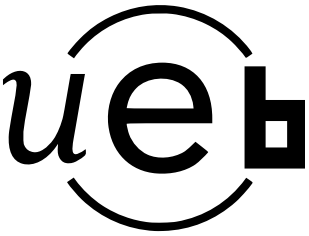
N^o d'ordre : 00000 ANNÉE 2014



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Julien RICHARD-FOY

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires ISTIC

Ingénierie des applications Web : réduire la complexité sans diminuer le contrôle

Thèse soutenue à Rennes le 9 décembre 2014

devant le jury composé de :

Lionel SEINTURIER
Professeur à l'université de Lille 1 / *Rapporteur*

Manuel SERRANO
Directeur de recherche à Sophia Antipolis / *Rapporteur*

Erwan LOISANT
Consultant à Zengularity / *Examinateur*

François BODIN
Professeur à l'IRISA / *Examinateur*

Jean-Marc JÉZÉQUEL
Directeur de l'IRISA / *Directeur de thèse*

Olivier BARAIS
Maître de conférence à l'IRISA / *Co-directeur de thèse*

Il semble que la perfection soit atteinte, non quand il n'y a plus rien à ajouter mais quand il n'y a plus rien à retrancher

par Antoine de Saint-Exupéry

Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

par Benjamin Pierce

Remerciements

Je remercie Lionel Seinturier, Professeur à l'université de Lille 1, et Manuel Serrano, Directeur de recherche à Sophia Antipolis, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Erwan Loisant, Consultant à Zengularity, et François Bodin, Professeur à l'IRISA, d'avoir bien voulu juger ce travail.

Je remercie enfin Olivier Barais et Jean-Marc Jézéquel, qui ont dirigé ma thèse.

Table des matières

Table des matières 1

Introduction 5

I Contexte et état de l'art 9

1 Contexte et portée de la thèse 11
1.1 Architecture des applications Web 11
1.2 Partage de code entre les parties client et serveur d'une application Web 13
1.3 Sûreté du typage 14
1.4 Portée de la thèse 17

2 État de l'art 19
2.1 Styles d'architecture pour les applications Web résilientes 19
2.1.1 Architecture des applications Web interactives 20
2.1.2 Prise en charge du mode déconnecté 23
2.1.3 Travail coopératif assisté par ordinateur 24
2.1.4 *Event sourcing* 26
2.2 Partage de code 27
2.2.1 Langages dédiés implémentés comme une bibliothèque 27
2.2.2 Langages lourds 29
2.2.3 Ingénierie dirigée par les modèles 30
2.2.4 Pouvoir d'expression des langages dédiés externes 31
2.2.5 Langages dédiés compilés et implémentés comme une bibliothèque 32
2.2.6 Méta-programmation et évaluation partielle 35
2.2.7 *Lightweight Modular Staging* et *js-scala* 35
2.3 Sûreté du typage 39

II Contributions 41

3 Partage de code 43

3.1 API des sélécteurs . . . 43
3.2 Manipulation de valeurs optionnelles . . . 46
3.3 Définition de fragments de HTML . . . 48
3.3.1 Représentation statiquement typée de chaînes de caractères contenant des expressions JavaScript . . . 50
3.4 Validation . . . 52
3.4.1 Objectifs . . . 52
3.4.2 Expérience . . . 53
3.4.3 Discussion . . . 57
3.5 Conclusion . . . 58

4 Sûreté du typage **61**
4.1 Inventaire des fonctions problématiques . . . 61
4.1.1 `createElement` . . . 61
4.1.2 `getElementsByTagName` . . . 62
4.1.3 `addEventListener` . . . 63
4.1.4 Discussion . . . 65
4.2 Solutions bien typés et conservant le même pouvoir d'expression . . . 65
4.2.1 Généralisation des solutions existantes . . . 65
4.2.2 Utilisation des types paramétrés . . . 66
4.2.3 Utilisation des types dépendants . . . 67
4.3 Discussion . . . 70
4.3.1 Analyse . . . 70
4.3.2 Limitations . . . 71
4.4 Conclusion . . . 71

5 Un style d'architecture pour les applications Web résilientes **73**
5.1 Présentation générale . . . 73
5.2 Fonctionnement du système de synchronisation . . . 74
5.2.1 Scénario mono-utilisateur . . . 75
5.2.2 Scénario multi-utilisateur . . . 76
5.3 Généralisation du composant de synchronisation . . . 77
5.4 Évolution : fonctionnement complètement hors-ligne . . . 77
5.5 Validation . . . 78
5.5.1 Objectifs . . . 78
5.5.2 Expérience . . . 79
5.5.3 Discussion . . . 79
5.6 Discussion . . . 81

III Conclusion et perspectives **83**

6 Conclusion et perspectives **85**
6.1 Conclusion . . . 85

6.2 Perspectives 86

Bibliographie **94**

Table des figures **95**

Introduction

En automatisant certaines tâches et traitements d'informations, les technologies de l'information et de la communication (TIC) permettent de réaliser des économies considérables sur nos activités. Elles sont une importante source d'activités économiques, parfois qualifiée de « troisième révolution industrielle »[Rif08].

Illustrons ce propos par un exemple concret : le co-voiturage. Ce système vise à diminuer le nombre de véhicules sur les routes, et, par voie de conséquence, leurs émissions de gaz à effet de serre, en maximisant leur taux d'occupation par la mutualisation des voyages de différentes personnes. Le co-voiturage nécessite la collecte d'informations sur les trajets proposés par les conducteurs et les passagers, leur mise en relation ainsi qu'un processus de paiement. Les outils numériques, en automatisant ces tâches et traitements d'informations, facilitent considérablement la pratique du co-voiturage. En effet, utiliser, par exemple, des petites annonces dans la presse pour informer les passagers potentiels des trajets proposés par un conducteur, puis utiliser des courriers postaux pour organiser le voyage entre les personnes intéressées serait beaucoup plus contraignant (par exemple, chercher un trajet nécessiterait de parcourir manuellement l'intégralité des annonces, et les délais de mise en relation entre les personnes leur imposeraient de s'organiser très en avance). L'essor des TIC a permis au leader français du co-voiturage, BlaBlaCar, de rendre service à plus de 6 millions d'usagers.

Ces outils numériques sont utilisés depuis des machines telles qu'un ordinateur, un téléphone ou une tablette. Les échanges d'information entre ces machines nécessitent qu'elles soient reliées en réseau. Dans ce contexte, le Web offre un environnement propice à la mise en place d'outils numériques : ceux-ci sont hébergés par des *serveurs*, qui centralisent les informations et coordonnent les utilisateurs. Ces derniers accèdent aux outils depuis leurs terminaux *clients*, quels qu'ils soient (ordinateur, téléphone ou tablette), en utilisant un navigateur Web, sans aucune étape d'installation. Lorsqu'un utilisateur interagit avec un tel outil, les actions qu'il effectue depuis sa machine sont éventuellement transmises au serveur. Par exemple, lorsqu'un passager décrit, depuis son ordinateur personnel, le voyage qu'il souhaite effectuer, cette information est transmise au serveur hébergeant l'application de co-voiturage qui lui indique, en retour, quels conducteurs prévoient de faire un voyage similaire.

Toutefois, la réalisation de ces applications Web présente des difficultés pour les développeurs. La principale difficulté vient de la *distance* entre les postes client et serveur.

D'une part, la distance physique (ou distance *matérielle*) entre les machines nécessite qu'une connexion réseau soit toujours établie entre elles pour que l'application fonctionne correctement. Cela pose plusieurs problèmes : comment gérer les effets de latence lors des échanges d'information entre machines ? Comment assurer une qualité de service même lorsque la connexion réseau est interrompue ? Comment choisir quelle part de l'application s'exécute sur le client et quelle part s'exécute sur le serveur ? Des travaux ont proposé de prendre en charge le mode déconnecté en s'appuyant sur un système de persistence des données, côté client, capable de se synchroniser automatiquement lorsque la connexion réseau le permet [KLYY12, MGPW13]. Cependant, cette approche n'offre aucun moyen de régler les conflits dus à des modifications concurrentes des données. La gestion de ces conflits, quant à elle, a fait l'objet de nombreuses recherches, et un algorithme adapté aux spécificités du Web a été proposé en 2011 [SLLG11]. Bien que ces technologies apportent isolément des éléments de solutions, elles ne fournissent pas aux développeurs un cadre architectural pour que leurs applications soient, par construction, résilientes aux pannes de réseau.

D'autre part, l'environnement d'exécution est différent entre les clients et serveurs, produisant une distance *logicielle*. En effet, côté client c'est un navigateur Web qui exécute le code du programme, lequel est généralement écrit en JavaScript. De l'autre côté, c'est un serveur Web, éventuellement implémenté avec un autre langage que JavaScript, qui traite les requêtes des clients. Certains aspects d'une application Web peuvent être communs aux parties client et serveur, par exemple la construction de morceaux de pages Web, la validation des données saisies dans un formulaire ou certains calculs métier. En effet, construire les pages Web depuis le serveur facilite le référencement de leur contenu par les moteurs de recherche, mais les construire depuis le client produit une meilleure expérience utilisateur. De même, effectuer la validation des données des formulaires depuis le client offre une meilleure expérience utilisateur mais cette validation doit être effectuée de nouveau sur le serveur pour des raisons de sécurité. Cependant les langages de programmation et, surtout, les interfaces de programmation (API) étant différentes entre les deux parties, cela entrave les possibilités de réutilisation de code entre elles. Par conséquent, les développeurs dupliquent des concepts entre les parties client et serveur, bien que cela soit plus coûteux à maintenir. Certaines technologies permettent le partage de code entre clients et serveurs [JW07, Can08], mais ce code partagé n'est pas capable de tirer parti des spécificités des environnements client et serveur, il nécessite une couche d'adaptation coûteuse en performance. En intégrant directement dans le langage les aspects communs aux parties client et serveur [RT10], le compilateur peut produire du code exécutable tirant parti des spécifités des plateformes cibles, mais cette approche demande un effort d'implémentation beaucoup plus important.

Pour faciliter le processus d'ingénierie des applications Web, un challenge consiste à raccourcir cette distance, tant matérielle que logicielle. Cela signifie offrir un modèle de développement permettant aux développeurs de ne pas avoir à gérer ces difficultés sans pour autant masquer l'aspect distribué des architectures Web ni les spécificités de chaque environnement d'exécution. En effet, masquer complètement l'aspect distribué d'un système diminuerait la capacité à bénéficier des avantages de cette ar-

chitecture [GF99] (e.g. ubiquité des données). De même, masquer les spécificités des environnements client et serveur diminuerait les performances d'exécution [RFBJ13a].

Les travaux de cette thèse cherchent à raccourcir cette distance entre les parties client et serveur des applications Web, tout en préservant la capacité à tirer parti de cette distance, c'est-à-dire en donnant autant de contrôle aux développeurs.

Le premier axe de travail est la réduction de la distance matérielle : comment concevoir une application Web de telle sorte qu'elle fonctionne aussi bien, quelle que soit la latence due à la distance entre les machines reliées en réseau, voire quand la connexion est rompue ? Nous répondons à cette question en proposant un modèle d'architecture découpant le code de l'application de façon à isoler la préoccupation de communication client-serveur et la logique métier. Grâce à ce modèle d'architecture nous avons été capables de capitaliser la logique de synchronisation entre clients et serveurs sous forme d'une bibliothèque réutilisable.

Le deuxième axe de travail complète le premier. En effet, notre modèle d'architecture conduit à partager certains concepts côtés client et serveur. En particulier, les types de données décrivant les actions du système existent des deux côtés. Le besoin de partager des concepts entre clients et serveurs n'est, cependant, pas caractéristique de notre modèle d'architecture, et l'on observe plusieurs initiatives visant à faciliter la réutilisation de code entre clients et serveurs. Notre solution réutilise la bibliothèque js-scala, qui fournit un mécanisme d'évaluation retardée permettant à un même programme d'être évalué dans un navigateur Web ou dans l'environnement d'exécution du serveur (une JVM) [KARO12]. Notre contribution consiste en un ensemble de bibliothèques bâties sur js-scala, fournissant des API de haut niveau pour traiter certains aspects des applications Web, et produisant du code tirant parti des spécificités des environnements client et serveur. En particulier, $DomOps$ pour la manipulation du DOM d'une page Web et $forest$ pour la définition de fragments de pages Web. En pratique, nous avons observé que la taille du code écrit avec nos outils est du même ordre de grandeur que celle d'un code utilisant des bibliothèques haut-niveau existantes et 35% à 50% plus petite que celle d'un code bas-niveau, mais que les performances d'exécution du programme obtenu sont du même ordre de grandeur que celles d'un code bas-niveau et 39% à 972% meilleures qu'un code haut-niveau.

Nos travaux sont basés sur le langage Scala, qui est statiquement typé. Bien que le typage dynamique apporte aussi des avantages, des travaux ont rapporté que le typage statique peut jouer un rôle important pour faciliter la gestion de la complexité d'un programme [KHR+12, MHR+]. Cependant, l'utilisation d'un langage statiquement typé pour écrire du code s'exécutant sur les navigateurs Web a soulevé un nouveau problème : ce code fait appel aux API du navigateur, dont les fonctions sont destinées à être utilisées depuis le langage dynamiquement typé JavaScript, et auxquelles il peut être difficile de faire correspondre une signature typée sans perte d'information. En effet, certaines fonctions fondamentales de l'API du navigateur exposent des types de données trop généraux, contraignant le développeur à écrire des conversions descendantes de type ($downcasting$), contournant ainsi le système de vérification de types, donc le conduisant éventuellement à écrire du code incorrect. En nous appuyant sur les mécanismes de paramètres de type ou de types membres des langages de pro-

grammation, notre troisième contribution est une façon d'exposer des signatures de type pour ces fonctions, de telle sorte qu'elles fournissent suffisamment d'information pour éviter aux développeurs de réaliser des conversions de types, tout en conservant le même pouvoir d'expression. Comparée aux autres approches utilisées pour exposer l'API du navigateur dans un langage statiquement typé, notre approche permet de réduire le nombre de fonctions tout en préservant le même pouvoir d'expression.

Ainsi, nos trois contributions permettent de diminuer la complexité du développement d'applications Web tout en donnant autant de contrôle aux développeurs.

Première partie

Contexte et état de l'art

Chapitre 1

Contexte et portée de la thèse

Ce chapitre détaille les problèmes d'ingénierie des applications Web dus aux distances matérielle et logicielle entre les postes client et serveur.

Nous présentons d'abord les enjeux architecturaux découlant de la distance matérielle entre clients et serveurs, puis les obstacles au partage de code entre les environnements client et serveur à cause de leur distance logicielle. Ensuite, nous soulignons les difficultés à exposer l'interface de programmation des navigateurs Web dans un langage de programmation statiquement typé. Enfin, nous terminons ce chapitre en exposant les questions de recherche adressées par nos travaux.

1.1 Architecture des applications Web

L'architecture d'un système peut supprimer certains problèmes techniques ou isoler certaines préoccupations, rendant le système moins complexe, donc plus facile à comprendre et faire évoluer. Dans le cas des applications Web, nous distinguons deux niveaux d'architecture : matériel et logiciel.

L'architecture matérielle correspond à la disposition matérielle des machines reliées entre elles. Dans le cas des applications Web, il s'agit d'une architecture centralisée : plusieurs clients se connecte à un même serveur pour accéder à ses services. L'architecture logicielle correspond à l'organisation du code d'une application donnée, elle décrit la façon dont les différentes parties du code s'assemblent et leur responsabilités respectives.

L'architecture matérielle du Web a pour avantage de permettre un accès ubiquitaire aux données : celles-ci étant stockées sur un serveur distant, l'utilisateur peut les consulter depuis plusieurs postes clients (par exemple depuis son ordinateur de bureau et son téléphone). En outre, si plusieurs utilisateurs accèdent à un même document, ils peuvent collaborer bien plus facilement que s'ils devaient s'envoyer, à tour de rôle, une copie modifiée d'un même document. Enfin, un dernier avantage est que le stockage des données est mutualisé : les postes clients n'ont pas besoin de disposer d'une grande quantité de mémoire pour utiliser une application.

Toutefois, les architectures Web ont également certains désavantages. Par exemple,

les utilisateurs peuvent percevoir de la latence due aux échanges d'information entre les postes clients et serveurs. De plus, la nature centralisée des architectures Web peut poser des problèmes de gestion de montée en charge pour les serveurs, si le trafic est irrégulier et intense. Enfin, une application Web ne peut être utilisée que si une connexion réseau est établie entre les postes client et serveur, or certains appareils (par exemple les téléphones mobiles) ne disposent que d'une connexion intermittente.

L'objectif de cette thèse est de proposer des moyens logiciels pour construire des applications Web bénéficiant des avantages présentés précédemment sans souffrir des inconvénients dus à la distance matérielle entre clients et serveurs.

En particulier, un aspect qui nous préoccupe est la *résilience* aux aléas du réseau : comment faire pour que la qualité de l'expérience utilisateur soit le moins possible affectée par une coupure de la connexion réseau ? Une telle coupure rend impossibles les échanges d'information entre le client et le serveur. Par conséquent, la préservation de l'expérience utilisateur n'est possible que si le client a déjà téléchargé suffisamment d'information pour pourvoir les fonctionnalités utilisées.

Illustrons cela par un cas pratique. Imaginons une application permettant de gérer une liste de tâches à effectuer : les utilisateurs peuvent ajouter des tâches et indiquer quand elles sont réalisées. Le fonctionnement traditionnel des applications Web est le suivant : lorsqu'un utilisateur crée une nouvelle tâche ou indique qu'une tâche est réalisée, cette demande est transmise au serveur, celui-ci met à jour l'état du système en lui ajoutant la tâche créée, puis il envoie au client les informations lui permettant de mettre à jour l'interface utilisateur de façon à montrer à l'utilisateur qu'une tâche a été ajoutée. Maintenir les fonctionnalités de cette application lorsque le client est déconnecté du serveur nécessite de faire évoluer et afficher l'état du système localement, sur le poste client, sans solliciter le serveur.

Cependant, le fait de permettre à l'utilisateur d'utiliser le système en étant déconnecté induit un nouveau problème : l'état du système sur son poste client diverge de l'état du système sur le serveur, il devient alors nécessaire de synchroniser l'état du système du client vers le serveur, lorsque la connexion est de nouveau établie. Dans le cas d'un système permettant le travail collaboratif, la synchronisation doit prendre en compte le fait que deux utilisateurs peuvent effectuer des modifications concurrentes et parfois conflictuelles sur une même ressource.

Reprenons notre exemple d'application de gestion de tâches. Lorsque la connexion est de nouveau établie, le client doit transmettre au serveur le nouvel état du système, tel qu'il a évolué suite à son utilisation. Cependant, si deux utilisateurs, Alice et Bob, utilisent le système en étant déconnecté, ils partent du même état et le font chacun évoluer différemment. Par exemple, Alice peut créer une nouvelle tâche, et Bob indiquer qu'une autre tâche a été réalisée. Quand ils se reconnectent, le processus de synchronisation doit faire converger les trois états (celui du serveur et ceux des clients d'Alice et Bob). En outre, si, en mode déconnecté, Alice supprime une tâche et Bob indique que cette même tâche a été réalisée, puis qu'Alice se reconnecte et se synchronise avec le serveur avant Bob, quand Bob se synchronise avec le serveur, celui-ci doit gérer le fait que la tâche que Bob a indiqué être réalisée a entretemps été supprimée par Alice.

Plusieurs approches sont possibles dans ce cas. La plus simple pour les développeurs est aussi la moins satisfaisante pour les utilisateurs, elle consiste à ne rien faire de particulier sur le serveur en cas de modifications conflictuelles. Dans notre cas, cela peut se traduire par la levée d'une erreur lors de la synchronisation de Bob, car il demande au serveur de modifier une tâche qui n'existe pas. Cette approche conduit, dans le meilleur des cas, à l'affichage d'erreurs chez les clients, mais elle peut également ne pas détecter les conflits et faire converger le système vers un état qui ne reflète pas l'intention initiale des utilisateurs. Par exemple, si, en mode déconnecté, Alice demande de supprimer toutes les tâches réalisées pendant que Bob indique qu'une nouvelle tâche a été réalisée. Si Bob se synchronise avant Alice, la synchronisation de cette dernière peut conduire à ne pas supprimer la tâche nouvellement indiquée comme étant réalisée, alors qu'Alice avait initialement l'intention de supprimer toutes les tâches réalisées.

Nous ne souhaitons par remettre en cause l'architecture matérielle des applications Web, car elle fournit de nombreux avantages. Notre objectif est de trouver une solution au niveau logiciel. Un style d'architecture pour construire des applications Web résilientes est un style d'architecture qui isole la préoccupation de résilience ou réduit le travail à effectuer par les développeurs pour parvenir à construire des applications résilientes.

Une conséquence du fait que les côtés client et serveur soient tous les deux amenés à exécuter les traitements métiers (côté client, pour obtenir la propriété de résilience, côté serveur pour maintenir l'état de référence du système) est que la logique métier risque de se trouver dupliquée dans le code des programmes client et serveur. Cette duplication n'est pas souhaitable par l'ingénieur logiciel qui doit alors maintenir deux bases de code cohérentes entre elles. Un moyen d'éviter la duplication consiste à partager le même code. Cette idée est détaillée dans la section suivante.

1.2 Partage de code entre les parties client et serveur d'une application Web

En fait, plusieurs aspects d'une application Web gagnent à être partagés entre clients et serveurs. Par exemple, vérifier, côté client, que les données saisies dans un formulaire sont valides contribue à améliorer l'expérience utilisateur car celui-ci n'a pas besoin d'attendre de soumettre l'intégralité d'un formulaire pour voir apparaître les erreurs. Cependant, ces règles de validation doivent également être appliquées côté serveur, au cas où un utilisateur mal intentionné contournerait la validation côté client. De même, la construction des pages HTML a intérêt à être réalisée côté serveur, pour obtenir un meilleur référencement du contenu par les moteurs de recherche, mais elle a aussi intérêt à être réalisée côté client, là encore, pour améliorer l'expérience utilisateur, en n'actualisant qu'une partie du document plutôt qu'en rechargeant la page entière. En outre, il est intéressant de gérer la navigation (la relation entre une URL et le contenu de la page correspondante) côté serveur, toujours pour des raisons de référencement par les moteurs de recherche, mais également côté client,

encore pour des raisons d'exérience utilisateur. Enfin, certains calculs effectués sur le domaine métier d'un système peuvent également être exécutés côtés client et serveur.

Tous ces exemples illustrent un besoin général, dans les applications Web, d'être capable de partager des concepts entre clients et serveurs. Cependant, ce besoin est difficile à satisfaire car les environnements d'exécution des programmes client et serveur ne sont pas les mêmes. Premièrement, les langages de programmation peuvent être différents : côté client le langage supporté par tous les navigateurs est JavaScript, tandis que côté serveur le choix du langage peut être déterminé par des raisons techniques (inter-opérabilité avec l'existant, compétences des collaborateurs, etc.). Mais surtout, le contexte d'exécution n'est pas le même. Côté client, le programme s'exécute dans un navigateur et est associé à un contexte local au client (document HTML, stockage local, etc.). Côté serveur, le programme s'exécute derrière un serveur HTTP et est associé à l'état global du système.

Illustrons ces différences par un exemple concret : la construction de fragments HTML. Côté client, celle-ci aboutit à la construction d'un fragment de DOM, c'est-à-dire un arbre d'éléments HTML sur lequel il est possible d'effectuer des manipulations (e.g. ajout, suppression, déplacement de nœuds) ou de réagir à des événements produits par l'utilisateur sur certains nœuds (e.g. clic de souris). Côté serveur, elle aboutit à un un texte représentant la structure des éléments HTML. Ces différences sont illustrés figure 1.1. Le listing 1.1 donne le code JavaScript qui permet de construire cet arbre, côté client, et le listing 1.2 donne le code Scala qui permet de construire le fragment de HTML, côté serveur. Ces deux exemples illustrent le fait que construire un même fragment de HTML, côté client ou côté serveur, ne se fait pas de la même façon.

1.3 Sûreté du typage

Nous avons réalisé nos travaux en nous appuyant sur le langage Scala, qui a la particularité d'être statiquement typé. Bien que le typage dynamique ait également ses avantages, nous observons que l'utilisation d'un langage à typage statique pour écrire la partie client d'une application Web n'est pas marginale : GWT [JW07], Dart [WL12], TypeScript [Fen12] [1], Kotlin [Kot], Opa [RT10], SharpKit [Sha], HaXe [Can08], Ocsigen [Bal06], Elm [Cza12], Idris [Bra13] ou PureScript [Pur] sont autant d'exemples de langages à typage statique utilisés pour écrire la partie client d'applications Web.

La partie client d'une application Web a pour rôle de réagir aux actions de l'utilisateur et de mettre à jour le document affiché. Cela s'effectue en utilisant des fonctions fournies par l'interface de programmation (API) du navigateur Web.

Or, cette API a été conçue pour être utilisée depuis le langage JavaScript, qui est dynamiquement typé, et il est parfois difficile de définir une signature de type correcte pour certaines fonctions.

1. Soulignons toutefois que, dans le cas de Dart et TypeScript, le système de types n'est pas aussi contraignant que les systèmes de types statiques habituels. En effet, les erreurs de typage produisent seulement des avertissements plutôt que des erreurs de compilation. Ce comportement est voulu et a vocation à faciliter une migration d'une base de code non typée vers du code typé [BAT14, EMSS].

