HAL
open science

Langage de mashup pour l'intégration autonomique de services de données dans des environements dynamiques

Mohamad Othman-Abdallah

To cite this version:

Mohamad Othman-Abdallah. Langage de mashup pour l'intégration autonomique de services de données dans des environements dynamiques. Base de données [cs.DB]. Université de Grenoble, 2014. Français. NNT : . tel-01011470

HAL Id: tel-01011470

https://theses.hal.science/tel-01011470

Submitted on 26 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Mohamad OTHMAN ABDALLAH

Thèse dirigée par **Christine COLLET** et
Co-encadrée par **Genoveva VARGAS-SOLAR**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale de Mathématiques, Sciences et Technologies de l'Information et de l'Informatique**

MELQART : un système d'exécution de mashups avec disponibilité de données

Thèse soutenue publiquement le **24 février 2014**,
devant le jury composé de :

Mme, Chirine GHEDIRA GUEGAN
Professeur, IAE Lyon, Rapporteur
Mme, Parisa GHODOUS
Professeur, Université Claude Bernard Lyon 1, Rapporteur
M. Omar BOUCELMA
Professeur, Université d'Aix Marseille, Examinateur
Mme Christine COLLET
Professeur, Grenoble INP, Directeur de thèse
Mme Genoveva VARGAS-SOLAR
Chargé de recherche, CNRS, Co-encadrant

UNIVERSITÉ DE GRENOBLE

REMERCIEMENTS

Je tiens à remercier :

Madame Cherine Ghedira Guegan, Professeur à l'IAE de Lyon, et Madame Parisa Ghodous, Professeur à l'Université Claude Bernard Lyon 1, de leur intérêt pour mon travail et pour avoir accepté de l'évaluer.

Monsieur Omar Boucelma, Professeur à l'Université d'Aix-Marseille, pour avoir accepté d'examiner mon travail et de faire partie du jury de cette thèse.

Madame Christine Collet, Professeur à Grenoble INP et responsable de l'équipe HADAS au LIG, pour avoir assuré la direction de cette thèse, et pour la confiance qu'elle m'a témoignée. Ce manuscrit doit beaucoup à ses suggestions et ses relectures attentives.

Madame Genoveva Vargas-Solar, Chargé de recherche au CNRS, pour avoir assuré le co-encadrement de cette thèse, et pour ses conseils avisés qui m'ont permis d'avancer vers des résultats meilleurs.

Monsieur José Luis Zechinelli Martini, professeur à l'Universidad de las Americas à Puebla, Mexique, pour m'avoir accueilli dans son équipe lors de mes séjours scientifiques au Mexique.

Les membres de l'équipe HADAS pour leur soutien tout au long de la réalisation de cette thèse. Un merci particulier à Noha pour son encouragement continu et à Mustafa pour sa participation active dans mes démarches administratives pour soutenir cette thèse. Je remercie, également, mes collègues du bureau pour les riches discussions que nous avons partagées.

Mes remerciements vont naturellement à Mireille et aux familles Khalaf, Bibent, Messmer et Suc pour leur amitié, leur gentillesse et pour les moments agréables passés ensemble.

Je tiens aussi à remercier Fatin, Sherine, Omar, Timo et Jordi pour leurs constants encouragements et pour avoir su me réconforter dans des moments difficiles.

Je ne saurai terminer sans adresser tous mes remerciements à ma famille dont la confiance et le soutien n'ont jamais fait défaut.

TABLE DES MATIERES

CHAPITRE 1 INTRODUCTION 11
1.1 Contexte 11
1.2 Problématique 12
1.3 Objectif et approche 13
1.4 Organisation du document 14

CHAPITRE 2 MASHUPS ET DISPONILBILITE DE DONNEES SUR LE WEB 15
2.1 Modèles de Mashups 15
2.1.1 Modèles de données 16
2.1.2 Opérateur de données 17
2.1.3 Mashlet 18
2.1.4 Wiring 19
2.1.5 Mashup 19
2.1.6 Modèle relationnel de mashups 20
2.1.7 Modèle de Hoyer 21
2.1.8 Modèle de mashups MACE 22
2.2 Systèmes d'exécution de mashups 22
2.2.1 Architecture générale 22
2.2.2 Déploiement 24
2.3 Disponibilité de données fraiches sur le web 26
2.3.1 Mesure de la disponibilité de données dans les mashups 26
2.3.2 Rafraichissement de données des mashups 27
2.3.3 Disponibilité de données par la réplication 28
2.3.4 Disponibilité de données par le cache 29
2.4 Conclusion 32

CHAPITRE 3 MODELE DE MASHUPS 33
3.1 Données de mashups 34
3.1.1 Types de données 34
3.1.2 Instances des types 35
3.1.3 Types et valeurs d'instances 38
3.1.4 Opérations sur les instances 38
3.2 Activité 39
3.2.1 Activités basiques 39
3.2.2 Activités composites 44
3.2.3 Définition du domaine des activités O 47
3.3 Mashlet 47
3.4 Wiring 49
3.5 Mashup 51
3.6 Conclusion 53

CHAPITRE 4 EXECUTION DE MASHUPS AVEC DISPONIBILITE DE DONNEES 55
4.1 Exécution d'un mashup 56
4.1.1 Exécution d'une activité 57
4.1.2 Exécution d'un mashlet 62
4.1.3 Exécution d'un wiring 62
4.2 Disponibilité des données dans les mashups 63
4.2.1 Organisation de données du Store 65
4.2.2 Gestion du Store 66
4.2.3 Remplacement de données 68
4.2.4 Rafraichissement de données 70
4.2.5 Exécution d'un nœud Retrieve++ 74
4.3 Conclusion 75

CHAPITRE 5 IMPLANTATION DE MELQART 77
5.1 Architecture du système 78
5.2 Valeurs complexes 80
5.3 Activité 81
5.3.1 Activités basiques 82
5.3.2 Activités composites 86
5.4 Mashlet 93
5.5 Wiring 94
5.6 Mashup 94
5.7 Item 96
5.8 Store 97
5.9 ReplacementManager 97
5.10 FreshnessManager 98
5.11 Interaction des mashups avec le Store 100
5.12 Validation 101
5.12.1 ItineraryPlanner 102
5.12.2 MyDashboard 104
5.12.3 Execution des instances des mashups 105
5.13 Conclusion 105

CHAPITRE 6 CONCLUSION 107
6.1 Contribution et bilan 107
6.2 Perspectives 108

BIBLIOGRAPHIE 111

ANNEXE A : SYSTEMES D'EXECUTION DE MASHUPS ETUDIES 119

ANNEXE B : SYNTAXE DE JSON 123

LISTE DES TABLEAUX

Tableau 1 : Modèle de données des mashups dans les outils/modèles étudiés.....17
Tableau 2 : Opérateurs proposés dans les outils étudiés.....18
Tableau 3 : Gestion de wirings dans les outils étudiés.....19
Tableau 4 : Protocoles d'accès aux données des fournisseurs dans les outils de mashups étudiés 23
Tableau 5 : Coût des opérations dans différentes structures de données.....74
Tableau 6 : Durées des vies des données du mashup ItineraryPlanner 103
Tableau 7 : Durées des vies des données du mashup MyDashboard 105

LISTE DES FIGURES

Figure 1 : Le mashup ItineraryPlanner 12
Figure 2 : Logique applicative de mashup ItineraryPlanner 13
Figure 3 : Échange de données entre les mashlets Map et Weather défini par un wiring..... 19
Figure 4 : Modèle de mashups de Hoyer..... 21
Figure 5 : Les composants d'une architecture générale d'un système d'exécution de mashups..... 23
Figure 6: Diagra .....
mme de déploiement des composants d'un système d'exécution de mashups (côté client)..... 24
Figure 7 : Diagramme de communication illustrant le fonctionnement d'un mashup exécuté côté Client..... 25
Figure 8 : Diagramme de déploiement des composants d'un système d'exécution de mashups (côté serveur) . 25
Figure 9 : Diagramme de communication illustrant le fonctionnement d'un mashup exécuté côté serveur..... 26
Figure 10 : Architecture du cache distribué proposé dans [91] 30
Figure 11 : Pile de concepts de mashups 34
Figure 12 : Représentation graphique d'une activité 39
Figure 13 : Représentation graphique d'une activité *Sequence* 45
Figure 14 : Représentation graphique d'un exemple d'une activité *Sequence* 45
Figure 15 : Représentation graphique d'une activité *Foreach* 46
Figure 16 : Représentation graphique d'un exemple d'une activité *Foreach* 46
Figure 17 : Représentation graphique d'une activité *Parallel* 47
Figure 18 : Représentation graphique d'un mashlet 48
Figure 19 : Représentation graphique du mashlet Map..... 48
Figure 20 : Représentation graphique du mashlet Weather 49
Figure 21 : Représentation graphique d'un wiring 50
Figure 22 : Représentation graphique du wiring entre les mashlets Map et Weather 51
Figure 23 : Représentation graphique du mashup ItineraryPlanner 52
Figure 24 : Diagramme d'activité UML de l'exécution d'un mashup 56
Figure 25 : Graphe du mashup ItineraryPlanner 57
Figure 26 : Exécution du mashup ItineraryPlanner 58
Figure 27 : Arbre de l'activité du wiring Map2Weather 58
Figure 28 : Exécution du mashup ItineraryPlanner 59
Figure 29 : Composition d'un nœud d'une activité basique..... 59

Figure 30 : Diagramme d'activité UML du processus d'exécution d'une activité basique 60
Figure 31 : Diagramme de communication entre le nœud *Retrieve* et le fournisseur de données 60
Figure 32 : Arbre de nœuds d'une activité Sequence 61
Figure 33 : diagramme d'activité UML du processus d'exécution d'une activité *Sequence* 61
Figure 34 : Diagramme d'activité UML de l'exécution d'un mashlet 62
Figure 35 : Diagramme d'activité UML de l'exécution d'un wiring 63
Figure 36 : Diagramme de communication UML entre le nœud *Retrieve++*, le fournisseur de données et le Store 63
Figure 37 : Exécution d'un mashup avec disponibilité de données 64
Figure 38 : Représentation du Store 66
Figure 39 : Utilisation des fonctionnalités de gestion du Store dans l'exécution d'un nœud *Retrieve++* 67
Figure 40 : Utilisation de la fonction makeroom dans l'exécution d'un nœud *Retrieve++* 68
Figure 41 : Rafraichissement des items 70
Figure 42 : Exécution d'un nœud *Retrieve++* 75
Figure 43 : Approche de MELQART pour la création et l'exécution des mashups 78
Figure 44 : Interaction des utilisateurs avec MELQART 78
Figure 45 : Architecture de MELQART 79
Figure 46 : Classe Activity 82
Figure 47 : Les classes des activités basiques 83
Figure 48 : Les classes des activités composites 87
Figure 49 : Exécution de la méthode run() d'une instance de la classe Sequence 88
Figure 50 : Exécution de la méthode run() d'une instance de WOEIDCitySeq 89
Figure 51 : Exécution de la méthode run() d'une instance de la classe Foreach 90
Figure 52 : Exécution de la méthode run() d'une instance de CitiesListForeach 91
Figure 53 : Interactions entre une activité Parallel et ses sous-activités 92
Figure 54 : Classe Mashlet 93
Figure 55 : Interaction entre une instance de Mashlet et son activité 93
Figure 56 : Classe Wiring 94
Figure 57 : Classe Mashup 95
Figure 58 : Classe Item 96
Figure 59 : Instance de la classe Item 96
Figure 60 : Classe Store 97
Figure 61 : La classe ReplacementManager 98
Figure 62 : La classe FreshnessManager 99
Figure 63 : Exécution de la méthode run() de l'activité Retrieve avec utilisation du Store 101
Figure 64 : Résultats du mashup ItinerayPlanner pour l'itinéraire de Grenoble à Paris 103
Figure 65 : Résultats du mashup MyDashboard avec Grenoble comme paramètre en entrée. 104
Figure 66 : Chronologie des exécutions 106

Figure 67 : Sources de données dans Montage.....119
Figure 68 : Construction de mashlets dans Yahoo! Pipes.....120
Figure 69 : Architecture de MSS.....122
Figure 70 : Syntaxe de JSON.....123

Chapitre 1

INTRODUCTION

1.1 Contexte

A l'origine, le terme mashup désigne un morceau musical hybride composé d'échantillons de musique issus de chansons différentes. Par analogie, en Informatique, un mashup est une application web qui combine des ressources (données, fonctionnalités etc.) provenant de sources hétérogènes. Ces ressources sont agrégées pour former un résultat affiché dans des composants appelés mashlets, comme iGoogle[1]. Une des premières définitions des mashups apparaît en 2006 dans [1] : "*Mashups are an exciting genre of interactive Web applications that draw upon content retrieved from external data sources to create entirely new and innovative services*". Les mashups sont utilisés dans différents domaines comme la biologie [2], la santé [3], l'électroménager [4], la formation en ligne (*E-learning*) [5][6] et la gestion des documents en entreprise [7].

Exemple de mashup

Supposons qu'Alice veuille planifier un voyage de Grenoble à Paris et qu'elle veuille déterminer l'itinéraire qu'elle va prendre, ainsi que les prévisions météorologiques dans les villes de passage. Comme le voyage est long, Alice aura faim à l'arrivée. Elle voudrait choisir un restaurant parmi une liste de restaurants proposés ayant une moyenne de notes supérieure à 3 sur 5. Au lieu de contacter des services sur le Web de manière séparée, Alice veut exprimer ses besoins d'une manière globale et avoir les réponses de différents services sur une même page Web. Pour cela, Alice a recours à un mashup qui combine des données issues des services disponibles sur le web comme Google maps et Yahoo météo et Google Places. Le mashup est composé de trois mashlets affichant l'itinéraire, les conditions météorologiques et la liste de restaurants. Dans ce document, nous nous référons à ce mashup sous le nom d'ItineraryPlanner (illustré dans la Figure 1). Nous avons choisi ce scénario parce qu'il intègre des données de nature hétérogènes : dynamiques et multimédia où l'on prend en considération des données spatiales comme dans [8][9].

Un mashup fonctionne selon une logique applicative. Les deux points clés de cette logique applicative sont les services de données et la coordination des appels des services. Les services font office de fournisseurs des données par des appels des méthodes. La coordination sert à coordonner la récupération et la transformation de données afin de pouvoir les échanger entre les composants du mashup. Les fonctions de gestion de données qui en ressortent sont :

[1] http://www.google.fr/ig, Google a mis fin à ce service en novembre 2013.

Figure 1 : Le mashup ItineraryPlanner

- La récupération de données :
  - À partir des services de données en appelant des méthodes exportées par ces services, par exemple obtenir l'itinéraire entre Grenoble et Paris sollicité depuis le service Google maps.
  - En interagissant avec les utilisateurs à travers des interfaces fournies par le mashup : par exemple, entrer la ville de départ et la destination.
- La transformation de données récupérées pour les afficher de manière agrégée ou adaptée au dispositif utilisé pour les visualiser. Il s'agit de mettre en place cette transformation au sein et entre les mashlets.
- L'échange de données : les données transformées peuvent servir comme entrée pour appeler d'autres services. Par exemple, les villes de passage dans l'itinéraire obtenu de Google maps, sont envoyées à Yahoo! Weather pour obtenir les informations sur les conditions météorologiques dans ces villes.

La Figure 2 présente la logique applicative du mashup ItineraryPlanner définie avec l'outil Yahoo! Pipes [10]. Le mashlet Map s'adresse à Google maps pour récupérer l'itinéraire, ceci est suivi par une transformation pour récupérer les villes intermédiaires et les mettre dans le bon format pour les envoyer d'une manière itérative à Yahoo! Weather pour récupérer la météo. Les coordonnées de la ville d'arrivée sont récupérées par le mashlet Restaurants pour récupérer une liste de restaurants auprès du service Google Places.

Cette gestion de données s'appuie sur une hypothèse forte et que nous adoptons : les interfaces des services ont des paramètres d'entrée/sortie typés selon un modèle de données de mashups. Ceci revient à considérer qu'il y a un mapping implicite entre les modèles des données sous-jacents aux services et le modèle de données propre aux mashups.

1.2 Problématique

Les travaux existants, sur les mashups, se sont intéressés principalement à leur fonctionnement et aux différentes manières de les construire [11][12][13][14], aux domaines de leur utilisation et à l'interaction avec les utilisateurs [15][16][17]. Dans cette thèse nous nous intéressons à la gestion de données dans les mashups et plus particulièrement à la disponibilité des données fraîches dans les mashups. Améliorer la disponibilité des données au sein d'un mashup se définit à travers les aspects suivants :

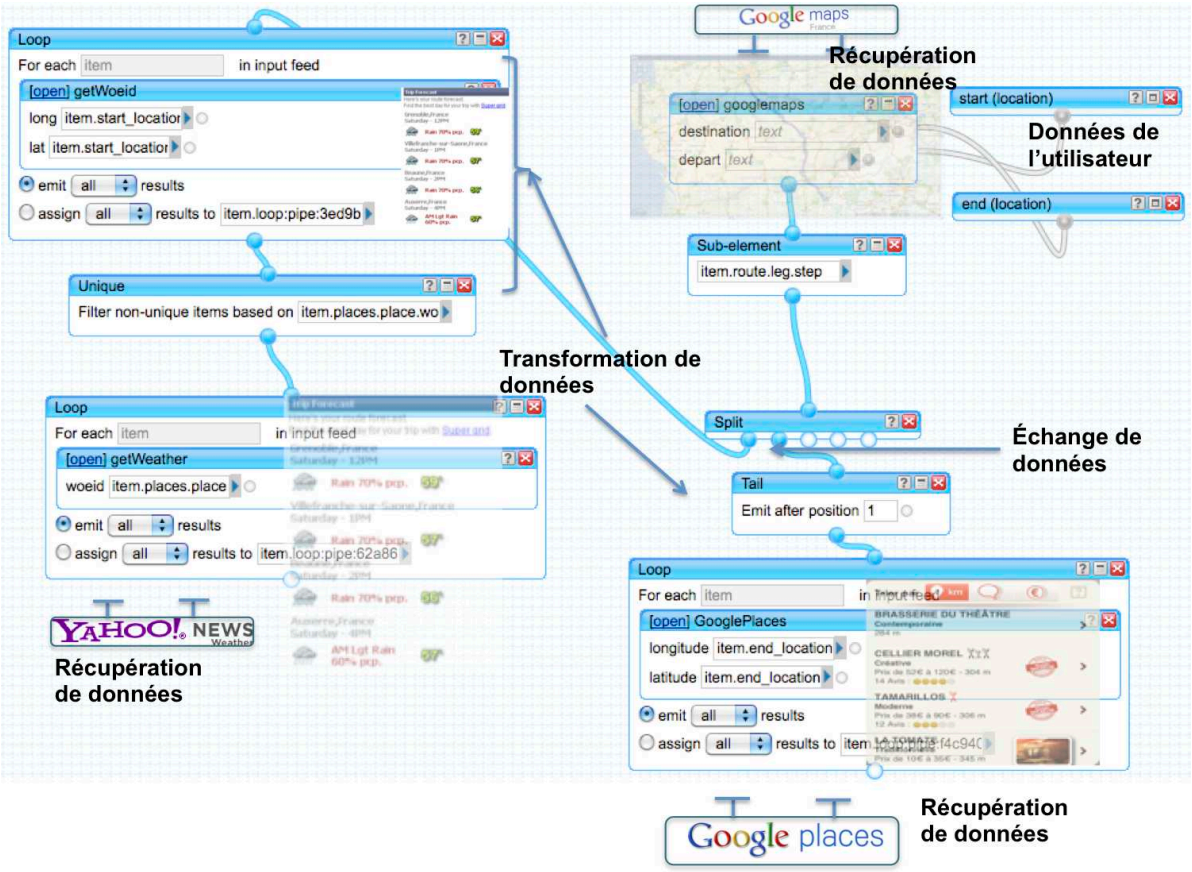

Figure 2 : Logique applicative de mashup ItineraryPlanner

- La continuité d'accès aux données même si le service est inaccessible.
- La gestion de la fraîcheur de données du mashup. Ces données sont de nature dynamique et une mauvaise stratégie de mise à jour peut engendrer des mashups avec des données périmées.
- Éviter de récupérer la même donnée plusieurs fois.

L'approfondissement de ces problèmes nous a amené à nous poser différentes questions :

- Quelles sont les données dont on a besoin pour le bon fonctionnement du mashup ? et comment les identifier dans les résultats des appels services ?
- Comment retrouver ces données ?
- Comment garder la fraicheur de ces données ?

1.3 Objectif et approche

L'objectif de cette thèse est de proposer une approche pour améliorer la disponibilité des données des mashups qui garantit :

- L'accès aux données même si le service est indisponible pour maintenir un mashup en état de fonctionnement.
- La fraicheur de données et pour cela il faut prendre en compte la durée entre deux appels qui doit faire l'objet d'un compromis entre les caractéristiques du service et les besoins de l'utilisateur.
- Un partage de données entre les mashups et les mashlets. Ce partage permet d'augmenter la probabilité de disponibilité de données.

Pour atteindre cet objectif, nous proposons MELQART[2] un système d'exécution de mashups avec un mécanisme de disponibilité de données. Notre contribution inclut :

- Une étude de l'état de l'art présentant une analyse des modèles de mashups existants et de leurs concepts ainsi que les travaux existants sur les systèmes d'exécution des mashups et sur la disponibilité et la fraicheur de données du web. L'étude de ces travaux constitue une base à partir de laquelle nous proposons notre approche pour améliorer la disponibilité des données des mashups.
- La définition d'un modèle de mashups qui s'appuie sur les modèles de valeurs complexes [19]. Ce modèle permet de spécifier les caractéristiques de la disponibilité de données. Les concepts de ce modèle sont illustrés avec notre scénario ItineraryPlanner.
- La définition du principe d'exécution de mashups selon le modèle de mashup proposé. Nous améliorons la disponibilité et la fraicheur des données du mashup par des fonctionnalités orthogonales à son processus d'exécution. Ces fonctionnalités prennent en charge la gestion et le rafraichissement de données

1.4 Organisation du document

Le reste du document est organisé comme suit :

- Le Chapitre 2 présente les concepts et les modèles des mashups (aspect statique des mashups) et étudie l'architecture générale des systèmes d'exécution de mashups existants (aspect dynamique des mashups). Ensuite, il s'intéresse à la disponibilité des données fraiches sur le web et aux solutions logicielles possibles pour la garantir.
- Le Chapitre 3 présente les concepts de notre modèle de mashup. Parmi eux nous retrouvons les données, les mashlets etc.
- Le Chapitre 4 commence par introduire les principaux éléments de l'exécution d'un mashup. Ensuite, il décrit notre proposition pour assurer la disponibilité de données lors de l'exécution des mashups et son intégration au processus d'exécution de mashups. Cette solution est à base de fonctions que nous associons d'une façon orthogonale au processus d'exécution de mashups.
- Le Chapitre 5 décrit l'implantation et la validation d'un prototype de MELQART. Il commence par présenter l'architecture du système et l'implantation des concepts de notre modèle de mashups et des fonctionnalité proposées pour améliorer la disponibilité de données fraiches des mashups. L'implantation est décrite avec les diagrammes de classes et de séquence de la notation UML. Ensuite le chapitre présente la validation de notre implantation avec l'exécution de plusieurs instances de deux scenarios de mashups.
- Le Chapitre 6 récapitule les principaux éléments de notre contribution et présente des orientations définissant la perspective de notre travail afin de l'améliorer et le consolider.

[2] Dans la mythologie phénicienne, Melqart [18] signifie le roi (melk) de la cité (qart). Il est le dieu et protecteur de Tyr et de ses activités commerciales. Il accompagna les navigateurs phéniciens dans leur colonisation de la Méditerranée. Il est assimilé à Héraclès dans la mythologie gréco-romaine.

Chapitre 2

MASHUPS ET DISPONILBILITE DE DONNEES SUR LE WEB

Ce chapitre présente les travaux existants dans les domaines des mashups (modèles et systèmes d'exécution de mashups) et de la disponibilité et la fraicheur de données du web. L'étude de ces travaux constitue une base à partir de laquelle nous proposons notre approche pour améliorer la disponibilité des données des mashups.

Le chapitre est organisé de la manière suivante : la section 2.1 présente les concepts et les modèles des mashups (aspect statique des mashups). La section 2.2 étudie l'architecture générale des systèmes d'exécution de mashups existants (aspect dynamique des mashups). Ensuite, la section 2.3 s'intéresse à la disponibilité des données fraiches sur le web et aux solutions logicielles possibles pour la garantir. Enfin, la section 2.4 conclut le chapitre.

2.1 Modèles de Mashups

Cette section décrit les modèles existants de mashups. Nous avons constaté qu'il y a peu de travaux qui ont proposé des modèles de mashups (ceci est dû à l'apparition récente de cette notion de mashups) et que la plupart se sont focalisés sur les outils de création et d'exécution de mashups et leur fonctionnement.

L'étude des modèles et des outils proposés (cf. Annexe A) nous a permis d'identifier les concepts nécessaires pour décrire un mashup. Donnée, Operateur, Mashlet, Wiring et Mashup. Un **mashlet** affiche des **données** définies selon un certain modèle de données. Elles sont récupérées auprès des fournisseurs de données. Des **opérateurs** décrivent la manipulation de ces données. Un **mashup** est un ensemble de mashlets qui échangent des données via des **wirings**. Dans le scenario ItineraryPlanner, introduit dans le Chapitre 1, ces concepts se présentent comme suit :

- Les **données** sont récupérées depuis des fournisseurs de données : Google maps, Yahoo! Weather et Google Places.
- Les **opérateurs** décrivent le traitement de ces données.
- Les **mashlets** (Map, Weather, Restaurants) constituent les composants qui affichent les résultats.
- Deux **wirings** décrivent le transfert de la liste de villes de passage de Map vers Weather, et celui des coordonnées de la ville d'arrivée de Map vers Restaurants).
- Le **mashup** ItineraryPlanner est constitué des mashlets Map, Weather, Restaurants et des wirings sous-jacents.

Ces concepts sont présentés dans les sous-sections 2.1.1 à 2.1.5. Ensuite les sous-sections 2.1.6, 2.1.7 et 2.1.8 présentent des modèles de mashups étudiés dans l'état de l'art.

2.1.1 Modèles de données

Les données des mashups peuvent être définies selon différents modèles de données[3]. Dans cette sous-section, nous présentons les modèles de données utilisées dans les modèles et les outils de mashups étudiés.

- Le **modèle relationnel** [21] fut introduit pour pallier aux lacunes du modèle hiérarchique et du modèle réseau, comme la redondance et la rigidité. Il permet d'assurer une indépendance de données et d'améliorer les langages de requête. Il est basé sur les concepts de la théorie des relations.
  Le modèle relationnel de données est adopté dans le modèle relationnel de mashups [3] (cf. sous-section 2.1.6). Les auteurs y considèrent, également, que toutes les données (provenant de fournisseurs externes ou manipulées au sein des mashups) sont décrites selon le modèle relationnel. Ce choix de modélisation de données est justifié par les bases mathématiques du modèle relationnel.
- Les **modèles orientés objet** furent proposés pour rapprocher les bases de données aux langages de programmation. Certains travaux ont focalisé sur l'intégration des caractéristiques de persistance et de transactions, présentes dans les SGBDs, aux langages de programmation orientées-objet. Tandis que d'autres ont focalisé sur l'intégration des caractéristiques de l'orienté-objet aux SGBDs comme l'encapsulation, la hiérarchie des classes ou l'attachement dynamique (dynamic binding). Ces travaux ont contribué au développement de SGBDs objets comme O2 [22] et SAMOS [23]. Un groupe ODMG (Object Data Management Group) fut crée. Il est responsable de la définition de standards pour les bases de données objets [24] dont le langage OQL (Object Query Language).
  Le modèle orienté-objet fut adopté par l'outil Popfly [25] de Microsoft. Ce choix est justifié par une proximité recherchée des langages de programmation. Nous n'avons pas pu étudier et tester cet outil, parce que Microsoft y a arrêté son développement avant le début des travaux sur cette thèse.
- Les **modèles de données semi-structurées** sont basés sur une organisation de données dans des structures d'arbres étiquetées ou des graphes et des langages de requêtes pour accéder et modifier les données [26][27]. Les modèles de données semi-structurées sont utilisés pour décrire des données sans schéma explicite. Dans les modèles de données structurés, on distingue entre le type de données (schéma) et les données mêmes. Cependant, cette distinction est estompée dans les modèles de données semi-structurées : on parle de données auto-décrites [27].
  Les données semi-structurées le plus répandues sont des données XML. La richesse d'XML provient des langages associés comme XPath, XQuery, XSLT qui facilitent la manipulation des données XML.

Le Tableau 1 indique les modèles de données manipulées dans les modèles et les outils de création et d'exécution de mashups étudiés. Nous constatons que la plupart des outils de mashups utilisent des modèles à base d'XML pour représenter les données manipulées. Ceci est justifié par le fait que la plupart des données du web sont décrites avec XML.

[3] Un modèle de données fournit les moyens pour (1) spécifier des structures de données, (2) définir un ensemble de contraintes associées à ces structures et (3) et les manipuler [20].

Modèle de données	Outil/Modèles de mashups
Relationnel	Modèle relationnel de mashups [3]
Orientés-objet	Popfly [25]
Semi-structurées	Montage [28] (XML), Yahoo! Pipes [10] (flux RSS), Intel MashMaker [29] (RDF), EMML/Presto [30] (XML), MSS [31] (XML), WSO2 Application Server [32] (XML)

Tableau 1 : Modèle de données des mashups dans les outils/modèles étudiés

2.1.2 Opérateur de données

Un opérateur de données est modélisé avec une fonction qui manipule des données en entrée pour en produire d'autres en sortie. Les principaux opérateurs de données, pour les différents modèles, sont la jointure, le filtrage, le tri, la troncature, le comptage, l'élimination de doublons etc. Certains fournisseurs de données (Wikipédia, Craiglist[4] etc.) n'exposent pas leurs contenus via des méthodes exportées. Ainsi, certains outils offrent des opérateurs décrivant l'extraction des données avec la technique du *web scraping*[5] [1][33]. Les données extraites sont décrites selon le modèle de données adopté dans le mashup. Par exemple, le mashup HousingMaps[6] a recours à cette technique pour extraire les annonces depuis le site Craiglist.

Le Tableau 2 indique la nature des opérateurs définis dans les outils de création et d'exécution de mashups étudiés. Nous constatons que les outils, qui s'adressent à des utilisateurs sans compétence en programmation (comme Montage, Intel MashMaker), proposent peu d'opérateurs, offrant ainsi une moindre complexité pour construire un mashup. D'autre part, certains outils nécessitent des compétences en programmation comme WSO2 Mashup Server. Les outils Yahoo ! Pipes et Presto offrent des opérateurs similaires pour manipuler les données. Le modèle relationnel de mashups ne propose pas d'opérateurs pour manipuler les données ; Ils font partie des perspectives évoquées par les auteurs. L'outil MSS propose un langage de manipulation de données : *Mashup Service Query Langage* (MSQL). Celui-ci est basé sur le langage SQL et il est enrichi avec des opérateurs exécutant des tâches spécifiques interprétées selon une ontologie propre à MSS (par exemple, « trouver des maisons », « trouver des vols »).

Modèle relationnel de mashups	• Récupération de données par appel de services. • Ce modèle ne propose pas d'opérateurs pour manipuler les données. A l'heure actuelle, les données récupérées sont rendues en entier. Néanmoins, les auteurs proposent, dans leur perspective, d'enrichir le modèle avec des opérateurs de manipulation de données. (cf. sous-section 2.1.6)
Montage	Récupération de flux RSS.

[4] Craiglist est un site web américain de petites annonces

[5] Un web scraper est un programme qui extrait des données depuis un site web dans le but de les exploiter.

[6] http://www.housingmaps.com/

Yahoo! Pipes	L'ensemble des opérateurs de Yahoo! Pipes comprend : • Les opérateurs de récupération de données : depuis des services REST ou bien des flux RSS. • Les opérateurs de transformation de données (tri, filtrage, élimination de doublons, troncature, inverser une liste, l'extraction de données) • les opérateurs arithmétiques • les opérateurs sur les expressions régulières.
Intel MashMaker	Les données sont manipulées avec des formules comme dans les feuilles de calcul.
WSO2 Mashup Server	Les opérations sont définies avec un code JavaScript.
EMML/Presto	L'ensemble des opérateurs de EMML / Presto : • Les opérateurs de récupération de données : depuis des services REST, SOAP ou bien des flux RSS ou bien par le web scraping. • Les opérateurs de transformation de données (jointure, tri, filtrage, élimination de doublons, troncature, inverser une liste, extraction de données). • Des opérateurs définis par encapsulation de code JavaScript, XQuery.
MSS	L'ensemble des opérateurs de MSS comprend : • Des opérateurs qui réalisent des tâches que nous retrouvons dans le langage SQL comme la sélection, la jointure. • Des opérateurs exécutant des tâches spécifiques interprétées selon l'ontologie de MSS (par exemple, « trouver des maisons », « trouver des vols »).

Tableau 2 : Opérateurs proposés dans les outils étudiés

2.1.3 Mashlet

Un mashlet est un composant graphique qui affiche des données sous la forme d'un widget. Il récupère des données auprès de fournisseurs de données, il les manipule avec des opérateurs de données et il les retourne en sortie. Les données produites sont alors visualisées graphiquement. Par exemple, le mashup ItineraryPlanner (défini dans le Chapitre 1) contient trois mashlets : le premier « Map » permet l'affichage de l'itinéraire entre deux villes sur une carte, le deuxième « Weather » se charge d'afficher les informations météorologiques dans les principales villes sur cet itinéraire et le troisième « Restaurants » affiche une liste de restaurants dans la ville d'arrivée.

Cette définition n'est pas unique. Elle est générale et couvre les différents points de vue des modèles et les outils étudiés. En effet certains utilisent des terminologies différentes pour désigner ce concept et d'autres ne couvrent pas tous les aspects mentionnées :

- Dans [3][34], un mashlet est un composant qui a des données en entrée et en sortie. Il interroge des sources des données, utilise des services web externes et définit la coordination entre ces composants. Il peut intégrer une représentation graphique de ses données.

- Dans [35][36][37], ce concept est décrit avec le terme largement connu : widget. Un widget est une interface graphique affichant des informations récupérées auprès d'une source de données externe.
- Dans [32][38][10][39], les auteurs utilisent même le terme mashup pour désigner ce concept. Un mashup est une application qui combine des données provenant de différentes sources. La combinaison est réalisée par le moyen d'un flot de données.
- Enfin, dans [40], les auteurs séparent l'aspect combinaison de données et l'aspect visualisation. Ils désignent la combinaison de données sous le nom de mashup alors que le terme mashlet désigne la visualisation de ces données.

2.1.4 Wiring

Un wiring définit un échange de données entre deux mashlets [35][14][41]. Par exemple, dans le mashup ItineraryPlanner, un wiring (cf. Figure 3) est nécessaire pour envoyer les données des villes de passage depuis le mashlet « Map » au mashlet « Weather ».

Dans [34][42], les auteurs intègrent ce concept dans leur outil sans y donner un nom. On parle de navigation entre mashlets. Dans [3], les auteurs intègrent le principe d'échange de données entre mashlet, mais sans lui donner de nom.

Figure 3 : Échange de données entre les mashlets Map et Weather défini par un wiring

Le Tableau 3 indique les modèles/outils qui permettent de définir des wirings entre les mashlets. Nous constatons que le principe d'échange de données entre les mashlets (définition de wirings) n'est pas omniprésent dans les modèles/outils de mashups étudiées. Par exemple dans iGoogle, les mashlets ne communiquent pas entre eux ; ainsi avec ce genre d'outil, nous ne pouvons pas modéliser le mashup ItineraryPlanner. Ce constat souligne l'hétérogénéité des visions de ce qu'est un mashup. Ceci est dû à l'apparition récente de cette notion de mashups

Modèles/Outils intégrant le concept Wiring	Modèle relationnel de mashups, Intel MashMaker, WSO2 Mashup Server, Presto
Modèles/Outils ne prenant pas en charge le concept Wiring	Montage, Yahoo ! Pipes

Tableau 3 : Gestion de wirings dans les outils étudiés

2.1.5 Mashup

Dans les travaux étudiés, nous avons constaté qu'un mashup est défini sous-différentes terminologies. Dans [31][36], il est défini en tant que **technologie** émergente qui consiste en la

création d'applications ad hoc par des utilisateurs finaux. Dans [1][43][37], un mashup est défini en tant qu'un nouveau **type d'applications** web qui permettent de combiner de données issues de différents sources. Dans [44], un mashup est défini en tant qu'une **approche** de développement d'applications qui permet à des utilisateurs finaux de combiner des données provenant de différentes sources. Dans ces définitions, le terme récurrent est application et l'idée sous-jacente est la combinaison de données provenant de différentes sources.

Dans les modèles/outils qui intègrent le principe de transfert de données entre mashlets (définition de wirings, voit sous-section 2.1.4), un mashup est vu comme un ensemble de mashlets et de wirings qui coordonnent ces mashlets. Alors que dans les modèles/outils qui n'intègrent pas la définition de wiring, un mashup est vu comme un ensemble de mashlets seulement.

Par exemple, le mashup ItineraryPlanner prend en entrée des paramètres indiquant les villes de départ et d'arrivé et un tarif moyen pour les restaurants. Il contient les trois mashlets « Map », « Weather » et « Restaurants ». Il contient aussi deux wirings : le premier pour transférer les villes de passages depuis le mashlet « Map » vers le mashlet « Weather » et le deuxième pour transférer les coordonnées géographiques du lieu d'arrivé depuis le mashlet « Map » vers le mashlet « Restaurants ».

Comme indiqué dans la sous-section 2.1.3, le concept de mashlet est parfois désigné avec le terme mashup dans certains modèles/outils. Ceux ci emploient le terme « dashboard » pour faire référence au mashup (en tant qu'ensemble de mashlets).

2.1.6 Modèle relationnel de mashups

Dans [3], les auteurs introduisent un modèle relationnel pour définir les mashups de données. Le modèle est basé sur les relations. Ce choix de modélisation est justifié par les bases mathématiques du modèle relationnel.

Un mashlet est défini comme un ensemble de relations. Celles ci sont classifiées comme suit :

- Relations internes : elles sont utilisées pour contenir des données propres au mashlet et non exposées à d'autres mashlets.
- Relations entrée/sortie : elles sont utilisées pour définir l'interface du mashlet via laquelle il interagit avec l'utilisateur et d'autres mashlets.
- Relations de service : elles représentent des web services que le mashlet peut importer.

