Plateforme extensible de modélisation et de construction d'applications web correctes et évolutives, avec hypothèse de variabilité

David Sferruzza

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE NANTES

THESE DE DOCTORAT DE

L'UNIVERSITE DE NANTES

COMUE UNIVERSITE BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique et applications, section CNU 27*

Par
David SFERRUZZA

Plateforme extensible de modélisation et de construction d'applications web correctes et évolutives, avec hypothèse de variabilité

Thèse présentée et soutenue à Nantes, le 13 septembre 2018
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

Jean-Paul BODEVEIX	Professeur, Université Toulouse 3, IRIT
Marianne HUCHARD	Professeur, Université de Montpellier, LIRMM

Composition du jury :

Président :	Régine LALEAU	Professeur, Université Paris-Est Créteil, LACL
Rapporteurs :	Jean-Paul BODEVEIX	Professeur, Université Toulouse 3, IRIT
	Marianne HUCHARD	Professeur, Université de Montpellier, LIRMM
Examinateurs :	Achour MOSTEFAOUI	Professeur, Université de Nantes, LS2N
	Jérôme ROCHETEAU	Enseignant-chercheur, ICAM de Nantes, LS2N
Dir. de thèse :	Christian ATTIOGBÉ	Professeur, Université de Nantes, LS2N

Invités :

Arnaud LANOIX	Maître de conférences, Université de Nantes, LS2N
Mathieu LE GAC	Fondateur, Startup Palace

Histoire d'un aller et retour

Lorsque je me suis lancé dans ce projet de thèse, c'était avant tout pour faire des rencontres, apprendre de nouvelles choses et me dépasser. Aujourd'hui je tiens à remercier une partie de celles et ceux qui m'ont soutenu, accompagné, inspiré et aidé à porter mon fardeau tout au long de ces trois ans et neuf mois :

Thomas B. Louma M.
Célia R.-U. Marie L. Ana Maria A.
Antoine B.-C. Soraya M. Laurence S. Caroline L.
Florian K. Dimitri P. Matthieu B. Quentin R.
Thibault M. Clément D. Christian A. Maxime B.
Killian B. Rachel T. Emmanuel F. Florian R.
Julien T. Romain G. Grégoire H. Carole S.
Julien R. Gwenola K. Romain A. Erwan D.
Maxime P. Charlotte S. Alexis G. Agathe V.
Adeline G. Olwen G. Geoffroy C. Franck L.
Jonathan P. Elisabeth H. Jérôme R. Vincent R.
Florian B. Quentin A. Audrey C. Joseph S.
Clément T. Mathieu L.G. Céline J.
Olivia B. Julien D. Pauline F.
Nadia L. Clément M. Arnaud L.
Benoît D. Nicolas D.
François-Guillaume R.
Godefroy de C.
Manon T.
♡

Sommaire

Table des figures

Liste des tableaux

Liste des listings

Première partie

Introduction et état de l'art

Chapitre 1

Introduction

Sommaire

Cette thèse a été effectuée dans le cadre d'une Convention Industrielle de Formation par la REcherche (CIFRE). Elle a donc été réalisée dans un contexte mixte, à la fois académique (LS2N [1]) et industriel (Startup Palace [2]). Le contexte de l'entreprise est un des éléments fondateurs du projet, c'est pourquoi cette introduction débute par la description de ce contexte. Le reste de l'introduction en découle : d'abord le contexte et les problèmes scientifiques, puis les contributions.

1.1 Genèse de la thèse

En janvier 2015, au début du projet de thèse, la société **Escale Digitale** réalisait des applications web « sur-mesure » à forte valeur ajoutée pour ses clients, qui étaient pour la majorité des PME. Cette valeur ajoutée se traduit notamment par la grande attention portée à l'expérience utilisateur des applications produites,

1. Laboratoire des Sciences du Numérique de Nantes ; https://www.ls2n.fr
2. https://www.startup-palace.com

sans pour autant reculer devant des projets ambitieux techniquement. C'était son positionnement stratégique sur ce marché très concurrentiel. En effet, de nombreux acteurs délèguent le développement de telles applications à des sociétés situées dans des pays émergents dont les coûts de développement sont beaucoup plus faibles.

Afin de minimiser le coût de ses réalisations, Escale Digitale avait initié depuis sa création en 2010 une démarche de recherche et développement lui permettant de produire rapidement, simplement et interactivement des applications web, tout en tentant de maximiser leur qualité, fiabilité et maintenabilité. Cette démarche était concrétisée par une plateforme d'ingénierie web nommée **GestionAIR**. Cette plateforme avait atteint ses limites mais avait montré l'intérêt pour Escale Digitale d'investir dans la poursuite de ce projet, car il offrait tout de même un gain de productivité intéressant pour les développeurs. Cette thèse a donc pour objectif de reprendre la démarche dans son ensemble, en donnant au projet une portée scientifique et en tentant de créer de meilleurs outils et solutions. La nature et les limites de GestionAIR sont développées dans le Chapitre 2 ; par exemple, la solution manquait d'un moyen de faire de la vérification, ce qui provoquait régulièrement des problèmes en production lors des évolutions.

En avril 2017, Escale Digitale a fusionné avec deux autres sociétés pour créer **Startup Palace**. Bien qu'ayant une mission différente de celle d'Escale Digitale, cette nouvelle entreprise a conservé la volonté d'accompagner ses clients dans la réalisation d'applications web. Cette évolution du partenaire industriel a donc peu influencé le contexte sur lequel se construisait cette thèse. Dans la suite du document, l'entreprise sera toujours nommée « Startup Palace » pour des raisons de simplicité, même si ces mentions font parfois référence à des éléments liés à Escale Digitale.

1.2 Contexte de l'ingénierie web et problématiques

Pour accompagner certains de ses clients (le plus souvent des *startups*), Startup Palace réalise des applications web. Ces applications web sont un moyen pour les startups de proposer de la valeur à leurs utilisateurs. Pour convaincre de la pertinence de sa proposition de valeur, une startup doit régulièrement tester et valider ou non des hypothèses de marché. Cela lui permet d'identifier un ou plusieurs marchés que sa proposition de valeur peut lui permettre d'exploiter. Le processus de réalisation de telles applications web est donc crucial. D'ailleurs, la prolifération ces dernières années de jeunes entreprises liées de près ou de loin aux technologies numériques illustre bien un des intérêts majeurs de ce

domaine : le coût nécessaire pour tester une proposition de valeur est plus faible que dans la plupart des autres industries dans lesquelles investir suffisamment de temps, de matériel et de main d'œuvre est très cher, ce qui rend l'entreprise d'autant plus risquée.

Il convient de définir ici le concept d'*application web*. Malheureusement, nous n'avons pas trouvé dans la littérature de définition suffisamment précise et adoptée par les acteurs de l'industrie (voir Section 3.1). En effet c'est un terme qui est employé couramment dans de nombreux contextes et milieux, lui donnant une signification floue. On abuse parfois du langage en le mélangeant avec les termes « services web », « site web » ou « site Internet ». Nous proposons donc les définitions suivantes :

Services web. *Application* répartie conçue pour permettre des interactions de machine à machine au sein d'un réseau, via le protocole Hypertext Transfert Protocol (HTTP) ;

Application web. Service web délivrant des données et documents, potentiellement interactifs, mettant en œuvre les technologies web (HTML, CSS, JavaScript, . . .) au travers d'un navigateur web ;

Site web (ou site Internet). Application web dont la fonction principale est de délivrer du contenu de manière simple et sans offrir beaucoup d'autres fonctionnalités interagissant avec l'utilisateur.

Ces définitions ne sont pas très précises, mais permettent d'acquérir de l'intuition concernant le lien entre services web et applications. Un site web est un cas particulier, simple, d'application web. Ce terme est mentionné ici uniquement parce qu'il est présent dans le langage courant. Les sites web ne seront pas évoqués directement dans ce document.

Pour réaliser le projet de construction d'une application web, il est souvent intéressant et sain d'adopter une démarche descendante et de commencer par concevoir des services web qui implémentent les fonctionnalités désirées. Cela se justifie par plusieurs aspects. Premièrement, cela permet de se concentrer uniquement sur la logique applicative métier (les données et leurs traitements) dans un premier temps. Il est d'ailleurs plus facile d'écrire de la documentation et des tests fonctionnels automatisés pour des services web que pour une application web. De plus, dans certains projets, il est nécessaire d'avoir plusieurs interfaces utilisateur différentes (web, mobile, . . .) pour agir sur un même système ; isoler le traitement et la centralisation des données (les services web) de ces interfaces utilisateur permet :

— de forcer le respect de certaines contraintes à l'ensemble du système ;

— de séparer les responsabilités ;
— de travailler à plusieurs équipes ;
— d'être facilement en mesure d'ajouter de nouvelles interfaces utilisateur au cours de la vie du projet ;
— de séparer les différents cycles de vie (par exemple, permettre des mises à jour rapides d'une application mobile).

Dans d'autres projets, qui ne sont constitués que d'une seule application web et dans lesquels il n'y a pas besoin d'exposer des services web, en écrire quand même ne fait pas perdre beaucoup de temps et augmente la maintenabilité.

Dans cette thèse, nous avons choisi de ne pas traiter des applications web dans leur ensemble mais de nous focaliser sur le sujet de la conception et la construction de services web. Comme argumenté précédement, réaliser des services web est une tâche qui peut être vue à la fois comme une fin et un moyen dans le développement de nombreux projets.

Conceptuellement, la plupart des services web sont des programmes qui transforment des requêtes HTTP en réponses HTTP. En réalité, le processus est plus complexe et l'écriture de ces programmes pose diverses questions, communes dans le domaine de l'ingénierie logicielle. Par exemple : comment être productif ? Comment ne pas faire d'erreur ? Comment faire évoluer le programme ? Par ailleurs, de nombreuses contraintes techniques qui ne sont pas directement liées à la logique métier viennent s'ajouter. Par exemple : comment interagir avec un Système de Gestion de Base de Données (SGBD) ? Comment utiliser le matériel disponible pour réaliser des calculs intensifs en parallèle ? Le contexte de l'ingénierie web ajoute du relief à ces questions : les services web sont des applications qui utilisent le réseau, sont parfois distribuées, ont des contraintes de disponibilité importantes et font souvent l'objet de prototypage rapide.

Dans cette thèse, nous enrichissons ce contexte en présentant les trois sous-problèmes qui nous intéressent : la variabilité des technologies, celle des spécifications et la productivité des développeurs.

1.2.1 Variabilité des technologies

Comme beaucoup de sous-domaines du génie logiciel, l'ingénierie web repose sur un ensemble de standards, langages et environnements d'exécution qui lui sont partiellement ou complètement spécifiques. Contrairement à beaucoup d'autres sous-domaines du génie logiciel, ces trois aspects évoluent très rapidement avec le temps dans le cas du développement web. En effet, les implémentations se succèdent en permanence ou, pire encore, les technologies sont elles-mêmes

éphémères. La variabilité, soit l'aptitude ou la propriété à subir des variations, concerne à la fois les standards, les langages et les clients. Par exemple :

Standards. Mise à jour occasionelle de nombreuses spécifications concernant le web par le W3C, le WHATWG et ECMAScript pour la partie client, ou évolutions du protocole HTTP par l'IETF ;

Langages. Apparition régulière de nouveaux langages et frameworks. Par exemple le framework Angular [22], un framework édité par Google pour le langage TypeScript [3], lui-même développé par Microsoft comme complément à JavaScript. On peut également citer le langage Scala [4] avec des outils comme Play Framework [5] (un framework web côté serveur pour Scala édité par Typesafe) ;

Clients. Accélération des mises à jour pour l'implémentation des standards, mais aussi ajout de nouvelles fonctionnalités non-standardisées. Les navigateurs web, comme Mozilla Firefox [6], ne sont pas les seuls clients web majeurs et les environnements d'exécution sont variés.

Le développement d'une application web dépend de nombreuses bibliothèques logicielles dans différentes technologies, parfois organisées en couches, sans que les développeurs ne maîtrisent complètement ces différentes strates. C'est l'un des intérêts d'utiliser des bibliothèques externes : pas besoin de maitriser leur implémentation, juste leur Application Programming Interface (API). Afin d'assurer productivité et réactivité, il est nécessaire pour des sociétés comme Startup Palace d'être à la pointe de ces technologies sous peine de produire des applications à l'aide de technologies obsolètes.

Les applications logicielles développées par Startup Palace doivent souvent être mises à jour. Ces mises à jour ne concernent pas que des ajouts de fonctionnalités : elles peuvent aussi apporter des changements liés aux technologies utilisées, par exemple, pour les accès aux données issues de divers systèmes d'information ou pour présenter les vues des applications. Nous sommes dans un contexte de forte variabilité technologique où la réactivité des développeurs par rapport aux nouveaux besoins des clients ou usagers des applications est un critère prépondérant.

Par exemple, des services web peuvent dépendre d'une technologie externe, sous la forme d'une API HTTP, qui indique le temps d'attente des prochains

3. https://www.typescriptlang.org/
4. https://www.scala-lang.org/
5. https://playframework.com/
6. https://www.mozilla.org/fr/firefox/

bus à un arrêt donné, potentiellement desservi par plusieurs lignes. Un jour, cette API externe peut évoluer et modifier la forme des données qu'elle fournit ; imaginons que tous les bus ne desservent plus l'intégralité la ligne et donc que certains sont marqués comme faisant uniquement un parcours partiel : l'API va alors changer son contrat (implicitement ou explicitement). Les services web qui sont basés sur elle vont peut-être alors devoir évoluer également : il est possible que cette évolution ne soit pas nécessaire car l'API externe a évolué de manière rétro-compatible, mais il est également possible que ça ne soit pas le cas et que certains paradigmes ou certaines conventions aient changé. Si elle est rétro-compatible, il arrive tout de même que les anciennes versions de cette API ne soient plus supportées au bout d'un moment. Dans tous les cas, garder les services web opérationnels peut nécessiter des évolutions, entraînant elles-même des problèmes en cascade.

Cet exemple illustre le concept de dette technique : si l'API évolue de manière rétro-compatible, les développeurs peuvent décider de ne pas altérer leurs services web, mais ils risquent d'acquérir de la dette technique. Si plus tard ils doivent tout de même faire évoluer leurs services web pour qu'ils soient compatibles avec la dernière version de l'API externe, par exemple pour ajouter une nouvelle fonctionnalité ou parce que l'ancienne version de l'API n'est plus supportée, alors ils devront solder cette dette technique, c'est-à-dire payer le prix du changement qu'ils n'ont pas fait par le passé, parfois avec des intérêts. Tous les programmes accumulent de la dette technique ; une partie du métier de développeur est d'évaluer ces « coûts » qui sont liés à la maintenabilité du programme, et de décider quand contracter de la dette et quand en solder.

Q1 : Comment donc faire face à la forte évolutivité des technologies web tout en produisant des applications web de qualité, en un temps réduit et en minimisant la dette technique du processus de développement ?

1.2.2 Variabilité des spécifications

À cette variabilité technologique vient s'ajouter une variabilité des spécifications des applications. Comme énoncé précédemment, une grosse partie des clients de Startup Palace sont des entreprises innovantes qui ont besoin d'une application web pour tester rapidement leur marché. Ces clients ont préféré faire appel à Startup Palace plutôt qu'à un de ses concurrents pour sa capacité à prendre en charge l'ensemble des étapes de la création de l'application, de l'étude du besoin au développement, l'ergonomie, le graphisme et pour sa **démarche itérative**.

Cette démarche est très importante : le client étant en train d'essayer de définir

et de stabiliser son activité en fonction du marché, il a besoin d'avoir des retours rapides sur son approche pour pouvoir l'ajuster. Startup Palace va l'aider à réaliser un *produit minimum viable* (ou Minimum Viable Product (MVP)) c'est-à-dire une solution minimale en termes d'efforts et de coûts, mais qui est tout de même crédible aux yeux de ses utilisateurs et leur apporte de la valeur.

Avant d'obtenir un produit de cette qualité, il est souvent nécessaire de remettre en question beaucoup d'hypothèses de conception, et de re-développer ou adapter les parties de l'application ou des spécifications qui dépendent de ces hypothèses. Ces évolutions sont grandement facilitées par la capacité de l'application à être maintenable.

Par ailleurs, ces changements fondamentaux ainsi que les modifications qu'ils induisent vont presque toujours introduire des erreurs qui peuvent être critiques, d'un point de vue technique ou pour l'expérience utilisateur. Il est donc important, en plus de favoriser la maintenabilité, de pouvoir limiter la portée des erreurs humaines qui sont à l'origine de ces erreurs. Ces deux aspects sont primordiaux lorsqu'il s'agit de développer de manière rapide et fiable des applications web dans ces conditions de variations de spécifications et de choix de conception.

Q2 : Quel compromis trouver entre maintenabilité et productivité pour permettre la variation des spécifications sans perte de qualité ?

1.2.3 Productivité des développeurs

Pour chaque application web, on peut envisager les différentes fonctionnalités de la manière suivante :

Game Changers. Les fonctionnalités qui vont pouvoir être faites tellement bien qu'il sera possible d'en retirer un gain direct (avantage concurrentiel du client, satisfaction du client, ...). La présence d'un *game changer* bien réalisé offre un gain très positif. Celle d'un *game changer* mal réalisé a souvent une influence neutre (le produit reste utilisable) ;

Show Stoppers. Les fonctionnalités qui sont absolument nécessaires mais qui ne brilleront pas par leur présence. La présence d'un *show stopper* bien réalisé entraîne un gain neutre ou faible. La présence d'un *show stopper* mal réalisé (ou son absence totale) a une influence très négative (le produit n'est plus utilisable).

Sous cet éclairage, un bon MVP doit avoir :

1. des *game changers* pertinents, impressionnants et innovants : on veut maximiser l'expérience utilisateur, et donner une identité forte à l'application ;
2. des *show stoppers* fiables : on veut à tout prix éviter qu'ils entachent, voire annihilent, l'expérience utilisateur.

Ainsi, avoir un défaut dans une fonctionnalité de type *game changer* porte évidement préjudice à la qualité globale du produit. Mais avoir un défaut dans une fonctionnalité de type *show stopper* peut être bien pire, car l'utilisateur ne peut plus utiliser les fonctionnalités de base du produit pour lesquelles il utilise le produit en premier lieu. Par exemple, dans une application de messagerie instantanée, si l'aperçu des liens envoyés dans la discussion (*game changer*) ne fonctionne pas, l'impact négatif est bien moins important que si les fuseaux horaires sont mal gérés et les messages mal ordonnés, ce qui rendrait le produit presque inutilisable. Il est donc plus important de posséder des *show stoppers* qui fonctionnent bien, que des *game changers* qui impressionnent. Les *game changers* sont forcément construits par dessus de bons *show stoppers*. L'inverse n'est pas vrai car il en résulterait dans la plupart des cas un produit inutile incapable d'assurer les fonctions de base qui sont sa raison d'être.

Cependant, un bon MVP doit par nature avoir un coût faible et maîtrisé. Une grosse partie de son coût est directement liée au temps passé par les développeurs. L'expérience utilisateur des projets étant réalisée « sur-mesure » et interagissant de manière directe avec des mécanismes très humains, créatifs, complexes et non formels, il est très difficile voire impossible de nos jours d'automatiser la réalisation des *game changers* pour gagner en temps et en qualité. En revanche, la réalisation technique de l'application peut être automatisée, ou partiellement automatisée, de manière à prendre en charge une grande partie de sa complexité. L'automatisation est moins problématique ici car elle vise des processus qui sont plus facilement formalisables, et dont une grande partie peut être vue comme des *show stoppers* et donc dépend moins de l'inventivité d'un humain.

Q3 : Comment permettre aux développeurs de réduire ou d'éviter les tâches répétitives (qui sont source d'erreur) pour qu'ils puissent passer plus de temps sur les tâches qui portent de la valeur ?

1.3 Contributions

Pour poursuivre la vision du projet GestionAIR et améliorer le processus de développement de services web dans le contexte décrit précédemment, nous

proposons une approche outillée. Cette approche se base sur l'Ingénierie Dirigée par les Modèles (IDM) : nous posons l'hypothèse qu'en trouvant le bon niveau d'abstraction pour modéliser des services web, il est possible de concevoir des processus de dévelopement et de construire des outils qui participent à résoudre les problématiques énoncées dans les sections antérieures.

Nos contributions répondent au problème à travers deux axes :

La modélisation des services web. Nous introduisons un méta-modèle pour les services web, des règles pour vérifier la cohérence des modèles correspondants et une manière d'intégrer ces modèles dans des modèles OpenAPI [35], un standard *de facto* de l'industrie.

Le support des développeurs. Nous présentons et évaluons un processus de développement et un outil, SWSG qui appliquent cette approche dans un contexte technologique proche de celui de Startup Palace et de nombreuses entreprises.

Publications. Nos publications présentent cette approche, à chaque fois à travers les deux axes, mais avancent de manière itérative pour converger vers cette thèse :

— dans [41], nous introduisons un méta-modèle pour les services web et un outil permettant la génération d'implémentation de services web pour Java EE ;
— dans [50], ce méta-modèle est remanié pour mieux correspondre au contexte de Startup Palace, permettre une forme superficielle de vérification de cohérence des modèles dont nous définissons les règles et être accompagné par un outil, SWSG. Cet article a été présenté à l'oral une deuxième fois dans une conférence française [51] ;
— dans [49] et [52], nous améliorons le méta-modèle suite à des retours d'expérience et le fusionnons à OpenAPI pour obtenir un processus plus proche des standards de l'industrie ;
— dans [48], nous montrons un résumé de notre approche, qui y est présentée uniquement comme une extension à OpenAPI.

En parallèle de ces publications académiques, notre approche a été discutée à l'occasion de nombreuses conférences industrielles [7]. Elle a été présentée à

7. http://www.doyoubuzz.com/david-sferruzza/cv/jobs/conferences-a-travers-le-monde

la communautée de la conférence *Snowcamp* [8] lors d'une session plénière de quelques minutes. Elle a par la suite été montrée plus en détail à la conférence PHP Tour 2018 [9].

Plan. Le plan de la thèse est articulé autour de quatre parties :

1. **Introduction et état de l'art :**
   — le Chapitre 2 décrit le projet GestionAIR et ses limitations ;
   — le Chapitre 3 fait l'état de l'art des approches existantes pour développer des services web ;
2. **L'ingénierie dirigée par les modèles pour le développement de services web :**
   — le Chapitre 4 introduit le méta-modèle pour les services web et une syntaxe concrète associée ;
   — le Chapitre 5 montre pourquoi et comment nous pouvons vérifier les modèles de services web pour offrir des garanties intéressantes permettant d'améliorer la qualité des projets ;
   — le Chapitre 6 intègre cette théorie dans OpenAPI 3.0, un standard de l'industrie pour la descriptions de services web ;
3. **Outillage et validation :**
   — le Chapitre 7 décrit SWSG, l'outil mettant en œuvre notre approche ;
   — le Chapitre 8 applique l'approche outillée dans des cas d'étude ;
4. **Conclusion :**
   — le Chapitre 9 fait le bilan de la thèse ;
   — le Chapitre 10 en donne les perspectives.

8. https://snowcamp.io
9. https://event.afup.org/phptourmontpellier2018/

Chapitre 2

Analyse de GestionAIR : ancienne plateforme d'ingénierie web

Sommaire

Pour illustrer certains éléments de contexte mentionnés en introduction dans le Chapitre 1, ce chapitre décrit la méthode de travail de Escale Digitale pour le développement d'applications web ainsi que GestionAIR, une plateforme d'ingénierie web conçue et développée par Startup Palace. Entre 2010 et 2014, plusieurs versions de cette plateforme ont été successivement utilisées en production et ont permis de construire des applications web pour des clients.

Ce chapitre se base sur la dernière version de GestionAIR, dont la démarche est à l'origine de cette thèse. Nous en montrons le principe et le fonctionnement, et faisons le bilan de ses forces et faiblesses.

2.1 Processus de réalisation d'applications web

Cette section décrit l'organisation de la production d'application web telle qu'elle était chez Escale Digitale en 2015.

Le processus illustré par la figure 2.1 s'arrête lorsque l'application web a été réalisée, testée et peut être livrée. D'autres activités de Escale Digitale liées à ce type de projets, comme par exemple le référencement, le webmarketing et l'hébergement, ne sont pas dans la portée de ce processus.

La maintenance des applications ne sera pas évoquée comme une étape du processus. Elle peut être vue comme des itérations supplémentaires sur les dernières étapes. Les évolutions et ajouts de fonctionnalités peuvent être modélisés par un nouveau projet et donc un re-parcours du processus depuis le début.

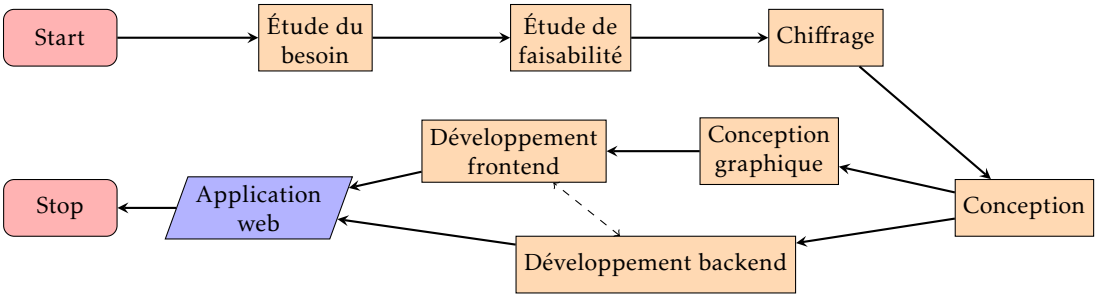

Figure 2.1 – Processus de réalisation d'applications web chez Escale Digitale

Flexibilité du processus Il est intéressant de remarquer que les différentes phases et les différents acteurs décrits ici n'intervenaient pas forcément dans tous les projets. Escale Digitale était capable de mettre en œuvre son expertise dans une grande partie des domaines liés aux projets web, mais ce n'était pas toujours nécessaire ou voulu : cela dépendait des besoins et du contexte de chaque projet. Ici, nous décrivons le processus complet de réalisation d'une application web, dans le cas le plus courant.

2.1.1 Étape 1 : Étude du besoin

Escale Digitale ne faisait pas de prospection active, c'était les clients qui initiaient les projets, bien souvent en demandant une rencontre ou un entretien téléphonique avec un chef de projet.

Le prospect vennait donc à la rencontre d'Escale Digitale avec un problème et une vision plus ou moins précise de la manière dont il aimerait le résoudre. La première mission d'Escale Digitale était donc d'identifier et de comprendre rapidement :

— le contexte du projet (temporalité, budget, position concurrentielle, ...) ;
— le métier du client ;
— le contexte du client et des tiers qui pouvaient être concernés ;
— les problèmes à résoudre ;
— les éventuels éléments connexes au projet déjà existants chez le prospect (solutions, méthodes, valeurs, identité graphique, ...).

2.1.2 Étape 2 : Étude de faisabilité

Fort de sa compréhension sur le projet et son contexte, le chef de projet allait, avec l'aide de l'équipe de production, déterminer si Escale Digitale était capable d'adresser tout ou partie du problème.

Cette capacité concernait plusieurs aspects, notamment :

— la capacité à apporter une solution de qualité suffisante en tenant compte des contraintes notamment budgétaires ;
— la rentabilité par rapport au temps à passer, potentiellement réduit par la réutilisation d'un savoir faire ou d'un composant maîtrisé par l'équipe ;
— la disponibilité des différentes personnes de l'équipe (planning de production, compatibilité avec le timing du projet) ;

— l'enthousiasme et la motivation pour le projet des différents membres de l'équipe de production (observé sur trois axes : technique, éthique et économique) ;
— l'intérêt stratégique pour Escale Digitale. Par exemple : être reconnu comme acteur web expérimenté dans les secteurs suivants : médias, matériels roulants (tracteurs) ou encore financement participatif, de manière à décrocher d'autres projets plus intéressants dans ces secteurs.

Un des pré-requis pour évaluer la capacité à apporter une solution pertinente était de pré-concevoir cette solution. Cela permettait d'avoir une vision macroscopique de l'architecture de la solution d'un point de vue fonctionnel, mais surtout d'anticiper les points bloquants, difficiles ou incertains qui pouvaient être rencontrés lors de la réalisation. Ces blocages pouvaient être techniques, mais aussi conceptuels, marketing, légaux, ... Ces éléments potentiellement problématiques faisaient l'objet d'un examen plus approfondi, voire de recherches et de prototypages [1], de manière à faire émerger des moyens de les résoudre ou de les contourner.

À ce stade là, il était possible qu'Escale Digitale décide de ne pas répondre au besoin du prospect (le projet s'arrêtait alors).

2.1.3 Étape 3 : Chiffrage

Si le chef de projet concluait qu'Escale Digitale pouvait et avait intérêt à apporter de la valeur au projet, il proposait un devis au prospect. Ce devis reprenait les différents éléments imaginés lors de la pré-conception. Il détaillait les différentes étapes du projet, et notamment, pour chacune d'entre elles, le prix, la période concernée et le livrable qui devait être remis pour les étapes nécessitant une validation ferme de la part du client avant de continuer.

Les personnes qui allaient travailler sur le projet étaient interrogées par le chef projet qui leur demandait de fournir des estimations du temps nécessaire pour réaliser les différentes prestations qui les concernaient. À partir de ces informations, le chef de projet calculait le prix qui allait être facturé au client. Il était possible et courant que des parties du devis soient optionnelles, auquel cas le client avait le choix de les prendre ou pas. Par ailleurs, la gestion de projet était facturée au client, ce qui représente souvent autour de 10-15 % du prix du projet.

À ce stade là, il était possible que le client décide de ne pas travailler avec Escale

1. Notamment à travers l'élaboration de « preuves de concept »

Digitale. Il arrivait aussi que celui-ci décide de lancer des négociations, ou une redéfinition du projet, le plus souvent pour être en cohérence avec son budget (dans le cas de la redéfinition, le processus recommençait à l'étape 1).

2.1.4 Étape 4 : Conception

Si le client décidait de continuer le projet avec Escale Digitale, alors la réalisation du projet pouvait véritablement commencer. Le chef de projet et un développeur allaient travailler ensemble pour faire émerger :

- les spécifications du projet et des différents modules composant la solution ;
- un schéma de base de données dans la plupart des projets ;
- une vision claire de l'architecture de la solution.

En parallèle, le chef de projet et un *UX designer* concevaient l'expérience utilisateur en cohérence avec les contraintes et le contexte du projet. Cela aboutissait à la création de *wireframes*, c'est-à-dire une sorte de *storyboard* semi-interactif décrivant les Interfaces homme-machine (IHM) et racontant comment l'utilisateur les manipule. En fonction du projet, il pouvait y avoir d'autres livrables complémentaires annexés, comme par exemple : une arborescence des vues, une arborescence des fonctionnalités sous forme de carte heuristique, des parcours utilisateurs, des scénarios, un audit ergonomique (évaluation des interfaces existantes), ...

2.1.5 Étape 5 : Conception graphique

Une fois que les *wireframes* avaient été validés par le client, le webdesigner pouvait intervenir. Son travail consistait à créer des maquettes à partir des *wireframes* et de l'identité graphique de l'organisation du client (grâce à sa charte graphique si elle en dispose) notamment. Ces maquettes représentaient les différents écrans de l'IHM, mais offraient un aperçu beaucoup plus proche du rendu final que les *wireframes*.

Pour tendre vers la qualité attendue dans le rendu final, le webdesigner concevait et dessinait l'ensemble des composants de l'IHM, et les assemblait de la manière décrite par les *wireframes*. Il essayait de prendre en compte les contraintes avec lesquelles le développeur frontend allait devoir travailler dans l'étape suivante, de manière à ne pas proposer de maquettes irréalisables (ou trop chronophages ou complexes à réaliser). Ce travail prenait en compte la problématique du

responsive design en se déclinant sur une sélection de tailles d'écran jugées significatives.

Les maquettes, une fois terminées, étaient mises à disposition du client pour que celui-ci les discute et les valide.

Elles allaient ensuite servir de référence pour toutes les productions liées aux IHM. Le webdesigner réalisait également des notices décrivant des détails qui n'étaient pas forcément lisibles sur les maquettes, et donnant des informations les concernant de manière à faire gagner du temps au développeur frontend : les codes des couleurs, les polices des éléments textuels, ...

Le webdesigner devait également préparer les ressources graphiques qui allaient être utilisées directement par le développeur frontend : images, logos, pictogrammes [2], ... Les ressources qui sont matricielles étaient prévues à la fois dans une version pour écrans normaux, et dans une version pour écrans à forte densité de pixels.

2.1.6 Étape 6 : Développement

Cette étape peut être vue comme deux étapes se déroulant plus ou moins en parallèle : le développement frontend et le développement backend ; il a été choisi de les considérer comme une seule étape ici pour décrire le cas le plus générique.

Le développement consistait à réaliser l'application en elle-même, c'est-à-dire en utilisant des technologies réelles et concrètes et non plus des modèles ou représentations abstraites qui sont non-exécutables. Ce travail s'appuyait massivement sur les étapes précédentes : le développement était complexe dans le sens où il fallait travailler avec toutes les contraintes de l'application et toutes les contraintes des environnements d'exécution. Il était donc important d'avoir préparé au maximum le travail en amont, de manière à perdre le moins de temps possible ici.

La plupart des applications web développées par Escale Digitale étaient constituées d'une partie qui s'exécute côté client, et d'une partie qui s'exécute côté serveur. Ce sont des environnements très différents, et ces deux parties de l'application répondent chacune à des ensembles de problèmes différents : les IHM concernent plutôt le client, et la persistance des données plutôt le serveur, par exemple. Ainsi, dans la suite, nous faisons une différence entre le développement frontend (côté client) et backend (côté serveur), car les développeurs associés

2. Unitaires, positionnés dans un *sprite*, ou rassemblés dans une police de caractères personnalisée

mettaient l'emphase sur des compétences différentes, même si la plupart d'entre elles étaient communes aux deux.

Développement frontend Le travail du développeur frontend était de s'approprier les maquettes et notices faites par le webdesigner, et de leur « donner vie » en réalisant une version exécutable des IHM avec des technologies web comme HTML, CSS et JavaScript, ou dérivées. C'était une tâche délicate car les environnements des clients web n'étaient pas toujours uniformes et évoluaient rapidement.

Par ailleurs, pour augmenter la cohérence et la qualité des IHM, le développeur frontend mettait en place un système de composants (potentiellement paramétrables). Il commençait son développement par l'élaboration d'une bibliothèque des composants de l'application, qu'il n'avait ensuite plus qu'à assembler. Cela permettait d'éviter la duplication du code des composants, facilitait la collaboration avec les autres développeurs et assurait la cohérence de l'ensemble. En effet, cette bibliothèque (ou *styleguide*) était la référence de l'utilisation et du rendu des composants pendant toute la durée de vie de l'application (ce styleguide avait beaucoup de points communs avec la documentation de Bootstrap [3], par exemple).