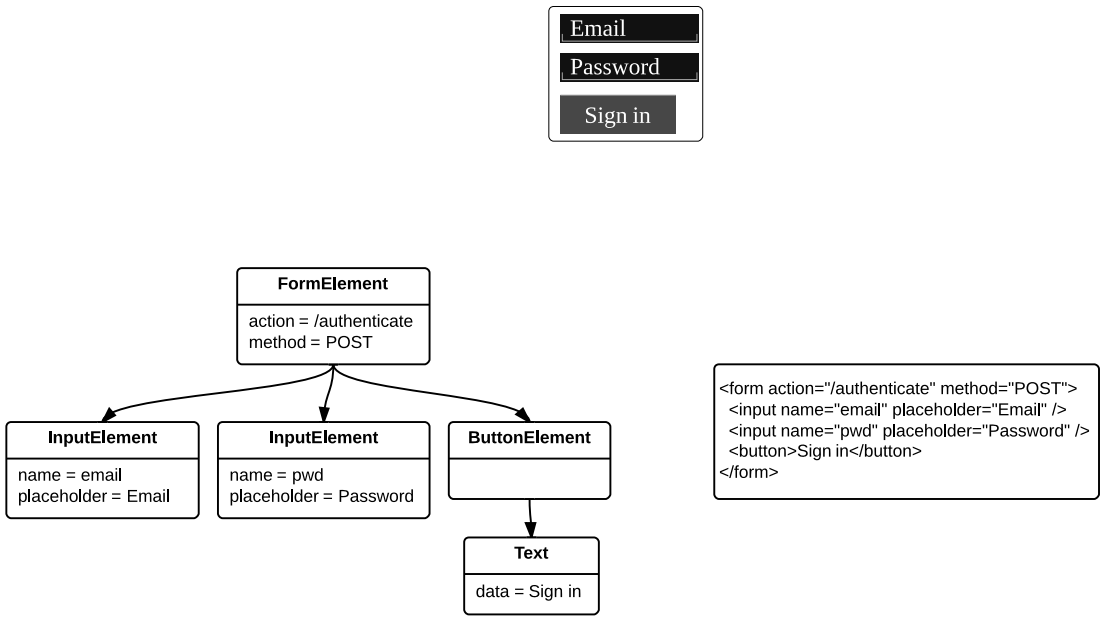


FIGURE 1.1 – Un formulaire tel qu'il est perçu par l'utilisateur (en haut), est représenté par un arbre d'éléments HTML côté client, et par un texte côté serveur.

```
var ui = function (emailValue) {
  var email = document.createElement('input');
  email.name = 'email';
  email.placeholder = 'Email';
  email.value = emailValue;
  var pwd = document.createElement('input');
  pwd.name = 'pwd';
  pwd.placeholder = 'Password';
  var button = document.createElement('button');
  button.appendChild(document.createTextNode('Sign in'));
  var form = document.createElement('form');
  form.action = '/authenticate';
  form.method = 'POST';
  form.appendChild(email);
  form.appendChild(pwd);
  form.appendChild(button);
  return form
}
```

Listing 1.1 – Définition du fragment de HTML représentant le formulaire montré dans la figure 1.1, côté client, en JavaScript : un arbre est construit progressivement, par ajout successif de nœuds.

```
def ui(email: String): Html =
  html"""
    <form action="/authenticate" method="POST">
      <input name="email" placeholder="Email" value="$email" />
      <input name="pwd" placeholder="Password" />
      <button>Sign in</button>
    </form>
  """
```

Listing 1.2 – Définition du fragment de HTML représentant le formulaire montré dans la figure 1.1, côté serveur, en Scala : une chaîne de caractères utilise une représentation textuelle du fragment de HTML. Le préfixe `html` permet une interpolation sûre de la valeur `email`

```
var img = createElement("img");
img.src = "/icon.png";
var input = createElement("input");
input.value = "foo";
```

Listing 1.3 – Utilisation de la fonction `createElement` en JavaScript

Par exemple, la fonction `createElement` a la signature de type suivante, d'après sa spécification standard [HTM] :

```
Element createElement(String name);
```

Notons que le type de retour indiqué est `Element`. Néanmoins, l'appel `createElement('input')` retourne plus précisément un nœud de type `InputElement` représentant un champ de formulaire. De même, l'appel `createElement('img')` retourne plus précisément un nœud de type `ImageElement` représentant une image.

Ces deux types, `InputElement` et `ImageElement`, n'ont pas les mêmes propriétés. Par exemple, le type `InputElement` possède une propriété `value` contenant la valeur du champ du formulaire. De même, le type `ImageElement` possède une propriété `src` contenant l'URL de l'image. Ce détail n'a aucune importance pour les développeurs utilisant le langage JavaScript mais est problématique dans un langage statiquement typé.

En effet, considérons le listing 1.3, écrit en JavaScript et créant une image et un champ de formulaire. Dans cet exemple, nous accédons à la propriété `src` de l'image et à la propriété `value` du champ de formulaire.

Dans un monde statiquement typé, ce listing, bien que correct, produirait une erreur de typage car les propriétés `src` et `value` ne sont pas définies pour le type général `Element`. Ainsi, la signature de type standard de la fonction `createElement` ne convient pas, telle quelle, aux langages statiquement typés.

Notons enfin, que le langage JavaScript permet d'écrire le listing 1.3 mais il permet également d'écrire des programmes incorrects. Par exemple, il est possible, en JavaScript, d'écrire un programme accédant à une propriété `src` d'un objet de type

`InputElement` bien que ce type ne possède pas de telle propriété.

Idéalement, on aimerait avoir une signature de type telle que le listing suivant, en Scala, compile sans problème, sauf la dernière ligne :

```
val img = createElement("img")
img.src = "/icon.png" // Ok
img.value = "foo" // Erreur: propriete 'value' non definie
```

De même, le listing suivant doit compiler, sauf la dernière ligne :

```
val input = createElement("input")
input.value = "foo" // Ok
input.src = "/icon.png" // Erreur: propriete 'src' non definie
```

Autrement dit, après avoir créé un élément, il doit être possible d'utiliser toutes ses propriétés. Par contre, l'utilisation d'une propriété non définie pour ce type d'élément doit provoquer une erreur de typage. Dès lors, quelle signature de type définir pour la fonction `createElement` ? Plus généralement, y a-t-il d'autres fonctions qui souffrent d'un problème similaire, et comment le résoudre ?

1.4 Portée de la thèse

Les sections précédentes ont soulevé plusieurs problèmes. Nos travaux s'intéressent en particulier aux questions de recherche suivantes :

RQ1 Quel style d'architecture logicielle adopter pour isoler les problèmes dus à la distance matérielle entre clients et serveurs (latence, indisponibilité de l'application en mode déconnecté) ?

RQ2 Comment partager du code entre les parties client et serveur malgré leurs différences ?

RQ3 Comment interfacer le monde dynamiquement typé des navigateurs avec un langage statiquement typé en préservant la sûreté du typage et sans perdre de pouvoir d'expression ?

Le reste de ce document est organisé comme suit. Le prochain chapitre détaille l'état de l'art relatif aux trois questions de recherches formulées. Le chapitre 3 montre comment nous avons défini des abstractions qui, bien qu'elles soient utilisables côtés client et serveur, ne souffrent pas de problèmes de performance (RQ2). Dans le chapitre 4 nous montrons qu'il est possible de définir, pour certaines fonctions de l'API du navigateur, des signatures de type garantissant une sûreté du typage et préservant le même pouvoir d'expression que l'API native (RQ3). Le chapitre 5 présente un sytle d'architecture ainsi que l'implémentation d'un composant logiciel réutilisable pour construire des applications Web résilientes (RQ1). Enfin, le chapitre 6 présente une synthèse des contributions ainsi que des perspectives de recherche.

Chapitre 2

État de l'art

Ce chapitre présente les réponses que des travaux antérieurs apportent aux questions de recherche traitées par cette thèse.

La préoccupation de résilience étant une préoccupation technique transverse, la section 2.1 s'intéresse aux styles d'architecture logicielle permettant d'isoler cette préoccupation.

La section 2.2 compare les méthodes existantes pour partager du code entre les parties client et serveur.

Enfin, la section 2.3 recense les travaux visant à interfacer les langages statiquement typés et dynamiquement typés.

2.1 Styles d'architecture pour les applications Web résilientes

L'ingénierie des applications Web a fait l'objet de nombreuses recherches. En 2002, Roy T. Fielding *et. al.* ont proposé le style d'architecture *REpresentational State Transfer* (REST) [FT02]. Ce style d'architecture définit un ensemble de contraintes sur le fonctionnement et l'interaction des services Web, de façon à minimiser les problèmes de latence tout en minimisant le couplage des services Web. Cependant, REST se limite au côté serveur des applications Web : il ne donne aucune préconisation sur la façon d'intégrer les parties client et serveur des applications Web.

En effet, les applications Web ont, entretemps, évolué. Notamment, la partie client est devenue beaucoup plus riche : elle ne se contente plus simplement d'afficher le contenu des pages et de naviguer entre les liens hypertextes, mais elle permet des interactions riches (déroulement de menus, etc.) et, surtout, une navigation asynchrone (quand l'utilisateur clique sur un lien hypertexte, le client ne remplace pas la page actuelle par la nouvelle, mais met à jour une partie du document actuel). Cette évolution se traduit par une plus grande quantité de code et de nouvelles responsabilités pour la partie client d'une application Web [Mar09].

Ainsi, une application Web comporte désormais deux applications : une application côté serveur, chargée de traiter les actions des clients et d'exposer les données du système, et une application côté client, chargée de répondre aux interactions de l'uti-

FIGURE 2.1 – Les modèles MVC (à gauche) et PAC (à droite)

lisateur, éventuellement en transmettant ses actions à l'application côté serveur, et en actualisant l'affichage en fonction de l'évolution de l'état du système. L'utilisateur final utilise l'application côté client, qui elle-même utilise l'application côté serveur.

2.1.1 Architecture des applications Web interactives

Cette section présente les styles d'architectures utilisés dans les applications interactives et leurs évolutions conduisant aux applications Web interactives.

2.1.1.1 Architecture des applications interactives

Les deux principaux styles d'architcture pour les applications interactives sont *Model-View-Controller* [KP88] et *Presentation-Abstraction-Control* [Cou87]. Dans les deux cas, les préoccupations identifiées sont la logique métier (*Model* ou *Abstraction*), l'interface utilisateur (*View* ou *Presentation*) et le lien entre les deux (*Controller* ou *Control*).

La principale différence entre les deux modèle, illustrée figure 2.1, est que, dans le cas du modèle PAC, la responsabilité de l'*Abstraction* s'arrête à la logique métier : elle ne communique pas avec l'interface utilisateur, contrairement au modèle MVC, et elle ne fait même pas l'hypothèse qu'une interface utilisateur existe. Notons également que la littérature mentionne aussi l'architecture *Model-View-Presenter* (MVP), qui est en réalité complètement équivalente au modèle PAC.

Ainsi, le modèle PAC isole mieux les différents aspects des applications interactives : la partie *Abstraction* contient la logique métier et est réutilisable dans une application non interactive. La partie *Presentation* contient tout ce qui est spécifique à la technologie d'affichage choisie. Elle n'est pas liée à une application en particulier. Enfin, la partie *Control* rend l'*Abstraction* interactive en l'affichant à l'aide d'une *Presentation* et en interprétant les actions de l'utilisateur en termes d'actions métier.

2.1.1.2 Architecture des applications collaboratives interactives

Les modèles PAC et MVC sont adaptés aux applications mono-utilisateur. Dans le cas où plusieurs utilisateurs manipulent en même temps une même ressource, de nouveaux problèmes se posent, notamment de gestion de la cohérence de l'information entre les différents utilisateurs.

Duval *et. al.* ont montré comment adapter le modèle PAC aux applications collaboratives. Ils proposent de dupliquer des instances de modèle PAC sur chaque client

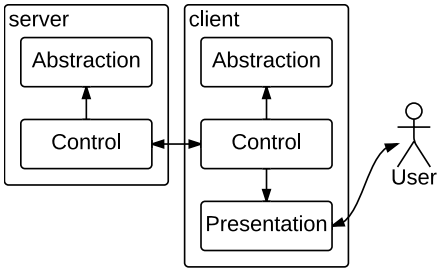


FIGURE 2.2 – Le modèle PAC-C3D

et de les faire communiquer entre elles au niveau des composants *Control*. Ces composants se voient infliger une nouvelle responsabilité : gérer la cohérence des données entre les différents clients. Cette responsabilité est coordonnée par un serveur, maintenant l'état de référence du système. La figure 2.2 illustre le modèle PAC-C3D dans le cas d'un serveur et un client. Le serveur n'a pas de composant de *Presentation* puisque son rôle se limite à de la synchronisation d'information : il reçoit les modifications effectuées par un client et les transmet aux autres clients.

Le modèle PAC-C3D a l'avantage d'être simple et bien découpé : chaque composant n'a que peu de responsabilités. Cependant, il fait l'hypothèse d'une connexion réseau fiable et ne gère pas les conflits dues à des modifications concurrentes par des utilisateurs distants.

2.1.1.3 Architecture des applications Web interactives

En 2007, Morales-Chaparro *et.al.* ont proposé un ensemble de modèles d'architecture pour les applications Web. Ces modèles sont tous des variations du modèle *Model-View-Controller* (MVC), couramment utilisé pour la conception d'applications interactives. Les variantes proposées par ces auteurs s'adaptent à l'importance de l'application côté client. Dans une première variante, ils suggèrent d'utiliser MVC côté serveur seulement. C'est-à-dire que c'est le serveur qui est responsable de la construction de la page HTML affichée par le serveur (partie *View*), d'interpréter les requêtes HTTP en termes d'actions métier (partie *Controller*) et de mettre à jour l'état du système en fonction des actions métier (partie *Model*). Dans cette variante, il n'y a pas d'application côté client, cela correspond donc au cas des applications Web d'antan. Une autre variante consiste à utiliser MVC côté client seulement. C'est-à-dire que c'est le client qui est responsable de la construction de la page HTML qu'il affiche (partie *View*), d'interpréter les interactions de l'utilisateur (e.g. clics sur des boutons) en termes d'actions métier (partie *Controller*), et de mettre à jour l'état du système en fonction des actions métier (partie *Model*). Dans leur modèle, cette dernière partie est également responsable de transmettre l'évolution de l'état du système à un *Model* analogue, côté serveur. Dans cette variante, l'application est plus maigre sur le serveur que sur le client. Néanmoins, la partie *Model* existe des deux côtés, client et serveur. Enfin, les auteurs ont également proposé des modèles hybrides, où les trois parties

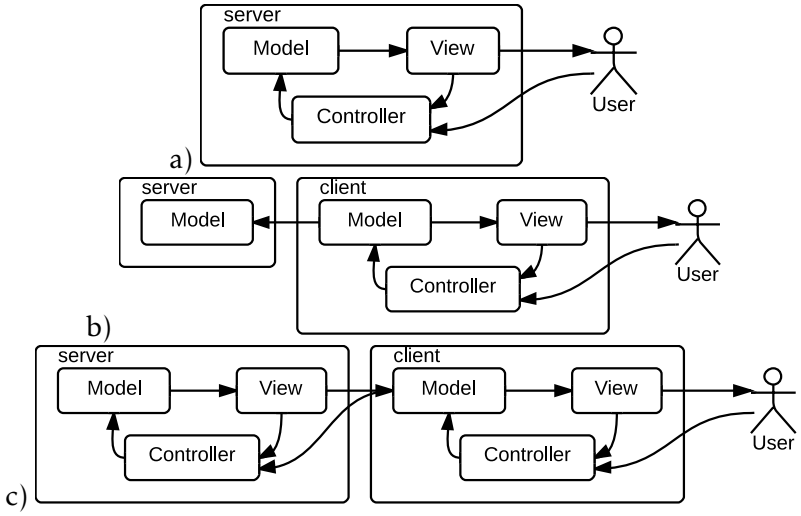


FIGURE 2.3 – Trois variantes d'architectures dérivées du modèle *Model-View-Controller*. a) MVC côté serveur ; b) MVC côté client ; et c) MVC hybride côtés client et serveur.

(*Model*, *View* et *Controller*) existent côtés client *et* serveur. La figure 2.3 illustre ces variantes. Notons que dans tous les cas les applications client et serveur sont reliées entre elles par leur composant *Model*, contrairement au modèle PAC-C3D présenté dans la section précédente, où les applications client et serveur sont reliées par leur composant *Control*.

Ces modèles d'architecture permettent de découper le code d'une application complète en parties correspondant à différents aspects (l'interface utilisateur, la logique métier et le lien entre les deux). Ce découpage a des avantages pour le développeur, qui peut plus facilement raisonner sur chaque aspect indépendamment des autres, mais il a également des avantages structurels : par exemple, la logique métier peut être utilisée indépendamment de l'interface utilisateur, elle est donc plus facilement testable.

Par ailleurs, ces modèles d'architecture ont chacun des caractéristiques techniques qui les rendent plus ou moins appropriés selon les usages attendus de l'application. Le premier modèle est le plus simple à mettre en œuvre car il comporte peu de composants. Il comporte également un aspect technique intéressant : les pages étant construites sur le serveur, leur contenu est mieux référencé par les moteurs de recherche. Son inconvénient est qu'une application bâtie sur ce modèle n'offre pas une expérience utilisateur agréable (navigation synchrone). Le modèle où l'essentiel du code de l'application est côté client permet d'offrir une meilleur expérience utilisateur, mais il nécessite de dupliquer la partie *Model* côtés client et serveur. Ce modèle, contrairement au précédent, ne permet pas au contenu des pages d'être bien référencé par les moteurs de recherche. Les modèles hybrides permettent d'avoir à la fois une bonne expérience utilisateur et un bon référencement du contenu par les moteurs de recherche, au prix d'une duplication des parties *Model* et *View* côtés client et serveur.

Ce panel de modèles d'architecture témoigne surtout d'un aspect caractéristique des applications Web : le code de l'application doit être réparti entre le client et le serveur. Selon le type d'application on peut choisir de mettre plus de code sur le serveur que sur le client, ou inversement. Roberto Rodríguez-Echeverría a d'ailleurs identifié que cet aspect est, avec le processus de synchronisation client-serveur, une source majeure de complexité dans l'ingénierie des applications Web [RE09]. Il a également souligné que les méthodes d'ingénierie des applications Web devraient prendre en compte le fait que la connexion réseau peut être intermittente.

À ce jour, personne n'a encore proposé de style d'architecture permettant le travail en mode déconnecté et automatisant la synchronisation au retour de la connexion. Intégrer la préoccupation de résilience nécessite de surveiller les échanges d'information entre clients et serveurs afin de choisir une stratégie de secours au cas où une requête échouerait. Les modèles d'architecture présentés précédemment font tous l'hypothèse d'une connexion réseau fiable. Si une interruption survient, l'effet d'une action de l'utilisateur sur l'état du système ne peut plus être transmis au serveur. Il est alors nécessaire de détecter le problème, d'en avertir l'utilisateur par une notification et, si possible, d'essayer d'envoyer à nouveau, un peu plus tard, la requête ayant échoué. De telles modifications impacteraient tant la partie interface utilisateur que la partie métier. En particulier, l'ensemble de la couche métier serait affectée car toutes les modifications des données doivent être transmises au serveur. Nous voyons ainsi que la préoccupation de résilience est une préoccupation transverse.

2.1.2 Prise en charge du mode déconnecté

Dès 2000, Yun Yang a formulé l'essentiels des besoins auxquels doivent répondre les applications Web pour permettre le travail collaboratif en mode déconnecté : notamment le fait que la partie client doit contenir suffisamment d'information pour pourvoir les fonctionnalités de l'application, et la nécessité de synchroniser l'état du système avec le serveur au retour de la connexion [Yan00].

Plus récemment, Marco *et. al.* ont proposé un modèle de description de site Web prenant en charge le mode déconnecté [MGPW13]. Leur modèle permet de définir comment l'utilisateur peut naviguer à travers les pages d'un site contenant de l'information et leur système génère un site Web utilisable en mode déconnecté. Cependant leur modèle se limite aux systèmes dont l'état n'évolue pas, il n'est donc pas approprié pour représenter une application dont l'état évolue suite aux actions de l'utilisateur.

Kao *et. al.* [KLYY12] ont implémenté une plateforme d'exécution d'applications Web prenant en charge le mode déconnecté et capable d'adapter le contenu des pages aux appareils mobiles (*i.e.* téléphones). Leur plateforme fournit des bibliothèques pour stocker des données sur les postes clients et les synchroniser avec le serveur. Cependant, leur contribution se limite à une infrastructure logicielle : ils n'indiquent pas comment concevoir une application bénéficiant de cette infrastructure. Cet effort de conception reste donc à faire par les développeurs. En outre, leur solution de synchronisation ne gère pas les conflits.

2.1.3 Travail coopératif assisté par ordinateur

Indépendamment des recherches sur les architectures des applications Web, dans le domaine du travail coopératif assisté par ordinateur des travaux se sont intéressés spécifiquement aux méthodes de synchronisation de l'état d'un système distribué. Une méthode de synchronisation a pour but de faire converger les multiples répliques du système qui existent sur chaque client, après qu'elles ont évolué indépendamment les unes des autres.

2.1.3.1 Synchronisation basée sur les données

Une première approche consiste à observer la façon dont les données évoluent sur les différentes répliques et à transmettre les changements aux autres répliques.

Ainsi, Brett Cannon *et. al.* ont rendu résilient un système de persistence en le dotant d'un mécanisme de synchronisation qui observe les modifications et les synchronise dès que possible avec le serveur [CW10]. L'intégration d'une telle technologie de persistence avec les modèles d'architectures présentés précédemment devrait permettre de traiter la préoccupation de résilience de façon transparente et isolée : la couche métier s'appuierait sur cette technologie de persistence résiliente, les autres parties du code n'auraient pas besoin d'être modifiées.

Les *Conflict-free Replicated Data Types* (CRDT) sont une approche similaire [SPBZ11]. L'idée est de modéliser le domaine du système à l'aide de types de données dont les opérations sont toutes commutatives, par construction. Ainsi, les effets de deux opérations concurrentes convergent toujours, sans générer de conflit. L'inconvénient de cette approche c'est que l'utilisation d'un tel type de données est contraignant, pour les développeurs, pour modéliser le domaine du système.

L'inconvénient des approches basées sur les données est leur mauvaise gestion de la résolution des conflits. Illustrons cela à l'aide de l'exemple d'application de gestion de tâches donné en section 1.1.

Prenons une situation où Alice supprime une tâche donnée, et Bob, de façon concurrente, fait passer l'état de cette même tâche de l'état réalisée à non réalisée. Un système de synchronisation placé sur le serveur et basé sur les données recevrait les modifications du système dues aux clients l'une après l'autre. S'il reçoit d'abord la modification de Bob, puis la suppression d'Alice, il applique les mêmes modifications sur son système, dans cet ordre, et termine dans un état où la tâche est supprimée. S'il reçoit d'abord la suppression d'Alice, puis la modification de Bob, il se retrouve dans une situation où il doit appliquer une modification sur une donnée qui a été supprimée. Dans ce cas, une solution simple consiste à ignorer la modification, et l'état d'arrivée est le même que précédemment : la tâche est supprimée.

Prenons maintenant une situation où Alice supprime toutes les tâche réalisées et Bob fait passer une tâche de l'état réalisée à non réalisée. Si l'algorithme de synchronisation reçoit d'abord la modification de Bob puis la suppression d'Alice, il modifie la tâche puis la supprime car elle faisait partie des tâches réalisées dans la réplique d'Alice. En revanche, s'il reçoit d'abord la suppression d'Alice puis la modification

de Bob, il se retrouve dans une situation similaire à celle du pragraphe précédent : il doit appliquer une modification sur une tâche qui a été supprimée. S'il applique la stratégie décrite au paragraphe précédent, qui consiste à ignorer la modification sur la tâche supprimée, celle-ci est supprimée. Dans les deux cas il n'est pas certain que l'état du système corresponde vraiment à ce qu'attendent Alice et Bob. En effet, au final l'action d'Alice supprime une tâche dans l'état non réalisée alors que son action initiale consistait justement à supprimer les tâches réalisées.

Le problème vient du fait que généralement il n'est pas possible de déduire l'intention initiale d'un utilisateur en fonction de l'évolution des données de sa réplique du système. En effet, différentes actions, correspondant à différentes intentions, peuvent conduire à des modifications de données similaires. Dans notre exemple, une tâche peut être supprimée parce qu'un utilisateur veut supprimer cette tâche en particulier, ou parce qu'elle fait partie des tâches réalisées et que son intention est de supprimer toutes les tâches réalisées.

2.1.3.2 Synchronisation basée sur les actions

En fait, dès 1992 Ernst Lippe *et. al.* distinguaient l'approche précédente, basée sur les données, d'une approche basée sur les actions [LV92]. La seconde approche consiste à comparer les suites d'actions qui ont été effectuées par les différents clients. Les auteurs notent que cette approche donne plus d'opportunités pour résoudre les éventuels conflits car elle permet de préserver l'intention de l'utilisateur.

Dans le domaine des systèmes distribués, des algorithmes de convergence basés sur les actions existent depuis plus de 25 ans [EG89, SE98]. Ceux-ci sont en général conçus pour des architectures décentralisées et ont un coût important en espace, nécessitant généralement des horloges vectorielles (ou quelque chose d'équivalent), ne permettant pas de passage à l'échelle.