L'interaction entre ces relations est définie par des règles actives (du style Datalog [45][46]). Par exemple, soit un mashlet $Docs$ dont les relations sont liées par la règle active suivante :

$$R_{out}(nom, adresse, num) :- R_{in}(m), R_{local}(m, nom), WS_{bff}(nom, adresse, num)$$

R_{in} est une relation qui représente un formulaire où l'utilisateur entre le nom d'une maladie m. La relation locale $R_{local}(m, nom)$ fournit les noms des médecins spécialistes dans le traitement de la maladie m. Soit le service web $WS_{bff}(nom, adresse, num)$ qui fournit l'adresse, et le numéro de téléphone pour un médecin identifié par son nom. La relation $R_{out}(nom, adresse, num)$ affiche le résultat contenant les noms des médecins, leurs adresses ainsi que leurs numéros de téléphones.

Dans le même esprit, les interactions entre mashlets (sans mention du terme wiring) peuvent être décrites avec des règles actives. Soit un mashlet $MapAdr$ qui affiche les adresses des médecins sur une carte. Il importe la relation de sortie du mashlet $Docs$ via la règle active suivante[7] :

$$MapAdr.Mrk_{in}(x, y, nom) :- Docs.R_{out}(nom, adresse, num), AddrToXY_{bff}(adresse, x, y)$$

L'adresse, donnée par la relation $Docs.R_{out}(nom, adresse, num)$, est convertie en coordonnées géographiques x et y via le service web $AddrToXY_{bff}$.

Un mashup est vu comme un réseau de mashlets qui interagissent entre eux. Les auteurs évoquent le besoin éventuel de manipuler les données (filtrage tri etc.). Pour répondre à ce besoin, ils proposent, dans une perspective ultérieure, la possibilité d'enrichir les règles actives avec des fonctions de manipulation de données ou de déléguer ces fonctionnalités à des services web.

2.1.7 Modèle de Hoyer

Dans [37], les auteurs proposent un modèle des concepts de mashups. Il est fait avec un digramme de classes de la notation UML (Figure 4[8]). Ils y introduisent la notion d'usage qui correspond à l'autorisation de l'utilisation d'une ressource ou d'un mashlet (widget) par une certaine personne (client) contre certaines conditions et certains frais. Les « pipings » correspondent aux opérateurs d'agrégation de données. Un wiring y est défini comme étant un moyen de communication entre deux widgets avec des paramètres de mapping indiquant les connections entre les entrées et les sorties des widgets.

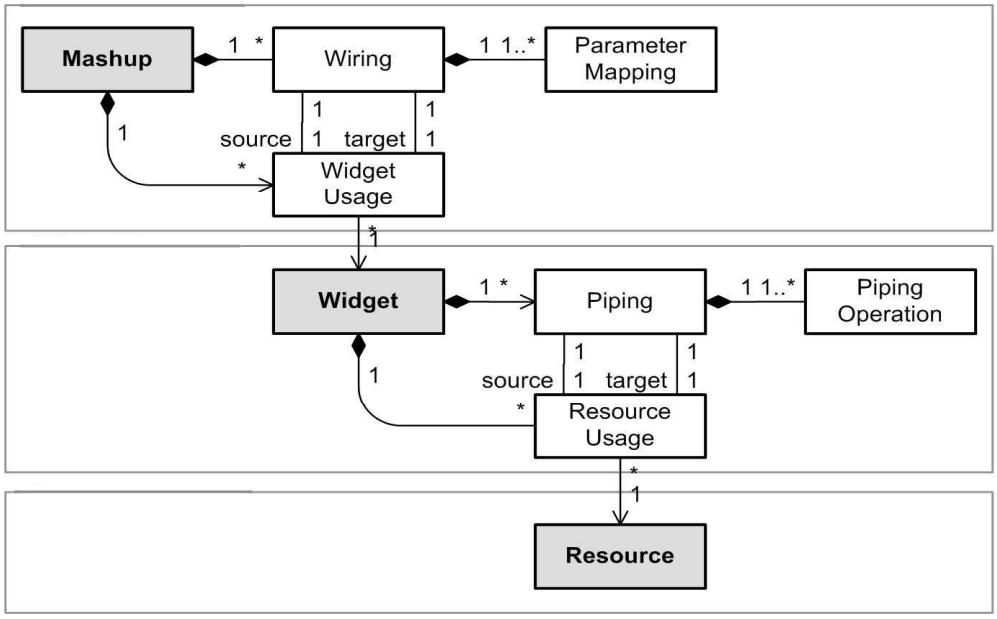

Figure 4 : Modèle de mashups de Hoyer

[7] Une relation R d'un mashlet M est notée $M.R$

[8] Image obtenue de [37].

2.1.8 Modèle de mashups MACE

MACE [39] est une plateforme de cache pour les mashups. Les auteurs y intègrent une proposition d'un modèle de mashups. Un mashup[9] (1) récupère des données depuis des sources différentes, (2) les manipules et (3) les retournes à l'utilisateur final.

$MpSet = \{Mp_0, Mp_1, \dots, Mp_{N-1}\}$ représente l'ensemble de mashups définis dans la plateforme de mashups à un moment donnée. La plateforme de mashup inclut un ensemble de classes d'opérateurs décrivant la manipulation des données comme le tri, le filtrage, l'élimination de doublons, la troncature, la jointure, le comptage, l'extraction de données. Les auteurs introduisent deux classes d'opérateurs décrivant la récupération de données depuis une source externe (*fetch*) et la livraison du résultat (*dispatch*). $OpSet = \{op_0, op_1, \dots, op_{M-1}\}$ dénote l'ensemble de classes d'opérateurs disponibles dans la plateforme de mashups. Chaque classe d'opérateur définit sa propre signature.

Un mashup comprend un ensemble d'instances d'opérateurs de $OpSet$. Cet ensemble contient un ou plusieurs instances de l'opérateur *fetch* et une seule instance de l'opérateur *dispatch*. Un mashup est modélisé avec une structure d'arbre. Chaque nœud correspond à une instance d'opérateur. Le nœud racine correspond à l'instance de l'opérateur *dispatch* et les nœuds feuilles correspondent aux instances de l'opérateur *fetch*. La valeur de la sortie de l'instance d'opérateur d'un nœud est l'entrée (ou une des entrées) de l'instance d'opérateur du nœud parent.

Les mashups, vus comme des arbres, peuvent partager les données provenant des mêmes sources. Ainsi l'ensemble des mashups forme des graphes orientés acycliques (*Directed Acyclic Graphs DAG*s).

2.2 Systèmes d'exécution de mashups

Cette section présente les composants d'une architecture générale des systèmes d'exécution de mashups (sous-section 2.2.1). Ensuite, elle présente les manières de déploiement de ces composants (sous-section 2.2.2). La présentation est accompagnée d'un positionnement des systèmes d'exécution des outils étudiés par rapport à l'architecture et au déploiement des composants.

2.2.1 Architecture générale

La Figure 5 présente l'architecture générale, d'un système d'exécution de mashups, déduite à partir de nos lectures des travaux existants. D'un point de vue architectural, les systèmes d'exécution de mashups mettent en jeu plusieurs composants disjoints qui interagissent entre eux pour réaliser l'exécution des mashups. Tous ces composants ne sont pas présents dans toutes les architectures :

- Les fournisseurs de données (**Providers**) sont des composants autonomes que l'on peut interroger (*Invoke*) à l'aide d'une requête ayant un ou plusieurs paramètres. L'interrogation de ces fournisseurs se fait selon un protocole d'accès. Ils ne font pas partie des systèmes de mashups. Cependant, nous les évoquons pour illustrer et comparer les capacités des outils étudiés à interagir avec des fournisseurs hétérogènes. Le Tableau 4 indique les protocoles

[9] Notons que cette définition correspond au concept mashlet présenté à la sous-section 2.1.3 où nous avons souligné que dans certains modèles/outils le concept de mashlet est parfois désigné avec le terme mashup.

d'accès aux données des fournisseurs pris en charge dans les outils de mashups étudiés. Pour un outil de mashup, plus le nombre de protocoles d'accès pris en charge est grand, plus le choix offert de fournisseurs de données est large lors de la création de mashups.

Montage	Yahoo ! Pipes	Intel MashMaker	WSO2 Mashup Server	Presto	MSS
REST[10]	REST, HTTP[11]	HTTP	REST, SOAP[12], HTTP	REST, SOAP, HTTP	SOAP

Tableau 4 : Protocoles d'accès aux données des fournisseurs dans les outils de mashups étudiés

- Le **Catalog** est un composant optionnel qui n'est pas présent dans tous les outils de mashups. Il enregistre les mashlets et les mashups crées et les mets à disposition des utilisateurs des mashups Tous les outils d'exécution de mashups étudiés intègrent le composant Store sauf MSS où l'utilisateur doit définir son besoin sans avoir recourir à un tel composant.
- Le composant **MashupGUI**, généralement un navigateur web, constitue l'interface via laquelle l'utilisateur choisit un mashup depuis le **Catalog** et demande son exécution au composant **MashupEngine**.
- Le composant **MashupEngine** est responsable de l'exécution des mashups. Il interagit avec les composants Providers afin de récupérer des données.
  - Dans les outils Montage (comme dans iGoogle), le composant retourne ces données directement à l'utilisateur (via le composant **MashupGUI**).
  - Dans les autres outils étudiés, ce composant manipule ces données et coordonne les exécutions des mashlets et des wirings des mashups. Il contient un buffer qui offre les outils pour le stockage temporaire de données en cours de traitement. Le résultat est retourné au composant **MashupGUI**.

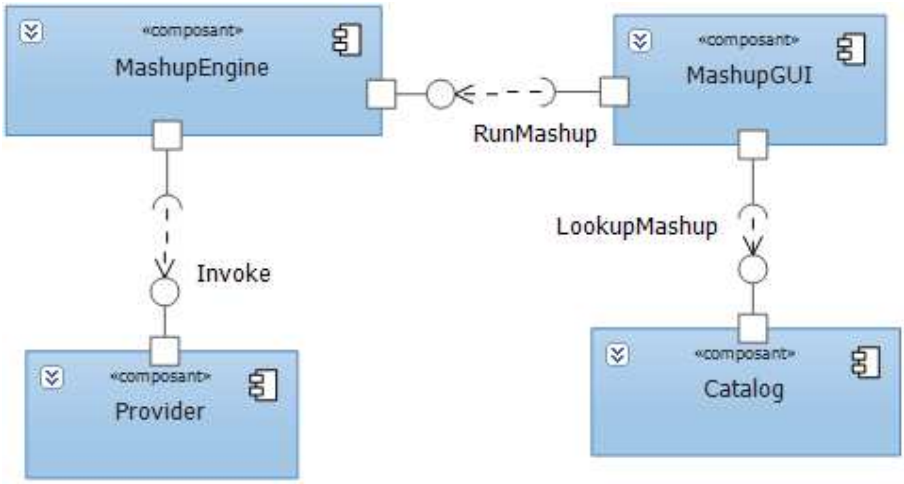

Figure 5 : Les composants d'une architecture générale d'un système d'exécution de mashups

[10] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[11] http://www.w3.org/Protocols/rfc2616/rfc2616.html
[12] http://www.w3.org/TR/soap/

2.2.2 Déploiement

L'amélioration de la disponibilité de données des mashups est impactée par les choix faits au niveau du déploiement des composants présentés dans la sous-section 2.2.1. En effet, sa portée est limitée à l'endroit où les mashups sont exécutés.

D'un point de vue du déploiement, les systèmes d'exécution de mashups sont classés en 2 catégories [47][48][49][11] : les systèmes où les mashups sont exécutés côté serveur (*server-side mashup*) et les systèmes où les mashups sont exécutés côté client (*client-side mashup*).

(a) Côté client

Dans un mashup exécuté côté client (*client-side mashup*) la récupération et la manipulation de données se produisent chez le client, généralement dans un navigateur Web. La Figure 6 présente le diagramme de déploiement dans un environnement où les mashups sont exécutés côté client. On y remarque que le composant responsable de l'exécution des mashups est déployé sur la machine de l'utilisateur final. Ce choix de déploiement est adopté par l'outil Intel MashMaker [50]. Celui ci fonctionne en tant qu'une extension de navigateur web qui manipule de données contenues dans des pages web.

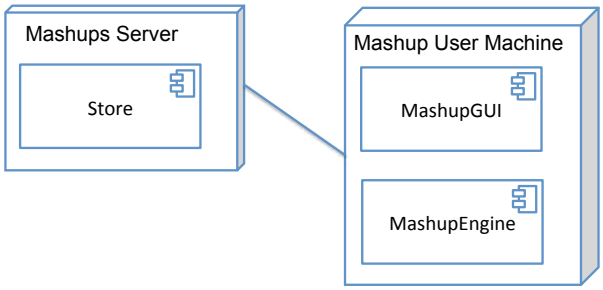

Figure 6: Diagramme de déploiement des composants d'un système d'exécution de mashups (côté client)

La Figure 7 présente le diagramme de communication[13] illustrant le fonctionnement d'un mashup exécuté côté client :

1. L'utilisateur lance le mashup via l'interface graphique (MashupGUI). Ainsi l'interface envoie une requête au serveur des mashups (MashupsServer) pour demander le code source du mashup des paramètres donnés.
2. Le code source du mashup est chargé sur la machine de l'utilisateur. Ce code comprend un script JavaScript qui contient des requêtes pour la récupération des contenus des services de données.
3. La machine du client envoie des demandes de récupération de données auprès des fournisseurs de données (Service).
4. Les fournisseurs de données retournent les données demandées.
5. Les données récupérées sont agrégées pour produire le résultat du mashup.

Pour des raisons de sécurité, les navigateurs appliquent la politique « *Same Origin Policy*[14] » : interdisant, dans une page web, la communication entre le client et plusieurs serveurs

[13] Dans la norme UML 1, le diagramme de communication s'appelait diagramme de collaboration.

[14] http://www.w3.org/Security/wiki/Same_Origin_Policy

(*cross-domain communication*)[15]. Pour cela la plupart de outils de mashups étudiés privilégient l'exécution des mashups sur un serveur dédié et non sur les machines des clients.

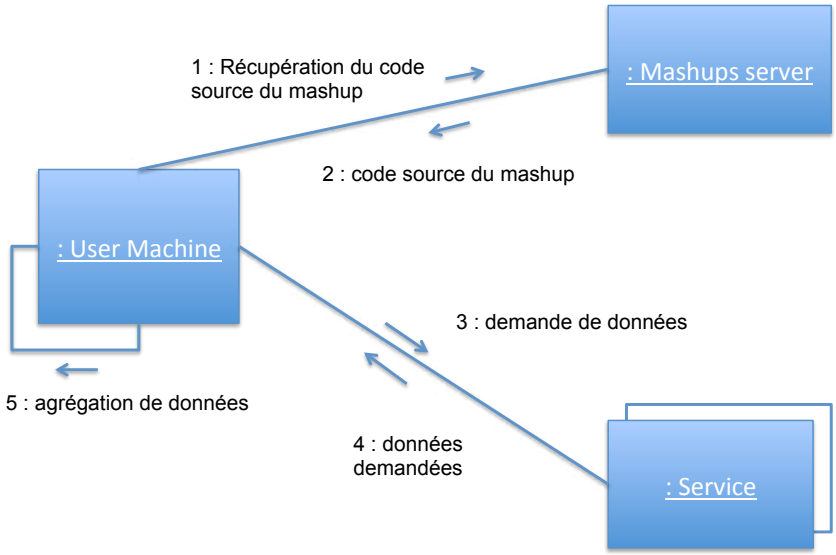

Figure 7 : Diagramme de communication illustrant le fonctionnement d'un mashup exécuté côté Client

(b) Côté serveur

Dans un mashup exécuté côté serveur (*server-side mashup*) la récupération et la manipulation de données se produisent sur le server de mashups. La Figure 8 présente le diagramme de déploiement dans un environnement où les mashups sont exécutés côté serveur. On y remarque que le composant responsable de l'exécution des mashups est déployé sur un serveur de mashups. Ce choix de déploiement est adopté par la plupart de systèmes d'exécutions de mashups étudiés. Plusieurs raisons justifient ce choix : la politique *Same Origin Policy* ne s'applique pas dans ce cas (voir sous-section 2.2.2(a)) . Ce choix permet d'alléger la quantité de données envoyées au client. Le serveur peut gérer les messages d'erreur des fournisseurs des données avant de retourner un résultat au client.

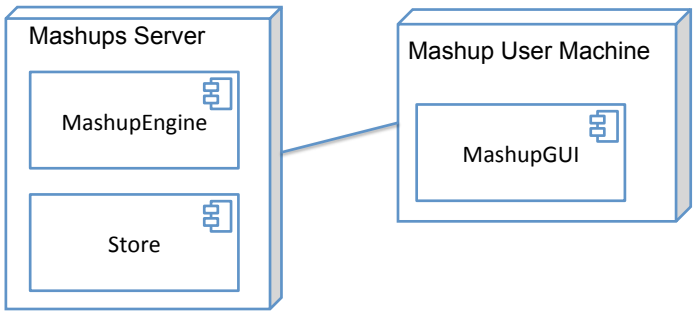

Figure 8 : Diagramme de déploiement des composants d'un système d'exécution de mashups (côté serveur)

La Figure 9 présente le diagramme de communication illustrant le fonctionnement d'un mashup exécuté côté serveur :

1. L'utilisateur lance le mashup via l'interface graphique (MashupGUI). Ainsi l'interface envoie une requête au serveur des mashups pour exécuter le mashup avec des paramètres donnés.

[15] Récemment, HTML5 [51] inclut dans ses spécification une caractéristique *Cross-Site XMLHttpRequest* autorisant la communication du client avec plusieurs serveurs.

2. Le serveur de mashups (MashupsServer) envoie des demandes de récupération de données auprès des fournisseurs de données (Service).
3. Les fournisseurs de données retournent les données demandées.
4. Les données récupérées sont agrégées pour produire le résultat du mashup.
5. Le résultat du mashup est retourné à l'interface graphique.

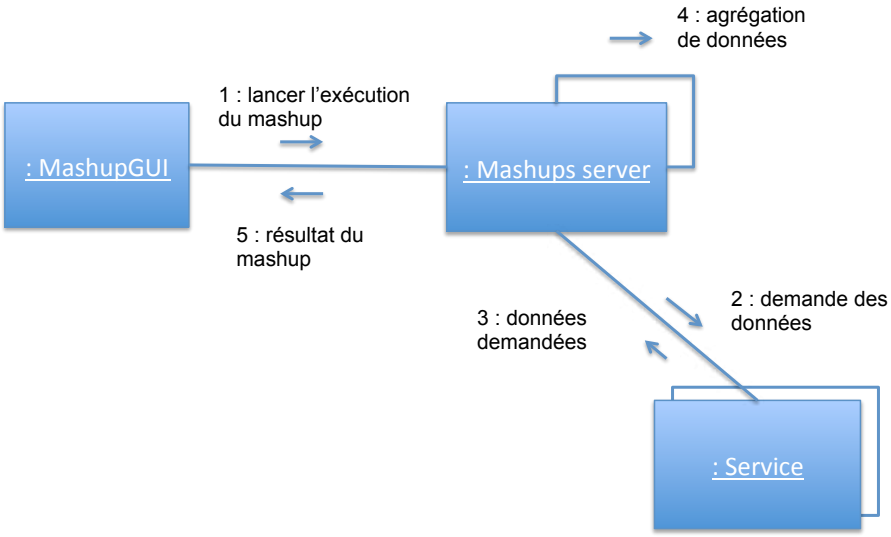

Figure 9 : Diagramme de communication illustrant le fonctionnement d'un mashup exécuté côté serveur

2.3 Disponibilité de données fraiches sur le web

La disponibilité désigne le fait de pouvoir être rapidement utilisé, d'être à la disposition de quelqu'un. En informatique, la disponibilité de données (*data availability*) signifie avoir les données obtenues et accessibles à n'importe quel moment [52][53][54].

La fraicheur caractérise l'état de ce qui est frais. En informatique, la fraicheur de données (*data freshness*) signifie que les données qu'un utilisateur a à disposition sont les mêmes que celles qui peuvent être fournies par le fournisseur de service au même moment [55][56][57].

Cette section étudie la disponibilité de données fraiches sur le web et en particulier dans les mashups. Lors de notre étude de l'état de l'art nous avons constaté que, bien qu'il existe des études qui ont identifié la disponibilité et la fraicheur de données comme des qualités à prendre en compte pour les mashups, il n'existe pas de travaux qui ont proposé des solutions pour garantir la disponibilité des données fraiches dans les mashups. Nous avons identifié (1) des travaux proposant des formules pour mesurer les qualités de la disponibilité et de la fraicheur de données dans les mashups et (2) d'autres travaux proposant des solutions pour améliorer la disponibilité des donnés sur le web.

Généralement, la disponibilité de données est garantie et améliorée par différentes solutions logicielles [58][54] comme la réplication [59][60][61] ou le cache [62][63]. Ces solutions peuvent être accompagnées d'une solution orthogonale matérielle qui consiste en la redondance du matériel au niveau des fournisseurs des données [64][65][66].

2.3.1 Mesure de la disponibilité de données dans les mashups

Dans les mashups, la disponibilité des données correspond à la capacité du mashup à fournir les données attendues par l'utilisateur. La disponibilité des données d'un mashup est

relative à la disponibilité de données des fournisseurs. L'exécution d'un mashup peut souffrir d'une indisponibilité des données qui pourrait être causée par de nombreux facteurs :

- L'indisponibilité des fournisseurs de données.
- La déconnexion du client.
- Des limitations d'accès possibles, telles que les licences d'utilisation des services. En fonction du contexte d'utilisation, on peut considérer ces limitations comme des restrictions diminuant la qualité des fournisseurs ou une décision nécessaire pour prévenir la surcharge des fournisseurs qui peut diminuer leur disponibilité.

Dans [67], les auteurs présentent cette qualité sous le nom la complétude des données. (*data completeness*). Celle ci fait référence à la capacité d'un mashup de produire toutes les valeurs de données attendues. Elle est évaluée en estimant le rapport entre la quantité de données récupérées $|RDS|$ et la quantité de données attendues $|IDS|$:

$$C = \frac{|RDS|}{|IDS|}$$

D'autre part, les auteurs définissent la disponibilité d'un mashup comme étant la possibilité de fournir les données d'au moins un des mashlets. Ainsi elle est exprimée selon la formule suivante :

$$Avail = 1 - \prod_{k=1}^{n} (1 - Avail_k)$$

Dans cette formule, la valeur de $Avail$ vaut 0 (non disponible) ou 1 (disponible). $Avail_k$ désigne la disponibilité d'un mashlet k du mashup. Si les données d'un des mashlets sont disponibles (i.e. $\exists k | Avail_k = 1$) alors le produit dans la formule est nul et le mashup est considéré disponible ($Avail = 1 - 0 = 1$). Inversement, si les données de tous les mashlets sont non disponibles (i.e. $\forall k, Avail_k = 0$) alors le produit dans la formule vaut 1 et toutes le mashup est non disponible ($Avail = 1 - 1 = 0$).

2.3.2 Rafraichissement de données des mashups

Dans certains cas, le mashup a besoin que les données soient générées et mises à jour en permanence (par exemple, marchés boursiers ou données météorologiques). Un grand nombre de décisions stratégiques, en particulier dans les entreprises, sont généralement faites selon les derniers états ou valeurs des données. Il est donc important qu'un système de mashups propage les mises à jour des sources de données pour les utilisateurs concernés.

Dans [68], les auteurs définissent la qualité de la fraicheur d'une donnée sous le terme *Timeliness*. Elle est mesurée avec la formule suivante :

$$Timeliness = \left(max\left(0, 1 - \frac{currency}{volatiliy}\right)\right)^s$$

Dans cette formule, l'exposant s est un facteur qui permet de contrôler la sensibilité de la valeur de la fraicheur au rapport $\frac{currency}{volatiliy}$. La valeur de s dépend du contexte et elle est fixée d'une façon subjective et elle est sujette d'un jugement de celui qui analyse les données. Dans cette formule la valeur de $Timeliness$ varie de 0 à 1. La valeur de $currency$ exprime l'âge de la donnée alors que la valeur de $volatiliy$ exprime la durée de vie fixée pour la donnée selon le contexte. Les données fraiches sont ceux dont le rapport $\frac{currency}{volatiliy}$ est inférieur à 1.

Dans [67], les auteurs expriment dans une formule la qualité de la fraicheur d'un mashup par rapports aux valeurs du paramètre $Timeliness$ des données du mashups. La formule est la suivante :

$$Timeliness_{mashup} = f(Timeliness_1, \dots, Timeliness_n)$$

Où f peut être la fonction maximum, minimum ou moyenne. Elle est définit en fonction du contexte par celui qui analyse les données.

La fraicheur des données du web est assurée par un mécanisme de rafraîchissement des données. Il peut être fait suivant deux stratégies [69][70] : *pull* et *push*. À l'heure actuelle, les systèmes d'exécution de mashup n'implantent que la stratégie *pull*. La valeur de la fréquence de récupération est statique. Il y a deux stratégies pour gérer les intervalles de rafraichissement dans les systèmes d'exécution de mashups [69] :

- Stratégie globale : Dans cette stratégie l'intervalle de rafraichissement est le même pour toutes les données. il est fixé par le système d'exécution de mashup ou l'utilisateur final. Ce choix est adopté dans :
  - Yahoo! Pipes : une heure selon [69].
  - Popfly : l'intervalle dépend de la fréquence du chargement de la page par l'utilisateur final.
  - Intel MashMaker : l'intervalle est fixé par défaut par le système d'exécution du mashup. Cependant l'utilisateur final peut demander le rafraichissement d'une page.
- Stratégie locale : Dans cette stratégie, un intervalle spécifique de rafraichissement est affecté à chaque source de données. Cette stratégie est adoptée par Damia [38] où les auteurs supposent les valeurs des intervalles de rafraichissement sont indiquées par les fournisseurs de données.

2.3.3 Disponibilité de données par la réplication

La réplication de données est la création de copies multiples (nœuds de réplication) des données avec l'objectif d'améliorer leur disponibilité et leur fiabilité et d'améliorer les performances d'accès à ces données [61][59]. La plupart des travaux visant à améliorer la disponibilité de données sont basés sur un processus de réplication. Malgré ces avantages, la réplication a un coût qui consiste en le maintien de la cohérence entre les copies d'une même donnée. D'où la naissance d'un compromis entre deux contradictions majeures : assurer la cohérences des copies toute en conservant des QoS de performance. Chaque copie de données est accompagnée de la copie du système d'interrogation sous-jacent. Une politique de réplication fait référence à l'algorithme gérant les différentes copies d'une même donnée [71]. Il existe deux stratégies de propagation des mises à jour : la stratégie impatiente (*eager replication*) [72]. et la stratégie paresseuse (*lazy replication*) [73]. Dans le premier cas, la mise à jour est appliquée sur toutes les copies en même temps, au sein d'une même transaction (le nœud traitant la transaction envoie des messages aux autres nœuds avant de procéder au commit). Alors que dans le second cas, la mise à jour est appliquée à une seule copie par la transaction originelle. Elle est propagée de manière asynchrone vers les autres copies.

Dans [74], les auteurs proposent d'améliorer la disponibilité des bases de données en adaptant la stratégie impatiente. Ce choix est justifié par la possibilité de détection de conflits avant la fin de la transaction ce qui augmente le niveau de cohérence entre les nœuds. Leur proposition d'adaptation vise à réduire les messages (qui correspondent à des mises à jour) entre les nœuds. Pour cela ils proposent de transférer ces messages au sein d'une même transaction et ceci une fois les opérations de lecture en cours sont terminées.

Dans [75], les auteurs proposent d'améliorer la disponibilité de données en adaptant la stratégie paresseuse. Ce choix est justifié par la rapidité de traitement des transactions. Normalement dans un processus de réplication, les opérations doivent être exécutées toutes dans le même ordre. Dans leur contribution, les auteurs décrivent un type d'opérations pouvant ne pas respecter l'ordre d'exécution donnant lieu à des meilleurs performance ceci sans altérer la cohérence des nœuds. Ils proposent trois types d'opérations (avec des exemples des opérations sur un système de messagerie, où les données sont répliquées) :

- Opérations causales : l'exécution de l'une affecte l'exécution de l'autre. Par exemple : l'envoie et la lecture de messages.
  - Opérations forcées : elles sont ordonnées entre elles mais pas forcement par rapport aux autres opérations. Par exemple : l'ajout et la suppression d'un contact.
  - Opérations immédiates : elles sont exécutées en ordre par rapport à toutes les autres opérations. Par exemple : la suppression multiple.
- Opérations non causales : elles n'ont aucune dépendance. Par exemple : la lecture d'un message m1 et celle d'un autre m2.

Dans [61], les auteurs étudient les approches de la propagation des messages entre les nœuds afin d'assurer la disponibilité de données :

- Dans l'approche de la réplication passive primaire-sauvegarde (Primary-Backup), un nœud est désigné comme étant primaire. Il reçoit la requête (d'interrogation ou de mise à jour) d'un client, l'exécute et propage le résultat aux autres nœuds (backup). Ces derniers interviennent en cas de défaillance au niveau de la copie primaire. L'avantage de cette approche est d'assurer que toutes les copies sont dans le même état.
- Dans l'approche réplication active ou en chaine (Chain Replication), le client soumet sa requête à un des nœuds, cette requête est propagée en chaine d'un nœud à un autre. Chaque nœud exécute la requête et le dernier nœud transmet le résultat au client. L'avantage de cette approche est que n'importe quel nœud peut être interrogé.

2.3.4 Disponibilité de données par le cache

L'utilisation de caches de données sur le réseau Internet a suscité l'attention de différents travaux de recherche dans différents domaines : données du web [76][63][77], les systèmes de gestion de bases de données [78][79][80], les entrepôts de données [81][82][83] ainsi que les moteurs de recherche [84][85][86]. Dans [63], les auteurs ont dressé les avantages de l'utilisation du cache. Parmi ces avantages, nous trouvons l'amélioration de la disponibilité de données en cas d'une défaillance au niveau du fournisseur ou du réseau.

Dans [58], les auteurs proposent une solution pour améliorer la disponibilité de données scientifiques utilisées par des chercheurs pour procéder à des calculs. Ils proposent d'utiliser les espaces disque disponibles sur leurs ordinateurs de bureau (au sein d'un réseau local LAN) pour cacher les données récupérées depuis des sources externes. Chaque ordinateur est considéré comme un nœud. Un de ces nœuds maintient des informations sur l'état des autres nœuds, la distribution des données cachées sur les nœuds et les informations sur la récupération des données (URI, protocoles de récupération etc.

Le cache de données pour les services web a été étudié dans différents travaux. Dans [87], les auteurs proposent une architecture de progiciels pour cacher les réponses XML des web services. Dans [88], les auteurs estiment que la description WSDL des services web manque d'informations pour mettre en place un mécanisme de cache. Ils proposent une version étendue de WSDL pour définir les données qui doivent être cachées. Dans [89], les auteurs constatent que les

réponses XML des services web nécessitent une analyse (avec l'interface DOM par exemple) avant d'être exploitées. Ainsi, ils proposent de cacher le résultat de l'analyse (arbre DOM).

D'autres travaux se sont intéressés au cache des résultats de services web dans les environnements mobiles où le cache permet de maintenir l'accès aux données même en cas de perte de connexion avec les fournisseurs des données. Dans [90], les auteurs présente une implantation d'un cache sur un dispositif mobile afin d'assurer la disponibilité de données en cas de déconnexion. Ils présentent ensuite des défis à traiter dans ce domaine :

- Développer la sémantique des opérations offertes par le web service. Ces opérations sont hétérogènes. Il faut au moins (1) pouvoir identifier si une opération exécute une mise à jour, (2) savoir si des données obsolètes d'une opération restent acceptables pour l'utilisateur.
- Maintenir la cohérence du cache. pendant la déconnexion du dispositif mobile, la cohérence ne peut pas être garantie fortement. En plus il faut prendre en compte des opérations des mises à jours qui affectent la cohérence de données dans le cache. Par exemple, supposons que le cache contient une liste de contacts. La demande de suppression d'un contact affecte la cohérence de la liste stockée dans le cache. Ainsi il faudrait mener des études qui permettent au cache de pouvoir invalider des données suite à l'exécution d'autres requêtes.
- Étudier l'expérience de l'utilisateur, pour évaluer l'impact de la solution proposée.
- En cas de déconnexion, comment le cache doit réagir lorsqu'il reçoit une nouvelle requête. Les auteurs proposent comment idée, d'étudier la possibilité d'une réponse par défaut adaptée à chaque requête.
- Étudier et proposer des mécanismes de préchargement de données afin d'améliorer la disponibilité des données.

Dans [91], les auteurs proposent un cache distribué pour améliorer la disponibilité des données dans les environnements mobiles. Les données sont cachées sur des terminaux mobiles. Chacun de ces terminaux met son cache à disposition des autres terminaux. L'architecture (cf. Figure 10[16]) du cache est basée sur l'attribution de rôles aux terminaux mobiles :

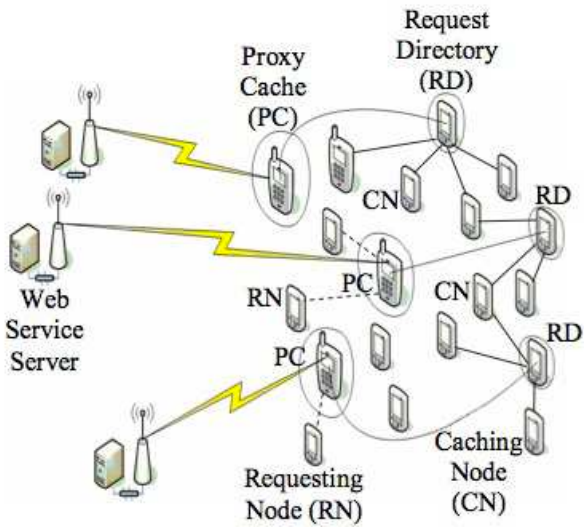

Figure 10 : Architecture du cache distribué proposé dans [91]

- Nœuds de cache (Caching Nodes CNs) : ils stockent les résultats des requêtes.

[16] Figure obtenue de [91]

- Répertoires de requêtes (Request Directories RDs) : pour une requête donnée, ils indiquent les nœuds de cache qui contiennent sont résultat.
- Les nœuds interrogateurs (Requesting Nodes RNs) : ce sont les terminaux qui émettent les requêtes et qui pourraient devenir ensuite des nœuds de cache s'ils sont amenés à stocker le résultat de la requête (au cas où aucun des autres nœuds de cache ne contient le résultat).
- Proxys des caches (Proxy Caches PCs) : ils font office d'intermédiaires entre les nœud interrogateurs qui soumettent les requêtes et les autres terminaux et les fournisseurs de données.

Cette proposition impose une bonne disponibilité matérielle des proxys des caches qui constituent un point central pour assurer la communication entre les terminaux et même avec les fournisseurs de données. En plus, ses bénéfices dépendent de la connectivité des terminaux mobiles. Les auteurs n'aborde pas dans leur proposition les problèmes liées aux remplacement et le rafraichissement des données du cache.

Dans [92], les auteurs proposent un cache de navigateur associé à iMashup [93] : un outil de création de mashups exécutés côté client. Le cache sert à stocker, sur la machine de l'utilisateur, les données récupérées depuis les fournisseurs de données externes. Une entrée du cache a une durée de vie adaptative V. Celle ci est inversement proportionnelle à la fréquence d'accès à cette donnée, et proportionnelle au nombre de succès de cache lors de la demande d'accès. Elle est définie avec les formules suivantes :

$$temp = e^{\frac{\alpha \times hit_ratio}{f(FA)}}$$

si $temp > S$ alors $V = temp$ sinon $V = S$

hit_ratio correspond au nombre des succès de cache avec un poids α (dans l'implantation $\alpha = 1000$). $f(FA)$ est une fonction croissante par rapport à la fréquence d'accès FA. Cette formule indique que pour une entrée du cache, (1) plus le nombre de succès de cache est élevé plus il faut augmenter la durée de vie, et (2) et plus la fréquence d'accès à une donnée est élevée, plus il faut diminuer la durée de vie.

Dans [39], les auteurs présentent MACE un système de cache pour les données des mashups. Ils proposent un modèle de mashups (cf. sous-section 2.1.8), où chaque mashup est vu comme un arbre dont les nœuds sont des opérateurs (similaire à un « pipe » dans Yahoo! Pipes). Un arbre peut être commun à plusieurs mashups ou utilisé comme un sous-arbre dans un arbre d'un autre mashup. Les auteurs y intègrent un cache qui sélectionne dynamiquement le nœud dont les données produites doivent être cachées. Chaque nœud n possède (1) une valeur de coût associée CV_n qui correspond à la latence de son exécution et une valeur de coût cumulatif CCV_n qui représente la somme définie par la formule suivante :

$$CCV_n = CV_n + \sum_{i \text{ est un noeud fils de } n} CV_i$$

Le choix du nœud à cacher prend en compte la fréquence d'exécution du nœud et la fréquence des mises à jour : il est plus intéressant de cacher les données du nœud qui est le plus fréquemment exécuté et qui demande le moins de mises à jour. La différence de fréquence d'exécution entre les nœuds d'un même arbre vient du fait qu'un nœud (et ses nœuds fils) peut faire partie d'un autre arbre. Ainsi le nœud k choisi est celui qui maximise la valeur de l'expression $CBT_k = RF_k \times CCV_k - UF_k \times CCV_k$ où RF_k et UF_k dénotent respectivement la fréquence d'exécution et la fréquence de mise à jour du nœud k. Si le cache ne peut pas contenir, dans son espace libre, les données produites par le nœud, alors les données d'un autre nœud (le suivant qui maximise

l'expression du CBT) sont choisies pour être cachées. Les auteurs n'aborde pas dans leur proposition les problèmes liées aux remplacement et le rafraichissement des données du cache.

2.4 Conclusion

Nous avons dressé un état de l'art des travaux sur les modèles des mashups, les systèmes d'exécution de mashups et leurs architectures ainsi que la disponibilité de données fraiches sur le web.

