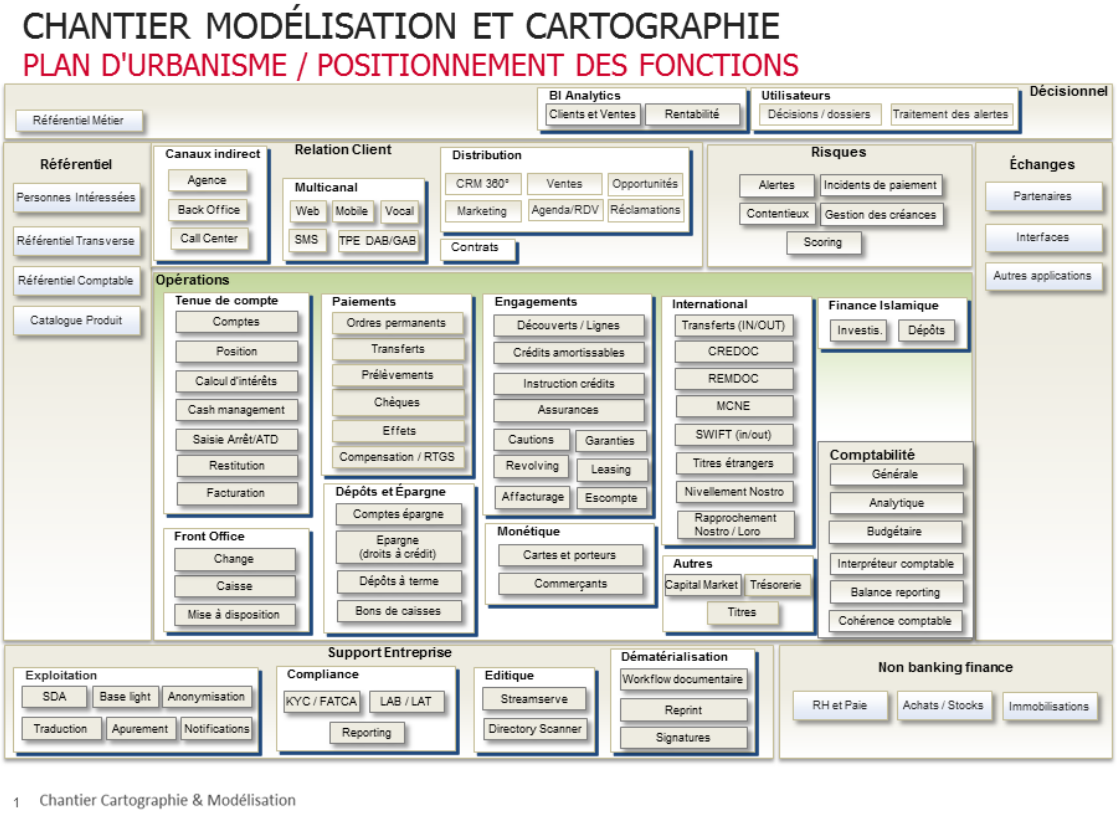


Figure 1 – *Cartographie fonctionnelle d'Amplitude*

D'un point de vue du code tel qu'il est implémenté, les sources d'une version *Amplitude* comporte les répertoires suivants :

BaseProg : contient la liste des fichiers **BaseProg**. Ceux-ci répertorient les modules utilisés pour un programme donnée

base_ref : contient le schéma de référence *Amplitude*, sert principalement à la compilation

bin : contient la liste des sources binaires

migration : contient la liste des scripts permettant de gérer les migrations de serveur

src : contient les répertoires sources 4gl et *per* (les écrans) organisés par module (ATTENTION : ne pas confondre module en tant que qu'entité pour l'organisation et module 4gl, c'est-à-dire les fichiers sources)

xcf : contient la liste des services métiers génériques (SMG) disponibles. Les SMG sont des programmes utilisés par *Amplitude* mais extérieur au progiciel.

xsd : contient les fichiers *xsd* nécessaires au produit.

Nous ne rentrerons pas dans le détail du fonctionnement de chacune des sources présentées ici, mais nous verrons davantage d'éléments sur ce qui nous servira pour le projet dans la suite du rapport (dans la Partie IV notamment).

3 Présentation du projet

1 Contexte

Le projet a été lancé à la demande des architectes fonctionnels du progiciel qui effectuent des revues d'analyses. Leurs objectifs sont les suivants :

- Maîtriser l'évolution du modèle de données *Amplitude* pour assurer la pérennité et l'évolutivité dans le temps d'*Amplitude* en urbanisant les données et en standardisant les développements,
- Orienter l'analyste pour qu'une solution ou un concept réutilisable soit développé(e) et rechercher s'il existe des pistes d'améliorations possibles sur la solution / le concept existant(e) et transmet les axes d'améliorations,
- Valider que les solutions proposées par l'analyste respectent bien les règles et principes d'architecture fonctionnelles et techniques du produit *Amplitude*, et qu'elles sont conformes à la politique éditeur *Amplitude Up*.

Or actuellement dans l'entreprise, aucun outil ne permet de recenser des flux entrants et sortants d'*Amplitude*, d'analyser les impacts suite au développement et de visualiser la cartographie. Les architectes fonctionnels utilisent des fichiers Excels répertoriant ses informations, ce qui n'est pas la meilleure solution d'un point de vue efficacité.

Ainsi, la réalisation d'un outil de fiabilisation des informations de recensement, des analyses et de leur visualisation améliorerait la maîtrise du modèle de données *Amplitude* et faciliterait donc la vie aux architectes fonctionnels, voire même aux développeurs.

2 Expression du besoin

Les besoins principaux demandés sont les suivants :

Dictionnaire des flux, interfaces et éditions : l'idée serait de faire un dictionnaire regroupant l'ensemble de flux entrants et sortants d'*Amplitude*. Cela permettrait de gagner en efficacité lors des revues d'analyses et de solution, d'améliorer l'efficacité de l'ensemble des acteurs sur l'analyse d'impact de l'évolution et d'apporter une meilleur plus-value pour les clients.

Analyse et alerte sur les impacts développements : avoir un aperçu des impacts que pourraient avoir la modification d'une fonction d'un module. Cela faciliterait l'analyse d'im-

pact pour les analystes, les testeurs et les architectes, de donner une vision globale aux nouveaux arrivants et de prévenir les risques en cas d'évolution futures.

Visualisation de la cartographie : avoir un aperçu global de la cartographie, pouvoir zoomer sur les sous-fonctions, les programmes, les tables, etc., ajouter des commentaires...

3 Description des exigences

L'outil de cartographie devrait supporter les versions d'*Amplitude Up* ouvertes aux clients (version 11 et supérieur). L'outil sera interfacé avec le dictionnaire des données qui regroupe les éléments de la cartographie fonctionnelle ainsi qu'aux versions d'*Amplitude* souhaitées par le biais de SVN.

Le but du projet étant de proposer une visualisation de la cartographie, l'outil devra avoir une interface claire, lisible, qui puisse perdurer à l'avenir. Ainsi, le choix d'utiliser *Angular* pour la partie front-end est venu (le fonctionnement d'*Angular* est détaillé dans le Chapitre 6). Des fonctions de recherche devront être implémentées pour retrouver facilement les entités que l'on souhaite observer.

Les programmes d'*Amplitude* devront être répertoriés avec, quand cela est possible, les informations suivantes :

- Le nom du programme
- La description du programme
- Les champs d'application du programme : standard, spécifique adhérent, spécifique non adhérent, etc. La liste doit être paramétrable
- Les caractéristiques du programme : le pays, la zone géographique, le client ou groupe, etc. La liste doit être paramétrable
- Le type de programme : batch, transactionnel, écran (attention, plusieurs choix possibles pour un programme)
- Les commentaires

4 Gestion de projet

Le projet suivra une méthode agile pour plusieurs raisons. Tout d'abord, il s'agit d'un projet en interne visant à améliorer la productivité des personnes travaillant sur *Amplitude*. Il n'y a pas d'obligation de mise en production pour un client par exemple. Ensuite, le projet sera divisé en plusieurs lots, avec un lot principal qui devrait terminer à l'issue de mon stage, et de futurs lots incluant des besoins qui n'ont pas été pris en compte dans le premier lot.

Je serai la seule personne qui va travailler sur l'outil de cartographie à temps plein. Néanmoins, je vais être accompagné pendant toutes les étapes pour mon encadrant et par mes collègues lorsque certains problèmes bloquants se présenteront et pour la réalisation de certaines tâches plus complexes (notamment la gestion de la sécurité de l'application).

Deuxième partie

Tutoriels et formations

Cette partie sera consacrée aux différents tutoriels et formations que j'ai effectué lors des premières semaines de stage (approximativement les 3 semaines du mois d'Avril). Après avoir pris connaissance du sujet et des locaux, une phase de mise à niveau sur les outils que nous allions utiliser s'imposait. Le principe était de réaliser des tutoriels préconisés par mon encadrant (tutoriels internes à *Sopra* ou sur Internet) et de réaliser des comptes-rendus (ou *Proof of Concept*) sur les outils observés pour que les personnes souhaitant s'intéresser au sujet puissent avoir un support disponible. Les formations étaient portées sur trois technologies principales : *Spring Boot*, *Spring Data* et *Angular*. Ces technologies seront détaillées dans cette partie.

Avant de rentrer dans les détails, il est important d'avoir quelques connaissances en développement au préalable, en particulier en langage Java, sur certains frameworks et sur l'outil de construction de projet *Maven*. Les notions basiques de *Maven* sont résumées rapidement sur le site de documentation d'Apache ici : `https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html`.

4 Spring Boot

Spring Boot a pour but de faciliter la création d'application utilisant *Spring* en automatisant ses configurations. *Spring Boot* permet par ailleurs de créer pour un projet, un exécutable unique contenant toutes les dépendances nécessaires. Les objectifs annoncés par la documentation de *Spring Boot* sont les suivants :

— Proposer des solutions rapides et accessibles pour les développements *Spring* ;
— Faciliter les configurations, même lorsque les paramètres souhaités diffèrent de ceux utilisés par défaut ;
— Proposer une panoplie d'options non-fonctionnelles (comme des serveurs embarqués *Tomcat*, des options de sécurité, de mesures de performances, etc.) ;
— Aucune génération de code ni de configuration XML.

Spring Boot ne génère pas de code ni ne modifie les fichiers du projet. Au démarrage de l'application, *Spring Boot* va dynamiquement « brancher » les composants et les configurations nécessaires au contexte du projet. Les deux principales modifications à opérer pour faire fonctionner *Spring Boot* se passent sur le fichier *Maven* et au travers d'annotations du framework *Spring* sur les composants en action. Le détail du fonctionnement sera détaillé dans les prochaines sections.

Spring Boot fonctionne avec les versions Java 1.8 et supérieur. Les différentes versions des composants utilisés par *Spring Boot* sont détaillés dans le guide de référence à l'adresse suivante : https://docs.spring.io/spring-boot/docs/2.0.1.RELEASE/reference/htmlsingle/.

1 Création d'un projet avec *Spring Initializr*

Il est possible de générer un projet *Spring Boot* avec toutes les dépendances que l'on souhaite *Spring Initializr* à l'adresse : https://start.spring.io/. Un aperçu du site est visible sur la Figure 1.

Pour générer un projet, il suffit d'indiquer les propriétés du projet et les dépendances que l'on souhait ajouter et un modèle de projet *Spring Boot* sera disponible au téléchargement. Les détails et le fonctionnement du fichier *pom.xml* généré sont détaillés dans la prochaine section.

La création d'un projet par ce biais n'est pas obligatoire, cependant dans les cas où l'on connaît par avance les principales dépendances de notre projet, cela s'avère être un gain de temps non négligeable. Le projet généré est sous forme de dossier compressé *.zip* contenant la structure du

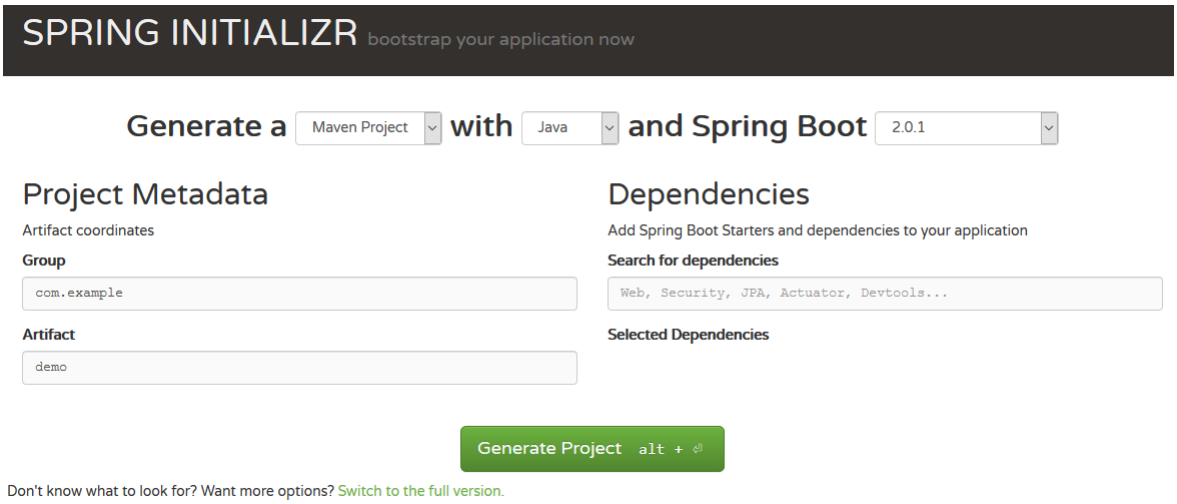


Figure 1 – *Aperçu du site Spring Initializr*

projet, le fichier de construction du projet *pom.xml* de *Maven* avec toutes les dépendances, et les classes principales.