Un algorithme centralisé, Jupiter OT, a été proposé en 1995 par David Nichols [NCDL95]. Cet algorithme tire parti des spécificités des architectures client-serveur pour réduire sa complexité en espace : il n'est plus nécessaire, pour un client, de connaître les horloges de tous les autres clients, l'horloge du serveur suffit. Un autre avantage de cet algorithme est qu'il permet d'exécuter l'application d'opérations concurrentes de façon optimiste : il n'est pas nécessaire d'attendre une réponse du serveur pour exécuter une opération sur un client. En effet, celui-ci peut exécuter l'opération directement sur sa machine, modifiant ainsi sa réplique de l'état du système. Si d'autres clients effectuent une modification de façon concurrente, l'algorithme crée une nouvelle opération permettant de compenser les changements de façon à ce que les états de toutes les répliques convergent. Cependant l'algorithme de résolution de conflits n'est pas déterministe : l'état final du système obtenu côté serveur dépend de l'ordre dans lequel il reçoit les mises à jour des clients. Autrement dit, les actions concurrentes effectuées par plusieurs utilisateurs ne conduisent pas toujours à une convergence de l'état du système entre clients et serveurs. En outre, l'algorithme fait l'hypothèse d'un canal de communication fiable et ordonné, ce qui n'est pas le cas du protocole HTTP.

Cette limitation a été résolue par Bin Shao *et. al.* avec TIPS, un autre algorithme

de synchronisation basée sur une architecture centralisée [SLLG11]. Les auteurs introduisent une notion de relation d'effets rendant déterministe l'évolution de l'état du système, quel que soit l'ordre dans lequel les opérations sont reçues par le serveur. En outre, leur algorithme est conçu pour être particulièrement adapté aux architectures Web : de nouveaux clients peuvent joindre ou quitter la session de travail collaboratif à n'importe quel moment, les clients peuvent décider indépendamment les uns des autres quand se synchroniser auprès du serveur et le serveur peut décider quand traiter les opérations transmises par les clients.

Un point commun à tous les algorithmes de synchronisation optimistes basés sur les actions est qu'ils s'appuient sur une fonction de transformation des actions, propre à chaque application. Dans le cas de Jupiter OT cette fonction a la forme suivante :

```
f(c, s) = (c', s')
```

Son implémentation doit satisfaire la propriété suivante : si `c` est une commande issue d'un client et `s` une commande issue du serveur, alors l'application des commandes `s` puis `c'` par le serveur et `c` puis `s'` par le client doivent produire le même effet. C'est cette fonction, spécifique à chaque application, qui permet de compenser les effets relatifs des commandes appliquées de façon concurrente.

C'est cette fonction de transformation des actions qui permet de propager l'intention initiale des utilisateurs. Ainsi, dans le cas de notre exemple d'application de gestion de tâches, l'implémentation de cette fonction peut être différente pour traiter la suppression d'une tâche en particulier ou la suppression de toutes les tâches réalisées. Par conséquent, dans les situations données dans la section précédente, le système conserverait effectivement la suppression d'Alice dans le premier cas mais pas dans le second, correspondant à une intention de suppression de toutes les tâches réalisées.

2.1.4 *Event sourcing*

Enfin, terminons cette section de l'état de l'art en reliant les approches de synchronisation basées sur les actions et l'*event sourcing*. L'*event sourcing* décrit un style d'architecture dans lequel l'état d'un système est représenté par une succession d'événements plutôt qu'à l'aide de types de données [Fow05].

Ainsi, dans notre exemple de logiciel de gestion de tâches, les événements permettant de représenter le système pourraient être :

`Created(id, content, isCompleted)` « une tâche a été créée, elle a pour identifiant `id`, pour contenu `content` et comme état de complétion `isCompleted` » ;

`Removed(id)` « la tâche identifiée par `id` a été supprimée » ;

`Edited(id, content, isCompleted)` « la tâche identifiée par `id` a été modifiée ».

Un système contenant une tâche dont l'état a été modifié une fois pourrait être représenté par la séquence d'événements suivante : `Created(42, "Rédiger une thèse", false)`, `Edited(42, "Rédiger une thèse", true)`.

```
def ui(email: String): Html =
  form(action := "/authenticate", method := "POST",
    input(name := "email", placeholder := "Email", value := email),
    input(name := "pwd", placeholder := "Password"),
    button("Sign in")
  )
```

Listing 2.1 – Définition de fragments de HTML à l'aide d'un langage dédié interne, en Scala.

L'*event sourcing* est surtout utilisé pour améliorer la montée en charge d'applications comportant beaucoup d'utilisateurs en consultation mais peu d'utilisateurs effectuant des modifications [BDM+13].

Notons que les algorithmes de synchronisation basés sur les actions correspondent à une approche *event sourced* : une action est l'équivalent d'un événement. L'algorithme de synchronisation calcule l'état du système en fonction de l'application des actions effectuées par les divers utilisateurs.

2.2 Partage de code

Cette section compare les différentes approches permettant de partager du code entre les parties client et serveur d'une application Web.

2.2.1 Langages dédiés implémentés comme une bibliothèque

Une première approche consiste à utiliser un langage généraliste (p. ex. Scala), et à le rendre exécutable côté client en implémentant un *backend* du compilateur générant du JavaScript. Il devient alors possible d'écrire du code partagé entre les côtés client et serveur. Par exemple, il devient envisageable de définir une abstraction pour représenter les fragments de HTML. L'utilisation d'une telle bibliothèque, pour définir un fragment de HTML correspondant à la figure 1.1, pourrait ressembler à ce qui est donné dans le listing 2.1.

L'ensemble des fonctions de l'API de cette bibliothèque forme un vocabulaire dédié à la construction de fragments de HTML. Comme le langage hôte sur lequel il s'appuie (ici, Scala), est utilisable côtés client et serveur, ce langage dédié est également utilisable côtés client et serveur.

Cette approche a été décrite par Paul Hudak [Hud96]. Son principal intérêt réside dans son faible coût d'implémentation : une fois que le langage hôte peut être exécuté côtés client et serveur, l'introduction de nouveaux éléments de langages dédiés à un aspect particulier ne nécessite que l'implémentation d'une bibliothèque.

GWT est un projet populaire suivant cette approche : il compile du code Java vers du JavaScript. Ainsi, il est possible d'écrire les parties client et serveur d'une application Web en Java, et de partager du code commun aux deux parties. HaXe, Kotlin, SharpKit, Ocsigen et ScalaJs [Doe13] sont d'autres projets similaires.

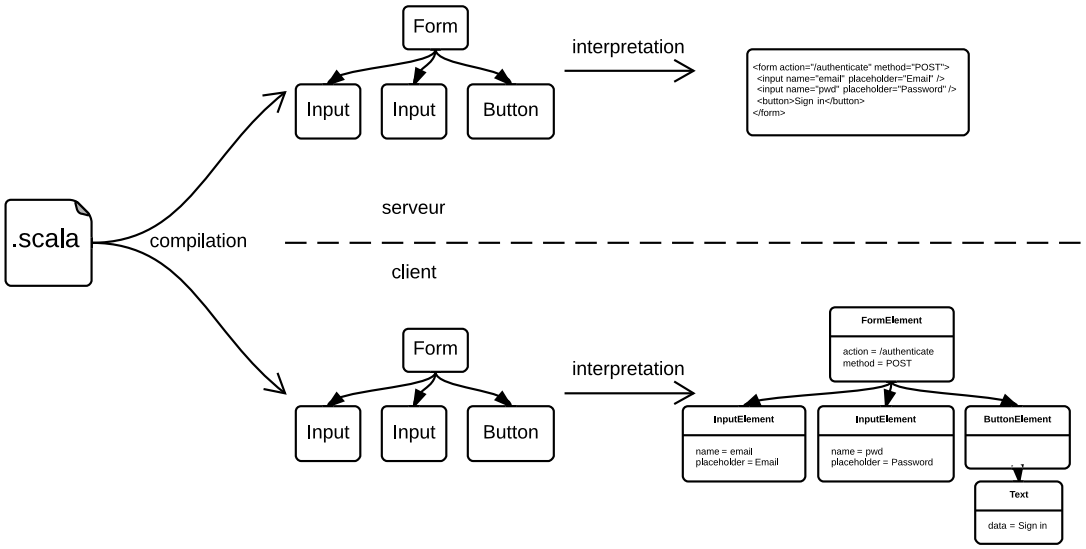


Figure 2.4 – Exécution du programme indiqué dans le listing 2.1. Le programme est compilé vers les environnements client et serveur. Dans chaque environnement, son exécution produit une représentation abstraite d'un fragment HTML. Celle-ci est ensuite convertie vers une représentation spécifique à chaque plateforme.

En général, les outils suivant cette approche fournissent un mécanisme permettant d'interfacer les API du langage hôte (*i.e.* Java, dans dans le cas de GWT) avec l'environnement du navigateur Web. Un tel mécanisme, appelé *foreign function interface* (FFI), permet d'utiliser les bibliothèques JavaScript existantes ou l'API native du navigateur directement depuis le langage hôte. Cependant, toutes les API utilisant ce mécanisme ne sont plus utilisables côté serveur. Par conséquent, il n'est pas possible de définir une bibliothèque partagée par les côtés client et serveur, pour, par exemple, construire des fragments de HTML, tout en exploitant directement les API natives des deux côtés, client et serveur.

Plus généralement, le système de génération de code du compilateur n'a aucun moyen de produire du code tirant parti des spécificités de chaque plateforme (client ou serveur) pour une bibliothèque donnée. En effet, le compilateur n'a connaissance que des concepts du langage hôte, dans lequel sont implémentées les bibliothèques.

En pratique, cela signifie qu'il n'est pas possible de faire en sorte que le listing 2.1 produise le code du listing 1.1 côté client et du listing 1.2 côté serveur. En effet, tout ce que la fonction `form`, par exemple, peut retourner, c'est une description abstraite d'un élément HTML `<form>`. Pour relier ce type de données abstrait à la représentation spécifique lui correspondant, côtés client et serveur, il est nécessaire d'introduire une couche d'interprétation, à l'exécution, comme indiqué en figure 2.4. Cette couche d'exécution peut diminuer les performances du programme.

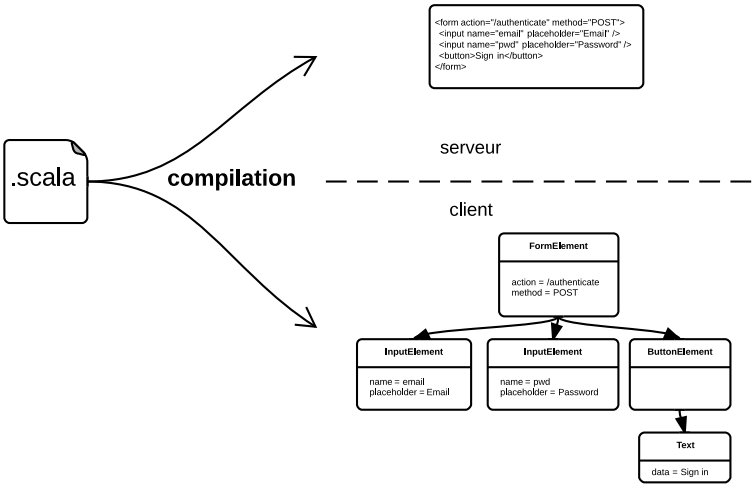


FIGURE 2.5 – Langages lourds. Le programme est compilé vers les environnements client et serveur. Dans chaque environnement, son exécution produit une représentation efficace du fragment HTML, en tirant parti des spécificités de chaque plateforme.

2.2.2 Langages lourds

Pour produire du code plus spécifique à chaque plateforme pour une abstraction donnée, le compilateur doit avoir connaissance de cette abstraction. Un moyen d'y parvenir consiste à intégrer les concepts spécifiques au Web dans le langage lui-même, plutôt que de les définir sous forme de bibliothèques.

Ainsi, si nous poursuivons l'exemple des fragments de HTML, un tel langage incluerait le concept de fragment de HTML, permettant aux développeurs de définir des fragments de HTML directement depuis le langage, avec une syntaxe dédiée et bien intégrée au reste du langage. Le compilateur du langage traduirait, dès la compilation, cette fois, les définitions de fragments de HTML en code natif correspondant, côtés client et serveur. Ce processus est illustré par la figure 2.5.

Le fait que le concept de fragment de HTML fasse partie du langage permet au compilateur de le traiter de façon particulière, et de générer du code spécifique pour chaque plateforme, tirant parti des API natives.

Opa et Links [CLWY07] sont des exemples de tels langages. Ils visent à couvrir, dans un unique langage, l'ensemble des aspects des applications Web. Par exemple, en Opa, les fragments de HTML peuvent être définis en utilisant la syntaxe textuelle de HTML, littéralement, comme le montre le listing 2.2.

Cette approche a deux intérêts par rapport à la précédente :

— La syntaxe du langage peut être adaptée aux aspects à intégrer (cf l'utilisation de littéraux HTML), offrant une meilleur expérience pour le développeur ;

— Les aspects supportés par le langages peuvent être compilés de façon à tirer parti des environnements des plateformes cibles.

```
function ui(email) {
  <form action="/authenticate" method="POST">
    <input name="email" placeholder="Email" value={email} />
    <input name="pwd" placeholder="Password" />
    <button>Sign in</button>
  </form>
}
```

Listing 2.2 – Définition d'un fragment de HTML avec Opa. La syntaxe du langage permet d'intégrer directement des littéraux HTML.

Malgré ces avantages, les outils suivant cette approche n'ont pas connu de grand succès. Cela vient probablement du fait que, d'une part, le développement de la chaîne de compilation d'un langage et de l'outillage à destination des développeurs demande un important effort, par conséquent peu de projets ont atteint un niveau de maturité suffisant et sont capables de s'adapter aussi vite que les besoins et infrastructures évoluent. Et, d'autre part, apprendre un nouveau langage demande également un effort pour les développeurs, par conséquent il est souvent plus rentable de valoriser les compétences d'un développeur sur plusieurs projets plutôt que de le former à un nouvel outil à chaque nouveau projet.

2.2.3 Ingénierie dirigée par les modèles

Une solution, pour limiter les deux inconvénients de l'approche précédente, consiste à utiliser plusieurs langages, chacun dédié à un aspect du développement d'applications Web. Ainsi, chaque développeur peut réutiliser ses compétences d'un projet à un autre, tant que certains aspects se recouvrent, et l'outillage propre à chaque langage peut également être réutilisé. La figure 2.6 illustre le fonctionnement de cette approche.

Une fois que le mécanisme de combinaison des langages et le système de génération de code sont définis, pour un ensemble de langages donnés, cette approche permet de produire les applications côtés client et serveur à partir de leurs aspects décrits dans différents langages dédiés. Les langages dédiés fournissent un vocabulaire permettant de décrire le système à un niveau de détail très abstrait. Cela rend le code du système plus facile à lire et faire évoluer, car il ne contient pas d'information purement technique ou bas niveau.

WebDSL est un exemple de langage de programmation dédié au développement d'applications Web, construit selon une approche dirigée par les modèles[Vis08].

Le principal inconvénient de cette approche est qu'il est difficile de coordonner les différents aspects du programme. En effet, le processus de compilation nécessite, au préalable, de combiner les différents aspects ensemble, tant au niveau syntaxique que sémantique. Or, les technologies basées sur les modèles ne sont pas capables, aujourd'hui, d'automatiser cette phase de composition de langages bien que cela fasse l'objet de nombreuses recherches [VV10, VP12, JCB+13]. Autrement dit, cette approche re-

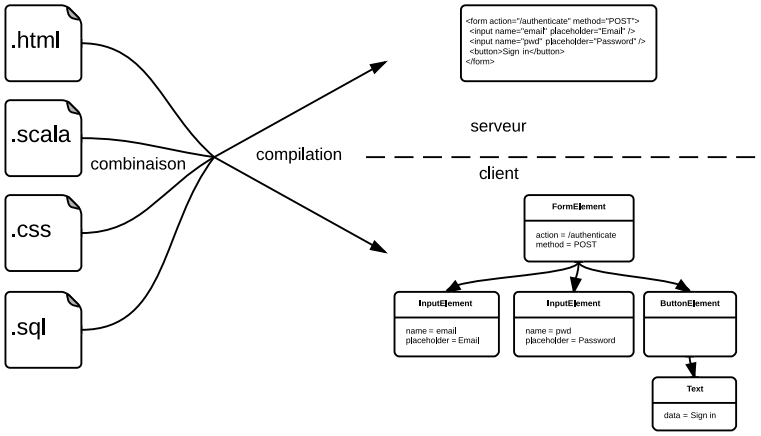


FIGURE 2.6 – Ingénierie dirigée par les modèles. Chaque aspect du programme est décrit dans un langage dédié. La combinaison de ces aspects constitue le programme global, qui est transformé en une application côté client et une application côté serveur.

quiert un effort d'implémentation important.

De même, en réalité les possibilités de réutilisation de l'outillage spécifique à une unité de langage sont minces quand il s'agit de composer ces langages entre eux : seule la partie chargée de l'analyse syntaxique peut éventuellement être réutilisée. La plupart des fonctionnalités des outils, telles que la gestion des références inter-langages, doivent être adaptées à chaque combinaison de langages.

Enfin, un autre inconvénient de cette approche est qu'il est difficile de définir précisément les bornes de chaque langage. L'argument principal de l'approche dirigée par les modèles est qu'on peut associer à chaque aspect du programme un langage simple à maîtriser. En réalité, un langage simple peut très vite devenir limitant en termes de pouvoir d'expression.

Ce problème n'est pas spécifique à l'ingénierie dirigée par les modèles, il survient plus généralement lorsque l'on utilise des langages dédiés dits « externes » [Gho10].

2.2.4 Pouvoir d'expression des langages dédiés externes

Prenons un exemple pour illustrer ce problème. Le langage CSS [CSS] permet de décrire l'aspect visuel des éléments d'une page HTML : la couleur de l'arrière-plan de la page, la couleur du texte, la taille et l'espacement des éléments, etc. Le principal intérêt de CSS est qu'il permet de séparer l'aspect visuel du contenu HTML : le contenu peut être modifié indépendamment de son aspect visuel et réciproquement. CSS est un langage facile à apprendre car il contient peu de concepts : un document CSS est constitué d'un ensemble de règles définissant les valeurs de certaines propriétés visuelles (couleur, espacement, etc.) d'un ensemble d'éléments. D'un autre côté, cette simplicité est pénalisante pour le développeur. Par exemple, il n'est pas pos-

sible de définir la couleur d'un élément en fonction de la couleur d'un autre élément. Il n'est pas non plus possible de définir une règle en réutilisant une règle existante. En d'autres termes, il est difficile d'écrire du code modulaire et réutilisable. Ces problèmes pourraient être résolus si le langage supportait des mécanismes de nommage, de composition et d'abstraction. Il deviendrait alors possible, par exemple, de définir un style abstrait, réutilisable et spécialisable. Cette limitation n'est pas propre au langage CSS, on la retrouve souvent dans les langages dédiés externes (p. ex. HTML ou SQL).

En 1966, Peter J. Landin a idientifé qu'un langage de programmation comportait en général deux facettes : un système linguistique général et un ensemble de concepts dédiés à un problème donné [Lan66]. Les concepts dédiés sont, comme leur nom l'indique, dédiés à un problème ou un aspect donné, et le système linguistique permet de construire des programmes en combinant les éléments de programmes entre eux (en s'appuyant sur des concepts généraux tels que le nommage, la composition et l'abstraction).

Les langages de programmation généralistes se focalisent surtout sur la première facette : ils fournissent des mécanismes généraux permettant de construire des programmes. Tandis que les langages dédiés externes se focalisent surtout sur la seconde facette : ils fournissent des concepts dédiés à un problème donné. Or, les deux facettes sont utiles pour construire des programmes.

Notons que l'on pourrait être tenté d'enrichir un langage externe trop pauvre avec des concepts supplémentaires fournissant des moyens de généralisation ou de composition. Mais, d'une part, cela rendrait le langage plus complexe, ce qui irait à l'encontre du but initial, et, d'autre part, cela conduirait à ré-inventer les mêmes choses dans chaque nouveau langage car les systèmes actuels d'ingénierie des langages ne savent pas produire des langages à partir d'unités de langages réutilisables.

2.2.5 Langages dédiés compilés et implémentés comme une bibliothèque

Ainsi, toutes les solutions aux limitations des langages dédiés implémentés sous forme de bibliothèques présentent de nouvelles limitations diminuant leur applicabilité.

En fait, le principal inconvénient des bibliothèques – l'impossibilité de spécialiser le code à exécuter pour une abstraction et une plateforme données – est exacerbé dans notre situation (le développement d'applications Web) mais n'en est pas l'apanage. En effet, ce problème peut être généralisé à la plupart des abstractions logicielles.

La plupart des composants logiciels sont écrits pour être paramétrables ou configurables. Ainsi, un même composant peut être utilisé différement selon qu'il est intégré dans un environnement de test ou de production. Un exemple typique est la connexion à une base de données : des URL différentes sont utilisées selon l'environnement (test ou production).

Cela signifie que le code du programme qui se connecte à la base de données est paramétré par la connexion effective à la base de données. Pourtant, une fois lancé, le programme se connecte toujours à la même base de données.

On observe donc une situation similaire à celle rencontrée dans l'exemple du langage dédié à la définition de fragments de HTML : un composant logiciel est souvent plus général que l'utilisation qui en est faite à l'exécution, et cela se paie à l'exécution par un niveau d'indirection supplémentaire.

L'idéal serait que le code finalement exécuté ne souffre pas de ce niveau d'indirection : une fois la connexion à la base de données connue, on aimerait pouvoir construire un programme directement relié à celle-ci.

Les systèmes de compilation à la volée (*JIT*) sont capables de faire certaines de ces optimisations, au prix d'une analyse du comportement du programme à l'exécution. Cependant des travaux ont montré que les JITs ne détectent pas autant d'opportunités d'optimisations qu'ils pourraient [RSB+14].

En fait, un autre système effectuant ce genre d'optimisations existe déjà et est utilisé par de nombreuses personnes sans le savoir. En effet, les outils de construction de projets remplissent exactement ce rôle. Ces outils permettent de compiler le code d'un projet et de gérer son cycle de vie (exécution des tests, déploiement, etc.). La plupart des ces outils de construction de projets sont capables de générer une partie du code source à compiler. Par exemple, dans maven, cela correspond à la phase *generate-sources*. Dans sbt, cela correspond à la tâche *sourceGenerators*. Dans les deux cas, les outils génèrent une partie du code source du programme à construire. Le code source du programme final est donc le produit de l'exécution d'un premier programme. Ce programme est un générateur de programmes.

Ainsi, on observe qu'il est très fréquent qu'un programme soit généré par un autre programme. Dans le cas des outils de construction de projet, la distinction est claire car il y a effectivement deux phases d'exécution : une première pour générer du code, et une seconde pour exécuter ce code. Dans le cas des composants logiciels généraux, bien qu'il n'y ait qu'un seul code source et qu'une seule phase d'exécution, on est dans une situation où un programme est obtenu à partir d'un autre programme : en effet, une partie de l'exécution du programme correspond à l'interprétation des paramètres et la seconde partie à l'exécution effective du programme.

Notons que cette observation a déjà été à l'origine de travaux en 1986 [JS86] : les auteurs introduisent le concept de *staging* visant à évaluer certaines parties d'un programme avant les autres selon leur fréquence d'exécution ou la disponibilité des données utilisées.

Les deux approches (outils de construction de programmes et composants généraux) ont leurs avantages et inconvénients. Les outils de construction de programme, grâce à leur première phase d'exécution sont capables de générer du code plus spécialisé, et donc bénéficient de meilleures performances lors de la seconde phase d'exécution. Les composants généraux sont des citoyens de première classe dans tout le reste du programme : ils peuvent être passés en paramètre d'une fonction ou retournés comme résultat, ou être combinés entre eux, ce qui donne une plus grande flexibilité pour le programmeur, au détriment d'une perte de performances à l'exécution.

Les langages dédiés compilés et implémentés sous forme de bibliothèque visent à bénéficier des avantages des deux approches en minimisant leurs inconvénients. L'idée essentielle est la suivante : les unités de langage sont fournies sous forme de bi-

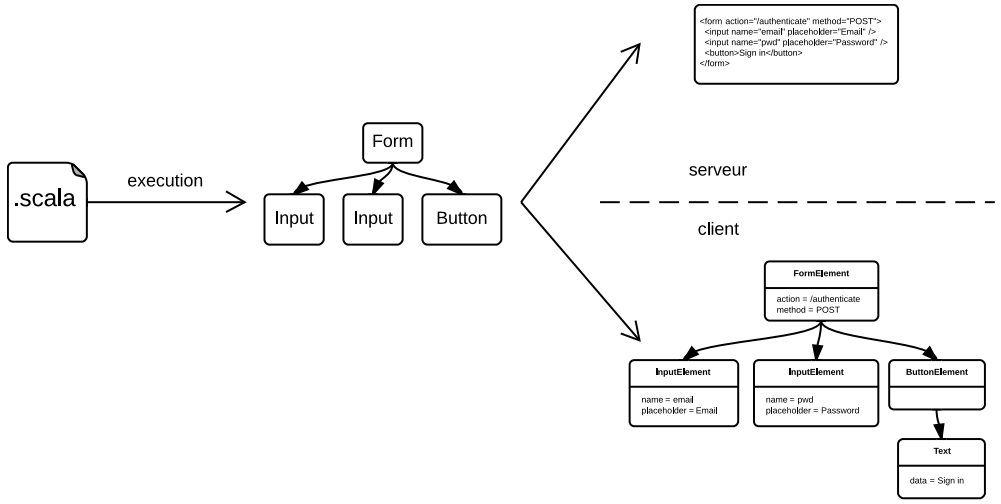


FIGURE 2.7 – Langages dédiés compilés et implémentés par des bibliothèques. Un langage dédié est fourni par une bibliothèque dont l'implémentation produit une représentation abstraite du programme à générer. Un générateur de code peut dériver, à partir de cette représentation abstraite, les programmes correspondant, côtés client et serveur.

bliothèques dont l'implémentation retourne le code source d'un programme optimisé. Des travaux ont introduit cette idée en 1996 [K+96, Hud96]. Une amélioration consiste à utiliser une représentation abstraite du programme à générer plutôt que directement son code source [EFDM00]. Cela permet de réaliser de nombreuses réécritures de code ou optimisations telles que l'élimination de sous-expressions communes ou du code mort. La figure 2.7 illustre cette approche sur l'exemple des définitions de fragments de HTML.