L'étude des modèles et de certains outils de création de mashups nous a permis d'identifier les concepts nécessaires pour décrire un mashup. Nous avons remarqué que dans ces travaux, la définition du mashup n'est pas la même : certains ont défini le mashup comme étant un ensemble de mashlets alors que d'autres ont restreint la définition d'un mashup à un mashlet. D'un autre côté les outils de création de mashups qui permettent l'échange de données entre les mashlets, définissent les wirings comme des simples intermédiaires qui permettent le transfert de données. Nous trouvons que cette conception de wiring est pauvre. Ceux ci peuvent être enrichis pour permettre d'effectuer un mapping de données lors du transfert entre le mashlet envoyeur et le mashlet receveur : ceci rend le mashlet receveur indépendant du format de données envoyées par le mashlet envoyeur. En plus, nous trouvons que ces modèles n'intègrent pas la spécification des caractéristiques de la disponibilité de données dans les mashups.

Lors de analyse des approches existantes pour améliorer la disponibilité de données sur le web, nous avons identifiée des solutions à base de réplication et à base de cache. Ces solutions sont difficilement applicables dans le contexte de mashups. En effet, la réplication doit être mise en place au niveau des fournisseurs de données qui sont autonomes ou bien au niveau du client de ces fournisseurs : le système d'exécution des mashups. Or, dans ce cas, il faut répliquer des données hétérogènes avec des systèmes d'interrogations spécifiques à ces données. Dans le contexte des mashups, les fournisseurs de données ne sont pas connus à l'avance (avant la création des mashups), ainsi pour chaque nouveau fournisseur identifié, il faut lui demander l'accès à ses données internes et à son systèmes d'interrogation pour les répliquer, ce que nous considérons comme non garanti. D'autre part, les solutions à base de cache sont adaptées à des applications construites par des développeurs : le choix des politiques de cache sont faites par des développeurs : ils décident quelles données et à quelles étapes de la composition des services il faut cacher. Or de telles décisions ne doivent pas être laissées aux constructeurs des mashups qui ont souvent des compétences limitées en programmation et ne maitrisent pas les aspects non fonctionnels de l'environnement de construction des mashups. Bien que la plateforme MACE propose de cacher les données des mashups, cependant le fonctionnement du cache est défini avec l'objectif de diminuer la latence de l'exécution des mashups et non pour assurer la disponibilité des données des mashups. D'autre part nous trouvons que dans la solution de cache de [92] l'estimation d'une durée de vie adaptative ne prend pas prendre en compte des facteurs comme l'ancienne valeur de la durée de vie, la date du dernier accès à la donnée et la comparaison d'une donnée récupérée à celle existante en cache. Dans toutes ces solutions il manque la prise en compte de la fraicheur de données. Dans le Chapitre 3, nous proposons un modèle de mashup basé sur les modèles de valeurs complexes [19]. Ce modèle nous permettra de spécifier les caractéristiques de la disponibilité de données dans le Chapitre 4.

Chapitre 3

MODELE DE MASHUPS

Ce chapitre présente les concepts de notre modèle de mashups : **Donnée**, **Activité**, **Mashlet**, **Wiring**, et **Mashup**. Une activité peut être basique ou composite. Une activité basique décrit une tâche spécifique permettant d'accomplir un traitement de données (le filtrage, la projection, l'extraction, l'élimination de doublons, le tri, l'union). Ces activités sont coordonnées par des activités composites (*Sequence*, *Foreach*, *Parallel*). Les mashlets affichent des données produites par des activités. Tandis que les wirings définissent le flot de données entre les mashlets. Ils peuvent aussi, décrire le traitement des données nécessaire pour les délivrer sous le bon format aux mashlets destinataires. L'ensemble de ces concepts décrit des composants qui constituent des mashups.

Dans le mashup ItineraryPlanner, présenté dans le Chapitre 1 (cf. Figure 1), ces concepts se présentent comme suit :

- Les données sont :
  - récupérées depuis des services (fournisseurs de données : Google Maps, Yahoo! Weather et Google Places). Ces services sont autonomes et indépendants des mashups.
  - ou saisies par l'utilisateur (viles de départ de et de destination)…
  - ou traitées et échangés entre les mashlets (villes de passages sur l'itinéraire).
- Les activités qui décrivent le traitement de ces données (extraction de données, filtrage…).
- Les mashlets (Map, Weather, Restaurants) constituent les composants qui affichent les résultats produits par des activités sous-jacentes. Ces composants sont réutilisables. C'est à dire un mashlet peut être utilisé dans la définition de différents mashups.
- Le transfert des données entre les mashlets se fait via des wirings (la liste de villes de passage de Map vers Weather, les coordonnées de la ville d'arrivée de Map vers Restaurants). Les wirings procèdent à la manipulation des données transférées afin de les livrer dans le bon format.
- Le mashup ItineraryPlanner est constitué des mashlets Map, Weather, Restaurants et des wirings qui définissent le flot de données entre ces mashlets.

La Figure 11 présente les concepts Mashup, Mashlet, Wiring et Activités sous la forme d'une pile. Chaque mashlet et wiring est associé à une activité (basique ou composite). Un mashlet est réutilisable : il peut faire partie de plusieurs mashups. Un wiring appartient au mashup qui contient les mashlets sous-jacents.

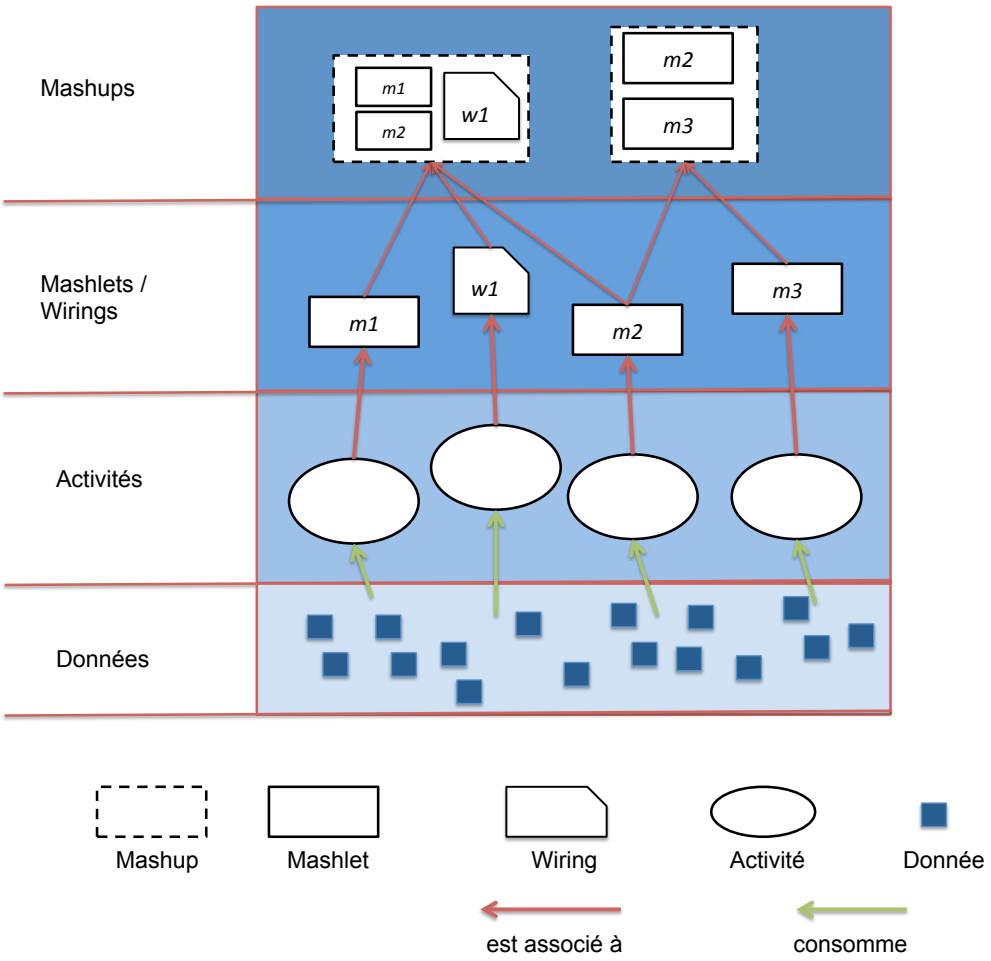

Figure 11 : Pile de concepts de mashups

Le chapitre est organisé comme suit. La section 3.1 présente la caractérisation des données des mashups. Les concepts Activités, Mashlet, Wiring, Mashup sont décrit respectivement dans les sections 3.2 à 3.5. Ces concepts sont illustrés avec des exemples décrivant le mashup ItineraryPlanner présenté dans le Chapitre 1. Enfin la section 3.6 conclut ce chapitre.

3.1 Données de mashups

Nous avons besoin d'une représentation de données qui permet l'homogénéisation de données de mashups. Pour cela, nous adoptons une caractérisation de données basée sur les valeurs complexes.

3.1.1 Types de données

Un domaine est un ensemble fini de valeurs. Nous considérons les domaines prédéfinies des chaines de caractères $\mathbb{S}$, des booléens $\mathbb{B}$, des entiers $\mathbb{Z}$, des entiers naturels $\mathbb{N}$, et des réels $\mathbb{R}$. Nous définissons le domaine des noms de types $\mathbb{A} = \{A_1, A_2, \dots\} \subseteq \mathbb{S}$ et celui des noms des instances $\mathbb{I} = \{I_1, I_2, \dots\} \subseteq \mathbb{S}$ avec $\mathbb{A} \cap \mathbb{I} = \emptyset$.

Un type de valeurs complexes t dénoté $\hat{t}$, est défini par une paire $A : def$, où A est le nom du type ($A \in \mathbb{A}$) et def est sa définition. Nous considérons les fonctions $name$ et def pour accéder respectivement aux composants de la paire. Par exemple pour le type $temperature : \mathbb{Z}$,

$name(temperature : \mathbb{Z})$ retourne $temperature$ et $def(temperature : \mathbb{Z})$ retourne $\mathbb{Z}$. Le nom est unique et il est utilisé pour référencer le type.

L'ensemble $\mathbb{T}$ de types de données est défini par les règles suivantes :

- Si D est un domaine alors $A : D$ est un type atomique.
- Si $\hat{t}$ est un type alors $A : [\hat{t}]$ est un type liste.
- Si $\hat{t}_1, \hat{t}_2, \dots, \hat{t}_n$ sont des types tels que $\forall\, i,j\ name\ (\hat{t}_i) \neq name\ (\hat{t}_j)$alors $A : \langle \hat{t}_1, \hat{t}_2, \dots, \hat{t}_n \rangle$ est un type tuple. Chaque $\hat{t}_i$ est un type caractérisant un attribut de nom $name\ (\hat{t}_i)$.
- Si $\hat{t}_1, \hat{t}_2, \dots, \hat{t}_m, \hat{t}_n, \dots \hat{t}_r$ sont des types tels que $\forall\, i,j\ name\ (\hat{t}_i) \neq name\ (\hat{t}_j)$, alors $A : (\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_m \rightarrow \hat{t}_n \times \dots \times \hat{t}_r)$ est un type fonction[17]. Pour $i \in \{1..m\}$, $\hat{t}_i$ est un type paramètre d'entrée de nom $name(\hat{t}_i)$. Pour $j \in \{n..r\}$, $\hat{t}_j$ est un type paramètre paramètre de sortie de nom $name(\hat{t}_j)$.

Dans la suite de ce document, lorsque nous introduisons un type, dans un souci de simplification, nous pourrons omettre le nom du type. Cette simplification s'applique également aux paramètres des types fonction.

Notons que l'ensemble de types $\mathbb{T}$ est lui même un domaine.

3.1.2 Instances des types

$[\![\hat{t}]\!]$ dénote l'ensemble des instances de type $\hat{t}$. Cet ensemble est défini par induction de la manière suivante :

- $[\![A : D]\!] = \{A : d | d \in D\}$. Les valeurs du domaine D sont connues.
- $[\![A : [\hat{t}]]\!] = \{A : [v_1, \dots, v_n] \mid n \in \mathbb{N}\ et\ \forall\, 1 \leq i \leq n, v_i \in [\![\hat{t}]\!]\}$.
- Pour un type tuple $A : \langle \hat{t}_1, \hat{t}_2, \dots, \hat{t}_n \rangle$ où $\hat{t}_i$est de la forme $A_i{:}\,def_i$, $[\![A : \langle \hat{t}_1, \hat{t}_2, \dots, \hat{t}_n \rangle]\!] = \{A : \langle A_1{:}\,v_1, A_2{:}\,v_2, \dots, A_n{:}\,v_n \rangle \mid \forall\, i \in [1..n], A_i : v_i \in [\![\hat{t}_i]\!]\}$.
- $[\![A : (\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_m \rightarrow \hat{t}_n \times \dots \times \hat{t}_r)]\!] = \{f \mid f\ est\ une\ fonction\ dont\ la\ signature\ est\ \hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_m \rightarrow \hat{t}_n \times \dots \times \hat{t}_r\}$.

Nous définissons le domaine $Data$ incluant toutes les instances possibles et le domaine $Functions$ incluant toutes les instances possibles de type fonction.

$$Data = \bigcup_{\hat{t} \in \mathbb{T}} [\![\hat{t}]\!]$$

$$Functions = \bigcup_{\hat{t}\ est\ de\ type\ fonction} [\![\hat{t}]\!]$$

$$Functions \subset Data$$

Nommage des instances

Chaque instance $A : v$ d'un type $\hat{t} \in \mathbb{T}$ peut être nommée comme suit :

$n = A : v$. Où $n \in \mathbb{I}$ est le nom de l'instance.

Exemples : La suite présente des exemples de types et instances utilisés dans le mashup ItineraryPlanner décrit dans le Chapitre 1 (cf. Figure 1).

[17] Dans notre modèle, une fonction possède plusieurs paramètres de sortie ; ce type de fonctions a été introduit par Maurice Fréchet dans ses travaux sur l'analyse fonctionnelle.

Le service Yahoo! Weather retourne la météo d'une localisation décrite sous format WOEID[18]. Par exemple l'instance $woeid : 12724717$ correspond à la localisation de la ville de Grenoble. Elle est de type $woeid : \mathbb{N}$.

Le type $woeids : [woeid : \mathbb{N}]$ décrit une liste de localisations de villes décrites sous format WOEID. Par exemple l'instance $woeids : [woeid : 12724717,\ woeid: 12724728,\ woeid : 12726958,\ \mathrm{woeid} : 20068148,\ woeid : 20068141]$ correspond à la liste de localisations des villes de passage entre Grenoble et Paris dans ItineraryPlanner et décrites sous format WOEID.

$restaurant : \langle name : \mathbb{S}, speciality : \mathbb{S}, address : \mathbb{S}, rating : \mathbb{R}\rangle$ décrit le type tuple $restaurant$ avec les attributs : le nom du restaurant, sa spécialité, son adresse et sa note. L'expression $rest = restaurant : \langle name: "nom1", speciality: "française", address : "ad1", rating : 4.5\rangle$ définit une instance de ce type nommée $rest$.

L'itinéraire entre deux villes est décrit dans un type tuple, nommé routes[19], donnée ci dessous. On y retrouve par exemple les attributs suivants :

- Un tuple nommé **legs** (manches). Il contient des attributs indiquant la distance, la durée totale, les adresses départ et d'arrivée de l'itinéraire et les étapes (**steps**) de l'itinéraire sous la forme d'une liste.
- Chaque étape contient des informations sur sa distance (**distance**), durée (**duration**), points de départ et d'arrivée (**start_location** et **end_location**) des instructions (**html_instructions**). Chacune est décrite sous la forme d'une chaine de caractère.

```
routes : ⟨
         bounds : ⟨
            northeast : ⟨
               lat : ℝ,
               lng : ℝ
            ⟩,
            southwest : ⟨
               lat : ℝ,
               lng : ℝ
            ⟩
         ⟩,
         copyrights : 𝕊,
         legs :⟨
               distance : ⟨
                  text : 𝕊,
                  value : ℝ
               ⟩,
               duration : ⟨
                  text : 𝕊,
                  value : ℝ
               ⟩,
               end_address : 𝕊,
               end_location : ⟨
                  lat : ℝ,
                  lng : ℝ
               ⟩,
               start_address : 𝕊,
```

[18] Un identifiant WOEID (Where on Earth IDentifier) est un identifiant de référence assigné par Yahoo! Pour identifier des points géographiques sur la terre.

[19] La définition du type routes est basée sur la structure du document JSON renvoyé par le service Google Maps

```
                start_location : ⟨
                   lat : ℝ,
                   lng : ℝ
                ⟩,
                steps : [
                   step : ⟨
                      distance : ⟨
                         text : 𝕊,
                         value : ℝ
                      ⟩,
                      duration : ⟨
                         text : 𝕊,
                         value : ℝ
                      ⟩,
                      end_location : ⟨
                         lat : ℝ,
                         lng : ℝ
                      ⟩,
                      html_instructions : 𝕊,
                      polyline : ⟨
                         points : 𝕊
                      ⟩,
                      start_location : ⟨
                         lat : ℝ,
                         lng : ℝ
                      ⟩,
                      travel_mode : 𝕊
                   ⟩
                ]
             ⟩
   ⟩
```

L'exemple suivant présente une partie d'une instance[20] de ce type donnant l'itinéraire entre Grenoble et Paris, nommé $routeGP$.

```
routeGP = routes : ⟨
          bounds : ⟨
             northeast : ⟨
                lat : 48.857240,
                lng : 5.724690000000001
             ⟩,
             southwest : ⟨
                lat : 45.18839000000001,
                lng : 2.306130
             ⟩
          ⟩,
          copyrights : "Données cartographiques ©2012 GeoBasis-DE/BKG (©2009), Google",
          legs :⟨
                distance : ⟨
                   text : "574 km",
                   value : 573623
                ⟩,
                duration : ⟨
                   text : "5 heures 21 minutes",
                   value : 19284
                ⟩,
                  end_address : "Paris, France",
```

[20] Le tuple $routeGP$ est similaire à l'objet JSON obtenu par invocation du service Google Maps : http://maps.googleapis.com/maps/api/directions/json?origin=Grenoble&destination=Paris&sensor=false&

```
                end_location : ⟨
                   lat : 48.857240,
                   lng : 2.35260
                ⟩,
                start_address : "Grenoble, France",
                start_location : ⟨
                   lat : 45.18839000000001,
                   lng : 5.724690000000001
                ⟩,
                steps : [
                   step : ⟨
                      distance : ⟨
                         text : "92 m",
                         value : 92
                      ⟩,
                      duration : ⟨
                         text : "1 minute",
                         value : 7
                      ⟩,
                      end_location : ⟨
                         lat : 45.188890,
                         lng : 5.725610000000001
                      ⟩,
                      html_instructions : "Prendre la direction \u003cb\u003enord-
est\u003c/b\u003e sur \u003cb\u003ePl. Victor Hugo\u003c/b\u003e vers \u003cb\u003eRue
Paul Bert\u003c/b\u003e",
                      polyline : ⟨
                         points : "mzxrGib}a@g@gA{@oB"
                      ⟩,
                      start_location : ⟨
                         lat : 45.18839000000001,
                         lng : 5.724690000000001
                      ⟩,
                      travel_mode : "DRIVING"
                   ⟩,
                   step : ⟨…⟩,
                   ……
                ],
         ⟩,
……
⟩
```


3.1.3 Types et valeurs d'instances

Pour accéder aux types et aux valeurs des instances de la forme $A : value$, nous définissons les fonctions $type$ et val de la manière suivante :

- $type(A : value) = A$. Nous rappelons qu'un type $A : def$ est référencé par son nom et qu'il est possible d'inférer le type $A : def$ à partir de A.
- $val(A : value) = value$.

Par exemple, si v dénote $woeids : [woeid : 12724717, woeid : 12724728, woeid : 12724752, woeid : 12726960, woeid : 582081]$, nous avons :

- $type(v) = woeids$
- $val(v) = [woeid : 12724717, woeid : 12724728, woeid : 12724752, woeid : 12726960, woeid : 582081]$

3.1.4 Opérations sur les instances

Chaque type possède un ensemble d'opérations qui définit le comportement de ses instances. Toute opération est d'un type fonction ($\in Functions$), en considérant qu'elle a pour paramètres de types $\hat{t}_1, \hat{t}_2, \ldots, \hat{t}_n$ et pour signature $\hat{t} \times \hat{t}_1 \times \hat{t}_2 \times \ldots \times \hat{t}_n \rightarrow \hat{t}_m$. Pour une instance i de $\hat{t}$ et

des instances $v_i \in [\![\hat{t}_i]\!]$, nous adoptons une notation pointée : $i.op(v_1, \dots, v_n)$ pour designer l'appel de l'opération op sur l'instance i avec les instances v_i en paramètres d'entrée.

Parmi les opérations sur les types, nous retrouvons des opérations qui définissent les relations d'ordre entre les instances d'un type $\hat{t}$ comme $less$, $greater$, $lessOrEqual$, $lessOrEqual$, $greaterOrEqual$ et $equal$. Les types de ces opérations sont de la forme $\hat{t} \times \hat{t} \rightarrow \mathbb{B}$. Ces opérations comparent les instances selon une des stratégies définies dans [94][95] :

- Identité d'instance : comparer les identificateurs des instances.
- Comparaison superficielle : comparer les valeurs des instances.
- Isomorphisme (comparaison structurelle) : égalité si les structures sont isomorphes.
- Comparaison profonde : comparer les valeurs des feuilles des instances.

Nous définissons également l'opération $length$ qui, pour une instance V de type liste, retourne sa taille. Le type de $length$ est de la forme $([\hat{t}] \rightarrow \mathbb{N})$. L'appel de l'opération se fait avec la notation $V.length\ ()$.

Nous considérons aussi l'opération $invoke$ définie sur tous les types fonction pour réaliser l'exécution de leurs instances. Pour une fonction f, l'opération $invoke$ l'exécute avec des valeurs des paramètres données dans une liste d'instances. Elle retourne la valeur du paramètre de sortie correspondante de la fonction f. Elle est de type $(Functions \times [input : Data] \rightarrow Data)$.

Fonction $Operations$

Nous définissons la fonction $Operations$ qui pour chaque type retourne la liste des opérations qui définissent le comportement de ses instances. Elle est de type $(\mathbb{T} \rightarrow [Functions])$.

3.2 Activité

Une activité consomme et produit des données. Elle peut être basique ou composite. Une activité basique décrit une tâche spécifique permettant d'accomplir un traitement de données. Une activité composite permet de coordonner les activités qu'elles soient basiques ou composites. La représentation graphique d'une activité est donnée dans la Figure 12.

Figure 12 : Représentation graphique d'une activité

Une activité est de type fonction. Nous considérons le domaine des activités $\mathbb{O} \subset Functions$. Notons que, dans notre modèle, certaines fonctions ne sont pas forcement des activités. Comme les fonctions définissant le comportement des instances d'un type : elle sont définies en tant qu'opérations (sous-section 3.1.4).

3.2.1 Activités basiques

Nous considérons un ensemble fini d'activités basiques comme l'extraction (*Extract*), le filtrage (*Filter*), l'élimination de doublons (*Unique*), le tri (*Sort*), l'union (*Union*) et l'annexe (*Append*). Ces activités sont présentées dans les sous-sections 3.2.1.1 à 3.2.1.6.

Les fournisseurs de données exportent des méthodes représentées par des activités. Des exemples de ces activités sont donnés au cours du chapitre lors du déroulement du scénario ItineraryPlanner.

3.2.1.1 Extract

Une activité *Extract* retourne une valeur composante d'une instance V d'un certain type $\hat{t}$. Le résultat est construit à partir d'un chemin d'accès $path$.

- Notation : $\chi_{path}(V)$

  L'expression du chemin d'accès $path$ est construite avec la règle suivante :

  $$\begin{aligned} path \quad := \quad & \boldsymbol{A} \\ & \mid \boldsymbol{A}.path \\ & \mid \boldsymbol{A}[\boldsymbol{i}] \\ & \mid \boldsymbol{A}[\boldsymbol{i}].path \end{aligned}$$

  Où A y désigne le nom d'un type tandis que i est un entier naturel.

  Validité du chemin par rapport à un type

  Pour un type $\hat{t}$ donné, nous définissons $paths(\hat{t})$ comme étant l'ensemble des expressions de chemins d'accès $path$ valides par rapport à $\hat{t}$. Cet ensemble est définit comme suit :

  1. Si $name(\hat{t}) = A$ alors $path = A \in paths(\hat{t})$.
  2. Si $\hat{t}$ est de la forme $A : \langle \dots, A' : def', \dots \rangle$ et $path' \in paths(A' : def')$. alors $path = A.path' \in paths(\hat{t})$.
  3. Si $\hat{t}$ est de la forme $A : [B : def]$ alors $path = A[i] \in paths(\hat{t})$.
  4. Si $\hat{t}$ est de la forme $A : [B : def]$ et $path' \in paths(B : def)$, alors $path = A[i].path' \in paths(\hat{t})$.

- Type de l'activité : $\chi : (\hat{t} \rightarrow \hat{t}')$
- Sémantique de l'activité : $\chi_{path}(V)$ est défini comme suit

  La valeur de $\chi_{path}(V)$ est donnée par les règles suivantes :

  1. Si $type(V) = A$ et $path = A$ alors $\chi_{path}(V) = V$.
  2. Si $V = A : \langle \dots, A' : v', \dots \rangle$ et $path = A.A'$ alors $\chi_{path}(V) = A' : v'$. Cette valeur est désignée avec la notation $V.A'$.
  3. Si $V = A : \langle \dots, A' : v', \dots \rangle$ et $path = A.A'.path'$ alors $\chi_{path}(V) = \chi_{A'.path'}(A' : v')$. Cette valeur est désignée avec la notation $V.A'.path'$.
  4. Si $V = A : [B : v_1, \dots, B : v_i, \dots, B : v_n]$ et $path = A[i]$ alors $\chi_{path}(V) = B : v_i$. Cette valeur est désignée avec la notation $V[i]$.
  5. Si $V = A : [B : v_1, \dots, B : v_i, \dots, B : v_n]$ et $path = A[i].path'$ alors $\chi_{path}(V) = \chi_{path'}(B : v_i)$. Cette valeur est désignée avec la notation $V[i].path'$.

Exemple

$\chi_{routes.legs.steps[0].step.start_location}(routeGP) = start_location : \langle lat : 45{,}18839000000001, lng : 5.724690000000001 \rangle$. Cette valeur est désignée avec la notation $routeGP.legs.steps[0].step.start_location$. Et $routeGP$ est l'instance du type $route$, tous les deux définis dans la sous-section 3.1.2.

3.2.1.2 Filter

Une activité *Filter* prend en entrée une liste L d'instances de $\hat{t}$ de la forme $B : [A : v_1, \dots, A : v_n]$. Elle sélectionne les éléments de L vérifiant une condition $cond$. Elle produit comme résultat une sous-liste de L.

- Notation : $\sigma_{cond}\ (L)$
- Type de l'activité : $\sigma : ([\hat{t}] \rightarrow [\hat{t}])$

L'expression de $cond$ est définie comme suit :

Nous commençons par définir les *termes* avec les règles suivantes :

- Les valeurs constantes c des instances $A : c$ sont des termes constants. Par extension, les termes constants dénotent aussi leurs valeurs.
- Si p est un chemin d'accès alors p est un terme (chemin).
- Si f est une fonction d'arité n et que $T1 \dots Tn$ sont des termes alors $f\ (T1 \dots Tn)$ est un terme (fonction).

Nous définissons des conditions basiques (*basic conditions bcond*) sur les termes ($T1$ et $T2$) avec les règles suivantes :

$$\begin{array}{lcl} bcond & := & T1\ \theta\ T2 \\ \theta & := & = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid \in \mid \subset \mid \subseteq \end{array}$$

Les opérateurs ensemblistes et de comparaisons doivent être compatibles avec les valeurs des termes. En plus, pour faciliter le calcul de la valeur booléenne d'une condition, nous définissons l'interprétation d'un terme T, par rapport à une instance r, dénoté T^r comme suit :

- Pour les termes constants C, $C^r = C$
- Pour les termes chemins p,

$$p^r = \begin{cases} [v_1, v_2, \dots, v_n] & si\ \chi_{path}(r)\ est\ une\ liste\ de\ la\ forme\ B : [A : v_1, A : v_2 \dots, A : v_n] \\ val\left(\chi_p(r)\right) & sinon \end{cases}$$

  Dans le deuxième cas, le chemin d'accès aboutit à une valeur atomique dont la valeur est utilisée comme opérande de la condition.

- L'interprétation d'un terme fonction $f\ (T_1 \dots T_n)$ par rapport à une instance r est la valeur de l'instance résultat de l'application de la fonction f sur les interprétations des termes $T_1 \dots T_n$ par rapport à r : $f(T_1 \dots T_n)^r = val\big(f(T_1^r, \dots, T_n^r)\big)$.

Une condition basique de la forme $T1\theta T2$ est considérée comme étant interprétable par rapport à un type $\hat{t}$ si $T1$ et $T2$ sont interprétables par rapport à $\hat{t}$. Une instance r de $\hat{t}$ satisfait cette condition basique, si $T_1^r \theta T_2^r$ est vrai. Dans ce cas, nous notons $r \vDash T_1 \theta T_2$.

L'expression $cond$ est définie avec la grammaire ci dessous.

$$\begin{array}{lcl} cond & := & bcond \\ & & \mid \neg\ cond \\ & & \mid cond \wedge cond \\ & & \mid cond \vee cond \\ & & \mid cond\ quant\ path \\ quant & := & \forall \mid \exists \end{array}$$

Où $path$ est un chemin d'accès qui identifie une liste et dont la grammaire est définie dans la sous-section 3.2.1.1.

- Sémantique de l'activité

$\sigma_{cond}(L) = B : [A : v_i, \dots, A : v_k] \mid \forall j \in [i, k], A : v_j$ est présente dans L et $A : v_j \vDash cond$

Satisfiabilité des expressions de filtrage

La satisfiabilité des conditions de filtrage par rapport à une instance r est définie avec les règles suivantes :

1. $r \vDash \neg cond$ si $r \vDash cond$ est faux.
2. $r \vDash cond_1 \wedge cond_2$ si $r \vDash cond_1$ et $r \vDash cond_2$
3. $r \vDash cond_1 \vee cond_2$ si $r \vDash cond_1$ ou $r \vDash cond_2$
4. $r \vDash cond \forall pathexp$ si pour chaque $a_i \in val\left(\chi_{pathexp}(r)\right)$, $a_i \vDash cond$ et $\chi_{pathexp}(r)$ est une liste
5. $r \vDash cond \exists pathexp$ si pour un certain $a_i \in val\left(\chi_{pathexp}(r)\right)$, $a_i \vDash cond$ et $\chi_{pathexp}(r)$ est une liste

Exemple

Soit l'instance liste suivante :

$$L = restaurants : \begin{bmatrix} restaurant : \langle name : "nom1", speciality : "française", address : "ad1", rating : 4.5 \rangle, \\ restaurant : \langle name : "nom2", speciality : "italienne", address : "ad2", rating : 4 \rangle, \\ restaurant : \langle name : "nom3", speciality : "libanaise", address : "ad3", rating : 4.1 \rangle, \\ restaurant : \langle name : "nom4", speciality : "japonaise", address : "ad4", rating : 3.6 \rangle \end{bmatrix}$$

L'expression $FilterRes = \sigma_{restaurant.rating \geq 4}(L)$ produit :

$$R = restaurants : \begin{bmatrix} restaurant : \langle name : "nom1", speciality : "française", address : "ad1", rating : 4.5 \rangle \\ restaurant : \langle name : "nom2", speciality : "italienne", address : "ad2", rating : 4 \rangle, \\ restaurant : \langle name : "nom3", speciality : "libanaise", address : "ad3", rating : 4.1 \rangle \end{bmatrix}$$

Lors de l'évaluation de $FilterRes$, la condition $restaurant.rating \geq 4$ est évaluée sur tous les éléments de L. Nous traçons son évaluation sur le premier élément $s_1 = restaurant : \langle name : "nom1", speciality : "française", address : "ad1", rating : 4.5 \rangle$:

$s_1 \vDash restaurant.rating \geq 4$ si $restaurant.rating^{s_1} \geq 4^{s_1}$.

Nous avons $restaurant.rating^{s_1} = val\left(\chi_{restaurant.rating}(s_1)\right) = val(rating : 4.5) = 4.5$ (deuxième cas). Nous avons également $4^{s_1} = 4$ (premier cas). Donc $restaurant.rating^{s_1} \geq 4^{s_1}$ est vraie. Ainsi s_1 vérifie la condition de filtrage de $FilterRes$.

3.2.1.3 Unique

Une activité *Unique* prend en entrée une instance V, de type liste, contenant des instances v_i d'un certain type $\hat{t}$. Pour un chemin d'accès $path \in paths(\hat{t})$[21], Elle retourne une liste avec les doublons éliminés.

- Notation : $\mu_{path}(V)$
- Type de l'activité : $\mu : ([\hat{t}] \rightarrow [\hat{t}])$
- Sémantique de l'activité : $\mu_{path}(V) = [r_1, \dots r_k]$ tel que :
  - $(\forall i, r_i \in V) \wedge \left(\nexists r_j, j \neq i \middle| \left(\chi_{path}(r_i)\right).equal\ \left(\chi_{path}(r_j)\right)\right)$ si une opération d'égalité $equal$ est définie sur le type sous-jacent.
  - Sinon $\mu_{path}(V) = V$

Exemple

Soit la liste d'instances ci dessous. Elle indique les villes intermédiaires entre Grenoble et Paris sous format WOEID[22].

$T = woeids$
$: [woeid : 12724717, woeid : 12724717, woeid : 12724717, woeid : 12724728, woeid$
$: 12724728, woeid : 12726958, woeid : 12726958, woeid : 20068148, woeid : 20068141, woeid$
$: 20068141]$

L'expression $uniqueWOEID(T) = \mu_{woeid}(T)$ retourne l'instance suivante :

$woeids : [woeid : 12724717, woeid : 12724728, woeid : 12726958, woeid : 20068148,$
$woeid : 20068141]$

3.2.1.4 Sort

Nous considérons le domaine $order = \{"ascending", "descending"\}$. Une activité *Sort* prend en entrée une instance V, de type liste, contenant des instances v_i d'un certain type $\hat{t}$ et une instance b de type atomique défini sur le domaine $order$. Pour un chemin d'accès $path \in paths(\hat{t})$[23], Elle retourne une liste avec les valeurs v_i ordonnées d'une manière croissante ou décroissante selon la valeur de b.

- Notation : $\phi_{path}(V, b)$
- Type de l'activité : $\phi : ([\hat{t}] \times order \rightarrow [\hat{t}])$
- Sémantique de l'activité : $\phi_{path}(V, b) = [r_1, \dots, r_n]$ tel que $(\forall i, r_i \in V)$ et :
  - $\left(\forall i \leq j < V.length() \in \mathbb{N},\ \text{nous avons}\ r_i.lessOrEqual\ (r_j)\right)$ si $b = "ascending"$ et une opération $lessOrEqual$ est définie sur le type sous-jacent.
  - $\left(\forall i \leq j < V.length() \in \mathbb{N},\ \text{nous avons}\ r_j.lessOrEqual\ (r_i)\right)$ si $b = "descending"$ et une opération $lessOrEqual$ est définie sur le type sous-jacent.
  - Sinon $\phi_{path}(V, b) = V$.

[21] $path$ est un chemin d'accès dont la grammaire est définie dans la sous-section 3.2.1.1.
[22] La présence de doublons est due au fait de la présence de plusieurs étapes de l'itinéraire dans la même ville.
[23] $path$ est un chemin d'accès dont la grammaire est définie dans la sous-section 3.2.1.1.

Exemple

L'expression $SortRes = \phi_{restaurant.rating}(R, "ascending")$[24] retourne la liste suivante :

$$restaurants : \begin{bmatrix} restaurant : \langle name : "nom1", speciality : "française", address : "ad1", rating : 4.5\rangle, \\ restaurant : \langle name : "nom3", speciality : "libanaise", address : "ad3", rating : 4.1\rangle, \\ restaurant : \langle name : "nom2", speciality : "italienne", address : "ad2", rating : 4\rangle \end{bmatrix}$$

3.2.1.5 Append

Une activité *Append* prend en entrée deux instances R et V de type liste contenant des instances de même type $\hat{t}$. Elle retourne une liste qui est le résultat de l'union des deux listes.

- Notation : $R \cup V$
- Type de l'activité : $\cup$: $([\hat{t}]\times[\hat{t}] \rightarrow [\hat{t}])$
- Sémantique de l'activité : $A : [r_1 \dots r_n] \cup A : [v_1 \dots v_m] = A : [r_1 \dots r_n, v_1 \dots v_m]$

3.2.1.6 Concatenate

Une activité *Concatenate* prend en entrée deux instances U et V de type tuple. Elle retourne un tuple ayant les attributs de U et V.

- Notation : $U \oplus V$
- Type de l'activité :
  $\oplus : (A : \langle \widehat{u_1}, \dots, \widehat{u_m}\rangle \times B : \langle \widehat{v_1}, \dots, \widehat{v_n}\rangle \rightarrow A_B : \langle \widehat{u_1}, \dots, \widehat{u_m}, \widehat{v_1}, \dots, \widehat{v_n}\rangle)$
  Il est nécessaire que $name(\widehat{u_i}) \neq name(\widehat{v_j})$ pour chaque i et j.
- Sémantique de l'activité :
  Pour un tuple U de la forme $A : \langle A_1 : u_1, \dots, A_m : u_m\rangle$ et un tuple V de la forme $B : \langle B_1 : v_1, \dots, B_n : v_n\rangle$, nous avons : $U \oplus V = A_B : \langle A_1 : u_1, \dots, A_m : u_m, B_1 : v_1, \dots, B_n : v_n\rangle$.

3.2.2 Activités composites

Les activités que nous venons de présenter permettent de manipuler les instances de données dans les mashups. La coordination de ces activités définit l'ordre de leur exécution. Elle est définie avec des activités composites qui sont de type fonction.

Nous considérons les activités composites suivantes[25] : *Sequence*, *Foreach* et *Parallel*.

3.2.2.1 Sequence

Une activité *Sequence* est définie à partir de deux ou plusieurs activités $act_1, act_2, \dots act_n \in \mathbb{O}$ considérées comme des sous-activités de l'activité *Sequence*.