2 Utilisation avec *Maven*

Pour utiliser *Spring Boot* via l'outil *Maven*, la solution la plus courante est de faire hériter son projet du *spring-boot-starter-parent* qui contient les dépendances de bases de *Spring Boot* en ajoutant cette partie au début du fichier *pom.xml* :

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.3.RELEASE</version>
</parent>
```

Figure 2 – *Extrait d'implémentation de la dépendance parent dans le pom.xml*

Ensuite, dans la partie *dependencies*, on ajoute les starters correspondant au type d'application que l'on souhaite réaliser. Les *starters* sont des dépendances *Maven* regroupant toutes les configurations voulues pour l'application. Par exemple, si l'on souhaite réaliser une application web, on ajoute la dépendance *spring-boot-starter-web* dans le fichier *pom.xml* :

```
<!-- Add typical dependencies for a web application -->
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
</dependencies>
```

Figure 3 – *Extrait d'implémentation d'une dépendance web dans le pom.xml*

Cela embarquera toutes les dépendances nécessaires au fonctionnement d'une application web, comme un serveur embarqué *Tomcat* par exemple. Tous les starters suivent la même syntaxe : *spring-boot-starter-**, avec * le type de projet souhaité. Parmi les différents starters disponibles, voici un liste des plus intéressants pour plusieurs types de projets :

spring-boot-starter Starter central, auto-configure le support, le logs et les fichiers YAML.
spring-boot-starter-web Utilisé pour réaliser des application web utilisant Spring MVC. Utilise Tomcat comme conteneur embarqué par défaut.
spring-boot-starter-thymeleaf Utilisé pour les applications Web MVC avec les vues de Thymeleaf.
spring-boot-starter-test Utilisé pour la réalisation de tests. Il implémente plusieurs bibliothèques de tests comme JUnit, Mockito, etc..
spring-boot-starter-actuator Utilisé pour la mise en production, ajoute des fonctionnalités pour monitorer et gérer l'application.
spring-boot-starter-jdbc Utilisé pour JDBC avec le pool de connexion HikariCP.
spring-boot-starter-data-jpa Utilisé pour Spring Data JPA avec Hibernate.
spring-boot-starter-security Utilisé pour Spring Security.

La liste complète des starters se trouvent à l'adresse suivante : https://docs.spring.io/spring-boot/docs/2.0.1.RELEASE/reference/htmlsingle/#using-boot-starter.

Ensuite, pour empaqueter l'application dans un exécutable, il suffit de rajouter le plug-in maven dans la partie *build* du fichier *pom.xml* comme on peut le voir dans la Figure 4.

```
<!-- Package as an executable jar -->
<build>
        <plugins>
                <plugin>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-maven-plugin</artifactId>
                </plugin>
        </plugins>
</build>
```

Figure 4 – *Implémentation du plug-in Maven*

Il s'agit de l'essentiel des modifications à apporter dans le fichier de configuration de *Maven* pour faire fonctionner une application *Spring Boot*. Il est possible d'ajuster d'autres paramètres ou d'ajouter davantage de fonctionnalités. Par exemple, pour les applications web, la dépendance *spring-boot-devtool* permet d'effectuer des rafraîchissements à la volée sans recompiler l'intégralité du code.

3 Éléments à ajouter dans le programme

La configuration des projets *Spring Boot* nécessite l'utilisation d'annotations du framework *Spring* pour définir l'architecture du projet et les interactions entre les composants. L'annotation principale à ajouter est `@SpringBootApplication`.

Cette annotation doit être placée sur la classe principale du projet et correspond à 3 annotations *Spring* :

— `@Configuration` qui indique que la classe est la source pour le contexte donné ;
— `@EnableAutoConfiguration` qui indique à *Spring Boot* d'ajouter les configurations définies au préalable ;
— `@ComponentScan` qui indique à *Spring* de regarder les configurations dans les autres classes du même package ;

Ensuite, la méthode *main()* utilise la fonction dédiée au lancement de l'application de *Spring Boot* : `SpringApplication.run()`. Sur la Figure 5, un exemple de classe principale basique utilisant la fonction `run()` et l'annotation pour le lancement de *Spring Boot*.

```java
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Figure 5 – *Fonction principale d'un projet Spring Boot*

Spring Boot se connecte automatiquement à la classe principale du projet. Or, il est possible qu'un projet comporte plusieurs classes principales. Pour pallier à cela, il est possible de définir la classe que l'on souhaite lancer dans le fichier *pom.xml* de deux manières : soit dans la partie *properties* dans une balise **<start-class>**, soit dans la partie *build* dans une balise **<configuration>** comme dans la Figure 6.

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
             <mainClass>com.sopra.SpringBootWebApplication</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Figure 6 – *Fichier pom.xml en cas de multiples fonctions principales*

Dans le cas où l'on souhaite réaliser des tests unitaires, on peut ajouter le starter *spring-boot-starter-test* dans le fichier *pom.xml*. Puis, on annote les classes de tests par `@SpringBoottest` ce qui permet de notifier à *Spring Boot* que la classe annotée par la balise correspond à une classe de tests et ainsi, d'ajouter les configurations automatiques nécessaires.

4 Exemple d'application web avec *Spring Boot*

Il existe plusieurs façons de réaliser une application Web avec Spring Boot. La dépendance permettant d'importer les principales configurations est spring-boot-starter-web qui utilise Spring MVC pour fonctionner. D'autres starters peuvent être utilisés pour certains cas, par exemple pour une application web MVC, si l'on souhaite gérer les vues avec Thymeleaf ou Mustache, la dépendance spring-boot-starter-thymeleaf ou spring-boot-starter-mustache configurera automatiquement les paramètres nécessaires.

Par défaut, Spring Boot utilise les conteneurs Tomcat. Il est possible d'utiliser un autre conteneur que Tomcat, comme Jetty par exemple, mais il faut au préalable exclure Tomcat des dépendances du fichier pom.xml, comme sur la Figure 7.

Le port d'écoute utilisé par défaut est le 8080. Pour le modifier, plusieurs possibilités existent :

— Indiquer le port souhaité dans un fichier *application.properties* ou *application.yml* ;
— Faire figurer dans une méthode le changement de port ;

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Figure 7 – *Fichier pom.xml en cas de multiples fonctions principales*

— Le faire figurer en ligne de commande au lancement du *jar* exécutable (exemple : `java -jar -Dserver.port=8888 spring-boot-example-1.0.jar`).

Dans le cas où un port est spécifié à la fois dans le fichier *application.properties* et dans le code, alors c'est le code spécifié dans le code qui sera pris en compte.

Ensuite, dans le code, il suffit d'annoter la classe main comme expliquée dans la Section 3 et d'ajouter un contrôleur basique pour faire fonctionner le tout.

5 Gestion de base de données

La gestion et la configuration d'une base de données avec *Spring Boot* se fait de façon simple et automatisé. Il existe plusieurs starters permettant de gérer une base de données avec *Spring Boot*, ici nous allons évoquer le fonctionnement avec JDBC, puis avec JPA et *Spring data*.

Il y a principalement 3 dépendances à ajouter dans le fichier *pom.xml* si l'on souhaite utiliser JDBC : *spring-boot-starter* pour la configuration principale de l'application, *spring-boot-starter-jdbc* pour les dépendances avec JDBC et le driver de la base de données que l'on souhaite utiliser. Comme pour les autres dépendances, *spring-boot-starter-jdbc* utilise Tomcat pour gérer les connexions. Pour modifier cela, il faut d'abord exclure la dépendance Tomcat, puis ajouter celle que l'on souhaite par la suite. Par exemple, si l'on utilise une base de données Oracle et une connexion via DBCP2, les dépendances du fichier *pom.xml* seront tels qu'on peut les trouver sur la Figure 8.

Ensuite, les paramètres de connexions de la base de données sont indiqués dans le fichier de configuration *application.properties* comme sur la Figure 9.

Spring Boot s'occupe de l'initialisation de la base de données automatiquement. Par défaut, il va charger les scripts *schema.sql* et *data.sql* dans le répertoire du projet et les lancer lorsque le projet est exécuté. Le fonctionnement est différent lorsque l'on utilise Spring Data JPA.

Pour l'utilisation avec Spring Data JPA, dans le fichier *pom.xml*, on utilise la dépendance *spring-boot-starter-data-jpa* à la place de *spring-boot-starter-jdbc*. De la même manière, on peut modifier la connexion *Tomcat* par défaut par celle de son choix. Pour l'initialisation de la base de données, en utilisant *Hibernate*, il est possible de définir le fonctionnement automatique lors du lancement de l'application au travers d'une variable *spring.jpa.hibernate.ddl-auto*. Cette variable peut prendre 5 valeurs qui correspondent à 5 mises en place différentes : *none*, *validate*, *update*, *create* et *create-drop*. Elle peut être explicitée dans le fichier *application.properties*. Si elle n'est pas définie explicitement, *Spring Boot* choisira une valeur par défaut selon la base de données intégrée au projet.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

<!-- Exclude the default Tomcat connection pool -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.tomcat</groupId>
            <artifactId>tomcat-jdbc</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<!-- Oracle JDBC driver -->
<dependency>
    <groupId>com.oracle</groupId>
    <artifactId>ojdbc6</artifactId>
    <version>11.2.0</version>
</dependency>

<!-- Common DBCP2 connection pool -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.1.1</version>
</dependency>
```

Figure 8 – *Exemple de dépendances dans le pom.xml pour la gestion d'une base de données*

```
# Set true for first time db initialization.
spring.datasource.initialize=true

# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:DB11P15
spring.datasource.username=bank
spring.datasource.password=oracle
spring.datasource.driver-class-oracle.jdbc.driver.OracleDriver

# dbcp2 settings
# spring.datasource.dbcp2.*

spring.datasource.dbcp2.initial-size=7
spring.datasource.dbcp2.max-total=20
spring.datasource.dbcp2.pool-prepared-statements=true
```

Figure 9 – *Fichier de configuration pour l'initialisation d'une base de données*

En utilisant la variable *spring.jpa.hibernate.ddl-auto*, il n'est pas nécessaire d'ajouter une variable *spring.datasource.initialize* étant donné que l'initialisation est gérée par la variable d'Hibernate.

6 Déploiement d'une application *Spring Boot* x *Angular*

Dans le cadre de la réalisation d'un projet multi-module, il est devenu courant de gérer une application Web avec une partie front-end qui utilise Angular pour profiter de ses facilités de développement et une partie back-end qui utilise *Spring Boot* et *Spring Data*, ce qui sera par

ailleurs le cas pour notre projet. Se pose alors la question du déploiement d'une telle application. Plusieurs possibilités existent pour déployer ce type de projet multi-module, nous allons ici en voir 3 parmi d'autre.

6.1 Déployer les parties séparément

La solution la plus simple, et la meilleure pratique, est de séparer la partie front et la partie back de l'application. Cela permet d'améliorer l'évolutivité et la gestion du projet. En effet, en séparant les parties, les développements peuvent se faire en parallèle, avec une équipe dédiée au front-end et une au back-end, sans impacter l'une ou l'autre équipe. Cela permet également de réduire les temps d'indisponibilité de la plateforme, car si la partie back est indisponible, la partie front sera accessible malgré tout pour le client.

Néanmoins, pour une petite équipe, gérer deux serveurs peut s'avérer difficile et réunir les deux modules en un seul serait un avantage. Pour cela, plusieurs solutions existent, utilisant les possibilités de Maven ou de *Spring Boot*, mais il faut garder à l'esprit qu'un projet *Angular* n'est pas optimisé pour fonctionner avec Maven.

6.2 Déploiement sous forme de *.war* avec *Maven*

Maven offre la possibilité d'assembler plusieurs modules en un seul. Pour cela, il faut disposer les projets dans un dossier parent et y ajoute un nouveau fichier pom.xml. Ce fichier va permettre à Maven de construire les deux projets fils avec une seul commande. Le fichier *pom.xml* du dossier parent ressemblera à l'image suivante. On peut voir que le projet dispose de deux modules correspondant à la partie Spring Boot et à la partie Angular.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.soprabanking</groupId>
    <artifactId>appli-test</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>pom</packaging>

    <name>appli test</name>
    <description>Application test de déploiement multi-module</description>

    <modules>
        <module>angular-test</module>
        <module>app-rest-angular</module>
    </modules>

</project>
```

Figure 10 – *Fichier pom.xml d'un projet multi-module*

Le déploiement se fera sous la forme d'un fichier *.war* qui va regrouper toutes les dépendances, les classes et les ressources de notre application web. Le fichier va se trouver dans le dossier target du module fils Spring Boot. Pour pouvoir le générer avec tous les éléments, plusieurs ajustements sont nécessaires. Tout d'abord, il faut ajouter un fichier *pom.xml* au projet Angular. Comme Angular n'est pas configuré pour fonctionner avec Maven mais avec Node, le fichier pom.xml va exécuter les commandes npm install et npm build pour compiler le projet. Par défaut, les ressources compilées se trouvent dans le fichier dist de la partie Angular.