Développement backend Le développeur backend, quant à lui, s'occupait notamment de gérer les données qui allaient abonder les IHM, ou qui allaient en provenir, souvent via des API HTTP. La plupart du temps, ces données étaient écrites dans une base de données relationnelle, mais ce n'était pas l'unique solution et la persistance des données pouvait impliquer de nombreux échanges d'informations avec divers systèmes externes.

Le code écrit par le développeur backend s'exécutant, par définition, sur un serveur non-contrôlé par l'utilisateur, le développeur backend était le garant de la cohérence des données qui allaient transiter par le serveur. Le serveur était vu comme un acteur de confiance, et donc la partie de l'application qui s'y exécutait était critique car elle était la seule en possibilité d'effectuer des modifications sur les données tout en assurant que la logique métier était respectée.

C'est pourquoi, le développeur backend travaillait en étroite collaboration avec le chef de projet, qui connaissait bien cette logique métier. Il fallait être certain que les données passées, présentes et futures de l'application allaient rester

3. Un framework HTML, CSS et JavaScript pour développer efficacement des sites web *responsive* ; http://getbootstrap.com/

cohérentes et exploitables, sinon l'expérience utilisateur pouvait devenir très mauvaise.

Tout comme le développeur frontend, le développeur backend travaillait, dans la mesure du possible, en écrivant des bibliothèques de composants paramétrables qu'il pouvait réutiliser dans l'expression des traitements.

Collaboration La collaboration entre le développeur frontend et le développeur backend était très importante. Si tous les deux n'avaient pas besoin du travail de l'un ou l'autre pour pouvoir faire une partie du leur, la zone et la méthodologie de jonction entre leurs travaux était critique.

Le développeur frontend s'occupant de créer les IHM, et le développeur backend d'y faire transiter des données, l'implémentation finale était bien souvent obtenue soit par plusieurs passages de l'un à l'autre, soit par la mise en place d'une relation « hiérarchique » : le développeur backend s'occupant de la fusion du travail du développeur frontend dans son propre travail, par exemple. Cette organisation était définie par les différents acteurs concernés et par le chef de projet, en fonction du contexte du projet, du niveau et de la nature de l'implication de ces acteurs.

De manière plus détaillée, il était possible de distinguer deux types de technologies : le cas où les données étaient injectées dans les vues par le serveur, et celui où elles l'étaient par le client. Dans le premier cas, le développeur backend avait souvent un rôle de finalisation où il rendait dynamique les templates livrés par le développeur frontend. Dans le second cas, les rôles étaient souvent inversés et le développeur backend fournissait au développeur frontend une documentation d'API HTTP lui permettant d'écrire le code allant chercher les données. Ces modes de fonctionnement étaient tous deux utiles et pertinents suivant le contexte du projet ou de la partie du projet étudiée. Ils pouvaient aussi être utilisés tous les deux au sein d'un même projet.

2.1.7 Qualité

Le contrôle qualité n'est pas mentionné comme une étape à part entière du processus. En effet, chaque acteur était responsable de la qualité de sa contribution au projet.

Le client attendant un certain niveau de qualité (qui avait été convenu), c'était le rôle du chef de projet d'agir comme ultime garant de cette qualité. C'est lui qui allait, si nécessaire, préciser le niveau de qualité attendu, en réaction à des

éléments non conformes qu'il avait pu observer lors de son suivi de l'élaboration des livrables.

L'équipe d'Escale Digitale était peu nombreuse et composée de personnes passionnées et appliquées, cette auto-gestion offrait un fonctionnement satisfaisant à l'époque mais qui commençait à atteindre ses limites. Cependant, ce système était régulièrement évalué et amélioré, notamment dans le cadre d'un processus d'amélioration continue, au travers d'un temps d'échange d'une heure minimum tous les 15 jours, que ce soit par des modifications du processus de production ou par l'introduction de meilleurs outils [4] ; le but était soit d'améliorer la qualité des contributions « en amont », soit de réduire les « dégâts » causés par les éléments non conformes.

2.2 Description de l'approche

Pour appuyer cette démarche, et notamment l'étape du développement backend, Escale Digitale a développé un outil nommé GestionAIR. GestionAIR a été conçu empiriquement dans une démarche finalement assez proche de l'IDM. Cependant, pour rester dans le même esprit qu'alors, nous présentons d'abord l'outil avant d'en extraire le méta-modèle par rétro-ingénierie.

2.2.1 Principe de fonctionnement de l'outil

GestionAIR est une plateforme d'ingénierie web, c'est-à-dire un outil d'aide à la conception et la création d'applications web. Il peut être vu comme l'assemblage de trois composants qui sont : une structure de données représentant l'application produite, une application web d'édition et une application d'exécution.

Structure de données. Cette structure de données est le cœur de GestionAIR. Elle représente et encapsule presque tout ce qui définit l'application web à produire. Elle est structurée de manière à exposer des concepts primitifs que le développeur peut paramétrer et composer pour réaliser son application web.

Il a été choisi de l'implémenter sous forme d'un schéma de base de données relationnelle. Ce schéma peut être vu comme méta-modèle d'application web, et les données contenues dans ses tables en sont le modèle. Il est décrit de manière plus détaillée dans la Section 2.2.2.

4. De développement ou de gestion, par exemple

Application d'édition. L'application d'édition est une application web qui permet de visualiser et de modifier la structure de données qui représente l'application web à produire.

Elle tente également d'adresser des problématiques de productivité et de travail collaboratif. Par exemple, elle contient un éditeur de code en ligne (voir la Figure 2.2) qui permet au développeur de profiter de fonctionnalités intéressantes pour sa productivité dans tous les contextes où il est amené à saisir du code. De plus, cet éditeur intègre un système d'alerte permettant d'éviter l'édition concurrente de la même ressource par plusieurs personnes différentes.

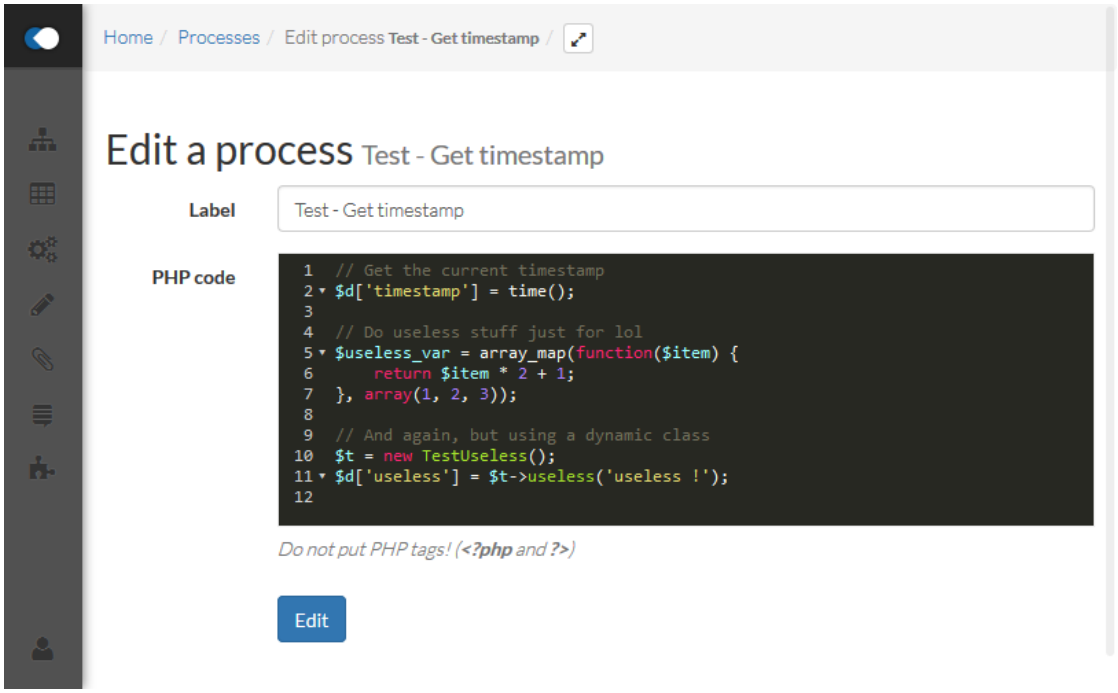

FIGURE 2.2 – L'écran d'édition de process dans GestionAIR

L'application d'édition est implémentée en PHP [60] et utilise notamment les technologies suivantes :

Laravel. Un framework PHP qui aide à réaliser la partie d'une application web qui s'exécute côté serveur [37] ;

AngularJS. Un framework JavaScript qui aide à réaliser la partie d'une application web qui s'exécute côté client [22] ;

Bootstrap. Une bibliothèque CSS, LESS et SASS qui aide à écrire des vues d'application web [5] ;

5. https://getbootstrap.com/

CodeMirror. Une bibliothèque JavaScript qui permet d'insérer des zones d'édition de code paramétrables dans une application web [6] ;

simpleSqlParser. Une bibliothèque JavaScript qui expose un analyseur syntaxique d'un sous-ensemble du langage SQL fonctionnant dans un navigateur web [7].

Application d'exécution. L'application d'exécution est un programme qui va dynamiquement analyser la structure de données produite grâce à l'application d'édition et servir l'application web qui y est décrite.

La Figure 2.3 illustre le fonctionnement de GestionAIR. La structure de données interne n'est manipulée directement par aucun des acteurs. L'application d'exécution va avoir un accès en lecture à cette structure de données. L'application d'édition aussi, mais aura également possibilité de la modifier.

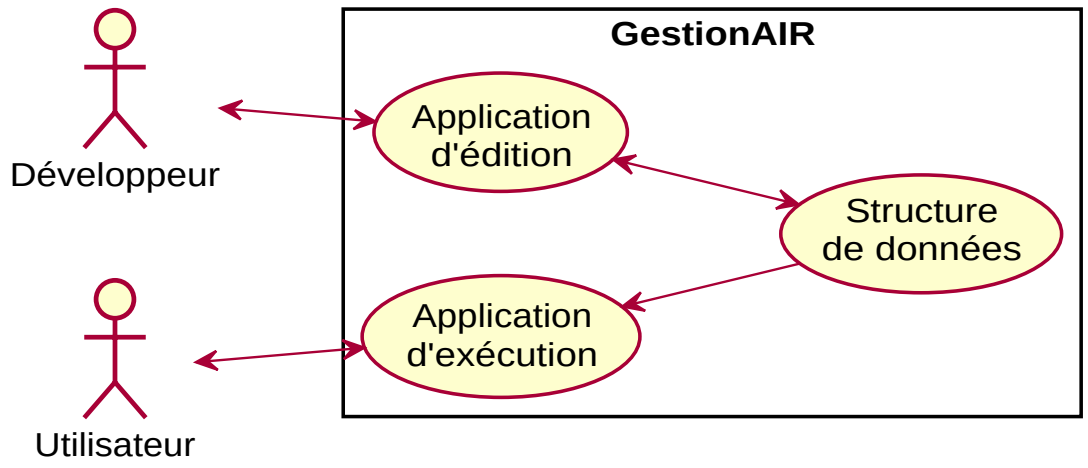

Figure 2.3 – Le fonctionnement de GestionAIR

2.2.2 Méta-modèle

Le schéma SQL de la base de données nécessaire au fonctionnement de GestionAIR est donné par la Figure 2.4.

6. https://codemirror.net/
7. https://github.com/dsferruzza/simpleSqlParser

FIGURE 2.4 – Le méta-modèle d'application web dans GestionAIR

Voici les entités issues du méta-modèle d'application web de GestionAIR. Il est intéressant de se reporter à la Figure 2.4 durant la lecture pour voir la liste exhaustive des propriétés de chaque entité.

Route

La `route` est l'entité centrale du méta-modèle. La figure Figure 2.5 montre l'écran associé aux `routes` dans l'application d'édition.

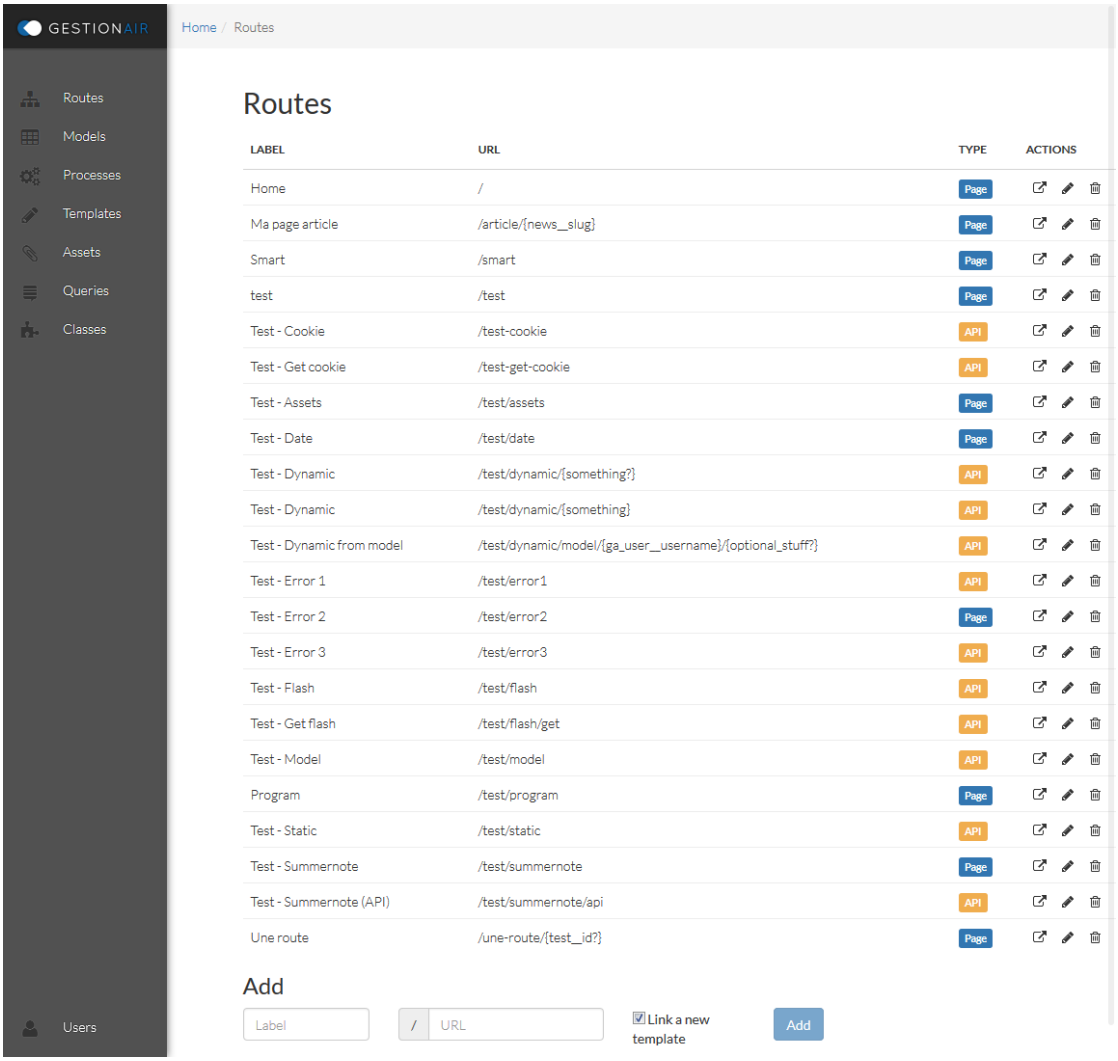

FIGURE 2.5 – L'écran des routes dans GestionAIR

Pour chaque requête HTTP, une application web construite à partir du méta-

modèle parcourra la liste des `routes`, pour en trouver une dont la propriété `url` est un motif pouvant correspondre à l'Uniform Resource Locator (URL) ciblée par la requête. S'il n'y en a pas, l'application renvoie une réponse HTTP `404 Not Found`.

Une fois une route sélectionnée, l'application va exécuter la liste des actions associées à cette `route` dans le but de produire les effets de bord désirés (écrire dans une base de données, par exemple) et de générer une réponse HTTP.

Chaque route contient donc une liste ordonnée de références vers des actions. Une action peut être vue comme un type abstrait ayant quatre types de valeurs concrètes :

— `template`;
— `query`;
— `process`;
— `group`.

Le découpage des traitements en actions exécutées séquentiellement permet la séparation et la réutilisation des sous-traitements, ce qui augmente la maintenabilité. Cependant, ces actions ne sont pas tout à fait indépendantes car elles vont avoir accès en lecture et en écriture à un état (ou contexte) qui sera initialisé à chaque requête HTTP. Les différentes actions liées à une `route` vont donc chacune à leur tour pouvoir utiliser le contexte transmis par l'action précédente, éventuellement le modifier, et le transmettre à l'action suivante. Par exemple, un `process` aura accès à une variable créée par une `query` exécutée précédemment, et pourra lui-même créer de nouvelles variables. Les actions sont donc souvent implicitement couplées entre elles par l'intermédiaire des noms de variables qu'elles lisent et écrivent.

Le dernier état est utilisé pour générer une réponse HTTP. Il y a plusieurs cas possibles :

— une des actions a écrit au moins une erreur dans une variable spéciale : les erreurs sont renvoyées dans une réponse de code `500`;
— une des actions a écrit un objet correspondant à une réponse HTTP dans une variable spéciale : l'objet est utilisé comme réponse ;
— une des actions a écrit une chaîne de caractères dans une variable spéciale : le contenu de cette variable est utilisé dans une réponse de code `200`;
— sinon, l'ensemble des variables du contexte est sérialisé en JavaScript Object Notation (JSON) et renvoyé dans une réponse de code `200`.

La Figure 2.6 montre l'exemple d'une `route` telle qu'elle apparaît dans l'application d'édition.

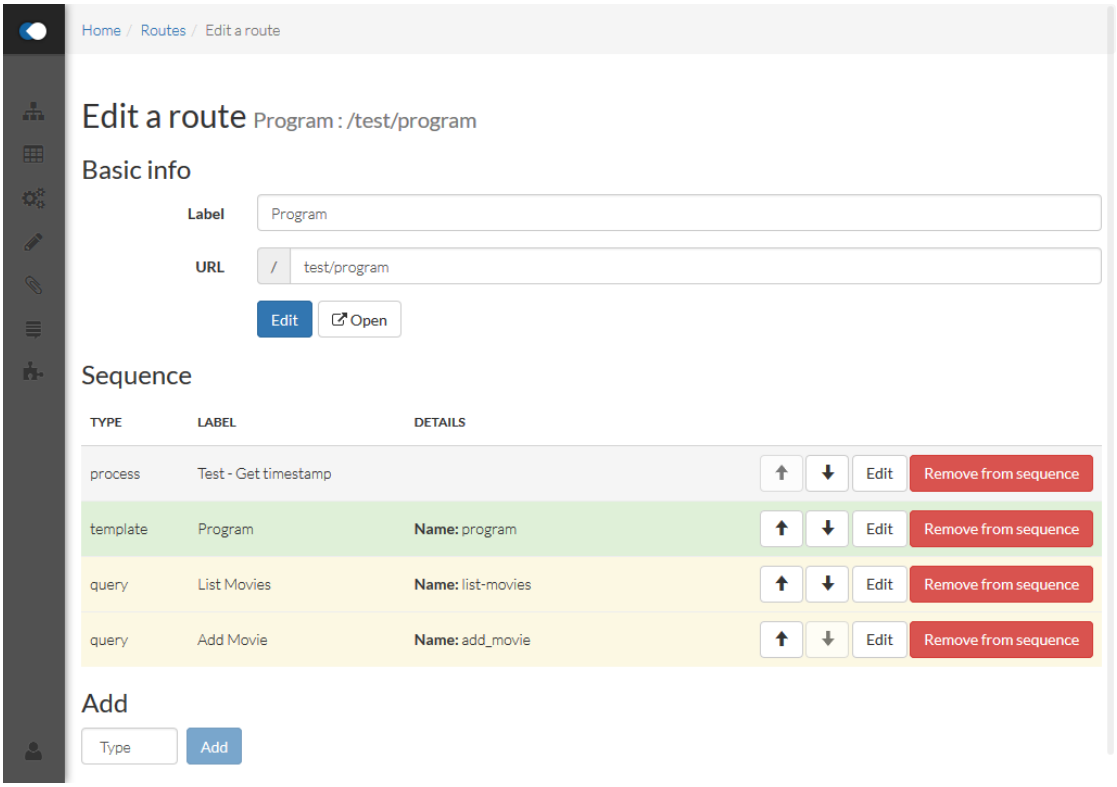

FIGURE 2.6 – Le détail d'une route dans GestionAIR

Template

Un `template` est une action qui va pouvoir utiliser des données brutes présentes dans le contexte pour générer une représentation structurée de ces données. Le plus souvent, il s'agit d'une page HTML. Cependant, il est possible de générer n'importe quel format pouvant être exprimé sous forme d'une chaîne de caractères.

La Figure 2.7 montre l'écran d'édition d'un `template`.

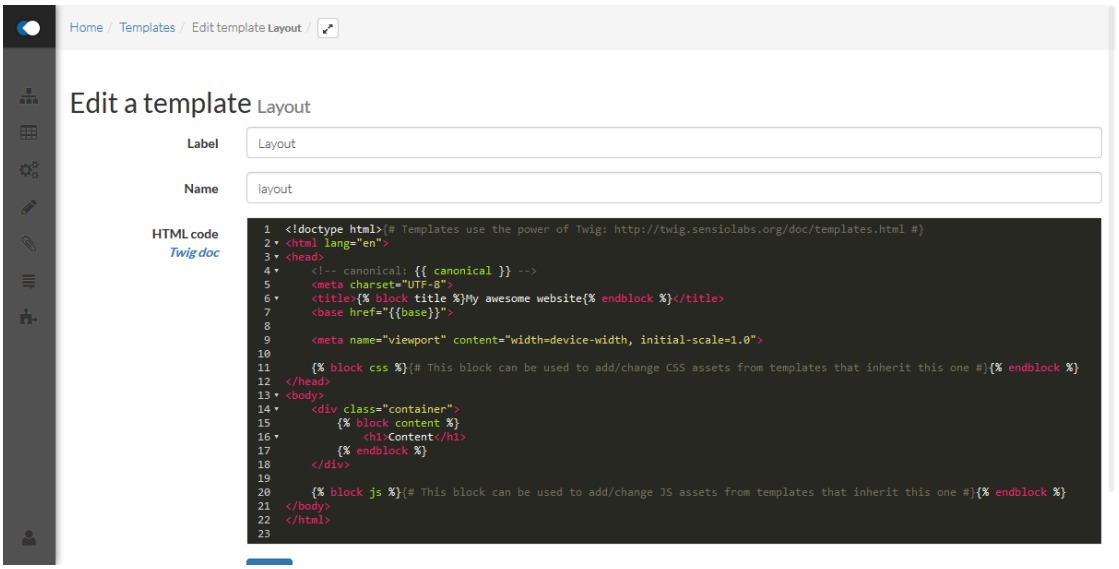

Figure 2.7 – L'écran d'édition de template dans GestionAIR

Query

Une `query` est une action qui va exécuter une requête SQL et stocker son résultat dans le contexte. GestionAIR offre une syntaxe identique à SQL pour exprimer la requête, à la différence qu'il est possible d'injecter des données déjà présentes dans le contexte. Par exemple, il est possible de saisir une requête telle que « `SELECT * FROM table WHERE col = '$col'` » ; l'expression `$col` sera alors remplacée par la valeur de la variable `col` avant d'être transmise au SGBD. Ces données seront alors envoyées comme paramètres de la requête préparée.

La Figure 2.8 montre l'écran d'édition d'une `query`.

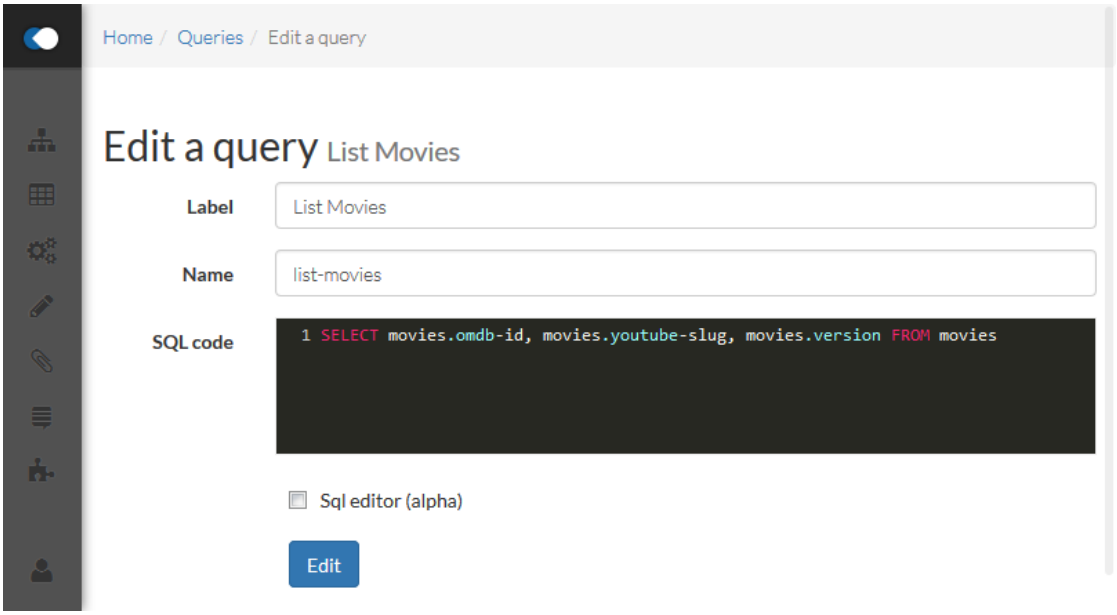

FIGURE 2.8 – L'écran d'édition de query dans GestionAIR

Process

Un `process` est une action qui va exécuter un code PHP arbitraire. Ce code PHP sera en fait encapsulé dans une fonction à laquelle sera passée en paramètre le contexte, et qui retournera automatiquement le nouveau contexte.

Il est possible d'exprimer l'intégralité de n'importe quelle séquence d'action avec un seul `process`. Cependant, cela reviendrait à renoncer aux avantages de maintenabilité apportés par le découpage des traitements en actions distinctes. Les `process` existent pour assurer la flexibilité de l'outil et sont faits pour être utilisés là où les autres actions ne pourraient pas assurer les fonctionnalités désirées.

La Figure 2.9 montre l'écran d'édition d'un `process`.

Group

Un `group` est une action qui contient une liste ordonnée de références vers d'autres actions. L'objectif de cette entité était de pouvoir factoriser des groupes d'actions qui apparaîssaient plusieurs fois dans le même ordre dans une application.

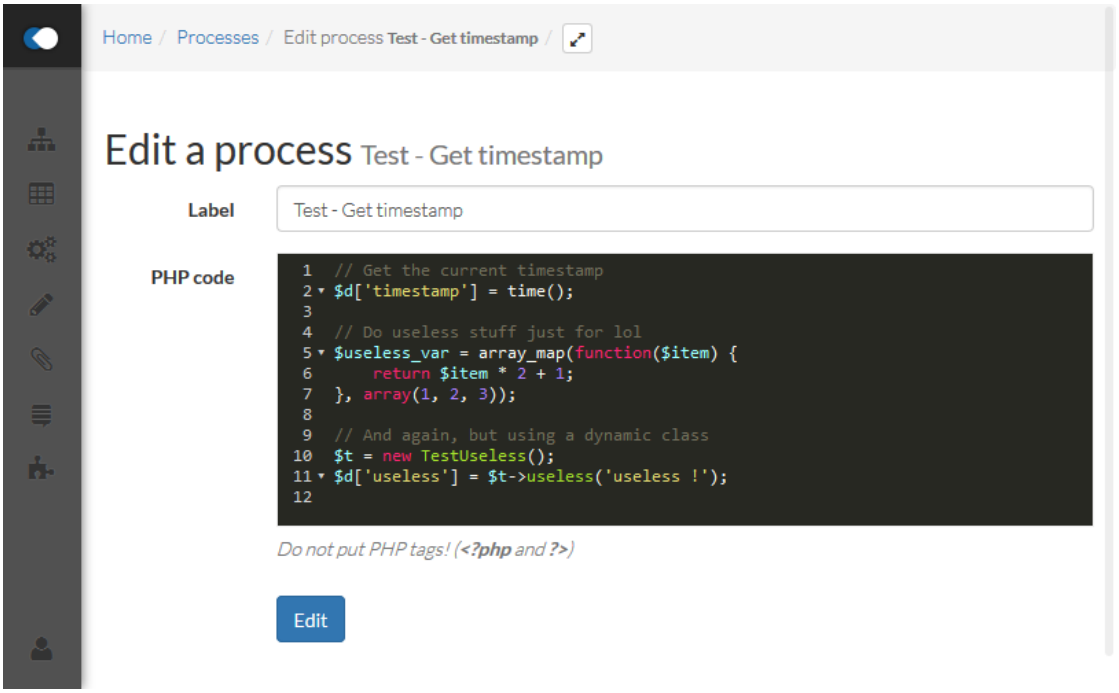

FIGURE 2.9 – L'écran d'édition de process dans GestionAIR

Même si elle est fonctionnelle et implémentée, cette fonctionnalité n'a jamais été utilisée en production. Cela est notamment dû à la difficulté et la faible priorité de concevoir et réaliser des interfaces graphiques ergonomiques pour accompagner cette fonctionnalité dans l'application d'édition, ce qui n'a jamais été fait.

Class

Une `class` est un code PHP arbitraire, qui est identifié par un nom et qui doit contenir uniquement la définition d'une classe PHP.

Il n'y a pas de lien avec les autres entités du méta-modèle : les `class` ne sont pas liées à des `routes`, comme les `process`. Par contre elle sont chargées automatiquement [8] par GestionAIR de manière à ce qu'il soit possible de les utiliser dans l'implémentation de n'importe quel `process`. Cela permet de modulariser le code des `process`.

La Figure 2.10 montre l'écran d'édition d'une `class`.

8. Via un mécanisme d'*autoloader* fourni par PHP.

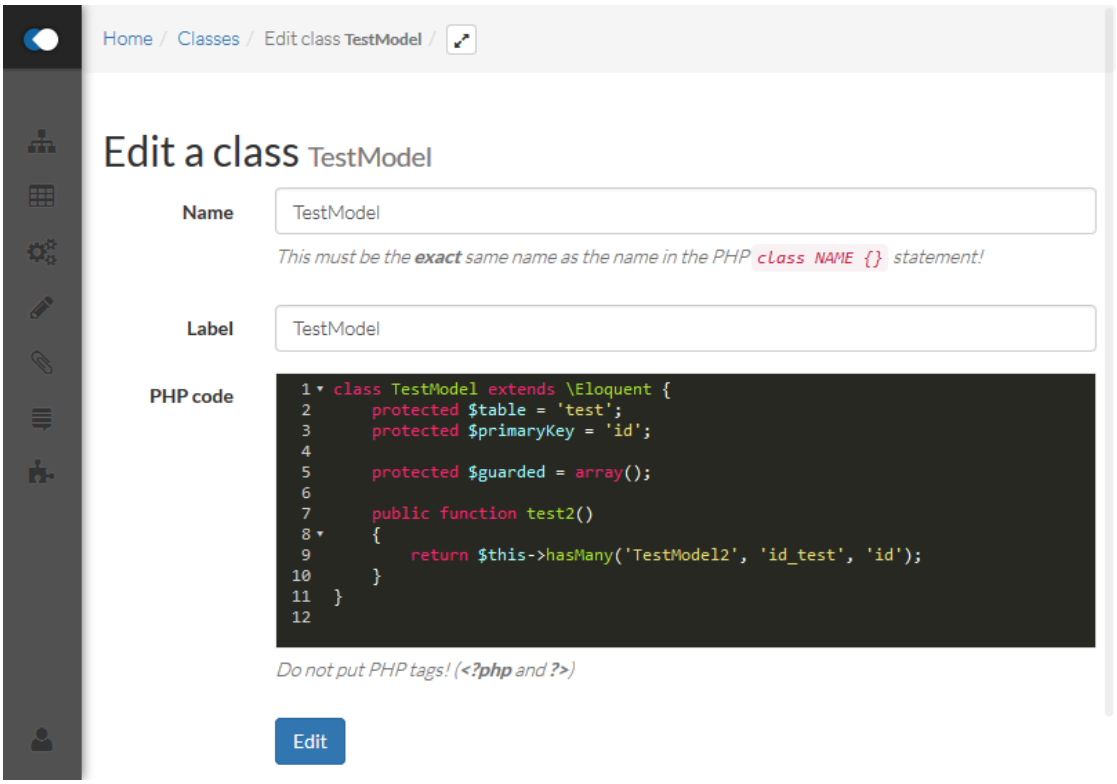

FIGURE 2.10 – L'écran d'édition de class dans GestionAIR

Asset

Les `assets` sont des ressources « statiques » aux formats JavaScript, CSS ou LESS. Inclure ces ressources dans le méta-modèle permet de les rendre éditables via l'application d'édition, ce qui est appréciable dans le cas de ressources personnalisées.

Comme les `class`, les `assets` ne sont pas liés aux autres entités du méta-modèle. Ils sont accessibles depuis une URL prédictible, ce qui permet leur inclusion dans l'implémentation des `templates` HTML. Ce mécanisme n'est pas adapté pour gérer des ressources externes (bibliothèques JavaScript, CSS ou LESS).

La Figure 2.11 montre l'écran d'édition d'un `asset`.

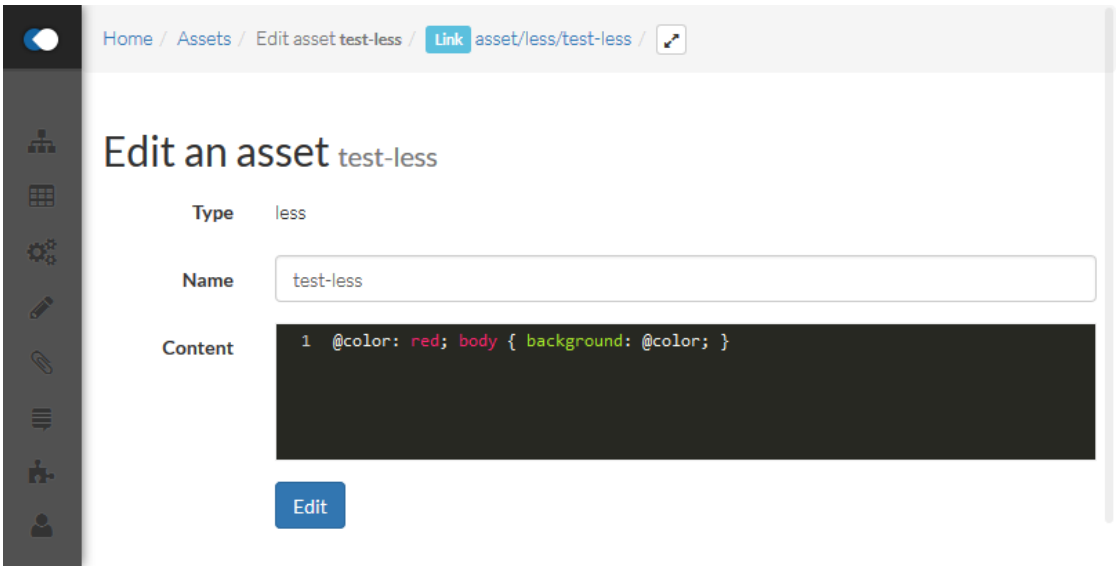

FIGURE 2.11 – L'écran d'édition d'asset dans GestionAIR

2.3 Évaluation et limitations

Cette dernière version de GestionAIR a été utilisée pour réaliser la majorité de la production de Startup Palace pendant un peu plus d'un an. Elle a apporté des gains de productivité et de réactivité constatés par toute l'équipe de Startup Palace et par ses clients.