Avec cette approche, le développeur bénéficie du système linguistique du langage hôte pour définir et combiner entre elles des représentations abstraites de programmes obtenues avec le langage dédié à son problème.

La figure 2.8 illustre les différences entre les langages dédiés implémentés par des bibliothèques, intérprétés ou compilés. Dans le premier cas, les concepts dédiés à un problème sont implémentés par des bibliothèques qui sont traduites, par le compilateur du langage hôte, vers du code pour les environnements client et serveur. Ce code, trop général, est adapté à l'exécution vers les spécificités des environnements client et serveur. Dans le second cas, les concepts dédiés à un problème sont également implémentés par des bibliothèques, mais cette fois-ci c'est leur exécution qui produit le code pour les environnements client et serveur. Ce code, produit par les bibliothèques plutôt que par le compilateur du langage, traduit chaque concept vers une représentation spécifique dans chaque environnement cible.

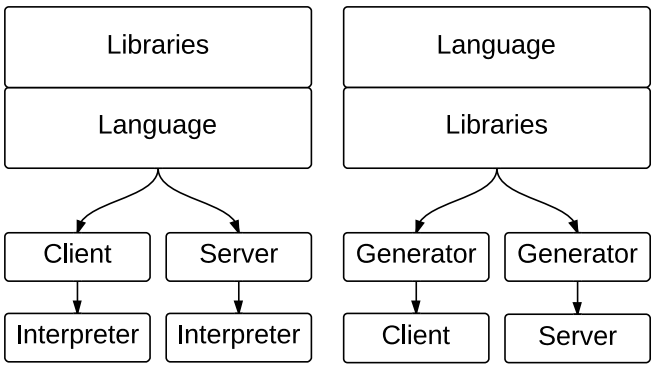


FIGURE 2.8 – Différence entre les langages dédiés implémentés par des bibliothèques, interprétés (à gauche) ou compilés (à droite).

2.2.6 Méta-programmation et évaluation partielle

Par ailleurs, il est notable que le programme écrit par le développeur est en réalité un générateur de programmes car c'est son exécution qui produit le programme final. Le fait que le programme bénéficie au total de deux phases d'exécution relie cette approche aux systèmes d'évaluation partielle [JGS93] : la première exécution peut déjà évaluer certaines parties du programme qui deviendront alors des constantes dans le programme final.

Autrement dit, l'évaluation du programme final est *délayée* par rapport au programe initial. Une idée similaire a déjà été explorée par Manuel Serrano, dans le cas des applications Web, avec le langage Hop [SGL06]. Un programme écrit avec Hop contient en fait deux programmes : celui évalué côté serveur et celui évalué côté client. Le programme évalué côté client peut être construit à partir du résultat d'une évaluation côté serveur, de façon similaire à de l'évaluation partielle. Notons que, dans le cas de Hop, seul le programme côté client bénéficie du mécanisme d'évaluation partielle.

Enfin, une autre manière d'effectuer de l'évaluation partielle est l'utilisation de macros. Les macros permettent d'évaluer des parties du code source au moment de la compilation. Cependant, les macros ne sont pas des entités de première classe au sein d'un langage : leur expansion ne peut avoir lieu qu'au moment de la compilation, elles ne peuvent donc pas être retournées comme résultat d'une fonction ou passées en paramètre. Cette caractéristique limite leurs possibilités d'utilisation. Par exemple, il n'est pas facile d'implémenter une macro extensible ou paramétrable.

2.2.7 *Lightweight Modular Staging* et *js-scala*

Récemment, Tiark Rompf a implémenté *lightweight modular staging* (LMS), une infrastructure logicielle, en Scala, pour définir des langages dédiés compilés et implémentés sous forme de bibliothèques [Rom12]. S'appuyant sur LMS, Kossakowski *et. al.* ont implémenté *js-scala*, un générateur de code JavaScript [KARO12]. Leurs travaux

ont montré le gain en confort obtenu par les développeurs utilisant le langage Scala plutôt que JavaScript pour écrire la partie client des applications Web, notamment grâce au typage statique. Ils ont également montré comment utiliser les bibliothèques JavaScript existantes depuis le langage Scala, à moindre effort, avec un système de typage graduel. Cependant, leur travail ne permet pas de produire, pour une abstraction donnée, du code tirant parti des spécificités des plateformes client et serveur.

Nos travaux se sont basés sur js-scala : nous avons défini des unités de langage produisant du code optimisé, spécifique aux plateformes client et serveur. La suite de cette section détaille le fonctionnement de LMS.

2.2.7.1 Unités de langages composables

Un programme LMS construit une représentation abstraite du programme à générer. De la même façon qu'un programme complexe s'obtient en combinant plusieurs morceaux de programmes plus simples, une représentation abstraite d'un programme complexe s'obtient en combinant entre elles des représentations abstraites de programmes simples.

Un exemple de programme simple est le programme constant : il retourne une valeur, toujours la même. On peut représenter un tel programme, avec LMS, à l'aide de la fonction `const` :

```
val un = const(1)
```

Le terme `un` a pour type `Rep[Int]` et a pour valeur une représentation abstraite d'un programme retournant le nombre 1. Pour l'utilisateur, il n'est pas nécessaire de connaître les détails de cette représentation abstraite.

On peut ensuite représenter un programme plus complexe en combinant plusieurs programmes simples. Par exemple, la fonction `plus` combine deux programmes retournant un nombre et produit un nouveau programme retournant un nombre :

```
val un = const(1)
val deux = const(2)
val trois = plus(un, deux)
```

La fonction `plus` ne calcule pas une somme mais retourne une représentation abstraite d'un programme calculant la somme des résultats des programmes `un` et `deux`.

Soulignons que les fonctions `const` et `plus` ne sont pas des mots-clé du langage Scala, mais bien des noms de fonctions implémentées dans une bibliothèque. Ces fonctions constituent le vocabulaire d'une *unité de langage*, `NumericOps`, permettant de représenter des programmes effectuant des sommes d'entiers.

Les unités de langage forment des modules que l'utilisateur combine pour enrichir son vocabulaire. Des exemples d'unités de langage sont `ArrayOps`, pour manipuler des tableaux, et `FunctionOps`, pour définir des fonctions. Enfin, le développeur peut étendre ce vocabulaire en implémentant ses propres bibliothèques.

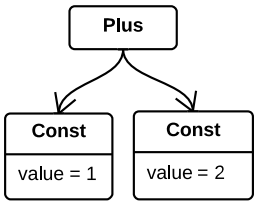


FIGURE 2.9 – Diagramme d'objets correspondant au terme `trois`.

2.2.7.2 Représentation abstraite des programmes

Du point de vue de l'utilisateur, les trois programmes, `un`, `deux` et `trois` ont tous le même type : `Rep[Int]`. Cela, bien que le programme `trois` soit un programme composite, contrairement aux programmes `un` et `deux`, qui sont des programmes constants. Pour l'utilisateur, ce qui compte c'est seulement de savoir qu'il manipule un programme retournant un entier, et c'est tout ce que le type `Rep[Int]` indique.

En pratique, la représentation abstraite des programmes est implémentée à l'aide d'un graphe dont chaque nœud représente une instruction. Dans le cas du programme `un`, ce graphe ne comporte qu'un seul nœud, représentant la valeur constante 1. Ainsi, la fonction `const` retourne un nœud de type `Const`, et la fonction `plus` retourne un nœud de type `Plus`. La figure 2.9 montre le diagramme d'objets correspondant au terme `trois`. Le code suivant montre l'implémentation de la fonction `plus` :

```
def plus(lhs: Rep[Int], rhs: Rep[Int]): Rep[Int] = Plus(lhs, rhs)
```

2.2.7.3 Génération de code

L'objectif final est d'exécuter les programmes ainsi construits. On utilise pour cela des générateurs de code qui prennent en paramètre une représentation abstraite de programme et produisent un fichier texte contenant le code du programme. Le niveau de granularité de la génération de code est définie par les nœuds du graphe représentant le programme. Les nœuds du graphe sont parcourus et le générateur de code écrit le code correspondant à chacun. Par exemple le générateur de code JavaScript pour le nœud `Plus` est défini comme suit :

```
case Plus(lhs, rhs) => gen"var $sym = $lhs + $rhs;"
```

Le code produit une variable contenant le résultat de la somme des deux opérandes.

Au final, la sortie du générateur de code JavaScript appliqué au programme `trois` est la suivante :

```
var x0 = 1 + 2;
```

2.2.7.4 Optimisations

En fait, comme les termes `un` et `deux` représentent des programmes constants, leur valeur est connue dès l'exécution du générateur de programmes, et cela constitue une opportunité d'optimisation : la somme des deux constantes pourrait être déjà évaluée dans le générateur de programmes et produire la constante 3 dans le programme final.

Ainsi, une implémentation plus optimisée de la fonction `plus` est la suivante :

```
def plus(lhs: Rep[Int], rhs: Rep[Int]): Rep[Int] = (lhs, rhs) match {
  case (Const(lhs), Const(rhs)) => const(lhs + rhs)
  case (lhs, Const(0)) => lhs
  case (Const(0), rhs) => rhs
  case _ => Plus(lhs, rhs)
}
```

Cette implémentation teste si les valeurs à additionner sont des représentations de programmes constants ou non. La somme de deux programmes constants est évaluée directement et produit un nouveau programme constant. Les cas particuliers d'addition d'un programme non constant avec la constante zéro sont également traités en éliminant l'addition. Enfin, le cas plus général, correspondant à deux programmes non constants, produit un nœud de type `Plus`.

2.2.7.5 Flexibilité syntaxique

Voici un exemple plus complexe de code décrivant un programme manipulant un intervalle de nombres :

```
val ns = range(1, 4)
val inc = fun { (n: Rep[Int]) =>
  plus(n, const(1))
}
val prog = range_map(ns, inc)
```

Le terme `prog` représente un programme qui applique une fonction à tous les éléments d'un intervalle de nombres. La fonction appliquée, représentée par le terme `inc`, retourne la valeur `n`, qui lui est passée en paramètre, majorée de 1. L'intervalle de nombres, représenté par le terme `inc`, contient les nombres de 1 à 4.

À titre de comparaison, un programme équivalent en Scala pur [1] s'écrirait comme suit :

```
for (n <- 1 to 4) yield n + 1
```

La comparaison des deux listings peut conduire le lecteur à estimer que la version écrite avec LMS est nettement plus verbeuse, donc plus difficile à lire. Par exemple, le corps de la fonction effectuant l'incrémentation s'écrit `plus(n, const(1))` dans la version LMS, tandis qu'il s'écrit simplement `n + 1` en Scala pur. Heureusement, le

1. Dans ce document, nous considérons les unités de langages définies au-dessus de LMS comme un langage différent de Scala bien que ce ne soit pas le cas puisque LMS est simplement une bibliothèque Scala.

langage Scala fournit un mécanisme de conversion implicite permettant à LMS de définir une conversion implicite de toute constante en un programme constant. Cela permet à l'utilisateur d'omettre le terme `const` : il peut simplement écrire `plus(n, 1)`. De plus, le langage Scala fournit également un mécanisme permettant de définir *a posteriori* des opérateurs pour des types de données. LMS utilise ce mécanisme pour définir un opérateur + ayant pour opérandes des valeurs de type `Rep[Int]`. Cela permet à l'utilisateur d'écrire simplement `n + 1`, comme en Scala pur.

Au final, grâce à ces facilités syntaxiques, l'utilisateur peut écrire son programme comme suit :

```
val prog: Rep[List[Int]] = for (n <- 1 to 4) yield n + 1
```

C'est-à-dire, exactement comme il l'aurait écrit en Scala pur ! L'idée essentielle est que LMS ne nécessite pas le recours à une syntaxe particulière pour distinguer les termes représentant des valeurs des termes représentant des programmes. La distinction se fait uniquement par les types : le type `Int` représente un nombre entier, tandis que le type `Rep[Int]` représente un programme retournant un nombre entier.

2.2.7.6 Synthèse

En résumé, définir une unité de langage avec LMS, consiste à :
— définir une API fournissant le vocabulaire du langage (section 2.2.7.1) ;
— écrire une implémentation de cette API produisant une représentation abstraite de chaque concept du langage (section 2.2.7.2) ;
— définir un générateur de code (ou plusieurs, si différentes plateformes d'exécution sont ciblées) produisant le code correspondant à chaque nœud représentant un concept du langage (section 2.2.7.3).

2.3 Sûreté du typage

Dans la section 2.2.7 nous avons mentionné le fait que js-scala permet d'utiliser les bibliothèques JavaScript existantes en utilisant un système de typage graduel. Ainsi, il n'est pas nécessaire de donner une signature de type aux fonctions de l'API JavaScript utilisée, mais, en contrepartie, aucune vérification du typage n'est effectuée. Cela peut donc conduire l'utilisateur à écrire des programmes incorrects sans que le compilateur ne l'en avertisse.

Des travaux ont été réalisés pour vérifier que des appels de fonctions depuis un langage vers un autre, utilisant des systèmes de type différents, soient bien typés [FF05]. Cependant, notre problème est un peu différent : la difficulté réside dans le fait qu'à l'utilisation les signatures de type des fonctions standard du navigateur ne se révèlent pas « pratiques » pour le développeur. En effet, elles l'obligent ensuite à effectuer une opération de transtypage. L'enjeu consiste à trouver une autre signature de type contenant plus d'informations afin d'éviter à l'utilisateur d'effectuer des opérations de transtypage.

```
var createTwoElements = function (fst, snd) {
  return [
    createElement(fst),
    createElement(snd)
  ]
};
```

Listing 2.3 – Fonction faisant appel à `createElement`

Le langage TypeScript permet de surcharger des opérations selon la valeur de certains paramètres. Ainsi, le compilateur attribue à l'expression `createElement("div")` le type `DivElement` car la valeur du paramètre passé à la fonction, `"div"`, est connue en temps de compilation (c'est une constante). Cette solution donne autant de pouvoir d'expression aux utilisateurs et est bien typée à condition qu'elle soit utilisée avec des valeurs connues en temps de compilation. Cette contrainte limite son applicabilité car une fonction prenant en paramètre un nom d'élément `name` et créant un élément en appelant `createElement(name)` ne peut pas bénéficier du bon typage : le type de retour inféré de l'appel à `createElement` n'est pas plus précis que `Element`. Cette limitation est illustrée par le listing 2.3. Dans ce programme, la fonction `createTwoElements` prend en paramètre deux noms d'éléments et retourne un tuple contenant les deux éléments créés correspondant. Tout ce que le système de type du langage TypeScript peut inférer comme type de retour pour cette fonction est un tuple d'`Elements`.

Une autre solution consiste à définir plusieurs fonctions spécialisées plutôt qu'une seule fonction générale. Ainsi, GWT propose une fonction `createInputElement()` ayant pour type de retour `InputElement`, une fonction `createImageElement()` ayant pour type de retour `ImageElement`, etc. Le développeur, en appelant `createInputElement()` plutôt que `createElement("input")` obtient alors une valeur du type le plus précis possible (`InputElement` plutôt que `Element`). L'inconvénient de cette solution, outre le fait qu'elle augmente significativement le nombre de fonctions de l'API, est qu'elle diminue le pouvoir d'expression de l'utilisateur. En effet, comme le paramètre représentant le nom de l'élément à créer n'existe plus, il n'est plus possible d'écrire du code lui-même paramétré par un nom d'élément et faisant appel à `createElement`. Autrement dit, avec cette approche il n'est pas possible d'implémenter la fonction `createTwoElements` du listing 2.3.

Une approche complètement différente a été suivie par Chugh *et. al.* [CHJ12]. Ils ont montré comment typer du code JavaScript à l'aide d'un système de typage à types dépendants. Leurs travaux visent à bien typer un sous-ensemble du langage JavaScript. Cependant, la complexité des annotations de type nécessaires rend leur approche difficile à maîtriser par les utilisateurs.

Deuxième partie

Contributions

Chapitre 3

Partage de code

Ce chapitre présente un moyen de partager de code entre les environnements client et serveur. En nous appyant sur LMS et js-scala, nous avons pu définir des unités de langage fournissant des APIs de haut niveau, parfois utilisables des deux côtés, client et serveur, et générant du code efficace. Ce chapitre présente ces abstractions et leurs détails d'implémentation.

Le contenu de ce chapitre a fait l'objet d'une publication [RFBJ13b].

3.1 API des sélécteurs

L'API des sélécteurs est une API du navigateur permettant de rechercher des éléments HTML dans une page. Elle contient plusieurs fonctions, résumées dans la figure 3.1. Les fonctions `querySelector` et `querySelectorAll` sont les plus générales : les autres correspondent seulement à des cas particuliers de ces fonctions. Par exemple, la fonction `getElementById` peut être implémentée comme ceci :

```
var getElementById = function (id) {
  return querySelector('#' + id)
};
```

De façon similaire, les fonctions `getElementsByClassName` et `getElementsByTagName` peuvent être implémentée avec `querySelectorAll`.

Fonction	Élément(s) recherché(s)
`querySelector(s)`	Premier élément correspondant au sélecteur CSS `s`
`getElementById(i)`	Élément dont l'attribut `id` vaut `i`
`querySelectorAll(s)`	Tous les éléments correspondant au sélecteur CSS `s`
`getElementsByTagName(n)`	Tous les éléments ayant pour nom `n`
`getElementsByClassName(c)`	Tous les éléments dont l'attribut `class` contient `c`

Figure 3.1 – L'API standard des sélecteurs.

```
function getWords() {
  var form = document.getElementById('add-user');
  var sections =
    form.getElementsByTagName('fieldset');
  var results = [];
  for (var i = 0 ; i < sections.length ; i++) {
    var words = sections[i]
      .getElementsByClassName('word');
    results[i] = words;
  }
  return results
}
```

Listing 3.1 – Utilisation de l'API native pour rechercher des éléments dans une page HTML

```
function getWords() {
  var form = $('#add-user');
  var sections = $('fieldset', form);
  return sections.map(function () {
    return $('.word', this)
  })
}
```

Listing 3.2 – Utilisation de jQuery pour rechercher des éléments dans une page HTML

Ces fonctions plus spécifiques ont l'avantage d'avoir une implémentation plus performante par le navigateur, comparées aux fonctions générales `querySelector` et `querySelectorAll`. D'un autre côté, elles ajoutent de la complexité au code en augmentant le nombre de fonctions à connaître pour manipuler l'API et en poussant l'utilisateur à penser à un bas niveau d'abstraction.

Le listing 3.1 illustre l'utilisation de cette API pour récupérer une liste de champs de formulaires. La fonction `getWords` commence par rechercher l'élément ayant pour id `add-user`, puis collecte tous les sous-éléments de type `fieldset`, et, pour chacun, retourne la liste de ses sous-éléments dont l'attribut class contient `word`.

Notons que le code utilise uniquement les trois fonctions bas niveau de l'API du navigateur. À titre de comparaison, la bibliothèque jQuery [BK08], largement utilisée par les sites Web [jQu], ne propose qu'une seule fonction pour rechercher des éléments dans une page HTML. Une version du programme `getWords` avec jQuery est donnée dans le listing 3.2.

Le code est à la fois plus court et plus simple, notamment grâce au fait que jQuery propose une API plus haut niveau que l'API du navigateur. Cependant, ce confort d'utilisation se paie par des performances d'exécution moindres (les benchmarks présentés en section 3.4 montrent des temps d'exécution de l'ordre de 10 à 30 fois plus longs avec jQuery).

```
def getWords() = {
  val form = document.find("#add-user")
  val sections = form.findAll("fieldset")
  sections map (_.findAll(".word"))
}
```

Listing 3.3 – Utilisation de DomOps pour rechercher des éléments dans une page HTML

```
def find(selector: Rep[String]) = selector match {
  case ConstIdCss(id) if receiver == document =>
    DocumentGetElementById(Const(id))
  case _ =>
    SelectorFind(receiver, selector)
}
```

Listing 3.4 – Optimisation de `find`

En nous appuyant sur le mécanisme d'évaluation partielle de LMS, notre idée consiste à définir une API de haut niveau, fournissant un confort d'utilisation similaire à jQuery, mais dont l'évaluation produit un programme utilisant les APIs bas niveau du navigateur, bénéficiant des bonnes performances d'exécution.

Ainsi, nous proposons une API de haut niveau, *DomOps*, comprenant deux fonctions : `find` et `findAll`, permettant de chercher un élément ou plusieurs éléments, respectivement. Ces fonctions sont directement équivalentes aux fonctions `querySelector` et `querySelectorAll` de l'API native. Elles prennent en paramètre un sélecteur CSS et retournent le premier élément ou tous les éléments correspondant.

Le listing 3.3 montre code de la fonction `getWords` avec notre API. Ce code est, lui aussi, simple et concis, comparé au listing 3.1 utilisant l'API native du navigateur.

L'implémentation des fonctions `find` et `findAll` analyse les sélecteurs qui leur sont passés en paramètre pour détecter une opportunité d'utilisation d'une fonction spécialisée plutôt que de toujours utiliser `querySelector` et `querySelectorAll`, respectivement. L'analyse consiste à détecter un motif particulier dans la chaîne de caractères représentant le sélecteur.

Ainsi, l'implémentation de `find` vérifie si le sélecteur correspond à un sélecteur d'id et si l'objet sur lequel est appelée la méthode est `document`, et, le cas échéant, retourne un nœud de type `DocumentGetElementById`. Sinon, elle retourne un nœud de type `SelectorFind`. Ces nœud sont traduits, par le générateur de code JavaScript, en code utilisant `getElementById` et `querySelector`, respectivement. Le listing 3.4 montre l'implémentation de la fonction `find`.

Du point de vue de l'utilisateur, la fonction s'utilise toujours de la même façon mais elle produit un code différent en fonction de la valeur des paramètres qui lui sont passés. Par exemple, l'évaluation de l'expression `document.find("#add-user button")` retourne un nœud de type `SelectorFind`, car le sélecteur ne correspond pas à un sélec-

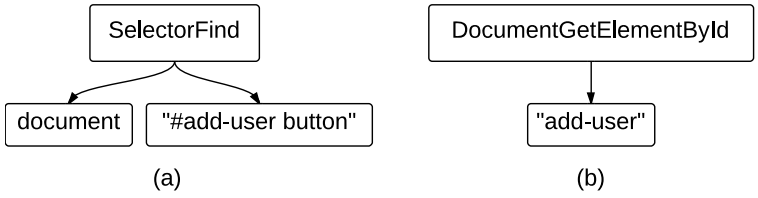


FIGURE 3.2 – Représentations abstraites de code retournées par l'évaluation des expressions (a) `document.find("#add-user button")` et (b) `document.find("#add-user")`

```
def findAll(selector: Rep[String]) = selector match {
  case ConstClassName(name) =>
    GetElementsByClassName(receiver, name)
  case ConstTagName(name) =>
    GetElementsByTagName(receiver, name)
  case _ =>
    SelectorFindAll(receiver, selector)
}
```

Listing 3.5 – Optimisation de findAll

teur d'id (c'est un sélecteur composite sélectionnant les éléments de type `button` à l'intérieur de l'élément ayant pour id `add-user`). D'un autre côté, l'expression `document.find("#add-user")` retourne un nœud de type `DocumentGetElementById`. La figure 3.2

L'implémentation de la fonction `findAll`, donnée dans le listing 3.5, est similaire : le sélecteur passé en paramètre est analysé pour détecter s'il s'agit d'un sélecteur de classe ou de nom d'élément, et, le cas échéant, retourne un nœud de type `GetElementsByClassName` ou `GetElementsByTagName`, respectivement. Sinon, elle retourne un nœud de type `SelectorFindAll`.

Notre solution offre donc à l'utilisateur une API de haut niveau dont le surcoût n'existe que lors de la première évaluation du programme : le programme final généré s'appuie, lorsque cela est possible, directement sur les APIs bas niveau performantes. En fait, le listing 3.3 produit exactement le code bas niveau du listing 3.1.

3.2 Manipulation de valeurs optionnelles

Certaines fonctions du navigateur peuvent, dans certains cas, ne pas retourner de résultat : typiquement, les fonctions de recherche d'éléments HTML peuvent échouer (dans le cas où l'élément recherché n'est pas trouvé dans la page) et, dans ce cas, la valeur retournée doit représenter l'absence de résultat. La majorité des API JavaScript utilisent l'objet `null` pour représenter l'absence de valeur.

Cependant, cette solution est connue pour être une importante source de fragilité dans le code des programmes [NS09, Hoa09]. Pour s'en protéger, les programmeurs peuvent faire de la programmation dite « défensive », en vérifiant toujours qu'une valeur ne vaut pas `null` avant de l'utiliser. À titre d'exemple, le listing 3.6 montre un

```
var loginWidget = document.querySelector("div.login");
var loginButton = loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", handler);
```

Listing 3.6 – Code non sûr manipulant des valeurs optionnelles

```
var loginWidget = document.querySelector("div.login");
if (loginWidget !== null) {
  var loginButton = loginWidget.querySelector("button.submit");
  if (loginButton !== null) {
    loginButton.addEventListener("click", handler);
  }
}
```

Listing 3.7 – Style défensif pour se protéger du déréférencement de valeurs `null`

programme JavaScript recherchant un composant dans la page, puis recherchant un bouton à l'intérieur de ce composant. Le code de ce listing n'est pas sûr car les termes `loginWidget` et `loginButton` sont utilisés sans vérifier qu'ils ne valent pas `null`. Par conséquent, l'exécution de ce programme sur une page ne contenant pas les éléments recherchés lèvera une exception au moment où le terme `loginWidget` sera déréférencé.