Dans la partie back-end, la compilation Maven devra générer le fichier .war en récupérant les données compilées dans la partie Angular. Pour cela, il faut utiliser les plug-in Maven War,

qui va générer le fichier, et Maven Resources, qui va copier les ressources compilées du projet Angular dans un dossier de la partie Spring Boot. Un exemple de fichiers pom.xml pour la partie Client et pour la partie Serveur est disponible sur le site http://www.devglan.com/spring-boot/spring-boot-angular-deployment.

6.3 Déploiement sous forme de *.jar* en utilisant *Spring Boot*

Il existe également la possibilité d'utiliser les fonctionnalités de Spring Boot pour déployer son application sous forme d'un unique fichier jar contenant toutes les dépendances. Pour cela, le projet doit avoir une structure similaire à celle d'un déploiement avec Maven dans la partie précédente, c'est-à-dire un projet parent avec un fichier pom.xml qui lui est propre, et les deux projets fils avec chacun un fichier pom.xml pour son propre fonctionnement.

La configuration du fichier *pom.xml* du projet Angular reste le même que pour un déploiement avec un .war. La différence se fera sur la configuration de la sortie des fichiers compilés. Dans le fichier package.json, dans la partie build, il faut ajouter le chemin de sortie avec la commande suivante : `ng build -prod -output-path dist/META-INF/resources`. *Spring Boot* est pré-configuré pour aller chercher les éléments statiques dans ce répertoire. Ainsi, à la création du jar, toutes les dépendances seront ajoutées.

5 Spring Data

Spring Data est un projet auxiliaire de Spring permettant de gérer plus simplement l'accès aux données, le but principal étant d'apporter une couche d'abstraction supplémentaire pour faciliter la manipulation des données qui peuvent provenir de différentes sources. Il fonctionne avec de nombreux types de bases de données, relationnel ou non, avec des frameworks de map-reduce, ainsi qu'avec des services de données basés sur le cloud. Il contient plusieurs sous-projets qui s'adaptent aux types de bases de données utilisées.

Parmi les fonctionnalités explicitées par la documentation, on retrouve :

- Un dépôt central puissant ;
- Création de requêtes dynamiques ;
- Intégration Spring simple via JavaConfig ou XML ;
- Intégration avancée avec les contrôleurs de Spring MVC ;
- Support pour faire de l'audit (indications sur les objets, leur date de création, dernière modification, etc.) ;
- Et d'autres...

Spring Data met à disposition de nombreux modules qui s'adaptent à la structure de gestion de données utilisée : *Spring Data JPA*, *Spring Data MongoDB*, *Spring Data Neo4j*, etc.

1 Fonctionnement de base de *Spring Data* et de *Spring Data JPA*

Le fonctionnement de Spring Data peut être représenté sous forme de couches, avec une couche principale appelée Spring Data Commons et de multiples modules reprenant les éléments de la couche principale et qui s'adaptent à une utilisation particulière. Spring Data Commons est basé sur trois interfaces, que l'utilisateur pour étendre selon ses besoins : Repository, CrudRepository et PagingAndSortingRepository. L'interface Repository permet d'indiquer un dépôt avec un type d'objets avec lequel on travaille. L'interface CrudRepository étend l'interface Repository et ajoute des méthodes CRUD (create, read, update, delete) pour manipuler les données. Enfin, l'interface PagingAndSortingRepository étend CrudRepository et ajoute des méthodes de pagination et de tri de données.

Ensuite, Spring Data permet d'écrire des requêtes à partir de noms de méthode. Par exemple, sur la Figure 1, à l'appel de la méthode *findByEmailAddressAndLastName()*, *Spring Data* va créer la requête qui recherche l'email et le nom de la personne automatiquement. Il n'est donc pas

nécessaire de rédiger la méthode soi-même, ce qui fait une des grandes forces de l'outil. Spring Data est capable de prendre en compte des mots-clés comme And, Or, LessThan, GreaterThan, etc.. La liste complète des mots-clés est à l'adresse https://docs.spring.io/spring-data/jpa/docs/2.0.6.RELEASE/reference/html/#jpa.query-methods.query-creation.

```
public interface UserRepository extends Repository<User, Long> {

      List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
    }
```

Figure 1 – *Exemple de repository*

Au cas où le nom de la méthode est trop long ou incommodant dans le programme, ou bien si l'on veut utiliser le nom d'une méthode en modifiant la requête qu'elle doit effectuer, il est possible d'utiliser l'annotation `@Query` pour changer le nom de la méthode à sa guise. Avant la déclaration de la méthode, cette annotation indiquera la requête que l'on souhaite réaliser lors de l'appel de la fonction.

```
public interface UserRepository extends Repository<User, Long> {

      List<User> findByEmailAddressAndLastname(String emailAddress, String lastname);
    }
```

Figure 2 – *Exemple de requête spécifié dans un repository*

Dans l'exemple de la Figure 2, la requête associée à la méthode *findByEmailAddress()* est modifiée, et est déterminée dans l'annotation `@Query`. Si la requête effectue des modifications sur les entités, il faut ajouter l'annotation `@Modifying` et éventuellement si les noms diffèrent, ajouter des annotations `@Param` pour préciser chaque entité modifiée.

```
@Query("update User u set u.name = :name where u.id = :id")
@Modifying
public int updateName(@Param("name")String name, @Param("id") Long id);
```

Figure 3 – *Exemple de requête modifiée*

Avec *Spring Data JPA*, un autre repository est mise à disposition : JPARepository qui regroupe toutes les possibilités des autres repositories et ajoute des méthodes supplémentaires propres à JPA, comme *flush()* pour le vidage du cache, etc.

2 Fonctionnement de *Spring Data REST*

Spring Data Rest est un des modules de *Spring Data* permettant de faciliter la création de web-services REST. Il constitue une couche supérieure aux dépôts *Spring Data* et peut exposer les données des dépôts à une application Web. Ainsi, *Spring Data* nécessite une configuration avec un module de repository comme *Spring Data JPA*, *Spring Data MongoDB*, etc.. En général, *Spring Data REST* n'ajoute pas de fonctionnalités au dépôt de données utilisé. En effet, par définition *Spring Data REST* doit fonctionner avec n'importe quel projet *Spring Data* qui utilise un modèle de repository.

La fonctionnalité principale de *Spring Data REST* est donc d'exporter les ressources des repositories. Ainsi, en configuration standard, un dépôt *PersonRepository* comme ci-dessous va exposer les ressources à l'URI /persons .

```
public interface PersonRepository extends CrudRepository<Person, Long> { }
```

Chaque objet *Person* sera également exposé selon un modèle d'URI `/persons/{id}`. Par défaut, les requêtes HTTP effectuées sur le web-service vont interagir avec les méthodes présentes dans CrudRepository de la manière suivante :

— Méthode *create* -> requête **POST**
— Méthode *read* -> requête **GET**
— Méthode *update* -> requête **PUT**
— Méthode *delete* -> requête **DELETE**

Si l'on souhaite modifier les liens sur lesquelles vont être exportés les ressources, il est possible d'ajouter une annotation `@RepositoryRestResource` avant la déclaration de l'interface.

```
@RepositoryRestResource(collectionResourceRel = "people", path = "people")
public interface PersonRepository extends PagingAndSortingRepository<Person, Long> {

    List<Person> findByLastName(@Param("name") String name);

}
```

Figure 4 – *Exposition REST d'un repository*

Dans l'exemple Figure 4, les objets *Person* seront exportés sur l'URI `/people` et plus sur l'URI `/persons` comme c'était par défaut. L'option *collectionResourceRel* détermine la valeur à utiliser lors de la génération des liens vers les ressources et l'option *path* détermine le chemin vers lequel la ressource sera exportée.

Spring Data Rest ajoute aussi des liens additionnels permettant de naviguer entre les enregistrements. Ces liens sont définis par les méthodes présentes dans le repository de la ressource que l'on recherche. Dans l'exemple précédent, une méthode *findByLastName()* était présente. Ainsi, il est possible de rechercher les personnes par leur nom de famille. Toutes les méthodes de recherche d'un repository sont accessibles avec l'URL `/people/search`. La syntaxe pour effectuer une recherche sur les noms de famille comme indiquer dans l'exemple serait la suivante : `http://localhost:8080/people/search/findByLastName?name=Johnson`. Le résultat de cette recherche donnera toutes les personnes ayant pour nom de famille : Johnson. Les résultats des recherches sont disposés au format JSON.

3 Configuration avec *Spring Boot* et *Spring Data JPA*

L'interfaçage entre *Spring Boot*, *Spring Data JPA* et *Spring Data REST* se fait de façon très simple et automatisé. *Spring Boot* utilise *Spring Data JPA* pour créer une implémentation des dépôts et les configurer via une base de données utilisant JPA. *Spring Data REST* va créer les contrôleurs *Spring MVC*, les convertisseurs JSON et les autres éléments nécessaires pour obtenir un front-end respectant les règles REST. *Spring Boot* va ensuite relier les composants du front-end avec les données de JPA de façon automatique.

Pour lancer un projet utilisant *Spring Boot*, *Spring Data JPA* et *Spring Data REST*, il faut ajouter les deux dépendances *spring-boot-starter-data-jpa* et *spring-boot-starter-data-rest* au fichier *pom.xml* comme évoqué dans la Section 2 (Chapitre 4).

6 Angular

Angular est un framework open-source basé sur le langage *TypeScript* permettant de réaliser des applications web. Il a été initié par Google en 2016 sous la forme d'un autre framework appelé *AngularJS*, basé sur *JavaScript*. Il a été totalement refait par les mêmes équipes pour créer Angular tel qu'on le connaît aujourd'hui. Il est néanmoins nécessaire d'être vigilant lors de la recherche d'informations sur la version d'Angular utilisée, car *AngularJS*, la version 1 d'Angular, est toujours maintenu et utilisé par certains sites. La version stable d'Angular la plus récente est la version 5, cependant une nouvelle version d'Angular sort environ tous les 6 mois. La version 6 est sortie courant Avril 2018 mais n'est pas encore stable.

L'objectif d'Angular est de faciliter la réalisation d'applications web en automatisant certaines parties et en permettant d'effectuer des animations complexes avec très peu de lignes de code. Les applications Angular sont destinées à fonctionner sur les navigateurs Internet récents (Chrome, Firefox, ...) et sont adaptées pour le cross-platform (fonctionner à la fois sur Desktop et sur Mobile). Le framework propose des facilités de développement comme par exemple, la compilation et la génération automatique du code permettant de voir en direct les modifications apportées. La gestion des tests unitaires est également facilitée avec le runner Karma qui permet de voir les résultats de tests directement sur le navigateur et pouvoir modifier à la volée les tests.

1 Installations nécessaires

1.1 *Node.js, npm* et *Angular CLI*

Avant de pouvoir créer un projet, il est nécessaire d'installer plusieurs éléments. Tout d'abord *Node.js*, une plate-forme logicielle en JavaScript permettant d'utiliser de nombreuses bibliothèques en ligne de commande. Les fichiers d'installation de *Node.js* se trouvent à l'adresse `https://nodejs.org/en/download/`. L'installation de *Node.js* inclut le gestionnaire de package *npm* qui permet d'installer des applications *Node.js* à partir du dépôt *npm*. L'application se lance depuis une invite de commande, il est possible de vérifier si l'installation s'est faite correctement avec la commande `npm -v` sur *Windows* qui indique la version de *npm*.