Cependant, il a été décidé d'abandonner son usage pour les nouveaux projets. Cette décision est justifiée par le constat grandissant que GestionAIR ne don-

nait plus satisfaction, notamment à cause des changements d'environnement suivants :

— les projets confiés à Startup Palace sont de plus en plus complexes, ambitieux et sensibles ;
— les technologies et usages web ont évolué (voir Section 1.2.1) ;
— l'équipe de Startup Palace s'est agrandie.

Les raisons du décalage entre GestionAIR et les nouveaux besoins de Startup Palace sont développés dans les sections suivantes.

2.3.1 L1 : Cohérence des séquences d'actions

Comme évoqué dans la Section 2.2.2, les différentes actions attachées à une `route` vont pouvoir lire et écrire dans un état (ou contexte) propre à chaque requête entrante. Cet état est une structure de données clé-valeur, dont les clés sont des chaînes de caractères et les valeurs sont de type quelconque.

Certaines actions vont dépendre de valeurs spécifiques de l'état pour réaliser leurs traitements. Pour qu'elles s'exécutent correctement, il est donc nécessaire que les actions placées précédemment dans la séquence aient stocké ces valeurs sous les bonnes clés.

Le problème réside dans le fait que ces dépendances sont implicites. Rien ne va permettre au développeur d'exprimer explicitement de quelles valeurs dépend l'action qu'il est en train de créer, ni de détecter s'il y a des cas où ces dépendances ne sont pas satisfaites. C'est au développeur d'avoir en tête les valeurs lues et écrites dans l'état par chacune des actions, et de veiller à la cohérence (c'est-à-dire à la bonne satisfaction des dépendances de toutes les actions) de la `route` sur laquelle il travaille.

Cela devient encore plus complexe à gérer lorsqu'il est nécessaire de modifier une action qui est déjà utilisée dans les traitements de plusieurs `routes`. Le contrat de l'action n'étant pas explicite ni vérifié, il est souvent très facile de le rompre sur une partie des instances de l'action sans s'en rendre compte, ce qui peut occasionner de graves dysfonctionnements.

2.3.2 L2 : Maîtrise du flot d'exécution des séquences d'actions

Dans la situation nominale, les différentes actions liées à une route vont s'exécuter dans l'ordre prédéfini. Cependant, il est possible que l'une d'entre elles

échoue à réaliser son traitement, et qu'on souhaite alors renvoyer une réponse HTTP d'erreur.

Dans GestionAIR, c'est quelque chose qu'il est possible de faire, en lançant une exception ou en retournant un objet représentant une réponse HTTP depuis un `process`. Le problème est que c'est très peu lisible car cela n'apparaît pas au niveau du modèle, mais dans l'implémentation d'une action.

Le contrôle du flot d'exécution de la séquence d'actions d'une `route` en particulier est donc mélangé avec l'implémentation d'une action qui peut être présente dans plusieurs séquences d'actions différentes. En plus de rendre la tâche complexe, cela réduit les possibilités de réutilisation des actions.

La question de rendre le contrôle du flot d'exécution de la séquence d'actions des `routes` explicite (dans le modèle) ou implicite (dans l'implémentation d'un `process`) se pose. La solution actuelle a certes des limitations, mais elle a aussi l'avantage d'être simple.

2.3.3 L3 : Réutilisation des composants

La Section 2.2.2 mentionnait la réutilisation des actions pour les traitements liés à plusieurs `routes` au sein du projet. C'est bien sûr un des intérêts de cette formalisation.

Cependant, il arrive que certaines actions soient suffisamment génériques pour qu'il soit tentant de les réutiliser dans d'autres projets.

Les dépendances des actions et leurs contrats n'étant pas définis explicitement, il est souvent risqué de les copier dans un autre projet car il est difficile d'être sûr qu'elles s'y intégreront bien.

2.3.4 L4 : Intégration de bibliothèques externes

Les applications web développées par Startup Palace dépendent de bibliothèques externes, qu'elles soient libres de droits ou écrites par l'équipe. Malheureusement l'intégration de ces bibliothèques dans une application réalisée avec GestionAIR n'est pas pratique. En effet, il faut que les bibliothèques n'utlisent pas les espaces de noms en PHP ou que ceux-ci soient synchronisés avec l'arborescence des fichiers, ce qui n'est pas toujours facile.

De nos jours, les bonnes pratiques pour gérer ces bibliothèques externes (ou dépendances) [9] sont d'**éviter** les situations suivantes :

— le code des bibliothèques externes est versionné dans le dépôt de l'application, à côté du code de celle-ci ;
— l'application dépend de bibliothèques devant être installées au niveau du système d'exploitation.

À la place, on préfèrera joindre un manifeste de déclaration de dépendances, qui contient des références explicites et exactes vers les bibliothèques nécessaires. Un outil de gestion de dépendances viendra ensuite résoudre, récupérer et installer en isolation dans le projet les différentes dépendances déclarées dans le manifeste.

Malheureusement, GestionAIR n'est pas adapté voire compatible avec de tels outils.

Dépendances côté serveur

Les dépendances côté serveur (des bibliothèques PHP ici, donc) peuvent être importées de deux manières :

— en créant une `class` dans le modèle de GestionAIR pour chaque classe de la bibliothèque, et en faisant un copier-coller du contenu ;
— en copiant l'ensemble de l'arborescence de la bibliothèque dans un dossier spécial de GestionAIR.

Dans les deux cas, c'est très contraignant et ne respecte pas les bonnes pratiques industrielles. La mise à jour des dépendances n'est pas aisée et requiert alors une longue intervention manuelle, et parfois même un accès aux fichiers du serveur de production.

Dépendances côté client

De façon similaire, les dépendances côté client (la plupart du temps des bibliothèques LESS, CSS ou JavaScript) peuvent être importées de deux manières :

— en créant des `assets` dans le modèle de GestionAIR pour chaque fichier de la bibliothèque, et en faisant un copier-coller du contenu ;
— en copiant l'ensemble de l'arborescence de la bibliothèque dans un dossier spécial de GestionAIR, qui sera accessible via une URL prédictible.

9. Voir `https://12factor.net/fr/dependencies`.

Encore une fois, aucun des deux cas n'est vraiment acceptable.

2.3.5 L5 : Compatibilité avec un système de versionnement

GestionAIR n'est pas compatible avec les systèmes de gestion de version, comme Git [10]. La principale raison est que le modèle de l'application web est persisté en base de données et édité en place, comme vu dans la Section 2.2.1.

Par le passé, cela avait été (probablement à tort) pensé comme une fonctionnalité. Cela permettait d'être très réactif sur les évolutions ou corrections de l'application, car il suffisait de modifier le modèle pour que cela se répercute sur l'application en production. À cette époque, les projets étaient beaucoup moins complexes et étaient souvent réalisés par une seule personne.

Ce n'est plus le cas aujourd'hui, et il est quasiment impossible pour Startup Palace d'offrir un bon niveau de qualité en se passant d'outils de gestion de version. Par exemple, il n'est pas possible pour chaque développeur de travailler sur une nouvelle fonctionnalité en isolation des autres changements, et de demander aux autres développeurs de procéder à une revue de son code avant qu'il ne soit fusionné avec la branche maîtresse du projet.

De plus, les systèmes de gestion de version bien utilisés permettent d'avoir un processus de construction d'application reproductible. Cela garantit que, pour une version du code source donnée, le résultat de la construction de l'application sera identique peu importe sur quelle machine ou à quelle date on opère la construction.

C'est une propriété très intéressante car elle facilite grandement le déploiement de l'application vers plusieurs environnements (test, pré-production, production, ...) ou plusieurs instances [11].

En conclusion, il y a des limitations en terme d'intégration dans des environnements de développement plus généraux.

10. https://git-scm.com/
11. Si l'application doit fonctionner sur plusieurs serveurs en parallèle, et qu'on veut pouvoir en ajouter ou en supprimer dynamiquement, la construction de l'application doit être reproductible.

2.3.6 L6 : Intégration avancée des composants côté client

Il est courant d'avoir le besoin d'étendre le processus de construction d'une application, pour ajouter des tâches quelconques.

Prenons l'exemple d'une application qui a besoin d'afficher une lourde image de fond. Pour réduire le temps d'attente de l'utilisateur, on va vouloir éviter d'avoir à recharger cette image à chaque fois qu'on change de page. Pour cela, on va configurer l'application web pour qu'elle indique au navigateur du client de garder l'image en cache très longtemps. Cependant, si on décide de modifier cette image, le navigateur du client continuera d'afficher l'ancienne qui est présente dans son cache.

Pour régler ce problème, on décide de faire apparaître dans le nom de l'image une somme de contrôle calculée à partir de son contenu. Ainsi, si on modifie l'image, on recalcule la somme de contrôle, renomme l'image, et le navigateur la considère comme nouvelle et la recharge. Mais faire toutes ces opérations à la main peut être long et source d'erreur. En effet, il ne faut pas oublier de modifier le nom de l'image partout où elle est référencée à chaque fois qu'on la modifie.

La solution est donc stocker avec les sources de l'application une image nommée normalement, et ajouter une étape dans la construction de l'application qui calcule la somme de contrôle de l'image, la nomme correctement et résout les différentes références vers celle-ci.

Malheureusement, GestionAIR n'a pas été pensé pour supporter ce genre de fonctionnalités. Au fil des projets, des techniques anti-productives ont émergé pour palier ce manque. Par exemple, lorsqu'il est nécessaire, pour des raisons de performances, de servir un fichier JavaScript contenant l'ensemble de l'application, la bonne pratique serait d'écrire l'application en plusieurs modules, et d'avoir une tâche de construction qui se charge de concaténer tous les fichiers en un. Comme ce n'est pas possible, il faut alors écrire toute l'application dans un seul fichier, ce qui pose des problèmes de maintenabilité.

Cela induit des difficultés à prendre en compte les évolutions des spécifications et de l'environnement, du côté client.

2.3.7 L7 : Aide à la gestion des services externes

Une application web est souvent amenée à interagir avec des services externes ; le cas le plus courant étant une base de données.

Ces services sont souvent liés à l'application web par une URL et des informations d'authentification. Les pratiques modernes [12] recommandent que ces liens soient configurables. Par exemple, il doit être possible de changer de base de données sans modifier le code de l'application.

De plus, certains services requièrent une phase de configuration avant de pouvoir être utilisés par l'application web. C'est le cas des bases de données relationnelles, qui ont besoin d'un schéma pour pouvoir stocker des données. Pour ces dernières, il est même crucial de versionner la configuration, c'est-à-dire le schéma. En effet, celui-ci risque d'évoluer alors que l'application est en production, et il faut donc que l'application web soit capable de configurer n'importe quelle base de données pour qu'elle utilise le dernier schéma, sans provoquer de pertes de données.

GestionAIR n'adresse pas cette problématique. Il a été décidé que le développeur avait l'entière responsabilité de la gestion du schéma.

2.3.8 L8 : Passage à l'échelle de l'éditeur

L'application web d'édition de GestionAIR permet de construire l'application web. Elle comporte des écrans qui permettent de visualiser sous forme de listes les différentes parties du modèle de l'application qui sont éditées (voir la Figure 2.5 pour la liste des `routes`).

Cependant, cette visualisation n'est pas adaptée pour les projets complexes, où il est courant d'avoir beaucoup d'éléments. Les listes deviennent rapidement très longues [13] et il est fastidieux de chercher un élément car elles ne sont pas filtrables ni triables facilement.

Ce problème de non-passage à l'échelle des interfaces graphiques face à la complexité du modèle ne concerne pas que les listes d'éléments. L'ensemble de l'application d'édition devient laborieuse à utiliser, ce qui annule les gains de productivité qu'elle pouvait apporter sur les projets plus petits.

Il s'agit d'un problème d'ergonomie et de passage à l'échelle de l'outil, lié à l'architecture du méta-modèle.

12. Voir https://12factor.net/fr/backing-services.
13. Il n'est pas rare qu'on puisse les faire défiler sur l'équivalent d'une quinzaine de hauteurs d'écran.

2.4 Conclusion

Nous avons décrit le processus et l'outil, GestionAIR, utilisés par Escale Digitale pour réaliser des applications web. Cette approche a été pertinente pendant plusieurs années, offrant des qualités intéressantes pour Escale Digitale en terme d'avantage concurrentiel. Malheureusement, avec l'augmentation de la complexité des projets et l'agrandissement de l'équipe de développement, nous avons identifié des limitations bloquantes qui ont abouti à l'abandon de GestionAIR. Une partie des enjeux de cette thèse est d'aller au-delà de ces limitations en proposant une méthode et des outils de développement d'application web qui conviennent au contexte de Startup Palace.

Chapitre 3

État de l'art

Sommaire

En guise d'introduction, les Sections 3.1 et 3.2 tentent de définir la notion de services web et d'expliquer les enjeux de flexibilité et de support en génie logiciel. Ensuite, les Sections 3.3 et 3.4 présentent les réponses que des travaux antérieurs apportent aux questions de recherche traitées par cette thèse.

3.1 Définitions

Dans la Section 1.2, nous avons proposé nos propres définitions pour les concepts d'application web et de services web. Ces définitions sont suffisantes pour comprendre la thèse. Cependant, pour refléter l'état de l'art sur ce point, nous référençons d'autres tentatives de définition dans les paragraphes suivants.

Application web. Comme souligné par un de ses membres dans [25] et par la communauté du site Stack Overflow dans [30], le World Wide Web Consortium (W3C) ne fournit pas de définition officielle et formelle d'une application web. Cependant, il est possible de trouver une définition partielle dans un de ses documents de travail [33] :

> For the purposes of this document, the term "Web application" refers to a Web page (XHTML or a variant thereof + CSS) or collection of Web pages delivered over HTTP which use server-side or client-side processing (e.g. JavaScript) to provide an "application-like" experience within a Web browser. Web applications are distinct from simple Web content in that they include locally executable elements of interactivity and persistent state.

La définition que nous donnons en introduction est proche de celle-ci. Une des différences principales est que la nôtre s'appuie sur le concept de « services web ».

Services web. Comme pour les applications web, il n'y a pas de définition officielle et formelle de ce que sont les services web, mais il est plus simple de trouver des définitions qui font consensus, comme celle-ci :

> A Web service is a service offered by an electronic device to another electronic device, communicating with each other via the World wide web. In a web service, web technology such as the HTTP protocol, originally designed for human-to-machine communication, is utilized for machine-to-machine communication, more specifically for transfering machine readable file formats such as XML and JSON. [1]

Notre définition d'application web dépend de celle de services web ; cela met en évidence que les services web peuvent être vus comme un sous-ensemble des applications web, où les documents transférés via HTTP ne sont pas interactifs et sont dans des formats facilement exploitables par une machine.

Architecture des services web. Dans certains contextes, la notion de services web fait implicitement référence à un sous-ensemble de services web qui respectent une certaine architecture. Par exemple, le protocole SOAP [64] peut

1. Source : https://en.wikipedia.org/wiki/Web_service

être mis en œuvre à travers des services web. Il s'agit d'un protocole qui permet l'échange de messages structurés et qui est compatible avec différentes couches de transport, dont HTTP. L'utilisation avec HTTP impose alors certaines contraintes sur la manière dont les services web sont utilisés et donc doivent être construits. Un autre exemple est REpresentational State Transfer (REST) [19]. Il s'agit plus d'une philosophie que d'un protocole, mais celle-ci va aussi donner lieu à des contraintes sur les services web : les URL des requêtes HTTP doivent cibler des ressources web et utilisent les méthodes HTTP pour indiquer l'opération à effectuer.

Dans cette thèse, nous traitons de la construction de services web dans le cas général. En pratique, dans le contexte de Startup Palace et de bien d'autres acteurs de l'industrie, ces services web s'inspirent de REST dans une certaine mesure et sont parfois qualifés de *RESTful*.

3.2 Flexibilité et support en génie logiciel

Le travail de l'ingénieur est de construire une solution à un problème complexe qui a un contexte et des enjeux particuliers. Les solutions qui sont à la disposition des ingénieurs sont imparfaites et offrent différents compromis entre flexibilité et support. Par « flexibilité » on entend ici la capacité qu'a une démarche, une solution, un système ou un outil à s'adapter à un grand nombre de problèmes. Le « support », quant à lui, indique le degré de qualité avec lequel une démarche, une solution, un système ou un outil aide l'ingénieur à résoudre un problème.

Par exemple, imaginons un instant qu'un ingénieur souhaite accrocher un tableau à un mur en utilisant un clou. Une solution possible est d'utiliser un marteau pour planter le clou. Une autre solution possible est d'utiliser le manche d'un tournevis pour planter le clou. Cette deuxième solution peut offrir une flexibilité supérieure suivant le contexte de l'ingénieur : un tournevis peut également être utilisé pour visser ou dévisser des vis, ce que ne peut pas faire un marteau. Cependant, le marteau offre plus de support : pour le problème du clou, il permet d'appliquer plus de force et réduit les risques de se blesser la main sur la tête de la vis.

En génie logiciel, plus spécifiquement, ce compromis entre flexibilité et support est partout ; voir [62] par exemple. Si on veut créer un programme qui contrôle une chaudière, on peut utiliser un langage généraliste comme C ; on aurait beaucoup de flexibilité, mais peu de support : beaucoup de manières d'introduire des bugs dans le programme, par exemple. On peut aussi utiliser un Domain

Specific Language (DSL), c'est-à-dire un langage spécialement conçu pour (dans notre exemple) paramétrer des chaudières et donc beaucoup moins flexible que C. Si le contexte du problème est compatible avec celui pour lequel le DSL a été créé, on aurait cependant plus de support qu'avec C : par exemple, le DSL serait plus simple à apprendre, ou empêcherait d'écrire des suites instructions qui pourraient endommager la chaudière.

Mais concevoir un DSL ne suffit pas à résoudre tous les problèmes d'ingénierie logicielle, qui ne sont pas seulement technologiques et ont également des composantes humaines, organisationnelles, budgétaires, ... Bien souvent, la valeur provient aussi des outils (éditeurs et générateurs) qui sont compatibles avec ce DSL, ainsi que des autres langages vers ou depuis lesquels il est capable de se transformer. Cette approche consistant à créer un langage (ou méta-modèle, ou DSL) adapté à l'écriture de solutions à une certaine catégorie de problèmes et d'outiller ce langage est la démarche de l'IDM [29]. La définition du langage fixe le niveau de flexibilité de la solution, et son bon outillage permet d'augmenter le support.

Ainsi, les technologies présentes dans le domaine de l'ingénierie web sont nécessairement positionnées vis-à-vis de ce compromis entre flexibilité et support. Cette thèse ne se base pas sur des insuffisances au niveau de ces technologies qui seraient bloquantes dans certains contextes, mais plutôt sur le souhait d'améliorer le compromis entre flexibilité et support, notamment dans les contextes plus restreints de la réalisation de services web et de Startup Palace. Les sections suivantes reprennent des concepts et technologies existants et détaillent leurs limitations et les oppportunités d'amélioration qui en découlent.

3.3 Construction de services web

Dans le Chapitre 1 nous avons présenté la problématique de cette thèse comme étant de faciliter le développement de services web dans un contexte similaire à celui de Startup Palace. Il existe des solutions à ce problème, et toutes ont leur compromis entre flexibilité et support.

La Section 3.3.1 commente quelques méthodes de construction de services web. Les Sections 3.3.2 et 3.3.3 commentent d'autres méthodes qui permettent également la construction de services web mais qui sont dans une philosophie proche de l'IDM. Cette séparation peut sembler artificielle et arbitraire : il s'agit dans tous les cas de langages qui permettent de créer des programmes exécutables ; ces langages offrent plus ou moins de flexibilité et sont accompagnés par des

outils qui offrent plus ou moins de support.

3.3.1 Programmation de services web

L'utilisation de langages de programmation pour la construction de services web est de très loin l'approche la plus populaire dans l'industrie. Nous revenons sur les principaux paradigmes et sur quelques outils intéressants.

Langages généralistes. Début 2018, *Stack Overflow* a publié un rapport sur l'industrie du logiciel [56]. Dans les langages de programmation les plus populaires, on retrouve JavaScript, Java, Python, PHP et d'autres, qui sont des langages de programmation généralistes. Parmi bien des usages, ces langages permettent la création de services web, que ce soit nativement ou via l'utilisation de bibliothèques ou de frameworks. Ces solutions sont très nombreuses et se basent sur des environnements d'exécution souvent éprouvés. Les langages de programmation généralistes peuvent être vus comme le point de départ pour les problématiques de cette thèse : des solutions qui fonctionnent mais offrent un compromis qui n'est pas optimal dans notre contexte. Certains de ces langages, comme par exemple *Eiffel* [31], offrent des propriétés intéressantes mais ne sont quasiment pas utilisés dans l'industrie et leur écosystème est de faible ampleur.

Langages fonctionnels. La plupart des langages de programmation fonctionnelle populaires, toujours d'après [56], sont également généralistes. Cependant, certains d'entre eux possèdent des propriétés intéressantes qui peuvent rendre les programmes plus maintenables, ce qui répond au moins partiellement à nos problématiques. En effet, [26] argumente que la transparence référentielle permet de déconnecter la nature des calculs de leur évaluation, ce qui facilite le raisonnement et donc la maintenabilité. De même, les fonctions d'ordre supérieur et l'évaluation paresseuse fournissent la « colle » qui permet de rassembler des solutions à des petits problèmes dans le but de faciliter la résolution de grands problèmes. Ces fonctionnalités étant au cœur des langages, leurs avantages peuvent être exploités voire sublimés par des bibliothèques ou des frameworks de manière à proposer, par exemple, des solutions de construction de services web offrant un compromis flexibilité/support très intéressant. Malheureusement, ces technologies demandent un investissement important en terme d'apprentissage, notamment pour être exploitables pour le prototypage rapide. De plus, d'autres approches comme celles basées sur le *Join Calculus* [20] ajoutent

à ça des propriétés avantageuses pour la programmation concurrente ; ces fonctionnalités ne sont pas primordiales pour le développement de services web et ajoutent de la complexité. Pour ces raisons, les langages de programmation fonctionnelle ne sont pour l'instant pas présents dans le contexte technologique de Startup Palace et ne constituent pas en eux-même une réponse satisfaisante aux problématiques de cette thèse.

Programmation multi-tiers. Une architecture multi-tiers modélise une application répartie comme un empilement de plusieurs couches logicielles. Chaque couche a un rôle clairement défini et communique avec les couches adjacentes d'une manière qui est spécifiée par un modèle d'échange. Les langages de programmation multi-tiers unifient le développement des différentes couches dans un formalisme et un environement d'exécution uniques. Par exemple, HopScript [45], héritier de Hop [44], permet de développer des applications web en écrivant les codes côté client et serveur en un seul programme dont la syntaxe est une version étendue de JavaScript. La programmation multi-tiers est une famille de solution intéressante pour le développement d'applications réparties telles que les applications web car elle permet d'abstraire le caractère réparti. Cependant, cette solution a moins de sens pour le développement de services web uniquement et souffre alors des limitations évoquées dans les paragraphes précédents.

3.3.2 Modélisation des services web pour la documentation

De nombreux standards permettant de décrire des services web dans le but de les documenter ont émergé au fil des années.

WSDL. Web Service Description Language (WSDL) [13] est une grammaire XML qui permet de décrire des services web. La version 1.1 qui date de 2001 [2] était principalement utilisée pour la description de services web utilisant le protocole SOAP. En effet, seules les méthodes HTTP `GET` et `POST` était disponibles, ce qui rendait impossible la description de services adoptant une architecture de type REST [19] ou hybride. La version 2.0 sortie en 2007 corrige ce problème, et est une recommandation officielle du W3C. Cependant, [23] argumente que WSDL n'est pas une bonne solution pour représenter des services web et qu'une solution plus généraliste comme Unified Modeling Language (UML)

2. « WSDL » signifiait alors « Web Service *Definition* Language ».

serait meilleure ; une des raisons avancées en retour d'expérience est que WSDL embarque trop de détails techniques.

WADL. Web Application Description Language (WADL) [24] est également une grammaire XML qui permet de décrire des services web. Elle a été développée par *Sun Microsystems* et soumise au W3C en 2009. WADL est plus adapté que WSDL pour la description de services web *RESTful* car il est plus orienté ressource que WSDL, qui est orienté service.

OpenAPI. Le principal problème des deux standards précédents est le manque d'adoption et d'outils, notamment dans l'industrie. Cependant, il existe d'autres standards proposés et portés par l'industrie, comme par exemple OpenAPI [35] (anciennement Swagger), RESTful API Modeling Language (RAML) [40] ou API Blueprint [5].

Dans cette thèse, nous avons construit notre approche autour de OpenAPI 3.0 (voir Chapitre 6), principalement pour deux raisons.

D'abord, il est plus connu dans l'industrie que la plupart de ses concurrents. Cela implique qu'il possède un écosystème d'outils assez développé qui facilite son utilisation dans de nombreux contextes technologiques.

Ensuite, il est également impliqué dans divers projets et publications académiques. Dans [12], il a été choisi pour sa popularité par rapport à WADL et d'autres standards de l'industrie dans le but de transformer automatiquement des documentations de services web écrites en HTML vers un format exploitable par des machines. De plus, [14] fournit un état de l'art de format de description de services web qui fait ressortir OpenAPI comme le choix le plus prometteur actuellement et l'enrichit avec des annotations. Il s'agit d'une version mise à jour du travail proposé dans [61, §3.2]. [43] montre une approche qui est agnotisque aux formats de description de services, mais utilise OpenAPI dans le corps de l'article. La popularité de OpenAPI est aussi soulignée par son utilisation dans des domaines différents du génie logiciel. Par exemple, [66] décrit un cas où OpenAPI 2.0 est utilisé en combinaison d'autres outils de la communauté des sciences biologiques et met en avant le méchanisme d'extension de OpenAPI 3.0 comme une oppportunité d'améliorer l'approche. Un autre exemple dans le domaine de la télécommunication est présenté dans [39] qui, à travers une section dédiée, élabore sur les compromis impliqués dans l'IDM et argumente qu'ils peuvent être surmontés en augmentant l'investissement dans les outils qui supportent les processus de développement ; nous partageons également cette vision.

Le périmètre de OpenAPI se limite à la description de services web. En effet, un modèle OpenAPI ne contient aucune logique métier ; cette logique métier doit être formalisée dans l'implémentation des services web en utilisant un langage ou procédé indépendant. Dans le Chapitre 6, nous proposons une manière d'exploiter cette opportunité de réconcilier description et implémentation des services web autour de OpenAPI 3.0.

3.3.3 Modélisation des services web pour la génération de code

D'autres solutions utilisent également l'IDM pour décrire des services ou applications web, cette fois-ci avec pour objectif principal d'en générer une implémentation exécutable.

BPEL. Business Process Execution Language (BPEL) (ou WS-BPEL) [34] est un langage de programmation qui permet d'exprimer des processus d'entreprise et la description de leur enchaînement sur de multitudes d'applications réparties. Sa syntaxe est basée sur XML. Même si BPEL permet de créer des services web, il a été conçu avec un objectif plus restreint : permettre des interactions entre les processus de différentes entreprises via des services web. Sa grammaire est expressive pour lui permettre de modéliser des processus complexes, mais cette flexibilité vient avec un coût : celui d'apprendre un langage plus complexe. Une grande partie de cette complexité n'est pas nécessaire pour construire des services web dans notre contexte.

Solutions basées sur EMF. Le projet Eclipse Modeling Framework (EMF) [57] est un framework de modélisation et une infrastructure pour la génération de code qui permet de construire des applications basées sur des modèles de données structurées. Il est très reconnu dans la communauté de l'IDM ; par exemple, toute la partie pratique de [29], un ouvrage d'introduction à l'IDM, repose sur EMF. Il comporte de nombreux outils permettant de concrétiser une démarche d'IDM, en fournissant par exemple des moyens pour construire des éditeurs graphiques, vérifier des modèles, … L'un de ces outils, Xtext [58], est un framework facilitant le développement de langages de programmation et de DSL. Xtext est similaire à Meta Programming System (MPS) [28], qui propose des fonctionnalités analogues hors de l'environnement EMF. Un autre de ces outils, Graphical Modeling Framework (GMF) [59], permet de produire des éditeurs graphiques pour modèles ; il peut être utilisé de pair avec Xtext, comme présenté dans [17].

Bien sûr, les outils de EMF peuvent être mis en œuvre pour construire des services web, mais leur généricité peut être vue comme une complexité inutile lorsqu'on se cantonne uniquement à cet objectif. D'autre part, [11] met en évidence la difficulté à enseigner l'IDM : le manque de documentation des outils, par exemple, est un aspect qui peut bloquer beaucoup de praticiens. Tout comme BPEL, EMF est trop complexe dans notre contexte.

M3D. M3D est un outil d'IDM pour la génération d'applications web introduit dans [8, 9]. Les modèles d'entrée reposent sur quatre couches :

- une couche d'information, sous la forme d'un diagramme de classes UML ;
- une couche de services, basée sur Business Process Model and Notation (BPMN) ;
- une couche de présentation, dans un méta-modèle similaire à ce qui existe dans Spring ou Android ;
- une couche de processus, mettant en œuvre le langage Declare [62].

Les auteurs de M3D s'attaquent à des problématiques proches de la nôtre, comme le prototypage rapide d'applications web ou plus spécifiquement l'alignement entre les modèles et l'implémentation, dans un contexte où les développeurs ont besoin de flexibilité et ne peuvent l'obtenir qu'en modifiant directement l'implémentation générée. Leur solution est également similaire à la nôtre, dans le sens où elle met en œuvre l'IDM pour résoudre des problèmes proches.

Cependant, M3D possède des limitations qui ne lui permettent pas d'être une solution viable dans notre contexte.

D'abord, les choix de se baser sur quatre méta-modèles différents et de permettre d'exprimer des interfaces utilisateur sont compréhensibles mais augmentent la complexité globale de la solution, notamment du point de vue de développeurs peu ou pas initiés à l'IDM. Cela rejoint les raisons évoquées précédemment pour BPEL et EMF.

Ensuite, l'approche globale se positionne dans un contexte de prototypage rapide uniquement, c'est-à-dire qu'elle pose l'hypothèse que les applications produites servent à stabiliser les spécifications et doivent être redéveloppées par ailleurs avant d'être mises en production. L'accent n'est donc pas mis sur la vérification de cohérence, qui permet d'augmenter la maintenabilité, ni sur la compatibilité avec plusieurs technologies ou standards de l'industrie. Les interfaces utilisateur générées sont basées sur JSF et les serveurs sur J2EE, ce qui ne correspond pas au contexte technologique de Startup Palace. Cependant, le point important est que M3D n'est pas mûr pour une utilisation industrielle en production, hors

prototype rapide.

Enfin, cette distance technologique avec l'industrie est confirmée par notre incapacité à trouver et essayer l'outil en lui-même : ni exécutable, ni code source, ni documentation technique, ni vidéo de démonstration disponible sur Internet à notre connaissance.

JHipster. JHipster [15] est une plateforme pour générer, développer et déployer des applications web ou des microservices, qui repose sur un outil en ligne de commande lui-même basé sur Yeoman. Cet outil permet de définir interactivement des options de configuration et de lancer la génération de code. Les applications ainsi générées mettent en œuvre un serveur Java basé sur Spring Boot[38], une interface graphique basée sur Angular[22] ou sur React[18], ainsi que divers éléments d'architecture facilitant l'opération de microservices. Le développeur est alors libre de modifier le code généré pour obtenir l'application dont il a besoin.

Pour continuer à faire gagner du temps aux développeurs après cette phase de génération initiale, JHipster propose un sous-générateur d'entités. Dans JHipster, les entités peuvent être vues comme des parties d'un modèle de données. Le sous-générateur d'entités permet donc d'étendre le modèle de données de l'application en générant divers artéfacts qui découlent des définitions des entités ajoutées. Ces définitions sont également stockées de manière à gérer l'ajout ou la suppression ultérieurs de champs ou de relations.

Le sous-générateur d'entités peut être remplacé par deux alternatives : JDL-Studio [42] et JHipster UML [2]. Le premier est un éditeur en ligne [3] qui permet de définir les entités de l'application via une syntaxe nommée JHipster Domain Language (JDL) [1] ; le résultat peut être téléchargé dans un fichier et importé dans une application JHipster. Le second fonctionne de manière similaire mais permet l'importation de diagrammes de classes UML en lieu et place de JDL. L'avantage de ces deux approches est de fournir une solution déclarative permettant une édition graphique des entités, qui sont au cœur de l'application.

Bien que relativement mature et issu de l'industrie, JHipster présente quelques limitations qui le rendent inadapté à nos cas d'usage. D'abord, il impose un certain nombre de choix technologiques, comme l'utilisation de Java et Spring Boot pour la partie serveur, qui ne sont pas compatibles avec le contexte technologique de Startup Palace. Ensuite, JDL-Studio et JHipster UML ne semblent pas maintenus très activement, ce qui laisse à penser qu'ils ne sont pas vraiment in-

3. https://start.jhipster.tech/jdl-studio/

tégrés à la démarche proposée par JHipster, mais sont des éléments dispensables et d'un niveau de maturité différent. Enfin, si l'intégration de JDL-Studio et de JHipster UML permet de mettre en œuvre une démarche d'IDM, celle-ci reste en pratique trop limitée dans les cas où les spécifications peuvent varier de manière importante : le code généré doit bien souvent être modifié manuellement, et les éléments de personnalisation qui en découlent peuvent être écrasés par des générations ultérieures.

3.4 Qualité des services web

Nous avons vu en introduction que cette thèse a pour objectif de faciliter la variabilité des spécifications (voir Section 1.2.2). Cet aspect s'illustre avec le scénario suivant, qui, bien que fictif, est très courant :

1. le projet de développer des services web commence ;
2. des développeurs se basent sur des spécifications (version 1) et implémentent des fonctionnalités ;
3. pour une raison quelconque, les spécifications évoluent et une version 2 est écrite ;
4. les développeurs étudient le différentiel entre la version 1 et la version 2 et corrigent les fonctionnalités concernées ;
5. plus tard, on découvre qu'une fonctionnalité pourtant inchangée entre les versions 1 et 2 n'est plus fonctionnelle car elle reposait sur des modules qui ont été modifiés sans que tous les autres modules qui en dépendent (directement ou pas) ne soient à nouveau testés complètement.