- Notation : $\gg_{(act_1, act_2, \dots act_n)}$
- Type de l'activité : $\gg$: $(\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_n \rightarrow \hat{t'}_1 \times \hat{t'}_2 \times \dots \times \hat{t'}_l)$
- Sémantique de l'activité :
  - $\gg_{(act_1, act_2, \dots act_n)} (i_1, \dots, i_n) = act_n\left(\dots\left(act_2\left(act_1(i_1, \dots, i_n)\right)\right)\right)$
  - act_1 est de type $(\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_n \rightarrow \hat{t}_{1.1} \times \dots \times \hat{t}_{1.k})$
  - $\forall\, 2 \leq i \leq n$, act_i est de type $(\hat{t}_{(i-1).1} \times \dots \times \hat{t}_{(i-1).k} \rightarrow \hat{t}_{i.1} \times \dots \times \hat{t}_{i.m})$
  - act_n est de type $(\hat{t}_{(n-1).1} \times \dots \times \hat{t}_{(n-1).m} \rightarrow \hat{t'}_1 \times \hat{t'}_2 \times \dots \times \hat{t'}_l)$

---

[24] R est la liste obtenue par filtrage dans la sous-section 3.2.1.2.

[25] Le modèle peut être enrichi avec d'autres activités composites dans le futur selon les besoins qui peuvent apparaître.

Représentation graphique

La Figure 13 donne la représentation graphique d'une activité *Sequence* ayant deux sous-activités.

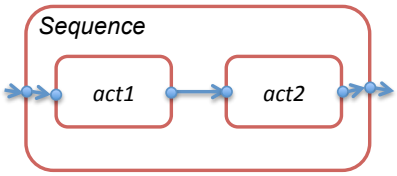

Figure 13 : Représentation graphique d'une activité *Sequence*

Exemple :

Soit le fournisseur de données Google Places qui offre une activité $getRestaurant$ de type $(cityGeo : \langle lat : \mathbb{N}, lng : \mathbb{N}\rangle \rightarrow restaurants : [restaurant])$

Nous définissons une activité *Sequence* $destRestaurants = \gg_{(getRestaurant, FilterRes, SortRes)}$. Elle est de type $\gg : (end_location : \langle lat : \mathbb{N}, lng : \mathbb{N}\rangle \rightarrow restaurants : [restaurant])$. L'activité $getRestaurant$ récupère la liste de restaurants présents près du point géographique défini par le paramètre d'entrée $cityGeo$. Ensuite l'activité $FilterRes$ (définie dans la sous-section 3.2.1.2) sélectionne les restaurants dont la note est supérieure à 4. Enfin l'activité $SortRes$ (définie dans la sous-section 3.2.1.4) trie la liste des restaurants selon la valeur de la note par ordre croissant. La représentation graphique de cette activité est donnée dans la Figure 14.

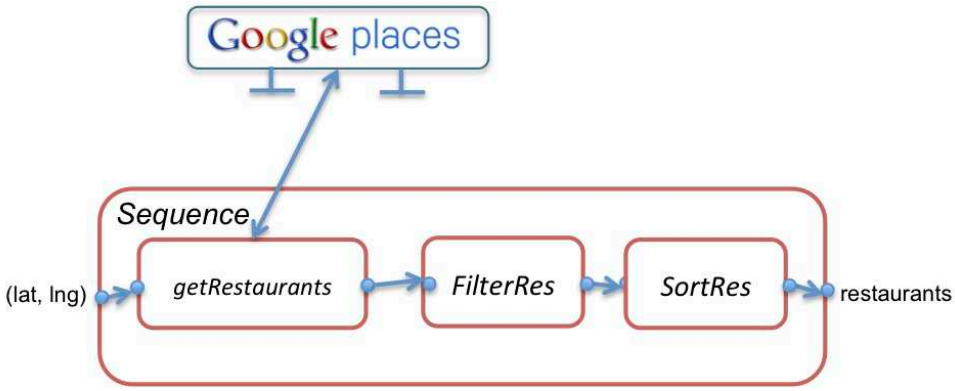

Figure 14 : Représentation graphique d'un exemple d'une activité *Sequence*

3.2.2.2 Foreach

Une activité *Foreach* prend en entrée une instance de type liste. Elle est définie à partir d'une activité $act \in \mathbb{O}$ considérée comme une sous-activité de l'activité *Foreach*.

- Notation : $\infty_{(act)}$
- Type de l'activité : $\infty : ([\hat{t}] \rightarrow [\hat{t}'])$
- Sémantique de l'activité :
  $\infty_{(act)}(T) = [act(T[1]), act(T[2]), \ldots, act(T[n])]$ tel que act est de type $(\hat{t} \rightarrow \hat{t}')$
  L'activité exécute itérativement sa sous-activité sur les éléments de la liste T retourne dans une liste les résultats des exécutions. L'exécution s'arrête avec la dernière exécution de la sous-activité sur le dernier élément de la liste T.

Représentation graphique

La Figure 15 présente une représentation graphique d'une activité *Foreach* et de sa sous-activité. La discontinuité des flèches représente le découpage le l'entrée de *Foreach* en valeurs et leur transmission à la sous-activité.

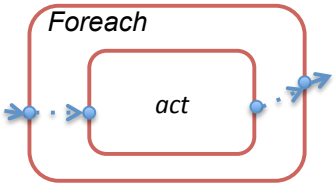

Figure 15 : Représentation graphique d'une activité *Foreach*

Exemple :

Soit le fournisseur de données Yahoo! Weather qui offre une activité $getWeather$ de type $(woeid : \mathbb{N} \rightarrow \widehat{w})$, où $\widehat{w}$ est un type décrivant des données météorologiques d'une certain ville identifiée par son identifiant $woeid$.

Nous définissons une activité *Foreach* $citiesWeather = \infty_{(getWeather)}$. Elle est de type $\infty : (woeids : [woeid : \mathbb{N}] \rightarrow citiesW : [\widehat{w}])$. Elle retourne les données météorologiques des villes passées en entrée. Elle est représentée graphiquement dans la Figure 16.

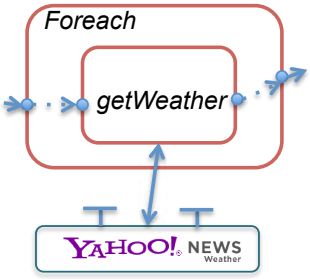

Figure 16 : Représentation graphique d'un exemple d'une activité *Foreach*

3.2.2.3 Parallel

Une activité *Parallel* est définie à partir de deux ou plusieurs activités $act_1, act_2, \dots act_n \in \mathbb{O}$ considérées comme des sous-activités de l'activité *Parallel*.

- Notation : $\|_{(act_1, act_2, \dots act_n)}$
- Type de l'activité : $\|: (\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_p \rightarrow \hat{t}_{1.1} \times \dots \times \hat{t}_{1.d} \times \hat{t}_{2.1} \times \dots \times \hat{t}_{2.f} \times \dots \times \hat{t}_{n.1} \times \dots \times \hat{t}_{n.h})$
- Sémantique de l'activité :
  act_1 est de type $(\hat{t}_i \times \dots \times \hat{t}_k \rightarrow \hat{t}_{1.1} \times \dots \times \hat{t}_{1.d})$ où $1 \leq i \leq k \leq p$
  act_2 est de type $(\hat{t}_g \times \dots \times \hat{t}_l \rightarrow \hat{t}_{2.1} \times \dots \times \hat{t}_{2.f})$ où $1 \leq g \leq l \leq p$
  …
  act_n est de type $(\hat{t}_f \times \dots \times \hat{t}_m \rightarrow \hat{t}_{n.1} \times \dots \times \hat{t}_{n.h})$ où $1 \leq f \leq m \leq p$
  $\|_{(act_1, act_2, \dots act_n)}(i_1, \dots, i_p) = act_1(i_i, \dots, i_k), act_2(i_g, \dots, i_l), \dots, act_n(i_f, \dots, i_m)$

Représentation graphique

La Figure 17 donne la représentation graphique d'une activité *Parallel* et de ses sous-activités.

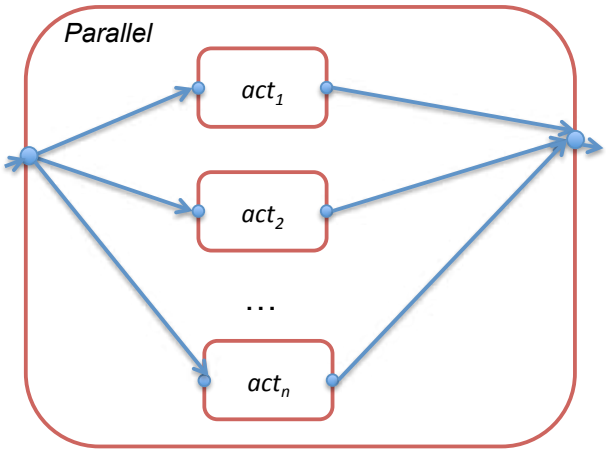

Figure 17 : Représentation graphique d'une activité *Parallel*

3.2.3 Définition du domaine des activités $\mathbb{O}$

Le domaine des activités $\mathbb{O} \subset Functions$ est construit avec les règles suivantes :

- Si a est une activité proposée par un fournisseur de données, alors $a \in \mathbb{O}$.
- Si a est une activité Extract de type $\chi : (\hat{t} \rightarrow \hat{t}')$, alors $a \in \mathbb{O}$.
- Si a est une activité Filter, de type $\sigma : ([\hat{t}] \rightarrow [\hat{t}])$, alors $a \in \mathbb{O}$.
- Si a est une activité Unique de type $\mu : ([\hat{t}] \rightarrow [\hat{t}])$, alors $a \in \mathbb{O}$.
- Si a est une activité Sort de type $\phi : ([\hat{t}] \times order \rightarrow [\hat{t}])$, alors $a \in \mathbb{O}$.
- Si a est une activité Append, de type $\cup : ([\hat{t}] \times [\hat{t}] \rightarrow [\hat{t}])$, alors $a \in \mathbb{O}$.
- Si a est une activité Concatenate, de type $\oplus : (A : \langle \widehat{u_1}, \dots, \widehat{u_m} \rangle \oplus B : \langle \widehat{v_1}, \dots, \widehat{v_n} \rangle \rightarrow A_B : \langle \widehat{u_1}, \dots, \widehat{u_m}, \widehat{v_1}, \dots, \widehat{v_n} \rangle)$, alors $a \in \mathbb{O}$.
- Si $a_1, a_2, \dots, a_n \in \mathbb{O}$, alors l'activité Sequence $\gg_{(a_1, a_2, \dots, a_n)} \in \mathbb{O}$.
- Si $a \in \mathbb{O}$, alors l'activité Foreach $\infty_{(a)} \in \mathbb{O}$. C
- Si $act_1, act_2, \dots act_n \in \mathbb{O}$, alors l'activité Parallel $\|_{(act_1, act_2, \dots act_n)} \in \mathbb{O}$.

Dans ces règles, $\hat{t}$, $\hat{t}'$, $\widehat{u_1}, \dots, \widehat{u_m}$, $\widehat{v_1}, \dots, \widehat{v_n}$ sont des types quelconques qui appartiennent à $\mathbb{T}$.

Le domaine $\mathbb{O}$ est extensible : il peut être enrichi dans le futur selon les besoins qui peuvent apparaître.

3.3 Mashlet

Un mashlet est une entité réutilisable d'un mashup. Il affiche des données sous la forme d'un widget. Ces données sont produites par une activité.

Le type $Mashlet$ est définit comme suit :

$$Mashlet : \langle inputsTypes : \langle \hat{t}_1, \hat{t}_2, \dots, \hat{t}_n \rangle, outputType : \langle \hat{t}_m \rangle, activity : \mathbb{O} \rangle.$$

Une instance de $Mashlet$ définit les types des entrées et des sorties et l'activité qui exécute le traitement de données. Pour chaque instance m de $Mashlet$, $activity$ est une fonction de type $(\hat{t}_1 \times \hat{t}_2 \times \dots \times \hat{t}_n \rightarrow \hat{t}_m)$.

$Opérations(Mashlet) = ops : [run, render]$

- run est une opération de type $(Mashlet \rightarrow o : \emptyset)$.
- $render$ est une opération de type $(Mashlet \rightarrow o : \emptyset)$.

La Figure 18 montre la représentation graphique d'un mashlet avec son activité. L'entrée et la sortie du mashlet sont liées à l'entrée et la sortie de l'activité.

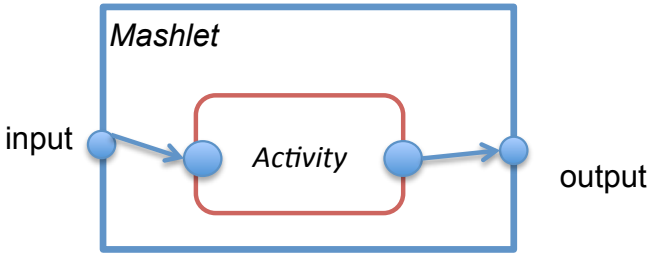

Figure 18 : Représentation graphique d'un mashlet

Exemple

Soit le fournisseur de données Google maps qui offre une activité $getRoute$ de type[26] $(bounds : \langle start : \mathbb{S}, end : \mathbb{S}\rangle \rightarrow routes)$. Dans le mashup ItineraryPlanner, le mashlet map reçoit en entrée une donnée contenant une ville de départ et une ville de destination. L'activité sous-jacente est $getRoutes$ qui récupère l'itinéraire du fournisseur de données Google Maps.

$map = Mashlet : \langle inputsTypes : \langle bounds : \langle start : \mathbb{S}, end : \mathbb{S}\rangle\rangle, outputType : \langle routes\rangle, activity : getRoutes\rangle.$

La représentation graphique de ce mashlet est donnée dans la Figure 19.

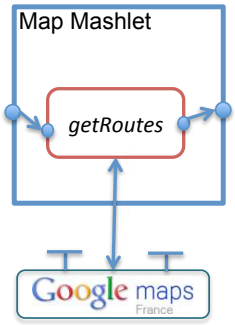

Figure 19 : Représentation graphique du mashlet Map

Dans le mashup ItineraryPlanner, le mashlet $weather$ reçoit en entrée la liste de localisations des villes de passage sur la route. L'activité sous-jacente est l'activité Foreach $citiesWeather$ (définie dans la sous-section 3.2.2.2) qui pour chaque ville, récupère les données météorologiques (activité $getWeather$) depuis le fournisseur de donnée Yahoo! Weather.

$weather = Mashlet$
$: \langle inputsTypes : \langle woeids : [woeid : \mathbb{N}]\rangle, outputType : \langle citiesW : [\widehat{w}]\rangle, activity : citiesWeather\rangle$

La représentation graphique de ce mashlet est donnée dans la Figure 20.

[26] Nous rappelons qu'un type peut être référencé par son nom. Ce qui est le cas de l'utilisation du type $routes$ ici et dont la définition est donnée à la page 3.

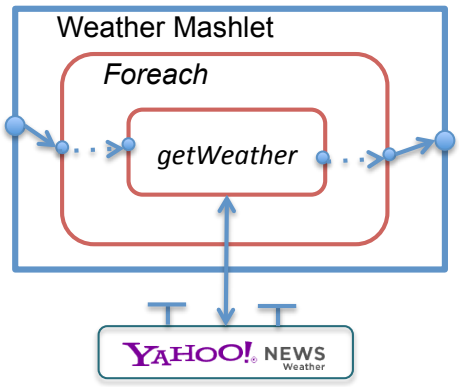

Figure 20 : Représentation graphique du mashlet Weather

3.4 Wiring

Un wiring est une entité du mashup qui décrit le flot de données entre deux mashlets. Dans notre modèle, les wirings ne décrivent pas qu'un simple transfert de données entre mashlets : ils peuvent aussi, intégrer une transformation des données pour les délivrer sous le bon format aux mashlets destinataires. Cette transformation définie avec une activité sous-jacente.

Un wiring est représenté par un type tuple :

$Wiring : \langle inputType : \langle \hat{t} \rangle, outputType : \langle \hat{t}' \rangle, sender : [\![Mashlet]\!], receiver : [\![Mashlet]\!], activity : \mathbb{O} \rangle.$

Une instance de $Wiring$ définit les types des entrées et des sorties, les mashlets envoyeur ($sender$) et receveur ($receiver$) et l'activité qui exécute le traitement de données. Pour chaque instance w de $Wiring$, nous avons :

- L'entrée du wiring est du même type que la sortie de l'instance de $Mashlet$ définie dans l'attribut $sender$:
  $val(w.inputType) = val\big((val(w.sender)).outputType\big)$
- $activity$ est une fonction de type $(\hat{t} \rightarrow \hat{t}')$.

$Opérations(Wiring) = ops : [fetch, run, dispatch]$

- $fetch$ est une opération de type $(Wiring \rightarrow o : \emptyset)$.
- run est une opération de type $(Wiring \rightarrow o : \emptyset)$.
- $dispatch$ est une opération de type $(Wiring \rightarrow o : \emptyset)$.

Le comportement d'un Wiring est décrit comme suit : il (1) prend les données ($fetch$) en entrée du mashlet envoyeur, (2) les manipule (run), en exécutant un processus de transformation de données (défini par l'attribut $activity$), pour (3) envoyer le résultat ($dispatch$) en sortie au mashlet receveur.

La Figure 21 montre la représentation graphique d'un wiring avec son activité. L'entrée et la sortie du wiring sont liées à l'entrée et la sortie de l'activité.

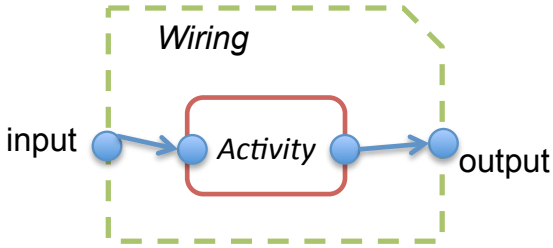

Figure 21 : Représentation graphique d'un wiring

Exemple

Dans le mashup ItineraryPlanner, un wiring est nécessaire pour transmettre les villes de passages du mashlet map au mashlet $weather$. Cependant, dans le mashlet map, une ville est décrite comme une paire de longitude et de latitude. D'un autre côté, le mashlet $weather$ nécessite qu'une ville soit représentée par un identifiant WOEID. Pour cela le wiring doit procéder à une transformation entre les deux formats. La Figure 22 présente la représentation graphique du wiring $Map2Weather$ entre les mashlets map et $weather$. Soit un fournisseur de données qui offre une activité $getWOEID$ de type $(cityGeo : \langle lat : \mathbb{R}, lng : \mathbb{R}\rangle \rightarrow woeid : \mathbb{N})$.

1. Le wiring prend en entrée la sortie du mashlet map : l'itinéraire entre les deux villes de type $routes$. L'activité (définie dans l'attribut $activity$) commence par extraire les villes de passage des étapes de l'itinéraire. Dans la Figure 22, pour des raisons d'espace, nous représentons cette tâche avec une seule activité *ExtractRouteCities* qui est une activité *Sequence*. Elle est composée comme suit :
   a. L'activité $extractSteps$ est une fonction dont la signature est $routes \rightarrow steps$. Le chemin d'accès correspondant est $path = routes.legs.steps$. Elle retourne la liste des étapes.
   b. Pour chaque étape de type $step$. Il faut extraire les coordonnées géographiques du point de départ avec l'activité $extractCity$ dont la signature est $step \rightarrow start_location$. Le chemin d'accès correspondant est $path = step.start_location$.
   c. Une activité *Foreach* permet d'extraire les points géographiques de toutes les villes étapes : $extractCities = \infty_{(extractCity)}$
   d. Une activité *Sequence* permet de coordonner les activités : $extractRouteCities = \gg_{(extractSteps,\ extractCities)}$
2. Pour chaque paire (longitude, latitude) un service est appelé pour récupérer les différentes descriptions géographiques du lieu (activité $getWoeid$). L'identifiant WOEID en est extrait (activité $extractWOEID$). Ces activités sont coordonnées avec des activités *Sequence* et *Foreach* : $woeidsLoop = \infty_{(\gg_{(getWoeid,\ extractWOEID)})}$
3. Ensuite, il faut éliminer les doublons[27] avec l'activité *Unique* $uniqueWOEID$ décrite dans la sous-section 3.2.1.3.
4. Les activités $extractRouteCities$, $woeids$ et $uniqueWOEID$ sont coordonnées avec une activité *Sequence* $map2weatherAct = \gg_{(extractRouteCities,woeidsLoop,uniqueWOEID)}$

En sortie le wiring $Map2Weather$ transmet au mashlet $weather$ la liste de localisations des villes décrites sous le format WOEID. Il est modélisé ainsi :

$$map2Weather = Wiring : \langle inputType : \langle routes\rangle, outputType : \langle woeids : [woeid : \mathbb{N}]\rangle, sender : map, receiver : weather, activity : map2weatherAct\rangle.$$

[27] La présence de doublons est due au fait de la présence de plusieurs étapes de l'itinéraire dans la même ville.

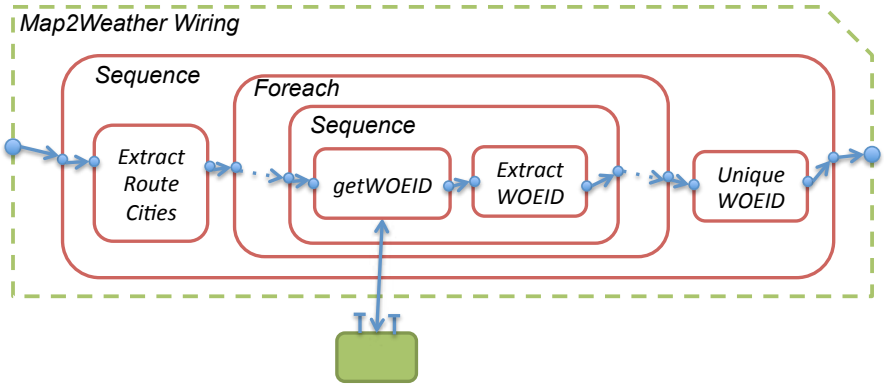

Figure 22 : Représentation graphique du wiring entre les mashlets Map et Weather

3.5 Mashup

Un mashup est un ensemble de mashlets et de wirings. Les wirings définissent le flot de données entre les mashlets du mashup. Un mashup définit les types des ses paramètres d'entrées $inputsTypes$, l'ensemble de ses mashlets $mashlets$ et de ses wirings $wirings$. Il définit également un ensemble de propriétés comme la fréquence d'exécution du mashup, la définition des durées de vie (fixées en nombre d'heures, etc.) des données caractérisées par les noms de leurs types. Il est représenté par un type tuple[28] :

$$Mashup : \langle inputsTypes : \langle \hat{t}_1, \hat{t}_2, \dots, \hat{t}_n \rangle, mashlets : [Mashlet], \quad wirings : [Wiring], \\ properties : \langle frequency : \mathbb{N}, \quad ttls : [dataTTLDef : \langle dataTypeName : \mathbb{A}, ttl : \mathbb{N} \rangle] \rangle \rangle$$

Une instance de $Mashup$ définit les types des entrées du mashup, les mashlets et les wirings qui constituent le mashup.

$Opérations(Mashup) = ops : [dispatchInputs, run]$

run et $dispatchInputs$ sont des opérations de type est $(Mashup \rightarrow o : \emptyset)$.

Le mashup ItineraryPlanner présenté dans le Chapitre 1 est défini de la façon suivante :

$ItineraryPlanner = Mashup : \langle\ inputsTypes : \langle start : \mathbb{S}, end : \mathbb{S} \rangle, mashlets : [map,\ weather,\ restaurants],\ wirings : [map2Weather, map2Restaurant], properties : \langle frequency : 4, ttls : [dataTTLDef : \langle dataTypeName : "route", ttl : 720 \rangle, dataTTLDef : \langle dataTypeName : "weather", ttl : 6 \rangle, dataTTLDef : \langle dataTypeName : "woeid", ttl : 720 \rangle,] \rangle \rangle$.

Où le mashlet $Restaurants$ et le wiring $Map2Restaurant$ sont définis d'une manière similaire que les autres mashlets et wirings définis dans les sections précédentes. Ils sont explicités dans la Figure 23. Celle ci montre la représentation graphique de tous les composants du mashup $ItineraryPlanner$. Les mashlets map et $weather$ et le wiring $map2Weather$ furent exposés dans les sections précédentes de ce chapitre. Dans cette figure, nous montrons aussi le mashlet $restaurants$ qui fournit la liste des restaurants dans la ville de destination. L'attribut $activity$ correspondant est l'activité $destRestaurants$ présentée dans la sous-section 3.2.2.1. Un wiring $map2Restaurant$ est nécessaire pour transférer la paire (longitude, latitude) de la ville de destination depuis le mashlet map vers le mashlet $restaurants$ (via une activité Extract).

[28] Nous rappelons que $\mathbb{A}$ est le domaine des noms de types défini dans la sous-section 3.1.1

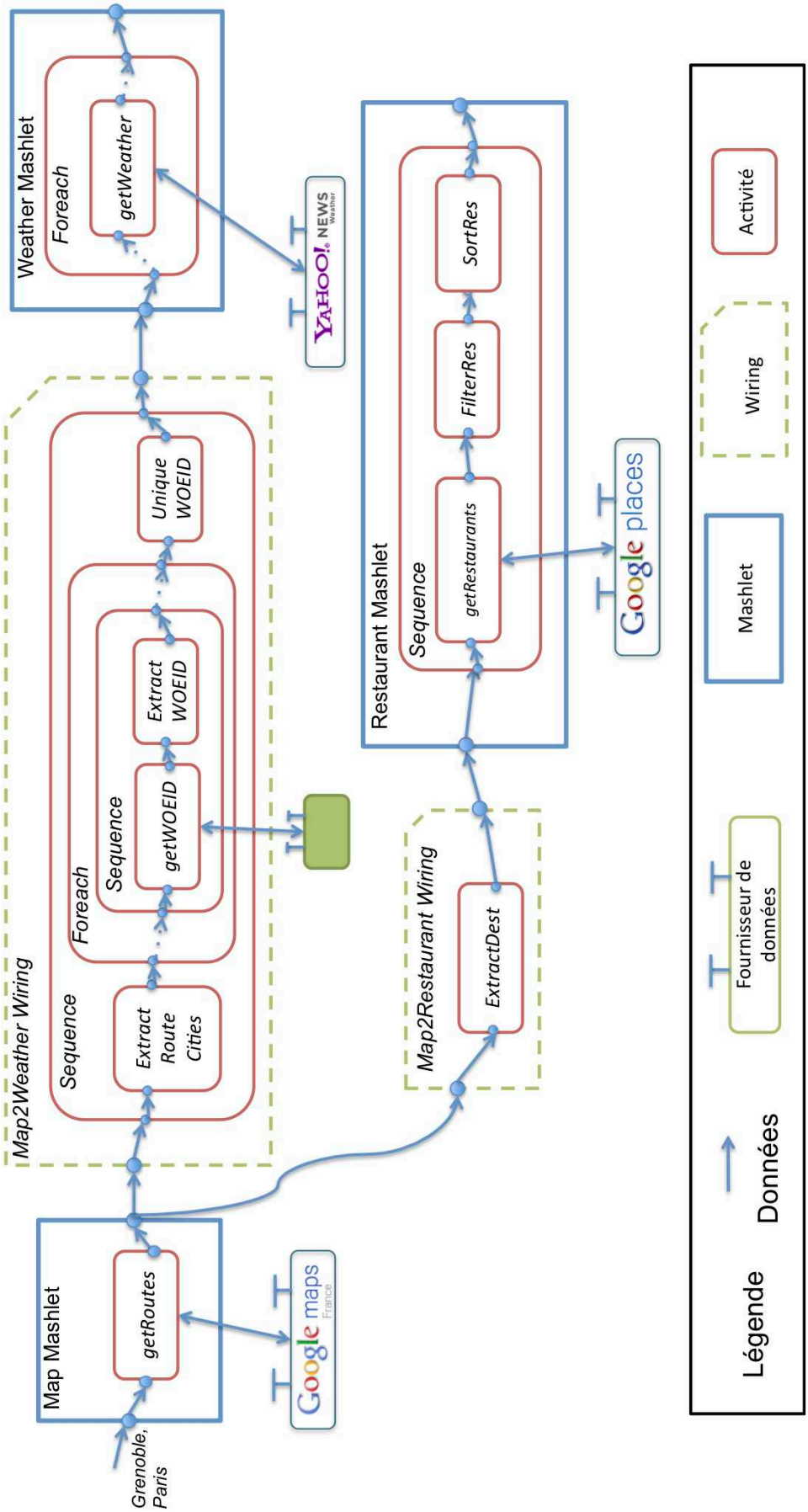

Figure 23 : Représentation graphique du mashup ItineraryPlanner

3.6 Conclusion

Ce chapitre a présenté les concepts de notre modèle de mashup. Nous avons adopté un modèle de données basé sur les valeurs complexes. Des activités basiques décrivent la manipulation des données de mashups (le filtrage, la projection, l'extraction, l'élimination de doublons, le tri, l'union et l'annexe). Elle sont coordonnées par des activités composites (*Sequence*, *Foreach*, *Parallel*). Les activités sont utilisées par des mashlet et des wirings. Les mashlets affichent des données produites par des activités tandis que les wirings coordonnent l'exécution des mashlets. Dans notre modèle, les wirings ne décrivent pas qu'un transfert de données entre mashlets. Ils peuvent aussi, décrire le traitement des données pour les délivrer sous le bon format aux mashlets destinataires. Ces concepts décrivent des entités qui forment des mashups. Chacun de ces concepts était illustré par une représentation graphique qui sert dans la description des activités et de la coordination des mashlets. Les composants du mashup ItineraryPlanner ont été définis selon les concepts du modèle présenté et ils ont été illustrés selon la représentation graphique proposée.

Chapitre 4

EXECUTION DE MASHUPS AVEC DISPONIBILITE DE DONNEES

Ce chapitre présente le processus d'exécution de mashups avec disponibilité de données. L'exécution d'un mashup repose sur la disponibilité de données. Celle ci peut être réduite ou altérée à cause d'une panne au niveau du fournisseur de données ou d'une restriction de nombre de requêtes imposée par ce dernier. Ceci entraîne le dysfonctionnement du mashup. Nous avons constaté dans le Chapitre 2 que les techniques actuelles de disponibilité de données ne sont pas adaptées à des applications comme les mashups. Nous avons constaté, également, que parmi le peu de travaux qui se sont intéressés à la disponibilité des données des mashups, le problème de la fraicheur des données disponibles n'est pas abordé. Pour cela nous proposons un processus pour assurer la disponibilité de données. Ce processus permet également (1) de définir les données dont la disponibilité n'est plus à assurer et (2) d'assurer la fraicheur de données rendues disponibles.

Le scénario ItineraryPlanner est utilisé comme exemple pour illustrer le processus d'exécution de mashups. Nous rappelons qu'il affiche l'itinéraire entre deux villes (mashlet map), les données météorologiques dans les villes de passage (mashlet $weather$) et une liste de restaurants situés dans la ville d'arrivée (mashlet $restaurants$). Les wirings $map2Weather$ et $map2Restaurants$ définissent le flot de données entre les mashlets. Des activités définissent le traitement des données dans les mashlets et les wirings.

Le chapitre est organisé comme suit : la section 4.1 présente les principaux éléments de l'exécution d'un mashup. Ensuite, la section 4.2 décrit notre proposition pour assurer la disponibilité de données lors de l'exécution des mashups. Cette solution est à base de fonctions que nous introduisons d'une façon orthogonale au processus d'exécution de mashups. Elles permettent d'améliorer la disponibilité des données de mashups et d'assurer leur fraicheur. Enfin, la section 4.3 conclut ce chapitre.

4.1 Exécution d'un mashup

Un mashup est vu comme une instance du modèle décrit dans le Chapitre 3. L'exécution d'un mashup (cf. Figure 24) déclenche les exécutions de mashlets et de wirings sous-jacents.

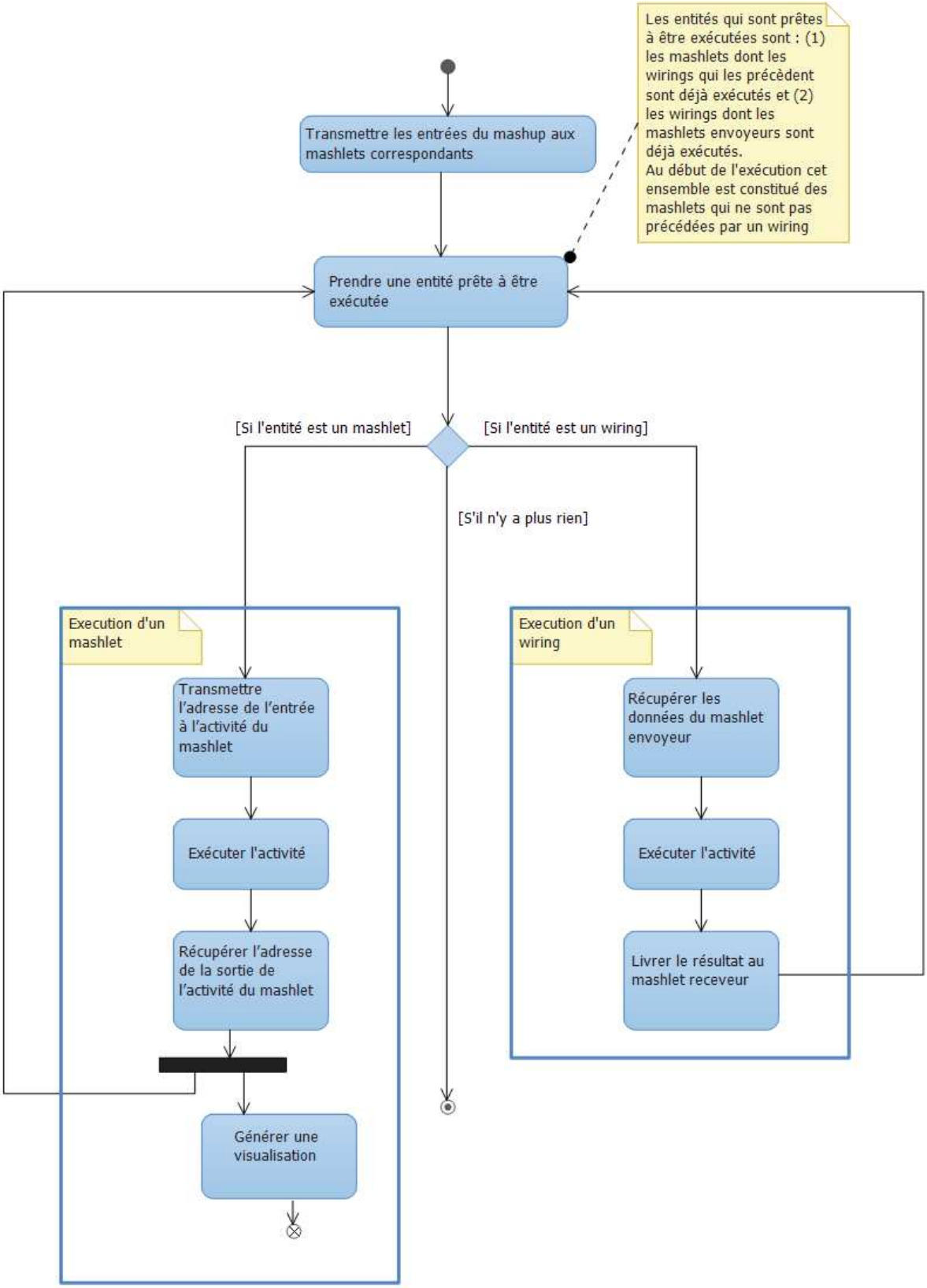

Figure 24 : Diagramme d'activité UML de l'exécution d'un mashup

Ces exécutions partagent leurs données à travers un espace mémoire (privée au mashup). Celui ci est partagé en lecture et en écriture. Ainsi les mashlets, les wirings et les activités d'un mashup partagent leurs données par des lectures et des écritures concurrentes. Le contenu de cet espace est supprimé, une fois l'exécution du mashup achevée.

L'ordre des exécutions des mashlets et des wirings est représenté à travers un graphe dirigé qui constitue le plan d'exécution du mashup appelé graphe de mashup. Les nœuds de ce graphe correspondent aux wirings et mashlets du mashup.

Un graphe de mashup est un quadruple $(M, W, B, start)$, où :

- M est un ensemble de nœuds mashlets.
- W est un ensemble de nœuds wirings.
- $B \subset M \times W \cup W \times M$ est un ensemble d'arcs construits ainsi :
  $\forall\, w \in W, \exists\, m_1, m_2 \in M$ tel que $(m_1, w), (w, m_2) \in B$.
- $start \subseteq M$ est un ensemble de nœuds de départ.

Par exemple, le graphe du mashup ItineraryPlanner est donné dans la Figure 25.

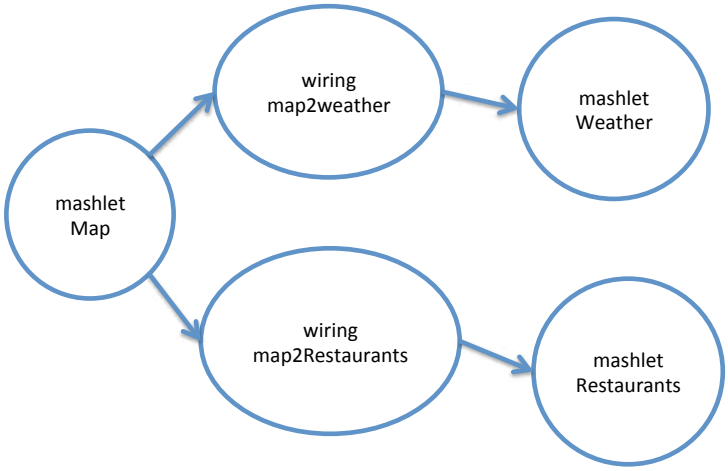

Figure 25 : Graphe du mashup ItineraryPlanner

L'exécution du mashup utilise le graphe pour réaliser l'exécution des mashlets et des wirings le composant, par exécution des activités correspondantes (voir sous-sections 4.1.2 et 4.1.3). Le graphe est parcouru de manière classique : un nœud est visité si tous ses prédécesseurs ont déjà été visités. La Figure 26 donne une vision globale de l'exécution du mashup ItineraryPlanner. Les numéros, dans le graphe, indiquent un ordre possible, des exécutions des nœuds, défini lors du parcours du graphe. Chaque exécution d'un nœud (mashlet et wiring) du graphe déclenche l'exécution de l'activité correspondante (cf. sous-sections 4.1.1 à 4.1.3).

Les sections suivantes décrivent les processus d'exécution de mashlets et de wirings. Comme ces deux processus exécutent tous les deux des activités, nous commençons par préciser l'exécution de ces activités.