Ensuite, la méthode la plus simple pour manipuler un projet *Angular* est d'utiliser *Angular CLI*. Il s'agit d'un outil en ligne de commande permettant de créer un projet, d'ajouter des fichiers

et d'exécuter des tâches comme l'exécution, le déploiement ou le test d'une application web. L'installation d'*Angular CLI* se fait avec *npm* avec la commande : `npm install -g @angular/cli`.

1.2 Quelques IDE intéressants pour le développement *Angular*

Il existe de nombreux IDE adaptés pour le développement sous *Angular*. Parmi ces IDE, 3 d'entre eux semblent se démarquer des autres : *Angular IDE*, *WebStorm* et l'éditeur en ligne *Stackblitz*.

1.2.1 Angular IDE

Angular IDE est un éditeur gratuit produit par *Genuitec* optimisé pour le développement *Angular*. Il a la particularité d'être disponible à la fois en version standalone mais aussi sous forme de plug-in pour *Eclipse* disponible sur le marketplace. Les téléchargements sont disponibles ici : https://www.genuitec.com/products/angular-ide/.

1.2.2 WebStorm

WebStorm est l'éditeur faisant parti de la suite produite par Jetbrains avec IntelliJ IDEA, PyCharm et d'autres. C'est un des éditeurs les plus populaires, néanmoins il nécessite l'achat d'une licence commerciale pour l'utilisation. Il apport des fonctionnalités intéressantes comme l'auto-sauvegarde des fichiers, permettant avec Angular de voir en direct les modifications apportées au programme sans avoir à sauvegarder manuellement ses fichiers.

1.2.3 Éditeur en ligne : *Stackblitz*

Stackblitz est un éditeur en ligne adapté à divers types de projets Web. Il comporte les fonctionnalités basiques des IDE locaux intégré directement sur navigateur. Le projet est généré à la volée et on peut voir les modifications en direct sur la même fenêtre. Trois panneaux composent l'interface : un avec l'arborescence, un avec l'éditeur et un autre avec l'émulation du projet.

Un des avantages de Stackblitz par rapport aux éditeurs locaux est de permettre à plusieurs personnes d'éditer un projet en même temps, à la manière de ce que propose Google Drive. Il est possible de partager le projet entier ou bien uniquement la fenêtre d'affichage. Les projets créés sur Stackblitz peuvent être téléchargés sous forme de dossier compressé.

2 Structure d'un projet *Angular*

Un projet *Angular* est basé sur deux concepts : les modules et les composants. La construction d'une application *Angular* se fait à partir des modules, qui donne un contexte pour la compilation des composants. Un module collecte les différentes fonctions qui lui sont liées pour l'assemblage du projet. Une application Angular est toujours composée d'au moins un module, appelé root module qui assemblent les éléments nécessaires au lancement du programme.

Les composants définissent les vues de l'application, et une application est constituée d'au moins un composant appelé root component. Chaque composant est définie par au minimum 2 fichier : un fichier HTML qui va définir le modèle d'affichage du composant et un fichier TypeScript avec les données et les fonctions associées à la classe. À ces deux fichiers, on peut en

rajouter deux autres qui ne sont pas obligatoires pour le fonctionnement d'un projet : un fichier CSS pour associer un style au template HTML et un autre fichier TypeScript, distingué par une extension *.spec.ts*, qui définit des tests unitaires.

Les modules et composants de base sont générés lors de la création du projet et donne une arborescence comme on peut le voir sur la Figure 1 dans le dossier */src*.

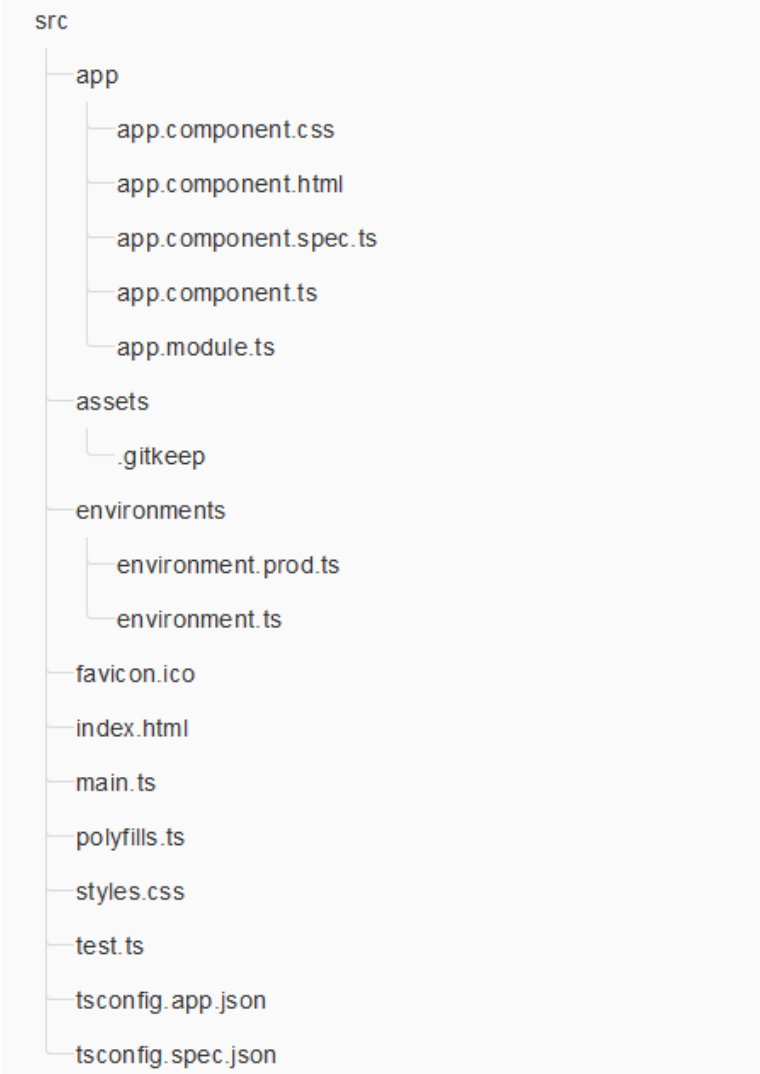


Figure 1 – *Arborescence d'un projet Angular*

Ici, le module et le composant par défaut sont disposés dans le dossier app. Les autres fichiers sont détaillés dans la documentation à l'adresse : https://angular.io/guide/quickstart#the-src-folder.

Le dossier *src* fait partie des nombreux autres éléments générés à la création d'un projet *Angular* avec *Angular CLI*. On retrouve par exemple un dossier *node_module* qui comportent tous les modules externes listés dans le fichier *package.json* présent à la racine du projet. Les autres fichiers sont présentés dans la documentation à l'adresse https://angular.io/guide/quickstart#the-root-folder.

3 Gestion des styles : *Bootstrap* et *PrimeNG*

Il est possible d'appliquer des styles sur les différents composants que l'on souhaite avec les fichiers CSS. Cependant, réaliser un thème cohérent CSS de A à Z est très chronophage. Il

existe plusieurs frameworks permettant d'appliquer des thèmes et de réaliser des animations responsives de manière très simple. Par défaut, un projet *Angular* dispose d'une feuille de style de base appelée *styles.css* qui est vide. Il permet d'appliquer des styles sur l'ensemble de l'application. Il est possible de changer la feuille de style par défaut appliquée au projet ou bien d'en ajouter plusieurs dans le fichier *.angular-cli.json* au niveau de l'option *styles*.

```
"styles": [
  "styles.css",
    "../node_modules/font-awesome/css/font-awesome.min.css",
    "../node_modules/primeng/resources/themes/omega/theme.css",
    "../node_modules/primeng/resources/primeng.min.css",
    "../node_modules/bootstrap/dist/css/bootstrap.css"
  ],
```

Figure 2 – *Sélection des styles en Angular*

Parmi les nombreux frameworks existants, en voici deux parmi les plus populaires pour le développement Web actuellement : *Bootstrap* et *PrimeNG*.

3.1 *Bootstrap*

Bootstrap est une bibliothèque gratuite et open-source permettant de concevoir des applications et des sites web. Utilisée sur un projet, elle permet d'ajouter des éléments responsives, comme des boutons, des panneaux dynamiques et bien d'autres facilement. La bibliothèque est compatible avec la majorité des navigateurs internet sur PC de bureau ainsi que sur mobile. Voici un visuel de ce que peut donner une application avec *Bootstrap*.

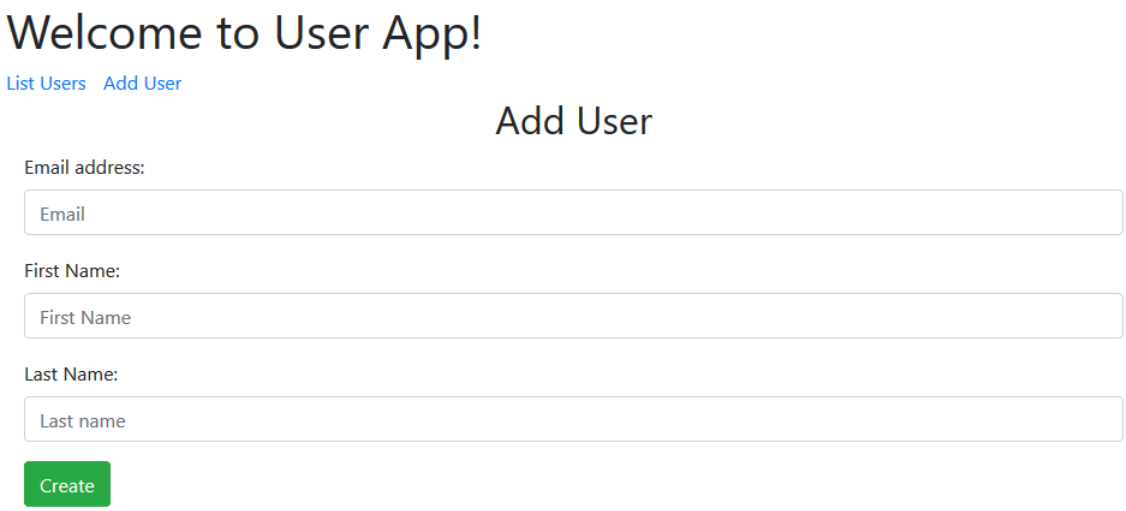


Figure 3 – *Exemple de page formatée avec Bootstrap*

Pour utiliser le template Bootstrap avec Angular, il suffit de l'installer en utilisant la commande npm install bootstrap, puis d'ajouter la feuille de style dans le fichier .angular-cli.json comme sur l'image vu auparavant. Ensuite, l'appel des classes se fait via les fichiers HTML relatifs au composant Angular que l'on souhaite voir impacter. La documentation pour pouvoir utiliser pleinement les capacités de Bootstrap se trouve à l'adresse : https://getbootstrap.com/docs/4.1/getting-started/introduction/.

3.2 PrimeNG

PrimeNG est une bibliothèque adaptée pour Angular permettant d'ajouter toute sorte de composants et de widgets pour les applications Web. PrimeNG est gratuit, open-source et peut être utilisé de façon complémentaire à Bootstrap. Parmi les composants disponibles, il y a divers types de boutons, de graphiques, de panneaux, compatibles avec différents thèmes. Le site de PrimeNG (`https://www.primefaces.org/primeng/#/`) permet de voir directement toutes les possibilités offertes par le framework.

Pour l'utiliser sur un projet, il faut tout d'abord l'installer avec la commande `npm install primeng --save`. Une fois installée, de la même manière que pour Bootstrap, il faut ajouter les styles que l'on souhaite dans le fichier *.angular-cli.json*. Par exemple, pour appliquer le thème Omega de PrimeNG, il faut ajouter les styles comme sur la Figure 4.

```
"styles": [
    "../node_modules/font-awesome/css/font-awesome.min.css",
    "../node_modules/primeng/resources/themes/omega/theme.css",
    "../node_modules/primeng/resources/primeng.min.css"
],
```

Figure 4 – *Exemple de styles utilisant PrimeNG*

Ensuite, pour ajouter un module en particulier, par exemple si l'on souhaite ajouter un tableau, il faut importer le module dans le root module de l'application et ensuite, suivre les instructions du site PrimeNG pour voir la balise HTML à utiliser pour ce module. Pour ajouter notre tableau par exemple, le fichier HTML ressemblera à Figure 5, avec des balises *p-table* pour représenter le tableau en question.

```
<p-table [value]="users">
    <ng-template pTemplate="header">
        <tr>
            <th class="hidden">Id</th>
            <th>FirstName</th>
            <th>LastName</th>
            <th>Email</th>
            <th>Action</th>
        </tr>
    </ng-template>
    <ng-template pTemplate="body" let-user>
        <tr *ngFor="let user of users">
            <td class="hidden">{{user.id}}</td>
            <td>{{user.firstName}}</td>
            <td>{{user.lastName}}</td>
            <td>{{user.email}}</td>
            <td><button class="btn btn-danger" (click)="deleteUser(user)">Delete User</button></td>
        </tr>
    </ng-template>
</p-table>
</div>
```

Figure 5 – *Implémentation HTML utilisant PrimeNG*

Troisième partie

Modélisation de l'outil et chiffrage du projet

Cette partie retrace mes activités dans l'entreprise de fin Avril jusqu'au début du mois de juin. L'essentiel du travail effectué était consacré à la modélisation de l'outil, aux spécifications et au chiffrage des phases de développement du projet.