Une version sûre du même programme est donnée dans le listing 3.7. Ce style de code, très verbeux, n'est pas pratique à l'utilisation, et, surtout, nécessite une grande rigueur de la part de l'utilisateur car rien ne le force à effectuer ces tests pour différencier l'absence de valeur des cas où une valeur est bien présente.

Une amélioration possible consiste à utiliser un type de données spécifique pour représenter l'absence éventuelle de valeur. Cela permet à l'utilisateur de distinguer les cas où il doit se protéger de l'absence de valeur des cas où il peut faire l'hypothèse que la valeur est présente. Cela réduit l'effort intellectuel à fournir par l'utilisateur mais pas la verbosité du code car les tests vérifiant que la valeur est présente sont toujours nécessaires.

Une seconde amélioration consiste donc à bénéficier d'une syntaxe allégée pour déréférencer les valeurs optionnelles, de façon à diminuer la verbosité du code. En Scala, la notation `for` permet de séquencer l'évaluation d'expressions de façon concise.

En nous appuyant sur ces améliorations, notre solution, *OptionOps*, fournit une syntaxe concise pour manipuler les valeurs optionnelles de façon sûre. Le listing 3.8

```
for {
  loginWidget <- document.find("div.login")
  loginButton <- loginWidget.find("submit.button")
} loginButton.on(Click)(handler)
```

Listing 3.8 – Manipulation de valeurs optionnelles avec OptionOps

montre l'implémentation du programme précédent avec OptionOps. La fonction `find` retourne maintenant une valeur de type `Rep[Option[Element]]` au lieu de `Rep[Element]` et l'implémentation du séquencement, dans les expressions `for` vérifie que la valeur est bien présente.

Encore une fois, cette abstraction a un coût : l'utilisation d'un type de données spécifique pour représenter l'absence éventuelle de valeur nécessite une allocation d'objet supplémentaire, et les fonctions d'ordre supérieur utilisées pour le séquencement, par la notation `for`, sont moins efficaces que des instructions `if`.

Heureusement, comme précédemment, notre abstraction et ses surcoûts n'existent que dans le programme initial. En effet, le séquencement d'expressions représentant des programmes contenant des valeurs optionnelles retourne une représentation de programme effectuant ce séquencement. Finalement, le générateur traduit cette représentation abstraite de séquencement en véritable séquencement en générant du code vérifiant qu'une valeur n'est pas absente en la comparant avec `null`. Le listing 3.9 montre l'implémentation de la génération de code pour le séquencement.

```
case OptionForeach(option, block) =>
  gen""if ($option !== null) {
          $block
        }"""
```

Listing 3.9 – Générateur de code JavaScript pour le langage de manipulation de valeurs optionnelles

Le programme haut niveau du listing 3.8 produit un programme JavaScript identique au listing bas niveau 3.7.

Enfin, comme l'implémentation de l'unité de langage retourne une représentation abstraite des opérations, celle-ci est indépendante de la plateforme d'exécution, et il est facile d'implémenter, en plus du générateur de code JavaScript, un générateur de code Scala. Ainsi, la même abstraction – l'unité de langage pour manipuler des valeurs optionnelles – est utilisable côtés client et serveur, et le code généré est efficace des deux côtés (il n'utilise pas d'objet supplémentaire pour représenter l'éventuelle absence de valeur).

3.3 Définition de fragments de HTML

Nous avons montré en introduction à quel point les plateformes client et serveur représentent différemment l'interface utilisateur (listings 1.1 et 1.2, page 15).

Côté client, l'interface utilisateur de l'application est représenté par un arbre organisant de façon hiérarchique les composants, le *Document Object Model* (DOM). L'utilisation d'un arbre permet facilement de remplacer un nœud par un autre, d'introduire un nouveau nœud ou de retirer un nœud, dynamiquement, pour mettre à jour l'interface utilisateur.

```
def ui(email: Rep[String]): Rep[Element] =
  el('form, 'action -> "/authenticate", 'method -> "POST")(
    el('input, 'name -> "email",
        'placeholder -> "Email", 'value -> email)(),
    el('input, 'name -> "pwd", 'placeholder -> "Password")(),
    el('button)("Sign in")
  )
```

Listing 3.10 – Définition d'un fragment de DOM avec Forest

Côté serveur, l'interface utilisateur est généralement représentée par du texte contenant du code HTML. Cette représentation permet au contenu d'une page Web d'être facilement transmis dans le corps d'une réponse HTTP.

Construire le contenu HTML depuis les côtés client et serveur ont tous les deux des avantages. Côté client, cela permet de faire évoluer l'interface utilisateur dynamiquement, sans rechargement complet de la page, ce qui procure une meilleur expérience utilisateur. Côté serveur, cela permet au contenu d'être mieux référencé par les moteurs de recherche (le référencement du contenu produit côté client ajoute beaucoup de complexité aux systèmes de référencement) et permet de gagner du temps sur l'affichage initial du contenu. Ces préoccupations techniques conduisent les développeurs à dupliquer, côtés client et serveur, le code définissant le contenu HTML de leurs applications.

En plus de ces considérations techniques, notons que l'API du navigateur pour la construction de fragments de DOM est particulièrement verbeuse : en effet, le listing JavaScript 1.1 est plus de deux fois plus long que son équivalent Scala, le listing 1.2. En outre, le style impératif de l'API du navigateur ne permet pas facilement de visualiser la structure hiérarchique des différents éléments d'un fragment de DOM, contrairement à la notation HTML.

En résumé, pour atteindre notre objectif de réduction de la complexité de la définition de contenus Web, nous devons proposer un seul langage, concis, permettant de générer les contenus côtés client et serveur.

Malgré les différences de représentation côtés client et serveur, les contenus Web sont, dans les deux cas, constitués d'éléments HTML et de texte. Ainsi, nous proposons un langage, appelé *Forest*, centré sur des deux concepts. L'implémentation de notre langage produit une représentation abstraite de contenu Web. Nos générateurs de code parcourent cette représentation abstraite et génèrent, pour chaque plateforme, du code bas niveau utilisant les API natives.

Le listing 3.10 montre comment construire le contenu Web présenté en figure 1.1. La fonction `el` construit un élément HTML, lequel peut contenir des attributs et des sous-éléments. Ceux-ci peuvent être des éléments HTML ou du texte, comme dans le cas du bouton. Le type de retour de la fonction `ui`, `Rep[Element]`, désigne une représentation abstraite de programme produisant un élément HTML.

Il est notable que notre unité de langage, bien que minimaliste (elle ne comporte qu'une fonction, `el`), n'a pas un pouvoir d'expression limité. En effet, comme elle est

```
def usersUi(users: Rep[Seq[User]]) =
    el('ul)(
        for (user <- users)
        yield el('li)(userUi(user))
    )
```

Listing 3.11 – Utilisation de boucles et composition de fragments Web avec Forest

définie sous forme de bibliothèque, elle s'intègre directement dans le langage Scala dont elle peut bénéficier de tous les concepts : fonction, boucles, tests, etc. Les représentations abstraites de programmes produisant du contenu Web (les valeurs de type `Rep[Element]`) sont des entités de première classes, elles peuvent être passées en paramètre à des fonctions, utilisées dans des boucles, etc. de telle sorte qu'au final, l'utilisateur bénéficie d'un système puissant de génération de contenus Web, similaire aux moteurs de gabarits (*template engines*, en anglais) généralement fournis par les solutions de construction d'applications Web.

Le listing 3.11 montre, ainsi, comment construire un contenu Web affichant une liste d'utilisateurs. La fonction `usersUi` crée un élément HTML `ul`, dont les sous-éléments sont obtenus en transformant une collections d'utilisateurs en une collection de fragments Web. Cette transformation utilise une unité de langage de manipulation de collections. Enfin, le fragment Web décrivant chaque utilisateur est obtenu en appelant la fonction `userUi`, dont le code n'est pas montré dans le listing. Le point important du listing étant de montrer qu'il est possible de découper la définition de contenus Web en fragments réutilisables.

Les listings 3.12 et 3.13 montrent les implémentations des générateurs de code JavaScript et Scala pour le langage Forest. Dans les deux cas, le fonctionnement est similaire. Il y a deux types de nœuds à traiter, les éléments HTML (`Tag`) et le texte (`Text`). La version JavaScript construit un arbre en utilisant l'API native du DOM. La version Scala génère du texte (le code du générateur est difficile à lire du fait que le code généré, représenté par des chaînes de caractères, contient lui-même des chaînes de caractères). Les deux générateurs de code gèrent le fait que les sous-éléments d'un élément Web peuvent être connus au moment de l'évaluation du programme initial et, le cas échéant, déroulent la boucle insérant chaque sous-élément dans son élément parent. Dans le cas contraire, ils génèrent une boucle dans le langage cible.

Au final, l'exécution du listing 3.10 produit le code bas niveau du listing 1.1 côté client et du listing 1.2 côté serveur.

3.3.1 Représentation statiquement typée de chaînes de caractères contenant des expressions JavaScript

Il est possible de définir, dans un document HTML, le code JavaScript à exécuter lorsqu'un événement se produit sur un élément HTML donné. Ainsi, le listing HTML 3.14 définit un élément exécutant le code JavaScript `alert('clicked')` lorsque l'utilisateur clique dessus.

```
case Tag(name, children, attrs) =>
  emitValDef(sym, src"document.createElement('$name')")
  for ((n, v) <- attrs) {
    gen"$sym.setAttribute('$n', $v);"
  }
  children match {
    case Left(children) =>
      for (child <- children) {
        gen"$sym.appendChild($child);"
      }
    case Right(children) =>
      val x = fresh[Int]
      gen""for (var $x = 0; $x < $children.length; $x++) {
              $sym.appendChild($children[$x]);
            }"""
  }
case Text(content) =>
  emitValDef(sym, src"document.createTextNode($content)")
```

Listing 3.12 – Générateur de code JavaScript pour l'unité de langage de définition de contenus Web

```
case Tag(name, children, attrs) =>
  val attrsFormatted =
    (for ((name, value) <- attrs)
     yield src" $name=$${$value}").mkString
  children match {
    case Left(children) =>
      if (children.isEmpty) {
        emitValDef(sym, src"html\"\"\"<$name$attrsFormatted />\"\"\"")
      } else {
        val qc = children.map(quote)
        emitValDef(sym,
          src"html\"\"\"<$name$attrsFormatted>$${$qc}</$name>\"\"\""
        )
      }
    case Right(children) =>
      emitValDef(sym,
        src"html\"\"\"<$name$attrsFormatted>$${$children}</$name>\"\"\""
      )
  }
case Text(content) =>
  emitValDef(sym, src"html\"\"\"$${$content}\"\"\"")
```

Listing 3.13 – Générateur de code Scala pour l'unité de langage de définition de contenus Web

```
<div onclick="alert('clicked')">click me</div>
```

Listing 3.14 – Code JavaScript inclus dans une chaîne de caractères

```
el('div, 'onclick -> "println(\"clicked\")")("click me")
```

Listing 3.15 – Code Scala inclus dans une chaîne de caractères

Il n'est pas possible de représenter de telles constructions avec Forest car nous n'avons pas de moyen d'obtenir, à l'intérieur d'un programme js-scala, la représentation textuelle d'un programme JavaScript à partir de sa représentation abstraite.

Si nous écrivions du code Scala directement dans une chaîne de caractères, cela serait fragile car notre code Scala ne bénéficierait pas des vérifications de type apportées par le compilateur Scala, mais cela serait surtout incorrect car les navigateurs Web ne sauraient pas interpréter le code Scala. Le listing 3.15 illustre ce non-sens.

Ce que nous voulons, c'est désigner une expression Scala et obtenir la représentation textuelle du code JavaScript correspondant. Nous pouvons résoudre ce problème en introduisant un nouveau mécanisme : une fonction `quote` qui prendrait en paramètre, sans l'évaluer, une expression Scala représentant un programme JavaScript, et retournerait une représentation textuelle de cette expression. La signature de cette fonction est donnée dans le listing 3.16.

Le listing 3.17 montre comment s'appuyer sur cette fonction pour implémenter l'équivalent du listing 3.14 avec Forest, de façon statiquement typée.

La fonction `quote` crée d'abord une fonction sans paramètre qui se contente d'évaluer l'expression donnée, puis retourne un nœud pour lequel le générateur de code JavaScript produit une chaîne de caractères contenant l'appel de la fonction produite à la première étape. Le code JavaScript généré par le listing 3.17 est donné dans le listing 3.18.

3.4 Validation

3.4.1 Objectifs

Les objectifs des travaux rapportés dans ce chapitre sont, d'une part, de proposer des abstractions de haut niveau, éventuellement utilisables côtés client et serveur, produisant du code efficace. Et, d'autre part, de proposer une méthode générale pour construire, à moindre effort, de telles abstractions efficaces et dédiées au développement d'applications Web.

```
def quote[A](a: => Rep[A]): Rep[String]
```

Listing 3.16 – Signature de la fonction `quote`

```
el('div, 'onclick -> quote { println("clicked") })("click me")
```

Listing 3.17 – Génération d'une chaîne de caractères contenant du code JavaScript

```
var x1 = function () {
  alert('clicked');
};
var x0 = document.createElement('div');
x0.setAttribute('onclick', 'x1()');
x0.textContent = 'click me';
```

Listing 3.18 – Code généré par le listing 3.17

3.4.2 Expérience

Pour évaluer si notre premier objectif est atteint, nous nous appuyons sur une validation empirique.

Mesures Nous utilisons, comme approximation inverse du niveau d'abstraction, le nombre de lignes de code que l'utilisateur doit écrire (plus un code est concis, plus il est haut niveau).

Nous mesurons l'efficacité d'un programme par sa vitesse d'exécution (plus un programme s'exécute vite, plus il est efficace).

Nous évaluons le niveau d'abstraction et l'efficacité de notre solution, que nous comparons aux autres approches actuelles permettant de partager du code entre les côtés client et serveur.

Hypothèses Les hypothèses testées sont les suivantes :

H1 l'utilisation de nos unités de langage montre de meilleures performances par rapport au nombre de lignes de code que les approches alternatives ;

H2 nos unités de langage sont utilisables côtés client et serveur.

Méthode Nous avons écrit plusieurs tests de performance sur des programmes ciblant l'utilisation d'unités de langages en particulier, et sur une application réelle. À chaque fois, nous avons écrit différentes implémentations du programme, en JavaScript, Java/GWT, HaXe et js-scala (avec nos unités de langages). Nous avons fait un effort pour respecter le style de programmation de chaque langage.

Les tests de performance ont été effectués sur un ordinateur portable DELL Latitude E6430 avec 8 Go de mémoire RAM, dans le navigateur Google Chrome version 27.

Les graphiques reflètent les types de mesures qui ont été effectuées : le premier groupe de mesures est la vitesse d'exécution du code JavaScript (plus l'indice est élevé, mieux c'est), le deuxième groupe mesure le nombre de lignes de code (plus l'indice est

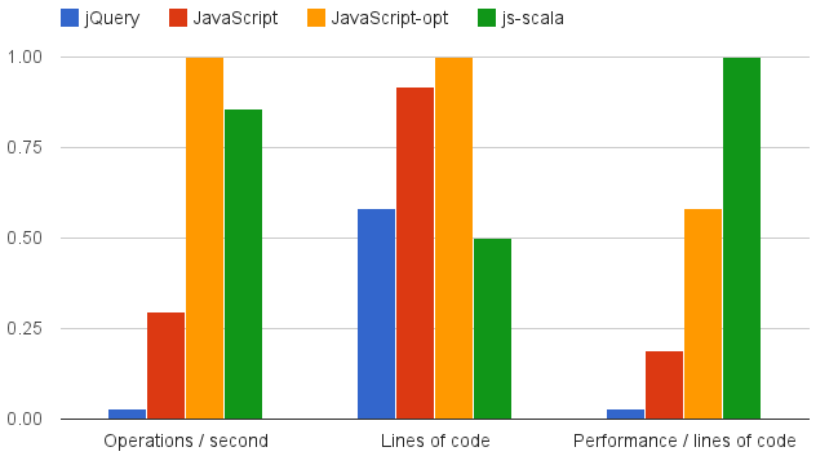


FIGURE 3.3 – Tests de performance sur un programme utilisant l'API des sélecteurs

bas, mieux c'est), et le troisième groupe montre le rapport entre la vitesse d'exécution et le nombre de lignes de code (plus l'indice est haut, mieux c'est). Les valeurs ont été normalisées de façon à ce que les graphiques puissent être affichés côte à côte sans problème d'échelle verticale.

3.4.2.1 Programmes ciblant des unités de langages

Nous avons écrit des petits programmes utilisant spécifiquement l'API des sélecteurs et de manipulation de valeurs optionnelles [1].

Sélecteurs Cette abstraction ne pouvant pas être implémentée avec GWT ou HaXe puisque ces langages ne fournissent pas de mécanisme d'évaluation partielle, nous avons comparé les temps d'exécution du code JavaScript généré par le listing 3.3 avec le temps d'exécution d'un programme similaire au listing 3.1, mais utilisant seulement les fonctions haut niveau `querySelector` et `querySelectorAll`. Le programme a été exécuté dans une page Web contenant les éléments suivants : quatre éléments `fieldset`, chacun contenant jusqu'à deux éléments ayant pour classe `word`.

La figure 3.3 montre les résultats du test de performance. La version JavaScript-opt correspond au listing 3.1 (qui utilise les fonctions bas niveau `getElementById`, `getElementsByTagName` et `getElementsByClassName`), la version JavaScript correspond au programme équivalent utilisant seulement l'API native haut niveau (c'est-à-dire

1. Le code des programmes utilisés pour réaliser les tests de performance est disponible à l'adresse suivante : https://github.com/js-scala/js-scala/tree/master/papers/gpce2013/benchmarks.

```
val maybe = fun { (x: Rep[Int]) =>
  some(x + 1)
}

def benchmark = for {
  a <- maybe(0)
  b <- maybe(a)
  c <- maybe(b)
  d <- maybe(c)
} yield d
```

Listing 3.19 – Programme manipulant des valeurs optionnelles

les fonctions `querySelector` et `querySelectorAll`) et la version jQuery correspond au listing 3.2.

La version js-scala est légèrement plus lente que la version JavaScript-opt (d'environ 14%), mais est 2.88 fois plus rapide que la version JavaScript, et 28.6 fois plus rapide que la version jQuery. Enfin, la version js-scala a un rapport performance sur nombre de lignes de code plus de 1.72 fois meilleur que les autres versions.

Valeurs optionnelles Nous avons implémenté l'abstraction de manipulation de valeurs optionnelles en JavaScript pur, Java et HaXe, et avons écrit un programme manipulant des valeurs optionnelles. Le listing 3.19 montre la version js-scala du programme. La fonction `maybe` est une fonction partiellement définie sur les valeurs de type `Int`.

La figure 3.4 montre les résultats du test de performance. La version js-scala est 3 à 10 fois plus rapide que les autres versions. Cette version est également la plus concise (cela provient essentiellement de la syntaxe `for`, qui n'a pas d'équivalent dans les autres langages). Finalement, la version js-scala a un rapport performance sur lignes de code plus de 4 fois supérieur aux autres versions.

3.4.2.2 Application réelle

Chooze [2] est une application complète pour effectuer des sondages en ligne. L'application permet aux utilisateurs de créer des sondages, de définir les différents choix possibles pour une question, de partager un sondage avec d'autres personnes, de voter et de consulter les résultats.

L'application contient du code JavaScript pour gérer le comportement dynamique de l'application : interdiction d'une double soumission d'un vote, mise à jour du formulaire de création de sondage, interactions avec l'interface utilisateur, etc.

La taille totale de l'application (incluant les côtés client et serveur) est de l'ordre du millier de lignes de code.

2. Le code source est disponible à l'adresse http://github.com/julienrf/chooze, dans les branches `vanilla`, `jquery`, `gwt`, `haxe` and `js-scala`.

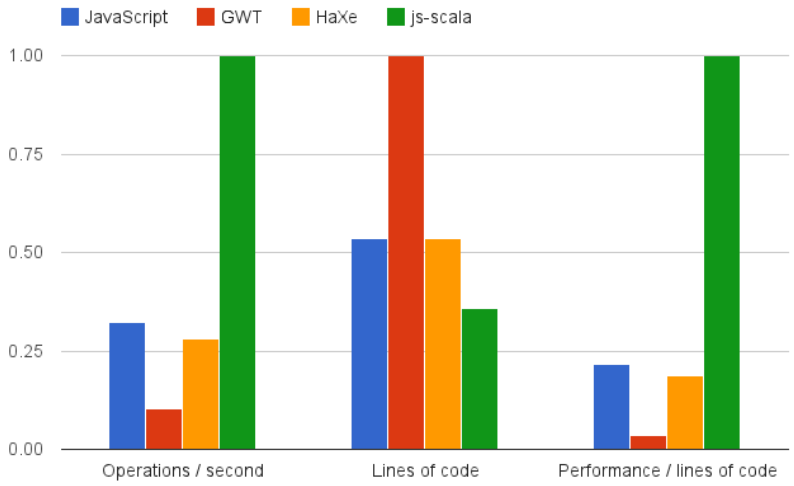


FIGURE 3.4 – Tests de performance sur un programme manipulant des valeurs optionnelles

L'application a été initialement écrite avec jQuery. Nous l'avons réécrite en JavaScript pur (version « vanilla », bas niveau et optimisé, sans utilisation de bibliothèque tierce), js-scala, GWT et HaXe.

Tests de performance Le test de performance a mesuré le temps d'exécution d'un scénario d'utilisation : 2000 clics sur un bouton ajoutant un champ de formulaire. Le code impliqué utilise les API de manipulation de valeurs optionnelles, de sélécteurs et de définition de contenus Web. La figure 3.5 montre les résultats du test.

Les performances d'exécution des versions vanilla, HaXe et js-scala sont similaires (bien que la version js-scala soit légèrement moins rapide, de 6%). Il est notable que les versions vanilla et HaXe utilisent toutes les deux du code bas niveau comparé à la version js-scala, comme le souligne la partie au milieu de la figure : la version js-scala tient en 74 lignes de code, tandis que la version vanilla tient en 116 lignes de code (57% plus longue) et la version HaXe tient en 148 lignes de code (100% plus longue). La version jQuery, dont le code est haut niveau (54 lignes de code, 27% plus concise que la version js-scala) est 10 fois plus lente que la version js-scala.

La dernière partie de la figure montre que la version js-scala dispose du meilleur rapport performances sur nombre de lignes de code. Ce rapport est 1.48 fois meilleur que pour la version vanilla, 1.88 fois meilleur que pour la version HaXe, 3.45 fois meilleur que pour la version GWT et 7.82 fois meilleur que pour la version jQuery.

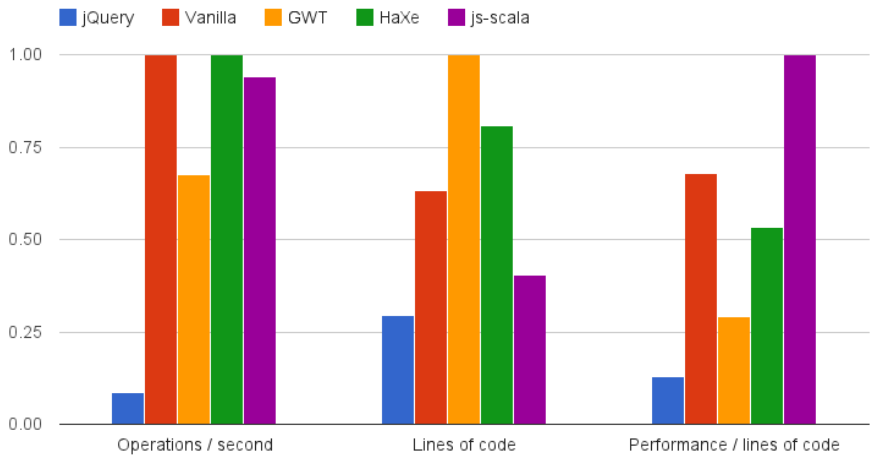


FIGURE 3.5 – Tests de performance sur une application réelle

Réutilisation de code La version js-scala du programme nous a permis de réutiliser quelques définitions de contenus Web entre les côtés client et serveur. Avec GWT nous n'avons pas le choix : le contenu des pages est forcément construit côté client. Dans les autres versions, le code définissant le contenu des pages est dupliqué entre les côtés client et serveur. Cela représente 20 lignes de code JavaScript (17% du total) et 15 lignes de HTML (5% du total) dans la version vanilla, 19 lignes de HaXe (13% du total) et 15 lignes de HTML (5% du total) dans la version HaXe. Dans la version js-scala, les définitions de fragments Web partagées entre les côtés client et serveur représentent 22 lignes de Scala (30% du total) et ont permis d'économiser 15 lignes de HTML (5% du total).

3.4.3 Discussion

3.4.3.1 Analyse des résultats

Nos résultats montrent que le code haut niveau écrit avec js-scala génère du code JavaScript bas niveau dont les performances d'exécution sont du même ordre de grandeur que du code JavaScript optimisé à la main.

Les tests ciblant l'utilisation d'une abstraction en particulier montrent de bons résultats, comme attendu. Les gains sur le rapport entre les performances d'exécution et le nombre de lignes de code varient d'un facteur 1.72 à 30 selon les technologies comparées.