Les conséquences peuvent être une indisponibilité du service, ou pire une corruption des données qui sont persistées. Ce genre de défaut peut être découvert par les développeurs du projet, mais il arrive aussi que l'application défectueuse arrive en production et que des utilisateurs se heurtent aux problèmes, ce qu'il faut bien évidement éviter à tout prix. Heureusement il existe des solutions pour éviter ce genre de régressions.

3.4.1 Tests automatisés

La solution la plus répandue dans l'industrie pour éviter les régressions est l'écriture de tests automatisés. Il s'agit d'écrire une série de programmes qui vont

initialiser un contexte, utiliser le programme qu'on souhaite tester pour réaliser des opérations sur ce contexte, et vérifier un certain nombre d'hypothèses. Ces tests peuvent interagir avec différents niveaux du programme à tester : fonction, module, application, ...

Par exemple, PHPUnit [7] est l'outil de référence pour écrire des tests unitaires dans l'écosystème PHP. Le framework Laravel [37] le met en œuvre pour permettre une forme de tests fonctionnels [4].

Bien que très utile, cette approche souffre de plusieurs défauts :

1. le fait que les tests s'exécutent avec un résultat positif ne prouve pas que le programme est conforme à sa spécification : seul un nombre fini de cas spécifiques est vérifié. Si tous les cas intéressants ne sont pas imaginés et écrits par un développeur, alors ils ne seront pas testés ;
2. la quantité de tests à écrire pour avoir une confiance suffisante dans le programme est parfois importante. La problématique de la maintenabilité du programme s'étend également au code de test, qui est fortement couplé avec celui-ci ;
3. les audits techniques de startups qui ont été menés par Startup Palace font ressortir que, dans la plupart des cas, les développeurs ont connaissance de l'intérêt des tests automatisés mais n'en écrivent pas ou très peu. Une des raisons de cet état de fait est que les commanditaires ou gestionnaires des projets ont rarement connaissance de ces enjeux de qualité et voient l'écriture de tests comme une perte de temps [5].

Certaines personnes profilent l'application pendant qu'elle est en train d'être testée pour mesurer la quantité de code utilisé par les tests, calculent le pourcentage de couverture des tests et se fixent pour objectif d'atteindre 100%. Cette approche n'est pas inintéressante, mais une couverture de 100% ne suffit pas à garantir que tous les cas sont testés et donc le défaut n°1 (voir ci-dessus) reste un problème. D'ailleurs, [4] montre qu'il ne semble pas exister de corrélation entre une haute couverture des tests et l'absence de défauts.

Comme pour le programme en lui-même, il est possible d'utiliser certains concepts de génie logiciel pour faciliter l'écriture et la maintenance des tests. Par exemple, un test de propriété peut remplacer plusieurs tests unitaires. Avec un test unitaire, on va définir un état inital, exécuter une fonction et vérifier que l'état final est celui attendu. Avec un test de propriété, on va définir en intention

4. https://laravel.com/docs/5.6/http-tests
5. Ou ils reconnaissent l'intéret de cette pratique mais n'y allouent pas de temps pour autant, ce qui montre en général une compréhension trop superficielle de cette problématique.

un ensemble d'états initiaux, exécuter la fonction et vérifier des propriétés sur l'état final. Par exemple, une fonction qui accepte un nombre entier et renvoie ce nombre au carré doit toujours renvoyer un nombre positif. Lors de l'exécution du test de propriété, le framework de test va générer aléatoirement un nombre fini d'états initiaux et les tester. Les tests de propriété sont intéressants car ils résolvent en partie le défaut n°2 (voir ci-dessus). Mais ils comportent également plusieurs problèmes :

— par nature, ils ne sont pas déterministes et donc ne peuvent remplacer complètement les autres approches (seulement les compléter) ;
— ils nécessitent un framework de test conçu pour ça ;
— ils sont peu connus des développeurs de l'industrie.

Les tests automatisés offrent une solution intéressante pour commencer à assurer la qualité d'un logiciel, mais ils peuvent être complexes à gérer une fois que leur volume devient important et surtout ils peuvent ne pas détecter certains bugs critiques si les développeurs n'écrivent pas les bons tests. C'est une solution qui peut gagner à être complétée par une autre approche.

3.4.2 Systèmes de types modernes

Une autre solution également représentée dans l'industrie est l'utilisation d'un système de types statiques moderne. De tels systèmes font souvent partie intégrante de l'environnement de développement des langages de programmation qui les supportent en étant indissociables du compilateur. Le principe est le suivant : chaque variable ou fonction du programme a un type qui est connu au moment de la compilation (soit parce qu'il est explicite, soit parce qu'il peut être inféré) ; le compilateur va vérifier qu'aucune des expressions du programme ne comporte d'incohérence de type. Par exemple, une fonction de type $A \to B$ ne peut pas être appliquée à une variable de type C. Par système de types *moderne* on désigne ici l'aptitude des types à encoder de la logique métier et pas uniquement les informations nécessaires au stockage d'une entité en mémoire.

Utiliser un langage proposant un système de types suffisamment expressif permet ainsi d'avoir un compilateur qui va pouvoir *prouver* certaines propriétés dans le programme. Par exemple, dans des langages comme Haskell [3] ou Idris [10], on pourra immédiatemment détecter qu'on cherche à additionner des kilomètres et des miles si on a défini que l'addition fait sens uniquement dans le cas où les termes d'entrée sont du même type.

Même si elle n'est pas parfaite, cette approche est intéressante car elle offre un bon compromis entre efforts et résultats. La contrainte pour les développeurs

d'avoir des types cohérents ferme théoriquement la porte à des programmes pourtant corrects, mais elle a l'avantage de les obliger à des réflexions saines. Par ailleurs, les types peuvent permettre une optimisation « gratuite » et automatique des performances du programme dans certaines technologies.

Cependant, de telles technologies ne sont pas présentes actuellement dans le contexte de Startup Palace, qui utilise principalement le langage PHP [60]. PHP possède un système de types rudimentaire et dynamique, ce qui n'est pas compatible avec cette approche dans le cas général. Il existe des projets comme PHPStan [32] qui permet la détection de certaines anomalies via une analyse statique de code, mais ce projet est encore très jeune et sa couverture loin d'être suffisante et au niveau d'un système de types moderne.

3.4.3 Méthodes formelles

Il existe différentes méthodes permettant d'établir la preuve formelle du bon fonctionnement d'un programme. Souvent, il s'agit de créer ou générer un modèle mathématique représentant un programme et de vérifier si ce modèle respecte les propriétés souhaitées. Le typage (exposé dans le paragraphe précédent) est une forme de vérification formelle, mais il en existe d'autres comme le *model checking* ou la *vérification déductive* (également appelée *preuve*).

Ces méthodes permettent d'obtenir d'excellents niveaux de qualité, mais ne passent pas à l'échelle car elles sont également très coûteuses en temps et requièrent des compétences spécifiques. Il est rare dans l'industrie de trouver des développeurs de services web les maitrisant, car leur coût de mise en œuvre les réserve aux systèmes critiques, dont le coût d'échec est extrêmement élevé.

Les développeurs de services peuvent retirer des bénéfices importants de la qualité assurée par les méthodes formelles, mais ne souhaitent pas, pour la plupart d'entre eux, mettre en place ces méthodes qui sont coûteuses en temps à la fois à la formation et à l'utilisation.

3.5 Conclusion

Le Tableau 3.1 récapitule les solutions abordées dans ce chapitre, et les positionne par rapport à plusieurs critères :

Exprime la logique métier. Est-ce que la solution permet d'exprimer la logique métier nécessaire à l'exécution des services web ?

Flexibilité. La liberté qu'offre la solution pour exprimer des cas différents. Voir la Section 3.2.

Facilité de prise en main. La facilité d'appropriation pour des développeurs. Cet indicateur prend notamment en compte la disponibilité (ou l'indisponibilité) de ressources officielles (documentation, vidéos, ...) et officieuses (tutoriels communautaires, sujet abordé en conférences, ...). [6]

Aide à la correction. La disponibilité de mécanismes ou outils permettant d'éviter l'introduction de bugs.

Écosystème d'outils. La richesse et la maturité industrielle de l'écosystème d'outils conçus pour opérer, assister ou compléter la solution.

Documentation. La capacité à fournir ou générer une documentation des services web à destination d'humains et de machines.

	Exprime la logique métier	Flexibilité	Facilité de prise en main	Aide à la correction	Écosystème d'outils	Documentation
Langages généralistes	●	●●●	●●●	●○○	●●●	○○○
Langages fonctionnels	●	●●●	●●○	●●○	●●○	●○○
Langages multi-tiers	●	●●●	●○○	●●○	●○○	●○○
WSDL	○	●○○	●○○	●●○	●○○	●●○
WADL	○	●○○	●○○	●○○	●○○	●●○
OpenAPI	○	●○○	●●●	●○○	●●○	●●●
Famille BPEL	●	●●○	●○○	●●○	●○○	●●○
Famille EMF	●	●●○	●○○	●●○	●○○	●●○
M3D	●	●●○	○○○	●○○	○○○	●●○
JHipster	●	●●●	●●○	●○○	●●●	●●○

● = 1 point ; ○ = 0 point ; ●●○ = 2 points sur 3

TABLE 3.1 – Comparatif des familles de solutions existantes pour réaliser des services web

Ce qui manque au contexte industriel ce sont des techniques assurant une construction aisée et flexible, un bon fonctionnement et une facilité de mainte-

6. Ainsi que de la solution en elle-même (dans le cas de M3D).

nance des services web, tout en étant abordables par des développeurs généralistes (non spécialistes d'approches pointues).

Les approches basées sur EMF offrent un compromis intéressant qui va dans ce sens. Cependant, n'étant pas spécialisées dans la réalisation de services web, elles offrent plus de flexilité que nécessaire, ce qui se traduit souvent par un amoindrissement du support et de la facilité de prise en main par les non spécialistes.

Deuxième partie

L'ingénierie dirigée par les modèles pour le développement de services web

Chapitre 4

Services web : méta-modèle et évaluation

Sommaire

Pour aller plus loin que les limitations vues dans les Chapitres 2 et 3 et produire une solution aux problèmes évoqués dans le Chapitre 1, nous avons conçu un méta-modèle pour la construction de services web.

Contrairement à d'autres solutions existantes, ce méta-modèle se veut volontairement simple, voire minimaliste. Il a été créé comme un moyen de poursuivre deux objectifs :

1. fournir aux développeurs de bonnes abstractions pour qu'ils puissent écrire du code ré-utilisable de manière flexible (développé dans les Sections 4.1 et 4.2) ;

2. permettre la création d'outils exploitant les modèles pour offrir du support aux développeurs, dans le but notamment de leur offrir des garanties dès l'étape de conception (illustré par les Sections 4.3 et 4.4 et les chapitres suivants).

4.1 Notations

Nous introduisons plusieurs éléments de notations nécessaires dans les sections suivantes.

Union. L'union (ou la somme) de deux types T_1 et T_2 est écrite $T_1 \uplus T_2$.

N-uplet. Un n-uplet (ou tuple) T est un type produit entre n types T_1 à T_n, avec $n \geq 2$. Il s'écrit $T \equiv T_1 \times \ldots \times T_n$. Une valeur t de type T s'écrit $t = (t_1, \ldots, t_n)$ avec $t_1 \in T_1, \ldots, t_n \in T_n$. La notation $t(x)$ est aussi utilisée pour désigner t_x, avec $x \in \{1, \ldots, n\}$.

Enregistrement. Un enregistrement R est un tuple dont les éléments sont associés à une étiquette. Il est noté $R \equiv \langle label_1 : T_1, \ldots, label_n : T_n \rangle$. Il s'agit de sucre syntaxique car un enregistrement est un tuple $T_1 \times \ldots \times T_n$ et n fonctions $label_1 : R \to T_1, \ldots, label_n : R \to T_n$. Comme pour les tuples, une valeur r de type R s'écrit $r = (t_1, \ldots, t_n)$ avec $t_1 \in T_1, \ldots, t_n \in T_n$. Les fonctions associées peuvent aussi s'écrire $r.label_1, \ldots, r.label_n$.

Par exemple, pour un enregistrement défini par $person = (\text{“}Batman\text{”}, 35) \in \langle name : String, age : Int \rangle$, on a $name(person) = person.name = \text{“}Batman\text{”}$.

Ensemble. Le type d'un ensemble dont les éléments sont tous de type T est noté $\mathcal{P}(T)$.

Liste. Une séquence ou liste de type T est un ensemble d'éléments de type T qui sont placés dans un ordre spécifique. Elle s'écrit $List(T)$. La notion est similaire à une fonction des indices I vers les valeurs de T, c'est-à-dire $List(T) : I \to T$ où I est un entier compris dans $\{1, 2, \ldots, N\}$ et $N = card(List(T))$. Elle peut aussi s'écrire $[t_1, \ldots, t_n] \in List(T)$ avec $t_1, \ldots, t_n \in T$.

Projection. La projection d'un ensemble ou d'une liste de tuples S sur le i^e élément de ses tuples s'écrit $Prj_i(S)$. De manière similaire, la projection d'un ensemble ou d'une liste de *records* sur les éléments $label_i$ de ses *records* s'écrit $Prj_{label_i}(S)$.

Par exemple, pour un ensemble de *records* $S \in \mathcal{P}(\langle name : String, age : Int \rangle)$ avec $S = \{(\text{“}Batman\text{”}, 35), (\text{“}Robin\text{”}, 26)\}$, on a $Prj_{name}(S) = \{\text{“}Batman\text{”}, \text{“}Robin\text{”}\} \in \mathcal{P}(String)$ et $Prj_1(S) = \{\text{“}Batman\text{”}, \text{“}Robin\text{”}\} \in \mathcal{P}(String)$.

4.2 Le méta-modèle

Notre méta-modèle de services web est défini par la grammaire en notation Backus-Naur Form (BNF) de la Figure 4.1. La Figure 4.2 montre une version légèrement simplifiée de cette même grammaire sous la forme d'un diagramme de classes UML, pour donner un aperçu rapide.

Ce méta-modèle n'a pas pour objectif de remplacer des méta-modèles standard existants comme OpenAPI [35] ou RAML [40], mais plutôt d'être compatible avec eux (voir Chapitre 6). Ces méta-modèles permettent des descriptions d'interfaces de programmation pour API HTTP qui ne sont pas couplées à un langage de programmation donné. Ces descriptions sont destinées à la fois à des humains et à des programmes, et leur permettent de découvrir et de comprendre comment manipuler les API HTTP décrites.

Notre approche se concentre plutôt sur la description des implémentations de ces API HTTP. Le méta-modèle a été conçu pour permettre une forme de vérification de cohérence des modèles (voir Chapitre 5) et de la génération de code (voir Chapitre 7), tout en essayant de favoriser la facilité d'écriture des modèles. Pour ces raisons et pour être plus accessible aux praticiens, notre approche ne repose pas sur des standards existants comme BPEL [21] ou WSDL [23].

Modèle. Un modèle de services web $m_i \in M$ (voir Équation (4.1)) est spécifié comme un enregistrement comportant trois éléments :

— un ensemble d'entités de type E, qui représente le modèle de données ;
— un ensemble de composants de type C, qui représente le modèle de traitements ;
— une liste de services de type S, qui expose certains composants au monde extérieur.

model	::=	⟨*entities* : entity*, *components* : component*, *services* : service*⟩
identifier	::=	`[A-Za-z][A-Za-z0-9_]*`
entity	::=	⟨*name* : identifier, *attributes* : variable*⟩
term	::=	variable | constant
variable	::=	⟨*name* : string, *type* : type⟩
constant	::=	⟨*type* : type, *value* : object⟩
type	::=	string | boolean | integer | float | date | datetime | entity-ref | seq-of | option-of
entity-ref	::=	⟨*entity* : identifier⟩
seq-of	::=	⟨*seqOf* : type⟩
option-of	::=	⟨*optionOf* : type⟩
component	::=	atomic-component | composite-component
atomic-component	::=	⟨*name* : identifier, *params* : variable*, *pre* : variable*, *add* : variable*, *rem* : variable*⟩
composite-component	::=	⟨*name* : identifier, *params* : variable*, *components* : component-instance*⟩
component-instance	::=	⟨*component* : identifier, *bindings* : binding*, *aliases* : alias*⟩
binding	::=	⟨*param* : variable, *argument* : term⟩
alias	::=	⟨*source* : variable, *target* : variable⟩
service	::=	⟨*method* : method, *path* : path, *params* : service-parameter*, *component* : component-instance⟩
method	::=	`[A-Z]+`
path	::=	`.+`
service-parameter	::=	⟨*location* : parameter-location, *variable* : variable⟩
parameter-location	::=	query | header | path | cookie | body

FIGURE 4.1 – Grammaire BNF du méta-modèle de services web

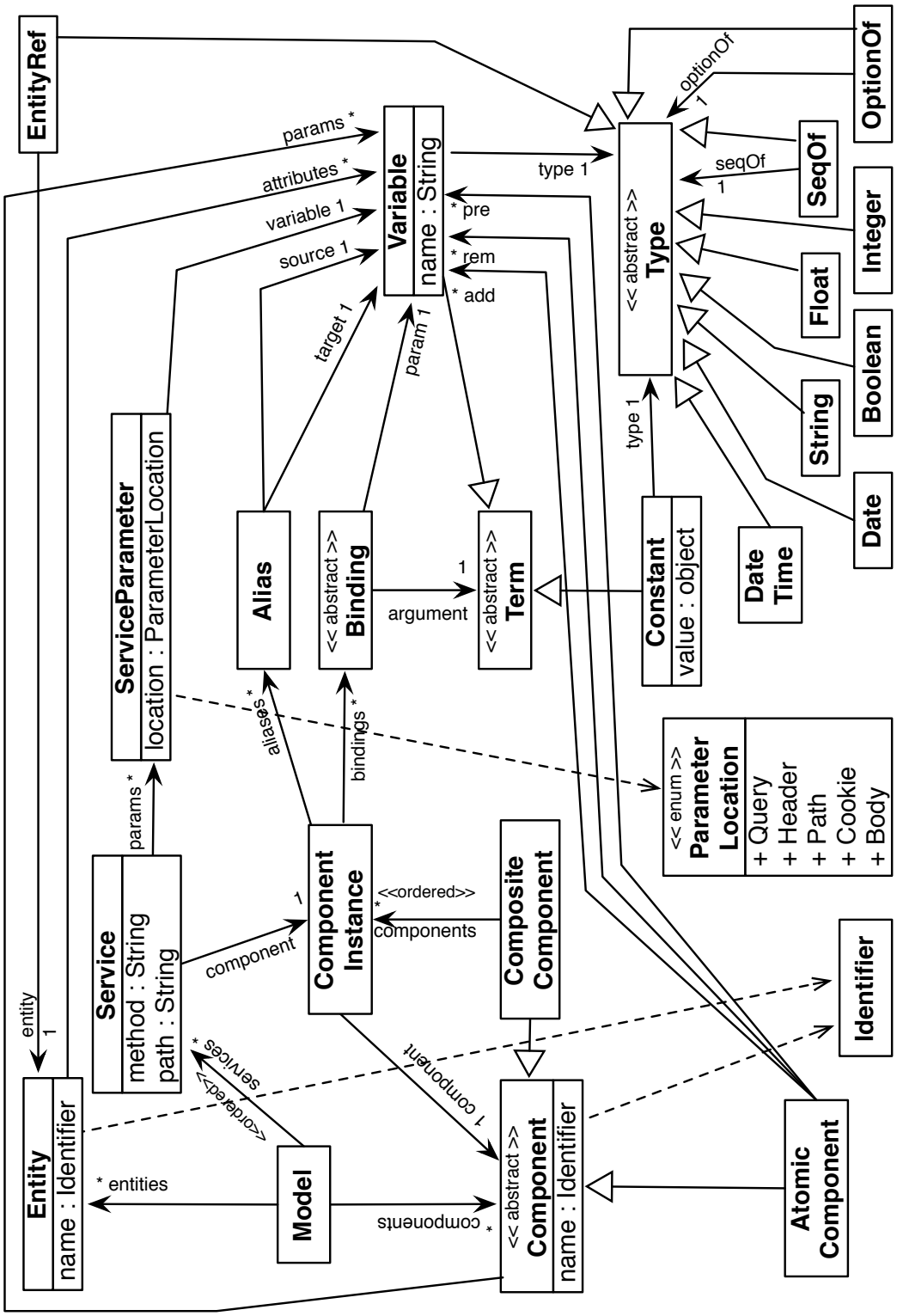

FIGURE 4.2 – Diagramme du méta-modèle de services web

La raison pour laquelle les services sont contenus dans une liste d'éléments *ordonnés* et pas un ensemble (comme les entités et composants) est liée à la stratégie d'évaluation des modèles (voir Section 4.4).

$$M \equiv \langle entities : \mathcal{P}(E), components : \mathcal{P}(C), services : List(S) \rangle \tag{4.1}$$

Les sections suivantes donnent plus de détails sur chacun de ces trois termes et leurs dépendances.

4.2.1 Entités et types

Entité. Une entité est un type de données non primitif, qui représente une structure de données « clé-valeur » (comme le sont les *records*). Une entité $e_i \in E$ (voir Équation (4.2)) est spécifiée comme un enregistrement comportant deux éléments :

- un nom ;
- un ensemble de variables, qui représentent les attributs de la structure de données.

$$E \equiv \langle name : Id, attributes : \mathcal{P}(V) \rangle \tag{4.2}$$

Identifiant. Un identifiant $id_i \in Id$ est un sous-ensemble contraint de chaînes de caractères utilisé pour donner des noms intelligibles à certains éléments d'un modèle. Les contraintes sont :

- le premier caractère doit être une lettre ;
- les autres caractères doivent être une lettre, un chiffre ou le symbole « _ ».

Autrement dit, un identifiant est une chaîne de caractères qui respecte l'expression régulière suivante : `^[A-Za-z][A-Za-z0-9_]*$`.

Variable. Une variable $v_i \in V$ (voir Équation (4.3)) est spécifiée comme un enregistrement comportant deux éléments :

- un nom ;
- un type.

$$V \equiv \langle name : String, type : T \rangle \tag{4.3}$$

Type. Le système de types utilisé par notre méta-modèle comporte des types primitifs (voir Tableau 4.1) et des types paramétriques (voir Tableau 4.2). Les types paramétriques sont représentés par des *records* comportant un élément par paramètre du type.

Type (T)	**Description**
$String$	Une chaîne de caractères.
$Boolean$	Un booléen, comportant deux valeurs : vrai ou faux.
$Integer$	Un nombre entier relatif.
$Float$	Un nombre décimal, à virgule flottante.
$Date$	Une date : année, mois et jour.
$DateTime$	Une date et heure : année, mois, jour, heure, minute et seconde.

Table 4.1 – Types primitifs

Constructeur du type (T)	**Paramètres**	**Description**
$EntityRef$	$\langle entity : Id \rangle$	Une référence vers une autre entité. L'élément `entity` donné en paramètre doit correspondre au nom d'une entité définie dans le modèle.
$SeqOf$	$\langle seqOf : T \rangle$	Une séquence d'éléments d'un type donné.
$OptionOf$	$\langle optionOf : T \rangle$	La version optionnelle d'un type donné, c'est-à-dire comportant tous les éléments du type ainsi qu'une valeur spéciale `null` représentant l'absence de valeur.

Table 4.2 – Types paramétriques

Tous ces types constituent le type générique T, l'ensemble des types valides.

Terme. Un terme peut être soit une variable soit une constante (voir Équation (4.4)).

$$Term \equiv V \uplus Const \tag{4.4}$$

Constante. Une constante représente une valeur littérale, c'est-à-dire précisée directement dans le modèle. Une constante $const_i \in Const$ (voir Équation (4.5)) est une valeur spécifiée par un enregistrement comportant deux élements :

— un type ;
— une valeur littérale.

$$Const \equiv \langle type : T, value : object \rangle \tag{4.5}$$

4.2.2 Composants

Un composant représente un traitement qui s'effectue dans un contexte d'exécution. En plus de son implémentation qui définit le traitement en lui-même, un composant contient des pré-requis sur le contexte d'exécution dans lequel il sera exécuté, ainsi que la description des effets que son exécution produira sur ce contexte. Un composant se comporte comme une fonction : il reçoit un contexte d'exécution en paramètre, et peut retourner un contexte d'exécution altéré. Par ailleurs, un composant est paramétrique c'est-à-dire qu'il peut définir un ensemble de paramètres dont va dépendre son implémentation ; cela permet de créer des composants génériques.

Composant. Un composant peut être soit atomique soit composite (voir Équation (4.6)).

$$C \equiv AC \uplus CC \tag{4.6}$$

Composant atomique. Le composant atomique correspond à la manière la plus flexible de créer un composant, c'est-à-dire en définissant un contrat qui sera complété par une implémentation dans un langage de programmation. Un composant atomique $ac_i \in AC$ (voir Équation (4.7)) est spécifié par un enregistrement comportant cinq éléments :

— un nom ;
— un ensemble de paramètres ;
— un ensemble de variables requises dans le contexte d'exécution ;

— un ensemble de variables qui seront ajoutées dans le contexte d'exécution ;
— un ensemble de variables qui seront retirées du contexte d'exécution.

$$AC \equiv \langle name : Id, params : \mathcal{P}(V), pre : \mathcal{P}(V), add : \mathcal{P}(V), rem : \mathcal{P}(V) \rangle \tag{4.7}$$

Composant composite. Le composant composite est une autre manière de créer un composant, dont le comportement sera constitué par l'exécution séquentielle des comportements de composants existants. Un composant composite $cc_i \in CC$ (voir Équation (4.8)) est spécifié par un enregistrement comportant trois éléments :

— un nom ;
— un ensemble de paramètres ;
— une liste [1] d'instances de composants.

$$CC \equiv \langle name : Id, params : \mathcal{P}(V), components : List(CI) \rangle \tag{4.8}$$

Instance de composant. Une instance de composant est juste une référence vers un composant, accompagnée de deux informations : des arguments pour chacun de ses paramètres et des alias permettant de renommer des variables utilisées dans son contrat et son implémentation, dans ce contexte d'instanciation uniquement. Une instance de composant $ci_i \in CI$ (voir Équation (4.9)) est spécifiée par un enregistrement comportant trois éléments :

— un nom correspondant au composant instancié ;
— un ensemble d'arguments ;
— un ensemble d'alias.

$$CI \equiv \langle component : Id, bindings : \mathcal{P}(B), aliases : \mathcal{P}(A) \rangle \tag{4.9}$$

Argument. Un argument est un paramètre auquel est associé une valeur sous la forme d'un terme, c'est-à-dire soit une constante soit une variable. Si c'est une variable, alors la valeur doit être récupérée en examinant l'argument du même nom que celle-ci passé au composant parent. Un argument $b_i \in B$ (voir Équation (4.10)) est spécifié par un enregistrement comportant deux éléments :

1. Ici encore, il s'agit d'une liste et pas d'un ensemble car les éléments doivent être ordonnés pour permettre une stratégie d'évaluation cohérente ; voir Section 4.4.

— un paramètre, qui doit être identique à un de ceux présents dans la définition du composant instancié ;
— un terme.

$$B \equiv \langle param : V, argument : Term \rangle \tag{4.10}$$

Alias. Un alias permet de renommer une variable définie dans le contrat d'un composant, uniquement pour une instanciation donnée du composant et donc sans modifier sa définition. Un alias $a_i \in A$ (voir Équation (4.11)) est spécifié par un enregistrement comportant deux éléments :

— le nom d'une variable tel que présent dans le contrat d'un composant ;
— un nouveau nom qui sera utilisé dans ce contexte d'instanciation.

$$A \equiv \langle source : Id, target : Id \rangle \tag{4.11}$$

4.2.3 Services

Service. Un service fait le lien entre un ensemble de requêtes HTTP et un composant qui sera chargé de produire une réponse HTTP. Un service $s_i \in S$ (voir Équation (4.12)) est spécifié par un enregistrement comportant quatre éléments :

— la méthode (ou verbe) HTTP des requêtes prises en charge par ce service ;
— un chemin (ou URL) pouvant contenir des paramètres variables, représentant l'ensemble des requêtes HTTP prises en charge par ce service (si leur méthode correspond également) ;
— un ensemble de paramètres à extraire de la requête HTTP et placé dans le contexte d'exécution initial ;
— une instance de composant.

$$S \equiv \langle method : M, path : P, params : \mathcal{P}(SP), component : CI \rangle \tag{4.12}$$

Méthode. Une méthode HTTP $m_i \in M$ est représentée par un sous-ensemble contraint de chaîne de caractères. La seule contrainte est que les caractères doivent être des lettres majuscules, c'est-à-dire que la méthode doit respecter

l'expression régulière suivante : `^[A-Z]+$`. N'importe quelle méthode respectant la RFC 7231 [2] peut être utilisée ici.

Chemin. Le chemin est une chaîne de caractères qui est une URL relative à la racine de l'application. Il est possible de préciser des paramètres en utilisant des accolades pour matérialiser l'emplacement du paramètre dans le chemin et le nommer. Par exemple : `/user/{id}`.

Paramètre de service. Un paramètre de service décrit comment extraire un paramètre d'une requête HTTP. Un paramètre de service $sp_i \in SP$ (voir Équation (4.13)) est spécifié par un enregistrement comportant deux éléments :

— la localisation du paramètre ;
— la variable qui servira à contenir la valeur extraite de la requête HTTP.

$$SP \equiv \langle location : L, variable : V \rangle \tag{4.13}$$

Localisation de paramètre. Une localisation de paramètre décrit dans quelle partie d'une requête HTTP extraire un paramètre de service. Une localisation de paramètre peut prendre cinq valeurs (voir Équation (4.14)) selon l'emplacement du paramètre :

— dans la chaîne de requête [3] ;
— dans la liste des entêtes ;
— dans le chemin ;
— dans la liste des cookies ;
— dans le corps de la requête (dans ce cas la valeur est le corps de la requête) ;

$$L \equiv \{query, header, path, cookie, body\} \tag{4.14}$$

4.3 Syntaxe concrète

Dans la section précédente, nous avons présenté une définition mathématique de notre méta-modèle. Ces notations servent de référence et vont permettre d'écrire des propriétés sur les modèles qui respectent notre méta-modèle (voir

2. https://tools.ietf.org/html/rfc7231
3. Ou « query string » en anglais.

Chapitre 5). Cependant, elles ne sont pas très conviviales à la lecture ou l'écriture de modèles au quotidien.

Pour résoudre ce problème, nous introduisons une syntaxe concrète qui est plus compacte et lisible que l'écriture mathématique. Comme cette syntaxe est équivalente à celle définie dans la Figure 4.1 et est le résultat de choix arbitraires, nous ne l'introduisons pas ici de manière formelle mais nous contentons de la décrire rapidement. L'objectif est de donner de l'intuition sur cette notation qui sera utilisée dans certains exemples.

Intuition sur la syntaxe. La syntaxe concrète représente un modèle comme une liste de définitions. Chaque définition commence par un identifiant correspondant au type de définition, comme montré dans le Tableau 4.3. Ces définitions sont ordonnées mais peuvent être mélangées. Par exemple, on peut définir un composant, puis un service, puis un autre composant.

Identifiant	Élément du modèle
e	Entité
ac	Composant atomique
cc	Composant composite
s	Service

Table 4.3 – Identifiants des définitions dans la syntaxe concrète

Une définition commence donc par un identifiant, seul sur sa ligne. Les différentes propriétés associées suivent, chacune sur une ligne indentée et préfixée par le nom de la propriété. Certaines propriétés définissant une liste ou un ensemble, comme les paramètres d'un service ou les instances de composants d'un composant composite, peuvent être écrites sur plusieurs lignes de manière à définir un seul élément par ligne ce qui rend l'écriture plus lisible.

Le Listing 4.1 montre un exemple de la définition d'un service.

```
1 s
2   method POST
3   path /getName/{email}
4   param path email: String
5   ci GetName
```

Listing 4.1 – Définition d'un service utilisant la syntaxe concrète

4.4 Évaluation d'un modèle de services web

Pour donner du sens à la définition de notre méta-modèle et mieux comprendre sa sémantique, nous donnons ici des règles quant à son évaluation. L'idée est de décrire comment un modèle de services web peut être réduit à une fonction d'une requête HTTP vers une réponse HTTP : $Req \rightarrow Res$.

4.4.1 Méthode d'évaluation

La méthode d'évaluation que nous proposons se déroule en quatre étapes successives : le routage, l'aplatissement, l'évaluation des composants et la réponse.

Routage

Les services web reçoivent une requête HTTP. La liste des services du modèle est parcourue de manière séquentielle jusqu'à trouver un service qui correspond à la requête ; c'est-à-dire dont la méthode HTTP est la même que la requête et le chemin est compatible avec l'URL de la requête. Si aucun service n'est trouvé, une réponse HTTP statique de code `404` est renvoyée.

Aplatissement

L'instance de composant contenue dans le service est aplatie et réduite à une liste d'instances de composants atomiques. Les arguments passés aux différents composants sont résolus pour devenir uniquement des constantes, et les alias sont propagés sur les composants enfants. Puis, les instances de composants composites sont remplacées récursivement par les instances de composants qu'elles contiennent. Une définition de la fonction $flatten(m,c)$ (avec $m \in M$) qui réalise ce remplacement récursif est donnée dans l'Équation (4.15).

$$flatten(m,c) = \begin{cases} [c] & \text{si } c \in AC \\ \bigcup_{ci \in c.components} flatten(m,ci) & \text{si } c \in CC \\ flatten(m,c.component) & \text{si } c \in CI \\ flatten(m,resolve_c(m,c)) & \text{si } c \in Id \end{cases} \tag{4.15}$$

La fonction $resolve_c$ utilisée dans l'Équation (4.15) accepte deux paramètres : un modèle et un nom de composant. Elle renvoie la définition du composant qui porte ce nom dans le modèle. Son type est donc $M \times Id \to C$. Grâce à certaines des règles de cohérence définies dans le Chapitre 5 (voir Équations (5.1), (5.6), (5.7) et (5.9)), cette fonction renvoie un résultat déterministe et est totale si on lui passe un modèle cohérent et un nom de composant présent dans une instance de composant.

Évaluation des composants

Un contexte d'exécution initial est créé en extrayant de la requête HTTP des paramètres éventuellement déclarés dans la définition du service et en les plaçant dans un contexte vide. La liste aplatie de composants est ensuite évaluée : chaque composant atomique reçoit le contexte d'exécution précédent et renvoie un nouveau contexte. Ce comportement est similaire au fonctionnement des monades d'état (voir [65, § 2.5]).

Réponse

Une réponse HTTP est construite. Il y a deux cas à considérer :

1. si un des composants évalués renvoie une réponse HTTP au lieu d'un nouveau contexte, aucun des composants suivants dans la liste aplatie n'est évalué et cette réponse HTTP est renvoyée au client ;
2. sinon, le contexte est sérialisé et encapsulé dans une réponse HTTP de code `200`.

4.4.2 Exemple d'évaluation

Pour illustrer cette méthode, voici un exemple d'une évaluation pas-à-pas d'un modèle.

Le Listing 4.2 montre la syntaxe concrète d'un modèle qui décrit des services web dont le but est de permettre :

1. de s'inscrire à un événement, en fournissant un nom et une adresse email ;
2. de lire la liste des inscrits, si on possède la clé nécessaire.