7 Modélisation UML

L'aboutissement de la modélisation finale a demandé beaucoup d'échanges au préalable entre les architectes fonctionnels et les personnes travaillant sur l'outil, c'est-à-dire mon encadrant et moi. Des phases de recherche pour comprendre le fonctionnement d'*Amplitude* afin de trouver les meilleurs méthodes pour récupérer la cartographie étaient nécessaires.

À l'issue des réflexions, nous avons pu aboutir à plusieurs diagrammes : l'un représentant les entités que nous allons manipuler pour créer notre cartographie, et les autres représentant des diagrammes d'activités qui correspondent à l'alimentation de notre base de données.

1 Diagramme d'entité

On peut séparer le diagramme en deux parties distinctes : d'un côté, ce qu'on peut appeler la cartographie technique représentant les programmes et leurs implémentations ; et de l'autre, la cartographie fonctionnelle qui représente les entités tels qu'elles figurent dans le dictionnaire des données.

Coté cartographie technique, on retrouve principalement les éléments suivants : Programme, Module_4GL, Fonction technique (qui sont les fonctions au sens développeur du terme), Fichier d'échange (qui représentent les différents flux de communication). La Figure 1 montre un extrait de cette partie du diagramme.

On peut remarquer que les programmes sont les éléments qui lient la plupart des entités entre eux. En effet, un programme est composé de plusieurs modules 4GL, qui contiennent eux-même diverses fonctions. De plus, un programme peut générer ou recevoir des fichiers d'échange. En effet, un programme a un ou plusieurs types (si c'est un programme batch avec écran ou non) et peut être un Service Métiers Générique (SMG).

Côté Cartographie fonctionnelle en revanche, on conserve la structure des dictionnaires des données, c'est-à-dire avec plusieurs niveaux de hiérarchie (domaine, fonction, sous-fonction, objet métier) liés aux tables de la base de données, puis qui seront reliés aux programmes de la cartographie technique.

L'intégralité du diagramme, pour mieux comprendre les liaisons et la structure des entités de notre cartographie, se trouve en Annexe A.

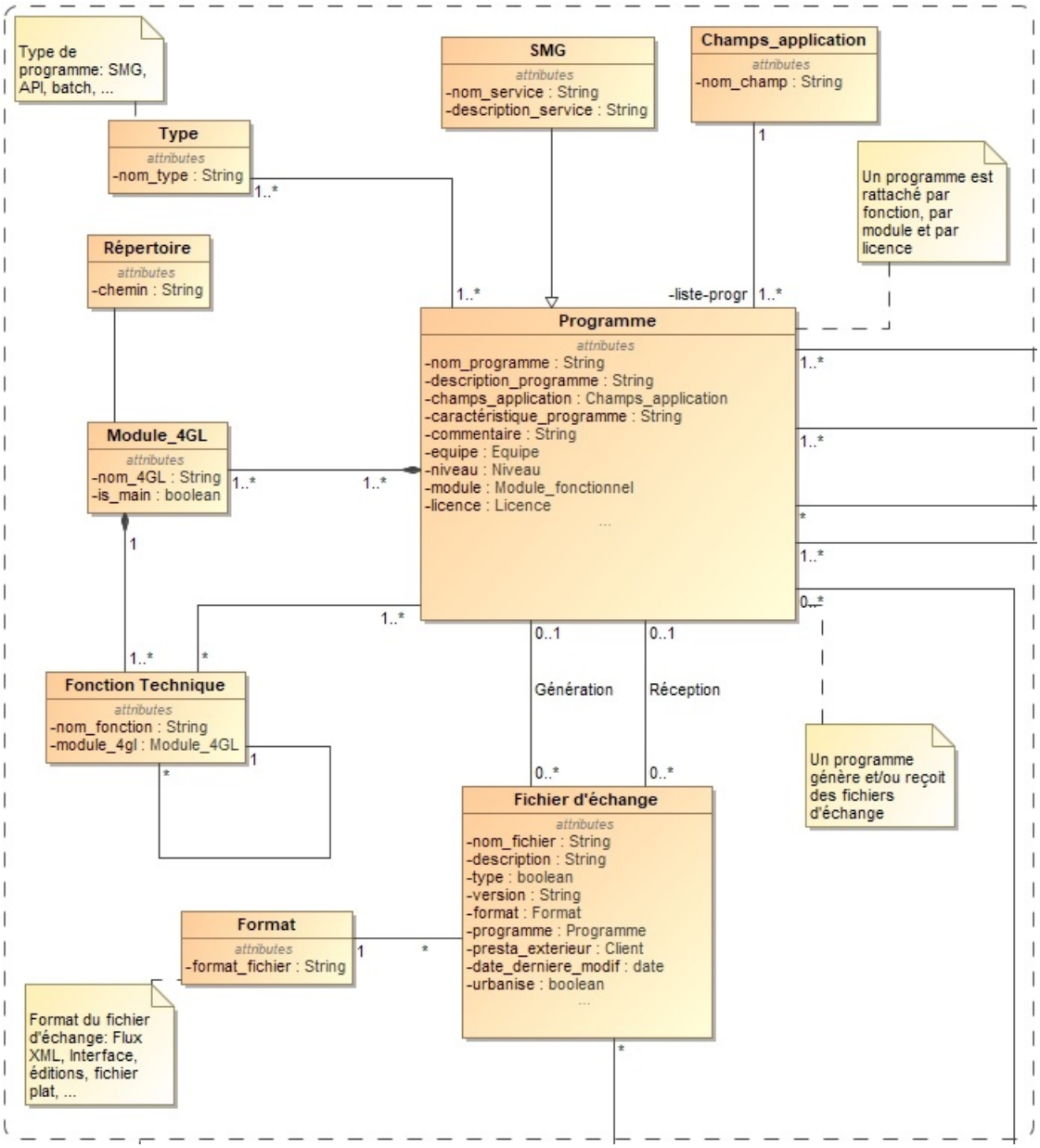


Figure 1 – *Extrait de la partie Cartographie Technique du diagramme d'entité*

2 Diagramme d'activité

Les diagrammes d'activité représentent les agissements au cours du temps d'un processus. Ici, ils vont permettre d'avoir une vision sur les méthodes à implémenter pour le développement. La Figure 2 représente l'alimentation de notre base de données étape par étape. Il s'agit ici du schéma tel que penser au mois de mai. Plusieurs modifications ont été opérées mais l'essentiel de la structure est présente.

Le point de départ du processus d'alimentation est la récupération d'une version d'*Amplitude* qui se fait par le biais de SVN, un outil de versioning. Effectuer un *checkout* comme on le voit sur le diagramme signifie récupérer les sources versionnées pour pouvoir les exploiter plus facilement.

Ensuite, le processus est divisé en deux parties avec d'un côté, la récupération de la cartographie technique et de l'autre, l'extraction du dictionnaire de données V3 pour la cartographie fonc-

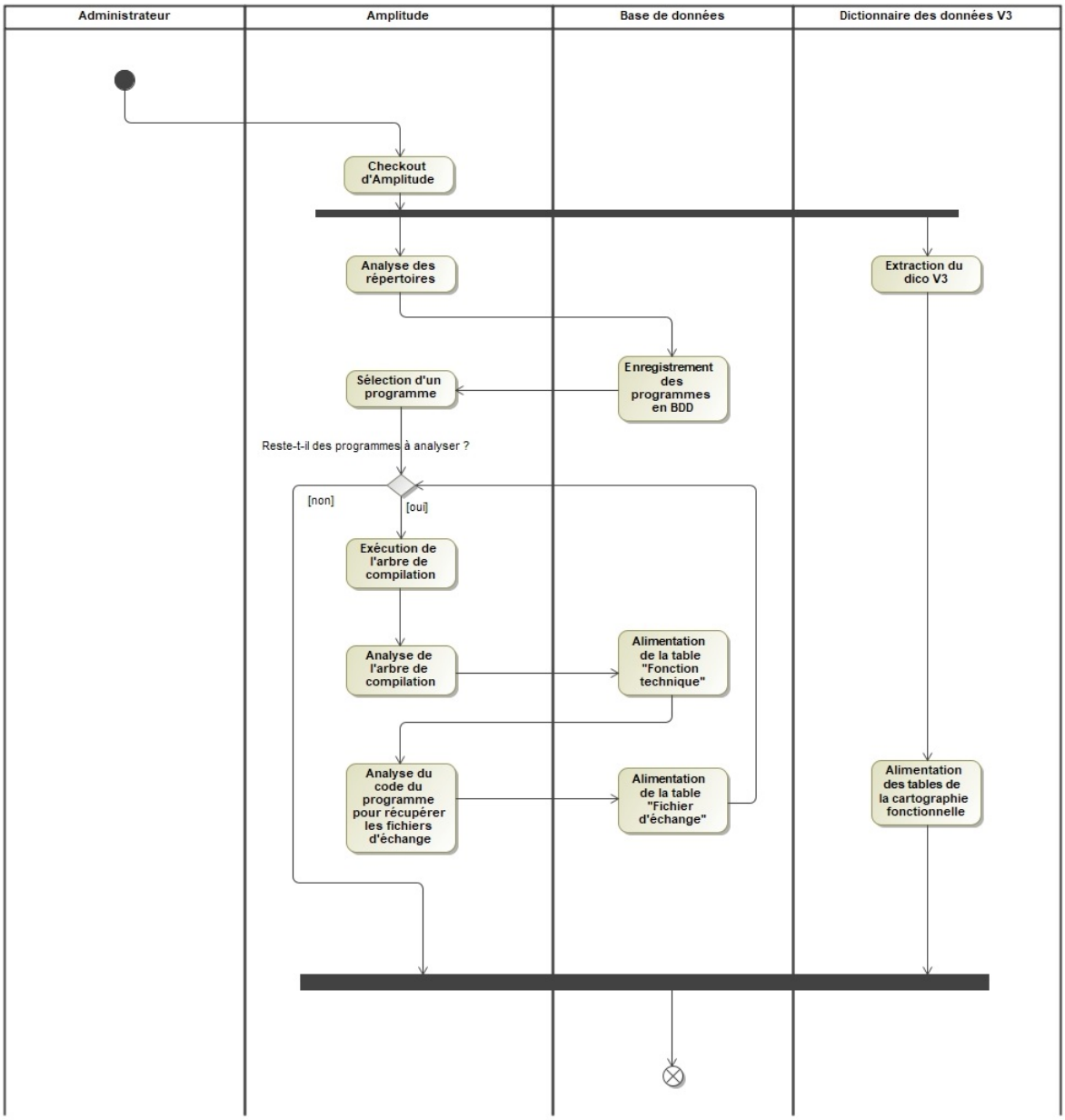


Figure 2 – *Diagramme d'activité de l'alimentation des éléments de la cartographie*

tionnelle. L'idée d'un premier temps était d'enregistrer tous les programmes d'*Amplitude*, puis d'analyser chacun d'entre eux pour récupérer les modules, les fonctions et les autres informations dont on a besoin, comme on peut le voir sur la Figure 2. Ce principe n'a pas été conservé dans l'implémentation finale car il ne permettait pas de voir des fonctions ou des modules qui sont présents sur *Amplitude* mais qui ne sont pas utilisés par des programmes. Ainsi, la solution que nous avons adoptée est une récupération complète des programmes, de modules et des fonctions séparément, puis une association des composants entre eux. Ce qui permet ainsi de voir l'intégralité des éléments composant *Amplitude* et de distinguer les fonctions non utilisées.

La récupération du dictionnaire des données pour la cartographie fonctionnelle se fera en réalité de manière plus simple que sur le diagramme d'activité. En effet, nous avons eu l'autorisation d'accès aux éléments de la base de données du dictionnaire de données. Ainsi, nous aurons la possibilité de les exposer en REST facilement pour les afficher par la suite.

8 Spécifications de l'outil de cartographie

Ce chapitre sera consacré aux étapes de développement à mettre en pour réaliser l'outil de cartographie. On distinguera deux parties distinctes sur le projet : une partie *back-end* avec les implémentations *Java* et *Spring* qui va comporter les fonctions d'analyse et de gestion des données d'Amplitude, et une autre côté *front-end* avec les composants *Angular* qui va gérer les affichages et les interfaces utilisateurs.