Le test sur une application réelle permet de tenir compte de la place relative, dans une application complète, des gains individuels décrits au paragraphe précédent. Ce

test affiche tout de même un gain sur le rapport entre les performances d'exécution et le nombre de lignes de code variant d'un facteur 1.48 à 7.82.

Ces résultats valident l'hypothèse H1 : le rapport entre les performances et le nombre de lignes de code est meilleur avec nos unités de langage qu'avec les approches alternatives.

Nos tests montrent également qu'une partie du code d'une application réelle peut être partagée entre les parties client et serveur, ce qui valide l'hypothèse H2.

3.4.3.2 Limites de notre validation

Mesure du niveau d'abstraction Notre objectif est de mettre en perspective les performances d'exécution par rapport au niveau d'abstraction. Nous avons choisi le nombre de lignes de code comme indicateur de l'inverse du niveau d'abstraction. Or, ce choix n'est pas sientifiquement établi. Cependant, aucune autre métrique n'a été proposée par la communauté scientifique pour mesurer le niveau d'abstraction.

Différences de concision propres aux langages Nous comparons la concision du code dans plusieurs langages. Or, la syntaxe propre à chaque langage peut introduire des différences inter-langages indépendantes du niveau d'abstraction. Par exemple, la taille du programme Java utilisant le langage de manipulation de valeurs optionnelles pourrait être quasiment divisée par deux en utilisant les *lambda expressions* [Jav]. Cependant, il n'existe aucun travail scientifique fournissant un facteur de correction pour gommer ces différences.

Représentativité des tests effectués Il n'est pas correct d'extrapoler les résultats des tests de performance, obtenus sur l'application réelle, à toutes les applications Web. Cependant, l'application que nous avons utilisée était de taille modeste et contenait peu de code côté client. Nous pensons donc que sur des applications de taille plus grosses et implémentant plus de fonctionnalités côté client, les tests pencheraient encore plus en faveur de js-scala.

Respect du style idiomatique des technologies comparées Pour gérer l'interface utilisateur, la version GWT n'utilise pas la bibliothèque de composants fournie par GWT. Au lieu de cela, nous interceptons les événements directement au niveau du DOM, comme nous le ferions en JavaScript pur. Cependant, cette façon d'utiliser GWT n'a pas d'impact sur les performances et a seulement un impact marginal sur la concision du code.

3.5 Conclusion

Les abstractions de haut niveau permettent à la fois de réduire la complexité du code et, dans le cas des applications Web, de masquer les différences entre les environnements client et serveur. Cependant, outre le coût en performance des abstractions,

en masquant les différences d'environnements client et serveur l'utilisateur perd l'opportunité d'exploiter leurs spécificités.

En implémentant nos abstractions sous forme de bibliothèques générant du code, nous conservons les avantages apportés par les abstractions concernant le confort d'ingénierie. Mais nous avons également la possibilité de générer du code efficace, capable de tirer parti des spécificités des environnements client et serveur.

Bien que le coût initial de cette approche soit non négligeable (il faut implémenter des générateurs de code pour un langage généraliste), l'ajout d'abstractions dédiés à un problème donné nécessite peu d'efforts pour l'utilisateur : la définition d'une API, une implémentation retournant une représentation abstraite de programme, et un générateur de code pour cette représentation abstraite, pour chaque plateforme ciblée. Tous ces composants peuvent être définis de façon indépendante, sans qu'il ne soit nécessaire de modifier le code d'un compilateur existant.

Ainsi, nous avons implémenté des unités de langage pour rechercher des éléments dans une page Web, manipuler des valeurs optionnelles, et représenter des contenus Web. Le code écrit avec les deux dernières unités de langage peut être partagé côtés client et serveur. Ces abstractions produisent du code exploitant les spécificités des environnements client et serveur. Enfin, les performances d'exécution sont du même ordre de grandeur que du code bas niveau optimisé à la main.

Chapitre 4

Sûreté du typage

Ce chapitre présente une solution au problème de l'exposition des interfaces de programmation du navigateur Web dans un langage statiquement typé. Nous commençons par détailler les limites des solutions actuelles puis nous présentons deux solutions préservant un maximum d'informations dans les types tout en donnant autant de pouvoir d'expression aux utilisateurs. La seconde solution est plus pratique pour l'utilisateur mais s'appuie sur le concept de *types dépendants*, peu répandu dans les systèmes de type des langages généralistes.

Le contenu de ce chapitre a fait l'objet d'une publication [RFBJ14].

4.1 Inventaire des fonctions problématiques

Illustrons les difficultés à définir des signatures de type pour des APIs dynamiquement typées en prenant comme exemples quelques-unes des principales [1] fonctions de l'API du navigateur : `createElement`, `getElementsByTagName` et `addEventListener` [2].

4.1.1 `createElement`

Cette fonction, ainsi que les difficultés pour lui donner une signature de type dans un langage statiquement typé, ont été présentées dans les chapitres 1 et 2.

La suite de cette section détaille les différentes signatures de type proposées pour cette fonction par les langages de programmation statiquement typés courants.

4.1.1.1 Signatures de type proposées par les langages actuels

API trop générale La première solution consiste simplement à conserver la même signature de type et à imposer au programmeur d'effectuer une opération de trans-

1. D'après le Mozilla Developer Network (https://developer.mozilla.org/en-US/docs/DOM/DOM Reference/Introduction).

2. Nous présentons seulement les fonctions pour lesquelles il est difficile de définir une signature de type.

typage (*type cast*) pour obtenir une valeur d'un type plus précis. Une telle solution ressemblerait, à l'utilisation, en Scala, au listing suivant :

```
val img = createElement("img").asInstanceOf[ImageElement]
img.src = "/icon.png" // Ok
img.value = "foo" // Erreur: propriete 'value' non definie
```

Évidemment, l'utilisation d'opérations de transtypage introduit une fragilité allant à l'encontre de l'utilisation même d'un langage statiquement typé.

API trop spécifique Une autre solution consiste à définir des fonctions distinctes, retournant chacune un type d'élément spécifique, plutôt qu'une seule fonction avec pour type de retour `Element`. Ainsi, au lieu d'avoir une seule fonction, `createElement`, il s'agit d'exposer autant de fonctions qu'il y a de types d'éléments possibles, par exemple `createImgElement`, `createInputElement`, etc :

```
def createImgElement(): ImageElement
def createInputElement(): InputElement
etc.
```

L'utilisation d'une telle solution est illustrée par le listing suivant :

```
val img = createImgElement()
img.src = "/icon.png" // Ok
img.value = "foo" // Erreur: propriete 'value' non definie
```

L'inconvénient de cette solution, évoqué dans la section 2.3 est qu'elle réduit le pouvoir d'expression de l'API. En effet, cette solution utilise des fonctions plus spécifiques que la fonction initiale ce qui rend impossible l'implémentation de certains programmes, qu'il est pourtant possible d'implémenter avec la fonction initiale, plus générale.

4.1.2 `getElementsByTagName`

4.1.2.1 Description

La fonction `getElementsByTagName` prend en paramètre un nom d'élément HTML et retourne l'ensemble des éléments de ce type dans le document. Sa signature est la suivante :

```
HTMLCollection getElementsByTagName(String name);
```

Le type `HTMLCollection` représente un tableau d'`Elements` HTML.

Le problème rencontré avec cette fonction est similaire à celui posé par la fonction `createElement` : les éléments retournés ont un type trop général. En effet, l'expression `getElementsByTagName('input')` retourne un tableau de `InputElement` mais la signature de type n'expose que le type général `Element`.

4.1.2.2 Signatures de type proposées par les langages actuels

Les langages statiquement typés conservent la signature de type standard pour cette fonction. Ils exposent donc un type de retour trop général, obligeant les utilisateurs à effectuer des opérations de transtypage sur le résultat.

4.1.3 `addEventListener`

4.1.3.1 Description

La fonction `addEventListener` permet de réagir aux événements produits par l'utilisateur d'une application Web. Elle prend en paramètre un type d'événement et une fonction à appeler lorsqu'un tel événement se produit :

```
void addEventListener(String type, Function callback);
```

Le type `Function` est peu précis, il représente une fonction prenant un nombre arbitraire de paramètres dont les types sont également arbitraires.

Le listing suivant montre comment l'utiliser, en JavaScript, pour réagir aux clics et appuis de touches de l'utilisateur :

```
document.addEventListener('click', function (event) {
  console.log(event.button);
});
document.addEventListener('keydown', function (event) {
  console.log(event.key);
})
```

Ce listing affiche les valeurs de la propriété `button` de l'événement représentant chaque clic et de la propriété `key` de l'événement représentant chaque appui de touche. Ici encore, dans un monde statiquement typé, le code ci-dessus produirait une erreur de typage car le type de l'objet `event` est indéfini, donc il ne possède pas de propriétés `button` ou `key`.

Notons que la propriété `button` est définie par le type `MouseEvent` et la propriété `key` est définie par le type `KeyboardEvent`. Ainsi, on aimerait, idéalement, trouver une signature de type pour la fonction `addEventListener` telle que le listing suivant, en Scala, compile sans erreur, sauf l'avant-dernière ligne :

```
document.addEventListener("click", event => {
  println(event.button)
  println(event.key) // Erreur: propriete 'key' non definie
})
```

De même, le listing suivant doit compiler, sauf l'avant-dernière ligne :

```
document.addEventListener("keydown", event => {
  println(event.key)
  println(event.button) // Erreur: propriete 'button' non definie
})
```

Autrement dit, le type de l'objet `event` doit être `MouseEvent` lorsque l'utilisateur réagit à l'événement `"click"`, et `KeyboardEvent` lorsque l'utilisateur réagit à l'événement `"keydown"`.

4.1.3.2 Signatures de type proposées par les langages actuels

API trop générale Comme précédemment, la première solution consiste à conserver une signature de type trop générale et donc à imposer à l'utilisateur d'effectuer une opération de transtypage pour obtenir une valeur d'un type plus précis. Une telle signature de type serait la suivante, en Scala :

```
def addEventListener(event: String, callback: Event => Unit): Unit
```

Cette solution souffre des même inconvénients que ceux décrits précédemment.

API trop spécifique Une autre solution consiste à définir des fonctions distinctes pour réagir aux différents types d'événements plutôt qu'une seule fonction trop générale :

```
def addClickEventListener(callback: MouseEvent => Unit): Unit
def addKeyDownEventListener(callback: KeyboardEvent => Unit): Unit
etc.
```

L'inconvénient de cette solution est qu'elle réduit le pouvoir d'expression de l'API. En effet, cette solution utilise des fonctions plus spécifiques que la fonction initiale, rendant impossible l'implémentation de certains programmes, qu'il est pourtant possible d'implémenter avec la fonction initiale, plus générale.

Par exemple, le programme JavaScript suivant ne peut pas être implémenté :

```
var observe = function (type) {
  return function (callback) {
    document.addEventListener(type, callback);
  }
};
```

Ce programme définit une fonction `observe` réalisant une application partielle des paramètres de la fonction `addEventListener` [3].

Une variante rencontrée consiste à définir un type abstrait `EventListener`, ne comportant aucune opération, et autant de sous-types qu'il y a de genres d'événements possibles, chacun comportant une seule opération abstraite correspondant au `callback`. Il est ensuite possible de définir une seule opération, `addEventListener`, prenant en paramètre un `EventListener` :

```
sealed trait EventListener
abstract class ClickListener(callback: MouseEvent => Unit) extends EventListener
abstract class KeyDownListener(callback: KeyboardEvent => Unit) extends EventListener
etc.
```

3. Le code de cette fonction est inspiré des bibliothèques JavaScript existantes de programmation fonctionnelle réactive : Rx.js [LB11] et Bacon.js [Bac]

```
def addEventListener(listener: EventListener): Unit
```

Cette solution souffre du même problème de réduction du pouvoir d'expression de l'API initiale du navigateur.

4.1.4 Discussion

Les solutions actuellement implémentées par les langages de programmation statiquement typés produisant du JavaScript soit réduisent le pouvoir d'expression soit ne sont pas bien typées.

4.2 Solutions bien typés et conservant le même pouvoir d'expression

4.2.1 Généralisation des solutions existantes

Les solutions bien typées existantes réduisent toutes le pouvoir d'expression de l'utilisateur. Elles ont toutes un point commun : elles réduisent le nombre de paramètres passés par l'utilisateur. Par exemple, au lieu d'écrire :

```
addEventListener("click", callback)
```

l'utisateur écrit :

```
addEventListener(new ClickListener(callback))
```

ou :

```
addClickEventListener(callback)
```

Dans les deux cas, le paramètre correspondant au nom de l'événement, `"click"`, disparaît. De même, au lieu d'écrire :

```
createElement("img")
```

l'utilisateur écrit :

```
createImgElement()
```

Le paramètre correspondant au nom de l'élément n'est plus présent.

Cette solution permet de distinguer entre les différents noms d'événements ou d'éléments HTML en utilisant des noms de fonction différents. Chaque nom correspond à une fonction dont la signature expose des types plus précis que la fonction, plus générale, de l'API native du navigateur. Plus précisément, cette solution remplace un paramètre, dont la signature de type de la fonction dépend, par un nom constant. C'est parce que cette solution élimine un paramètre de la fonction initiale qu'elle est moins expressive.

Notons que cette solution ne fonctionne que parce que la plage de valeurs possibles pour ces paramètres est connue à l'avance et fixée : la fonction `createElement` ne prend en paramètre que des noms d'éléments HTML. De même, la fonction `addEventListener` ne prend en paramètre que des noms d'événements.

4.2.2 Utilisation des types paramétrés

Notre but est de trouver, pour les fonctions présentées précédemment, une signature de type qui ne réduise pas le pouvoir d'expression et qui soit bien typée (ou qui ne contraigne pas l'utilisateur à écrire du code mal typé).

Il est possible de représenter la relation de dépendence entre un type `T` et un paramètre `p` impliqués dans une fonction en procédant comme suit :

— définir un type paramétré `P[U]`,
— affecter le type `P[U]` au paramètre `p`,
— remplacer les occurrences de `T` par `U`,
— définir autant de valeurs de type `P[U]` qu'il y a de valeurs possibles pour `p`, chacune fixant le paramètre de type `U` à un type plus précis.

L'application de cette solution aux fonctions `createElement` et `getElementsByTagName` est illustrée par le code suivant :

```
class ElementName[E]

def createElement[E](name: ElementName[E]): E
def getElementsByTagName[E](name: ElementName[E]): Array[E]

val Img = new ElementName[ImageElement]
val Input = new ElementName[InputElement]
// etc. pour chaque nom d'element possible
```

Les deux fonctions, `createElement` et `getElementsByTagName`, ont leur type de retour (`Element` ou `Array[Element]` dans l'API initiale) qui dépend de la valeur d'un paramètre `name`. Nous introduisons un type `ElementName[E]`, et attribuons ce type, plutôt que `String`, au paramètre `name`. Nous attribuons également le type `E` plutôt qu'`Element` au type de retour de la fonction (ou `Array[E]` plutôt que `Array[Element]`, dans le cas de `getElementsByTagName`). Enfin, nous définissons autant de valeurs, `Img`, `Input`, etc. qu'il y a de types d'éléments HTML possibles.

L'idée essentielle de notre solution consiste à utiliser des types différents pour représenter les différents noms d'éléments HTML (*i.e.* le type `ElementName[ImageElement]` pour l'élément `img`, le type `ElementName[InputElement]` pour l'élément `input`, etc.), et à s'appuyer sur un mécanisme (les types paramétrés) permettant de retrouver le type d'un élément à partir du type du nom de l'élément : le type `ElementName[E]` définit la relation entre un nom d'élément HTML et son type. Le paramètre de type `E` est parfois appelé *type fantôme* car les valeurs de type `ElementName[E]` ne contiennent jamais de valeur de type `E` [LM99].

Le listing suivant montre l'utilisation de notre solution :

```
val img = createElement(Img)
img.src = "/icon.png" // Ok
img.value = "foo" // Erreur
```

L'utilisation de la valeur `Img` fixe le type du paramètre `name` à `ElementName[ImageElement]`, et, par conséquent, fixe le type de retour de l'appel à la fonction `createElement` à `ImageElement`.

Ainsi, notre solution est bien typée.

En outre, notre solution ne réduit pas le pouvoir d'expression de l'utilisateur car elle ne réduit pas le nombre de paramètres de la fonction. Il est donc possible d'implémenter la fonction `createTwoElements` comme suit :

```
createTwoElements[A, B](fst: ElementName[A], snd: ElementName[B]): (A, B) =
  (createElement(fst), createElement(snd))
```

La même solution est applicable à la fonction `addEventListener` :

```
class EventName[E]

def addEventListener[E](name: EventName[E])(callback: E => Unit): Unit

val Click = new EventName[MouseEvent]
val KeyDown = new EventName[KeyboardEvent]
// etc. pour chaque nom d'evenement possible
```

L'utilisation est la suivante :

```
addEventListener(Click) { event =>
  println(event.button)
}
```

Comme précédemment, notre solution est bien typée et donne autant de pouvoir d'expression à l'utilisateur. Nous sommes ainsi capable d'implémenter la fonction `observe` :

```
def observe(name: EventName[A]): (A => Unit) => Unit =
  callback => addEventListener(name)(callback)
```

Notre solution comporte cependant un inconvénient à l'utilisation : toute fonction prenant en paramètre un nom d'élément HTML ou un nom d'événement doit également prendre un paramètre de type (un type fantôme), correspondant au type de l'élément HTML ou de l'événement désigné. En effet, la fonction `createTwoElements` prend deux paramètres de type, `A` et `B`, et la fonction `observe` prend un paramètre de type, `A`.

4.2.3 Utilisation des types dépendants

Cette section montre comment nous pouvons nous débarrasser des type fantômes introduits dans la section précédente en utilisant des types *chemin-dépendants* [OCRZ03]. Plus précisément nous représentons les paramètres de type en utilisant des types *membres*, comme cela a été décrit dans [OZ05].

4.2.3.1 Types membres

Les types membres étant peu présents dans les langages de programmation généralistes, commençons par rappeler ce concept.

Les langages de programmation gèrent en général deux mécanismes généraux d'abstraction : les paramètres et les membres abstraits. Par exemple, le langage Java gère

la paramétrisation des valeurs (paramètres de méthodes) et des types (*generics*) et les membres abstraits pour les valeurs (méthodes abstraites).

Illustrons la différence entre les paramètres et les membres par un exemple. Supposons que l'on veuille écrire une classe réalisant une addition entre deux nombres. Nous souhaitons que notre classe soit capable d'additionner deux nombres quels qu'ils soient. Autrement dit, nous souhaitons que les nombres à additionner soit abstraits pour la classe. Nous pouvons réaliser cela en écrivant une méthode paramétrée par les nombres à additionner, comme suit :

```
class Plus {
  public int apply(int a, int b) {
    return a + b;
  }
}
```

Une autre façon de faire, tout à fait équivalente, consiste à utiliser des membres abstraits pour représenter les nombres à additionner :

```
abstract class Plus {
  abstract int a();
  abstract int b();
  public int apply() {
    return a() + b();
  }
}
```

Ainsi, les paramètres et les membres sont deux moyens de généralisation équivalents. Dans le cas de Java, le langage gère à la fois les paramètres et les membres pour généraliser au niveau valeur, mais il ne gère que les paramètres pour généraliser au niveau type (les *generics*).

Le langage Scala, quant à lui, gère à la fois les paramètres et les membres pour généraliser au niveau type [CGLO06, OCRZ03]. Illustrons ce concept avec, par exemple, un type représentant une collection d'objets, générique, contenant des objets ayant tous le même type. Nous pouvons représenter une telle collection avec le type suivant :

```
trait Collection[A] {
  def get(i: Int): A
}
```

Le paramètre de type `A` représente le type des éléments contenus dans une collection donnée. Par exemple, le type `Collection[Int]` représente une collection spécifique, contenant uniquement des nombres entiers.

Nous pouvons représenter la même collection générique en utilisant un type membre à la place du paramètre de type `A` :

```
trait Collection {
  type A
  def get(i: Int): A
}
```

Nous pouvons spécialiser le type `Collection` pour ne contenir que des nombres entiers en instanciant le type membre `A` comme suit :

```
trait IntCollection extends Collection {
  type A = Int
}
```

L'instanciation d'un type membre est l'équivalent au niveau type de l'implémentation d'une méthode abstraite au niveau valeur.

Enfin, il est possible de faire référence au type membre `A` depuis l'extérieur du trait `Collection` comme suit :

```
def head(collection: Collection): collection.A =
  collection.get(0)
```

La méthode `head` récupère le premier élément d'une collection donnée. Son type de retour, `collection.A`, correspond à l'instance du type membre `A` de la collection passée en paramètre.

Plus généralement, un type membre abstrait d'une classe est un type interne qui peut être utilisé pour qualifier des valeurs et qui peut être instancié par des sous-classes. Depuis l'extérieur de la classe, on peut désigner un type membre en effectuant une sélection de type sur une instance de la classe : l'expression `p.C` désigne le type membre `C` d'une valeur `p`, et correspond à l'instance de `C` du type singleton de la valeur `p`.

4.2.3.2 Utilisation des types membres

Le type `ElementName[E]` peut être réécrit à l'aide d'un type membre comme suit :

```
abstract class ElementName { type Element }

val Img = new ElementName {
  type Element = ImageElement
}
val Input = new ElementName {
  type Element = InputElement
}
// etc.

def createElement(name: ElementName): name.Element
```

Nous avons remplacé le paramètre de type `E` par un type membre `Element`. Les valeurs `Img`, `Input`, etc. instancient ce membre en lui donnant le type d'élément HTML correspondant. Enfin, la fonction `createElement` retourne une valeur de type `name.Element`, correspondant à l'instance du membre `Element` pour leur paramètre `name`.

Utilisation :

```
val img = createElement(Img)
img.src = "/icon.png" // Ok
img.value = "foo" // Erreur
```

Cette représentation conserve autant de pouvoir d'expression que la précédente, comme en atteste l'implémentation suivante de la fonction `createTwoElements` :

```
def createTwoElements(fst: ElementName,
                      snd: ElementName): (fst.Element, snd.Element) =
  (createElement(fst), createElement(snd))
```

Enfin, le même procédé s'applique au type `EventName` :

```
trait EventName {
  type Event
}

object Click extends EventName { type Event = MouseEvent }
// etc. pour chaque nom d'evenement possible

def addEventListener(name: EventName)
                    (callback: name.Event = > Unit): Unit

def observe(name: EventName): (name.Event => Unit) => Unit =
  callback => document.addEventListener(name)(callback)
```

Notons que les fonctions `createTwoElements` et `observe` ne comportent plus de types fantômes.

4.3 Discussion

4.3.1 Analyse

Implémentation dans js-scala Nous avons implémenté, au sein du projet js-scala, une API basée sur la solution utilisant les types chemin-dépendants. Nous avons également implémenté plusieurs applications utilisant cette API, notamment une application de discussion instantanée et une application de sondages. La taille de ces applications est de l'ordre de plusieurs centaines de lignes de code. Du fait que notre API est bien typée, le code est exempt d'opérations de transtypage.

Clarté de l'API Notre solution offre une correspondance un pour un avec l'API native du navigateur. À titre de comparaison, les solutions existantes bien typées font souvent correspondre une seule fonction de l'API native à plus de 30 fonctions. Ainsi, GWT fait correspondre 31 fonctions spécialisées pour la fonction `createElement`, et Dart en fait correspondre 62. De même, GWT fait correspondre 32 fonctions spécialisées pour la fonction `addEventListener`, et Dart en fait correspondre 49. Notre solution a l'avantage d'être bien typée sans multiplier le nombre de fonctions ayant un rôle similaire.

Commodité syntaxique Les annotations de type des langages à typage statique peuvent être perçues par le développeur comme une complexité indésirable. Dans notre cas,

la solution s'appuyant sur des types paramétrés est utilisable avec de nombreux langages généralistes tels que Java, Dart, TypeScript, Kotlin, HaXe, Opa, Idris ou Elm. Cependant cette solution nécessite d'inclure des types fantômes, indésirables, dans les signatures des fonctions prenant en paramètre des noms d'événements ou d'éléments HTML. La seconde solution, en revanche, conduit à des signatures de fonctions d'une verbosité équivalente à celle de la spécification standard des API HTML et DOM. Nous estimons qu'il n'y a, dans ce cas, pas de prix syntaxique à payer. Cependant, cette solution n'est utilisable qu'avec des langages supportant les types dépendants, tels que Scala, Idris ou Agda.

4.3.2 Limitations

Nos solutions remplacent un paramètre de type `String` par un paramètre d'un type plus précis permettant de déduire le type de l'élément ou événement correspondant. Ce faisant, nos solutions diminuent légèrement le pouvoir d'expression par rapport à l'API native : il n'est plus possible, par exemple, de passer en paramètre une valeur résultant de la concaténation de deux chaînes de caractères ou d'une quelconque autre opération de manipulation de chaîne de caractères.

Nos solutions s'appliquent seulement aux fonctions dont la signature comporte des types qui dépendent de la valeur d'un des paramètres. Si ce genre de fonctions est commun dans les API JavaScript, ce n'est pas le cas de toutes les fonctions. Par exemple, la fonction `getElementById` ne peut pas bénéficier de notre solution. Cette fonction récupère un élément du document à partir de la valeur de son attribut `id`, or le type de l'élément ne peut pas être déduit depuis la valeur de cet attribut.

4.4 Conclusion

S'appuyer sur un langage statiquement typé produisant du JavaScript n'est pas suffisant pour profiter du typage statique pour le développement de la partie client d'une application Web. En effet, l'API du navigateur Web doit également être exposée dans ce langage et son système de type. Or, cela n'est pas facile à réaliser pour certaines fonctions et conduit généralement soit à perdre en sûreté de typage, soit à perdre en pouvoir d'expression.