Dans cet exemple, la clé nécessaire à l'authentification de l'organisateur est écrite directement dans le code. *Il s'agit d'un exemple fictif, il ne faut bien sûr jamais gérer l'authentification de cette manière dans le monde réel* [4] *!*

```
e
  name Registration
  attributes (name: String, email: String, date: DateTime)

s
  method POST
  path /register/{name}/{email}
  param path name: String
  param path email: String
  ci Registration
s
  method GET
  path /attendees
  param query key: String
  ci GetAttendees(apiKey = "mykey")

cc
  name Registration
  ci ValidateEmail
  ci CheckDupRegistration
  ci CreateRegistration
  ci SaveRegistration
  ci RegistrationSerializer
cc
  name GetAttendees
  params (apiKey: String)
  ci CheckKey(correctKey = apiKey)<userKey -> key>
  ci FetchRegistrations
  ci RegistrationsSerializer

ac
  name ValidateEmail
  pre (email: String)
ac
  name CheckDupRegistration
```

4. La bonne pratique est expliquée ici : https://12factor.net/fr/config

```
36    pre (name: String, email: String)
37  ac
38    name CreateRegistration
39    pre (name: String, email: String)
40    add (registration: Registration)
41  ac
42    name SaveRegistration
43    pre (registration: Registration)
44  ac
45    name RegistrationSerializer
46    pre (registration: Registration)
47  ac
48    name CheckKey
49    params (correctKey: String)
50    pre (userKey: String)
51  ac
52    name FetchRegistrations
53    add (registrations: SeqOf(Registration))
54  ac
55    name RegistrationsSerializer
56    pre (registrations: SeqOf(Registration))
```

Listing 4.2 – Exemple complet de modèle

Requête. On envoie une requête HTTP à une implémentation de nos services web accessible à l'adresse `http://127.0.0.1:8000`. Cette requête est montrée dans le Listing 4.3.

```
1  POST /register/batman/batman@wayne-corp.com HTTP/1.1
2  Host: 127.0.0.1:8000
3  User-Agent: curl/7.59.0
4  Accept: */*
```

Listing 4.3 – Requête HTTP d'inscription

Routage. Les deux services définis dans le modèle vont être testés dans l'ordre de leurs définitions pour voir s'ils correspondent à la requête. Le premier service correspond car :

- la requête utilise la méthode `POST` ;
- le chemin de la requête `/register/batman/batman@wayne-corp.com` correspond au chemin du service `/register/{name}/{email}` où :
  - `{name}` vaut `batman` ;
  - `{email}` vaut `/register/batman/batman@wayne-corp.com`.

Le premier service est donc sélectionné et le second n'est pas testé.

Aplatissement. Le service instancie le composant `Registration`. Aucun argument n'est passé à ce composant ni à ses enfants, donc pas besoin de les résoudre pour obtenir des constantes. Le composant `Registration` est ensuite aplati (voir Équation (4.16)).

$$flatten(m, \text{“}Registration\text{”}) = [ValidateEmail, CheckDupRegistration, \\ CreateRegistration, SaveRegistration, RegistrationSerializer] \quad (4.16)$$

Évaluation des composants. Un contexte d'exécution initial est extrait de la requête. Il a la forme décrite par l'Équation (4.17), qui correspond aux paramètres déclarés dans la définition du service.

$$ctx_0 = [(\text{“}name\text{”}, String), (\text{“}email\text{”}, String)] \quad (4.17)$$

L'implémentation du premier composant de la liste est exécutée dans ce contexte. S'il renvoie un nouveau contexte, le second composant sera exécuté dans ce nouveau contexte et ainsi de suite. Si un des composants renvoie une réponse HTTP, alors les composants suivants ne sont pas évalués. Le Tableau 4.4 montre les résultats de l'évaluation, composant par composant.

Réponse. Le composant `RegistrationSerializer` a renvoyé un objet qui représente une réponse. Cette réponse est transmise au client.

4.5 Conclusion

Nous avons introduit un méta-modèle de services web. Les modèles de services web qui en découlent sont composés de trois parties : un modèle de données (les

Composant	Type de sortie	Sortie
`ValidateEmail`	Contexte	$(\text{“}name\text{”}, String)$ $(\text{“}email\text{”}, String)$
`CheckDup Registration`	Contexte	$(\text{“}name\text{”}, String)$ $(\text{“}email\text{”}, String)$
`Create Registration`	Contexte	$(\text{“}name\text{”}, String)$ $(\text{“}email\text{”}, String)$ $(\text{“}registration\text{”}, EntityRef(\text{“}Registration\text{”}))$
`SaveRegistration`	Contexte	$(\text{“}name\text{”}, String)$ $(\text{“}email\text{”}, String)$ $(\text{“}registration\text{”}, EntityRef(\text{“}Registration\text{”}))$
`Registration Serializer`	Réponse	`HTTP/1.1 200 OK` `{"name":"batman","email":"batman@wayne-corp.com"}`

Table 4.4 – Résultats de l'évaluation des composants

entités), un modèle de traitements (les composants) et un modèle de services. Pour faciliter l'écriture de modèles, nous avons également introduit une syntaxe concrète, plus lisible que les notations mathématiques utilisées pour définir le méta-modèle. Enfin, nous avons présenté une forme de sémantique associée au méta-modèle, qui illustre comment les différents attributs d'un modèle peuvent être évalués pour obtenir le comportement des services web décrits par le modèle.

Chapitre 5

Cohérence des modèles de services web

Sommaire

Dans le Chapitre 4, nous avons introduit un méta-modèle de services web en précisant que celui-ci a été conçu non seulement pour structurer syntaxiquement les services mais aussi pour permettre une forme de vérification de cohérence des modèles (relevant de la sémantique). Dans ce chapitre, nous expliquons les enjeux de cette vérification dite « structurelle » et comment elle fonctionne.

Ce chapitre utilise des notations introduites dans la Section 4.1.

5.1 Enjeux de la cohérence structurelle

Dans la Section 3.4, nous avons vu des méthodes qui permettent d'améliorer la qualité des programmes en vérifiant leur cohérence. Ces méthodes comportent des limitations dans notre contexte. Dans cette section, nous présentons le

concept de cohérence structurelle, qui réduit ces limitations et offre un compromis intéressant dans notre contexte.

Pour donner plus de confiance dans la fiabilité des services web produits en utilisant notre méta-modèle et utilisant comme cible d'exécution un langage comme PHP, nous recommandons deux niveaux complémentaires de vérification :

1. une vérification de cohérence structurelle des modèles ;
2. une vérification de la conformité des services web produits par rapport aux spécifications en utilisant des tests automatisés.

Le deuxième niveau ne sera pas détaillé ici. En effet, on peut se reposer sur les outils existants déjà présents dans l'industrie (voir Section 3.4.1).

Pour le premier niveau, nous parlons de « cohérence structurelle des modèles » car il ne s'agit pas de vérifier la conformité par rapport aux spécifications (c'est le rôle du deuxième niveau) mais plutôt de vérifier que les modèles sont construits conformément à des exigences de cohérence autres que syntaxiques. En effet, il est possible de créer des modèles conformes au méta-modèle (la structure syntaxique) présenté dans le Chapitre 4 qui présenteront pourtant des incohérences ou ambiguïtés bloquant la production des services web ou leur stabilité.

Dans la Section 5.2, nous définissons cette cohérence structurelle à travers un ensemble de règles. Cette approche est très similaire à celle du typage, mais intervient ici à une échelle différente. En effet, elle va par exemple laisser toute liberté aux développeurs quant aux implémentations des composants atomiques (puisque celles-ci sont en dehors des modèles) mais ira contraindre l'assemblage de ces composants en se basant sur la compatibilité de leurs contrats respectifs.

Cette cohérence se base donc en grande partie sur les contrats des composants. Exprimer ces contrats sous la forme de préconditions et d'effets est une des principales différences avec notre approche précédente, Reifier [41], où les contrats étaient exprimés sous forme d'entrées et de sorties.

5.2 Règles de cohérence structurelle d'un modèle de services web

Cette section présente la définition de la cohérence structurelle, ainsi que les règles qui la composent.

Définition. Un modèle de services web $m \in M$ est structurellement cohérent s'il vérifie les propriétés suivantes, définies par les Équations (5.1) à (5.12), (5.14) et (5.16) à (5.23).

Les règles sont exprimées comme des propositions logiques dans lequelles on note m le modèle qui est vérifié ; on a donc toujours $m \in M$.

Certaines notations utilisées dans les équations sont définies dans la Section 4.1.

5.2.1 Règles élémentaires

Unicité des noms de composants

Chaque composant (atomique ou composite) possède un nom (voir Équations (4.7) et (4.8)). Ce nom est un identifiant qui doit être unique pour chaque composant d'un modèle.

$$\forall c, c' \in m.components \cdot (c.name = c'.name \Rightarrow c = c') \tag{5.1}$$

Unicité des noms d'entité

Chaque entité possède un nom (voir Équation (4.2)). Ce nom est un identifiant qui doit être unique pour chaque entité d'un modèle.

$$\forall e, e' \in m.entities \cdot (e.name = e'.name \Rightarrow e = e') \tag{5.2}$$

Unicité des noms d'attribut

Chaque entité possède des attributs (voir Équation (4.2)) qui possèdent un nom (voir Équation (4.3)). Ce nom est un identifiant qui doit être unique pour chaque attribut au sein d'une entité.

$$\forall e \in m.entities \cdot (\forall a, a' \in e.attributes \cdot (a.name = a'.name \Rightarrow a = a')) \tag{5.3}$$

Unicité des noms de paramètre de service

Chaque service possède des paramètres (voir Équation (4.12)) qui possèdent un nom (voir Équation (4.13)). Ce nom est un identifiant qui doit être unique pour chaque paramètre au sein d'un service.

$$\forall s \in m.services \cdot (\forall p, p' \in s.params \cdot (p.name = p'.name \Rightarrow p = p')) \tag{5.4}$$

Unicité des paramètres de service de type body

Chaque service possède des paramètres (voir Équation (4.12)) qui possèdent une localisation (voir Équation (4.13)). Cette localisation peut avoir la valeur body, ce qui signifie que le paramètre se verra attribué la valeur du corps de la requête HTTP reçue par le service. Pour éviter les ambigüités, et comme il n'y a pas vraiment d'usage à cette pratique, un même service ne peut pas avoir plusieurs paramètres de type body.

$$\forall s \in m.services \cdot (\forall p, p' \in s.params \cdot (p.location = p'.location = \mathsf{body} \Rightarrow p = p')) \tag{5.5}$$

Cohérence des références

À plusieurs endroits du modèle, il est possible d'utiliser des propriétés de type *Id* pour référencer des éléments définis à d'autres endroits du modèle. Le Tableau 5.1 montre les types d'éléments pouvant être référencés et les emplacements possibles de ces références.

Ces références doivent être cohérentes, c'est-à-dire désigner un élément qui existe dans le modèle.

$$\forall ref \in (refs_{c1} \cup refs_{c2}) \cdot (\exists c \in m.components \wedge c.name = ref) \tag{5.6}$$

$$\forall ref \in (refs_{e1} \cup refs_{e2} \cup refs_{e3} \cup refs_{e4} \cup refs_{e5} \cup refs_{e6}) \cdot (\exists e \in m.entities \wedge e.name = ref) \tag{5.7}$$

Cibles de la référence	**Emplacements de la référence**
Composant C	$refs_{c1} = Prj_{components.component}(m.components \cap CC)$
	$refs_{c2} = Prj_{component.component}(m.services)$
Entité E	$refs_{e1} = Prj_{entity}(Prj_{attributes.type}(m.entities) \cap EntityRef)$
	$refs_{e2} = Prj_{entity}(Prj_{params.type}(m.components) \cap EntityRef)$
	$refs_{e3} = Prj_{entity}(Prj_{pre.type}(m.components \cap AC) \cap EntityRef)$
	$refs_{e4} = Prj_{entity}(Prj_{add.type}(m.components \cap AC) \cap EntityRef)$
	$refs_{e5} = Prj_{entity}(Prj_{rem.type}(m.components \cap AC) \cap EntityRef)$
	$refs_{e6} = Prj_{entity}(Prj_{params.variable.type}(m.services) \cap EntityRef)$

TABLE 5.1 – Description des différentes possibilités pour référencer des éléments principaux du modèle

Unicité des noms de variables dans les contrats des composants

Les contrats des composants atomiques (leurs propriétés `pre`, `add` et `rem`) sont des ensembles de variables (voir Équation (4.7)) dont chacune possède un nom (voir Équation (4.3)). Pour éviter les ambiguïtés, ce nom doit être unique au sein de la définition d'un composant atomique, sauf s'il s'agit bien de la même variable.

$$\forall c \in (m.components \cap AC) \cdot (\forall v, v' \in (c.pre \cup c.add \cup c.rem) \cdot (v.name = v'.name \Rightarrow v = v')) \quad (5.8)$$

Présence de sous-composants des composants composites

Les composants composites contiennent une ou plusieurs instances de composants (voir Équation (4.8)). Pour éviter les comportements indéfinis, cette liste d'instances de composants ne doit pas être vide.

$$\forall c \in (m.components \cap CC) \cdot (c.components \neq \emptyset) \quad (5.9)$$

Unicité des sources des alias

Les alias des instances de composants possèdent une référence vers un nom de variable source (voir Équation (4.11)). Ce nom doit être unique au sein d'une

même instance de composant.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot (\forall a, a' \in ci.alias \cdot (a.source = a'.source \Rightarrow a = a')) \quad (5.10)$$

Unicité des cibles des alias

Les alias des instances de composants possèdent une référence vers un nom de variable cible (voir Équation (4.11)). Ce nom doit être unique au sein d'une même instance de composant.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot (\forall a, a' \in ci.alias \cdot (a.target = a'.target \Rightarrow a = a')) \quad (5.11)$$

5.2.2 Règles avancées

Non-circularité des références

Une entité ne doit pas être elle-même dans les références transitives de sa propre définition. Il en va de même pour les composants composites qui ne doivent pas être référencés transitivement dans leurs sous-composants. Nous définissons les règles qui permettront d'empêcher cette circularité.

$$\forall e \in m.entities \cdot (EntityRef(e.name) \notin deps_e(m, EntityRef(e.name))) \quad (5.12)$$

La fonction $deps_e : M \times T \rightarrow \mathcal{P}(T)$ renvoie l'ensemble des dépendances transitives d'un type donné. $deps_e(m, t)$ est définie de la manière suivante :

$$\text{Rappel : } T \equiv OptionOf(t') \cup SeqOf(t') \cup EntityRef(name) \cup \{String, Boolean, Integer, Float, Date, DateTime\}$$
$$\text{avec } t' \in T, name \in Id$$

$$deps_e(m,t) = \begin{cases} deps_e(m,t') & \text{si } t = OptionOf(t') \\ deps_e(m,t') & \text{si } t = SeqOf(t') \\ t'' \cup \bigcup_{t_i \in t''} deps_e(m,t_i) & \text{si } t = EntityRef(name) \\ deps_e(m,t) = \emptyset & \text{sinon} \end{cases} \quad (5.13)$$

avec $t'' = Prj_{type}(resolve_e(m,name).attributes)$

La fonction $resolve_e : M \times Id \rightarrow E$ renvoie la définition de l'entité qui porte le nom passé en deuxième paramètre.

$$\forall c \in (m.components \cap CC) \cdot (c.name \notin deps_c(m,c)) \quad (5.14)$$

La fonction $deps_c : M \times C \rightarrow \mathcal{P}(Id)$ renvoie l'ensemble des dépendances transitives d'un composant donné. $deps_c(m,c)$ est définie de la manière suivante :

$$deps_c(m,c) = \begin{cases} \emptyset & \text{si } c \in AC \\ Prj_{name}(c.components) \cup \bigcup_{ci_i \in c.components} deps_c(m, resolve_c(m,ci_i)) & \text{si } c \in CC \end{cases} \quad (5.15)$$

La fonction $resolve_c : M \times Id \rightarrow C$ renvoie la définition du composant qui porte le nom passé en deuxième paramètre. Elle est définie dans la Section 4.4.1.

Validité des sources des alias

Les alias des instances de composants possèdent une référence vers un nom de variable source (voir Équation (4.11)). Il doit exister une variable du même nom dans le contrat du composant instancié (ses propriétés `pre`, `add` et `rem`).

$$\begin{aligned} \forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot \\ (\forall a \in ci.alias \cdot (\exists c \in C \wedge c = resolve_c(m, ci.component) \Rightarrow \\ (a.source \in Prj_{name}(c.pre \cup c.add \cup c.rem)))) \end{aligned} \quad (5.16)$$

Validité des cibles des alias

Les alias des instances de composants possèdent une référence vers un nom de variable cible (voir Équation (4.11)). Il doit exister une variable du même nom dans les variables qui seront ajoutées au contexte par le composant instancié.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot (\forall a \in ci.alias \cdot (\exists c \in C \wedge c = resolve_c(m, ci.component) \Rightarrow (a.target \notin Prj_{name}(c.add)))) \quad (5.17)$$

Validité des chemins des services

Les services possèdent un chemin (voir Équation (4.12)) exprimé sous la forme d'une chaîne de caractères. Ce chemin peut contenir des paramètres, qui sont représentés dans la chaîne de caractères par leurs noms entourés d'accolades. Pour chacun des paramètres de type `path` déclarés par le service, il doit exister exactement un paramètre du même nom dans son chemin.

$$\forall s \in m.services \cdot (\{p \in s.params \mid p.location = \texttt{Path}\} = extract(s.path)) \quad (5.18)$$

La fonction *extract* est de type $P \rightarrow \mathcal{P}(V)$. Elle permet d'extraire les noms des paramètres d'un chemin donné. En d'autres termes, pour un chemin donné, elle récupère le contenu de la première parenthèse capturante de l'ensemble des occurences de l'expression régulière `\{([A-Za-z0-9_]+)\}`.

Non interférence entre contexte et dépendances

Un composant atomique ne doit pas ajouter au contexte une variable dont il dépend.

$$\forall c \in (m.components \cap AC) \cdot (c.add \cap c.pre = \emptyset) \quad (5.19)$$

Exhaustivité des préconditions des composants

Un composant atomique c doit avoir dans ses préconditions $c.pre$ chaque variable du contexte qu'il retire.

$$\forall c \in (m.components \cap AC) \cdot (c.rem \subseteq c.pre) \tag{5.20}$$

Cohérence des arguments des instances de composants

Les instances de composants possèdent un ensemble d'arguments (voir Équation (4.9)) qui associent un terme à une variable. Pour chacune de ces associations, le type du terme doit être identique au type de la variable.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot (\forall b \in ci.binding \cdot (b.variable.type = b.argument.type)) \tag{5.21}$$

Exhaustivité des paramètres des instances de composants

Les instances de composants fournissent des arguments (voir Équation (4.9)) pour chaque paramètre du composant instancié (voir Équations (4.7) et (4.8)). Chaque instance de composant doit fournir autant d'arguments que le composant instancié a de paramètres ; les noms et types des arguments doivent correspondre à ceux des paramètres.

$$\forall ci \in (Prj_{components}(m.components \cap CC) \cup Prj_{component}(m.services)) \cdot (\exists c \in m.components \cdot (c.name = ci.component \wedge Prj_{param}(ci.binding) = c.params)) \tag{5.22}$$

Validité des contextes

Les composants atomiques ont un contrat qui définit des préconditions sur leur contexte d'exécution (la présence de variables) et décrit les effets qu'ils auront sur celui-ci (ajout ou suppression de variables). Lorsque des composants sont aplatis (voir Section 4.4.1 et Équation (4.15)), chaque composant atomique doit

s'exécuter dans un contexte qui satisfait ses préconditions. Cette assertion pose l'hypothèse que les alias ont été propagés sur les sous-composants. Elle est formalisée par l'Équation (5.23) :

$$\forall s \in S \cdot (flatten(m, s.component) \blacktriangleleft Pjr_{variable}(s.params)) \tag{5.23}$$

Nous introduisons $\blacktriangleleft$ comme une fonction de type $List(AC) \times \mathcal{P}(V) \rightarrow Boolean$ en notation infixée [1]. Cette fonction retourne un résultat vrai lorsqu'elle est appliquée à un composant et à un contexte qui satisfait les préconditions du composant. Elle est définie par les règles sémantiques suivantes :

$$\frac{ctx_0 \in \mathcal{P}(V)}{[\,] \blacktriangleleft ctx_0} \tag{5.24}$$

$$\frac{\begin{array}{c} ctx_0 \in \mathcal{P}(V) \\ \forall i \in [0,n], c_i \in CI \end{array} \qquad c_0.pre \subseteq_{\blacktriangleleft} ctx_0 \qquad \begin{array}{c} ctx_1 = ctx_0 \cup c_0.add \setminus c_0.rem \\ [c_1,\ldots,c_n] \blacktriangleleft ctx_1 \end{array}}{[c_0,\ldots,c_n] \blacktriangleleft ctx_0} \tag{5.25}$$

où $c_i.pre \subseteq_{\blacktriangleleft} ctx$ signifie que les préconditions pre d'un composant c_i sont satisfaites par le contexte ctx. Nous notons l'inclusion $\subseteq_{\blacktriangleleft}$ à la place de $\subseteq$ parce que l'inclusion classique nécessite que les types soient strictement identiques, ce qui ne permet pas de gérer les types optionnels. $\subseteq_{\blacktriangleleft}$ est défini par l'Équation (5.26).

$$\frac{\begin{array}{c} pre, ctx \in \mathcal{P}(V) \\ \forall v \in pre \cdot v \in ctx \vee (\exists v' \in ctx, v.name = v'.name \\ \wedge v'.type = OptionOf(v.type)) \end{array}}{pre \subseteq_{\blacktriangleleft} ctx} \tag{5.26}$$

5.3 Exemples de violations

Pour illustrer certaines des règles précédentes et discuter de leurs implications, nous proposons quelques exemples de modèles qui violent ces règles.

Exemple 1. Soit $m_1 \in M$, le modèle défini par le Listing 5.1.

1. $cs \blacktriangleleft ctx$ est équivalent à $\blacktriangleleft (cs, ctx)$

```
e
  name User
  attributes (name: String, email: String)

s
  method GET
  path /users
  ci GetUser

ac
  name FetchUsers
  add (users: SeqOf(User))

ac
  name SerializeUsers
  pre (users: SeqOf(User))

cc
  name GetUsers
  ci FetchUsers
  ci SerializeUsers
```

Listing 5.1 – Exemple de modèle enfreignant une règle de cohérence structurelle

Si on évalue les règles simples (définies dans la Section 5.2.1) pour ce modèle, la règle de cohérence des références (voir Équation (5.6)) est enfreinte.

Pour évaluer cette règle, on réduit les valeurs de $refs_{c1}$ et $refs_{c2}$ (voir Tableau 5.1) :

$$\begin{aligned} refs_{c1} &= Prj_{components.component}(m_1.components \cap CC) \\ &= Prj_{components.component}(\{(\text{“}GetUsers\text{”}, \emptyset, [(\text{“}FetchUsers\text{”}, \emptyset, \emptyset), \\ &\qquad\qquad (\text{“}SerializeUsers\text{”}, \emptyset, \emptyset)])\}) \\ &= \{\text{“}FetchUsers\text{”}, \text{“}SerializeUsers\text{”}\} \end{aligned} \tag{5.27}$$

$$\begin{aligned} refs_{c2} &= Prj_{component.component}(m_1.services) \\ &= Prj_{component.component}(\{(\text{“}GetUser\text{”}, \emptyset, \emptyset)\}) \\ &= \{\text{“}GetUser\text{”}\} \end{aligned}$$

La règle s'appliquant à une valeur $ref \in (refs_{c1} \cup refs_{c2})$ (voir Équation (5.6)), elle se décline dans notre cas en trois propositions p_1, p_2 et p_3 :

$$\begin{aligned} p_1 &: \exists c \in m_1.components \wedge c.name = \text{“}FetchUsers\text{”} \\ p_2 &: \exists c \in m_1.components \wedge c.name = \text{“}SerializeUsers\text{”} \\ p_3 &: \exists c \in m_1.components \wedge c.name = \text{“}GetUser\text{”} \end{aligned} \tag{5.28}$$

p_1 et p_2 sont vraies, mais pas p_3. En effet, il n'existe pas de composant nommé `GetUser`. Il s'agit ici d'une faute de frappe : le composant voulu est défini sous le nom `GetUsers`.

Exemple 2. Soit $m_2 \in M$, le modèle défini par le Listing 5.2.

```
s
  method GET
  path /forecast/{city}
  param path city: String
  ci Forecast

cc
  name Forecast
  ci FetchTemperature
  ci SerializeWeather

ac
  name FetchTemperature
  pre (city: String)
  add (temperature: Float)

ac
  name FetchPrecipitation
  pre (city: String)
  add (precipitation: Float)

ac
  name SerializeWeather
  pre (temperature: Float, precipitation: Float)
```

Listing 5.2 – Exemple de modèle enfreignant une règle de cohérence structurelle

On va évaluer la règle de validité des contextes (voir Équation (5.23)) pour ce modèle. Dans l'exemple il n'y a qu'un seul service :

$$s_1 = (\text{“}GET\text{”}, \text{“}/forecast/\{city\}\text{”}, \{(path, (\text{“}city\text{”}, String))\}, (\text{“}Forecast\text{”}, \emptyset, \emptyset)) \tag{5.29}$$

Par commodité, on note également :

$$\begin{aligned} cc_1 &= (\text{“}Forecast\text{”}, \emptyset, [(\text{“}FetchTemperature\text{”}, \emptyset, \emptyset), \\ &\qquad (\text{“}SerializeWeather\text{”}, \emptyset, \emptyset)]) \\ ac_1 &= (\text{“}FetchTemperature\text{”}, \emptyset, \{(\text{“}city\text{”}, String)\}, \\ &\qquad \{(\text{“}temperature\text{”}, Float)\}, \emptyset) \\ ac_2 &= (\text{“}SerializeWeather\text{”}, \emptyset, \{(\text{“}temperature\text{”}, Float), \\ &\qquad (\text{“}precipitation\text{”}, Float)\}, \emptyset, \emptyset) \end{aligned} \tag{5.30}$$

Premièrement, on aplatit les assemblages de composant liés à chaque service du modèle (voir Équation (4.15)) :

$$\begin{aligned} &flatten(m_2, s_1.component) \\ &\quad = flatten(m_2, (\text{“}Forecast\text{”}, \emptyset, \emptyset)) \\ &\quad = flatten(m_2, \text{“}Forecast\text{”}) \\ &\quad = flatten(m_2, cc_1) \\ &\quad = flatten(m_2, (\text{“}FetchTemperature\text{”}, \emptyset, \emptyset)) \\ &\qquad \cup flatten(m_2, (\text{“}SerializeWeather\text{”}, \emptyset, \emptyset)) \\ &\quad = flatten(m_2, \text{“}FetchTemperature\text{”}) \\ &\qquad \cup flatten(m_2, \text{“}SerializeWeather\text{”}) \\ &\quad = flatten(m_2, ac_1) \cup flatten(m_2, ac_2) \\ &\quad = [ac_1, ac_2] \end{aligned} \tag{5.31}$$

On doit ensuite évaluer $flatten(m, s_1.component) \blacktriangleleft Prj_{variable}(s.params)$, c'est-à-dire $[ac_1, ac_2] \blacktriangleleft \{(\text{“}city\text{”}, String)\}$. Cela revient à vérifier le système suivant :

$$\begin{cases} ac_1.pre \subseteq_{\blacktriangleleft} \{(\text{“}city\text{”}, String)\} \\ [ac_2] \blacktriangleleft \{(\text{“}city\text{”}, String)\} \cup ac_1.add \setminus ac_1.rem \end{cases} \tag{5.32}$$

La deuxième équation étant équivalente à $[ac_2] \blacktriangleleft \{(\text{“}city\text{”}, String), (\text{“}temperature\text{”}, Float)\}$, on obtient :

$$\begin{cases} ac_1.pre \subseteq_{\triangleleft} \{(\text{“}city\text{”}, String)\} \\ ac_2.pre \subseteq_{\triangleleft} \{(\text{“}city\text{”}, String), (\text{“}temperature\text{”}, Float)\} \end{cases} \tag{5.33}$$

C'est-à-dire :

$$\begin{cases} \{(\text{“}city\text{”}, String)\} \subseteq_{\triangleleft} \{(\text{“}city\text{”}, String)\} \\ \{(\text{“}temperature\text{”}, Float), (\text{“}precipitation\text{”}, Float)\} \\ \qquad\qquad\qquad\qquad \subseteq_{\triangleleft} \{(\text{“}city\text{”}, String), (\text{“}temperature\text{”}, Float)\} \end{cases} \tag{5.34}$$

Cette deuxième équation est fausse. Cela signifie que la variable (“*precipitation*”, *Float*) manque dans le contexte d'exécution du composant `SerializeWeather`. L'échec de cette règle de vérification permet de constater que le composant `FetchPrecipitation` aurait dû être instancié avant `SerializeWeather` dans la définition de `Forecast`, ce qui aurait réglé le problème.

5.4 Conclusion

Nous avons maintenant des règles de vérification qui permettent de garantir que les services web respectent les exigences de cohérence (sémantique) que nous avons considérées. Ces règles peuvent être utilisées comme base théorique pour développer des outils permettant notamment l'aide à l'écriture de modèles structurellement cohérents.

Chapitre 6

Une approche intégrée à OpenAPI 3.0

Sommaire

Dans le Chapitre 4 nous avons introduit un méta-modèle de services web ainsi qu'une syntaxe concrète permettant de simplifier l'écriture de modèles (voir Section 4.3). Bien que théoriquement suffisante pour écrire des modèles de services web (vérifiables en utilisant les règles présentées dans le Chapitre 5), nous proposons de compléter cette syntaxe concrète par une seconde manière d'écrire les modèles : en utilisant des modèles OpenAPI 3.0 étendus. Les raisons du choix de OpenAPI parmi ses concurrents sont développées dans la Section 3.3.2.

Dans la Section 6.1, nous présentons OpenAPI, les enjeux qui y sont liés et un processus de développement couramment utilisé dans l'industrie. Dans la Section 6.2, nous montrons comment s'intègre notre approche à celle de OpenAPI.

6.1 OpenAPI : fonctionnement et enjeux

La spécification OpenAPI [35] définit un standard pour décrire des API HTTP d'une manière qui est agnostique aux langages de programmation utilisés pour leur implémentation. L'objectif de telles descriptions est de permettre à la fois à des humains et des machines de découvrir et de comprendre comment opérer les fonctionnalités offertes par les services web, sans qu'il soit pour autant nécessaire de recourir à des moyens comme l'étude du code source desdits services, de leur documentation ou du trafic réseau de leurs instances.

L'ensemble du projet OpenAPI a une approche d'IDM. Cette approche se base sur trois composantes :

1. la spécification de OpenAPI, qui définit un méta-modèle ;
2. des modèles conformes au méta-modèle, qui décrivent des services web ;
3. un écosystème d'outils compatibles avec le méta-modèle, qui peuvent exploiter ou manipuler les modèles et ainsi apporter de la valeur.

Exemple de cas d'usage. Il arrive souvent dans l'industrie qu'on souhaite réaliser une application présentant n interfaces utilisateur : pour les navigateurs web, pour plusieurs types de mobiles ou tablettes, pour être utilisées en ligne de commande, … Comme ces n interfaces doivent manipuler des données communes et ont chacune leur propre base de code et leurs propres environnements d'exécution, la solution la plus simple est souvent de centraliser la source de vérité des données et la logique métier associée en un seul point sous la forme de services web. Les n interfaces doivent alors définir un protocole de communication avec les services web, pour que les $n+1$ applications puissent échanger des données. Dans le cas où elles sont chacune développée par des équipes dédiées et ont des cycles de vie différents, cela peut être difficile : les différentes équipes doivent s'assurer que le protocole de communication est cohérent entre toutes les applications, et parfois même dans des cas où elles ne peuvent pas imposer aux utilisateurs finaux de mettre à jour les applications sur leurs terminaux.

La présence d'un modèle OpenAPI décrivant les services web permet alors de simplifier la communication entre les équipes en définissant ce protocole de communication comme un contrat semi-formel décrivant comme utiliser l'API HTTP. Les développeurs écrivant les n interfaces doivent alors se conformer à ce modèle, et ceux écrivant les services web peuvent maintenir leur implémentation librement tant qu'elle s'y conforme également.

Écosystème d'outils. L'exemple précédent laisse entrevoir les avantages d'une solution comme OpenAPI lors de la réalisation d'applications comportant plusieurs backends. Cependant, même dans des contextes différents, il est rarement inintéressant d'avoir un modèle OpenAPI pour chaque API HTTP mise en œuvre. En effet, il existe de nombreux outils conçus pour fonctionner avec de tels modèles et apporter de la valeur grâce à l'automatisation de certaines tâches, notamment. Par exemple, certains comme *Swagger Editor* [54] et *Swagger UI* [55] fournissent une interface graphique à partir d'un modèle donné, rendant son exploration et sa manipulation par des humains bien plus facile. D'autres comme *Dredd* [6] sont capables de générer des tests fonctionnels pour vérifier qu'une implémentation donnée respecte un modèle OpenAPI. On citera finalement *PHPSwaggerGen* [63] qui permet la génération d'un modèle OpenAPI à partir de services web PHP annotés. Il existe bien sûr beaucoup d'autres outils ; ceux cités ici donnent un aperçu de la valeur qu'ils peuvent apporter aux développeurs.

Dans les sous-sections suivantes, nous présentons un exemple officiel de modèle OpenAPI et discutons des différents processus de développement possibles lorsque OpenAPI est impliqué dans un projet.

6.1.1 L'exemple du *Petstore*

Pour faciliter le développement d'outils et permettre un apprentissage inductif de la spécification OpenAPI, celle-ci fournit plusieurs exemples officiels de modèles s'y conformant. Un de ces exemples, nommé *Petstore*, décrit une petite application composée de quatres services. Ces services web permettent de lister, afficher, ajouter et supprimer des enregistrements de données représentant des animaux de compagnie.

Le modèle OpenAPI complet du *Petstore* [36] est disponible dans le dépôt de la spécification [1]. Il s'agit d'un document au format YAML, respectant le schéma défini par la spécification. Les paragraphes suivants détaillent des extraits de ce document, pour donner de l'intuition sur la manière dont il est construit.