Après un premier chiffrage, nous avons pu remarquer que le temps nécessaire à la finalisation de l'outil sera supérieur à mon temps de stage restant. Ainsi, nous avons décidé de nous focaliser davantage sur la partie *back-end* du projet. Les spécifications sont donc bien plus détaillées pour cette partie que pour le *front-end*.

La partie *back-end* de l'application regroupe les éléments de gestion des données présentes dans *Amplitude*. L'outil devra tenir compte des programmes d'Amplitude, des modules 4GL présents dans l'arborescence, des fonctions composant ces modules, et plus généralement des éléments présents dans le diagramme de classe. Le modèle de données pour l'outil sera réalisé avec *Spring Data*, qui permet un interfaçage automatique entre le code source en Java et le gestionnaire de base de données configuré (dans notre cas, Oracle). Une fois le modèle réalisé, nous avons songé à faire séparer le processus d'alimentation en deux : l'alimentation initiale des tables de notre modèle dans un premier temps et la mise à jour régulière des éléments de la base dans un second. Au final, la séparation des processus d'alimentation n'est pas nécessaire, et un unique job prenant en compte les deux cas de figure suffit.

La partie front-end de l'outil sera développée sous Angular et suivra une architecture REST. Les écrans devront être à même de récupérer les données exposées par la partie back-end. L'objectif des écrans est de donner un aperçu général de la cartographie. Un panneau de recherche des données devra permettre de retrouver les informations que l'on souhaite le plus simplement possible.

1 Création du modèle

L'architecture du back-end sera divisée en deux parties principales : une partie cartographie fonctionnelle et une partie cartographie technique. La cartographie fonctionnelle correspond aux entités définies à partir du dictionnaire des données. Pour le moment, l'outil ira chercher directement les informations dans la base de données du dictionnaire et les exposera en REST à la partie *front-end*. Ce n'est pas une méthode qui sera fiable à long terme car avoir un accès

direct une base de données peut provoquer des problèmes de sécurité. Cependant, étant donné que la base de données du dictionnaire est temporaire avant une future évolution sur une autre technologie, nous avons opté pour cette solution en envisageant les possibilités avec les futures évolutions. La partie cartographie technique regroupera les analyses et les méthodes liées aux programmes et aux modules d'Amplitude.

2 Alimentation de la base

Plusieurs étapes sont nécessaires afin de récupérer l'intégralité des informations nécessaire à la cartographie : la création des programmes en base, la création des modules 4GL, la création des fonctions techniques, l'association des modules 4Gl et des programmes, l'association des fonctions et des programmes, l'extraction du dictionnaire des programmes et la récupération des SMG.

Auparavant, il sera nécessaire de faire interagir le projet avec les versions d'*Amplitude* en utilisant l'outil de gestion de version SVN. Ainsi, pour chaque version d'Amplitude que l'on souhaite étudier, il faudra effectuer un *checkout* d'Amplitude. C'est à partir des *working copy* récupérées que l'on va effectuer les analyses et les créations d'entités en base. Il sera nécessaire par la suite d'effectuer des *update* des versions lors des alimentations régulières pour observer les différences et mettre à jour les versions.

2.1 Création des programmes en base

La première phase de l'alimentation de la base est la création des programmes dans la base de données. La récupération des programmes se fait à partir du dossier contenant les *BaseProg* (voir la Section 2 pour voir l'architecture d'*Amplitude*). Ceux-ci vont nous permettre de récupérer plusieurs données qui nous intéressent : le nom des programmes, leur type quand celui-ci est indiqué et certaines caractéristiques comme le client ou la zone géographique quand elles sont disponibles. Ces données peuvent être récupérées d'après les noms des *BaseProg* qui respectent la convention de nommage suivante : `module.type.nom_programme`

Dans certains cas, on retrouve le nommage suivant, ajoutant des caractéristiques : `module.caractéristique.type.nom_programme`.

Ici, l'objectif sera de lister les fichiers de *BaseProg*, de parser chaque résultat en conservant les données que l'on souhaite, puis de créer les requêtes pour alimenter notre base.

2.2 Création des modules 4GL en base

Une fois les programmes enregistrés, l'étape suivante est l'alimentation des modules 4GL en base. La récupération des modules 4GL se fait en parcourant l'arborescence des fichiers sources d'Amplitude. Pour chaque dossier de l'arborescence, on liste les modules présents et on conserve en base les noms des fichiers avec leur répertoire d'origine.

2.3 Création des fonctions techniques en base

Plusieurs solutions sont possibles pour récupérer les fonctions techniques. Tout d'abord, on appelle fonctions techniques, les fonctions définies dans les modules 4GL. La méthode brute pour les récupérer serait donc de parser l'intégralité des fichiers sources pour y retrouver les fonctions définies. Cependant, il existe un outil en Python permettant de retrouver les

fonctions appelées dans un programme : *check_func_xml.py*. L'idée est donc de modifier ce script afin de retrouver les fonctions définies par module 4GL. Les modifications à effectuer devront permettre de sortir un flux texte exploitable par la suite dans le programme Java avec les indications suivantes : nom de la fonction, module 4GL où elle est définie et la liste des fonctions appelées par cette fonction. Ces informations devront être ensuite conservées en base avec Spring Data.

2.4 Association des modules 4GL et des programmes

À ce stade, les modules 4GL et les programmes ont été enregistrés en base. L'association des modules 4GL et des programmes se fera en utilisant le script Python *l4gl.py* qui affiche la liste des modules présents dans le baseprog d'un programme. À partir de cette liste, on associe le programme sélectionné aux modules affichés suite à l'exécution du script python. Parmi ces modules, on retrouve un module portant le même nom que le programme : le module principal du programme. Cette information sera à conserver en base.

2.5 Association des fonctions et des programmes

L'association des modules et des programmes se fera en utilisant le même script que pour la création des programmes en base : *check_func_xml.py*. Pour rappel, ce programme permet d'afficher les fonctions appelées par un programme donnée. Dans notre cas, il suffit d'appeler le script pour chaque programme et d'associer les fonctions affichées avec le programme. Un modification du code Python sera nécessaire pour afficher une sortie qui correspond à notre besoin.

2.6 Extraction du dictionnaire des programmes

L'utilisation des données du dictionnaire des programmes va nous permettre d'ajouter des informations supplémentaires aux programmes enregistrés dans notre base et aussi d'y ajouter les modules fonctionnels. Le dictionnaire des programmes se présente sous un fichier Excel avec deux feuilles de calcul : la première décrivant la liste des programmes avec leurs descriptions, leurs modules fonctionnels associés et d'autres informations, et la seconde lisant les modules fonctionnels ainsi que leur représentation sous forme de code à 3 caractères. L'objectif pour nous sera de parcourir la liste des modules et de les enregistrer en base dans un premier temps, puis, pour chaque programme du dictionnaire, on enregistre la description associée.

2.7 Récupération des SMG

Les SMG sont des programmes particuliers que l'on peut retrouver avec le type webservice. Les SMG ont néanmoins un autre nom qui désigne leur action (exemple d'un SMG : *createAccount*). La récupération de ces informations se fera à partir des fichiers xcf dans l'arborescence d'Amplitude. L'objectif sera de lister les fichiers xcf, de garder en mémoire les noms de fichier, et de parser chaque fichier xcf pour récupérer le nom du programme associé.

D'autres données concernant les SMG sont disponibles dans le dictionnaire des SMG. Il s'agit d'un fichier Excel similaire au dictionnaire des programmes qui va nous permettre d'ajouter des informations complémentaires, notamment la description du SMG.

9 Chiffrage du projet

Une fois les différentes étapes de développement posées, nous avons pu effectué un chiffrage du projet avec le nombre de jours attribué à chaque tâche ce qui va nous permettre d'avoir une idée sur la complexité de certaines mises en place.

Tâches	Nombre de jours
Architecture du projet	
Partie Modèle	1
Partie View	1
Back-end	
Création du modèle :	10
• tables + entities + dao + services avec Spring Data	
Alimentation	
Création des programmes en base :	3
• Récupération de la liste des programmes dans le BaseProg	
• Parsing des résultats (récupération du nom, du type, éventuellement des caracs pour certains programmes)	
• Alimentation de la table	
Création des modules 4gl en base :	2
• Parcours de l'arborescence d'Amplitude	
• Alimentation de la table	
Création des fonctions techniques en base :	2
• Modification du script *check_func_xml.py*	3
• Appel du script python et lecture du résultat	2
• Création des fonction techniques	2

• Liste des fonctions appelées par la fonction sélectionnée avec alimentation	2
Association des modules 4gl et des programmes :	
• Appel de *l4gl.py* et lecture du résultat	2
• Alimentation	2
• Détermination du module 4gl principal	1
Association des fonctions et des programmes :	
• Modification du script *check_func_xml.py*	1
• Appel du script	2
• Alimentation de la base	2
Extraction du dictionnaire des programmes :	
• Lecture du fichier Excel	3
• Alimentation de la base	2
Récupération des SMG :	
• Lecture du XCF	1
• Parse du fichier XCF	2
• Alimentation	1
• Lecture du fichier Excel pour alimentation	2
Implémentation des connexions SVN	2
Création du job	6
Front End	
Écrans	20

Table 1 – *Chiffrage des phases de développement de l'outil de cartographie*

Au total, le nombre de jours attribués pour finaliser l'outil est de 75, ce qui sera trop long par rapport à mon temps de stage restant. Selon nos comptes, à l'issue de mon stage, la partie *back-end* de l'application sera terminée, ce qui ne sera pas le cas pour la partie *front-end*.

Quatrième partie

Développement de l'outil

Les premiers développements ont débuté au mois de juin et se sont prolongés jusqu'à la fin de mon stage au mois d'août. Cette partie sera consacrée à l'implémentation des différents éléments de l'outil de cartographie.

10 Outils utilisés et arborescence du projet

Ce chapitre traite des différents outils utilisés pour le développement ainsi que la structure générale des sources du projet.

1 IDE et outils de développement utilisés

Les développements du projet se sont faits avec l'éditeur *Eclipse*. Il a l'avantage d'être l'un des plus complets et efficaces du marché, tout en étant open source. L'éditeur donne accès à une vaste bibliothèque d'extensions permettant d'améliorer l'efficacité du développement. Parmi ces extensions, j'ai notamment rajouté le plug-in *Sonarlint* qui permet d'analyser le code et de détecter les mauvaises pratiques de développement pouvant mener à des bugs ou des ralentissements. C'est une extension pratique pour garantir une qualité de code et pour améliorer les développements en général.

Une autre extension qui va nous servir sur le projet est *Angular IDE*. Il s'agit d'une extension permettant de prendre en charge les projets *Angular*, ce qui va nous permette de gérer le côté Java et le côté *Angular* du projet dans un seul éditeur.

2 Arborescence du projet

Le projet est séparé en deux parties avec le *back-end* d'un côté et le *front-end* de l'autre. L'idée est de respecter un pattern MVVM (Modèle - Vue - Vue/Modèle). Il s'agit d'un design pattern permettant de faciliter la mise en place d'interface graphique. Il a spécialement été conçu pour améliorer la séparation entre les données et la vue qui les affichent. Le lien entre la vue et le modèle de données est fait par des mécanismes de liaison dynamique. Ainsi la partie *back-end* contiendra les éléments du Modèle et du lien entre Modèle et Vues, et la partie *front-end* s'occupera des Vues. La Figure 1 représente l'arborescence du *back-end* de l'application.