Dans ce chapitre, nous avons présenté deux méthodes pour exposer les fonctions problématiques de cette API. La première méthode s'appuie sur les types paramétrés et est utilisable avec la plupart des langages de programmation courants (*e.g.* Java ou C#). La seconde méthode apporte un confort syntaxique supplémentaire et nécessite que le langage gère les types dépendants (*e.g.* Scala). Nos deux méthodes évitent aux développeurs d'avoir recours à des opérations de transtypage non sûres et conservent le même pouvoir d'expression que les fonctions natives.

Nos résultats suggèrent que la prise en charge des types dépendants peut être un atout pour les langages à typage statique ciblant la programmation Web.

Chapitre 5

Un style d'architecture pour les applications Web résilientes

Ce chapitre présente une approche pour développer des applications Web résilientes par construction. Nous nous appuyons, pour cela, sur un style d'architecture découpant le code de l'application de telle sorte qu'il soit à la fois facile pour les développeurs de raisonner dessus, et facile techniquement d'implémenter les différents aspects de façon réutilisable et indépendamment les uns des autres.

La suite de ce chapitre donne une vision générale du style d'architecture puis montre progressivement, en détaillant certaines parties, comment il répond à nos objectifs.

5.1 Présentation générale

La figure 5.1 donne une vue d'ensemble des différents composants impliqués dans notre style d'architecture. Nous nous appuyons sur plusieurs concepts présentés dans l'état de l'art. Les client et serveur contiennent tous les deux des composants simi-

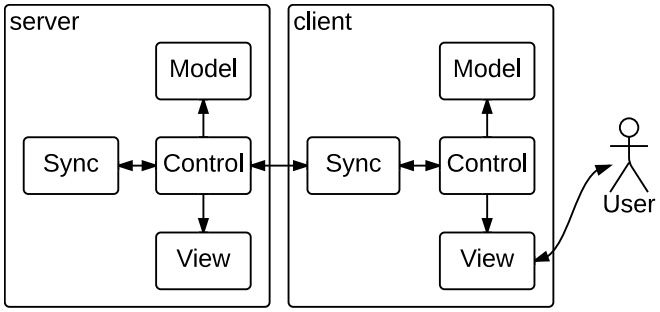


Figure 5.1 – Vue d'ensemble de notre style d'architecture pour applications Web résilientes

laires :

Model Données représentant l'état du système ;

View Affichage d'informations destiné aux utilisateurs ;

Control Interprétation des actions de l'utilisateur en termes d'actions métier (p. ex. un clic de l'utilisateur est traduit en une action de suppression de tâche), actualisation du $Model$ et de la $View$;

Sync Synchronisation de l'état du système entre clients et serveurs.

Les spécificités de cette architecture sont les suivantes :
— La communication client-serveur se situe au niveau $Control$ (tout comme dans le modèle PAC-C3D) plutôt que $Model$ (comme c'est le cas dans les modèles dérivées de MVC présentés dans l'état de l'art) ;
— Le composant de synchronisation côté client est intercalé entre les composants de contrôle client et serveur. Il joue un rôle de tampon en stockant localement toutes les actions réalisées par l'utlisateur avant de les synchroniser quand le contexte le permet ($i.e.$ quand la connexion est établie) ;
— L'état du système est représenté par la succession des actions effectuées par les utilisateurs, selon le principe d'$event\ sourcing$;
— La synchronisation est basée sur les actions, afin de profiter des algorithmes sophistiqués de résolution de conflits tels que TIPS ;
— Différentes stratégies de synchronisation basées sur les actions peuvent être employées (p. ex. Jupiter OT, TIPS, etc.) ;
— Chaque utilisateur fait évoluer sa propre réplique du système de façon optimiste (c'est-à-dire sans attendre l'aval du serveur après chaque action), et toutes les répliques convergent éventuellement ;
— Les composants $Model$ et $View$ sont optionnels côté serveur. En effet, comme l'état du système peut être représenté par une suite d'actions effectuées par les utilisateurs, un journal suffit pour le stocker. La partie $Model$ permet, en plus, de maintenir une représentation en mémoire de l'état du système, utilisable pour transmettre aux clients un $snapshot$ représentant le système. Cela permet de réduire le temps de reconstruction de la réplique de chaque client. La partie $View$, sur le serveur, permet d'exposer le contenu du système aux moteurs de recherche.

5.2 Fonctionnement du système de synchronisation

L'objectif du système de synchronisation est de permettre à l'utilisateur de continuer à utiliser l'application même lorsque la connexion réseau est interrompue, et de synchroniser automatiquement ses modifications lorsque la connexion est de nouveau établie. Cette section décrit son fonctionnement en allant du scénario le plus simple au plus complexe.

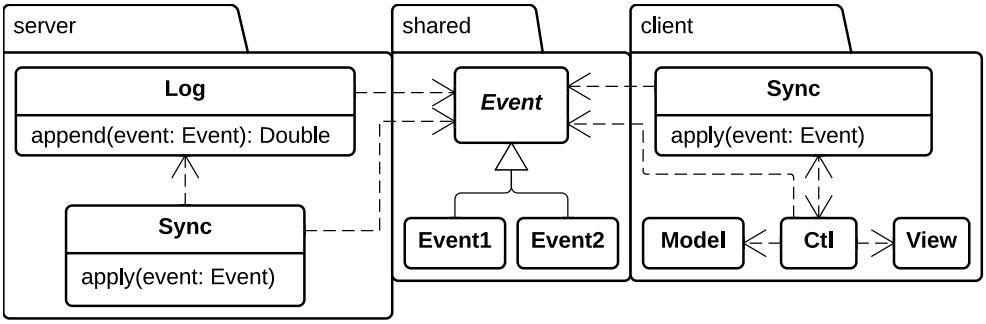


FIGURE 5.2 – Diagramme des classes impliquées dans la synchronisation

5.2.1 Scénario mono-utilisateur

La figure 5.2 montre le découpage, sous forme de diagramme de classes, d'une application basée sur notre architecture. Le diagramme montre seulement les classes impliquées dans le processus de synchronisation. La classe abstraite `Event` représente une action métier. Elle est spécialisée en autant de sous-classes que l'application gère d'actions métier (ici, `Event1` et `Event2`). Ces actions métier sont partagées par les côtés client et serveur. Côté serveur, un journal stocke les actions transmises par les différents utilisateurs.

Dans le cas d'une application de gestion de tâches, les `Events` possibles pourraient être les suivants : `Added` (une tâche est ajoutée), `Removed` (une tâche est supprimée) et `Toggled` (l'état d'une tâche passe de « non réalisée » à « réalisée », ou inversement). Les `Models` seraient `Task(content, isCompleted)` et `TaskList(tasks)`. Côté client, à chaque `Model` serait associé un `Control`, utilisant des `Views` pour l'affichage.

La figure 5.3 décrit le fonctionnement dynamique de l'application, dans un scénario mono-utilisateur, lorsque celui-ci effectue une action. Les interactions entre les différents composants sont les suivantes :

1. L'utilisateur interagit avec l'interface utilisateur (p. ex. il clique sur un bouton) ;
2. Le composant de contrôle traduit cet événement en action métier (ici, `Event1`) ;
3. Cette action métier est transmise au composant de synchronisation ;
4. Le composant de synchronisation, d'une part, execute la logique métier correspondant à cette action ;
5. La logique métier impacte le *Model* et la *View* est mise à jour ;
6. Le composant de synchronisation, d'autre part, transmet l'action au serveur, si la connexion réseau le permet. Autrement, l'action est stockée dans une file d'attente jusqu'à ce que la connexion réseau permette de la transmettre au serveur ;
7. Le serveur ajoute l'action à son journal ;
8. Le serveur indique au client que son action a bien été enregistrée. Le client supprime alors l'action de sa file d'attente.

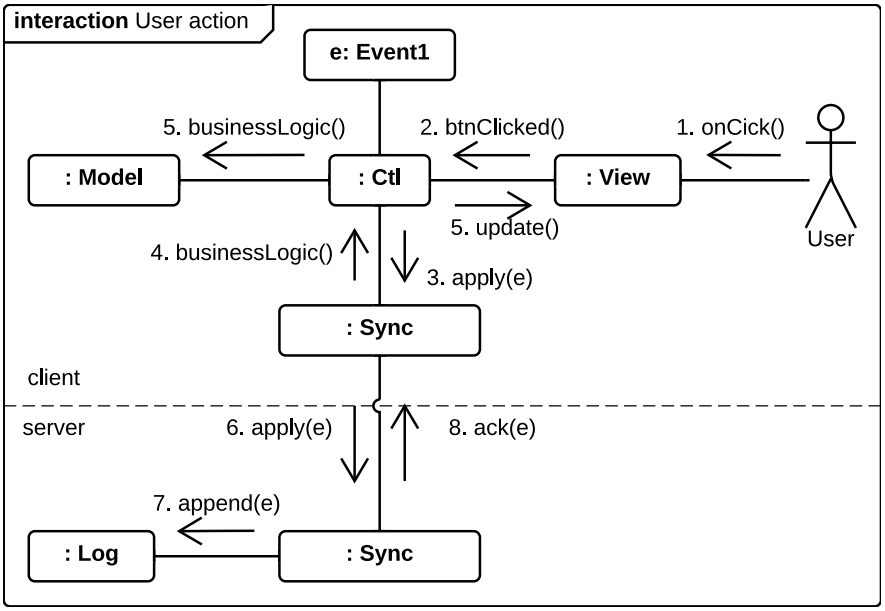


FIGURE 5.3 – Interactions entre les différents composants impliqués dans la synchronisation, dans un scénario mono-utilisateur

Cette architecture permet à l'utilisateur de modifier l'état du système sans avoir à attendre l'aval du serveur à chaque action : toutes ses actions sont enregistrées et transmises au serveur si la connexion réseau le permet.

5.2.2 Scénario multi-utilisateur

La prise en compte de plusieurs utilisateurs simultanés nécessite d'effectuer deux modifications : i) les clients doivent être notifiés lorsqu'un autre utilisateur a effectué une action, et ii) le système de synchronisation doit résoudre les éventuels conflits de modifications concurrentes.

La figure 5.4 illustre ces modifications. Jusqu'à l'étape 6 les interactions sont les mêmes que précédemment. Puis, lorsque le serveur reçoit l'action e d'un utilisateur, il la transforme éventuellement en une action e' (étape 7), selon l'algorithme de convergence utilisé. De même, il notifie les clients (étape 8) en transformant éventuellement l'action en une action e'', spécifique à chaque client, afin que l'état de tous les clients converge. Notons que pour certains algorithmes, la transformation de l'action a lieu côté serveur (p. ex. TIPS), et, dans d'autres cas, côté client (p. ex. Jupiter OT).

La logique métier correspondant à l'action est ensuite exécutée sur les clients notifiés (étapes 9 et 10).

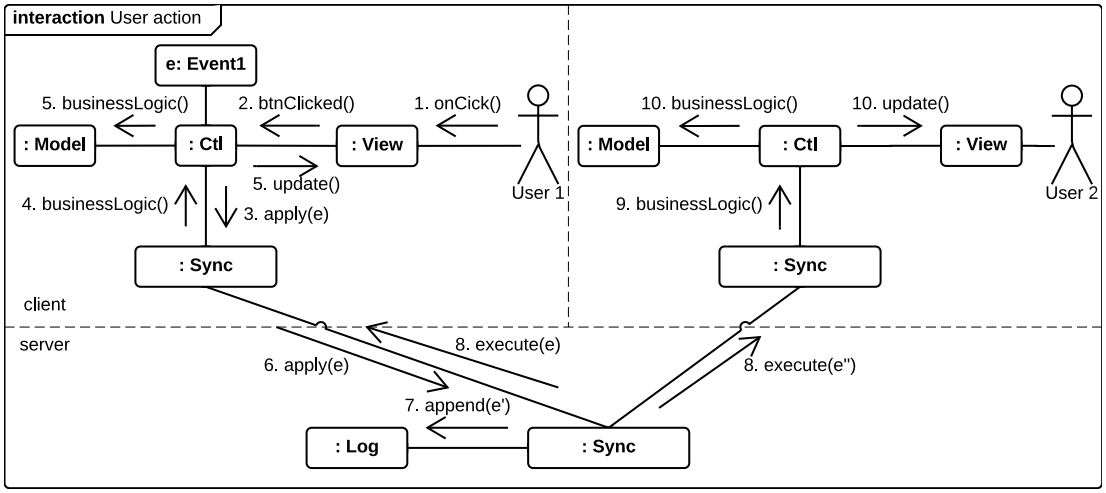


FIGURE 5.4 – Interactions entre les différents composants impliqués dans la synchronisation, dans un scénario multi-utilisateur

5.3 Généralisation du composant de synchronisation

La dernière étape de notre contribution consiste à généraliser le système de synchronisation de façon à isoler la stratégie de synchronisation du reste de l'application (afin de réutiliser une stratégie dans différentes applications).

La figure 5.5 montre cette évolution. Les classes en gris sont spécifiques à une application donnée, les classes en blanc sont plus générales, donc réutilisables. Le composant de synchronisation implémente une stratégie de synchronisation en particulier. Sa méthode abstraite `interprete` est implémentée par chaque application. Cette méthode définit la logique métier correspondant à la réalisation d'une action donnée. Ainsi, lorsqu'une action est traitée par un composant de contrôle, celui-ci la transmet au composant de synchronisation *via* la méthode `apply`, lequel appelle ensuite sa méthode `interprete` pour invoquer la logique métier associée (il s'agit du patron de conception *Template Method*).

Les classes en pointillés représentent des composants qui ne sont pas couplés au composant de synchronisation.

5.4 Évolution : fonctionnement complètement hors-ligne

L'architecture décrite précédemment gère les interruptions de réseau mais elle ne stocke pas les données du système de façon persistante. La connexion est donc nécessaire pour le chargement initial de l'application. Une fois l'application chargée dans le navigateur, si l'utilisateur actualise la page il a besoin de la connexion réseau. Autrement dit, le support du mode hors-ligne, tel que décrit précédemment, n'est effectif que si l'utilisateur a déjà chargé l'application dans son navigateur et qu'il ne ferme

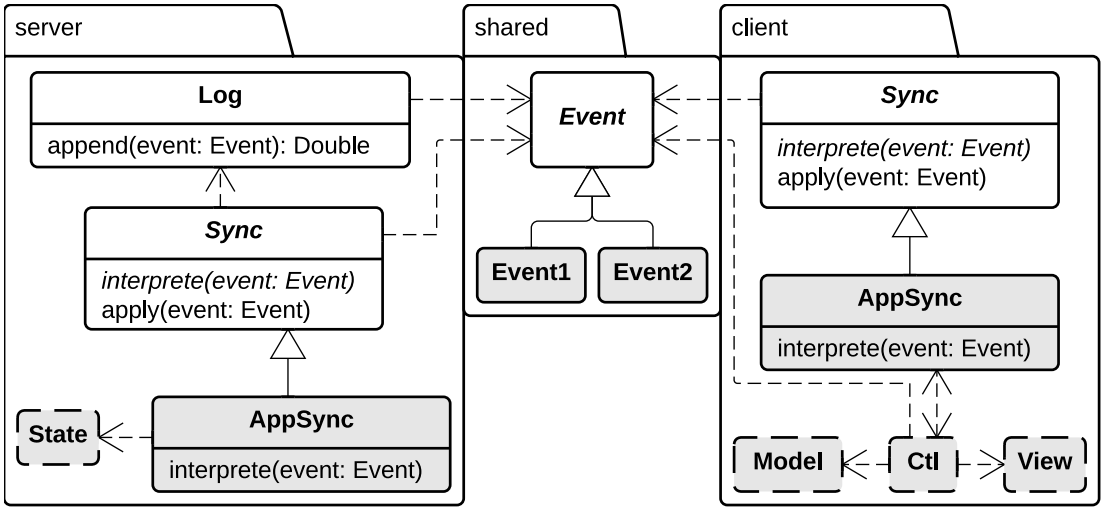


Figure 5.5 – Généralisation de la synchronisation

pas son onglet.

Il est techniquement possible de pallier ce problème en stockant de façon persistante le journal d'actions pour pouvoir reconstruire l'état de l'application et en utilisant le cache du navigateur pour que la connexion réseau ne soit pas nécessaire pour le chargement de l'application.

Le journal des actions est déjà stocké de façon persistante sur le serveur. Des solutions récentes de stockage persistant côté client telles que IndexedDB [Ind] permettent de stocker le journal côté client également. L'*Application Cache* [App] permet de stocker le code client de l'application dans le cache du navigateur, et donc permet de lancer l'application sans connexion réseau.

En utilisant ces deux systèmes de stockage, il est possible de gérer le fonctionnement complètement hors-ligne (une connexion réseau reste toutefois nécessaire pour télécharger l'application la toute première fois).

5.5 Validation

5.5.1 Objectifs

Le premier objectif de ce chapitre est de proposer un style d'architecture pour contsruire des applications Web résilientes. C'est-à-dire un style d'architecture qui encourage la séparation des différentes préoccupations impliquées dans ce type d'applications : l'interaction avec l'utilisateur, la synchronisation avec le serveur et les traitements métier. Cette séparation des préoccupation a elle-même un double objectif : permettre au développeur de se focaliser sur une seule préoccupation à la fois et rendre l'architecture plus modulaire (il doit être possible de modifier un aspect

indépendamment des autres).

5.5.2 Expérience

Hypothèses Les hypothèses que nous cherchons à valider sont les suivantes :

H1 la logique métier est effectivement isolée ;

H2 la préoccupation de résilience est effectivement isolée.

Méthode Nous avons écrit plusieurs applications suivant le style d'architecture décrit précédemment, notamment une application de gestion de tâches et un éditeur de texte.

Ces deux applications fonctionnent en mode complètement hors-ligne et se synchronisent avec le serveur lorsque la connexion réseau est active.

L'application d'édition de texte est l'exemple type d'applications illustrant le travail collaboratif assisté par ordinateur. Au total (côtés client et serveur, sans les commentaires), le code de l'application fait 566 lignes.

L'application de gestion de tâches est une implémentation du projet TodoMVC [Tod]. Ce projet vise à comparer les différents outils de construction d'applications Web interactives. Au total, le code de l'application fait 1344 lignes.

Le code prenant en charge la préoccupation de résilience et la stratégie de synchronisation est partagé entre les deux applications. Il est implémenté sous forme de deux bibliothèques réutilisables (une pour le côté client et une pour le côté serveur), et s'appuie sur les WebSockets pour la communication client-serveur. Le code de la bibliothèque côté client fait 334 lignes de JavaScript, CoffeeScript et Less. Le code de la bibliothèque côté serveur fait 178 lignes de Scala.

La logique métier est implémentée uniquement dans la partie *Model* : l'ajout ou la modification d'une fonctionnalité n'impacte que cette couche. Cependant, pour rendre la fonctionnalité accessible à l'utilisateur, il est nécessaire d'effectuer des modifications dans plusieurs parties du code. Il faut :

- réifier la fonctionalité en une action métier (c'est-à-dire ajouter un cas particulier d'*Event*) ;
- définir l'implémentation de cette action au niveau de la couche *Control*, laquelle s'appuie sur la couche *Model* ;
- définir les éléments d'interface utilisateur permettant d'utiliser la fonctionnalité, au niveau *View*.

5.5.3 Discussion

5.5.3.1 Analyse des résultats

Le découpage du code induit par notre style d'architecture conduit à séparer les préoccupations de la façon suivante :

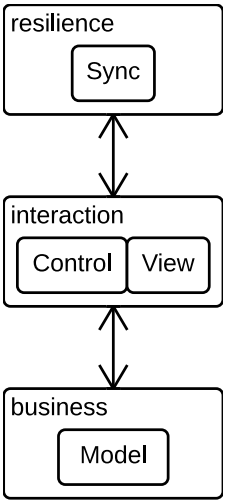


FIGURE 5.6 – Préoccupations des applications Web resilientes et responsabilités des différents composants de notre architecture

— La partie *Model* contient les données et fonctionnalités métier. Elle est complètement ignorante des préoccupations d'interaction ou de résilience, ce qui valide l'hypothèse H1 ;
— La partie *Control* rend le *Model* interactif. Cette partie est ignorante de la préoccupation de résilience ;
— La partie *Sync* rend l'application résiliente. Elle est ignorante de la logique métier ;
— La partie *View* affiche de l'information à l'utilisateur. Elle est ignorante de la logique métier et de la préoccupation de résilience.

Autrement dit, notre architecture permet de développer l'application sans se soucier de la préoccupation de résilience, laquelle est prise en charge par une bibliothèque réutilisable, ce qui valide l'hypothèse H2. La figure 5.6 illustre les différentes préoccupations et les responsabilités des composants.

Notons que le fait que l'ajout d'une fonctionnalité ait un impact à plusieurs endroits dans le code n'est pas en contradiction avec nos objectif. Cela est simplement une conséquence normale du fait qu'une fonctionnalité a plusieurs facettes : logique métier, affichage et interaction avec l'utilisateur.

5.5.3.2 Limites de notre validation

Notre architecture n'est appropriée que dans les situations où la logique d'une action peut être entièrement pourvue par le client. Or, dans certains cas (p. ex. la recherche d'information) la logique métier nécessite d'utiliser des informations qui ne sont disponibles que sur le serveur (les transférer côté client coûterait trop cher). Néanmoins, dans tous les cas où l'application sert à manipuler un document ou de

l'information (p. ex. un agenda, une liste de tâche, un éditeur de texte, de présentation, etc.), notre architecture est appropriée.

Comme indiqué en section 5.1, la construction de l'état de l'application en « rejouant » son histoire, c'est-à-dire en exécutant toutes les actions qui ont été enregistrées, peut être coûteuse et conduire à l'utilisation de *snapshots* pour accélérer le processus. En effet, les clients ont besoin de construire une représentation en mémoire de l'état de l'application (ou d'un partie de l'application), et avec le temps l'historique d'une application peut être particulièrement important. La solution consiste à envoyer aux clients un *snapshot* représentant l'état actuel du système. Ainsi, les clients peuvent directement partir de ce snapshot plutôt que de tout recalculer. Cependant, il n'existe pas, à ce jour, de système automatisant la sérialisation d'un tel *snapshot* de façon à ce que le système puisse être reconstruit côté client.

Notons également que, pour construire ces *snapshots*, la totalité du modèle de données doit tenir en mémoire côté serveur. Il est cependant possible de sectoriser le modèle de données de façon à ne traiter qu'une partie des données à la fois.

5.6 Discussion

Les applications Web s'appuient sur une connexion réseau pour fonctionner. Lorsque la connexion est intermittente, ou lente, comme cela peut notamment être le cas sur les appareils mobiles, la qualité de l'expérience utilisateur s'amoindrit, à moins de rendre l'application résiliente aux aléas du réseau. La préoccupation de résilience étant transversale, sa prise en charge pourrait augmenter la complexité du code des applications Web.

Nous avons défini un style d'architecture isolant cette préoccupation et permettant donc de construire des applications Web collaboratives résilientes. Les applications adhérant à cette architecture peuvent être utilisées en mode hors-ligne et leur expérience utilisateur n'est pas affectée par la latence du réseau.

L'architecture que nous proposons réduit la complexité de développement en définissant clairement les limites des responsabilités de chaque composant. La couche métier est ainsi ignorante des préoccupations d'interactivité ou de résilience. La couche gérant l'interactivité est ignorante de la préoccupation de résilience. Cette dernière est ignorante de la logique métier.

Ce faible couplage entre les différents composants de notre architecture nous a permis d'implémenter la gestion de la résilience sous forme d'une bibliothèque générique. En réutilisant cette bibliothèque, qui s'intègre facilement dans notre architecture, le développement d'une application Web résiliente demande peu d'efforts.

Enfin, notons que notre style d'architecture conduit à partager certains concepts entre les applications client et serveur : les actions métier et, dans le cas où le serveur maintient l'état du système en mémoire, la partie *Model*. Le chapitre 3 a présenté un moyen de partager du code entre clients et serveurs, pouvons-nous utiliser cet outil pour mutualiser les concepts partagés ?

Dans le cas des actions métier, c'est effectivement possible, bien que le gain soit

assez faible car les actions métier ne sont définies que par des données sans opération. Le code partagé n'est alors constitué que des constructeurs de types de données.

Dans le cas de la partie *Model*, le modèle de données et l'implémentation des transitions de l'état du système en fonction des actions métier sont partagés. Cependant, nous avons observé que, en pratique, nos implémentations utilisent des idiomes différents côtés client et serveur. Côté serveur, l'état du système est implémenté à l'aide de types de données immuables, ce qui permet d'implémenter les transitions de façon atomique (cela est nécessaire sur le serveur qui peut être accédé par plusieurs clients de façon concurrente). Côté client, le style est impératif car cela est plus idiomatique dans le langage JavaScript et que nous n'avons pas les mêmes contraintes de concurrence que sur le serveur.

Nous aurions pu, cependant, définir un langage commun pour décrire notre modèle de données et les transitions d'état, et définir comment générer le code correspondant côtés client et serveur, afin de mutualiser cet aspect de notre application. Cela aurait eu l'avantage de maintenir les applications client et serveur cohérentes entre elles, mais aurait demandé un effort d'implémentation relativement important pour un résultat non réutilisable (puisque propre au domaine d'une application donnée). Un bon compromis aurait pu être de définir un langage commun plus général de définition et manipulation de types de données. Un tel langage aurait pu être réutilisé pour modéliser le domaine métier et les transitions d'état de chaque application. Nous avons exploré cette voie sans toutefois atteindre un niveau de maturité suffisant.