4.1.1 Exécution d'une activité

Une activité est représentée sous la forme d'un arbre. Les feuilles d'un arbre d'activités correspondent à des activités basiques, alors que les nœuds intermédiaires correspondent à des activités composites (cf. le modèle de mashups Chapitre 3).

Un arbre d'activité est définie comme suit :

- Soit $\mathfrak{B}$ l'ensemble des activités basiques et $\mathfrak{C}$ celui des activités composites
- Un arbre est constitué :
  - Soit d'une feuille $f \in \mathfrak{B}$
  - Soit d'une racine $r \in \mathfrak{C}$ et d'un ou plusieurs arbres disjoints $s_1, \dots, s_n$ appelés arbres fils. On le note $(r, s_1, \dots, s_n)$.

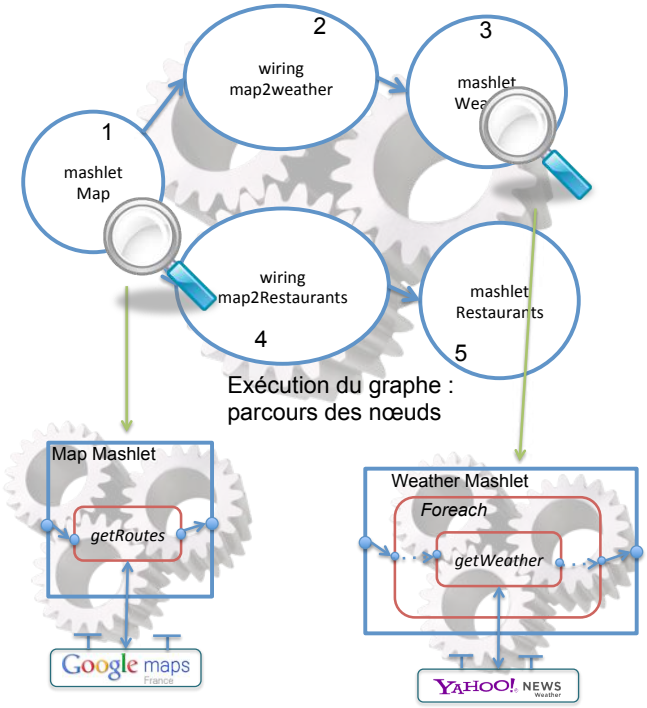

Figure 26 : Exécution du mashup ItineraryPlanner

Par exemple, l'arbre de l'activité du wiring $Map2Weather$ (cf. section 3.4) est représenté dans la Figure 27. Cette activité extrait (séquence d'activités *Extract*) les villes de passages sous format (longitude, latitude). Ensuite pour chaque ville (boucle *Foreach*) un service est appelé (via le nœud *Retrieve*) pour procéder à la conversion en format WOEID. Ensuite, la valeur entière est extraite (*Extract*). Enfin les doublons sont éliminés (*Unique*).

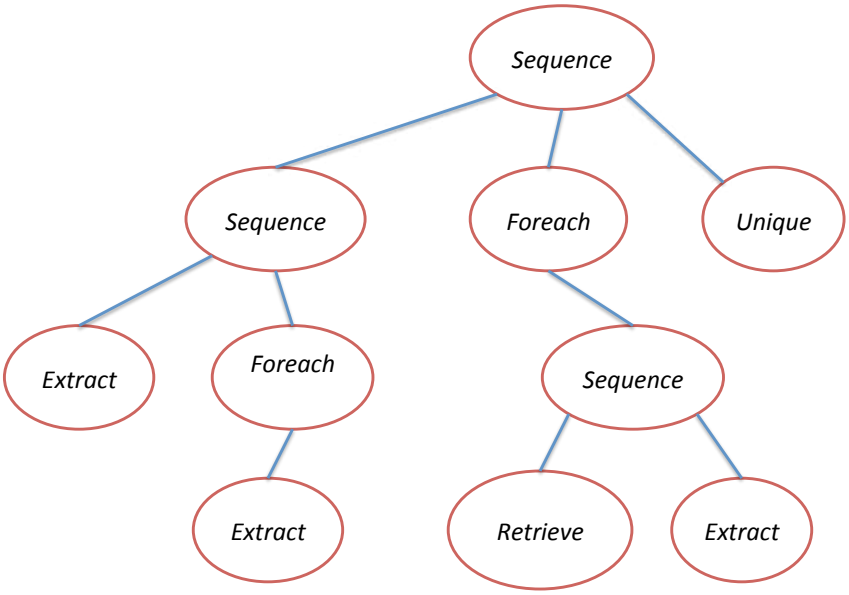

Figure 27 : Arbre de l'activité du wiring $Map2Weather$

L'exécution de l'activité consiste à parcourir l'arbre selon une stratégie en profondeur préfixe [96]. C'est un parcours récursif qui lors de la visite d'un nœud *n*, continue la visite en parcourant les sous-arbres de gauche à droite. L'exécution d'activités est une phase commune aux processus d'exécutions de mashlets et de wirings. La Figure 28 présente le processus d'exécution du mashup ItineraryPlanner tout en prenant en compte la représentation des arbres d'activités.

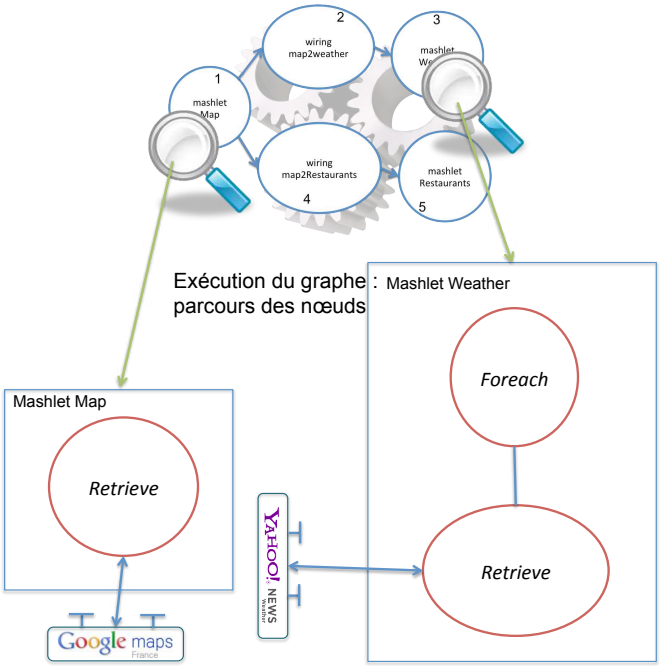

Figure 28 : Exécution du mashup ItineraryPlanner

Chaque nœud possède son propre tampon. Celui ci stocke les adresses des valeurs entrée/sortie de l'activité sous-jacentes. Ces valeurs sont stockées dans l'espace mémoire partagé.

Dans la suite de cette sous-section, nous allons décrire le nœud d'une activité basique et à titre d'exemple celui de l'arbre de l'activité composite *Sequence*.

4.1.1.1 Activité basique

Une activité basique est représentée par un nœud feuille (Figure 29) dans un arbre d'activités. Le nœud contient les paramètres de l'activité sous-jacente, par exemple : condition de filtrage, expressions de chemins d'accès.

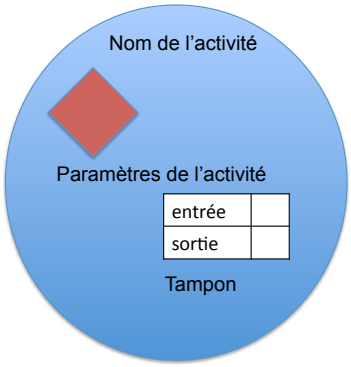

Figure 29 : Composition d'un nœud d'une activité basique

L'exécution d'une activité basique (cf. Figure 30) commence par la lecture de la valeur de l'entrée de l'activité. Celle ci est écrite dans la mémoire partagée et son adresse est indiquée dans

le tampon de l'activité. La deuxième phase correspond à l'exécution de l'opération sous-jacente : filtrage extraction, élimination de doublons… Ces opérations sont proposées dans la plupart des langages de d'interrogation de données. Enfin le résultat de l'exécution est écrit dans la mémoire partagée à l'adresse indiqué dans le tampon de l'activité.

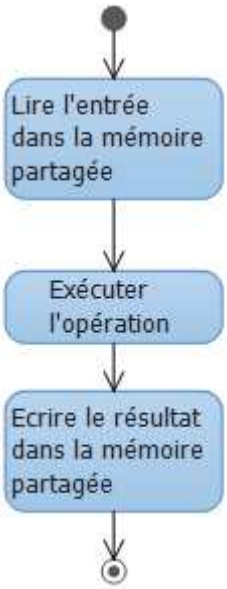

Figure 30 : Diagramme d'activité UML du processus d'exécution d'une activité basique

Cas des activités proposées par des fournisseurs de données

Nous introduisons le nœud de type *Retrieve* (cf. le diagramme de communication de la Figure 31) pour lancer l'exécution une activité proposée par un fournisseur de données et indiquée dans les paramètres du nœud.

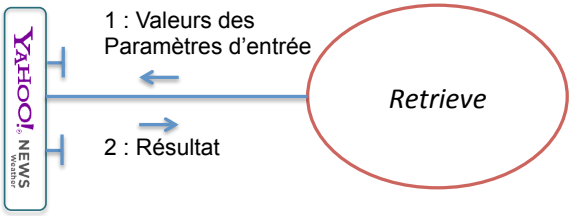

Figure 31 : Diagramme de communication entre le nœud *Retrieve* et le fournisseur de données

L'exécution d'un nœud *Retrieve* envoie les valeurs des paramètres d'entrée (écrites dans le champ « entrée ») d'une activité proposée par un fournisseur de données et demande son exécution. Le résultat reçu est stocké dans l'espace mémoire partagé. L'adresse de ce résultat est indiquée dans le champ « sortie » du tampon du nœud.

4.1.1.2 Activité *Sequence*

Pour une activité *Sequence* $\gg_{(act_1, act_2, \ldots act_n)}$, l'arbre de nœuds correspondant est présenté dans la Figure 32, où « Nœud 1 », « Nœud 2 », … et « Nœud n » sont les nœuds représentant les arbres correspondants aux sous-activités $act_1, act_2, \ldots$, et act_n. Les nœuds échangent les adresses des données stockées dans la mémoire partagée.

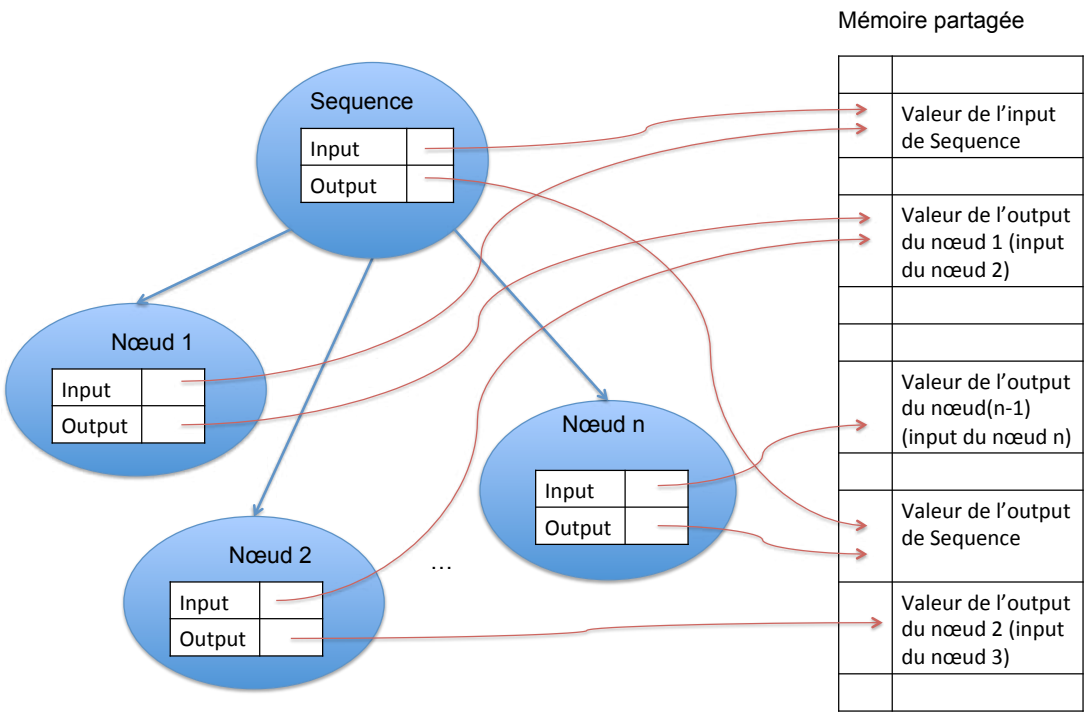

Figure 32 : Arbre de nœuds d'une activité Sequence

L'exécution d'une activité *Sequence* (cf. Figure 33) déclenche l'exécution de ses sous-activités d'une façon séquentielle. Pour chaque sous-activité, elle (1) commence par écrire l'adresse de l'entrée dans le tampon. Pour la première sous-activité, l'entrée correspond à celle de l'activité *Sequence* alors que pour chacune des autres sous-activités, elle correspond à la sortie de la sous-activité précédente. Ensuite elle (2) déclenche l'exécution de la sous-activité. Enfin (3), elle récupère l'adresse de la sortie pour l'écrire dans le champ « entrée » du tampon de la sous-activité suivante. La valeur du champ « sortie » du tampon de la dernière sous-activité est écrite dans le champ « sortie » du tampon du nœud parent.

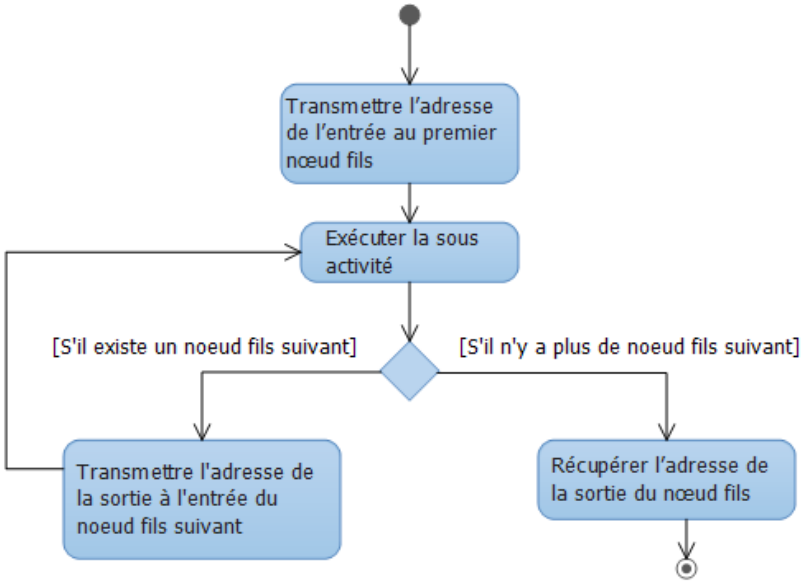

Figure 33 : diagramme d'activité UML du processus d'exécution d'une activité *Sequence*

4.1.2 Exécution d'un mashlet

La Figure 34 présente les phases du processus d'exécution d'un mashlet et montre qu'il déclenche principalement l'exécution de l'activité sous-jacente et la génération d'une visualisation définissant l'affichage graphique de la sortie du mashlet.

Chaque mashlet possède son propre tampon. Celui ci stocke les adresses des valeurs de ses entrée/sortie. Ces valeurs sont stockées dans l'espace mémoire du processus exécutant le mashup. Les valeurs des entrées de certains mashlets sont connues dès le début de l'exécution du mashup comme les villes de départ et d'arrivée dans le mashlet map du le mashup ItineraryPlanner. Les adresses de ces valeurs sont écrites dans le tampon du mashlet avant le parcours du graphe. C'est le processus exécutant le mashup qui s'occupe de cette tâche. Les valeurs des entrées d'autres mashlets sont connues au cours de l'exécution du mashup (parcours du graphe) et transférées par des wirings (voir sous-section 4.1.3) : par exemple, la liste de villes de passages comme entrée du mashlet $weather$ obtenue via le wiring $map2weather$.

Les premières phases du processus d'exécution d'un mashlet correspondent au transfert de l'adresse de l'entrée du mashlet à son activité sous-jacente ainsi qu'à l'exécution de cette activité d'activités (cf. sous-section 4.1.1). Ensuite, l'adresse de la sortie de l'activité est écrite dans le tampon du mashlet. Cette adresse peut être lue par des wirings afin qu'ils accèdent à la donnée, la manipuler, et la transférer à d'autres mashlets : par exemple les données de l'itinéraire du mashlet map sont manipulées par le wiring $map2weather$ afin de transférer la liste de villes de passage au mashlet $weather$. Enfin, la dernière phase consiste en la génération d'une visualisation graphique de ces données. Cette visualisation est affichée sur le terminal de l'utilisateur sous la forme d'un widget.

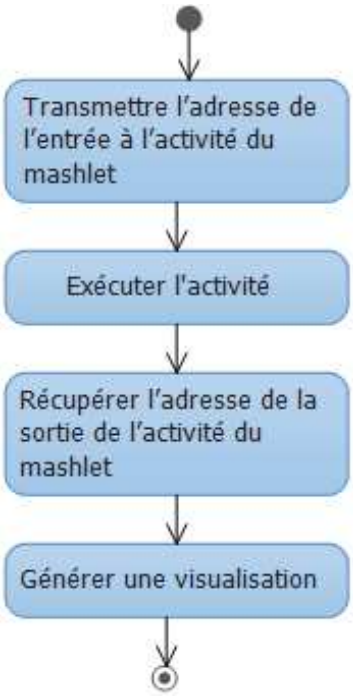

Figure 34 : Diagramme d'activité UML de l'exécution d'un mashlet

4.1.3 Exécution d'un wiring

Le processus d'exécution d'un wiring est constitué de trois phases (cf. Figure 35). La première phase consiste en la récupération de l'adresse de la valeur de la sortie du mashlet envoyeur : par exemple le wiring $map2weather$ récupère l'adresse des les données de l'itinéraire

depuis le tampon du mashlet map. La deuxième phase correspond à l'exécution de l'activité correspondante. Enfin, lors de la phase de Livraison, les données produites à la fin de la phase précédente sont livrées au mashlet receveur (écriture de l'adresse dans le tampon du mashlet) : par exemple le wiring $map2weather$ écrit l'adresse de la liste de villes de passage dans le tampon du mashlet $weather$.

Figure 35 : Diagramme d'activité UML de l'exécution d'un wiring

4.2 Disponibilité des données dans les mashups

Nous proposons une approche pour assurer la disponibilité de données récupérées auprès des fournisseurs de données. Ces données sont stockées dans une structure que nous nommons $Store$. Nous introduisons dans l'arbre d'activités une nouvelle famille de nœuds : *Retrieve++*. A la différence d'un nœud *Retrieve*, un nœud *Retrieve++* vérifie la présence des données recherchées dans le $Store$ avant de lancer l'exécution de l'activité sous-jacente proposée par un fournisseur de données. Le processus proposé permet de rendre, disponibles, les données récupérées. Ce processus est illustré dans la Figure 36.

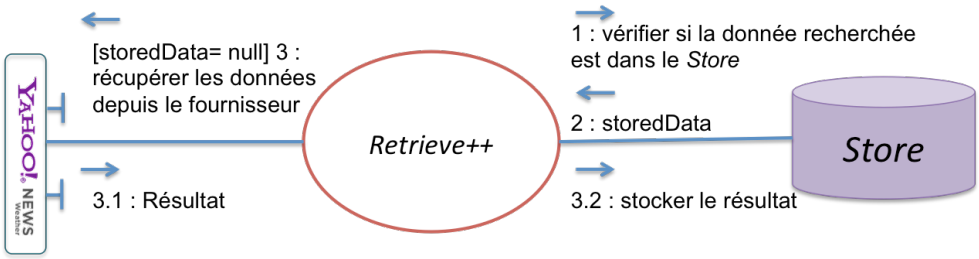

Figure 36 : Diagramme de communication UML entre le nœud *Retrieve++*, le fournisseur de données et le $Store$

La Figure 37 qui donne une vision globale de la phase d'exécution du mashup ItineraryPlanner avec intégration du processus de disponibilité de donnée.

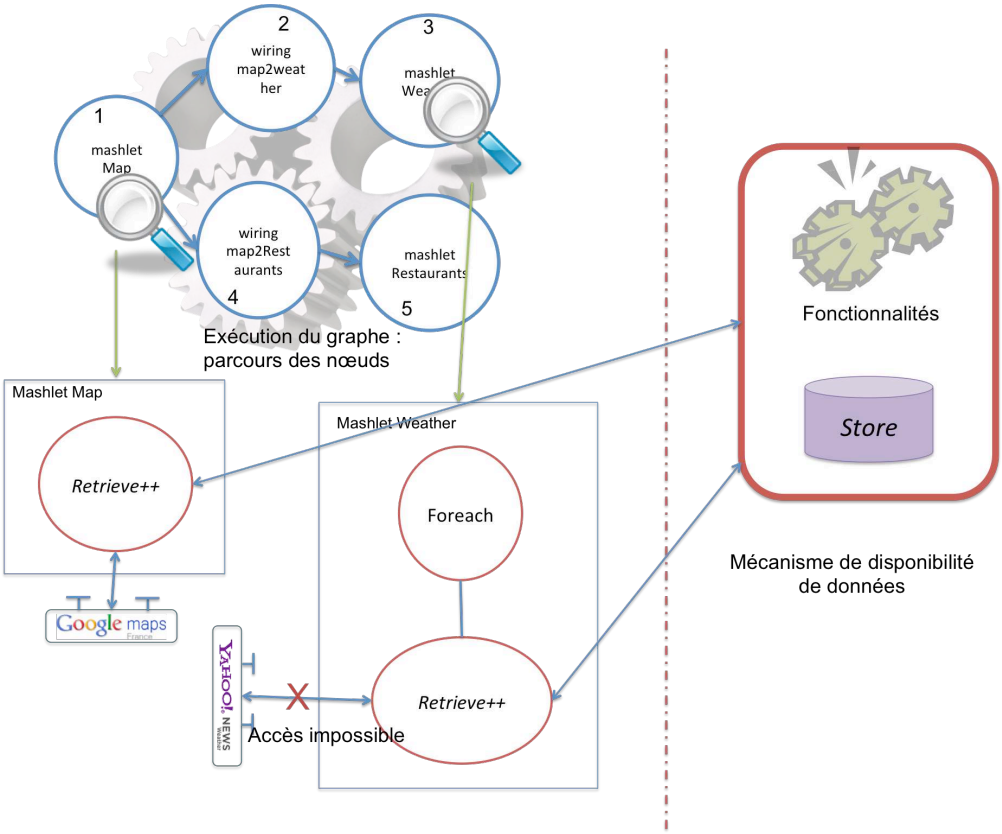

Figure 37 : Exécution d'un mashup avec disponibilité de données

Dans cette section, nous présentons un diagramme d'activité décrivant le processus d'exécution d'un nœud *Retrieve++*. Ce diagramme est présenté dans plusieurs figures (Figure 39, Figure 40 et Figure 42). Il est dans un premier temps minimaliste. Ensuite, il sera enrichi tout au long de cette section lors de la présentation des fonctionnalités du processus de disponibilité de données. Il s'intègre au diagramme de la Figure 24, en faisant partie du processus de l'exécution des activités.

Lorsque le $Store$ devient saturé (capacité maximale atteinte), une partie des données est supprimée pour libérer de l'espace et pouvoir stocker des nouvelles données. Les données des mashups sont dynamiques et ont une durée de vie limitée. Pour cela ces données doivent être rafraichies en anticipant les nouvelles demandes de récupération lors d'une exécution ultérieure d'un mashup.

Ainsi, notre processus de disponibilité est basé sur des fonctions de gestion, de remplacement et de rafraichissement de données du $Store$ (sous-sections 4.2.2, 4.2.3 et 4.2.4). Celui ci est administré par un administrateur qui doit faire certains choix au niveau des fonctions du $Store$. Ces choix seront explicités au cours des sous-sections qui suivent. Ces fonctions sont orthogonales par rapport au processus d'exécution de mashups réalisé par un moteur de mashups. La section 4.2.5 récapitule le processus d'exécution d'un nœud *Retrieve++* avec exploitation du $Store$ et des fonctions associées.

Comme ces fonctions opèrent sur des données rendues disponibles, nous commençons par préciser l'organisation de ces données.

4.2.1 Organisation de données du $Store$

Les données sont organisées, au sein du $Store$, sous la forme d'items. Un item est un tuple qui possède les attributs « id » et « object ». Il est identifié avec la valeur de son attribut « id ». Dans le contexte des mashups, l'attribut « id » correspond à une activité et les valeurs de ses paramètres d'entrée (sous la forme d'une liste de valeurs) alors que l'attribut « objet » correspond à la valeur du paramètre de sortie produite par l'exécution de l'activité.

Un item maintient, de plus, d'autres informations nécessaires pour assurer les fonctions comme le remplacement et le rafraichissement des données. Il est représenté par un tuple :

$Item : \langle id : \langle activity : \mathbb{O}, inputs : [input : Data]\rangle, object : Data, ttl : \mathbb{N}, accessNb : \mathbb{N}, birthDate : Date, expireOn : Date, lastAccess : Date, latency : \mathbb{N}\rangle$.

où :

- $Date$[29] est une définition de type définie comme suit :
  $\langle minute : \{1,2, \dots 60\}, hour : \{1,2, \dots 24\}, month : \{1,2, \dots 12\}, day : \{1,2, \dots 31\}, year : \mathbb{N}\rangle$.
- **accessNb** de type entier, il indique le nombre d'accès à item.
- **ttl** de type entier, il indique la durée de vie de l'item (en nombre d'heures).
- **birthDate** indique la date de création de l'item.
- **expireOn** indique la date d'expiration de l'item.
- **lastAccess** indique la date du dernier accès à l'item.
- **latency** indique le temps mis pour récupérée les données depuis le fournisseur.

$Operations(Item) = ops : [refresh, isExpired, adjustTTL]$ où :

- Pour une instance i de type $Item$, l'opération $refresh$ exécute l'opération $i.id.activity$ avec les valeurs des paramètres indiquées par $i.id.inputs$. Elle écrit le résultat de l'opération dans $i.object$. Ensuite elle retourne un booléen indiquant le bon déroulement de l'opération.
  - Type de $refresh$: $(Item \rightarrow \mathbb{B})$
  - Sémantique : l'appel de $i.refresh()$ exécute l'instruction suivante[30] :
    $i.object = invoke(i.id.activity, i.id.inputs)$
- Pour une instance i de type $Item$, l'opération $isExpired$ vérifie si la donnée enregistrée dans i est encore fraiche.
  - Type de $isExpired$: $(Item \rightarrow \mathbb{B})$
  - Sémantique : l'appel de $i.isExpired()$ retourne $i.expireOn < now()$
- Pour une instance i de type $Item$, l'opération $adjustTTL$ permet de modifier la valeur de l'attribut « ttl » de i et de lui affecter une valeur passée en entrée. La fonction retourne un booléen indiquant le bon déroulement du processus d'ajustement.
  - Type de $adjustTTL$: $(Item \times newttl : \mathbb{N} \rightarrow \mathbb{B})$
  - Sémantique : l'appel de $i.adjustTTL(ttlVal)$ exécute l'instruction suivante : $i.ttl = ttlVal$.
    où $ttlVal$ est une instance du type du paramètre d'entrée d'$adjustTTL$.

La durée de vie d'un item peut être modifiée au cours du temps. Ce point sera expliqué dans la sous-section 4.2.4.2 lors de la description du processus de rafraichissement des items.

---

[29] Nous considérons la fonction $now()$ qui retourne une instance de $Date$ correspondante à l'instant courant.

[30] Nous rappelons que $invoke$ est une opération définie sur tous les types fonction pour réaliser l'exécution de leurs instances (cf. sous-section 3.1.4)

Exemple

Nous considérons $routeGPItem$ une instance de $Item$. Elle correspond à la donnée $RouteGP$ (cf. sous-section 3.1.2) et récupérée avec l'activité $getRoutes$ définie dans la section 3.3. Elle est définie comme suit :

$$routeGPItem = Item : \langle id : \langle activity : getRoutes, inputs : [input : \text{Grenoble}, input : \text{Paris}]\rangle, object : RouteGP, \quad ttl : 720, \quad accessNb : 8, birthDate : \langle minute : 05, hour : 15\ month : 11, day : 1, year : 2012\rangle, expireOn : \langle minute : 05, hour : 15\ month : 12, day : 1, year : 2012\rangle, lastAccess : \langle minute : 30, hour : 7\ month : 11, day : 10, year : 2012\rangle, latency : 1000\rangle$$

La Figure 38 montre la représentation graphique de $Store$ contenant des items pour différents mashups. Le premier ItineraryPlanner et d'autres mashups à la iGoogle contenant des mashlets dont certains affichent la météo de la ville de résidence des utilisateurs. Les différents mashlets météo partagent leurs items.

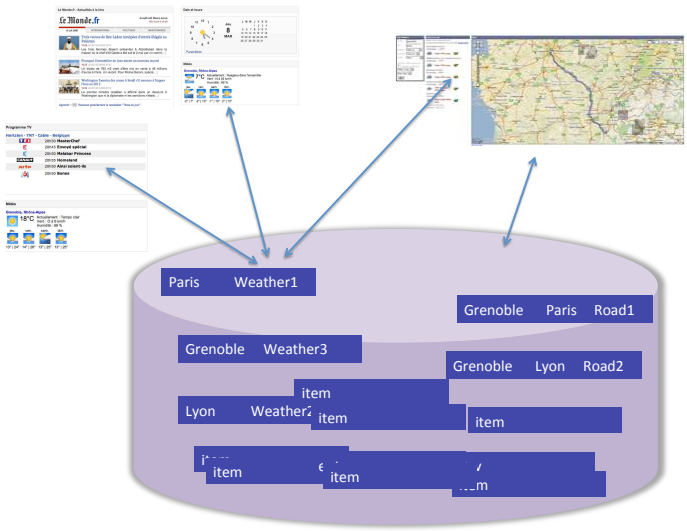

Figure 38 : Représentation du $Store$

4.2.2 Gestion du $Store$

La gestion du $Store$ se fait par des fonctions qui permettent d'ajouter, de supprimer et de rechercher des items. Elles sont définies comme suit :

4.2.2.1 Ajout d'un item

Nous définissons la fonction $bind$. Elle crée une instance d'$Item$ dont les valeurs des attributs « id » « objet » et « ttl » sont passées en paramètre. La valeur de l'attribut « accessNb » est initialisée à 1. Les valeurs des attributs « birthDate » est « lastAccess » sont initialisées à la date courante. Et la valeur de l'attribut « expireOn » est calculée à partir de la date courante et la valeur de l'attribut « ttl ». Ensuite l'instance créée est ajoutée au $Store$. La fonction retourne un booléen indiquant le bon/mauvais déroulement du processus.

- Type de $bind$:
  $(id : \langle activity : \mathbb{O}, inputs : [input : Data]\rangle, \quad object : Data, ttl : \mathbb{N} \rightarrow \mathbb{B})$
- Sémantique :
  Pour $bind(idVal, objVal, ttlVal)$:
  - $newItem = Item$:
    $\langle id : idVal, object : objVal, ttl : ttlVal, accessNb : 1, birthDate : now(), expireOn : now() + ttlVal, lastAccess : now()\rangle$

- $Store = Store + newItem$
- $idVal, objVal, ttlVal$ sont respectivement des instances de type des paramètres d'entrée de $bind : id, object, ttl$.

4.2.2.2 Suppression d'un item

Nous définissons la fonction $unbind$. Elle prend en entrée la valeur de l'attribut « id » d'une instance i d'$Item$ présente dans $Store$. La fonction permet de supprimer i du $Store$. Elle retourne un booléen indiquant le bon déroulement du processus de suppression.

- Type de $unbind$:
  $(id : \langle activity : \mathbb{O}, inputs : [input : Data]\rangle \rightarrow \mathbb{B})$
- Sémantique :
  Pour $unbind(idVal)$ où $idVal$ est une instance du type du paramètre d'entrée de $unbind : id$.
  $Store = Store - i \mid i.id = idVal$

4.2.2.3 Recherche d'un item

Nous définissons la fonction $lookup$. Elle prend en entrée une valeur possible $idVal$ de l'attribut « id » du type $Item$. La fonction vérifie si une instance i d'$Item$ (avec $i.id = idVal$) est présente dans $Store$ et dans ce cas elle la retourne en sortie.

- Type de $lookup$:
  $(id : \langle activity : \mathbb{O}, inputs : [input : Data]\rangle \rightarrow Item)$
- Sémantique :
  $lookup(idVal) = i \mid i \in Store \text{ et } i.id = idVal$

La Figure 39 présente l'utilisation des fonctions $lookup$ et $bind$ dans le processus d'exécution d'un nœud *Retrieve++* afin de (1) rechercher des données au sein du $Store$ avant de les récupérer auprès du fournisseur de données et (2) d'ajouter les données récupérées dans le $Store$ pour une utilisation ultérieure. Rappelons que le diagramme présenté dans la figure est une version minimaliste du processus d'exécution du nœud *Retrieve++* et qu'elle sera enrichie toute au long de la section 4.2.

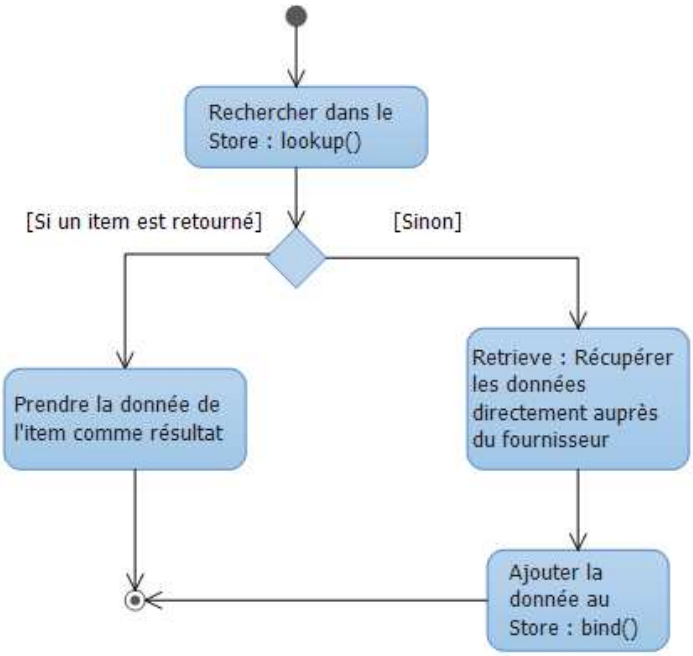

Figure 39 : Utilisation des fonctionnalités de gestion du $Store$ dans l'exécution d'un nœud *Retrieve++*

4.2.3 Remplacement de données

Le remplacement des items dans le $Store$ se fait selon une **politique de remplacement**. Celle-ci détermine les items devant être supprimés du $Store$ pour libérer de la place pour pouvoir stocker des nouveaux items. Elle vise à maximiser les avantages de l'utilisation du $Store$.

Nous définissons la fonction $makeroom$. Elle libère du $Store$ un espace dont la taille est égale ou supérieure à une valeur donnée $length$. Elle retourne une valeur booléenne indiquant si l'opération s'est bien exécutée ou pas.

- Type de la fonction : $(length : \mathbb{N} \rightarrow \mathbb{B})$.
- Sémantique de la fonction :
- Pour $makeroom(u)$ avec $u \in \mathbb{N}$:
  - Taille des données dans le $Store$ avant l'exécution $= v$
  - Taille des données dans le $Store$ après l'exécution $= v - u$

La Figure 40 présente l'utilisation de la fonction $makeroom$ dans le processus d'exécution d'un nœud *Retrieve++* afin de libérer de l'espace lorsque le $Store$ devient saturé.

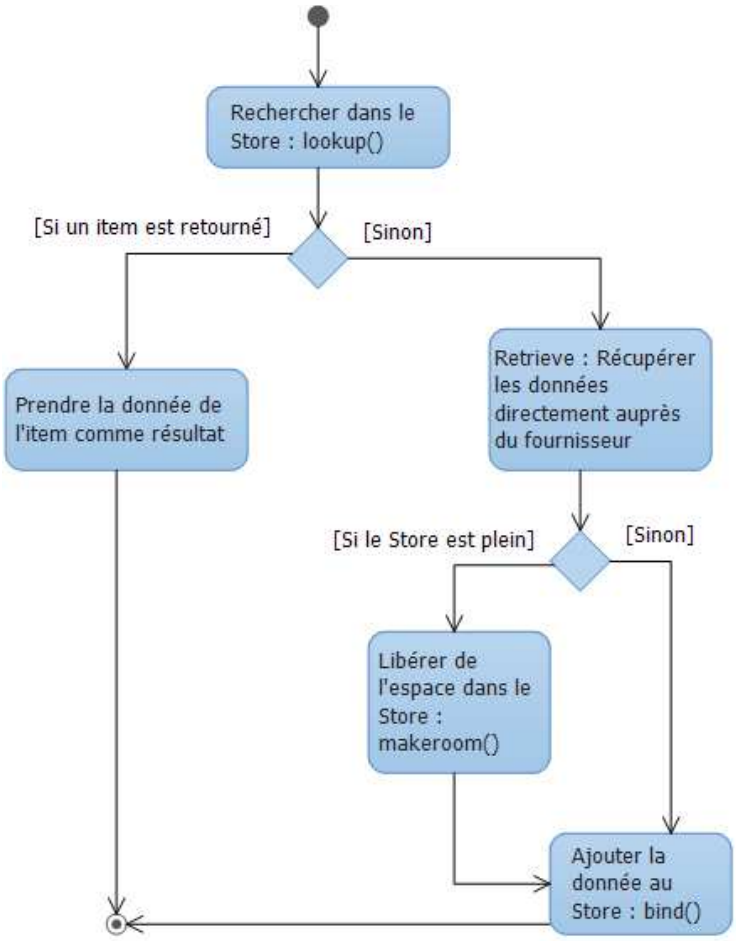

Figure 40 : Utilisation de la fonction $makeroom$ dans l'exécution d'un nœud *Retrieve++*

Le remplacement se fait selon plusieurs facteurs (sous-section 4.2.3.1) qui sont pondérés dans une formule de poids d'item (sous-section 4.2.3.2).