La structure de la partie *back-end* reprend une arborescence classique de projet Java, avec les sources séparées en deux packages principaux : **main** pour les fichiers de sources et **test** pour les fichiers de tests unitaires. On peut voir la présence du fichier *pom.xml* à la racine du projet et qui contient les dépendances nécessaires pour notre outil. Parmi les dépendances (voir la Section 2 pour comprendre le fonctionnement), on retrouve les starters de *Spring Boot* (*spring-boot-starter-data-jpa*, *spring-boot-starter-web*, *spring-boot-starter-test*), deux dépendances pour *Spring Security*,

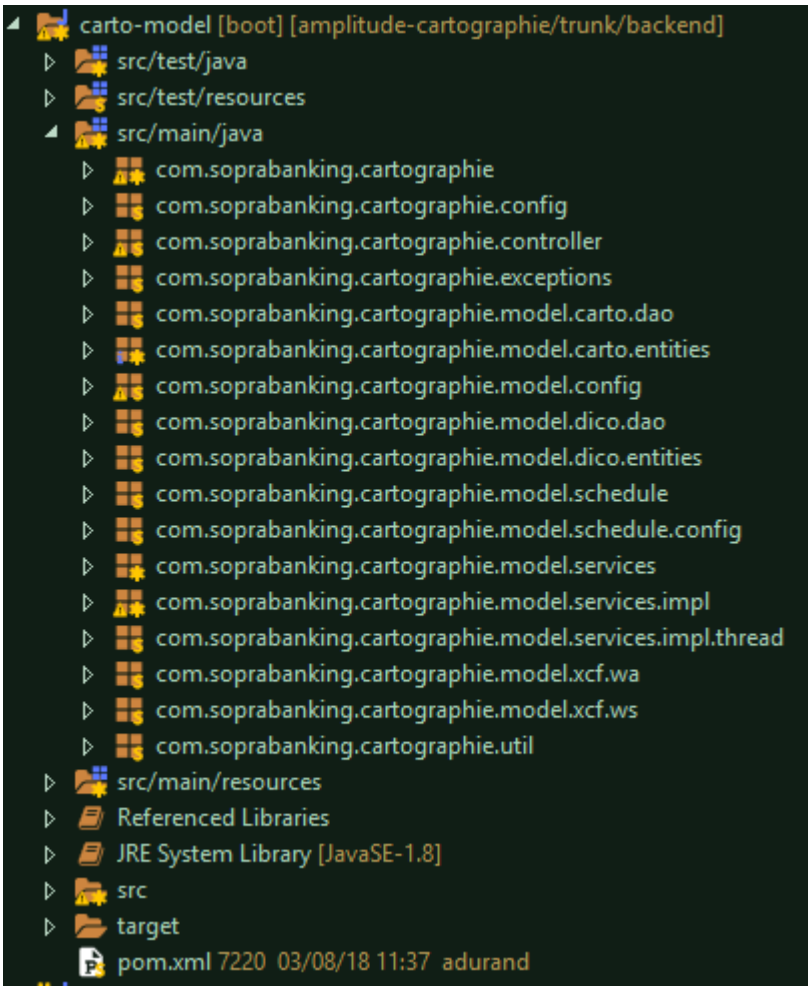


Figure 1 – *Arborescence du back-end de l'outil de cartographie*

les drivers pour les connexions à la base de données (une pour notre base sous *Oracle* et une pour la base du dictionnaire de données sous *Informix*).

Ensuite, dans les sources présentes dans le package **main**, les fichiers Java sont répertoriés dans un package principal *com.soprabanking.cartographie*. À la racine, nous avons le fichier **main** du programme et les packages suivants :

controller qui contient deux classes de contrôleur, une pour la partie Dictionnaire et une pour la partie Cartographie. Ce sont ces deux contrôleurs qui vont exposer les données en REST au *front-end*.

exceptions avec les exceptions pour certains cas de figure.

model avec toute la gestion des données de l'outil : les classes d'entités de *Spring Data*, les *repositories*, la gestion du job d'alimentation et les services *Spring Data*.

util qui contient les classes utilitaires comme par exemple, la gestion des connexions SVN.

Comme expliqué, on retrouve de nombreux éléments différents dans la partie **model** qui respectent la structure suivante :

carto contient les entités de la cartographie technique (programmes, modules 4GL, fonctions techniques, ...) et leurs *repositories* respectifs permettant la gestion des données avec *Spring Data*

dico contient les entités du dictionnaire des données (domaines, fonctions, sous-fonctions, tables, ...) et leurs *repositories* respectifs permettant la gestion des données avec *Spring Data*

config contient deux classes de configuration pour les transactions vers les bases de données.

schedule contient les éléments du job d'alimentation.
xcf contient les classes pour l'extraction du dictionnaire des programmes.

Les fichiers de ressources contenant les propriétés de l'application et du job d'alimentation sont contenus dans le package **main/resources**.

Du côté *Angular*, la structure reprend celle d'un projet *Angular* classique dont nous verrons plus de détails dans le Chapitre 13.

L'empaquetage de l'application se fait par *Maven*. Nous avons donc un projet parent et deux projets fils, le tout relié par *Maven*. Le côté *back-end* se comporte comme un projet *Maven* classique, tandis que le côté *front-end*, lors de son déploiement, installe au préalable les dépendances et les outils nécessaires au fonctionnement d'*Angular* (*Angular-cli*, *npm*, *ng*, tous les éléments détaillés dans le Chapitre 6). Ainsi, au déploiement de l'application, les deux sous-projets sont déployés chacun de leur côté.

11 Exposition REST des données

1 Principe du REST

L'un des points central de notre outil est l'exposition des données de la partie *back-end* à la partie *front-end*. Et l'exposition des données se fera par le biais d'une architecture REST. Le REST (Representational State Transfer) est un style d'architecture permettant de construire des applications. Il s'agit davantage d'un ensemble de conventions et de bonnes pratiques plutôt qu'une technologie à part entière. Cette architecture utilise les spécifications originelles du protocole HTTP.

REST se base sur les URI (Uniform Resource Identifier) afin d'identifier une ressource. Ainsi une application se doit de construire ses URI (et donc ses URL) de manière précise, en tenant compte des contraintes REST. Il est nécessaire de prendre en compte la hiérarchie des ressources et la sémantique des URL pour les éditer. Dans notre cas par exemple, pour récupérer l'ensemble des domaines présents dans le dictionnaire des données, nous pouvons utiliser l'URL : `/dico/domaines`. Pour en récupérer un avec un identifiant précis, on utilise l'URL : `/dico/domaines/{id}` avec l'identifiant que l'on souhaite.

Ensuite, la gestion des ressources se fait en utilisant des requêtes HTTP. Ainsi, on peut effectuer 4 opérations (les 4 opérations du CRUD) :

- Créer (create) ⇒ **POST**
- Afficher (read) ⇒ **GET**
- Mettre à jour (update) ⇒ **PUT**
- Supprimer (delete) ⇒ **DELETE**

Une exemple avec notre outil, une mise à jour d'un programme de la cartographie se fera avec une requête **PUT** sur l'URL `/carto/programmes/{id}` avec l'identifiant du programme que l'on souhaitera modifier.

2 Implémentation avec *Spring Data*

L'implémentation du REST avec *Spring Data* se fait facilement à l'aide des différentes annotations à disposition. Tout d'abord, il faut préciser que les contrôleurs de l'application prennent en compte le REST avec l'annotation `@RestController`. Ensuite, il faut indiquer *Spring* le *mapping*

de l'application, en d'autres termes, les URL qui seront utilisées pour le partage des ressources. Le *mapping* se fait avec des annotations `@RequestMapping` sur la classe du contrôleur pour indiquer le contexte général, puis sur les méthodes qui vont gérer les ressources. La Figure 1 montre une implémentation de l'architecture REST.

```java
@RestController
@RequestMapping("/dico")
public class DicoController {

    @Autowired
    private DicoService dicoService;

    @PreAuthorize("isAuthenticated()")
    @RequestMapping(value = "/domaines/{id}", method = RequestMethod.GET)
    public ResponseEntity<DicoDomaine> domaineById(@PathVariable("id") String id) {
        Optional<DicoDomaine> dicoDomaine = dicoService.findDomaineById(id);
        if (!dicoDomaine.isPresent()) {
            return ResponseEntityBuilder.getResponseEntityNotFound("DicoDomaine with id " + id + " not found");
        }
        return ResponseEntityBuilder.getResponseEntityOk(dicoDomaine);

    }
```

Figure 1 – *Prise en charge du REST avec Spring Data*

On peut voir sur l'illustration l'annotation `@RequestMapping("/dico")` qui précède la déclaration du contrôleur ainsi que l'annotation `@RequestMapping("/domaines/{id}")` avant la déclaration de la méthode qui récupère un domaine selon l'identifiant. Ainsi, il est possible de retrouver les domaines que l'on souhaite par le biais de l'URL : `/dico/domaines/{id}`.

12 Gestion du job d'alimentation

L'un des aspects principales de l'outil est l'alimentation des différents ressources. Le processus d'alimentation doit nous permettre de récupérer les éléments de la cartographie technique, les données du dictionnaires des données nous sont déjà directement accessibles depuis la base de données. L'alimentation et la mise à jour des ressources doit se faire régulièrement, pour avoir la meilleure cartographie possible à un instant T. Nous avons choisi de l'effectuer tous les jours à minuit, pour permettre d'avoir la bonne cartographie chaque jour.

Pour la gestion du processus d'alimentation, nous avons utilisé la bibliothèque open-source *Quartz*. Elle permet de programmer des processus pour qu'ils se lancent au moment que l'on souhaite. La mise en place d'un job se fait avec à minima deux fichiers : un décrivant le job et l'autre contenant les configurations. Le fichier qui décrit le processus à accomplir doit implémenter l'interface *Job* de la bibliothèque *Quartz*. Ensuite la classe doit avoir une méthode `execute()` qui indique les traitements à effectuer. La configuration du job se trouve dans une classe qui précise les détails du job et qui met en place le déclencheur du processus (le *trigger*).

Pour notre outil de cartographie, le job va appeler les méthodes de récupération d'*Amplitude* avec SVN et d'alimentation des données par la suite. La Figure 1 représente l'implémentation dans la classe *AlimentationCartoService* de l'appel aux méthodes d'alimentation.

Comme on le voit sur l'image, les méthodes appelées sont les suivantes :

- Des appels à la méthode *svnCall()* pour récupérer les outils de développement d'*Amplitude* (pour utiliser le script python pour la récupération des fonctions techniques notamment ;
- Un appel à la méthode *svnCall()* pour récupérer les sources *Amplitude* pour la version indiquée en paramètre ;
- Un appel à la fonction de création des programmes ;
- Un appel à la fonction de création des modules 4GL ;
- Un appel à la fonction de créations des fonctions techniques ;
- Un appel à la fonction d'associations des programmes et des modules 4GL ;
- Un appel à la fonction d'extraction du dictionnaire des programmes ;
- Un appel à la fonction de récupération des SMG.

La fonction `svnCall()` effectue un *checkout* des sources dans le cas où il s'agit de la première alimentation ou un *update* dans le cas où les sources sont déjà présentes.

Les méthodes sont appelées une à une de façon séquentielle. La question du multi-threading s'est posée pour améliorer les temps de traitements. En effet, la création initiale des entités en base peut prendre beaucoup de temps. La création des programmes et des modules se fait

```
try {

    svnCall(repeOutil.getCanonicalPath(),
            "https://svn.tour.fr.sopra/svnbank/svn/UTI/repository/trunk/pytools_carto/bin/");
    svnCall(repeEtc.getCanonicalPath(),
            "https://svn.tour.fr.sopra/svnbank/svn/UTI/repository/trunk/pytools_carto/etc/");

    svnCall(path, version.getUrlSvn());

    logger.info("Début de l'alimentation des programmes");
    alimentationProgrammeService.creationListeProgramme(version, repeTravail);
    logger.info("Fin de l'alimentation des programmes");

    logger.info("Début de l'alimentation des modules");
    alimentationModule4GLService.creationModule4GL(version, repeTravail);
    logger.info("Fin de l'alimentation des modules");

    logger.info("Début de l'alimentation des fonctions techniques");
    alimentationFonctionService.creationFonction(version, repeOutil);
    logger.info("Fin de l'alimentation des fonctions techniques");

    logger.info("Début de l'association entre programmes et modules 4GL");
    associationFonctionModuleService.associationProgrammeModule4GL(version, repeOutil);
    logger.info("Fin de l'association entre programmes et modules 4GL");

    logger.info("Début de l'extraction du dico des programmes");
    extractionDicoProgrammeService.extractionDico(version);
    logger.info("Fin de l'extraction du dico des programmes");

} catch (Exception e) {
    logger.error(e.getMessage());
    throw e;
}
```

Figure 1 – *Implémentation des méthodes d'alimentation*

relativement rapidement tandis que la création des fonctions prend à l'heure actuelle, un temps qui n'est pas concevable pour une mise en production (aux alentours de 10 heures de traitement). Cela est dû à l'utilisation du script Python *check_func_xl.py* qui nécessiterait une amélioration et au fait que chaque programme lance une instance de Python avec une exécution de script. Ainsi, nous avons mis en place un multi-thread sur ces traitements, ce qui permet de prendre en charge plusieurs scripts en simultané.

13 Gestion du front-end et de la sécurité

La partie *front-end* et la gestion de la sécurité a été pris en charge par les architectes. Je ne vais donc pas rentrer dans les détails des implémentations mais plutôt évoquer le fonctionnement général de ces aspects.

1 Gestion de la sécurité

L'outil final pourrait être accessible à de nombreuses personnes au sein de l'entreprise une fois la mise en production. La nécessité de sécuriser la plate-forme est présente étant donné qu'il est possible de manipuler des données faisant parties intégrantes de la cartographie. Une modification même minime sur les données pourrait entraîner des problèmes, notamment si cela touche des données associées avec plusieurs entités.