Définition d'un service. Le Listing 6.1 montre l'extrait de ce document qui définit le service permettant de récupérer les informations d'un animal à partir de son identifiant. Dans cet extrait, cette définition est constituée des éléments suivants :

1. https://github.com/OAI/OpenAPI-Specification/blob/3.0.1/examples/v3.0/petstore-expanded.yaml

— un chemin ;
— une méthode HTTP ;
— une description en langage naturel ;
— une liste de paramètres (comportant un seul élément ici : un entier nommé `id` devant être présent dans le chemin) ;
— un ensemble de réponses HTTP que le service peut renvoyer.

```yaml
paths:
  /pets/{id}:
    get:
      description: Returns a user based on a single ID, if the
       ↪  user does not have access to the pet
      operationId: find pet by id
      parameters:
        - name: id
          in: path
          description: ID of pet to fetch
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: pet response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
```

Listing 6.1 – Exemple d'un service du Petstore

Définition de structures de données. On peut voir que certaines des propriétés de cet extrait comportent des références vers d'autres parties du modèle. C'est le cas de la ligne `$ref: '#/components/schemas/Pet'` qui désigne le

schéma nommé `Pet`. Le Listing 6.2 montre l'extrait du document qui définit ce schéma. On peut constater que la spécification OpenAPI permet de définir des structures de données comportant des attributs ayant un nom et un type. De plus, ces attributs peuvent être marqués comme obligatoires et il est possible de définir un schéma en ajoutant des champs à un autre schéma.

```
components:
  schemas:
    Pet:
      allOf:
        - $ref: '#/components/schemas/NewPet'
        - required:
          - id
          properties:
            id:
              type: integer
              format: int64
    NewPet:
      required:
        - name
      properties:
        name:
          type: string
        tag:
          type: string
```

Listing 6.2 – Exemple de deux schémas de données du Petstore

Les spécifications complètes sont disponibles en ligne [36].

6.1.2 Processus de développement et limitations

Il existe deux solutions principales pour exploiter OpenAPI et son écosystème d'outils dans le processus de développement de services web. La différence majeure entre ces deux solutions est la place qu'occupe le modèle OpenAPI dans le processus, notamment s'il est une entrée ou une sortie des étapes automatisées.

Méthodes par raffinement. Les développeurs écrivent manuellement un modèle OpenAPI, puis utilisent des outils pour réaliser des projections de ce modèle

dans le but de faciliter l'implémentation de services web conformes. Par exemple, *Swagger Code Generator* [53] permet, à partir d'un modèle OpenAPI de générer un squelette d'implémentation. L'outil supporte différents langages et frameworks comme cible de génération. Ce squelette d'implémentation contient du code dit *boilerplate*, c'est-à-dire nécessaire mais à faible valeur ajoutée et répétitif pour les développeurs, dont le contenu est le résultat de la projection d'informations déjà présentes dans le modèle OpenAPI. Une fois le squelette généré, les développeurs peuvent le compléter avec le reste du code nécessaire au bon fonctionnement des services web et à la logique métier.

Méthodes par abstraction. Les développeurs implémentent des services web, puis utilisent des outils pour en extraire un modèle OpenAPI. Cela permet notamment de générer *gratuitement* une documentation des services web. Un de ces outils, *PHPSwaggerGen* [63], analyse des applications écrites en PHP [60] pour générer des modèles OpenAPI. Il est souvent nécessaire d'ajouter des annotations (ou équivalent) dans l'implémentation pour y indiquer des informations qui n'y figurent pas normalement mais sont nécessaires à la création de modèles OpenAPI ; par exemple les descriptions en langage naturel.

Dans les deux cas, l'état final souhaité est d'avoir un modèle OpenAPI et une implémentation de services web conforme à ce modèle. Cela permet ensuite d'utiliser des outils se basant sur le modèle pour améliorer la qualité des services web ou leur documentation. Ce deuxième aspect est crucial dans certains contextes, comme vu au début de la Section 6.1.

Limitations. Les méthodes par abstraction sont très utiles pour documenter des services web existants. Elles ont également l'avantage d'assurer la synchronisation entre l'implémentation et le modèle, car ce dernier est généré à partir de l'implémentation et peut donc suivre le même cycle de vie et être regénéré par la suite si elle évolue. Cependant, ces méthodes ne sont, par définition, pas adaptées à l'approche descendante ou « top-down ». Cette approche est pourtant très intéressante dans le contexte présenté dans l'exemple du début de la Section 6.1 : lorsque plusieurs équipes développent des applications communiquant avec une même instance de services web, il est intéressant que les équipes puissent rapidement converger vers un modèle OpenAPI commun qui satisfasse les contraintes de chaque équipe ; cette tâche est plus simple quand les équipes peuvent travailler uniquement sur le modèle, sans avoir à maintenir des implémentations incomplètes en parallèle.

Les méthodes par raffinement n'ont pas ce problème, puisqu'elles partent du modèle pour générer l'implémentation. Cependant, elles présentent une autre limitation : les outils de génération actuels ne sont utiles que pour démarrer un projet ou un module. En effet, si le modèle évolue ensuite et que les développeurs veulent lancer une nouvelle génération, les changements manuels qu'ils ont faits sur le code issu de la précédente génération seront écrasés, faute de méthode et de mécanisme pour les fusionner avec le nouveau code généré. Dans les projets qui ont initialement adopté une méthode par raffinement à partir d'un modèle OpenAPI pour gagner du temps au début de la phase d'implémentation, il n'est pas rare que ce modèle soit laissé à l'abandon par les développeurs une fois l'application mise en production ; celle-ci continue d'évoluer, mais n'est plus synchronisée avec le modèle.

6.2 Extensions à OpenAPI 3.0 pour la construction de services web

Pour résoudre la limitation des approches par raffinement, et pouvoir bénéficier de leurs avantages ainsi que de l'écosystème de OpenAPI, nous proposons des extensions à la spécification OpenAPI 3.0. Ces extensions ont pour objectif de fournir le cadre pour ajouter dans les modèles OpenAPI des informations concernant la manière dont les services sont implémentés. Grâce à elles, il n'est plus nécessaire que les développeurs modifient le code généré, puisque cette approche leur donne suffisamment de flexibilité pour exprimer la logique métier en amont de la phase de génération. Il devient alors anodin de regénérer l'implémentation, ce qui augmente les chances en pratique que modèle et implémentation restent alignés.

Dans les sous-sections suivantes, nous comparons notre méta-modèle avec OpenAPI 3.0 pour prendre acte des informations communes et de celles inédites présentes dans chacune des approches, ce qui permettra ensuite de spécifier de quelle manière étendre OpenAPI pour nos besoins.

6.2.1 Comparaison entre OpenAPI 3.0 et notre méta-modèle

Avant que les références qu'il contient ne soient résolues, un modèle OpenAPI peut être vu comme un arbre ayant pour nœud racine un objet de type `OpenApi`. Pour faire référence à certaines parties de cet arbre, nous utiliserons une notation de *chemins* : il s'agit d'une liste de noms de nœuds séparés par le

symbole > pour parcourir l'arbre depuis sa racine jusqu'à la branche désignée. Par exemple, `OpenAPI > components > requestBodies` désigne la valeur du nœud `requestBodies`, enfant du nœud `components`, lui-même enfant de la racine.

Nous comparons notre méta-modèle et OpenAPI 3.0 en regardant trois aspects : le modèle de données, le modèle de processus et le modèle de services.

Modèle de données. Un modèle OpenAPI 3.0 contient un ensemble d'objets `Schema` dans le chemin `OpenAPI > components > schemas`. Un objet de type `Schema` définit une structure de données pouvant être référencée et réutilisée depuis plusieurs endroits du modèle. Par exemple, le Listing 6.1 référence le schéma `Pet`, défini dans le Listing 6.2 au chemin `OpenAPI > components > schemas`. La définition de ces schémas est spécifiée par un standard appelé *JSON Schema* [27]. Ce standard permet bien plus d'expressivité que le système d'entités et de types que nous avons introduit dans la Section 4.2.1. Par conséquent, notre méta-modèle ne comporte pas plus d'informations que OpenAPI 3.0 en ce qui concerne le modèle de données, au contraire.

Modèle de processus. De par sa nature et son périmètre, OpenAPI 3.0 ne contient pas d'information concernant le comportement des services. Par contre, notre méta-modèle est construit autour d'un système de composants permettant l'expression et la réutilisation de ces comportements. Il est donc nécessaire d'inclure dans OpenAPI 3.0 la possibilité de définir des composants. L'objet de type `Component` situé au chemin `components` est le bon candidat pour contenir nos définitions de composants : sa fonction est précisément de contenir un ensemble d'objets réutilisables n'ayant d'effet que s'ils sont explicitement référencés depuis d'autres endroits du modèle.

Modèle de services. Décrire des services est la raison d'être de OpenAPI. Il est donc possible de les décrire de manière expressive. Cependant, le modèle de services de OpenAPI 3.0 ne peut pas être vu comme un sur-ensemble du nôtre pour deux raisons.

D'abord, nos services contiennent une instance de composant, qui permet de relier le service à un comportement. OpenAPI 3.0 comporte un champ `operationId` dans ses objets de type `Operation` (qui correspondent à nos services). Bien qu'on pourrait imaginer qu'il puisse faire le lien avec un composant, ce champ a pour fonction de contenir une chaîne unique désignant le service en lui-même, pas une référence vers un objet externe. Par ailleurs, il ne pourrait pas

contenir les autres informations que nos instances de composants contiennent (voir Équation (4.9)).

Ensuite, l'autre manque concerne la propriété `requestBody`. Son type, `RequestBody`, permet de décrire le schéma que le corps des requêtes doit présenter. Pour que les données des corps des requêtes en elles-mêmes puissent être exploitées par les composants, il est nécessaire d'indiquer le nom d'une variable qui les contiendra et sera placée dans le contexte d'exécution des composants.

Donc, notre méta-modèle de services web comporte des aspects qui sont plus riches que OpenAPI. Il faut donc étendre ce dernier pour qu'il prenne en compte les éléments suivants :

— définition de composants réutilisables et paramétrables ;
— instanciation d'un composant pour chaque service.

6.2.2 Spécification des extensions à OpenAPI 3.0

La spécification en elle-même des extensions est disponible dans [47] où elle reprend le même format que la spécification OpenAPI [35] à la fois en terme d'organisation et de forme. Elle résume les concepts vus dans ce chapitre et dans le Chapitre 4 en décrivant la grammaire des extensions.

Elle n'est pas paraphrasée intégralement ici car une grande partie de son contenu ne fait que syntétiser des éléments déjà vus :

— le système de types, tel que décrit dans la Section 4.2.1 ;
— la grammaire des composants, telle que décrite dans la Section 4.2.2.

Cependant, une troisème section montre comment certains éléments de la grammaire de OpenAPI doivent être modifiés. Cette partie est intéressante car elle permet de comprendre la jonction entre un modèle OpenAPI classique et les parties de notre méta-modèle que nous souhaitons ajouter.

Ces modifications sont uniquement des ajouts d'attributs. En effet, la spécification OpenAPI prévoit un mécanisme d'extension qui consiste à ajouter des attributs dont le nom commence par « `x-` ». Les différents outils de l'écosystème ne sont bien sûr pas tenus de prendre en compte ou d'interpréter ces attributs spécifiques, mais leur présence ne rend pas les modèles incompatibles avec ces outils.

Schéma `OpenAPI`

Le Tableau 6.1 montre l'attribut ajouté au schéma `OpenAPI` qui est le schéma racine de la spécification. Cet attribut doit contenir le numéro de version des extensions utilisées par le modèle.

Attribut	Type
`x-swsg-version`	Chaîne de caractères

Table 6.1 – Attribut ajouté au schéma `OpenAPI` dans OpenAPI 3.0

Schéma `Component`

Le Tableau 6.2 montre les attributs ajoutés au schéma `Component`. Ces attributs contiennent les définitions de composants pouvant être réutilisés dans le modèle.

Attribut	Type
`x-swsg-ac`	Liste de composants atomiques
`x-swsg-cc`	Liste de composants composites

Table 6.2 – Attributs ajoutés au schéma `Component` dans OpenAPI 3.0

Schéma `Operation`

Le Tableau 6.3 montre l'attribut ajouté au schéma `Operation` qui est le schéma qui correspond à un service. Cet attribut doit contenir l'instance du composant qui contient la logique à attacher au service.

Attribut	Type
`x-swsg-ci`	Instance de composant

Table 6.3 – Attribut ajouté au schéma `Operation` dans OpenAPI 3.0

Schéma `RequestBody`

Le Tableau 6.4 montre l'attribut ajouté au schéma `RequestBody` qui permet de décrire la structure attendue des corps de requêtes. Cet attribut doit contenir le nom de la variable du contexte qui contiendra le corps de la requête.

Attribut	Type
`x-swsg-name`	Chaîne de caractères

TABLE 6.4 – Attribut ajouté au schéma `RequestBody` dans OpenAPI 3.0

6.3 Conclusion

Pour inscrire notre approche dans le contexte industriel, nous avons proposé des extensions à OpenAPI 3.0 qui permettent d'intégrer notre méta-modèle de services web dans les modèles OpenAPI. Cela offre une solution intéressante pour développer des services web par raffinement, tout en permettant de bénéficier de l'interopérabilité de OpenAPI et de l'écosystème d'outils développés par l'industrie autour de lui.

Troisième partie

Outillage et mise en œuvre

Chapitre 7

Automatisation de la construction et de la vérification de services web

Sommaire

Dans les chapitres précédents, nous avons défini une approche conceptuelle pour construire des services web, basée sur l'IDM. Nous avons spécifié notre méta-modèle dans le Chapitre 4, précisé des règles de cohérence structuelle des modèles dans le Chapitre 5 et fusionné l'approche avec OpenAPI, un méta-modèle utilisé dans l'industrie, dans le Chapitre 6. Pour permettre l'utilisation de cette approche dans des cas réels, nous avons développé un outil.

La Section 7.1 présente cet outil et le processus de développement dans lequel il s'inscrit. Les Sections 7.2 et 7.3 détaillent la manière dont il vérifie et génère les services web. Enfin, les Sections 7.4 et 7.5 montrent deux aspects qui peuvent être travaillés pour améliorer l'expérience développeur de l'approche.

7.1 SWSG : un outil au cœur de l'approche

Pour supporter notre approche de construction automatique de services web à partir d'un modèle OpenAPI 3.0 étendu (voir Chapitre 6), nous proposons un outil nommé **Safe Web Services Generator** (SWSG) [46].

SWSG est un outil en ligne de commande qui a besoin de deux entrées pour fonctionner :

— un fichier de modèle ;
— le chemin d'un dossier contenant les implémentations des composants atomiques de ce modèle.

Processus de fonctionnement. La Figure 7.1 montre le fonctionnement de SWSG, qui se base sur quatre étapes séquentielles :

1. **Analyse du modèle.** Le modèle d'entrée est analysé pour déterminer s'il s'agit de la syntaxe concrète présentée dans la Section 4.3 ou d'un modèle OpenAPI 3.0 étendu (voir Chapitre 6). Dans le premier cas, la syntaxe concrète est transformée en une représentation interne proche de celle montrée dans la Section 4.2 et l'étape 2 est ignorée. Dans le second cas, le modèle OpenAPI 3.0 est transformé en une représentation interne d'un modèle OpenAPI étendu ;
2. **Transformation du modèle.** Si le modèle est un modèle OpenAPI 3.0 étendu, il est transformé en une représentation interne proche de celle montrée dans la Section 4.2 ;
3. **Vérification de cohérence.** La cohérence structurelle (voir Chapitre 5) du modèle est vérifiée ;
4. **Génération de code.** Le modèle et les implémentations des composants atomiques sont utilisés pour générer une implémentation des services web.

Transformation de modèle. Transformer des modèles OpenAPI 3.0 étendus en modèles compatibles avec notre méta-modèle est plutôt simple, sauf pour les parties liées aux schémas et types. OpenAPI définit des types de données primitifs et repose sur une version modifiée de *JSON Schema Specification* [27] pour les types de données complexes. Deux difficultés se posent :

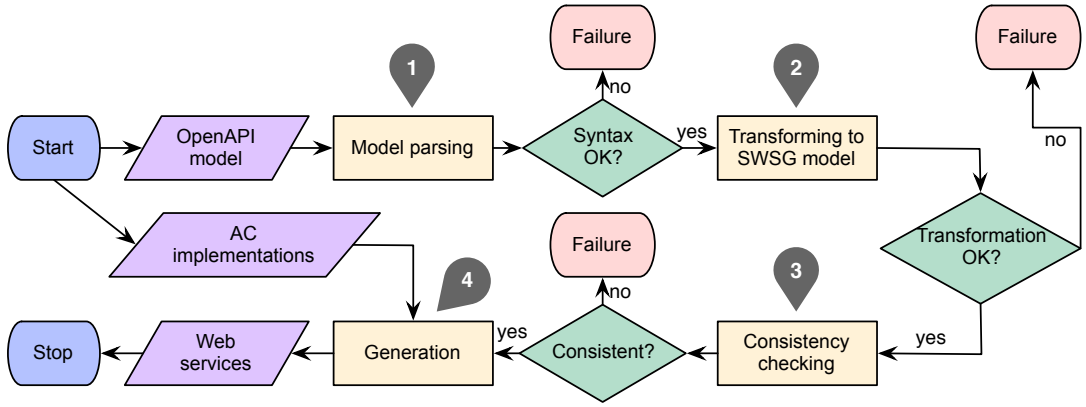

FIGURE 7.1 – Fonctionnement de l'outil SWSG

1. *JSON Schema Specification* est plus expressif que le système de types de notre approche. Par exemple, il permet de définir des types raffinés, comme lorsqu'on spécifie la longueur minimum d'une chaîne de caractères ;
2. *JSON Schema Specification* supporte à la fois la définition littérale et par référence des types des attributs de types complexes. En comparaison notre approche impose des types littéraux pour les types primitifs et des références vers d'autres entités pour les types complexes.

La deuxième difficulté est plutôt un problème technique qui peut être résolu en améliorant l'algorithme de transformation. Le premier problème est plus complexe car il nécessiterait une profonde modification de notre système de types pour que celui-ci supporte l'ensemble des types qu'il est possible d'exprimer dans OpenAPI.

Cependant, pour conserver la simplicité de notre méta-modèle et de SWSG, nous avons choisi de ne pas résoudre ces problèmes. En effet, ils ne nous empêchent pas de tester notre approche à ce stade. En l'état, SWSG peut donc renvoyer des erreurs lorsqu'on lui fournit des modèles OpenAPI contenant de tels types ou références dans les schémas, pourtant valides.

Technologies. SWSG est écrit en Scala [1]. Ce langage a été choisi pour le compromis qu'il offre entre vérification (système de types moderne), flexibilité pour le prototypage et écosystème de bibliothèques. Parmi les bibliothèques utilisées, on retrouve :

1. https://www.scala-lang.org/

Cats. Des structures de données algébriques et autres mécanismes de la programmation fonctionnelle [2] ;

Circe. Analyse, validation et sérialisation de données au format JSON [3] ;

Parboiled. Bibliothèque de *parser combinators* qui simplifient l'écriture de programme d'analyse syntaxique ; utilisé ici pour analyser la syntaxe concrète des modèles d'entrée [4] ;

Twirl. Moteur de template ; utilisé ici pour la génération de code [5] ;

Better Files. API plus simple et sûre que celle de Java pour la lecture et l'écriture de fichiers ; utilisé ici pour la lecture des fichiers d'entrée et l'écriture du code généré [6] ;

Scopt. Analyse des arguments de l'interface en ligne de commande [7] ;

Scala.js. Compilation de code Scala vers du code JavaScript [16].

Une fois compilé, SWSG est facile à distribuer sous la forme d'un fichier JAR et peut être exécuté dans de nombreux environnements. Par ailleurs, SWSG peut être compilé via Scala.js [16] pour obtenir un artefact exécutable par un interpréteur JavaScript ; certains avantages associés à cette possiblité sont développés dans la Section 7.4.

Interface en ligne de commande. Le JAR de SWSG peut être manipulé à travers une interface en ligne de commande. Le Listing 7.1 montre l'aide intégrée à cette interface.

La commande `java -jar swsg.jar` permet d'exécuter le JAR de SWSG. Directement à la suite de cette commande, il est possible de spécifier des arguments séparés par une espace. Le premier argument doit avoir la valeur `check`, pour uniquement vérifier le modèle d'entrée, ou `gen`, pour le vérifier et générer du code. Les paramètres suivants sont des options qui peuvent être placées dans un ordre quelconque et permettent de spécifier les chemins du modèle d'entrée, des implémentations d'entrée et du code généré, ainsi que le nom du *backend* à utiliser pour réaliser ladite génération [8].

2. https://typelevel.org/cats/
3. https://circe.github.io/circe/
4. https://github.com/sirthias/parboiled2
5. https://github.com/playframework/twirl
6. https://github.com/pathikrit/better-files
7. https://github.com/scopt/scopt
8. Actuellement il n'y a qu'un seul *backend* d'implémenté : `laravel`. Cette option est donc facultative tant qu'un second *backend* n'est pas présent.

```
$ java -jar swsg.jar --help
Usage: java -jar swsg.jar [check|gen] [options]

  --help                    prints this usage text
  -m, --model <file>       path to the model
  -i, --implementation <path>
                            path to implementation
  -b, --backend <name>      backend name
  -o, --output <path>       path to the output directory
Command: check
Check a web app model
Command: gen
Generate from a web app model
```

Listing 7.1 – Documentation de l'interface en ligne de commande de SWSG

7.2 Processus de vérification

Le Chapitre 5 introduit formellement des règles de vérification de cohérence structurelle de modèles. Pour qu'un modèle soit cohérent, il doit vérifier toutes ces règles.

Règle de vérification. Le processus de vérification dans SWSG (étape 3 dans la Figure 7.1) implémente ces règles en Scala, mais comporte un apport supplémentaire : si la vérification échoue, il fournit la liste des règles ayant échoué, ainsi que des éléments de contexte permettant de comprendre rapidemment la cause du problème. Le Listing 7.2 montre un trait Scala qui définit la signature de la fonction de vérification. Celle-ci renvoie simplement une liste d'erreurs ; s'il n'y a pas d'erreur, la liste est vide.

```scala
abstract trait Verification {
  def run(model: Model): Seq[ModelError]
}
```

Listing 7.2 – Trait représentant une règle de vérification dans SWSG

Dans la base de code de SWSG, chaque règle de vérification est implémentée dans une classe qui lui est propre et qui étend le trait du Listing 7.2. Cependant, ces règles ont parfois des dépendances entre elles. Par exemple, certaines règles

(comme dans l'Équation (5.16)) ont besoin de résoudre des références vers des composants et d'accéder à leur définition qui doit donc être unique ; ces aspects sont vérifiés par les Équations (5.1) et (5.6) respectivement.

Pour favoriser la maintenabilité du prototype et l'expérience utilisateur, les règles sont implémentées en suivant ces trois containtes :

1. Chaque règle est implémentée dans une fonction pure ;
2. Un maximum de règles est vérifié à chaque exécution de la vérification, pour éviter de renvoyer les erreurs une à une à l'utilisateur ;
3. Les règles dépendant d'autres règles ne doivent pas être exécutées si ces autres règles échouent.

Le troisième point implique le corollaire suivant : si on donne à la fonction principale de vérification un modèle enfreignant théoriquement une seule règle, elle ne doit retourner que les erreurs liées à cette règle. Cela permet d'éviter le bruit et aide l'utilisateur à comprendre plus rapidement pourquoi la vérification a échoué. Pour reprendre l'exemple précédent : si on vérifie un modèle qui comporte plusieurs composants du même nom, on ne veut pas d'erreur liée à la validité des sources d'alias.

Le Listing 7.3 montre l'implémentation de la règle d'unicité des noms de composants (voir Équation (5.1)). Il est intéressant de constater qu'elle renvoie une liste d'objets de type `ComponentNameUnicityError`. Le Listing 7.4 montre que ce type est un sous-type de `ModelError`. Pour chaque règle de vérification, il existe au moins un type d'erreur qui est prévu pour décrire le contexte des erreurs possibles.

Fonction de vérification principale. Pour respecter ces contraintes exposées précédemment, la fonction de vérification principale implémente un processus. Chaque étape du processus est un ensemble de règles de vérification. La première étape comporte un maximum de règles qui ne dépendent d'aucunes autres et la deuxième étape comporte les autres règles qui dépendent d'une ou plusieurs règles de la première étape. Lors de l'exécution de la fonction de vérification principale, celle-ci va exécuter les règles de vérification étape par étape ; si au moins une règle échoue, les étapes suivantes ne sont pas exécutées et la vérification renvoie toutes les erreurs qu'elle a détectées à ce stade. Le Listing 7.5 montre les étapes telles qu'implémentées dans SWSG. Les commentaires en face de chaque règle permettent d'associer les noms de ces règles dans le code de l'outil et les équations correspondantes dans le Chapitre 5.

```scala
final case object ComponentNameUnicity extends Verification {
  def run(model: Model): Seq[ComponentNameUnicityError] = {
    val components       =
     ↪  model.components.toVector.map(_.name)
    val uniqueComponents =
     ↪  model.components.map(_.name).toVector
    val diff             = components.diff(uniqueComponents)
    val duplicates       =
     ↪  diff.groupBy(identity).mapValues(_.size + 1)
    duplicates.toSeq.map(ComponentNameUnicityError.tupled)
  }
}
```

Listing 7.3 – Implémentation de la règle d'unicité des noms de composants dans SWSG

```scala
final case class ComponentNameUnicityError(name: Identifier,
 ↪  occurences: Int) extends ModelError
```

Listing 7.4 – Définition d'un type d'erreur de vérification dans SWSG

7.3 Génération de code

Le processus décrit par la Figure 7.1 est générique : il ne repose pas sur un langage ni sur une technologie spécifique.

Cependant, il est nécessaire que les langages et technologies utilisés pour implémenter les composants atomiques soient les mêmes que ceux pour lesquels on souhaite générer du code. En effet, la génération de code ne génère pas les implémentations des composants atomiques en elles-mêmes mais utilise celles qui lui sont fournies ; il s'agit d'un compromis qui vise à assurer un maximum de flexibilité aux développeurs. Pour se placer dans le contexte technologique de Startup Palace, SWSG génère du code PHP [60] qui met en œuvre le framework Laravel [37].

Intégration à Laravel. Une approche possible pour la génération est de générer un artefact qui correspond à une implémentation autonome des services web, c'est-à-dire pouvant être déployé sans nécessiter de connaissances métier. Une autre approche possible est de générer des fichiers de code source dans une base

```
val levels = Seq(
  Set(
    ComponentNameUnicity, // Eq. 5.1
    EntityNameUnicity, // Eq. 5.2
    AttributeNameUnicity, // Eq. 5.3
    ServiceParameterNameUnicity, // Eq. 5.4
    ServiceBodyUnicity, // Eq. 5.5
    ReferenceConsistency, // Eq. 5.6 & 5.7
    ComponentContextVariableNameUnicity, // Eq. 5.8
    CompositeComponentNonEmptiness, // Eq. 5.9
    ContextVariablesTypeValidity, // **Implementation details**
    AliasSourceUnicity, // Eq. 5.10
    AliasTargetUnicity // Eq. 5.11
  ),
  Set(
    RecursiveReferenceConsistency, // Eq. 5.12 & 5.14
    AliasSourceValidity, // Eq. 5.16
    AliasTargetValidity, //Eq. 5.17
    ServicePathValidity, // Eq. 5.18
    ComponentContextImmutability, // Eq. 5.19
    ComponentPreconditionExhaustivity, // Eq. 5.20
    ComponentInstanceBindingsConsistency, // Eq. 5.21
    ComponentInstanceParametersExhaustivity, // Eq. 5.22
    ContextValidity // Eq. 5.23
  )
)
```

Listing 7.5 – Répartition des règles dans les différentes étapes de vérification de SWSG

de code existante, à côté de ceux écrits par les développeurs.

Notre approche se situe entre les deux : des fichiers de code source sont générés dans une base de code existante, mais ils ne doivent jamais être modifiés manuellement. Les chemins et noms des fichiers générés sont conçus pour ne rien écraser qui serait déjà présent dans la base de code d'un projet Laravel. Une fois ces fichiers générés, l'application Laravel ainsi augmentée ne présente aucun changement observable depuis l'extérieur : pour que le code généré soit actif, il est nécessaire de l'activer en un unique point dans la base de code Laravel.

En effet, Laravel met en œuvre des « service providers ». C'est un mécanisme qui permet d'enregistrer divers services qui pourront ensuite être utilisés pendant le cycle de vie de l'application. L'un de ces « service providers » est le `RouteServiceProvider`. Il s'agit du point d'entrée du routeur de l'application : il détermine quels fichiers de routes seront actifs, et peut forcer l'exécution de « middlewares » pour certains groupes de routes de manière à leur ajouter des comportements en amont de l'exécution des « controllers ». Plutôt que d'écraser un fichier de routes existant, SWSG déclare tous les services dans un nouveau fichier de routes `routes/generated.php`. Il génère également un nouveau « service provider » pour le routeur nommé `GeneratedRouteServiceProvider` ; celui-ci hérite de `RouteServiceProvider` de manière à enregistrer les routes du fichier `routes/generated.php`, puis à exécuter le comportement de `RouteServiceProvider`. Cela permet de mélanger des services générés par SWSG et d'autres écrits manuellement au sein de la même application Laravel, ce qui donne de la flexibilité supplémentaire aux développeurs. La seule étape destructive nécessaire est le remplacement de « `RouteServiceProvider::class` » par « `GeneratedRouteServiceProvider::class` » dans `config/app.php`.

Fichiers générés. SWSG génère quatre types de fichiers :

1. **Composants atomiques.** Les composants atomiques fournis à SWSG sont placés dans le répertoire `app/Components/` ;
2. **Composants composites.** Pour chaque composant composite, SWSG génère un fichier PHP dans le répertoire `app/Components/` [9] ;
3. **Fichier de routes.** L'ensemble des déclarations de services est rassemblé dans un unique fichier `routes/generated.php` [10] ;
4. **Classes statiques.** Indépendamment du contenu du modèle, SWSG a besoin de certaines classes PHP pour que les services web générés puissent

9. Le Listing B.1 montre un exemple de code généré pour un composant composite.
10. Le Listing B.2 montre un exemple de code généré pour déclarer les services.

fonctionner. Ces classes sont toutes placées dans le répertoire `app/SWSG/`, à l'exception du « service provider » mentionné précédemment qui est localisé dans `app/Providers/GeneratedRouteServiceProvider.php`.

7.4 Interface graphique d'aide au développement

Outre la vérification et la génération de code, l'approche par l'IDM permet de nombreuses possiblités en terme d'outillage accompagnant les modèles de services web. Ces outils sont des moyens d'offrir du support aux développeurs.

Une des forces de GestionAIR (voir Chapitre 2) était de proposer une interface graphique. Même si elle présentait de nombreux défauts, elle permettait aux développeurs de visualiser l'architecture macroscopique des services web. De manière similaire, *Swagger Editor* [54] et *Swagger UI* [55] proposent de visualiser un modèle OpenAPI en même temps qu'on l'édite. Même si, pour des raisons techniques, il n'est pas toujours possible de manipuler directement ces interfaces utilisateur et de s'affranchir des moyens d'expression bas-niveau, elles apportent de la valeur en améliorant l'expérience développeur.

Concevoir et réaliser une bonne interface graphique est une tâche longue et complexe. Cependant, durant le développement de SWSG, nous avons identifé un besoin simple et créé une interface graphique minimaliste pour y répondre, à titre de preuve de concept. Le problème est simple : pendant l'écriture d'un modèle de services web, les développeurs créent des assemblages de composants ; il est important qu'ils aient en tête le graphe des dépendances de ces composants, pour savoir dans quels contextes un composant donné est instancié, par exemple.

Comme mentionné précédemment, SWSG peut être compilé vers du JavaScript, via Scala.js [16]. En effet, la logique de SWSG est divisée en trois catégories :

— celle qui peut être compilée pour la machine virtuelle Java et pour JavaScript : la lecture des fichiers de modèle et le moteur de vérification ;
— celle qui peut être compilée pour la machine virtuelle Java uniquement : la génération de code et l'interface en ligne de commande ;
— celle qui peut être compilée pour JavaScript uniquement : une API publique pour la lecture des fichiers de modèle et le moteur de vérification.

Le fichier JavaScript ainsi généré est une bibliothèque qui expose quelques fonctions pures qui permettent d'analyser un fichier de modèle (écrit avec notre syntaxe concrète ou avec le format OpenAPI 3.0 étendu) et de vérifier sa cohérence. Cette bibliothèque JavaScript peut être exécutée dans un navigateur web,

et il est donc possible pour une application web côté client d'en tirer parti.

Pour permettre la visualisation des dépendances des composants d'un modèle, nous avons développé une page web comportant trois éléments :

1. une zone d'édition de texte ;
2. un bouton de validation ;
3. une zone de rendu graphique.

L'utilisateur va interagir avec ces trois éléments dans le même ordre :

1. il saisit (ou copie-colle) un modèle dans la zone de texte ;
2. il clique sur le bouton : le modèle va être analysé et vérifié (étapes 1 à 3 de la Figure 7.1) ;
3. s'il est cohérent, un graphe sera dessiné dans la zone de rendu : chaque nœud est un composant, et chaque arête est une relation d'instantiation.

Le code de cette interface graphique est disponible dans le dossier `gui/` de la base code de SWSG [46]. La Figure 7.2 montre un exemple de rendu. Les nœuds verts représentent des composants composites et les nœuds bleus des composants atomiques ; les flèches indiquent une relation de dépendance.

Cette interface graphique est minimaliste, mais elle montre qu'il est possible d'exploiter une partie des fonctionnalités de SWSG dans le contexte d'une page web.

7.5 Cohérence des composants atomiques

Dans le processus de développement présenté dans la Figure 7.1, les développeurs doivent fournir un modèle de services web et des implémentations pour les composants atomiques qui y figurent. Si le modèle ne capture pas les subtilités de leurs comportements telles qu'elles sont définies dans l'implémentation, il comporte un contrat qui définit comment ils peuvent interagir avec leur contexte d'exécution. La phase de vérification comporte des règles qui se basent sur ce contrat ; il faut donc qu'il soit cohérent par rapport à la réalité de l'implémentation.

Dans l'état actuel de SWSG, c'est à la charge des développeurs de s'assurer que l'implémentation de chaque composant atomique est conforme au contrat annoncé dans le modèle. Sans ça, la vérification se base sur des prémisses qui sont en partie fausses, et n'a donc pas beaucoup d'intérêt.