4.2.3.1 Facteurs de la politique de remplacement

La plupart des politiques de remplacement existantes (comme FIFO, LIFO, LRU (Last Recently Used), MRU (Most Recenty Used) …) sont basées sur le **moment d'ajout** et **d'accès** à une données. D'autres politiques sont basées sur des facteurs comme :

- La **taille des données** : en effet, il est plus intéressant de supprimer des items de grande taille pour laisser la place à plusieurs items de petite taille.
- La **latence** du fournisseur des données : en effet, il est plus intéressant de garder les items dont la récupération est plus lente.

Ces facteurs sont présents en tant qu'attributs dans la définition du type $Item$. Les valeurs de certains sont déterminées lors de la création de l'item (voir la fonction $bind$ dans la sous-section 4.2.2.1) alors que les valeurs d'autres attributs évoluent avec le temps comme le nombre d'accès et la date du dernier accès à l'item.

Dans le contexte des mashups les données sont dynamiques, elles ont une durée de validité déterminée dont il faut tenir compte. Les facteurs à prendre en compte sont

- La **durée de vie restante** pour un item. En effet, il est plus intéressant de garder un item i1 dont la durée de vie restante est supérieure à celle d'un item i2. Pour un item i du $Store$ ce facteur est calculé avec la formule : $i.expireOn - now()$
- Le **nombre d'accès** à un item i en prenant en compte son âge (calculé avec la formule $now() - i.birthDate$). En effet, il est plus intéressant de maintenir dans le $Store$ un item i1 pour lequel il y a eu 10 accès en une heure qu'un item i2 pour laquelle il y a eu 20 accès en 24 heures. Ce facteur est calculé avec la formule : $\frac{i.accessNb}{now()-i.birthDate}$

4.2.3.2 Le poids d'un item

La politique de remplacement associe un **poids** à chaque item présent dans le $Store$. Le poids d'un item définit la probabilité que cet item soit accédé dans le futur. Les items dont les poids sont les moins élevés, sont retenus pour être évincés lors du processus de remplacement. L'expression du poids est définie par une formule algébrique ayant comme variables des facteurs de remplacement de données. Une politique de remplacement choisie peut être adaptative : la formule correspondante contient des paramètres (coefficients ou exposants) de poids pour favoriser des facteurs par rapport à d'autres. Ces paramètres sont modifiables par l'administrateur du $Store$.

Une politique de remplacement adaptative peut donc être définie par la formule suivante :

$$poids = \alpha.LT + \beta.\frac{AN}{A} + \frac{\gamma}{Lat} + \frac{\delta}{Len}$$

Où :

- *LT* : durée de vie restante avant expiration.
- *A* : âge de l'item.
- *AN* : nombre d'accès à l'item.
- *Lat* : latence de la récupération de la donnée.
- *Len* : taille de l'item.
- α, β, γ et δ sont des coefficients attachés aux facteurs de la politique remplacement de données pour définir leurs poids.

Le type de fonctions de poids est $itemWeight : (Item \rightarrow weight : \mathbb{R})$. Une fonction de ce type permet de définir le poids (selon une formule déterminée) d'une instance de $Item$ donnée en entrée. Lors de l'éviction des données (par appel de la fonction $makeroom$), les instances, dont les poids sont les moins élevés, seront supprimées. Cette fonction peut être modifiée au cours du temps. Ceci donne à la fonction de remplacement une caractéristique d'adaptabilité.

4.2.4 Rafraichissement de données

Intuitivement, lorsqu'un item du $Store$ expire, il doit émettre une notification pour demander sa mise à jour. Ce processus nécessite d'associer à chaque item un mécanisme (thread) pour surveiller sa date d'expiration. Or le $Store$ est amené à stocker, au minimum, des milliers d'items. Techniquement, ceci revient à lancer autant de threads. Nous considérons que cette solution n'est pas optimale (gourmande en ressources) et donc nous l'écartons.

Nous proposons un processus de rafraichissement où les items doivent être parcourus périodiquement, pour mettre à jour ceux qui sont périmés. Pour éviter de parcourir tous les items, nous proposons de stocker des références aux items du $Store$ dans une structure de données ordonnée : $ItemsRefs$ (cf. Figure 41[31] et la sous-section 4.2.4.3).

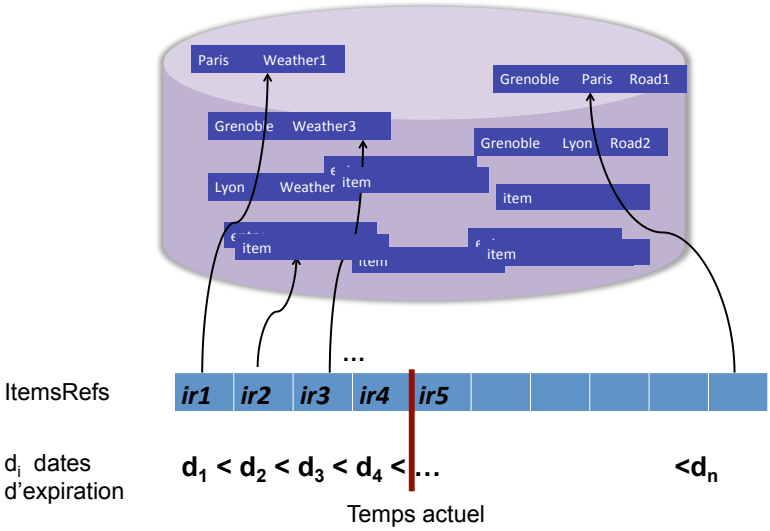

Figure 41 : Rafraichissement des items

Les éléments de $ItemsRefs$ sont des instances de type $ItemRef$. Chacune de ces instances contient une référence à un item du $Store$ et sa date d'expiration. L'ordre de ces instances dans $ItemsRefs$ est défini selon la date d'expiration. Le type $ItemRef$ est défini comme suit :

$$ItemRef : \langle ref : \mathbb{E}, expiryDate : Date \rangle$$

où $\mathbb{E}$ est l'ensemble des références des instances de type $Item$.

$Operations(ItemRef) = \{getItem\}$ où l'opération $getItem$ est de type $(iref : ItemRef \rightarrow output : Item)$. Pour une instance i de $ItemRef$, $i.getItem$ retourne l'item référencé par la référence $i.id$.

La fréquence de rafraichissement ainsi que la gestion des durées de vie des items seront étudiées dans les sous-sections 4.2.4.1 et 4.2.4.2. Ensuite, le choix de la structure de données de

[31] La représentation graphique linéaire de $ItemsRefs$ est indépendante de la structure de données choisie.

$ItemsRefs$ est fait dans la sous-section 4.2.4.3 en prenant en compte le coût de l'ajout, de la suppression et de la recherche des instances de $ItemRef$.

4.2.4.1 Fréquence de rafraichissements

Les rafraichissements sont réalisés selon une certaine fréquence. A chaque fois, il suffit de parcourir les premiers éléments d'$ItemsRefs$: ceux dont les dates d'expiration sont dépassées. Le parcours s'arrête dès lors qu'on accède à un item dont la date d'expiration est encore valide. Le parcours $ItemsRefs$ par étapes donne au mécanisme un aspect incrémental. Dans l'exemple de la Figure 41, les items référencés par *ir1*, *ir2, ir3* et *ir4* sont périmés et seront mis à jour lors du prochain parcours d' $ItemsRefs$.

La fréquence de rafraichissements $refreshFreq$ est fixée par l'administrateur du $Store$. Elle peut avoir une valeur statique ou dynamique calculée via une fonction $defineRefreshFreq$ de type $(\rightarrow frequency : \mathbb{N})$. Après chaque rafraichissement des items périmés, la fonction $defineRefreshFreq$ assigne à la $refreshFreq$ une valeur qui peut correspondre à :

- La durée de vie restante du prochain item qui va expirer : celle ci contenue dans la première référence dans la structure $entriesIds$.
- La moyenne des durées de vie restantes des prochains n (nombre fixé par l'administrateur du $Store$) items à mettre à jour.
- Ou tout autre calcul décidé par l'administrateur du $Store$.

4.2.4.2 Durée de vie des items

Le choix de la valeur d'une durée de vie TTL d'un item donne lieu à un compromis entre le risque de garder des données périmées longtemps et le cout de récupérations multiples de la même donnée. En effet, des grandes valeurs TTL induisent une augmentation du nombre des items périmés dans le $Store$. Tandis que des petites valeurs TTL engendrent un travail supplémentaire pour lancer des requêtes afin de récupérer des données. Dans ces deux cas de figure, les bénéfices de notre approche pour améliorer la disponibilité des données fraiches, sont diminués.

La durée de vie d'un item est estimée par le constructeur de mashup lors de la définition de celui ci. Le constructeur associe aux activités, proposées par des fournisseurs de données utilisées, des estimations de durées de vie. Ces estimations sont ajustées lors du rafraichissement. Nous adaptons la technique du TTL incrémental [86] au contexte des mashups pour définir le processus d'ajustement de la durée de vie des items du $Store$. Cette technique est considérée comme étant adaptative parce qu'elle adapte la valeur de TTL en prenant en compte son ancienne valeur, et des données statistiques.

La stratégie du TTL incrémental vise à ajuster la valeur de la durée de vie TTL d'un item. En se basant sur le TTL actuel et sur des données statistiques. Elle est mise en œuvre par la fonction $processItem$ de $(ItemRef \rightarrow \mathbb{B})$. Pour une référence d'un item dont la durée de vie est expirée, la fonction $processItem$ récupère la donnée correspondante depuis le fournisseur de données et la compare avec la donnée que l'item contient (dans l'attribut « object »). Si l'item est jugé périmé, la valeur de TTL est diminuée. Tandis que, si l'item est jugé encore valide. La valeur de TTL est recalculée. L'emplacement de l'identificateur de l'item dans $entriesIds$ est ajusté. Ce fonctionnement est illustré dans le diagramme d'activité de la Figure 42 dans la section 4.2.5.

L'ajustement de la valeur de la durée de vie TTL se fait selon deux cas de figure :

(a) Item périmé

Si l'item est jugé périmé, ceci constitue une justification pour considérer que la valeur de TTL est surestimée et qu'il faut la diminuer. Le **décrément** est calculé par une fonction dec en fonction de la valeur actuelle de TTL et des données statistiques comme le nombre d'accès à l'item, la date du dernier accès. dec est de type $(oldttl : \mathbb{N}, an : \mathbb{N}, la : Date \rightarrow decrement : \mathbb{N})$,

Plus (1) le nombre d'accès est grand (plus grand qu'un seuil donné), ou (2) plus le temps écoulé depuis la dernière date d'accès est petit (plus petit qu'une durée donnée) alors plus le décrément doit être grand. Et inversement, plus le nombre d'accès est petit et le temps écoulé depuis la dernière date d'accès est grand, plus le décrément doit être petit. Ce choix est justifié par le souci de maintenir le plus possible la fraicheur (retrait d'un grand décrément) des données qui sont fréquemment ou récemment demandées.

La valeur de TTL peut être décrémentée jusqu'à une valeur minimale T_{min}. Ce seuil constitue une limite inférieure des valeurs TTL des items pour éviter l'explosion de nombre de requêtes pour récupérer les données depuis les fournisseurs des données. Ainsi la valeur de TTL est définie selon la formule suivante :

$$TTL = Max\left(T_{min}, \widehat{TTL} - dec\left(\widehat{TTL}, AN, LA\right)\right)$$

Où :

- $\widehat{TTL}$ est la valeur actuelle de la durée de vie de l'item.
- AN est le nombre d'accès à l'item.
- LA est la date du dernier accès à l'item.

Par exemple, un décrément peut être défini selon la formule suivante :

$$dec\left(\widehat{TTL}, AN, LA\right) = \begin{cases} \dfrac{\widehat{TTL}}{4} & Si\ AN > \gamma\ et\ le\ temps\ depuis\ le\ dernier\ accès < \delta \\ \dfrac{\widehat{TTL}}{10} & Sinon \end{cases}$$

Où γ et δ sont des seuils définis par l'administrateur du $Store$.

(b) Item valide

Si l'item est jugé encore valide, ceci constitue une justification pour considérer que la valeur de TTL est sous-estimée et qu'il faut l'augmenter. L'**incrément** est calculé par une fonction inc en fonction de la valeur actuelle de TTL et des données statistiques comme le nombre d'accès à l'item, la date du dernier accès. inc est de type $(oldttl : \mathbb{N}, an : \mathbb{N}, la : Date \rightarrow increment : \mathbb{N})$.

Plus (1) le nombre d'accès est grand (plus grand qu'un seuil donné), ou (2) plus le temps écoulé depuis la dernière date d'accès est petit (plus petit qu'une durée donnée) alors plus l'incrément doit être petit. Et inversement, plus le nombre d'accès est petit et le temps écoulé depuis la dernière date d'accès est grand, plus l'incrément doit être grand. Ce choix est justifié par le souci de maintenir le plus possible la fraicheur (ajout d'un petit incrément) des données qui sont fréquemment ou récemment demandées.

La valeur de TTL peut être incrémentée jusqu'à une valeur maximale T_{max}. Ce seuil constitue une limite supérieure des valeurs TTL des items pour éviter le risque de la présence des données périmées dans le $Store$ pour longtemps. Ainsi la valeur de TTL est définie selon la formule suivante :

$$TTL = Min\left(T_{max}, \widehat{TTL} + inc\left(\widehat{TTL}, AN, LA\right)\right)$$

Où :

- $\widehat{TTL}$ est la valeur actuelle de la durée de vie de l'item.
- AN est le nombre d'accès à l'item.
- LA est la date du dernier accès à l'item.

Par exemple, un incrément peut être défini selon la formule suivante :

$$inc\left(\widehat{TTL}, AN, LA\right) = \begin{cases} \frac{\widehat{TTL}}{10} & Si\ AN > \alpha\ et\ le\ temps\ depuis\ le\ dernier\ accès < \beta \\ \frac{\widehat{TTL}}{2} & Sinon \end{cases}$$

Où α et β sont des seuils définis par l'administrateur du $Store$.

4.2.4.3 Choix de la structure de données de $ItemsRefs$

Le choix de la structure de $ItemsRefs$ peut se faire entre différentes structures comme un tableau ordonné, une liste chainée ordonnée, un arbre binaire ou un arbre rouge-noire. Pour des détails à propos de ces structures, le lecteur est invité à consulter la référence [96].

Pour toutes ces structures, La recherche d'un objet (instance de $ItemRef$) à partir de la valeur de l'attribut ref nécessite le parcours séquentiel du tableau (Complexité $\mathcal{O}\ (n)$). Ceci est dû au fait que les objets sont triés par rapport à la date d'expiration de l'item référencé et non pas par rapport à la référence de l'item.

Dans un tableau ordonné, les objets sont accédés par leurs indices. Les coûts de l'insertion et la suppression d'un objet sont linéaires par rapport à la taille du tableau.

Dans une liste chainée, les objets sont arrangés linéairement. Les coûts de l'insertion et la suppression d'un objet sont linéaires par rapport à la taille de la liste.

Dans un arbre binaire de recherche les coûts de l'insertion et de la suppression dépendent de la profondeur de l'arbre. Chacun est de l'ordre $\mathcal{O}\ (h)$ sur un arbre de profondeur maximale h. Les coûts peuvent aller de $\mathcal{O}\ (\lg(n))$ dans les cas les plus favorables (arbres binaires complets) jusqu'à $\mathcal{O}\ (n)$ dans les cas les plus défavorable (arbre constituée d'une chaine linéaire de *n* nœuds).

Un arbre rouge-noir est un type particulier d'arbre binaire de recherche. Chaque nœud est coloré (rouge ou noire) selon les règles suivantes :

- La racine et les feuilles sont noires.
- Si un nœud est rouge alors ses deux nœuds fils sont noirs.
- Pour tout nœud, les chemins reliant ce nœud à des feuilles contiennent le même nombre de nœuds noirs.

La coloration garantie qu'aucun des chemins allant de la racine aux feuilles n'est plus de deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement équilibré. Les coûts de l'insertion et la suppression d'un objet sont de l'ordre logarithmique (Complexité $\mathcal{O}\ (\lg(n))$).

Le Tableau 5 dresse la complexité des opérations d'insertion, de suppression et de recherche dans différentes structures. Nous constatons que l'arbre rouge-noir offre de meilleures performances en ce qui concerne l'insertion et la suppression de nouveaux objets.

	Insertion/suppression	**Recherche**
Tableau	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Liste chainée	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Arbre binaire	$\mathcal{O}(h)$	$\mathcal{O}(n)$
Arbre rouge-noir	$\mathcal{O}(\lg(n))$	$\mathcal{O}(n)$

Tableau 5 : Coût des opérations dans différentes structures de données[32]

Cette comparaison nous permet de choisir l'arbre rouge-noir comme structure de données pour stocker les références des items du $Store$. Nous définissons le type arbre rouge noir comme suit :

- Si t est un type alors $A : \lfloor t \rfloor \in \mathbb{T}$ est un type arbre rouge noir nommé A. Ses nœuds sont des instances du type t.
- Pour un type arbre rouge noir $A : \lfloor t \rfloor$, $[\![A : \lfloor t \rfloor]\!] = \{A : S \mid S \in [\![t]\!] \cup ([\![t]\!] \times [\![A : \lfloor t \rfloor]\!] \times [\![A : \lfloor t \rfloor]\!])\}$.

$Operations(A : \lfloor t \rfloor) = \{insert, delete, lookup, getNode, getLeftChild, getRightChild\}$

4.2.5 Exécution d'un nœud *Retrieve++*

L'interaction entre le mashup et le $Store$ se déroule lors de l'exécution des nœuds *Retrieve++*. Elle se fait en exploitant les fonctions associées au $Store$. Elle est illustrée dans le diagramme d'activité de la notation UML dans la Figure 42. L'interaction se déroule comme suit :

- Vérifier (avec la fonction $lookup$) si un item du $Store$ contient la donnée recherchée.
  - Si l'item existe :
    - S'il est périmé, il est rafraichi et sa durée de vie est ajustée selon la stratégie TTL incrémental décrite dans la sous-section 4.2.4.2
    - Dans tous les cas, la donnée qu'il contient est prise comme résultat de l'exécution du nœud Retrieve++.
  - Sinon :
    - La donnée est récupérée directement auprès du fournisseur. Et elle est prise comme résultat de l'exécution du nœud Retrieve++.
    - Si le Store est plein, des items sont supprimés avec la fonction $makeroom$. La suppression se fait selon une formule de remplacement comme celle définie dans la sous-section 4.2.3.2.
    - Un nouvel item, contenant la donnée récupérée, est crée et ajouté au $Store$ avec la fonction $bind$.

Nous rappelons que, parallèlement à ce processus, le $Store$ est parcouru périodiquement pour rafraichir les items périmés d'une façon proactive sans attendre l'arrivé d'une nouvelle

[32] *n* fait référence au nombre des éléments dans la structure tandis que *h* fait référence à la hauteur de l'arbre (profondeur maximale d'une branche)

demande d'accès aux items. Ce processus prend en compte des formules de rafraichissement comme celles définies dans la sous-section 4.2.4.

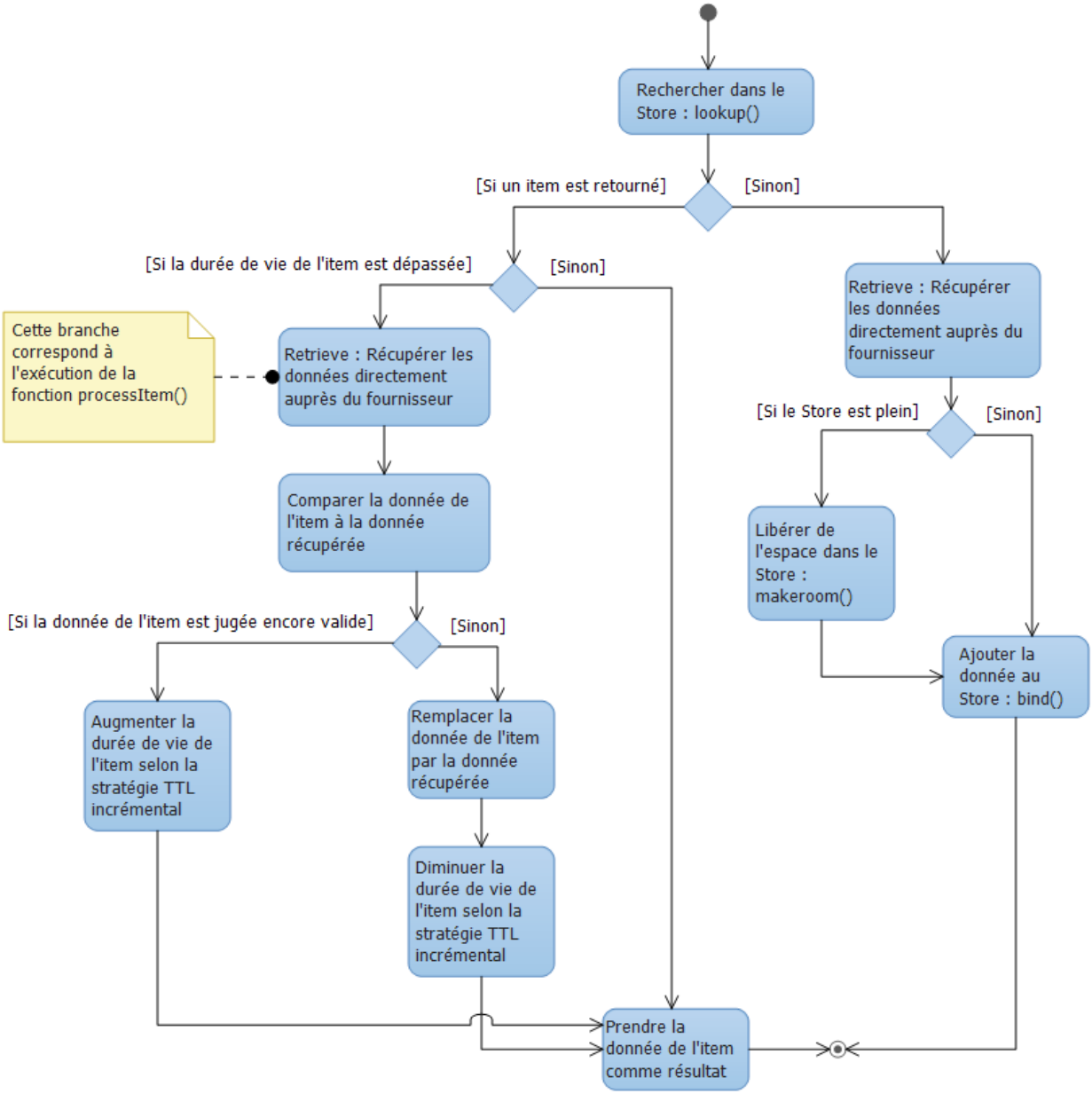

Figure 42 : Exécution d'un nœud *Retrieve++*

4.3 Conclusion

Dans ce chapitre, nous avons présenté les phases du processus d'exécution d'un mashup. et notre solution pour améliorer la disponibilité de données fraiches des mashups. Celle ci est à base de fonctions orthogonales au processus d'exécution des mashups.

Dans une première section, nous avons décrit la représentation d'un mashup sous la forme d'un graphe dont le parcours permet de coordonner l'exécution des mashlets et des wirings du mashup. Nous avons décrit les phases des processus d'exécution des mashlets et des wirings et de leurs activités sous-jacentes.

Dans une deuxième section, Nous avons présenté des fonctions permettant d'améliorer la disponibilité des données récupérées auprès des fournisseurs de données. Ces données sont rendues disponibles dans un $Store$. Pour cela, nous avons décrit l'organisation du contenu du $Store$ en un ensemble d'items. Nous avons ensuite décrit la gestion de ce contenu et les fonctions qui y sont relatives : ajouter, modifier, supprimer et rechercher des items. Ensuite, nous avons décrit les fonctions de remplacement et de rafraichissement des items. Pour le remplacement des items, nous avons dressé une liste de facteurs entrant en jeu lors de la définition de la politique de remplacement dans le contexte des mashups. Le rafraichissement des items est exécuté d'une façon incrémentale. Nous avons adopté une stratégie adaptative pour la définition des durées de vie des items ainsi nous avons adapté la stratégie TTL incrémental au contexte des mashups. Enfin nous avons illustré, avec des diagrammes d'activité de la notation UML, le processus d'exécution d'un nœud *Retrieve++* avec l'exploitation du Store et des fonctions associées.

Chapitre 5

Implantation de MELQART

L'implantation de MELQART consiste en la séparation et l'abstraction des concepts des mashups ainsi que des gestionnaires de cache implantant les fonctionnalités proposées pour améliorer la disponibilité des données des mashups. Ceux ci sont définis avec des classes abstraites avec des signatures d'opérations (Figure 43). Cette abstraction permet de représenter l'interdépendance entre les composants en terme d'opérations fournies et utilisées par les classes. Ainsi la modification de l'implantation d'une opération n'affecte pas les classes qui l'utilisent.

Les concepts des mashups sont définis par des classes abstraites et concrètes par les programmeurs de MELQART. Lors de la définition de mashups, les classes abstraites sont spécialisées par des classes concrètes. Des instances de ces classes permettent d'obtenir des mashups.

Nous avons utilisé le canevas de cache ACS [62]. Il est à base de gestionnaires ; chaque gestionnaire est un composant réalisant une fonctionnalité spécifique (gestion de données, remplacement de données etc.). Dans MELQART, nous avons adapté ces gestionnaires au contexte des mashups et nous avons introduit un gestionnaire de rafraichissement.

Les gestionnaires de remplacement et de rafraichissement MELQART sont spécifiés par des classes abstraites. Leurs comportements sont spécialisés par des classes concrètes. Par exemple définir des gestionnaires de remplacement, chacun ayant sa propre politique de remplacement. Des instances de ces classes permettent de faire fonctionner un cache pour stocker les données des instances de mashups. Elles sont manipulées par des administrateurs de MELQART qui peuvent contrôler le cache : augmenter sa taille, le vider, ou changer de politique de remplacement.

Le chapitre est organisé comme suit : l'architecture du système est présentée dans la section 5.1. L'implantation des concepts de notre modèle de mashups et l'adaptation des gestionnaires d'ACS sont présentées dans les sections 5.2 à 5.11. L'implantation est décrite avec les diagrammes de classes et de séquence de la notation UML. Dans la section 5.12, nous avons validé notre implantation avec l'exécution de plusieurs instances de deux scenarios de mashups. Enfin, la section 5.13 conclut ce chapitre.

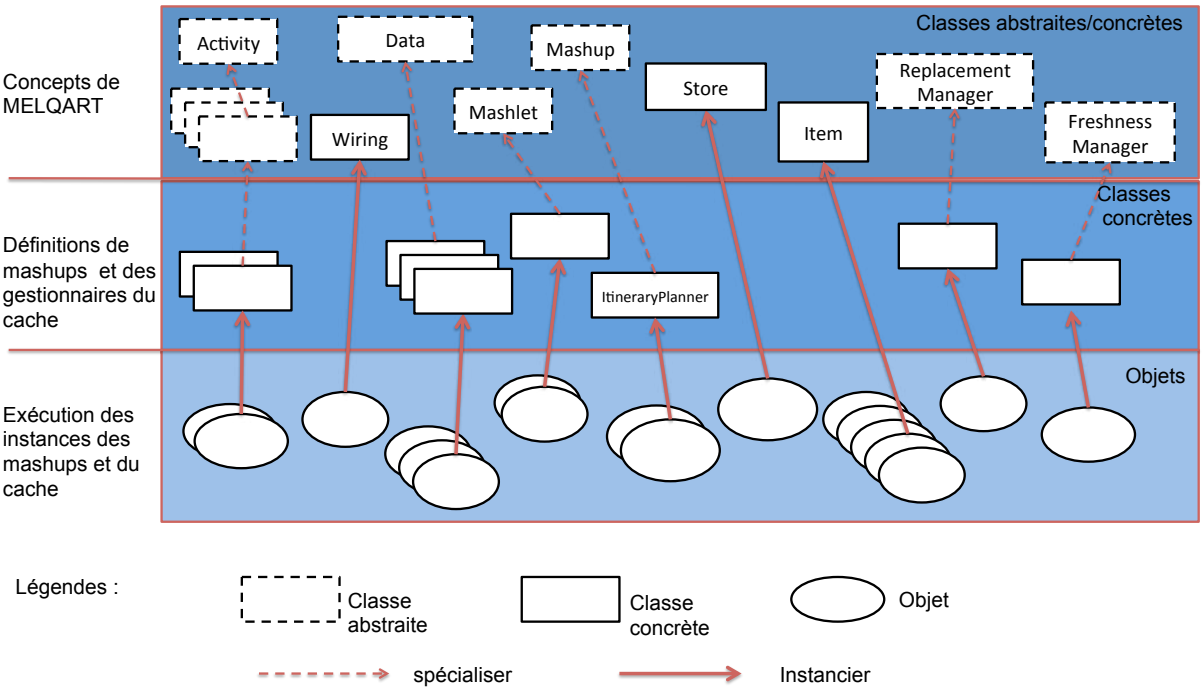

Figure 43 : Approche de MELQART pour la création et l'exécution des mashups

5.1 Architecture du système

L'interaction avec le système MELQART se fait selon une architecture client-serveur (cf. Figure 44). Les mashups sont exécutés côté serveur et les résultats sont affichés sur les machines des utilisateurs (clients). Ainsi, deux utilisateurs peuvent demander l'exécution d'un même mashup.

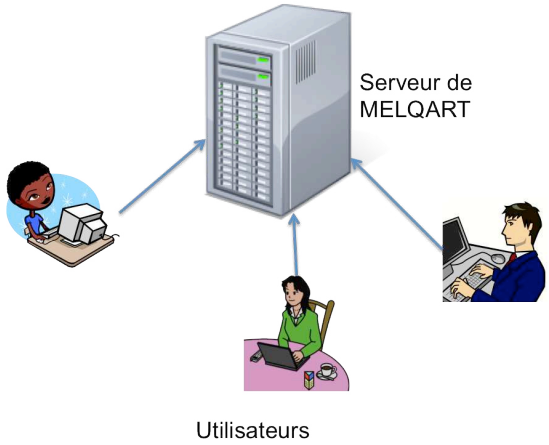

Figure 44 : Interaction des utilisateurs avec MELQART

La Figure 45 présente les composants de MELQART : MashupEngine, Mashup, Mashlet, Wiring, Activity, Store, ReplacementManager et FreshnessManager. Il interagit avec des composants externes qui représentent les fournisseurs de données : Services.

Un fournisseur de donnée (Service) est un composant autonome que l'on peut interroger (Invoke) par le biais des activités qu'il propose. L'exécution d'un mashup est accomplie par le

composant MashupEngine. Celui ci exécute les composants Mashup, Mashlet, Wiring et Activity[33]. Il contient un buffer qui offre les outils pour le stockage temporaire de données en cours de traitement. Il interagit avec les composants Services afin de les interroger et récupérer des données. Il interagit également avec le composant Store qui stocke des résultats des activités de fournisseurs de données. Dans notre implantation, les fonctions du composant MashupEngine sont assurées par le moteur d'exécution d'Eclipse (notre environnement de développement).

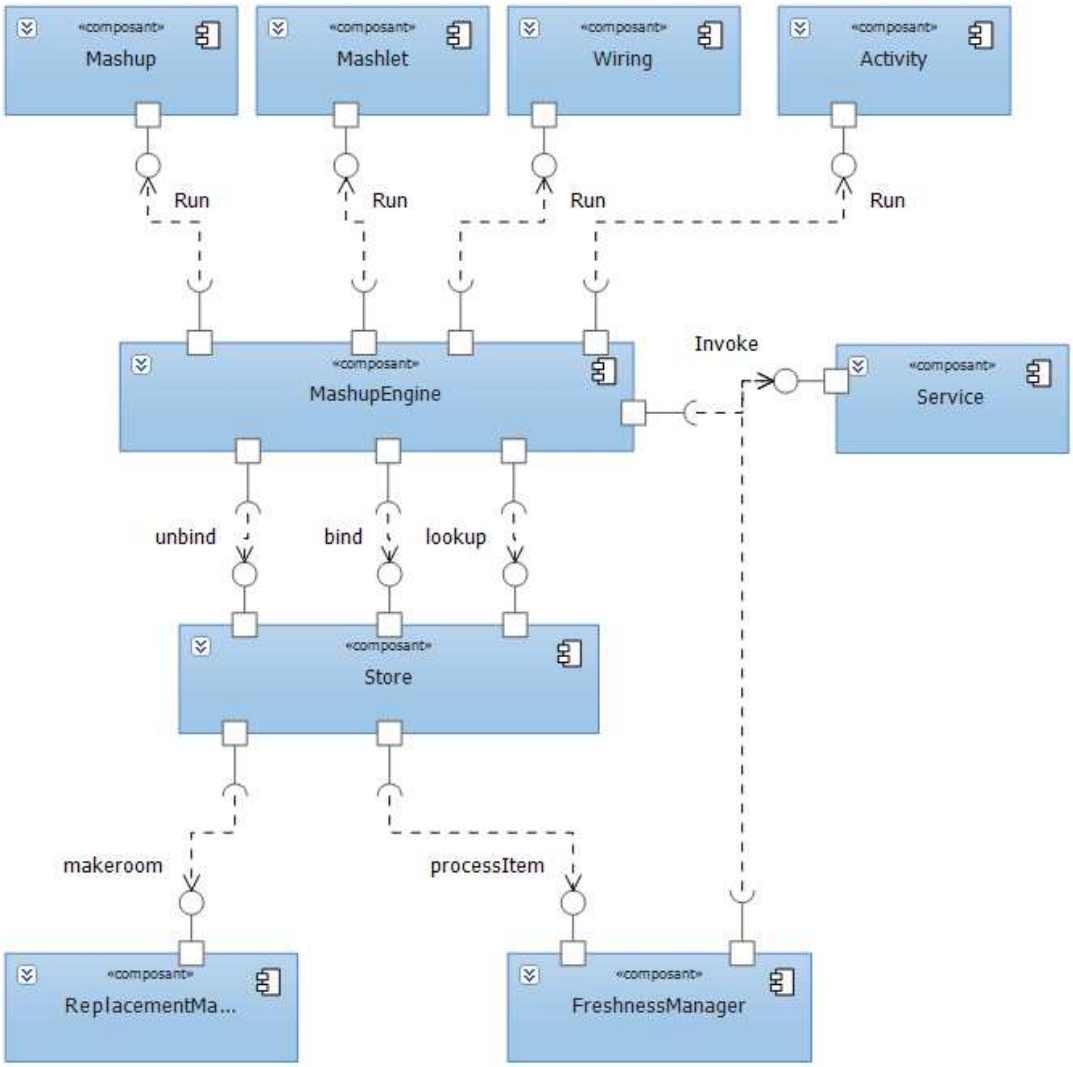

Figure 45 : Architecture de MELQART

Le composant Store est un cache. Il fournit les opérations de base permettant l'ajout (bind), la suppression (unbind) et la recherche de données (lookup). Il interagit avec le gestionnaire de remplacement (ReplacementManager). Celui ci définit les données qu'il faut supprimer pour laisser de la place à des nouvelles données. Le Store interagit également avec le gestionnaire de rafraichissement (FreshnessManager) qui vise à maintenir la fraicheur des données dans le cache. Le fonctionnement des ces composants est défini en prenant en compte le caractère dynamique des données des mashups comme décrit dans le Chapitre 4.

[33] Il existe différents composants activité (Filter, Extract…). Dans cette section, nous nous y référons sous le nom générique Activity

Dans notre implantation le cache est déployé sur le serveur d'exécution des mashups. Néanmoins, il peut très bien être déployé aussi sur un autre serveur et ceci sans affecter le principe de son fonctionnement.

Nous exploitons le Store pour stocker les résultats des activités proposées par des fournisseurs de données. Ceci permet d'assurer (1) leur disponibilité même en cas de défaillance d'un des fournisseurs et (2) leur partage entre différents mashlets de différents mashups, et ceci avant de les manipuler.

5.2 Valeurs complexes

Les données consommées par les mashups sont disponibles sur le web ou saisies par l'utilisateur en entrée. Les données du web sont récupérées chez des fournisseurs de données (des services web) et chacune est identifiée par un URI (*Universal Resource Identifier*). Les récupérations des données se font via des appels de méthodes REST.

Nous adoptons la notation JSON[34] pour représenter les données. En effet, les types des valeurs complexes sont compatibles avec la notation JSON[35]. En principe, les objets JSON peuvent représenter les valeurs complexes de type tuple. Alors que les tableaux JSON peuvent représenter les valeurs complexes de type liste ou ensemble. Ce choix n'écarte pas les données décrites sous format XML ou un autre langage de la même famille. En effet une donnée décrite sous XML peut très bien être convertie en donnée JSON et vice versa. Dans notre environnement, nous utilisons la librairie[36] de référence JSON de Java pour implanter les valeurs complexes.

Classe JSONObject

Nous considérons la classe JSONObject[37] pour implanter les données des mashups. Cette classe est offerte par la librairie de référence JSON de Java. Elle contient des méthodes qui permettent de lire les valeurs des attributs, d'ajouter et modifier des attributs.

Représentation des valeurs complexes via JSON

La représentation des valeurs complexes via JSON peut être faite avec l'ensemble de règles suivantes :

1. Une valeur complexe de type tuple de la forme $A : \langle A_1 : v_1, \dots, A_n : v_n \rangle$ est représentée par deux objets JSON imbriqués de la forme $\{"A" : \{"A_1" : v_1, \dots, "A_n" : v_n\}\}$.
2. Une valeur complexe de type liste de la forme $A : [A' : v_1, \dots, A' : v_n]$ est représentée par un tableau d'objets JSON imbriqué lui même dans un objet JSON de la forme $\{"A" : [\{"A'" : v_1\}, \dots, \{"A'" : v_n\}]\}$.
3. Une valeur complexe de type ensemble de la forme $A : \{A' : v_1, \dots, A' : v_n\}$ est représentée par un tableau d'objets JSON imbriqué lui même dans un objet JSON de la forme $\{"A" : [\{"A'" : v_1\}, \dots, \{"A'" : v_n\}]\}$.

Étant donné que les valeurs JSON n'ont pas de nom associé, à l'exception des valeurs contenues dans des objets, nous utilisons un objet JSON englobant pour spécifier les noms des valeurs complexes (tuples, listes, ensembles). Bien que ce choix puisse paraître encombrant pour

[34] http://www.json.org/json-fr.html
[35] La syntaxe de JSON est présentée dans l'Annexe B
[36] http://www.json.org/java/
[37] http://www.json.org/javadoc/org/json/JSONObject.html

une écriture manuelle des données JSON, il ne représente pas un problème pour une implantation de données par programmation.