Les authentifications à la plateforme sont gérées avec le framework *oauth2*. C'est un framework qui permet de limiter les accès d'un compte sur les services HTTP. Il est utilisé par de nombreux sites réputés comme *Facebook* ou *Github*. L'utilisation du framework sur notre application est gérée automatiquement par *Spring Security*. La gestion des authentifications à l'aide de jetons attribués à un compte une fois celui-ci connecté.

On peut voir sur la Figure 1 (Chapitre 11) dans le chapitre précédent la présence de l'annotation `@PreAuthorize("isAuthentificated()")`. Celle-ci permet de refuser l'accès à la méthode si le visiteur ne s'est pas authentifié au préalable.

2 Aperçu de l'affichage

Les deux images suivantes montrent un aperçu du rendu de l'outil de cartographie. La première page que l'on trouve lorsque l'on souhaite utiliser l'outil est la Figure 1, la page d'authentification. Une fois authentifié, nous arrivons sur la page d'accueil de l'application que l'on peut voir sur la Figure 2. L'aperçu de la page d'accueil ici est provisoire, néanmoins la structure finale sera la même, à savoir un aperçu des domaines de la cartographie fonctionnelle. Un clic sur une tuile représentant un domaine permet d'avoir un visuel du contenu de l'entité.

Parmi les autres éléments notables de la page d'accueil, on peut retrouver les onglets *Recherche*, *Statistiques* et *Habilitations*. L'onglet de recherche, comme son nom l'indique, permet de rechercher un élément que l'on souhaite analyser. L'onglet *Statistiques* va regrouper différentes

Figure 1 – *Page d'authentification de l'outil de cartographie*

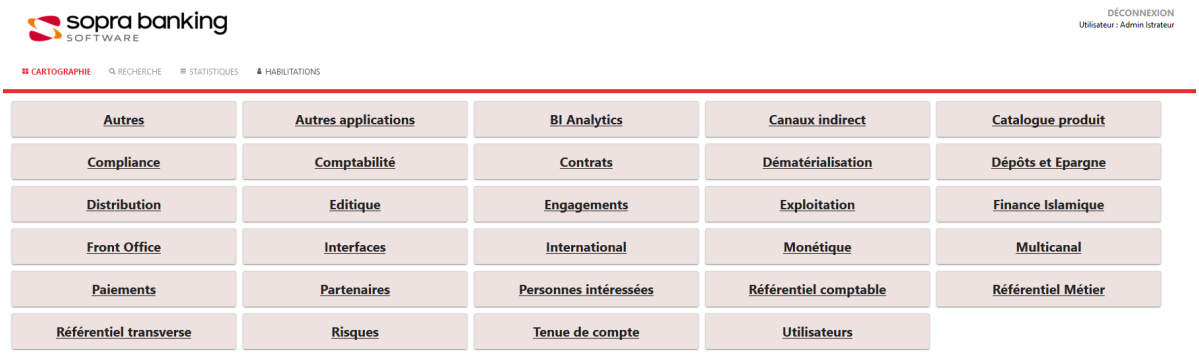


Figure 2 – *Page d'accueil de l'outil de cartographie*

informations générales comme les fonctions inutilisées, les modules les plus utilisés, etc.. Les habilitations seront accessibles uniquement pour les administrateurs de l'outil.

Ces éléments ne sont pas encore intégrés et seront les prochaines étapes de développement dans le futur.

Bilan du stage

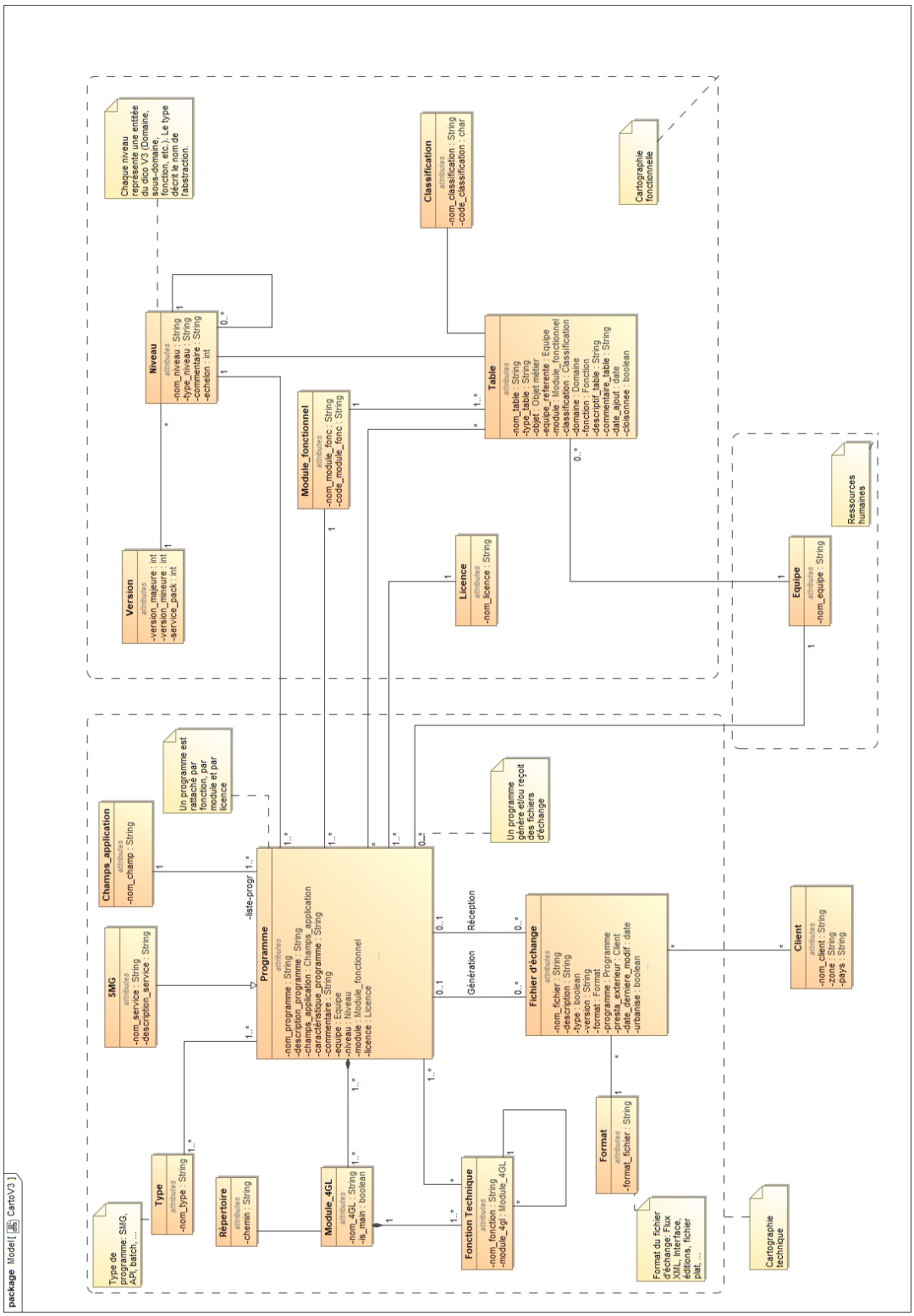
À l'heure où s'achève mon stage, le moment est venu de dresser le bilan de ces derniers mois. Dans un premier temps, concernant l'état du projet à l'issue du stage, nous avons respecté les délais fixés par les chiffrages de développement. Après l'analyse de l'expression des besoins, nous savions que les temps pour finir intégralement seraient très courts et qu'il serait quasiment impossible de terminer l'outil à la fin du mois d'août. Néanmoins, notre objectif était d'avoir au moins la partie *back-end* de l'application fonctionnelle, ce qui est le cas. Des améliorations sont cependant encore nécessaires, notamment en ce qui concerne les temps de traitement pour l'alimentation des données.

D'un point de vue personnel, ce stage a été pour moi l'expérience professionnelle la plus enrichissante que j'ai connu, et de loin, et cela pour plusieurs raisons. Tout d'abord, ça a été la première fois pour moi d'être intégré dans un projet à plein temps sur une période aussi longue. Ensuite, le sujet était particulièrement complet et m'a permis de voir de toucher à de nombreux aspects de la vie d'un ingénieur. Commencer par comprendre les besoins du "client" pour ensuite modéliser une solution qui convient avant de la mettre en œuvre était vraiment intéressant. La variété aussi bien des tâches à accomplir que des technologies utilisées a fait que je n'ai jamais ressenti de moments d'ennui ou de lassitude pendant le stage, ce qui peut parfois arriver pour certains étudiants-ingénieurs.

Pour conclure, je tiens à remercier mon encadrant, M. Alexandre DURAND, qui m'a fait confiance dès le passage des portes de *Sopra Banking Sotware* lors de mon premier entretien jusqu'à mes derniers jours de stage et même dans un futur proche en me proposant un contrat de travail. J'espère pouvoir ainsi terminer cet outil de cartographie et qu'il soit utile au plus grand nombre dans l'entreprise. Je voudrais ensuite remercier mes collègues pour l'intégration dans la cellule dès les premiers jours, pour la bonne ambiance qui règne au quotidien et pour leur humour à toute épreuve ! Et enfin merci à Polytech pour ses années d'étude qui vont me permettre de rentrer dans la vie active de la meilleure des façons.

Annexes

A Diagramme de classe


package Model [CartoV3]
Chaque niveau correspond à une entité du dico V3 (Domaine, sous-domaine, fonction, etc.). Le type décrit le nom de l'abstraction.
Niveau
-nom_niveau : String
-type_niveau : String
-commentaire : String
-echelon : int
Version
-version_majeure : int
-version_mineure : int
-service_pack : int
Classification
-nom_classification : String
-code_classification : char
Cartographie fonctionnelle
Table
-nom_table : String
-type_table : String
-objet : Objet métier
-equipe_referente : Equipe
-module : Module_fonctionnel
-classification : Classification
-domaine : Domaine
-fonction : Fonction
-description_table : String
-commentaire_table : String
-date_ajout : date
-cloisonnee : boolean
Module_fonctionnel
-nom_module_fonc : String
-code_module_fonc : String
Licence
-nom_licence : String
Equipe
-nom_equipe : String
Ressources humaines
Champs_application
-nom_champ : String
Un programme est attaché par fonction, par module et par licence
Un programme génère et/ou reçoit des fichiers d'échange
SMG
-nom_service : String
-description_service : String
Programme
-nom_programme : String
-description_programme : String
-champs_application : Champs_application
-caractéristique_programme : String
-commentaire : String
-equipe : Equipe
-niveau : Niveau
-module : Module_fonctionnel
-licence : Licence
-liste-progr
Réception
Génération
Fichier d'échange
-nom_fichier : String
-description : String
-type : boolean
-version : String
-format : Format
-programme : Programme
-presta_exterieur : Client
-date_derniere_modif : date
-urbanise : boolean
Client
-nom_client : String
-zone : String
-pays : String
Type
-nom_type : String
Répertoire
-chemin : String
Module_4GL
-nom_4GL : String
-is_main : boolean
Fonction Technique
-nom_fonction : String
-module_4gl : Module_4GL
Format
-format_fichier : String
Type de programme: SMG, API, batch, ...
Format du fichier d'échange: Flux XML, Interface, éditions, fichier plat, ...
Cartographie technique

Réalisation d'un outil de cartographie sur le progiciel Amplitude

Résumé

Le stage de cinquième année d'école d'ingénieur est la dernière expérience avant l'entrée dans la vie active. Mon stage consiste en la mise en place d'un outil de cartographie pour le progiciel bancaire *Amplitude* développé par l'entreprise *Sopra Banking Software*. Le progiciel étant très vaste et complexe à manipuler, la mise en place d'une cartographie permettrait d'améliorer considérablement le quotidien des architectes fonctionnels. L'outil sera une plateforme interne à l'entreprise développée avec des technologies récentes comme *Spring Boot*, *Spring Data* ou encore *Angular*.

Mots-clés

stage, outil, cartographie, spring boot, spring data, angular

Abstract

The internship of the fifth year in engineering school is the last experience before the entrance in the working life. My internship consists in the built of a tool designed for mapping the software package *Amplitude* designed by *Sopra Banking Software*. The software has a very wide range of different fiels, making a map of it would improve the everyday life of the developpers. The tool will be a intern platform for the company developped with new technologies like *Spring Boot*, *Spring Data* or *Angular*.

Keywords

internship, tool, mapping, spring boot, spring data, angular

Entreprise
Sopra Banking

Étudiant
Romain ROUSSEAU (DI5)

Tuteur académique
Yannick KERGOSIEN

Tuteur entreprise
Alexandre DURAND (Responsable Cellule Architecture)