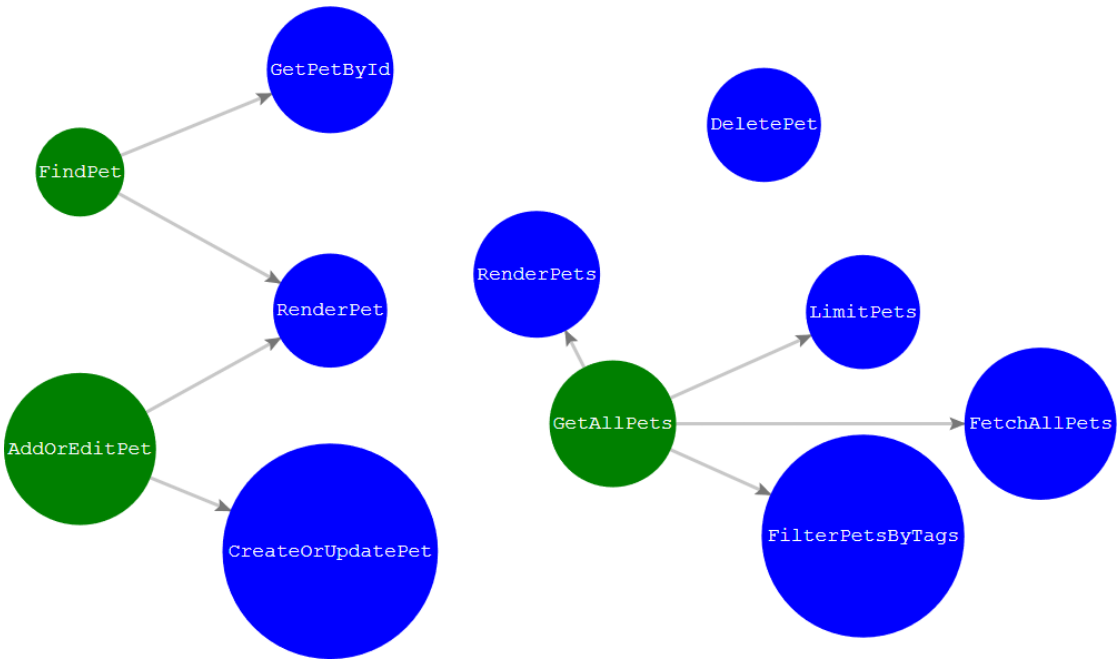

FIGURE 7.2 – Exemple du rendu de l'interface graphique

Dans un prototype précédent non publié, une solution avait été expérimentée :

— les développeurs pouvaient ne pas indiquer les contrats des composants atomiques dans les modèles (ils devaient juste déclarer les composants en leur donnant un nom) ;
— avant l'étape de vérification, l'outil invoquait un utilitaire externe qui analysait les implémentations des composants atomiques et complétait leurs déclarations avec leurs contrats ;
— la vérification et la génération de code se déroulaient ensuite comme dans le SWSG actuel.

Pour extraire le contrat depuis une implémentation de composant atomique, l'outil analysait de manière statique l'implémentation pour répérer comment elle manipulait son contexte d'exécution. Par exemple, si une variable du contexte est lue, alors le composant doit avoir cette variable dans ses préconditions. Cependant, cette tâche est complexe dans le cas général à cause de la nature du langage PHP :

1. sa nature dynamique rend possible des comportements qui sont difficiles à détecter : par exemple, on pourrait accéder à la variable `myVar` en

l'écrivant « `strtolower("MY")."Var"` »[11] voire même en déterminant son nom à partir d'informations qui ne sont disponibles qu'à l'exécution, ou uniquement dans une seule des branches d'une condition ;

2. l'absence de typage statique rend difficile l'identification des types des variables du contexte, qui est nécessaire dans le contrat du composant.

Pour pouvoir tout de même réaliser une analyse statique fiable, nous avions imposé des contraintes pour éviter ces problèmes. Par exemple, les noms des variables devaient être spécifiés de manière littérale et les actions d'écriture dans le contexte réalisées hors de tout bloc conditionel.

Pour gagner du temps sur l'implémentation, cette analyse statique était réalisée par un programme externe, lui-même écrit en PHP. Ce choix était motivé par la présence de la bibliothèque *PHP-Parser*[12] dans l'écosystème PHP. Cette bibliothèque permet d'analyser du code PHP et offre des outils pour traverser et inspecter l'Abstract Syntax Tree (AST) correspondant.

Cette approche était fonctionnelle et simplifiait le processus de développement. Elle n'est cependant pas présente dans la version actuelle de SWSG car ce n'était qu'une preuve de concept. Elle nécessiterait d'être re-conçue complètement, tant au niveau théorique qu'au niveau de l'implémentation en elle-même, ce qui demande du temps.

7.6 Conclusion

Pour outiller notre approche, nous avons développé SWSG : un outil en ligne de commande qui permet de vérifier la cohérence structurelle d'un modèle de services web (ou d'un modèle OpenAPI étendu) et de générer le code d'une implémentation de ces services web. D'autres fonctionnalités ont été imaginées et prototypées : l'ajout d'une interface graphique et la vérification de cohérence des implémentations des composants atomiques (fournies par les développeurs) avec les définitions correspondantes dans le modèle.

11. L'opérateur « `.` » en PHP permet la concaténation de chaînes de caractères.
12. https://github.com/nikic/PHP-Parser

Chapitre 8

Études de cas

Sommaire

Pour tester notre approche, nous montrons dans ce chapitre comment la mettre en œuvre pour réaliser des services web réels. Pour mieux simuler les contraintes réelles du développement de MVP, notre deuxième cas d'étude comporte deux phases entre lesquelles les spécifications des services web évoluent.

Les modèles et codes complets de nos cas d'étude sont disponibles dans le dépôt de SWSG [1].

8.1 Processus de développement

Dans la Section 6.1.2, nous avons présenté et comparé deux solutions principales pour exploiter OpenAPI et son écosystème d'outils dans le processus de développement de services web : les méthodes par raffinement et celles par abstraction.

1. https://gitlab.startup-palace.com/research/swsg/tree/master/examples

Dans le Chapitre 7, nous avons introduit SWSG, un outil qui exploite des modèles OpenAPI 3.0 étendus et les raffine pour générer une implémentation des services web.

Pour pouvoir utiliser SWSG dans des situations réelles, voici le processus de développement que nous recommandons :

1. Réaliser un modèle OpenAPI stable qui décrit les services web. « Stable » signifie ici que ce modèle a fait l'objet de plusieurs aller-retours entre les mainteneurs et les consommateurs des services : le modèle doit converger vers une version qui capture les contraintes de ces deux types d'acteurs ;
2. Concevoir un système de composants associé à chaque service. Là encore, plusieurs itérations sont probablement nécessaires pour identifier les composants similaires et choisir s'il faut les fusionner dans un composant plus générique ou pas. Il est possible d'utiliser SWSG en mode « vérification-seule » pour tester la cohérence de ce système de composants en ayant uniquement besoin du modèle ;
3. Écrire l'implémentation de chaque composant atomique ainsi spécifié. Il est possible d'utiliser une interface graphique (voir Section 7.4) pour mieux visualiser les contrats ou les différents contextes d'exécution de chaque composant, par exemple ;
4. Utiliser SWSG pour générer une implémentation fonctionnelle des services web.

8.2 Cas d'étude : *Registration*

Pour illustrer le fonctionnement de SWSG, nous proposons de revenir sur le cas d'étude fictif et minimaliste qui a été brièvement introduit dans la Section 4.4.2. Ainsi, ce cas n'exploite pas l'intégration avec OpenAPI présentée dans le Chapitre 6 et utilise à la place la syntaxe concrète de notre méta-modèle présentée dans la Section 4.3.

Objectif. Pour maintenir une liste d'inscriptions numériques pour un événement, on souhaite des services web qui répondent aux besoins suivants :

1. Permettre aux utilisateurs de s'inscrire à l'événement en donnant un nom et une adresse email ;
2. Fournir aux organisateurs la liste des personnes inscrites à l'événement.

Spécification informelle. Pour l'enregistrement d'un utilisateur, on doit envoyer une requête HTTP `POST` au service `/register/[name]/[email]` où `[name]` est son nom et `[email]` son adresse email. Si cet utilisateur est déjà enregistré, le service doit renvoyer un message d'erreur dans une réponse HTTP de code `403`. Si l'adresse email donnée est invalide, le service doit renvoyer un message d'erreur dans une réponse HTTP de code `422`. Si aucun de ces deux cas ne se réalise, les informations transmises par l'utilisateur doivent être persistées dans une base de données relationnelle avec la date et l'heure actuelle, et le service doit renvoyer une réponse HTTP de code `200` contenant ces informations.

Pour la consultation de la liste des personnes inscrites par un organisateur, on doit envoyer une requête HTTP `GET` au service `/attendees/[key]` où `[key]` est une chaîne de caractères qui doit être identique à celle spécifiée dans l'implémentation du service, de manière à authentifier la requête. Si la clé fournie est incorrecte, le service doit renvoyer un message d'erreur dans une réponse HTTP de code `401`. Sinon, il doit récupérer les inscriptions dans la base de données relationnelle, les sérialiser et les renvoyer dans une réponse HTTP de code `200`.

Pour implémenter ces services web, nous utilisons le processus montré dans la Section 8.1 à une différence près : comme nous n'utilisons pas l'intégration avec OpenAPI, la première étape n'est plus d'écrire un modèle OpenAPI standard, mais d'utiliser la syntaxe concrète de notre méta-modèle pour définir entités et services. C'est en fait une différence plutôt mineure car il s'agit de définir les mêmes informations qu'avant, seule la manière de le faire change.

Étape 1. Premièrement, on définit l'ensemble des entités de notre modèle. Dans ce cas, il n'y en a qu'une seule qui représente les inscriptions et que nous nommons `Registration`. Comme on peut le voir dans le Listing 8.1, elle contient un nom, une adresse email et une date d'inscription.

```
1 e
2   name Registration
3   attributes (name: String, email: String, date: DateTime)
```

Listing 8.1 – L'entité `Registration`

Puis, on définit deux services qui correspondent aux spécifications. Pour chaque service on instancie un composant qui sera chargé de générer des réponses HTTP. Le résultat est montré dans le Listing 8.2.

```
5  s
6    method POST
7    path /register/{name}/{email}
8    param path name: String
9    param path email: String
10   ci Registration
11 s
12   method GET
13   path /attendees
14   param query key: String
15   ci GetAttendees(apiKey = "mykey")
```

Listing 8.2 – Les services d'inscription et de consultation de la liste des inscrits

Étape 2. On définit un composant composite qui va gérer le service d'inscription. Le Listing 8.3 montre qu'il est défini en termes de cinq composants, chacun responsable d'une étape du processus : validation de l'adresse email, vérification d'unicité de l'inscription, création de l'objet représentant l'inscription (avec la date et l'heure courante), enregistrement de cet objet en base de données et sérialisation et envoi de ses données.

```
17 cc
18   name Registration
19   ci ValidateEmail
20   ci CheckDupRegistration
21   ci CreateRegistration
22   ci SaveRegistration
23   ci RegistrationSerializer
```

Listing 8.3 – Le composant composite `Registration`

La même chose est faite pour le composant composite qui va afficher la liste des inscrits en authentifiant la requête, récupérant les inscriptions et les sérialisant, comme en atteste le Listing 8.4.

Enfin, dans le Listing 8.5, on définit les contrats de tous les composants atomiques instanciés.

La ligne 27 du Listing 8.4 montre l'utilisation d'un alias : on décide de renommer localement la variable `userKey` de la définition de `CheckKey` (voir la ligne 50 du Listing 8.5) de manière à faire le lien avec la variable `key` extraite de la requête

```
cc
  name GetAttendees
  params (apiKey: String)
  ci CheckKey(correctKey = apiKey)<userKey -> key>
  ci FetchRegistrations
  ci RegistrationsSerializer
```

Listing 8.4 – Le composant composite `GetAttendees`

```
ac
  name ValidateEmail
  pre (email: String)
ac
  name CheckDupRegistration
  pre (name: String, email: String)
ac
  name CreateRegistration
  pre (name: String, email: String)
  add (registration: Registration)
ac
  name SaveRegistration
  pre (registration: Registration)
ac
  name RegistrationSerializer
  pre (registration: Registration)
ac
  name CheckKey
  params (correctKey: String)
  pre (userKey: String)
ac
  name FetchRegistrations
  add (registrations: SeqOf(Registration))
ac
  name RegistrationsSerializer
  pre (registrations: SeqOf(Registration))
```

Listing 8.5 – Les composants atomiques utilisés par `Registration` et `GetAttendees`

HTTP entrante (voir la ligne 14 du Listing 8.2). Dans ce cas précis on aurait pu directement nommer la variable `key` dans la définition de `CheckKey` car ce composant n'est instancié qu'une seule fois ; on illustre ici la fonctionnalité des alias qui est utile lorsqu'un composant est instancié plusieurs fois dans des contextes différents.

Étape 3. Pour chaque composant atomique, on développe une implémentation dans un langage de programmation. Ici, on utilise PHP, dans le contexte du framework Laravel. Le Listing 8.6 montre l'implémentation du composant `CheckDupRegistration`, qui va récupérer les variables `name` et `email` du contexte, compter le nombre d'inscriptions présentes en base de données avec ce couple, et renvoyer une erreur s'il y en a au moins une.

Étape 4. Enfin, on exécute SWSG en lui donnant le modèle complet et les implémentations des composants atomiques. SWSG va vérifier la cohérence structurelle du modèle en utilisant les règles décrites dans le Chapitre 5. Par exemple, la règle de validité des contextes (voir Équation (5.23)) stipule que chaque composant doit être exécuté dans un contexte comptatible avec les préconditions du composant. Ainsi, le composant `SaveRegistration` requiert une variable `registration` ; celle-ci est ajoutée dans le contexte par le composant `CreateRegistration` qui est exécuté juste avant, dans le seul assemblage où `SaveRegistration` est instancié. Si on décide de ne pas instancier `CreateRegistration` en supprimant la ligne 21 du Listing 8.3, on obtient l'erreur montrée dans le Listing 8.7. Cette erreur contient des informations permettant de comprendre pourquoi la règle de vérification a échoué ; dans ce cas :

— le nom de l'erreur ;
— le service concerné ;
— le chemin (dans l'assemblage de composants associé au service) du composant dont les préconditions ne sont pas respectées ;
— la variable qui manque dans le contexte d'exécution.

S'il n'y a pas d'erreur, SWSG génère le code des services web. Pour vérifier que cette implémentation générée respecte les contraintes énoncées dans les spécifications, nous avons également implémenté des tests fonctionnels. Ces tests sont montrés dans les Listings A.1 et A.2.

Ce cas d'étude est conçu pour illustrer l'utilisation de SWSG : nous avons montré à travers ce cas comment SWSG réussit à identifier des problèmes dans les premières versions du modèle. Ces erreurs étaient souvent des erreurs de frappe

```php
<?php

namespace App\Components;

use App\SWSG\Component;
use App\SWSG\Ctx;
use App\SWSG\Params;

use DB;

class CheckDupRegistration implements Component
{
    public static function execute(Params $params, Ctx $ctx)
    {
        $dups = DB::table('registration')
            ->where('name', $ctx->get('name'))
            ->where('email', $ctx->get('email'))
            ->count();
        if ($dups > 0) {
            return response('Already registered!', 403);
        }
        return $ctx;
    }
}
```

Listing 8.6 – L'implémentation du composant atomique `CheckDupRegistration`

```
ComponentPreconditionError(
  "POST /register/{name}/{email}",
  List(Registration, SaveRegistration),
  Variable(registration, EntityRef(Registration))
)
```

Listing 8.7 – Une erreur de vérification structurelle

mais parfois des erreurs de conception. Le fait d'avoir un outil qui détecte ce genre de problèmes rapidement dans le processus de développement réduit le temps perdu au total.

8.3 Cas d'étude : *Petstore*

Ce cas d'étude se déroule en deux phases espacées dans le temps.

8.3.1 Phase 1

Pour mettre en œuvre notre approche, nous nous donnons pour objectif de réaliser les services web spécifiés par l'exemple du *Petstore* [36] fourni avec OpenAPI et introduit dans la Section 6.1.1.

Nous parcourons les différentes étapes du processus.

Étape 1. Nous considérons que le modèle OpenAPI du *Petstore* répond à nos besoins. Les Listings 6.1 et 6.2 montrent certaines parties de ce modèle.

Étape 2. Nous avons besoin d'un composant pour contenir la logique du service qui renvoie un animal à partir de son identifiant, comme spécifié dans le Listing 6.1. Nous créons un composant composite nommé `FindPet` et le référençons dans le service, comme montré dans les lignes 27 et 28 du Listing 8.8. Ce composant a deux enfants qui sont des composants atomiques :

- `GetPetById` récupère un identifiant d'animal dans le contexte, requête la base de données pour obtenir un animal comportant cet identifiant et ajoute le résultat de type `Pet` au contexte ;
- `RenderPet` récupère un animal dans le contexte, le sérialise au format JSON et génère une réponse HTTP contenant le résultat.

Ces trois composants sont définis dans le Listing 8.9. Cette tâche est également réalisée pour les autres services du *Petstore*.

Pour valider cette étape, nous lançons SWSG en mode « vérification-seule ». SWSG indique une erreur de type `PreconditionError`. Cette erreur montre que la règle définie par l'Équation (5.23) n'est pas vérifiée, c'est-à-dire que les variables spécifiées dans les préconditions d'un composant ne sont pas présentes dans un des contextes d'instanciation de ce composant. Dans le cas présent,

```yaml
paths:
  /pets/{id}:
    get:
      description: Returns a user based on a single ID, if the
       ↪ user does not have access to the pet
      operationId: find pet by id
      parameters:
        - name: id
          in: path
          description: ID of pet to fetch
          required: true
          schema:
            type: integer
            format: int64
      responses:
        '200':
          description: pet response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Pet'
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Error'
      x-swsg-ci:
        component: FindPet
```

Listing 8.8 – Exemple d'un service du Petstore référençant un composant

```yaml
components:
  x-swsg-cc:
    - name: FindPet
      components:
        - component: GetPetById
        - component: RenderPet
  x-swsg-ac:
    - name: RenderPet
      pre:
        - name: pet
          type:
            entity: Pet
    - name: GetPetById
      pre:
        - name: id
          type: String
      add:
        - name: pet
          type:
            entity: Pet
```

Listing 8.9 – Quelques composants dans le modèle du Petstore

le détail de l'erreur nous apprend que la variable `id` de type `String` requise par le composant `GetPetById` n'est pas présente dans le contexte lorsque ce dernier est instancié par le composant `FindPet`, lui-même instancié par le service `GET /pets/{id}`. En effet, nous avons volontairement introduit une erreur dans le Listing 8.9 : le composant `GetPetById` nécessite un identifiant de type `String` alors que le service lui donne un identifiant de type `Int` (voir la ligne 12 du Listing 6.1).

Dans cet exemple et à cette phase du développement, la meilleure solution est de changer cette précondition au niveau de `GetPetById` pour qu'il ait besoin d'un identifiant de type `Int`. Cependant, dans des projets plus complexes et avancés, ce composant aurait peut-être été utilisé dans d'autres parties du modèle et cette solution aurait peut-être cassé la cohérence dans ces parties sans que les développeurs ne s'en rendent compte rapidement. C'est dans la résolution de ce genre d'erreurs que SWSG fait gagner du temps : comme elles sont détectées tôt, au moment de la conception, et non pas après avoir implémenté une grande partie du système ou pire, à l'exécution, ces erreurs sont moins coûteuses à réparer.

Étape 3. Maintenant que l'erreur est réparée et que SWSG indique un modèle cohérent, nous implémentons tous les composants atomiques qui sont déclarés dans celui-ci.

Le Listing 8.10 montre l'implémentation de `GetPetById`. La classe PHP qui y est définie implémente l'interface `Component` et utilise les classes `Ctx` et `Params`. Ces trois entités sont introduites dans des fichiers statiques générés par SWSG (voir Section 7.3) et sont juste des détails d'implémentation pour ce générateur de code en particulier. D'autres générateurs de code fonctionnant avec SWSG pourraient imposer des contraintes différentes sur les implémentations des composants atomiques.

Étape 4. Grâce au modèle OpenAPI que nous avons étendu (étape 2) et aux implémentations des composants atomiques que nous avons écrites (étape 3), SWSG peut générer une implémentation fonctionnelle des services web du *Petstore*. Par exemple, le Listing B.1 montre le code généré correspondant au composant composite `FindPet` (défini dans le Listing 8.9).

```php
<?php
namespace App\Components;
use App\SWSG\Component;
use App\SWSG\Ctx;
use App\SWSG\Params;
use DB;

class GetPetById implements Component
{
    public static function execute(Params $params, Ctx $ctx)
    {
        $pet = DB::table('pet')
            ->where('id', $ctx->get('id'))
            ->first();
        $ctx->add('pet', $pet);
        return $ctx;
    }
}
```

Listing 8.10 – Implémentation du composant atomique `GetPetById`

8.3.2 Phase 2

Quelques temps après que les services web du *Petstore* aient été implémentés, nous décidons que nous voulons les étendre pour ajouter un nouveau service pour gérer les requêtes de type `PUT /pets/{id}`. Ces requêtes permettent de définir les informations liées à l'animal portant une valeur d'identifiant donnée ; si celui-ci n'existe pas, il sera créé avec l'identifiant donné, sinon l'existant sera modifié. Ce comportement est similaire à celui du service `POST /pets` ; la différence est que ce dernier ne reçoit pas d'identifiant dans la requête et va donc en choisir un qui n'est pas utilisé, ce qui fait que ce service ne permet que de créer des nouveaux animaux alors le nouveau service permet également de les modifier.

Pour réaliser cette évolution, nous utilisons le même processus que précédemment.

Étape 1. Nous ajoutons la description du nouveau service dans notre modèle OpenAPI.

Étape 2. Nous devons instancier un composant depuis ce service. À cause de la similarité entre ce service et le service `POST /pets`, nous décidons de réutiliser et étendre les composants existants et utilisés par ce deuxième service. Le composant atomique `CreatePet` est ainsi altéré de la manière suivante :

— il est renommé `CreateOrUpdatePet`;
— on lui déclare un paramètre booléen : `createOnly`;
— on ajoute une nouvelle précondition sur une variable `id` de type `OptionOf(Int)`.

Lorsque ce composant est instancié dans le service `POST /pets`, on donne au paramètre `createOnly` la valeur `true`; lorsqu'il est instancié dans le service `PUT /pets/{id}`, on lui donne la valeur `false`. Ce paramètre va pouvoir être utilisé dans l'implémentation du composant pour déterminer le comportement à choisir.

Le composant composite `AddPet`, parent de `CreatePet`, est également renommé en `AddOrEditPet` et altéré pour instancier `CreateOrUpdatePet` au lieu de `CreatePet`. On lui ajoute également un paramètre booléen, dont la valeur sera transmise au paramètre `createOnly` vu précédemment. Le Listing 8.11 montre la nouvelle version de ce composant, et le Listing 8.12 montre comment il est instancié dans le nouveau service `PUT /pets/{id}`.

```
x-swsg-cc:
  - name: AddOrEditPet
    params:
      - name: addOnly
        type: Boolean
    components:
      - component: CreateOrUpdatePet
        bindings:
          - param:
              name: createOnly
              type: Boolean
            argument:
              name: addOnly
              type: Boolean
      - component: RenderPet
```

Listing 8.11 – Définition du composant composite `CreateOrUpdatePet`

Un appel à SWSG en mode « vérification-seule » permet de valider ce modèle

```
paths:
  /pets/{id}:
    put:
      ...
      x-swsg-ci:
        component: AddOrEditPet
        bindings:
          - param:
              name: addOnly
              type: Boolean
            argument:
              type: Boolean
              value: false
```

Listing 8.12 – Instanciation de `AddOrEditPet` dans le nouveau service

altéré.

Étape 3. L'implémentation de `CreateOrUpdatePet` est modifiée pour créer ou modifier un animal en fonction de la valeur du paramètre `createOnly` et de la présence d'une valeur dans la variable du contexte `id` de type `OptionOf(Int)`. Ces changements reflètent ceux opérés sur la définition de `CreateOrUpdatePet` à l'étape 2.

Étape 4. SWSG peut générer une implémentation fonctionnelle des services web de notre version améliorée du *Petstore*.

8.4 Discussion

Notre approche a plusieurs avantages par rapport à un processus de développement classique, par exemple écrire l'application complète en utilisant un langage de programmation et un framework web comme Laravel.

D'abord, l'approche par raffinement force les développeurs à penser à leur système (ou à l'évolution de leur système) à une échelle macroscopique avant d'écrire des implémentations de bas-niveau. Associée à la vérification automatique de la cohérence des modèles, cette approche leur offre une manière simple de repérer des erreurs de conception tôt dans le processus, leur faisant ainsi

gagner du temps. Dans un processus classique, les développeurs devraient uniquement se reposer sur les outils de gestion de la qualité proposés par le langage ou le framework, comme des tests unitaires par exemple. Parce qu'il n'y a pas de phase de vérification statique en PHP et parce que beaucoup de développeurs n'écrivent pas de tests automatisés, ils seraient facilement tentés de sauter la vérification complète de la cohérence de leur système, surtout lorsqu'ils opèrent des évolutions mineures.

Ensuite, comme le modèle OpenAPI étendu est utilisé pour générer à la fois implémentation et documentation, les deux restent alignées pendant toute la durée de vie du projet. Dans un processus classique, cet alignement repose sur la volonté des développeurs : ils pourraient arrêter de maintenir le modèle OpenAPI et continuer à faire évoluer l'implémentation, perdant ainsi une grande partie du support offert par l'écosystème d'outils de OpenAPI.

Comme illustré par la seconde phase de notre cas d'étude, ces avantages sont particulièrement intéressants pour le développement incrémental. En effet, ils garantissent différentes formes de cohérence à différents niveaux du projet, pour un coût faible en termes de flexibilité et de productivité. Dans des contextes comme celui de Startup Palace où il faut construire et faire évoluer des services web pour des MVP, cette maîtrise du développement incrémental est cruciale.

Quatrième partie

Conclusion

Chapitre 9

Bilan

Startup Palace développe des applications à base de services web pour de nombreuses startups qui cherchent à lancer un produit numérique, souvent dans le but de tester leurs hypothèses de marché. Pour développer ces MVP, Startup Palace utilise une méthode de développement plutôt courante dans l'industrie, basée sur le langage PHP, le framework web Laravel et les outils présents dans ces deux écosystèmes. Cependant, cette approche comporte des limitations qui découlent de la nature des MVP et des technologies web :

— la variabilité des technologies ;
— la variabilité des spécifications ;
— la productivité des développeurs.

Partant de ces limitations, cette thèse introduit une méthode outillée pour le développement de services web, qui se base sur un méta-modèle compatible avec le standard OpenAPI et accompagné de règles de cohérence structurelle pour fixer la sémantique désirée.

Les problèmes que nous cherchons à résoudre dans cette thèse sont exprimés au travers du point de vue de Startup Palace mais sont des problèmes de génie logiciel, voire d'ingénierie en général. Il existe de nombreuses solutions qui contribuent à les résoudre, mais elles comportent des défauts. La plupart d'entre elles impliquent des changements profonds au contexte technologique de Startup Palace, ce qui n'est pas souhaitable à court terme. D'autres ont un champ d'application qui va bien au-delà de la construction de services web, ce qui ajoute de la complexité aux solutions du point de vue des praticiens et les prive d'un niveau de support dans les outils ; ce support étant difficile ou impossible à implémenter dans le cas général, même si possible en réduisant leur portée.

Nous nous sommes penchés dans cette thèse sur des solutions conceptuellement solides et évolutives avant de pouvoir rebâtir une nouvelle plateforme d'ingénierie web. Pour ce faire, nous avons introduit un méta-modèle qui permet d'exprimer des services web, accompagné par une syntaxe concrète qui simplifie l'écriture des modèles. Les comportements des services web sont exprimés grâce à la définition et l'assemblage de composants. Les comportements de certains de ces composants, dits atomiques, doivent être exprimés en utilisant un langage de programmation, ce qui offre un haut niveau de flexibilité. Par ailleurs, tous les composants disposent d'un contrat, qui spécifie à la fois le contexte d'exécution dont ils ont besoin pour fonctionner grâce à des préconditions et les effets qu'ils auront sur ce contexte d'exécution. Ces contrats sont élémentaires dans l'état actuel de la mise en œuvre.

Nous avons proposé des solutions pour soutenir le développement d'applications correctes. En effet, notre méta-modèle est accompagné par des règles de vérification qui exploitent notamment les contrats des composants pour déterminer la cohérence structurelle d'un modèle. Toutes les règles ont été formalisées, fondant ainsi la cohérence structurelle. La cohérence est dite « structurelle » car elle ne vise pas à vérifier le programme dans son ensemble mais à empêcher certaines catégories de problèmes qui font perdre du temps aux développeurs et sont en même temps détectables sans que ceux-ci ne mettent en œuvre des méthodes formelles plus lourdes. Nous avons montré comment utiliser ces règles pour vérifier la cohérence structurelle des services développés.

Nous avons outillé nos propositions et les avons ouvertes sur des approches déjà industrialisées, de manière à les rendre plus accessible aux praticiens.

Ainsi, nous avons proposé une manière d'intégrer notre méta-modèle dans un standard de l'industrie : OpenAPI. OpenAPI permet la description de services web, mais ne permet pas d'exprimer les comportements liés à ces services ; notre approche est donc complémentaire. L'intégration dans OpenAPI permet aussi de bénéficier sans coût supplémentaire d'un vaste écosystème d'outils qui offrent du support aux développeurs, notamment lors de la phase de conception des services web.

Afin de concrétiser la démarche proposée, nous avons réalisé un prototype d'outil : SWSG, noyau du successeur de GestionAIR. Il s'agit actuellement d'un programme en ligne de commande qui accepte en entrée un modèle ainsi que des implémentations pour chaque composant atomique de ce modèle, et produit une version exécutable des services web décrits. Cette étape de génération de code ne se produit que si le modèle est correct d'un point de vue syntaxique et sémantique ; ce deuxième point étant testé grâce aux règles de cohérence structu-

relle mentionnées précédemment. Dans le cas contraire, l'utilisateur obtient un message d'erreur qui lui permet de comprendre la nature et le contexte de chaque règle enfreinte. SWSG projette notre approche dans le contexte technologique de Startup Palace.

Grâce à SWSG, nous avons pu tester notre approche en réalisant plusieurs cas d'étude. Par exemple, l'un de ces cas d'étude est l'implémentation des services web décrits par un des exemples officiels de OpenAPI. Pour montrer la réaction face à la variablité des spécifications, nous étendons le cas d'étude pour y ajouter une deuxième phase de développement dans laquelle nous modifions la spécification et montrons les implications pour les développeurs qui doivent prendre en compte ces évolutions. Cela démontre la viabilité de notre approche outillée pour la réalisation de services web réels.

Par rapport à la liste de problèmes initiale, à la fin de ces travaux nous avons apporté des solutions à :

Q1 : La variabilité des technologies. Notre approche n'est pas dépendante d'une technologie en particulier, et abstrait une partie de la logique sous la forme d'un modèle lui-aussi indépendant de la technologie choisie ;

Q2 : La variabilité des spécifications. À travers la démarche d'IDM qui est mise en œuvre par notre approche et renforcée par la vérification de la cohérence structurelle des modèles, la variabilité des spécifications a moins de chances d'avoir des conséquences néfastes sur la stabilité des services web ;

Q3 : La productivité des développeurs. L'intégration avec OpenAPI qui facilite grandement une démarche descendante (de la conception vers l'implémentation) pour la réalisation de services web, la possibilité d'exploiter l'écosystème d'outils de OpenAPI complété par des outils liés à SWSG (comme des interfaces graphiques de visualisation des modèles, par exemple) et la génération de code sont trois aspects qui aident les développeurs à se concentrer sur les tâches qui portent de la valeur.

Ainsi, le Tableau 9.1 montre comment nous situons nos travaux par rapport à certaines solutions alternatives identifiées dans l'état de l'art. La description des critères de comparaison (les colonnes) est disponible dans la Section 3.5.

Enfin, cette nouvelle approche surmonte une grande partie des limitations de GestionAIR (exposées dans le détail dans la Section 2.3), comme le montre la liste récapitulative suivante. Les éléments sont préfixés par un symbole : ■ indique que la limitation est complètement ou partiellement surmontée ; □ qu'elle n'est pas surmontée.

	Exprime la logique métier	Flexibilité	Facilité de prise en main	Aide à la correction	Écosystème d'outils	Documentation
Langages fonctionnels	●	● ● ●	● ● ○	● ● ○	● ● ○	● ○ ○
Famille EMF	●	● ● ○	● ○ ○	● ● ○	● ○ ○	● ● ○
M3D	●	● ● ○	○ ○ ○	● ○ ○	○ ○ ○	● ● ○
SWSG	●	● ● ●	● ● ●	● ● ○	● ● ○	● ● ●

● = 1 point ; ○ = 0 point ; ● ● ○ = 2 points sur 3

TABLE 9.1 – Comparaison de SWSG avec des solutions de réalisation de services web existantes

- ■ **L1 : Cohérence des séquences d'actions.** SWSG vérifie la cohérence structurelle des modèles, et notamment que les composants sont bien assemblés en accord avec leurs contrats.
- □ **L2 : Maîtrise du flot d'exécution des séquences d'actions.** Ce problème reste d'actualité car SWSG fonctionne de manière similaire à GestionAIR sur ce point. La Section 10.2.4 explique pourquoi et donne des pistes de solution.
- ■ **L3 : Réutilisation des composants.** En vérifiant la cohérence structurelle des composants et en permettant de leur définir des paramètres, SWSG favorise leur réutilisation.
- ■ **L4 : Intégration de bibliothèques externes.** En générant du code qui s'intègre dans un projet Laravel, SWSG délègue la gestion de l'intégration des bibliothèques externes à des technologies éprouvées et bien connues des développeurs ; par exemple : Composer (le gestionnaire de dépendances de PHP) pour le côté serveur ou npm (celui de JavaScript) pour le côté client.
- ■ **L5 : Compatibilité avec un système de versionnement.** Les éléments en entrée de SWSG (le modèle, les implémentations des composants atomiques), ceux en sortie (le code généré) voire même SWSG lui-même peuvent être facilement ajoutés dans un dépôt géré par un système de versionnement de code (comme Git) et ainsi bénéficier de ses avantages [1].

1. C'est le cas des exemples vus dans le Chapitre 8, qui sont présents dans le dépôt de SWSG

- ■ **L6 : Intégration avancée des composants côté client.** Comme pour la gestion des dépendances externes, le problème est réglé en donnant la possibilité d'utiliser des solutions connues des développeurs en parallèle de SWSG.
- ■ **L7 : Aide à la gestion des services externes.** SWSG n'apporte pas directement de solution à ce problème, mais repose sur celles proposées par le framework Laravel (par exemple, son système de migration de schéma pour les bases de données relationnelles).
- □ **L8 : Passage à l'échelle de l'éditeur.** Peu de temps a été investi dans la réalisation d'un nouvel éditeur (c'est-à-dire d'une interface graphique). Une partie de ce qui est disponible pour SWSG est décrite dans la Section 7.4 ; le reste repose sur les outils de l'écosystème de OpenAPI. Des éléments pour aller plus loin sur ce point sont présentés dans la Section 10.2.2.

Chapitre 10

Perspectives

Sommaire

Ce chapitre présente les sources de complexité et les limitations que nous avons identifiées dans notre approche et qui pourraient faire l'objet de travaux ultérieurs.

10.1 Perspectives pour le méta-modèle

10.1.1 Système de types

Notre méta-modèle comporte un système de types (voir Section 4.2.1). Il sert notamment à typer les paramètres des composants, les variables présentes dans

leur contrat ou les attributs des entités. Ces types sont ensuite utilisés lors de la vérification de cohérence structurelle (voir Chapitre 5) pour détecter d'éventuelles erreurs dans les modèles.

Comme mentionné précédemment (voir Section 6.2.1 et Section 7.1), le système de types de OpenAPI 3.0 (qui est à peu de choses près celui de *JSON Schema Specification* [27]) est plus riche et flexible que le nôtre. Cela peut poser des problèmes : par exemple, un modèle OpenAPI utilisant des types avancés pourrait être à tort déclaré incohérent par SWSG.