Exemple

La valeur complexe

$$restaurants : \begin{bmatrix} restaurant : \langle name : "nom4", speciality : "italienne", address : "ad1", rating : 3.5\rangle, \\ restaurant : \langle name : "nom1", speciality : "française", address : "ad2", rating : 4.5\ \rangle, \\ restaurant : \langle name : "nom2", speciality : "libanaise", address : "ad3", rating : 4\rangle, \\ restaurant : \langle name : "nom3", speciality : "japonaise", address : "ad4", rating : 3.3\rangle \end{bmatrix}$$

peut être représentée avec l'objet JSON suivant :

```
{"restaurants" : [
      {
         "restaurant" : {
            "name" : "nom4",
            "speciality" : "italienne",
            "address" : "ad1",
            "rating" :3.5
      },
      {
         "restaurant" : {
            "name" : "nom1",
            "speciality" : "française",
            "address" : "ad2",
            "rating " :4.5
      },
      {
         "restaurant" : {
            "name" : "nom2",
            "speciality" : "libanaise",
            "address" : "ad3",
            "rating " :4
      },
      {
         "restaurant" : {
            "name" : "nom3",
            "speciality" : "japonaise",
            "address" : "ad4",
            "rating " :3.3
      }
    ],
 }
```

5.3 Activité

Les activités sont implantées en tant que classes. La classe Activity (Figure 46), est une classe abstraite : elle ne peut pas être instanciée directement. Deux associations avec la classe JSONObject spécifient qu'une instance de la classe Activity consomme une instance de la classe JSONObject en entrée (**input)** et en produit une autre en sortie (**output)** de type JSONObject.

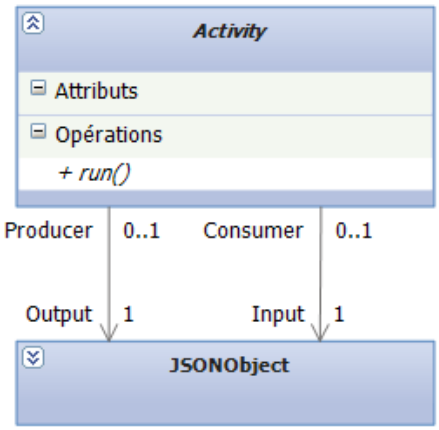

Figure 46 : Classe Activity

La classe Activity contient une méthode abstraite **run()**. Les classes filles spécialisent le fonctionnement de cette méthode. Celle ci définit le traitement que l'activité doit opérer pour produire la sortie.

Une activité peut être basique ou composite. Une activité basique exécute une opération alors qu'une activité composite coordonne l'exécution de certaines activités qu'elles soient basiques ou composites.

5.3.1 Activités basiques

Dans notre modèle, Nous avons considéré un ensemble fini d'activités basiques pour le filtrage (**Filter**), l'extraction (**Extract**), le tri (**Sort**), la récupération de données (**RetrievePP**) et l'élimination de doublons (**Unique**). L'implantation de chacune de ces activités est faite avec une classe portant le même nom. Ces classes sont données dans la Figure 47. Elles étendent la classe Activity, et spécialisent le fonctionnement de la méthode `run()`. Elles sont abstraites et chacune possèdent des méthodes abstraites. Celles ci sont spécialisées par des classes filles qui sont définies lors de l'implantation des mashups.

Dans les sous-sections suivantes, nous présentons (1) les classes de ces activités basiques, (2) comment elles spécialisent la méthode `run()` et (3) illustrer comment elle peuvent être étendues (définition de classes filles) pour l'implantation du mashup ItineraryPlanner. Des instances de ces classes sont présentées dans la sous-section 5.3.2 qui définit la coordination des activités (illustrant ainsi comment ces instances peuvent interagir entre elles).

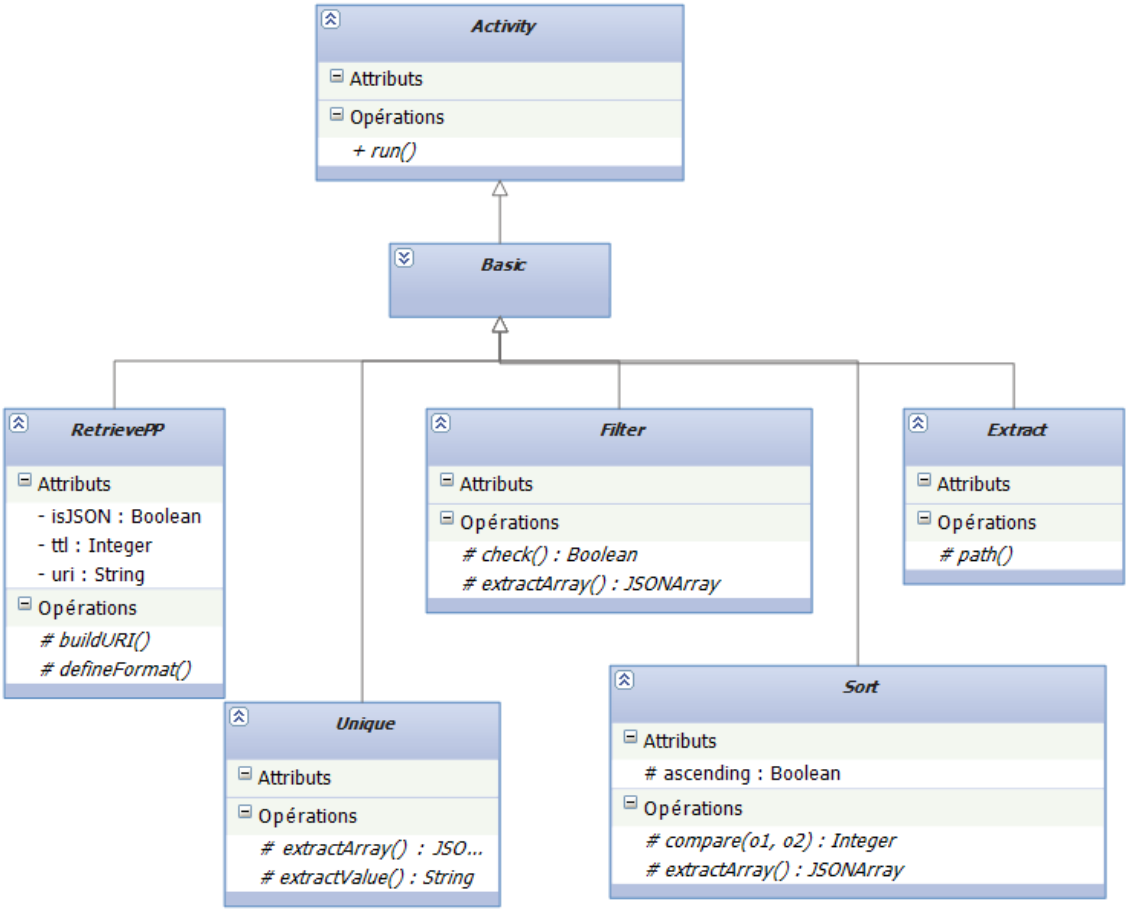

Figure 47 : Les classes des activités basiques

5.3.1.1 RetrievePP

La classe RetrievePP contient un attribut **uri** de type chaine de caractères, un attribut **isJSON** de type booléen et un attribut **ttl** de type entier. Elle contient également deux méthodes abstraites `buildURI()` et `defineFormat()`. Une instance de la classe RetrievePP récupère une donnée (1) du Store si celle ci s'y trouve elle y est, ou (2) du web auprès d'un fournisseur de données avec la méthode run(). La donnée récupérée est retournée en sortie. Cette donnée est identifiée par l'attribut uri. L'activité prend en entrée une donnée dont la valeur intervient dans la construction de l'URI via la méthode buildURI(). L'attribut **ttl** (Time To Live) de type entier, définit la durée de vie de la donnée récupérée auprès du fournisseur des données. Sa valeur est fixée par le constructeur du mashup pour chaque classe fille de la classe Retrieve. La valeur dépend de la nature des données récupérées. Le fonctionnement de la méthode `run()` est décrit dans la sous-section 5.11 décrivant l'interaction entre les mashups et le Store. La méthode `defineFormat()` définit la valeur de l'attribut isJSON. Celui ci indique si la donnée récupérée est sous format JSON (`isJSON = true`) ou sous format XML (`isJSON = false`). Dans ce dernier la méthode `run()` convertit la donnée dans un format JSON.

Les classes filles spécialisent le fonctionnement des méthodes `buildURI()` et `defineFormat()`. Par exemple, lors de l'implantation du mashup ItineraryPlanner, le constructeur définit une classe concrète RouteRetrieve qui étend la classe Retrieve. Cette classe spécialise le fonctionnement des deux méthodes. `buildURI()` définit l'URI pour la récupération de l'itinéraire entre deux villes alors que `defineFormat()` indique que la donnée récupérée est sous format JSON. Le fonctionnement de ces méthodes est le suivant :

```
void buildURI() {
   depart = input.departure ;
   destination = input.destination ;
   this.uri =
"http://maps.googleapis.com/maps/api/directions/json?origin="+departure+"&des
tination="+destination+"&sensor=false" ;
}

void defineFormat () {
   this.isJSON = true ;
}
```

5.3.1.2 Filter

La classe Filter contient les méthodes abstraites `check(JSONObject value)` et `extractArray()`. Une instance de la classe Filter prend en entrée une instance de JSONObject contenant un tableau d'objets JSON. Celui ci est extrait avec la méthode `extractArray()`. La méthode `run()` retourne le tableau des objets qui vérifient une certaine condition. La vérification de la condition est exécutée par la méthode `check(JSONObject value)`. Elle se fait, éventuellement, par rapport à des valeurs contenues dans l'entrée de l'activité.

La méthode `run()` est décrite dans le pseudo-code suivant :

```
void run() {
   JSONArray valuesArray = this.extractArray() ;
   JSONArray resultArray;
   for (int i=1 ; i< valuesArray.length() ; i++) {
      value = valuesArray[i] ;
      if (check(value)) then
         resultArray.put(value) ;
   }
JSONObject output = {"result" : resultArray} ;
this.output = output ;

}
```

Les classes filles spécialisent le fonctionnement des méthodes `extractArray()` et `check()`. Par exemple, lors de la définition du mashup ItineraryPlanner, le constructeur définit une classe concrète RestaurantFilter. L'entrée de l'activité contient la liste de restaurants sous la forme d'un tableau JSON et la note minimale pour les restaurants recherchés (`"ratingMin"`) Les méthodes `extractArray()` et `check()` sont décrites avec le pseudo-code suivant :

```
JSONArray extractArray() {
   JSONArray valuesArray = this.input.getJSONArray("restaurants") ;
   return valuesArray ;
}

Boolean check(JSONObject value) {
   If (value.getInt(("rating") >= this.input.getInt("ratingMin")) then
      return true ;
   else
      return false ;
}
```

5.3.1.3 Unique

La classe Unique contient les méthodes abstraites `extractValue(JSONObject o)` et `extractArray()`. Une instance de la classe Unique prend en entrée une instance de JSONObject contenant un tableau d'objets JSON. Celui ci est extrait avec la méthode `extractArray()`. La méthode `run()` élimine les doublons parmi les objets de ce tableau. L'élimination des doublons se fait selon des valeurs contenues dans les objets et désignées par un même chemin d'accès. La méthode `extractValue (JSONObject o)` retourne la valeur contenue dans l'objet o sous la forme d'une chaine de caractère.

La méthode `run()` est décrite dans le pseudo-code suivant :

```
void run() {
   JSONArray objectArray = this.extractArray() ;
   JSONArray resultArray;

   String[] objKeys
   for (int i=1 ; i< objectArray.length() ; i++) {
      obj = objectArray [i] ;
      if ( ! objKeys.contains(extractValue(obj))) then
         resultArray.put(obj) ;
   }
JSONObject output = {"result" : resultArray} ;
this.output = output ;
}
```

Les classes filles spécialisent le fonctionnement des méthodes `extractArray()` et `extractValue(JSONObject o)`. Par exemple dans le mashup ItineraryPlanner, la liste de villes (sous format WOEID) de passage entre les villes de départ et de destinations contient des doublons qu'il faut éliminer. Pour cela, le constructeur définit une classe concrète UniqueWOEID qui étend la classe Unique. Le fonctionnement des méthodes `extractArray()` et `extractValue(JSONObject o)` de cette classe est décrit avec les pseudo-codes suivants :

```
JSONArray extractArray() {
   JSONArray valuesArray = this.input.getJSONArray("result") ;
   return valuesArray ;
}

String extractValue(JSONObject o) {
   int woeid = o.getInt("woeid") ;
   return String(woeid) ;
}
```

5.3.1.4 Extract

La classe Extract contient la méthode abstraite `path()`. Une instance de la classe Extract prend en entrée une ressource, et retourne dans un objet JSON la valeur indiquée par la méthode `path()`.

La méthode `run()` est décrite dans le pseudo-code suivant :

```
void run() {
   this.output = this.path()
}
```

Les classes filles spécialisent le fonctionnement de la méthode `path()`. Par exemple, lors de l'implantation du mashup ItineraryPlanner, le constructeur du mashup définit une classe concrète StepsExtractor qui étend la classe Extract. La méthode `path()`, de cette classe, extrait les étapes de l'itinéraire. Elle est décrite avec le pseudo-code suivant :

```
JSONObject path() {
   JSONArray steps = this.input.getJSONArray("routes").getJSONObject(0).
getJSONArray("legs").getJSONObject(0).getJSONArray("steps")

   return {"steps" : steps} ;
}
```

5.3.1.5 Sort

La classe Sort contient un attribut `ascending` de type booléen et les méthodes abstraites `extractArray()` et `compare(JSONObject o1, JSONObject o2)`. Une instance de la classe Sort prend en entrée une ressource contenant un tableau d'objets JSON. Celui ci est extrait avec la méthode `extractArray()`. La méthode `run()` trie les objets que le tableau contient. Le tri se fait selon une relation d'ordre définie par la méthode `compare()` qui défini la relation d'ordre entre deux objets. Le tri des objets peut être ascendant ou descendant selon la valeur de l'attribut `ascending`. Les classes filles spécialisent le fonctionnement des méthodes `extractArray` et `compare`.

La méthode `run()` est décrite par le pseudo-code suivant :

```
void run() {
   JSONArray objectArray = this.extractArray() ;
   JSONArray resultArray;
   resultArray = sort(objectArray) ;[38]
}
```

5.3.2 Activités composites

Les classes des activités composites sont données dans la Figure 48. Elles étendent la classe Activity, et spécialisent le fonctionnement de la méthode `run()`.

Dans les sous-sections suivantes, nous présentons les classes permettant de décrire des activités composites, et comment elles spécialisent la méthode `run()`. Nous présentons également des classes concrètes définies lors la phase de l'implantation du mashup ItineraryPlanner. Ensuite nous allons montrer comment des instances de ces classes interagissent entre elles.

[38] Nous considérons que la méthode `sort(JSONArray a)` implante un des algorithmes de tri proposés dans l'état de l'art [97].

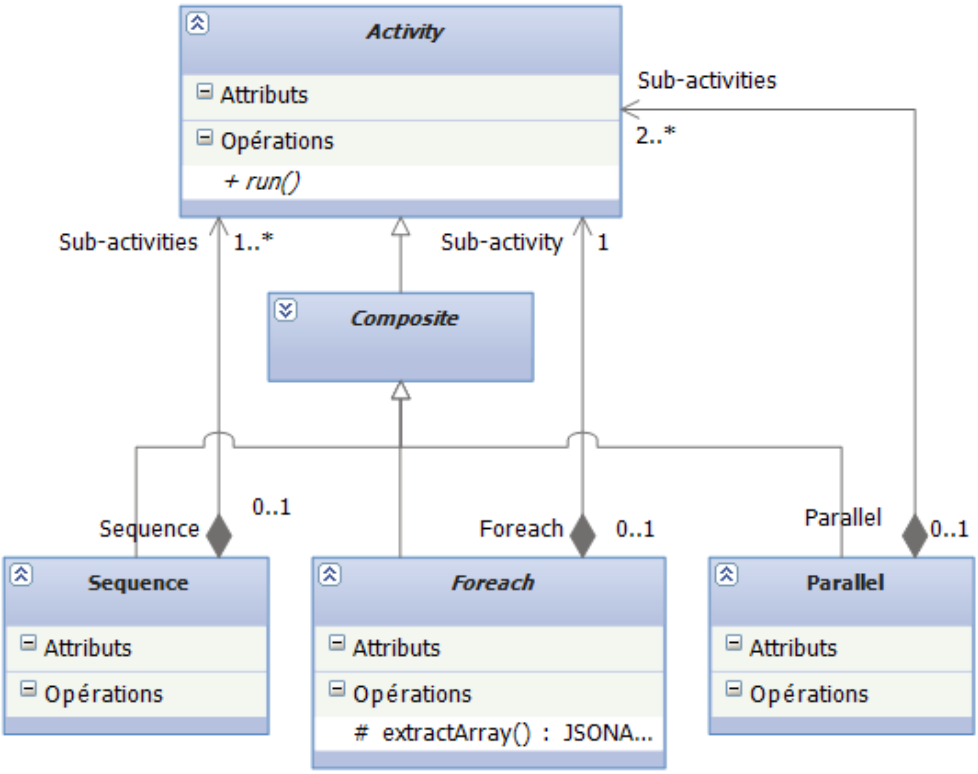

Figure 48 : Les classes des activités composites

5.3.2.1 Sequence

La classe Sequence est une clase concrète en association avec la classe Activity. Cette association indique qu'une instance de la classe Sequence contient un ensemble ordonné d'activités appelées sous-activités (**Sub-activities**).

L'exécution d'une instance `iseq` (Figure 49) de la classe Sequence (ayant deux sous-activités) lance l'exécution des sous-activités séquentiellement. Chacune des sous-activités transmet sa sortie à la sous-activité suivante. La dernière sous-activité transmet sa sortie à la sortie de `iseq`.

La méthode `run()` de l'activité Sequence correspond donc au pseudo-code suivant :

```
run() {
   Activity a = this.sub-activities[0] ;
   a.input = this.input ;
   a.run() ;

   for (int i=1 ; i<this.sub-activities.length() ; i++) {
      sub-activities[i].input = a.output ;
      a = sub-activities[i] ;
      a.run() ;
   }

   this.output = a.output ;
}
```

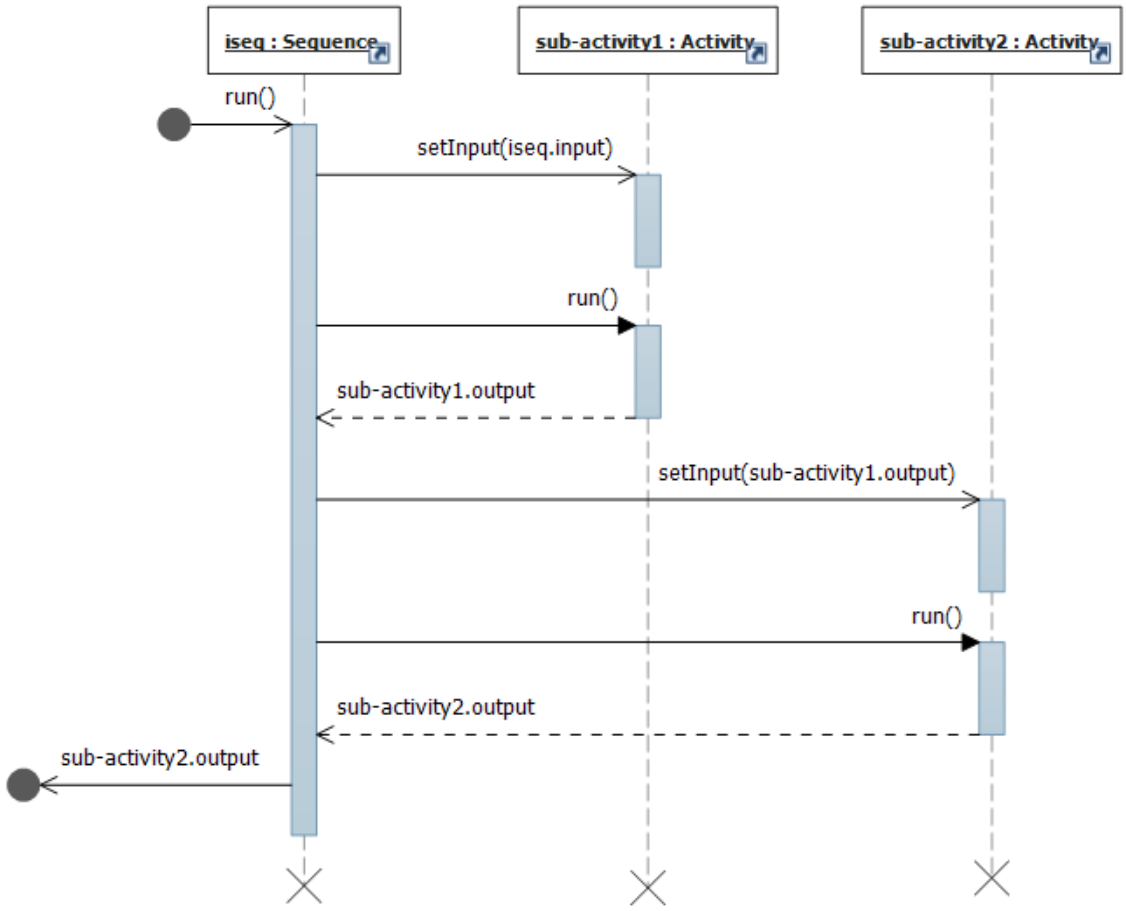

Figure 49 : Exécution de la méthode run() d'une instance de la classe Sequence

Lors de l'implantation du mashup ItineraryPlanner, le constructeur définit une classe concrète WOEIDCitySeq. Une instance, de cette classe, contient deux sous activités :

- Une instance d'une classe concrète RetrieveWOEID qui étend la classe Retrieve. Cette instance appelle un service pour accomplir le mapping entre le format (longitude, latitude) et WOEID. Dans l'exemple, l'utilisation du *Store* est omise (elle ne fait pas partie de l'objectif de l'exemple). Elle sera illustrée dans la sous-section 5.11.
- Une instance d'une classe concrète ExtractWOEID qui étend la classe Extract. Cette instance extrait la valeur de l'identifiant WOEID.

La Figure 50 montre un diagramme de séquence illustrant l'exécution de la méthode `run()` d'une instance de WOEIDCitySeq et ses interactions avec les instances de ses sous-activités. La valeur de l'attribut input de l'instance est définie à la première valeur du tableau step[39] (steps[0]). L'exécution de la méthode `run()` s'exécute selon les étapes suivantes :

[39] Présenté dans le document JSON RouteGP dans l'Annexe B

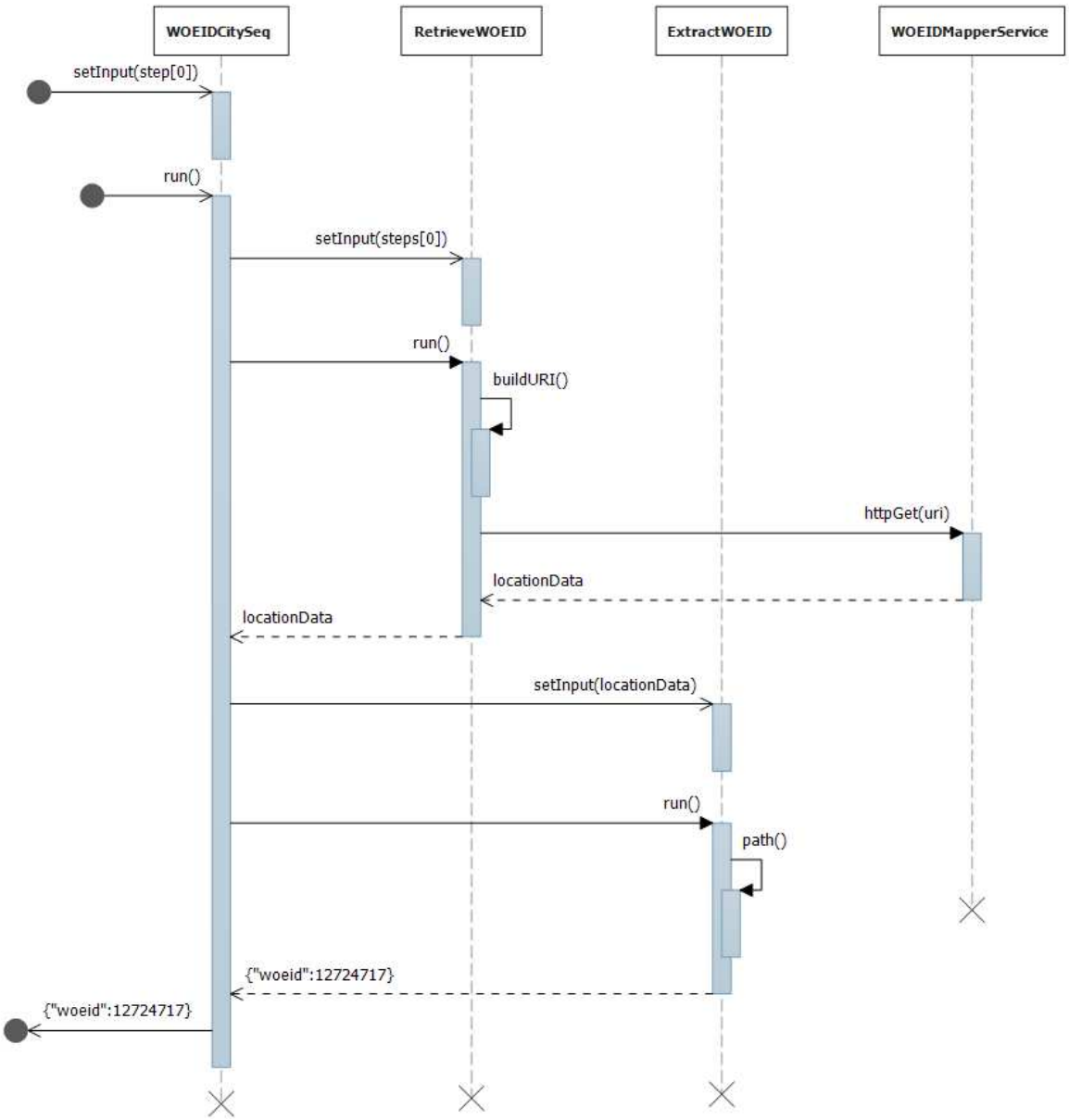

Figure 50 : Exécution de la méthode run() d'une instance de WOEIDCitySeq

- Elle délivre la valeur step[0] à l'instance de sa première sous-activité RetrieveWOEID avec la méthode `setInput()`.
- Elle lance l'exécution de la méthode `run()` de l'instance de RetrieveWOEID. Celle ci construit l'`uri` (avec la méthode `buildURI()`) correspondante à la paire longitude/latitude du lieu d'arrivée de la première étape. Et elle appelle le service[40] WOEIDMapperService pour récupérer la ressource contenant les différentes descriptions géographiques du lieu (locationData).
- Elle transmet la ressource locationData à l'instance de sa deuxième sous-activité ExtractWOEID et lance sa méthode `run()`. Celle ci extrait (`path()`) la valeur de la paire nommée woeid {"woeid" : 12724712} et la transmet à l'instance de WOEIDCitySeq
- Elle définit l'attribut output à {"woeid" : 12724712}.

[40] Yahoo ! GeoPlanet web service : http://developer.yahoo.com/geo/geoplanet/guide/index.html

5.3.2.2 Foreach

La classe Foreach est une classe abstraite en association la classe Activity. Elle contient une méthode abstraite `extractArray()`. Une instance de la classe Foreach reçoit en entrée un document JSON contenant un tableau. Celui ci est extrait avec la méthode `extractArray()`.

L'exécution de la méthode `run()` (Figure 51) commence par (1) exécuter la méthode `extractArrays()` pour extraire le tableau foreachArray d'objets JSON. Ensuite (2), elle lance l'exécution de la sous-activité sur chacun des objets du tableau, et retourne les résultats dans un tableau encapsulé dans un document JSON en sortie.

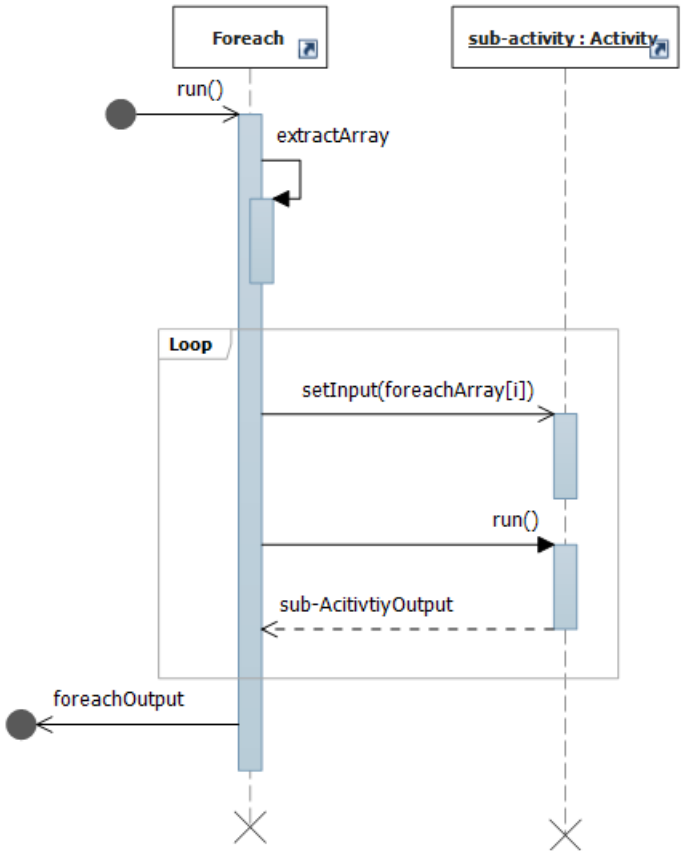

Figure 51 : Exécution de la méthode run() d'une instance de la classe Foreach

La méthode `run()` de la classe Foreach est décrite par le pseudo-code suivant :

```
run() {
   JSONArray foreachArray = this.extractArray ;
   JSONArray result ;
   Activity a = this.subActivity ;
   for int i=0 ; i< foreachArray.length() ; i++ {
      a.input = foreachArray[i] ;
      a.run() ;
      result.put(a.output) ;
   }
this.output = {"array" : result} ;
}
```

Les classes filles spécialisent le fonctionnement de la méthode `extractArray()`. Par exemple, lors de l'implantation du mashup ItineraryPlanner, le constructeur définit une classe

concrète CitiesListForeach avec la classe WOEIDCitySeq comme sous-activité. La méthode `extractArray()` de la classe CitiesListForeach est décrite avec le pseudo-code suivant :

```
JSONArray extractArray() {
   return (this.input.getJSONArray("steps"))
}
```

La Figure 52 montre un diagramme de séquence illustrant l'exécution de la méthode `run()` d'une instance de CitiesListForeach.

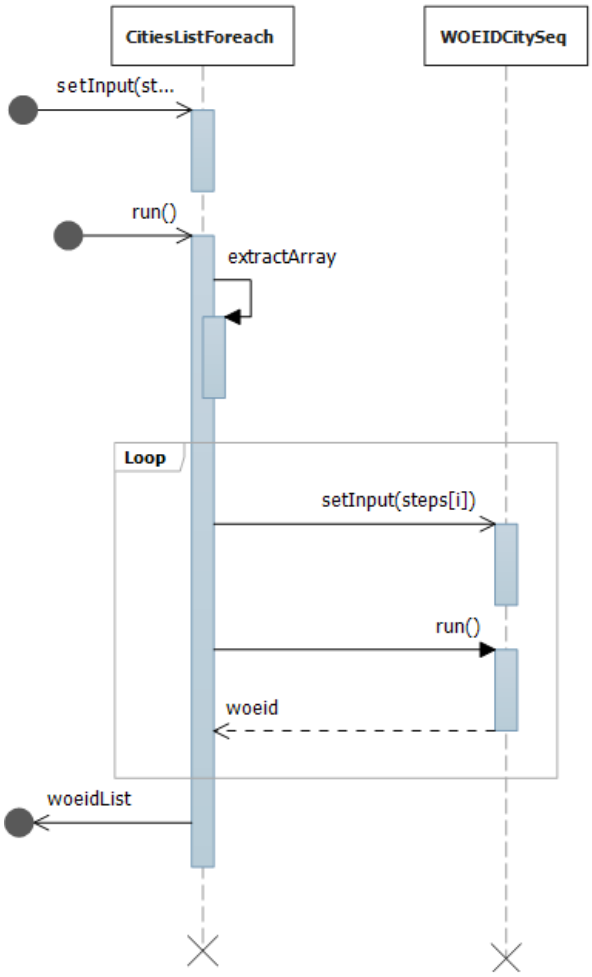

Figure 52 : Exécution de la méthode run() d'une instance de CitiesListForeach

Elle extrait le tableau JSON nommé "steps" avec la méthode `extractArray()`. Ensuite et par le biais d'une boucle loop, elle transmet, à une instance de la sous-activité WOEIDCitySeq, chacune des valeurs (**steps[i]**) du tableau et lance sa méthode `run()`. Ensuite la valeur de l'attribut `output` de l'instance de CitiesListForeach est définie comme étant une ressource dont le document contient un tableau de villes de passage décrites sous format WOEID.

5.3.2.3 Parallel

La classe Parallel est une classe concrète association avec la classe Activity. Cette association indique qu'une instance de la classe Parallel contient trois instances de la classe Activity considérées comme des sous-activités. La Figure 53 illustre les interactions entre une instance de Parallel et ses sous-activités. Cette instance transmet son entrée à ses deux premières sous-activités

et lance en parallèle leurs exécutions. Finalement les sorties de ces deux sous-activités, sont traitées par la troisième sous-activité qui produit la sortie de l'activité Parallel.

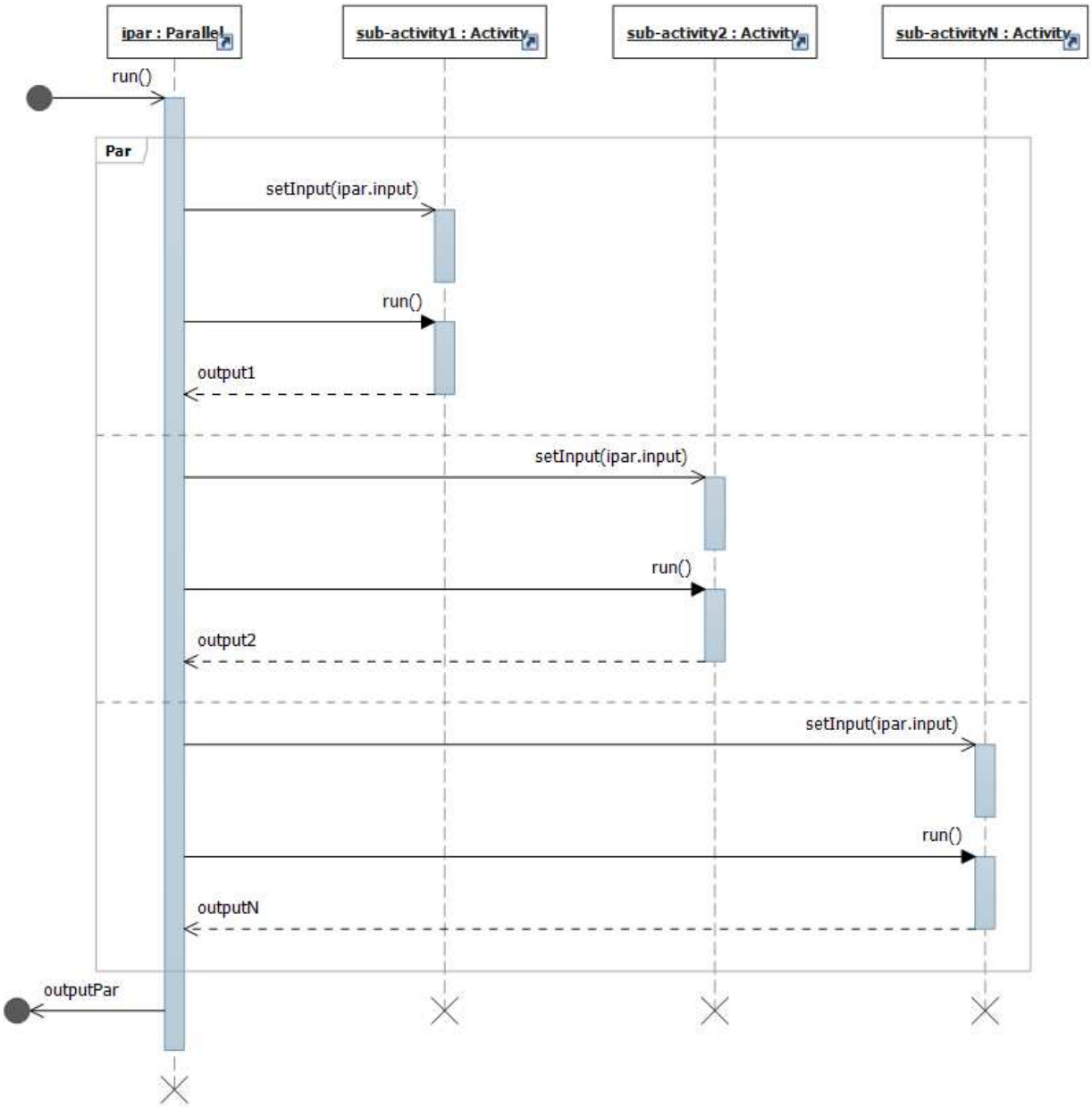

Figure 53 : Interactions entre une activité Parallel et ses sous-activités

La méthode `run()` de la classe Parallel est décrite par le pseudo-code suivant :

```
run() {
   Foreach (a in this.sub-activities) do in parallel {
      a.input = this.input ;
      a.run() ;
      this.output.put(a.output)
   }
}
```

5.4 Mashlet

La Figure 54 illustre la classe Mashlet. Cette classe est abstraite. Elle possède deux attributs `input` et `output` de type JSONObject. Elle est en association avec la classe Activity. Elle possède deux méthodes. La première, `run()`, lance le processus d'exécution de l'activité correspondante. Tandis que la seconde, `render()` est abstraite. Elle est spécialisée par les classes filles de Mashlet pour définir une visualisation des données produites.