Cependant, même en terminant ce travail le problème n'aurait été que partiellement résolu, une autre facette du problème est l'intégration du code généré dans le programme final. Le système de génération de code sur lequel nous nous sommes appuyé est conçu pour produire des programmes avec un seul point d'entrée, c'est-à-dire des fonctions. Or dans certains cas, il est plus commode d'intégrer des modules exposant plusieurs fonctions et comportant des dépendances vers d'autres composants. La différence n'est pas grande, et il est possible d'émuler les modules avec les fonctions ou de rendre le système de génération de code paramétrable. Néanmoins l'intégration, dans chaque variante de l'application, du code généré à partir du code mutualisé comporte une part de friction diminuant l'intérêt du processus de génération de code.

Finalement, nous pensons que, à part pour certains aspects comme la navigation et le contenu HTML, il n'est pas certain que les avantages de la mutualisation du code contrebalancent toujours les inconvénients de l'intégration du code généré.

Troisième partie

Conclusion et perspectives

Chapitre 6

Conclusion et perspectives

Ce chapitre présente une synthèse des contributions de cette thèse et discute les perspectives qu'elles ouvrent.

6.1 Conclusion

La distance, tant matérielle que logicielle, entre les postes ou environnements client et serveur est une source de complexité pour le développement d'applications Web.

La distance matérielle crée une préoccupation technique transverse : la résilience aux aléas du réseau. D'un côté, pour simplifier le processus d'ingénierie des applications Web, on souhaite que cette préoccupation soit gérée automatiquement, sans intervention du développeur. D'un autre côté, il n'est pas souhaitable de masquer l'aspect distribué des architectures Web : le développeur doit pouvoir préciser les modalités de synchronisation ou décider au cas par cas de la stratégie à suivre lorsque la connexion réseau est rompue.

La distance logicielle empêche le développeur de réutiliser du code entre les client et serveur et l'oblige à décider très tôt où exécuter le code de l'application. Ici encore, d'un côté, pour simplifier le processus d'ingénierie des applications Web, on souhaite que les développeurs puissent écrire le code de l'application et décider *a posteriori* d'exécuter ce code côté client ou serveur, voire partager ce code des deux côtés. D'un autre côté, il n'est pas souhaitable de masquer les spécificités des environnements client et serveur : le développeur doit pouvoir tirer parti des environnements natifs (notamment pour des raisons de performances).

Les travaux de cette thèse proposent des solutions pour réduire la complexité de développement des applications Web, sans pour autant diminuer le contrôle donné aux développeurs.

Concernant la distance logicielle, nous proposons un ensemble de bibliothèques Scala fournissant des abstractions dédiées à la programmation Web et produisant des programmes optimisés pour les environnements client et serveur. Cela permet aux développeurs de réutiliser du code entre les côtés client et serveur tout en tirant parti

des spécificités de ces environnements.

Notre deuxième contribution permet d'utiliser l'interface de programmation du navigateur depuis les langages de programmation statiquement typés sans diminution du pouvoir d'expression et de manière plus sûre que les autres approches.

Enfin, concernant la distance matérielle, nous proposons un modèle d'architecture isolant la préoccupation de résilience. Cela permet aux développeurs d'écrire les traitements métier de leur application sans avoir à gérer les aléas du réseau, ainsi que de réutiliser des implémentations de stratégies de synchronisation.

Mentionnons également que, en parallèle de ses travaux académiques, l'auteur de cette thèse a publié un livre sur l'ingénierie des applications Web avec le framework Play [RF14].

6.2 Perspectives

Nos trois contributions réduisent la complexité de développement des applications Web sans pour autant diminuer le contrôle donné aux développeurs.

Nous pensons qu'il est possible, et souhaitable, de prolonger notre effort afin de réduire encore d'autres sources de complexité. La suite de cette section présente les sources de complexité que nous avons identifiées et qui pourraient faire l'objet de travaux ultérieurs.

Navigations synchrone et asynchrone Nous avons mentionné le fait que les navigations synchrone et asynchrone ont toutes les deux des avantages. La navigation synchrone (le fait de recharger complètement la page en suivant un lien hypertexte) permet un meilleur référencement du contenu par les moteurs de recherche et diminue le temps d'affichage initial des pages. La navigation asynchrone (le fait de ne mettre à jour qu'une partie de la page en suivant un lien hypertexte) diminue le temps de transition d'une page à une autre.

Cependant, la gestion des deux types de navigation pose plusieurs problèmes. D'une part, les deux navigations doivent être cohérentes entre elles (elles doivent gérer les mêmes URLs désignant les mêmes ressources). Ce problème devrait toutefois être facilement être résolu avec un outil comme LMS.

D'autre part, le code côté client ou le modèle de données peuvent varier selon les pages. Par exemple, une page affichant une liste d'articles dans une boutique en ligne peut nécessiter du code pour gérer la pagination, et peut s'appuyer sur un modèle de données définissant une liste d'articles. Tandis qu'une page affichant un article seul n'a pas besoin du code gérant la pagination, et le modèle de données peut se limiter à la définition d'un seul article. Avec une navigation synchrone, chaque page contient le code nécessaire à son affichage. Mais avec une navigation asynchrone, le code et le modèle de données doivent pouvoir être mis à jour selon les besoins : le passage de l'affichage d'un article à une liste d'articles doit télécharger le morceau de code manquant et enrichir le modèle de données.

Éditeurs de données Malgré les bénéfices apportés par nos travaux, le développement de certaines classes d'applications Web reste très laborieux. C'est le cas des applications de gestion de données. Développer une interface Web pour gérer la saisie et l'affichage de données nécessite beaucoup d'étapes : l'affichage d'une interface utilisateur, la sérialisation des données pour leur transfert entre clients et serveurs et le retour d'informations à l'utilisateur. Automatiser ces étapes est difficile car chacune requiert des informations qui lui sont spécifiques (bien que toutes soient dédiées au même type de données) : libellés d'affichage, règles de validation, protocole de sérialisation, etc.

Toujours en respectant notre philosophie visant à automatiser sans diminuer le contrôle, il faudrait pouvoir générer un éditeur de données à partir d'un simple modèle de données, tout en conservant la possibilité d'affiner chaque aspect de l'éditeur (libellés, règles de validation, etc.) et de faire évoluer le modèle de données. Des travaux préliminaires ont déjà été réalisés dans cette direction [edi].

Générer des variantes d'applications natives pour téléphones Enfin, si le Web permet de définir du contenu visualisable sur des plateformes très variées, allant de l'ordinateur de bureau au téléphone, il comporte également quelques inconvénients. En effet, les applications Web manifestent généralement un surcoût en performances d'exécution par rapport aux applications natives. En outre, l'environnement du Web ne permet pas toujours d'accéder à toutes les fonctionnalités des terminaux clients. Pour ces raisons, ceux-là même qui naguère vantaient le Web (*e.g.* Google, Facebook) s'en détournent aujourd'hui, au moins dans le cas des téléphones mobiles, et investissent leurs efforts de développement dans des applications natives.

Or, le développement d'applications natives pour les téléphones mobiles conduit à une importante duplication d'efforts, car les environnements de ces terminaux peuvent être très différents (p. ex. il existe trois systèmes d'exploitation majeurs — iOS, Android et Windows Phone – qui ne gèrent pas tous les mêmes langages de programmation – Swift, Java et C#).

Des outils tels que Xamarin [Xam] permettent déjà de générer des applications natives pour les différentes plateformes mobiles du marché. Cependant, ces outils s'appuient sur les mêmes solutions techniques que GWT, et souffrent donc des mêmes limitations (en particulier l'impossibilité de définir des abstractions tirant parti des spécificités de l'environnement cible sans surcoût à l'exécution).

De façon analogue à notre travail permettant de mutualiser le code des applications client et serveur, nous pourrions définir un langage pour le développement d'applications sur téléphones mobiles, et générer des applications natives performantes, pour chaque plateforme.

Bibliographie

[App] Application cache. http://web.archive.org/web/20140816085956/http://www.whatwg.org/specs/web-apps/current-work/multipage//browsers.html. Accédé le 11 septembre 2014.

[Bac] Bacon.js. http://web.archive.org/web/20140802222641/http://baconjs.github.io/. Accédé le 18 août 2014.

[Bal06] Vincent Balat. Ocsigen : typing web interaction with objective caml. In *Proceedings of the 2006 workshop on ML*, pages 84–94. ACM, 2006.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP 2014–Object-Oriented Programming*, pages 257–281. Springer, 2014.

[BDM+13] Dominic Betts, Julian Dominguez, Grigori Melnik, Fernando Simonazzi, and Mani Subramanian. *Exploring CQRS and Event Sourcing : A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. Microsoft patterns &#38 ; practices, 1st edition, 2013.

[BK08] B. Bibeault and Y. Kats. *jQuery in Action*. Dreamtech Press, 2008.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language : Design and implementation. *Journal of Functional Programming*, 23(05) :552–593, 2013.

[Can08] Nicolas Cannasse. Using haxe. In *The Essential Guide to Open Source Flash Development*, pages 227–244. Springer, 2008.

[CGLO06] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *Mathematical Foundations of Computer Science 2006*, pages 1–23. Springer, 2006.

[CHJ12] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. *SIGPLAN Not.*, 47(10) :587–606, October 2012.

[CLWY07] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links : Web programming without tiers. In *Formal Methods for Components and Objects*, pages 266–296. Springer, 2007.

[Cou87] Joëlle Coutaz. Pac : An object oriented model for implementing user interfaces. *ACM SIGCHI Bulletin*, 19(2) :37–41, 1987.

[CSS] Css. http://web.archive.org/web/20140823082659/http://www.w3.org/Style/CSS/. Accédé le 27 août 2014.

[CW10] Brett Cannon and Eric Wohlstadter. Automated Object Persistence for JavaScript. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 191–200, New York, NY, USA, 2010. ACM.

[Cza12] Evan Czaplicki. Elm : Concurrent frp for functional guis, 2012.

[Doe13] Sébastien Doeraene. Scala. js : Type-directed interoperability with dynamically typed languages. Technical report, 2013.

[edi] editors. https://github.com/julienrf/editors. Accédé le 30 septembre 2014.

[EFDM00] Conal Elliott, Sigbjørn Finne, and Oege De Moor. Compiling embedded languages. In *Semantics, Applications, and Implementation of Program Generation*, pages 9–26. Springer, 2000.

[EG89] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Rec.*, 18(2) :399–407, June 1989.

[EMSS] Erik Ernst, Anders Møller, Mathias Schwarz, and Fabio Strocco. Managing gradual typing with message-safety in dart.

[Fen12] Steve Fenton. Typescript for javascript programmers. 2012.

[FF05] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 62–72, New York, NY, USA, 2005. ACM.

[Fow05] Martin Fowler. Event sourcing. *Online, Dec*, 2005.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern Web architecture. *ACM Trans. Internet Technol.*, 2 :115–150, May 2002.

[GF99] Rachid Guerraoui and Mohamed E Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4) :101–104, 1999.

[Gho10] Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[Hoa09] T. Hoare. Null references : The billion dollar mistake. *Presentation at QCon London*, 2009.

[HTM] Html. http://web.archive.org/web/20140702124213/http://www.w3.org/TR/domcore/. Accédé le 31 juillet 2014.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[Ind] Indexed database api. http://web.archive.org/web/20140906110326/http://www.w3.org/TR/IndexedDB/. Accédé le 11 septembre 2014.

[Jav] Lambda expressions. http://web.archive.org/web/20140625121030/http://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html. Accédé le 4 septembre 2014.

[JCB+13] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[JGS93] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[jQu] Utilisation de jquery. http://web.archive.org/web/20140704091353/https://trends.builtwith.com/javascript. Accédé le 2 septembre 2014.

[JS86] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM.

[JW07] Bruce Johnson and Joel Webber. *Google web toolkit*. Addison-Wesley, 2007.

[K+96] Samuel Kamin et al. Standard ml as a meta-programming language. Technical report, 1996.

[KARO12] Grzegorz Kossakowski, Nada Amin, Tiark Rompf, and Martin Odersky. Javascript as an embedded dsl. In *ECOOP 2012–Object-Oriented Programming*, pages 409–434. Springer, 2012.

[KHR+12] Sebastian Kleinschmager, Stefan Hanenberg, Romain Robbes, É Tanter, and Andreas Stefik. Do static type systems improve the maintainability of software systems ? an empirical study. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 153–162. IEEE, 2012.

[KLYY12] Yung-Wei Kao, ChiaFeng Lin, Kuei-An Yang, and Shyan-Ming Yuan. A Web-based, Offline-able, and Personalized Runtime Environment for executing applications on mobile devices. *Computer Standards & Interfaces*, 34(1) :212–224, 2012.

[Kot] Kotlin. http://web.archive.org/web/20140712061248/http://kotlin.jetbrains.org/. Accédé le 18 juillet 2014.

[KP88] G Krasner and S Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80 j. object oriented program., 1, 26-49, 1988.

[Lan66] Peter J Landin. The next 700 programming languages. *Communications of the ACM*, 9(3) :157–166, 1966.

[LB11] Jesse Liberty and Paul Betts. Reactive extensions for javascript. In *Programming Reactive Extensions and LINQ*, pages 111–124. Springer, 2011.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *ACM Sigplan Notices*, volume 35, pages 109–122. ACM, 1999.

[LV92] Ernst Lippe and Norbert Van Oosterom. Operation-based merging. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 78–87. ACM, 1992.

[Mar09] Nora Koch Marianne Busch. Rich Internet Applications. State-of-the-Art. Technical Report 0902, Ludwig-Maximilians-Universität München, 2009.

[MGPW13] Félix Albertos Marco, José A. Gallud, Victor M. Ruiz Penichet, and Marco Winckler. A model-based approach for supporting offline interaction with web sites resilient to interruptions. In *ICWE Workshops*, pages 156–171, 2013.

[MHR+] Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. Static type systems (sometimes) have a positive impact on the usability of undocumented software : An empirical evaluation. *self*, 18 :5.

[NCDL95] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[NS09] M.G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 133–143. IEEE, 2009.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003–Object-Oriented Programming*, pages 201–224. Springer, 2003.

[OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. *ACM Sigplan Notices*, 40(10) :41–57, 2005.

[Pur] Purescript. `http://web.archive.org/web/20140731132649/http://www.purescript.org/`. Accédé le 31 juillet 2014.

[RE09] Roberto Rodríguez-Echeverría. RIA : more than a Nice Face. In *Proceedings of the Doctolral Consortium of the International Conference on Web Engineering*, volume 484. CEUR-WS.org, 2009.

[RF14] Julien Richard-Foy. *Play Framework Essentials*. Packt Publishing Limited, Birmingham, UK, 1st edition, 2014.

[RFBJ13a] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In *Proceedings of the 12th international conference on Generative programming : concepts & experiences*, pages 53–60. ACM, 2013.

[RFBJ13b] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Efficient high-level abstractions for web programming. In Jaakko Jarvi and Christian

Kastner, editors, *Generative Programming : Concepts and Experiences, GPCE'13*, pages 53–60, Indianapolis, États-Unis, 2013. ACM.

[RFBJ14] Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel. Using Path-Dependent Types to Build Type Safe JavaScript Foreign Function Interfaces. In *ICWE - 14th International Conference on Web Engineering*, Toulouse, France, July 2014.

[Rif08] Jeremy Rifkin. The third industrial revolution. *Engineering & Technology*, 3(7) :26–27, 2008.

[Rom12] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers : Abstraction without Regret for High-Level High-Performance Programming*. PhD thesis, 2012.

[RSB+14] Tiark Rompf, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Hassan Chafi, and Kunle Olukotun. Surgical precision jit compilers. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 8. ACM, 2014.

[RT10] David Rajchenbach-Teller. Opa : Language support for a sane, safe and secure web. *Proceedings of the OWASP AppSec Research 2010*, 2010.

[SE98] Chengzheng Sun and Clarence Ellis. Operational Transformation in Real-time Group Editors : Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[SGL06] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop : a language for programming the web 2. 0. In *OOPSLA Companion*, pages 975–985, 2006.

[Sha] Sharpkit. `http://web.archive.org/web/20140716013416/http://sharpkit.net/`. Accédé le 18 juillet 2014.

[SLLG11] Bin Shao, Du Li, Tun Lu, and Ning Gu. An operational transformation based synchronization protocol for web 2.0 applications. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 563–572. ACM, 2011.

[SPBZ11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400. Springer, 2011.

[Tod] Todomvc. `http://web.archive.org/web/20140903120512/http://todomvc.com/`. Accédé le 11 septembre 2014.

[Vis08] Eelco Visser. Webdsl : A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.

[VP12] Markus Voelter and Vaclav Pech. Language modularity with the mps language workbench. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1449–1450. IEEE, 2012.

[VV10] Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–304. ACM, 2010.

[WL12] Kathy Walrath and Seth Ladd. *What is Dart?* " O'Reilly Media, Inc.", 2012.

[Xam] Xamarin. `http://web.archive.org/web/20140912234132/http://xamarin.com/`. Accédé le 30 septembre 2014.

[Yan00] Yun Yang. Supporting online Web-based teamwork in offline mobile mode too. In *Web Information Systems Engineering, 2000. Proceedings of the First International Conference on*, volume 1, pages 486–490. IEEE, 2000.

Table des figures

1.1 Un formulaire tel qu'il est perçu par l'utilisateur (en haut), est représenté par un arbre d'éléments HTML côté client, et par un texte côté serveur. 15

2.1 Les modèles MVC (à gauche) et PAC (à droite) 20
2.2 Le modèle PAC-C3D 21
2.3 Trois variantes d'architectures dérivées du modèle *Model-View-Controller*. a) MVC côté serveur ; b) MVC côté client ; et c) MVC hybride côtés client et serveur. 22
2.4 Exécution du programme indiqué dans le listing 2.1. Le programme est compilé vers les environnements client et serveur. Dans chaque environnement, son exécution produit une représentation abstraite d'un fragment HTML. Celle-ci est ensuite convertie vers une représentation spécifique à chaque plateforme. 28
2.5 Langages lourds. Le programme est compilé vers les environnements client et serveur. Dans chaque environnement, son exécution produit une représentation efficace du fragment HTML, en tirant parti des spécificités de chaque plateforme. 29
2.6 Ingénierie dirigée par les modèles. Chaque aspect du programme est décrit dans un langage dédié. La combinaison de ces aspects constitue le programme global, qui est transformé en une application côté client et une application côté serveur. 31
2.7 Langages dédiés compilés et implémentés par des bibliothèques. Un langage dédié est fourni par une bibliothèque dont l'implémentation produit une représentation abstraite du programme à générer. Un générateur de code peut dériver, à partir de cette représentation abstraite, les programmes correspondant, côtés client et serveur. 34
2.8 Différence entre les langages dédiés implémentés par des bibliothèques, interprétés (à gauche) ou compilés (à droite). 35
2.9 Diagramme d'objets correspondant au terme `trois`. 37

3.1 L'API standard des sélecteurs. 43

3.2 Représentations abstraites de code retournées par l'évaluation des expressions (a) `document.find("#add-user button")` et (b) `document.find("#add-user")` 46
3.3 Tests de performance sur un programme utilisant l'API des sélecteurs . 54
3.4 Tests de performance sur un programme manipulant des valeurs optionnelles 56
3.5 Tests de performance sur une application réelle 57

5.1 Vue d'ensemble de notre style d'architecture pour applications Web résilientes 73
5.2 Diagramme des classes impliquées dans la synchronisation 75
5.3 Interactions entre les différents composants impliqués dans la synchronisation, dans un scénario mono-utilisateur 76
5.4 Interactions entre les différents composants impliqués dans la synchronisation, dans un scénario multi-utilisateur 77
5.5 Généralisation de la synchronisation 78
5.6 Préoccupations des applications Web resilientes et responsabilités des différents composants de notre architecture 80

Résumé

L'automatisation de certaines tâches et traitements d'information grâce aux outils numériques permet de réaliser des économies considérables sur nos activités. Le Web est une plateforme propice à la mise en place de tels outils : ceux-ci sont hébergés par des serveurs, qui centralisent les informations et coordonnent les utilisateurs, et ces derniers accèdent aux outils depuis leurs terminaux clients (ordinateur, téléphone, tablette, etc.) en utilisant un navigateur Web, sans étape d'installation. La réalisation de ces applications Web présente des difficultés pour les développeurs. La principale difficulté vient de la *distance* entre les postes client et serveur.

D'une part, la distance physique (ou distance *matérielle*) entre les machines nécessite qu'une connexion réseau soit toujours établie entre elles pour que l'application fonctionne correctement. Cela pose plusieurs problèmes : comment gérer les problèmes de latence lors des échanges d'information ? Comment assurer une qualité de service même lorsque la connexion réseau est interrompue ? Comment choisir quelle part de l'application s'exécute sur le client et quelle part s'exécute sur le serveur ? Comment éviter aux développeurs d'avoir à résoudre ces problèmes sans pour autant masquer la nature distribuée des applications Web et au risque de perdre les avantages de ces architectures ?

D'autre part, l'environnement d'exécution est différent entre les clients et serveurs, produisant une distance *logicielle*. En effet, côté client, le programme s'exécute dans un navigateur Web dont l'interface de programmation (API) permet de réagir aux actions de l'utilisateur et de mettre à jour le document affiché. De l'autre côté, c'est un serveur Web qui traite les requêtes des clients selon le protocole HTTP. Certains aspects d'une application Web peuvent être communs aux parties client et serveur, par exemple la construction de morceaux de pages Web, la validation de données saisies dans les formulaires, la navigation ou même certains calculs métier. Cependant, comme les API des environnements client et serveur sont différentes, comment mutualiser ces aspects tout en bénéficiant des mêmes performances d'exécution qu'en utilisant les API natives ? De même, comment conserver la possibilité de tirer parti des spécificités de chaque environnement ?

Les travaux de cette thèse ont pour but de raccourcir cette distance, tant logicielle que matérielle, tout en préservant la capacité à tirer parti de cette distance, c'est-à-dire en donnant autant de contrôle aux développeurs.

La première contribution concerne la distance matérielle. Nous proposons un modèle d'architecture pour les applications interactives et collaboratives fonctionnant en mode connecté et déconnecté. Notre modèle isole la préoccupation de synchronisation client-serveur, offrant ainsi un modèle de développement plus simple pour les développeurs.

La seconde contribution concerne la distance logicielle. Nous nous appuyons sur un mécanisme d'évaluation retardée, permettant à un même programme d'être exécuté côté client ou serveur, pour bâtir des abstractions, offrant un modèle de programmation homogène et haut-niveau, produisant du code tirant parti des spécificités des environnements client et serveur. Nous avons observé que la taille du code

écrit avec nos outils est du même ordre de grandeur que celle d'un code utilisant des bibliothèques haut-niveau existantes et 35% à 50% plus petite que celle d'un code bas-niveau, mais les performances d'exécution sont du même ordre de grandeur que celles d'un code bas-niveau et 39% à 972% meilleurs qu'un code haut-niveau.

Une troisième contribution s'adresse aux adeptes du typage statique. En effet l'API du navigateur est conçue pour un langage dynamiquement typé, JavaScript, et certaines de ses fonctions peuvent être difficiles à exposer dans un langage statiquement typé. En effet, les solutions actuelles perdent en général de l'information, contraignant les développeurs à effectuer des conversions de type contournant le système de typage, ou proposent des fonctions avec un pouvoir d'expression réduit. Nous proposons deux façons d'exposer ces fonctions dans des langages statiquement typés, en nous appuyant respectivement sur les types paramétrés ou les types dépendants. Notre approche, tout en étant bien typée, permet de réduire le nombre de fonctions exposées aux développeurs tout en conservant le même pouvoir d'expression que l'API native.

Abstract

Thanks to information technologies, some tasks or information process can be automated, thus saving a significant amount of money. The web platform brings numerous of such digital tools. These are hosted on web servers that centralize information and coordinate users, which can use the tools from several kinds of devices (desktop computer, laptop, smartphone, etc.), by using a web browser, without installing anything. Nevertheless, developing such web applications is challenging. The difficulty mainly comes from the *distance* between client and server devices.

First, the physical distance between these machines requires them to be networked. This raises several issues. How to manage latency ? How to provide a good quality of service even when the network is down ? How to choose on which side (client or server) to execute a computation ? How to free developers from addressing these problems without yet hiding the distributed nature of web application so that they can still benefit from their advantages ?

Second, the execution environment is different between clients and servers. Indeed, on client-side the program is executed within a web browser whose API provides means of reacting to user actions and of updating the page. On server-side, the program is executed on a web server that processes HTTP requests. Some aspects of web applications can be shared between client and server sides (*e.g.* content display, form validation, navigation, or even some business computations). However, the APIs and environments are different between clients and servers, so how to share the same code while keeping the same execution performance as with native APIs ? How to keep the opportunity to leverage the specificities of a given platform ?

This work aims at shortening this distance while keeping the opportunity to leverage it, that is while giving developers as much expressive power.

Our first contribution consists of an architecture pattern to build interactive and

collaborative web applications handling on-line and off-line modes. Our pattern captures the client-server synchronization concern, thus giving developers a simpler programming model.

Our second contribution shows how to use a delayed evaluation mechanism to build high-level abstractions that can be shared between client and server sides and that generate efficient code leveraging the specificities of each platform. We observed that the size of the code written using our abstractions is similar to code that uses high-level libraries, and 35% to 50% smaller than low-level code, while execution performance are similar to low-level code and 39% to 972% faster than high-level code.

Our third contribution makes it easier to use the web browser's API from a statically typed language. Indeed, this API is designed to be used with the dynamically typed language JavaScript, and some functions are hard to encode in a static type system. Current solutions either loose type information, requiring users to perform unsafe typecasts or reduce the expressive power. We show two ways to encode challenging web browser's functions in a static type system by leveraging parameterized types and dependent types. Our approach is typesafe and keeps the same expressive power as the native API.