Pour éviter ces cas et offrir plus de flexibilité aux développeurs, il serait intéressant d'étudier à quel niveau le système de types de SWSG pourrait être compatible avec celui de OpenAPI 3.0 sans perdre trop de simplicité et de support.

10.1.2 Complexité des contrats des composants

Actuellement, les contrats des composants sont constitués de :

— préconditions, sous la forme d'un ensemble de variables devant être présentes dans le contexte d'exécution ;
— postconditions ou effets, sous la forme de deux ensembles de variables : un qui indique les variables qui seront ajoutées au contexte d'exécution et l'autre celles qui en seront retirées.

Ces contrats sont volontairement simples car ils ne doivent pas être difficiles à écrire pour les développeurs. Cependant, il serait intéressant d'étudier leur extension à deux niveaux.

Premièrement, les préconditions pourraient être plus expressives et permettre l'utilisation d'expressions logiques, dans une certaine mesure. Deuxièmement, les types d'effets pourraient être étendus : par exemple, en ajoutant la possibilité de déclarer les réponses HTTP comme des effets, ou en autorisant également l'utilisation d'expressions logiques pour pouvoir les conditionner par rapport à certaines préconditions.

Ces extensions ne sont pas triviales. Elles pourraient nécessiter l'utilisation de solveurs de contraintes ou la remise en question d'une grande partie du processus de développement lié à SWSG, en plus de la réécriture de ce dernier. Néanmoins, s'il est possible de ne pas ajouter trop de complexité, elles permettraient de faciliter la réutilisation des composants.

10.1.3 Composition de modèles

Notre méta-modèle est conçu pour contenir certaines parties réutilisables, comme par exemple les composants qui sont définis une seule fois et peuvent être instanciés à plusieurs endroits du modèle. Cette fonctionnalité est importante car elle permet la réutilisation de logique au sein du programme.

Il serait intéressant d'étudier comment étendre cette possibilité de réutilisation à des modèles complets. Cela permettrait de créer des bibliothèques contenant des parties de services web, des composants ou des entités.

La spécification de OpenAPI permet de faire des références vers des éléments distants (c'est-à-dire contenus dans un autre fichier que le fichier courant), mais celles-ci ne sont pas supportée par SWSG actuellement. Cette fonctionnalité pourrait être intéressante à exploiter dans l'optique de permettre la composition de modèles.

10.2 Perspectives pour le prototype SWSG

10.2.1 Évaluation de l'approche

Dans le Chapitre 8, nous présentons des cas d'étude qui montrent les avantages de notre approche. Dans l'optique du passage à l'échelle, il serait judicieux de mener d'autres cas d'études plus importants, plus réalistes et comportant plus d'étapes d'évolution des spécifications.

Par ailleurs, la méthode d'évaluation en elle-même constitue un travail important. De nombreux articles du domaine se basent sur des indicateurs partiaux comme le nombre de lignes de code écrites ou le temps de développement en jour-homme. Ces indicateurs ont l'avantage d'être concrets et faciles à mesurer mais sont loin d'être absolus ou reproductibles. Il semble possible d'en imaginer d'autres qui serait correlés à la complexité des éléments manipulés par les développeurs (code et modèle), par exemple. Cela rendrait possible une évaluation comparative des solutions de manière plus reproductible.

10.2.2 Interface graphique

Dans la Section 7.4, nous avons montré une tentative minimaliste pour fournir une interface graphique à SWSG. Celle-ci permettait de lire un modèle et de

visualiser les relations de dépendance entre ses composants. Il s'agissait d'une preuve de concept, qui ouvre des perspectives intéressantes.

Par exemple, il est théoriquement possible de modifier *Swagger Editor* [54] pour qu'il supporte la vérification de modèles OpenAPI 3.0 étendus. Cela en ferait un éditeur de choix pour le processus de développement que nous présentons dans la Section 7.1. En effet, en plus des retours visuels sur la validité du modèle OpenAPI écrit par les développeurs, ceux-ci pourraient également profiter de la vérification de SWSG de manière immédiate.

Par ailleurs, il serait également intéressant d'aller encore plus loin et de proposer un affichage graphique d'informations qui aident les développeurs à concevoir leurs modèles. Notre exemple de la Section 7.4 peut être étendu pour montrer plus de choses, comme les contrats des composants, voire même la structure des contextes intermédiaires, et ainsi fluidifier l'expérience développeur.

10.2.3 Cohérence entre définition et implémentation des composants atomiques

Dans la Section 7.5, nous avons montré l'intérêt qu'il pouvait y avoir à vérifier que les implémentations des composants atomiques respectent effectivement les contrats déclarés dans le modèle. Nous avons aussi montré une précédente tentative d'implémentation de cette fonctionnalité.

Cette tentative mettait en évidence la nécessité de contraindre le code PHP des implémentations de composants pour permettre la vérification. En fonction de certains paramètres, comme la cible de génération, certaines de ces contraintes peuvent sembler arbitraires voire gênantes.

Par conséquent, il serait intéressant de modulariser à la fois cette vérification et la génération, c'est-à-dire d'en extraire la logique du cœur de SWSG. SWSG aurait alors à charge d'invoquer ces modules lorsqu'il a besoin de vérifier la cohérence entre définition et implémentation des composants atomiques ou lorsqu'il a besoin de générer du code. Ces modules pourraient prendre la forme de programmes externes dont l'interface est spécifiée de manière précise, de manière à permettre des cycles de vie différents entre SWSG et ces modules.

Tester voire vérifier automatiquement la cohérence entre la définition et l'implémentation de certains composants atomiques éviterait aux développeurs la tâche pénible de vérifier manuellement, tout en réduisant le risque d'erreur. Modifier SWSG pour que cette vérification (voire même la génération de code) soit

déléguée à des programmes externes permettrait d'offrir plus de flexibilité aux développeurs et une meilleure réactivité face à la variabilité des technologies.

10.2.4 Compromis liés au flot d'exécution des composants

Dans la Section 2.3.2, nous avons montré une limitation de GestionAIR relative à la maîtrise du flot d'exécution des séquences d'actions. En effet, l'implémentation d'un `process` pouvait renvoyer une réponse HTTP ce qui avait pour effet d'arrêter l'exécution des actions et de renvoyer la réponse au client.

Dans notre approche, nous avons conservé ce fonctionnement : l'implémentation d'un composant atomique peut renvoyer un objet représentant une réponse HTTP ce qui a également pour conséquence l'interruption de l'exécution des composants. Pour des raisons de simplicité des modèles, il a été choisi de ne pas offrir beaucoup de flexibilité lors de la définition des composants composites : leurs comportements sont définis par une liste ordonnée d'instances de composants qui représente le cas d'exécution nominal, et il n'est pas possible d'utiliser des constructions plus avancées comme les conditions ou les boucles. Cela permet de faciliter le support, en l'occurrence en simplifiant certaines règles de vérification structurelle.

Nous pensons que ce compromis a du sens dans notre contexte. Cependant, sans aller jusqu'à un niveau de flexibilité comparable à des langages comme BPEL voire à des langages de programmation généralistes, il semble possible d'améliorer cet aspect tout en conservant un compromis proche. Le principal problème opérationnel découlant de la limitation exposée dans la Section 2.3.2 était la difficulté d'identifier les composants qui sont en mesure de stopper l'exécution en renvoyant une réponse HTTP et, par extension, la fiabilité du programme lors de leur réutilisation.

La détection automatique de tels composants en analysant leur implémentation (dans la suite de ce qui est évoqué dans la Section 10.2.3) et en les mettant en évidence dans une interface graphique (voir Section 10.2.2) serait une fonctionnalité intéressante pour les développeurs et un premier pas dans la résolution de ce problème.

Annexe A

Exemples de tests fonctionnels

Cette annexe contient des exemples de tests fonctionnels écrits pour tester les services web générés pour le cas d'étude *Registration* :

— le Listing A.1 montre `tests/Feature/RegistrationTest.php` : le fichier qui définit des tests fonctionnels pour le service d'inscription ;
— le Listing A.2 montre `tests/Feature/GetAttendeesTest.php` : le fichier qui définit des tests fonctionnels pour le service de listage des inscrits.

```php
<?php

namespace Tests\Feature;

use Tests\DBTestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class Registration extends DBTestCase
{

    public function testWrongEmail()
    {
        $response = $this->json(
            'POST',
            '/register/name/notanemail'
        );
        $response->assertStatus(422);
    }

    public function testAlreadyRegistered()
    {
        $existingData =
        ↪  \RegistrationSeeder::registrationData();
        $response = $this->json(
            'POST',
            '/register/' . $existingData[0]['name'] . '/' .
            ↪  $existingData[0]['email']
        );
        $response->assertStatus(403);
    }

    public function testInsertNewItem()
    {
        $newItem = [
            'name' => 'Name',
            'email' => 'email@localhost.local'
        ];
        $response = $this->json(
```

```
            'POST',
            '/register/' . $newItem['name'] . '/' .
             ↪  $newItem['email']
        );
        $response
            ->assertStatus(200)
            ->assertJson($newItem);
        $this->assertDatabaseHas('registration', $newItem);
    }
}
```

Listing A.1 – Tests pour le service d'inscription

```
<?php

namespace Tests\Feature;

use Tests\DBTestCase;
use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class GetAttendeesTest extends DBTestCase
{
    protected $key = "mykey";

    public function testNoKey()
    {
        $response = $this->json(
            'GET',
            '/attendees'
        );
        $response->assertStatus(400);
    }

    public function testWrongKey()
    {
        $response = $this->json(
            'GET',
            '/attendees?key=wrongkey'
        );
```

```
29          $response->assertStatus(401);
30      }
31
32      public function testAttendees()
33      {
34          $expectedData =
            ↪  \RegistrationSeeder::registrationData();
35          $response = $this->json(
36              'GET',
37              '/attendees?key=' . $this->key
38          );
39          $response
40              ->assertStatus(200)
41              ->assertJson($expectedData);
42      }
43  }
```

Listing A.2 – Tests pour le service de listage des inscrits

Annexe B

Exemples de code généré

Cette annexe contient des exemples de code générés par SWSG :

— le Listing B.1 montre le contenu du fichier `app/Components/FindPet.php` qui correspond au code du composant composite `FindPet` tel qu'il est généré par SWSG dans l'exemple du *Petstore* (voir Chapitre 8) ;
— le Listing B.2 montre le contenu du fichier `routes/generated.php` qui déclare les services pour le routeur de Laravel tel qu'il est généré par SWSG dans l'exemple du *Petstore* (voir Chapitre 8).

```php
<?php
// This is a generated file, do not edit

namespace App\Components;

use App\SWSG\Component;
use App\SWSG\Ctx;
use App\SWSG\Params;

class FindPet implements Component
{
  public static function execute(Params $params, Ctx $ctx)
  {

    $ctx0 = \App\Components\GetPetById::execute(new
    ↪  \App\SWSG\Params([]), $ctx);
    if ($ctx0 instanceof \Illuminate\Http\Response) {
      return $ctx0;
    }
    if ($ctx0 instanceof \SWSG\Ctx) {
      $ctx0 = $ctx0;
    }

    $ctx1 = \App\Components\RenderPet::execute(new
    ↪  \App\SWSG\Params([]), $ctx0);
    if ($ctx1 instanceof \Illuminate\Http\Response) {
      return $ctx1;
    }
    if ($ctx1 instanceof \SWSG\Ctx) {
      $ctx1 = $ctx1;
    }

    return $ctx1;
  }
}
```

Listing B.1 – Code généré du composant composite `FindPet`

```php
<?php
// This is a generated file, do not edit

use \Illuminate\Http\Request;
use \Illuminate\Support\Facades\Validator;
use App\SWSG\Ctx;
use App\SWSG\Params;


Route::match(['get'], '/pets', function (Request $req) {
  $initialContext = new Ctx(['tags' => $req->query('tags'),
   ↪  'limit' => $req->query('limit')]);

  $validatorRules = ['tags' => 'array|nullable', 'tags.*' =>
   ↪  'string', 'limit' => 'integer|nullable'];
  $validator = Validator::make($initialContext->dump(),
   ↪  $validatorRules);
  if ($validator->fails()) {
    return response($validator->errors()->all(), 400);
  } else {
    return \App\Components\GetAllPets::execute(new
     ↪  \App\SWSG\Params([]), $initialContext);
  }
});

Route::match(['post'], '/pets', function (Request $req) {
  $initialContext = new Ctx(['newPet' =>
   ↪  json_decode($req->getContent(), true)]);

  $validatorRules = ['newPet.name' => 'string|required',
   ↪  'newPet.tag' => 'string|nullable'];
  $validator = Validator::make($initialContext->dump(),
   ↪  $validatorRules);
  if ($validator->fails()) {
    return response($validator->errors()->all(), 400);
  } else {
    return \App\Components\AddOrEditPet::execute(new
     ↪  \App\SWSG\Params([new \App\SWSG\Variable('addOnly',
     ↪  'Boolean', true)]), $initialContext);
  }
```

```
32 });
33
34 Route::match(['get'], '/pets/{id}', function (Request $req) {
35   $initialContext = new Ctx(['id' =>
     ↪  $req->route()->parameter('id')]);
36
37   $validatorRules = ['id' => 'integer'];
38   $validator = Validator::make($initialContext->dump(),
     ↪  $validatorRules);
39   if ($validator->fails()) {
40     return response($validator->errors()->all(), 400);
41   } else {
42     return \App\Components\FindPet::execute(new
       ↪  \App\SWSG\Params([]), $initialContext);
43   }
44 });
45
46 Route::match(['put'], '/pets/{id}', function (Request $req) {
47   $initialContext = new Ctx(['id' =>
     ↪  $req->route()->parameter('id'), 'newPet' =>
     ↪  json_decode($req->getContent(), true)]);
48
49   $validatorRules = ['id' => 'integer', 'newPet.name' =>
     ↪  'string|required', 'newPet.tag' => 'string|nullable'];
50   $validator = Validator::make($initialContext->dump(),
     ↪  $validatorRules);
51   if ($validator->fails()) {
52     return response($validator->errors()->all(), 400);
53   } else {
54     return \App\Components\AddOrEditPet::execute(new
       ↪  \App\SWSG\Params([new \App\SWSG\Variable('addOnly',
       ↪  'Boolean', false)]), $initialContext);
55   }
56 });
57
58 Route::match(['delete'], '/pets/{id}', function (Request $req)
   ↪  {
59   $initialContext = new Ctx(['id' =>
     ↪  $req->route()->parameter('id')]);
60
```

```
61    $validatorRules = ['id' => 'integer'];
62    $validator = Validator::make($initialContext->dump(),
        ↪  $validatorRules);
63    if ($validator->fails()) {
64      return response($validator->errors()->all(), 400);
65    } else {
66      return \App\Components\DeletePet::execute(new
          ↪  \App\SWSG\Params([]), $initialContext);
67    }
68  });
```

Listing B.2 – Code généré pour le routeur de l'exemple du *Petstore*

Bibliographie

[1] Mathieu Abou-Aichi. *JHipster UML*. Version 2.0.3. 14 jan. 2017. URL : https://github.com/jhipster/jhipster-uml (visité le 10/08/2018) (cf. p. 64).

[2] Mathieu Abou-Aichi, Julien Dubois et Deepu K Sasidharan. *JHipster Domain Language*. 27 nov. 2017. URL : https://www.jhipster.tech/jdl/ (visité le 10/08/2018) (cf. p. 64).

[3] Christopher Allen et Julie Moronuki. *Haskell Programming from First Principles*. Gumroad, 23 sept. 2017. URL : http://haskellbook.com (visité le 03/08/2018) (cf. p. 67).

[4] Vard Antinyan et al. « Mythical Unit Test Coverage ». In : *IEEE Software* 35.3 (2018), p. 73-79 (cf. p. 66).

[5] Apiary. *API Blueprint*. 2015. URL : https://apiblueprint.org/ (visité le 04/04/2018) (cf. p. 61).

[6] Apiary. *Dredd*. 2017. URL : https://github.com/apiaryio/dredd (visité le 26/01/2018) (cf. p. 107).

[7] Sebastian Bergmann. *PHPUnit*. Version 7.0. 2018. URL : https://phpunit.de/ (visité le 15/02/2018) (cf. p. 66).

[8] Mario Luca Bernardi, Giuseppe A Di Lucca et Damiano Distante. « A Model-Driven Approach for the Fast Prototyping of Web Applications ». In : *Web Systems Evolution (WSE), 2011 13th IEEE International Symposium On*. IEEE, 2011, p. 65-74 (cf. p. 63).

[9] Mario Luca Bernardi et al. « M3D : A Tool for the Model Driven Development of Web Applications ». In : *Proceedings of the Twelfth International Workshop on Web Information and Data Management, WIDM 2012, Maui, HI, USA, November 02, 2012*. 2012, p. 73-80 (cf. p. 63).

[10] Edwin Brady. *Type-Driven Development with Idris*. Manning Publications, mar. 2017. 480 p. ISBN : 978-1-61729-302-3 (cf. p. 67).

[11] Jordi Cabot et Dimitrios S Kolovos. « Human Factors in the Adoption of Model-Driven Engineering : An Educator's Perspective ». In : *International Conference on Conceptual Modeling*. Springer. 2016, p. 207-217 (cf. p. 63).

[12] Hanyang Cao, Jean-Rémy Falleri et Xavier Blanc. « Automated Generation of REST API Specification from Plain HTML Documentation ». In : *International Conference on Service-Oriented Computing*. Springer, 2017, p. 453-461 (cf. p. 61).

[13] Roberto Chinnici et al. « Web Services Description Language (Wsdl) Version 2.0 Part 1 : Core Language ». In : *W3C recommendation* 26 (2007), p. 19 (cf. p. 60).

[14] Marco Cremaschi et Flavio De Paoli. « Toward Automatic Semantic API Descriptions to Support Services Composition ». In : *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, p. 159-167 (cf. p. 61).

[15] Julien Dubois. *JHipster*. Version 5.1.0. 12 juil. 2018. url : https://www.jhipster.tech/ (visité le 09/08/2018) (cf. p. 64).

[16] École polytechnique fédérale de Lausanne. *Scala.Js*. 2014. url : https://www.scala-js.org/ (cf. p. 122, 128).

[17] Moritz Eysholdt et Johannes Rupprecht. « Migrating a Large Modeling Environment from XML/UML to Xtext/GMF ». In : *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM. 2010, p. 97-104 (cf. p. 62).

[18] Facebook. *React*. 2013. url : https://reactjs.org/ (visité le 13/08/2018) (cf. p. 64).

[19] Roy T Fielding et Richard N Taylor. « Representational State Transfer (REST) ». In : *Architectural Styles and the Design of Network-Based Software Architectures*. T. 7. University of California, Irvine Doctoral dissertation, 2000 (cf. p. 57, 60).

[20] Cédric Fournet et Georges Gonthier. « The Join Calculus : A Language for Distributed Mobile Programming ». In : *Applied Semantics*. Springer, 2002, p. 268-332 (cf. p. 59).

[21] Xiang Fu, Tevfik Bultan et Jianwen Su. « Analysis of Interacting BPEL Web Services ». In : *In Proc. 13th Int. World Wide Web Conf.* Citeseer, 2004 (cf. p. 75).

[22] Google. *Angular*. 2011. url : https://angular.io/ (visité le 13/08/2018) (cf. p. 21, 36, 64).

[23] Roy Gronmo et al. « Model-Driven Web Services Development ». In : *E-Technology, e-Commerce and e-Service*. EEE'04. IEEE, 2004, p. 42-45 (cf. p. 60, 75).

[24] Marc J Hadley. « Web Application Description Language (WADL) ». In : (2006) (cf. p. 61).

[25] Dominique Hazaël-Massieux. *What Is a Web Application ?* 3 août 2010. URL : http://people.w3.org/~dom/archives/2010/08/what-is-a-web-application/ (visité le 09/04/2018) (cf. p. 56).
[26] John Hughes. « Why Functional Programming Matters ». In : *Computer Journal* 32.2 (1989), p. 98-107 (cf. p. 59).
[27] Internet Engineering Task Force. *JSON Schema : A Media Type for Describing JSON Documents*. 13 oct. 2016. URL : https://tools.ietf.org/html/draft-wright-json-schema-00 (visité le 26/01/2018) (cf. p. 112, 120, 158).
[28] JetBrains. *MPS : Domain-Specific Language Creator*. Version 2018.1.5. 15 juin 2018. URL : https://www.jetbrains.com/mps/ (visité le 02/08/2018) (cf. p. 62).
[29] Jean-Marc Jézéquel, Benoit Combemale et Didier Vojtisek. *Ingénierie Dirigée Par Les Modèles : Des Concepts à La Pratique...* Sous la dir. d'Ellipses. Références sciences. Ellipses, fév. 2012. 144 p. URL : https://hal.inria.fr/hal-00648489 (cf. p. 58, 62).
[30] Jukka K. Korpela et skeryl. *How Does the W3C Define a "Web Application" ?* 17 avr. 2014. URL : https://stackoverflow.com/questions/23140274/how-does-the-w3c-define-a-web-application (visité le 09/04/2018) (cf. p. 56).
[31] Bertrand Meyer. *Eiffel : The Language*. Prentice-Hall, Inc., 1992 (cf. p. 59).
[32] Ondřej Mirtes. *PHPStan*. Version 0.10.2. 22 juil. 2018. URL : https://github.com/phpstan/phpstan (visité le 02/08/2018) (cf. p. 68).
[33] *Mobile Web Application Best Practices*. Avec la coll. d'Adam Connors et Bryan Sullivan. W3C, 14 déc. 2010. URL : https://www.w3.org/TR/mwabp/#webapp-defined (visité le 09/04/2018) (cf. p. 56).
[34] OASIS WSBPEL TC. *Web Services Business Process Execution Language Version 2.0*. 11 avr. 2007. URL : http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visité le 06/04/2018) (cf. p. 62).
[35] Open API Initiative. *OpenAPI Specification*. 7 déc. 2017. URL : https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.1.md (visité le 26/01/2018) (cf. p. 25, 61, 75, 106, 113).
[36] Open API Initiative. *The Petstore Example*. Version 3.0.1. 7 déc. 2017. URL : https://github.com/OAI/OpenAPI-Specification/blob/3.0.1/examples/v3.0/petstore-expanded.yaml (visité le 26/01/2018) (cf. p. 107, 109, 140).
[37] Taylor Otwel. *Laravel*. 2016. URL : https://laravel.com/ (visité le 26/01/2018) (cf. p. 36, 66, 125).
[38] Pivotal. *Spring Boot*. 2013. URL : https://spring.io/projects/spring-boot (visité le 13/08/2018) (cf. p. 64).

[39] Jack Pugaczewski et al. « Software Engineering Methodology for Development of APIs for Network Management Using the MEF LSO Framework ». In : *IEEE Communications Standards* 1.1 (2017), p. 92-96 (cf. p. 61).

[40] RAML Workgroup. *RAML*. 2016. url : https://raml.org/ (visité le 26/01/2018) (cf. p. 61, 75).

[41] Jérôme Rocheteau et David Sferruzza. « Reifier : Model-Driven Engineering of Component-Based and Service-Oriented JEE Applications ». In : ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. Saint-Malo, France, 5 oct. 2016 (cf. p. 25, 92).

[42] Deepu K Sasidharan. *JDL-Studio*. 2017. url : https://github.com/jhipster/jdl-studio (visité le 10/08/2018) (cf. p. 64).

[43] Simon Schwichtenberg, Christian Gerth et Gregor Engels. « From Open API to Semantic Specifications and Code Adapters ». In : *Web Services (ICWS), 2017 IEEE International Conference On*. IEEE, 2017, p. 484-491 (cf. p. 61).

[44] Manuel Serrano, Erick Gallesio et Florian Loitsch. « Hop : A Language for Programming the Web 2. 0 ». In : *OOPSLA Companion*. 2006, p. 975-985 (cf. p. 60).

[45] Manuel Serrano et Vincent Prunet. « A Glimpse of Hopjs ». In : *ACM SIGPLAN Notices*. T. 51. 9. ACM. 2016, p. 180-192 (cf. p. 60).

[46] David Sferruzza. *Safe Web Services Generator*. Startup Palace. 2017. url : https://gitlab.startup-palace.com/research/swsg (visité le 26/01/2018) (cf. p. 120, 129).

[47] David Sferruzza. *Specification of SWSG Extensions for OpenAPI*. 2018. url : https://gitlab.startup-palace.com/research/swsg/blob/master/openapi-extensions-specification/1.0.0.md (visité le 17/04/2018) (cf. p. 113).

[48] David Sferruzza. « Top-Down Model-Driven Engineering of Web Services from Extended OpenAPI Models ». In : *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France : ACM, 3 sept. 2018, p. 940-943. isbn : 978-1-4503-5937-5. doi : 10.1145/3238147.3241536. url : https://hal.archives-ouvertes.fr/hal-01861990 (cf. p. 25).

[49] David Sferruzza et al. « A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models ». In : *International Conference on Model-Driven Engineering and Software Development*. Version étendue de : A Model-Driven Method for Fast Building Consistent Web Services in Practice. Springer, 4 avr. 2018 (cf. p. 25).

[50] David Sferruzza et al. « A Model-Driven Method for Fast Building Consistent Web Services in Practice ». In : 6th International Conference on

Model-Driven Engineering and Software Development. Funchal, Madeira, Portugal, 23 jan. 2018. URL : https://hal.archives-ouvertes.fr/hal-01654287 (cf. p. 25).

[51] David Sferruzza et al. « A Model-Driven Method for Fast Building Consistent Web Services in Practice ». 13 juin 2018. URL : https://afadl2018.ls2n.fr/ (visité le 11/05/2018) (cf. p. 25).

[52] David Sferruzza et al. « Extending OpenAPI 3.0 to Build Web Services from Their Specification ». In : 14th International Conference on Web Information Systems and Technologies. Seville, Spain, 19 sept. 2018. URL : https://hal.archives-ouvertes.fr/hal-01868498 (cf. p. 25).

[53] SmartBear Software. *Swagger Code Generator*. Version 3.0.0-rc1. 29 mai 2018. URL : https://github.com/swagger-api/swagger-codegen/ (visité le 10/06/2018) (cf. p. 110).

[54] SmartBear Software. *Swagger Editor*. 2018. URL : https://github.com/swagger-api/swagger-editor (visité le 26/01/2018) (cf. p. 107, 128, 160).

[55] SmartBear Software. *Swagger UI*. 2018. URL : https://github.com/swagger-api/swagger-UI (visité le 26/01/2018) (cf. p. 107, 128).

[56] Stack Overflow. *Developer Survey Results 2018*. 2018. URL : https://insights.stackoverflow.com/survey/2018/ (visité le 07/04/2018) (cf. p. 59).

[57] The Eclipse Foundation. *Eclipse Modeling Framework*. 2002. URL : https://www.eclipse.org/modeling/emf/ (visité le 02/08/2018) (cf. p. 62).

[58] The Eclipse Foundation. *Eclipse Xtext*. Version 2.10.0. 25 mai 2016. URL : https://www.eclipse.org/Xtext/ (visité le 02/08/2018) (cf. p. 62).

[59] The Eclipse Foundation. *Graphical Modeling Framework*. Version 3.2.1. 28 sept. 2014. URL : https://www.eclipse.org/modeling/gmp/ (visité le 02/08/2018) (cf. p. 62).

[60] The PHP Group. *PHP*. 2016. URL : https://php.net/ (visité le 26/01/2018) (cf. p. 36, 68, 110, 125).

[61] Romanos Tsouroplis et al. « Community-Based API Builder to Manage APIs and Their Connections with Cloud-Based Services. » In : *CAiSE Forum*. 2015, p. 17-23 (cf. p. 61).

[62] Wil M.P. van der Aalst, Maja Pesic et Helen Schonenberg. « Declarative Workflows : Balancing between Flexibility and Support ». In : *Computer Science-Research and Development* 23.2 (2009), p. 99-113 (cf. p. 57, 63).

[63] Martijn van der Lee. *PHPSwaggerGen*. 2017. URL : https://github.com/vanderlee/PHPSwaggerGen (visité le 26/01/2018) (cf. p. 107, 110).

[64] W3C. *SOAP Specifications*. 27 avr. 2007. URL : https://www.w3.org/TR/soap/ (visité le 18/04/2018) (cf. p. 56).

[65] Philip Wadler. « The Essence of Functional Programming ». In : *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1992, p. 1-14 (cf. p. 86).

[66] Egon Willighagen et Jonathan Mélius. « Automatic OpenAPI to Bio.Tools Conversion ». In : *bioRxiv* (2017). doi : 10.1101/170274 (cf. p. 61).

Glossaire

API Application Programming Interface. 21, 22, 33, 34, 75, 105, 106, 122, 128

AST Abstract Syntax Tree. 131

bibliothèque Ensemble de fonctions prêtes à être utilisées par des programmes. 21, 36, 37, 46, 48, 49, 59, 121, 122, 128, 131, 154, 159

BNF Backus-Naur Form. 75

boilerplate Sections de code qui doivent être incluses dans de nombreux endroits d'un programme avec peu ou pas d'altérations. 109

BPEL Business Process Execution Language. 62, 63, 69, 161

BPMN Business Process Model and Notation. 63

CIFRE Convention Industrielle de Formation par la REcherche. 17

dette technique Métaphore avec le concept de dette financière qui désigne ici le coût supplémentaire engendré par une absence d'optimisation du code. 22

DSL Domain Specific Language. 57, 58, 62

ECMAScript Langage de programmation de type script standardisé par Ecma International. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages de script, notamment JavaScript. 21

EMF Eclipse Modeling Framework. 62, 63, 69, 70, 153

framework Ensemble cohérent de composants logiciels structurels, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel. 21, 36, 59, 62, 66, 67, 109, 125, 138, 146, 151, 155

GMF Graphical Modeling Framework. 62

HTTP Hypertext Transfert Protocol. 19, 20, 21, 33, 34, 39, 40, 47, 48, 56, 60, 75, 82, 83, 84, 85, 86, 88, 89, 94, 105, 106, 108, 134, 135, 136, 140, 158, 161

IDM Ingénierie Dirigée par les Modèles. 24, 35, 58, 61, 62, 63, 64, 106, 119, 128, 153

IETF Internet Engineering Task Force. 21

IHM Interface homme-machine. 31, 32, 33, 34

J2EE Java 2 Platform, Enterprise Edition. 63

JAR Format de fichier utilisé pour stocker les définitions des classes, ainsi que des métadonnées, constituant l'ensemble d'un programme Java. 122

JDL JHipster Domain Language. 64

JSF JavaServer Faces. 63

JSON JavaScript Object Notation. 40, 140

MPS Meta Programming System. 62

MVP Minimum Viable Product. 22, 23, 24, 133, 147, 151

PME Petite et Moyenne Entreprise. 17

RAML RESTful API Modeling Language. 61

requête préparée Fonctionnalité des SGBD permettant d'analyser une requête SQL une seule fois mais de l'exécuter plusieurs fois avec des paramètres différents. Les requêtes préparées sont aussi un moyen efficace de lutter contre les attaques par injection SQL. 42

REST REpresentational State Transfer. 56, 57, 60, 61

SGBD Système de Gestion de Base de Données. 20, 42, 180

transparence référentielle Propriété des expressions d'un langage de programmation qui implique qu'une expression peut être remplacée par son résultat sans changer le comportement du programme. 59

UML Unified Modeling Language. 60, 63, 64, 75

URL Uniform Resource Locator. 39, 46, 49, 51, 56, 82, 83, 85

UX designer Designeur travaillant pour améliorer l'expérience utilisateur d'un produit en le rendant plus utilisable, accessible et agréable à utiliser. 31

W3C World Wide Web Consortium. 21, 55, 60, 61

WADL Web Application Description Language. 61, 69

webdesigner Designer spécialisé dans la conception et le maquettage d'interfaces utilsateur destinées au web. 31, 32, 33

WHATWG Web Hypertext Application Technology Working Group. 21

wireframe Maquette fonctionnelle d'une interface utilisateur. 31

WSDL Web Service Description Language. 60, 61, 69

YAML Langage de sérialisation de données conçu pour être lu et écrit par des humains. 107

Index

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Plateforme extensible de modélisation et de construction d'applications web correctes et évolutives, avec hypothèse de variabilité

Mots clés : Génie logiciel, Services web, Ingénierie Dirigée par les Modèles, Vérification formelle, Génération de code, OpenAPI 3.0

Résumé : De nombreuses sociétés œuvrant dans le logiciel dépendent des technologies web pour tester des hypothèses de marché et ainsi développer des entreprises viables. Elles ont souvent besoin de construire rapidement des services web qui sont au cœur de leurs « Minimum Viable Products » (MVP). Dans ce contexte, la construction des services web doit permettre la variabilité des technologies et des spécifications, et favoriser la productivité des développeurs. Même si de nombreuses solutions existent pour développer des services web, le contexte industriel manque de techniques assurant une construction aisée et flexible, un bon fonctionnement et une facilité de maintenance des services web, tout en étant abordables par des développeurs généralistes. Dans le but de réduire ces limitations, nous proposons une méthode de développement de services web basée sur l'Ingénierie Dirigée par les Modèles et adaptée pour (i) le prototypage rapide, (ii) la vérification de modèle, (iii) la compatibilité avec les langages de programmation classiques et (iv) l'alignement automatique entre documentation et implémentation. Cette méthode se base sur un méta-modèle volontairement minimaliste, qui est accompagné de règles de cohérence sémantique, et dont les modèles peuvent être dérivés de modèles OpenAPI 3.0 étendus. Nous fournissons également un outil, SWSG, qui automatise une grande partie de ce processus. SWSG permet notamment de générer le code des services web de manière à l'intégrer à une application utilisant le framework PHP Laravel. Enfin, nous évaluons notre approche au travers de cas d'étude.

Title: Towards an extensible framework for modelling and implementing correct and evolutive web applications, under variability hypothesis

Keywords: Software Engineering, Web Services, Model-Driven Engineering, Formal Verification, Code Generation, OpenAPI 3.0

Abstract: Lots of software companies rely on web technologies to test market hypotheses and thereby develop viable businesses. They often need to be able to quickly build web services that are at the core of their "Minimum Viable Products" (MVP). In this context, building web services must allow the variability of technologies and specifications and improve developers' efficiency. Even if numerous approaches to build web services exist, the industrial context lacks methods that ensure an easy and flexible building, a proper functioning and a good maintainability of the web services, while being accessible by generalist developers. For the purpose of reducing these limitations, we propose a method based on Model-Driven Engineering (MDE) to develop web services. It focuses on (i) rapid prototyping, (ii) model verification, (iii) compatibility with common programming languages and (iv) alignment between documentation and implementation. This method is based on a voluntarily minimalist meta-model, along with rules for semantic consistency. Models can be derived from extended OpenAPI 3.0 models. We also provide a tool, SWSG, that automatize parts of this process. Especially, SWSG allows to generate code in order to integrate the described web services inside an application made with the Laravel PHP framework. Finally, the whole approach is evaluated through case studies.