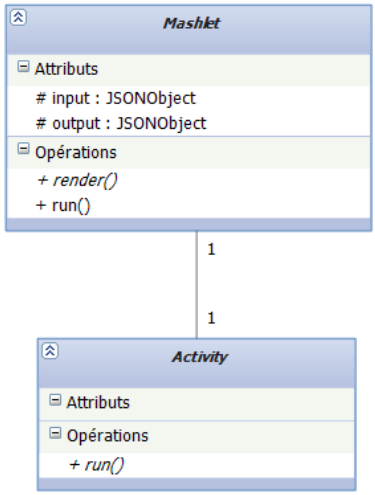

Figure 54 : Classe Mashlet

La Figure 55 montre un diagramme de séquence illustrant l'exécution de la méthode `run()` d'une instance map de la classe Mashlet. Elle définit l'attribut input (`{"depart" : "Grenoble", "destination" : "Paris"}`) de son activité et lance son exécution.

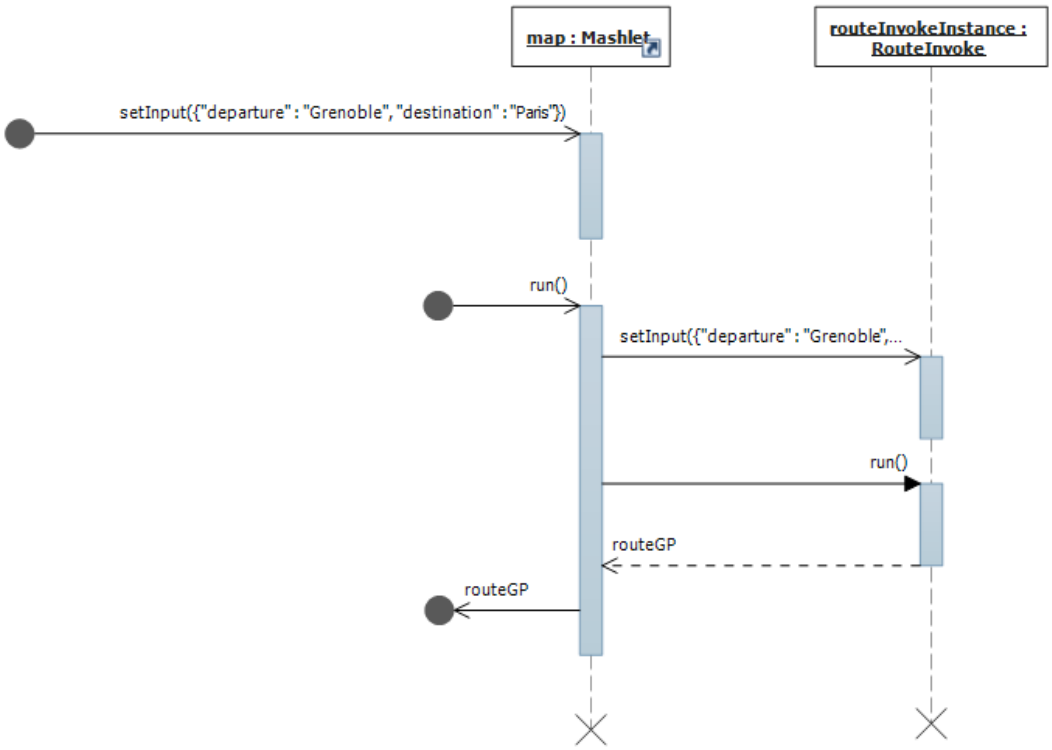

Figure 55 : Interaction entre une instance de Mashlet et son activité

5.5 Wiring

La Figure 56, montre la classe Wiring. Elle est concrète. Elle contient deux attributs `input` et `output` de type JSONObject et une méthode `run()`. Deux associations avec la classe Wiring spécifient qu'une instance de Wiring interagit avec deux instances de Mashlet : l'envoyeur (en amont du wiring) et le receveur (en aval du wiring). La classe Wiring est aussi, en association avec la classe Activity.

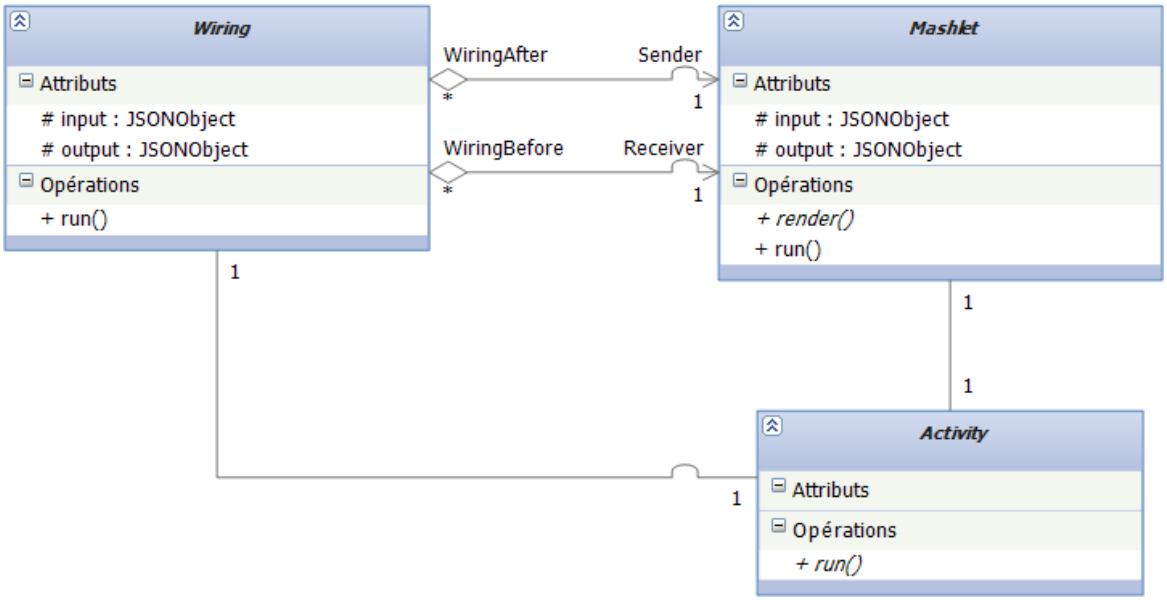

Figure 56 : Classe Wiring

Une instance de Wiring prend les données du mashlet envoyeur, elle les manipule (en exécutant un processus de mapping en général) pour envoyer le résultat au mashlet receveur. La manipulation de données se fait avec l'activité du wiring via la méthode `run()`.

5.6 Mashup

La classe Mashup (Figure 57) est abstraite. Elle contient un attribut `input` de type JSONObject qui représente les entrées de l'utilisateur. Elle contient aussi deux méthodes abstraites : `defineMashupComponents()`, `setMashletsInputs()` et deux méthodes concrètes : `prepareMashup()` et `run()`. La classe est en association avec les classes Mashlet et Wiring : une instance de Mashup contient un ensemble d'instances de Mashlet et les instances de Wiring qui définissent la coordination entre ces mashlets.

Une classe fille de Mashup définit l'implantation d'un mashup répondant à un certain besoin. Elle spécialise les méthodes abstraites suivantes :

- La méthode `setMashletsInputs()` définit la distribution des valeurs contenues dans l'attribut input d'une instance de Mashup aux mashlets concernés.
- La méthode `defineMashupComponents()` définit les instances de mashlets et de wiring que le mashup doit contenir.

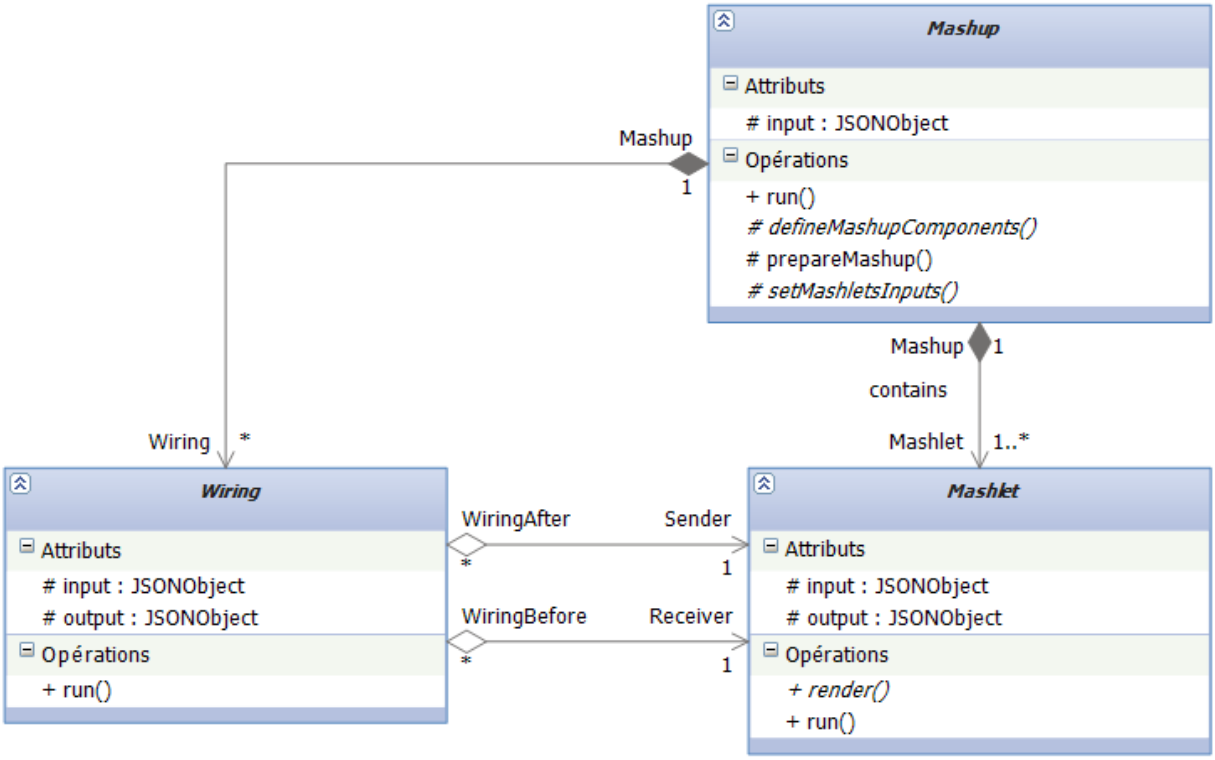

Figure 57 : Classe Mashup

La méthode `run()` (cf. le pseudo-code ci dessous) lance l'exécution de l'ensemble de mashlets et wirings. Elle commence par construire une liste `ready` (avec la méthode `prepareMashup()`) qui contient au début les mashlets qui ne sont pas précédés par un wiring. Ensuite avec une boucle while la fonction exécute les composants (mashlets et wirings) présents dans `ready`. Lors de l'exécution d'un mashlet, les wirings, qui récupèrent sa sortie, notifiés et ajoutés à la liste `ready`. Et lors de l'exécution d'un wiring, le mashlet, qui récupère sa sortie, est notifié et ajouté à la liste `ready`.

```
run() {
   List<object> ready ;
   this.setMashletsInputs() ;

   //ajouter les mashlets, prêts à l'exécution, à ready
   ready = prepareMashup() ;
   while (not ready.empty()){
      object o = ready.pop() ;
      if type(o) = Mashlet {
         o.run() ;
         foreach w in o.wiringAfter do {
            notify (w) ;
            ready.put(w) ;
         }
      }

      else {// o est de type Wiring
         o.run() ;
         Mashlet m = o.receiver ;
         notify (m) ;
         ready.put(m) ;
      }
}
```

5.7 Item

La classe Item est présentée dans la Figure 58. Elle contient les attributs `uri` de type chaine de caractère, `result` de type JSONObject et des attributs donnant des informations statistiques :

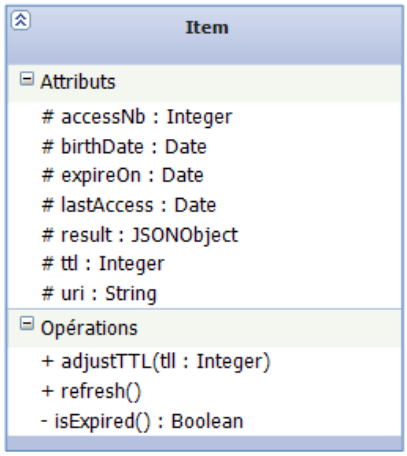

Figure 58 : Classe Item

- `accessNb` de type entier, il indique le nombre d'accès à l'item.
- `ttl` de type entier, il indique la durée de vie de l'item.
- `birthDate` de type Date, il indique la date de création de l'item.
- `expireOn` de type Date, il indique la date d'expiration de l'item.
- `lastAccess` de type Date, indique la date du dernier accès à l'item.

La classe contient également trois méthodes : la première `refresh()` permet de rafraichir la valeur de l'attribut `result` en récupérant les données depuis le fournisseur via la valeur de l'attribut `uri`. Alors que la seconde `adjustTTL(ttl : Integer)` permet de modifier la durée de vie `ttl` d'un item au sein du cache. Enfin, la méthode `isExpired()` indique si la donnée enregistrée dans l'attribut `result` est périmée ou pas.

La Figure 59 montre un exemple d'une instance de la classe. Elle correspond au document JSON `routeGP` défini dans l'Annexe B et récupéré avec l'uri `uriGP` où :

```
uriGP =
"http://maps.googleapis.com/maps/api/directions/json?origin="+departure+"&des
tination="+destination+"&sensor=false"
```

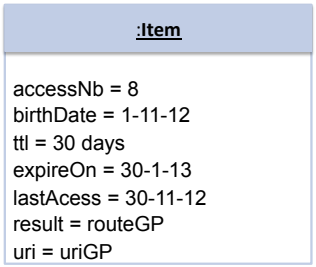

Figure 59 : Instance de la classe Item

5.8 Store

Les fonctionnalités de gestion des items sont définies dans la classe Store (Figure 60). Celle ci est en association avec la classe Item. Cette association indique qu'une instance de la classe Store contient un ensemble d'items. Elle contient un attribut `length` de type entier définissant la capacité du *Store*.

La classe Store contient aussi les méthodes `lookup(uri : String)`, `bind(uri : String, result : JSONObject, ttl : Integer)` et `unbind(uri : String)`.

- La méthode `lookup(uri : String)` permet de vérifier l'existence d'un item, ayant l'identifiant `uri`, au sein du Store. Le résultat de cette méthode est l'item correspondant, s'il est présent, ou `null` sinon. Si l'item est périmé, la méthode demande au gestionnaire de rafraichissement de le rafraichir avant de le retourner.
- La méthode `bind(uri : String, result : JSONObject, ttl : Integer)` permet de construire un item ayant l'identifiant `uri`, la donnée `result` et une durée de vie `ttl`. Et elle ajoute cet item au Store.
- La méthode `unbind(uri : String)` permet de supprimer du Store l'item dont l'identifiant possède la valeur de `uri`.

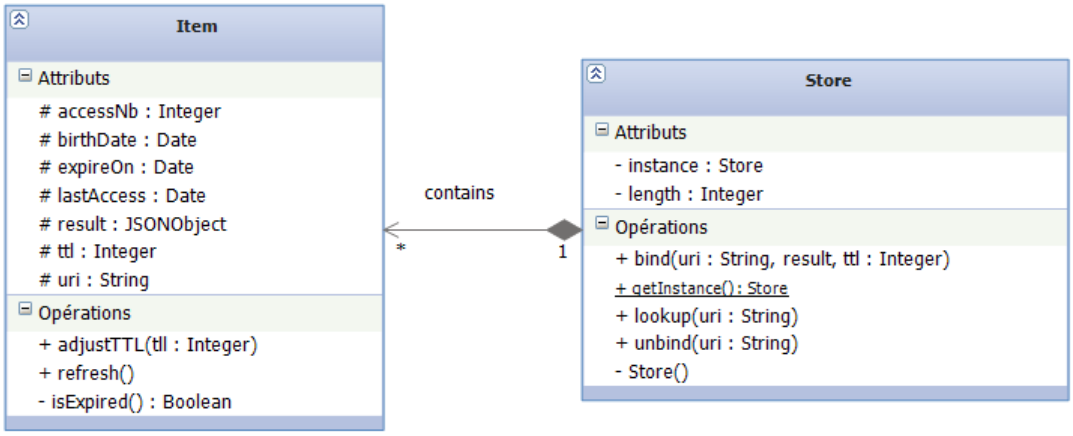

Figure 60 : Classe Store

La classe Store est spécifiée avec un patron de conception[41] singleton : c'est à dire que son instanciation est restreinte à un seul objet. En effet, dans notre architecture, il y a un seul cache pour stocker les données provenant des services de données. Lors de la poursuite de ces travaux pour les étendre aux caches distribués pour les mashups, le patron de conception singleton sera écarté. Pour s'assurer de l'unicité de l'instance, dans la représentation de diagramme de classes dans la notation UML, les constructeurs de la classe sont définis avec une visibilité privée. Pour permettre l'accès à l'instance unique on crée une méthode `getInstance()` qui permet de la créer, si elle n'existe pas encore, et de la retourner.

5.9 ReplacementManager

La classe ReplacementManager est présentée dans la Figure 61. Elle implante les méthodes de l'interface ReplacementManager de ACS. Comme dans des implantations fournies

[41] Un patron de conception (design pattern) est la description d'un problème logiciel et des éléments d'une solution à ce problème [98]

dans ACS, La méthode `addForReplacement` ajoute une référence à l'item dans une liste ordonnée gérant les items éligibles lors d'un remplacement. L'ordre des éléments est défini selon les poids des items[42]. Les deux méthodes `addForReplacement` et `adjustForReplacement` réalisent des actions pour mettre à jour le poids de l'item utilisé par la politique de remplacement.

La classe ReplacementManager est abstraite : elle contient une méthode abstraite `itemWeight(Item e)`. Celle ci est spécialisée par des classes filles. Son objectif est de définir le poids de l'item. Lors de l'éviction des données, les items, dont les poids sont les moins élevés, seront supprimées. Les coefficients de la formule de poids sont définis comme des attributs des classes filles et sont modifiables avec des méthodes *setter*. La possibilité de modifier ces attributs donne au gestionnaire une caractéristique d'adaptabilité.

La classe ReplacementManager est en association avec la classe Store. Cette association indique que l'instance unique de Store possède une instance de ReplacementManager pour gérer la politique de remplacement dans MELQART.

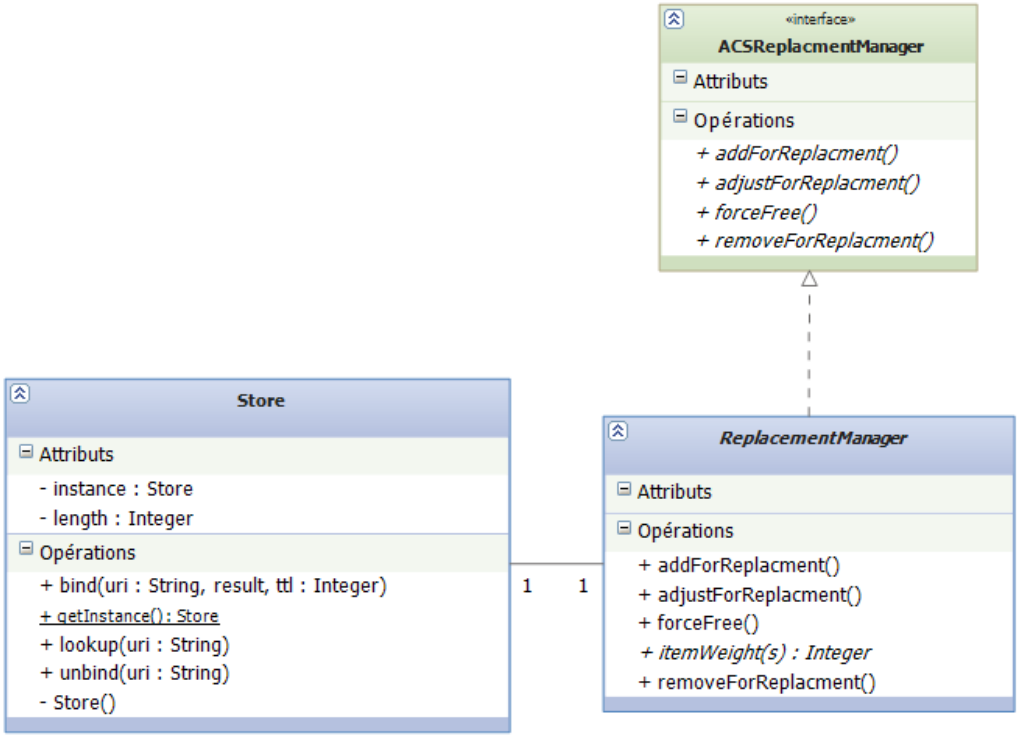

Figure 61 : La classe ReplacementManager

5.10 FreshnessManager

La Figure 62 présente la classe FreshnessManager. Elle possède les attributs `refreshFresq`, `ttlMin`, `ttlMax` et `itemsRefs` et les méthodes `run()`, `addForFreshness()`, `removeForFreshness()` et les méthodes abstraites `defineRefreshFreq()`, `increment()` et `decrement()`.

[42] Dans l'implantation des classes d'ACS, les items (entrées de cache) sont ordonnés selon une donnée temporelle.

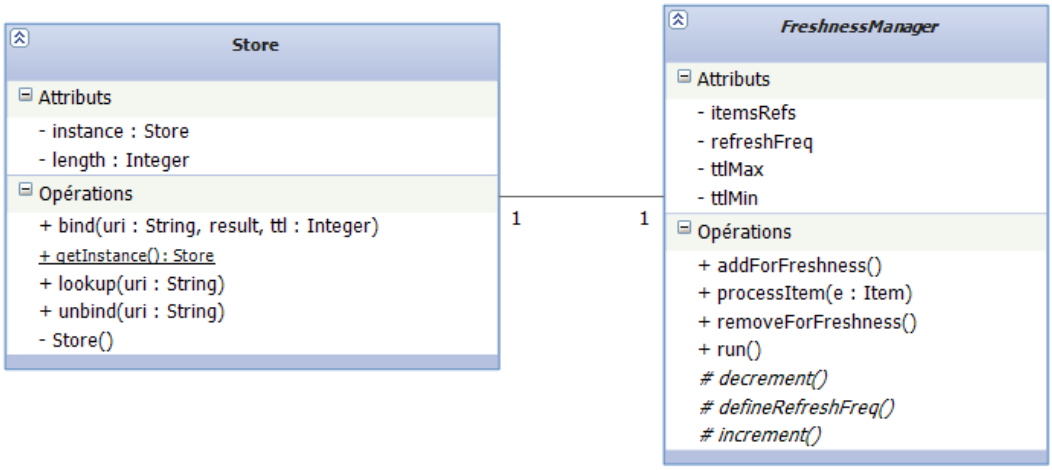

Figure 62 : La classe FreshnessManager

La classe FreshnessManager est en association avec la classe Store qui précise que l'instance unique de Store possède une instance de FreshnessManager pour gérer le rafraichissement des items.

La méthode `defineRefreshFreq` est une méthode abstraite. Elle est spécialisée par des classes filles de la classe FreshnessManager. Elle définit la valeur de l'attribut `refreshFreq`. Celui ci indique la fréquence d'exécution de la méthode `run()`. Tandis que l'attribut `itemsRefs` est de type arbre rouge et noir. Les attributs `ttlMin` et `ttlMax` définissent les valeurs minimales et maximales que peuvent avoir les durées de vies des items T_{min} et T_{max}.

Les méthodes `increment` et `decrement` sont des méthodes abstraites. Elles sont spécialisées par des classes filles de la classe FreshnessManager. Elles définissent les valeurs à ajouter à (ou à retirer de la valeur de TTL d'un item. Elles prennent en paramètres : la valeur de TTL, le nombre d'accès à l'item et la date du dernier accès à l'entrés. Par exemple, une classe fille de FreshnessManager peut implanter les méthodes `increment` et `decrement` selon les formules définies dans les sous-sections 4.2.4.2(a) et 4.2.4.2(b). Les paramètres α, β, γ *et* δ sont alors définis dans des attributs. Ils sont modifiables par des méthodes *setter*.

La méthode `addForFreshness` est appelée par le Store pour signaler au gestionnaire de rafraichissement l'ajout d'un nouvel item au Store (par le biais de la méthode `bind`). Une référence à cet item est ajoutée dans `itemsRefs`.

La méthode `removeForFreshness` prend en comme paramètre un identifiant d'un item. Elle est invoquée par le Store pour signaler au gestionnaire de rafraichissement la suppression de l'item correspondant depuis le Store (par le biais de la méthode `unbind`). La référence correspondante est supprimée depuis `itemsRefs`.

La méthode `processItem` prend en paramètres un item dont la durée de vie est expirée. Elle exécute le processus de rafraichissement selon la stratégie TTL incrémental définie dans la sous-section 4.2.4.2.

La méthode `run()` parcourt les premiers éléments de la liste ordonnée `itemsRefs` et les met à jour. Elle est lancée périodiquement, selon la valeur de l'attribut `refreshFreq`. Elle fonctionne selon le pseudo code suivant :

```
run() {
   int i = 0 ;
   Item  it= itemsRefs[i].getItem() ;
   while (it.isExpired() == true) {
      processItem(it) ;
      i++ ;
      it= itemsRefs[i].getItem ;
   }
}
```

5.11 Interaction des mashups avec le Store

L'interaction entre les mashups et le Store se déroule au niveau des activités de type RetrievePP. La méthode `run()` fait appel aux méthodes de la classe Store. L'exécution de la méthode est illustrée dans le diagramme de séquence de la Figure 63. Elle commence par construire l'uri, et vérifier si le Store contient un item ayant un identifiant la valeur de l'attribut uri (avec la méthode `lookup(uri)`) et si oui, Elle retourne le résultat stocké dans le Store. Sinon, elle récupère les données auprès du fournisseur, les convertit en format JSON si nécessaire et les stocke dans le Store (`bind(uri, result, ttl)`). Nous rappelons que la méthode `lookup` vérifie l'état de l'item avant de la retourner : si elle est déjà périmée, la méthode demande au gestionnaire de rafraichissement de rafraichir l'item (avec la méthode `processItem()`).

La méthode `run()` de la classe RetrievePP est décrite avec le pseudo-code suivant

```
void run() {
   this.buildURI() ;

   Store  st= Store.getInstance() ;
   Item it = st.lookup(this.uri) ;

   if (it != null) {
      this.output = it.getResult();
   }
   else {
      String data= getData(uri) ;
      // conversion du résultat en JSON
      if ( this.isJSON()){
         this.output = new JSONObject(data)
      }
      else {
         this.output = XML.toJSONObject(data)
      }
      st.bind(this.uri, this.output, this.ttl) ;
   }
}
```

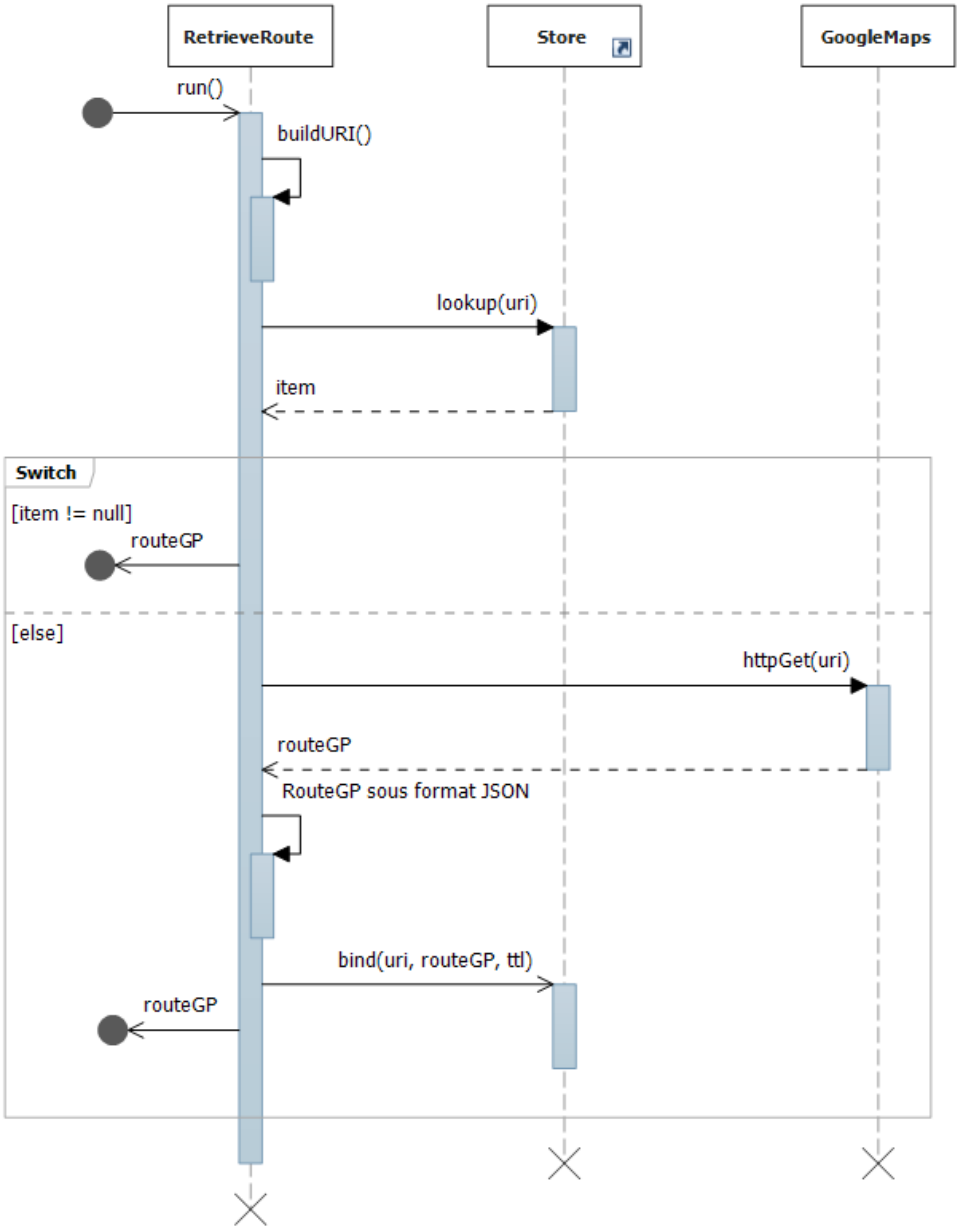

Figure 63 : Exécution de la méthode run() de l'activité Retrieve avec utilisation du Store

5.12 Validation

Cette section présente la validation de notre contribution. Elle a pour objectif de démontrer que les fonctionnalités que nous avons proposées et implantées avec un cache adapté au contexte des mashups répondent au problème de la non disponibilité de données dans les mashups.

Nous avons implanté un prototype de MELQART avec le langage Java dans un environnement Eclipse[43]. Le cache de données de mashups **storeM** est implanté en tant qu'une instance de la classe Store. Il associée à toutes les instances de mashups avec une taille de 1000 items (dans ACS, le prototype de cache est implanté avec un gestionnaire de taille de cache basé

[43] http://www.eclipse.org/

sur le nombre des items). Nous avons associé à **storeM** des gestionnaires de remplacement et de rafraichissement **rm** et **fm** exécutant les politiques données en exemple dans le Chapitre 4. La fréquence de rafraichissement est fixée à deux heures. Nous soulignons que l'objectif de cette validation n'est pas de démontrer que telle politique (de remplacement ou de rafraichissement) est meilleure qu'une autre ou de trouver la meilleur politique (de remplacement ou de rafraichissement).

Pour valider notre implantation, nous avons défini deux scenarios de mashups ItineraryPlanner (sous-section 5.12.1) et MyDashboard (sous-section 5.12.2). L'exécution de plusieurs instances de ces mashups pour valider notre approche et notre implantation. Nous ne nous sommes pas intéressé à la visualisation des données. En effet, la visualisation ne constitue pas un aspect nécessaire de la validation de notre approche pour améliorer la disponibilité de données dans les mashups. Les données des mashlets sont présentées sous la forme d'objets JSON dans des widgets.

5.12.1 ItineraryPlanner

Nous avons implanté le mashup ItineraryPlanner avec les mashlets Map et Weather. Ce mashup a servi de scenario pour illustrer notre approche. Son fonctionnement fut décrit tout au long de ce document. Le corps de la méthode `defineMashupComponents()` de la classe ItineraryPlanner et définissant la logique applicative du mashup est présenté dans le code JAVA commenté suivant :

```
public void defineMashupComponents() {
        ////Mashlet Map
        RetrievePP rt = new RouteRetrieve();
        Mashlet map = new Mashlet("map",rt);
        this.addMashlet(map.getName(),map);

        ////Mashlet Weather
        WeatherRetrieve wr = new WeatherRetrieve();
        WeatherExtractor extractWeatherAct = new WeatherExtractor();

        // Sequence
        Sequence weatherSeq = new Sequence();
        weatherSeq.addActivity(wr);
        weatherSeq.addActivity(extractWeatherAct);
        // Foreach
        CitiesWeatherForeach itpWeatherForeach = new CitiesWeatherForeach();
        itpWeatherForeach.setSubActivity(weatherSeq);

        Mashlet weather = new Mashlet("weather", itpWeatherForeach);
        this.addMashlet(weather.getName(),weather);

        ////Wiring Map2Wiring
        //extraction des villes de passage
        StepsExtractor stepsExtO = new StepsExtractor();

        // conversion au format WOEID
        RetrieveWOEIDPlace rw = new RetrieveWOEIDPlace();
        WoeidPlaceExtractor woeidExtractorAct = new WoeidPlaceExtractor();
        Sequence woeidSeq = new Sequence();
        woeidSeq.addActivity(rw);
        woeidSeq.addActivity(woeidExtractorAct);
        CitiesListForeach citiesListForeachAct = new CitiesListForeach();
        citiesListForeachAct.setSubActivity(woeidSeq);

        // élimination des doublons WOEID
        UniqueWOEID uniqueWOEIDAct = new UniqueWOEID();

        // mise en séquence
        Sequence stepsSeq = new Sequence();
```

```
        stepsSeq.addActivity(stepsExtO);
        stepsSeq.addActivity(citiesListForeachAct);
        stepsSeq.addActivity(uniqueWOEIDAct);

        Wiring map2weather = new Wiring("map2weather", map,weather, stepsSeq);
        this.addWiring(map2weather.getName(), map2weather);
    }
```

La Figure 64 présente les résultats du mashup pour un itinéraire allant de Grenoble jusqu'à Paris.

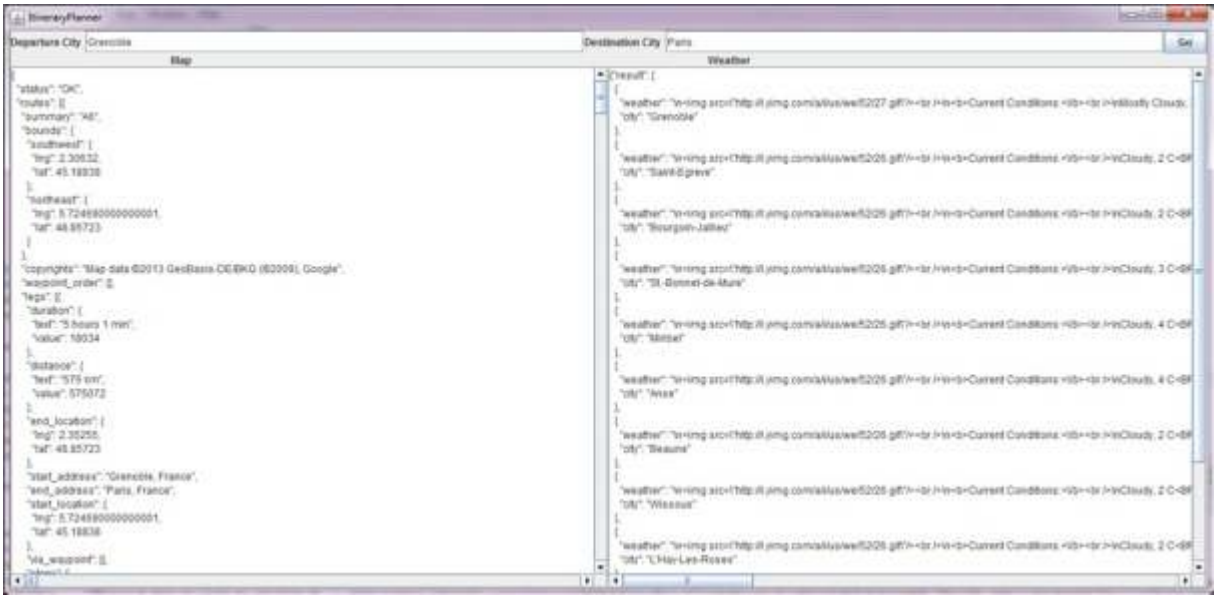

Figure 64 : Résultats du mashup ItinerayPlanner pour l'itinéraire de Grenoble à Paris

Le mashup a deux entrées : la ville de départ et la ville d'arrivées. Toutes les deux sont de type chaine de caractères. Les données, du mashlet maps, proviennent du service Google maps[44]. Alors que les données du mashlet weather proviennent du service Yahoo! Weather[45]. Un mapping est nécessaire pour transformer les descriptions des villes de passage du format (longitude, latitude) au format WOEID. Ce mapping est fait via le service Yahoo! BOSS Geo Services[46]. Le Tableau 6 donne les durées de vie que nous avons estimées pour les données récupérées de chaque service.

Fournisseur des données	**Durée de vie des données**
Google maps	30 jours
Yahoo! Weather	5 heures
Yahoo! Boss	60 jours

Tableau 6 : Durées des vies des données du mashup ItineraryPlanner

[44] https://developers.google.com/maps/documentation/directions/
[45] http://developer.yahoo.com/weather/
[46] http://developer.yahoo.com/boss/geo/

5.12.2 MyDashboard

Le deuxième mashup est nommé MyDashboard. Il contient deux mashlets. Un mashlet « Weather » qui donne la météo pour une ville saisie par l'utilisateur et un Mashlet « Programmes TV » qui affichent les programmes TV de différentes chaines de la soirée. Le corps de la méthode `defineMashupComponents()` de la classe MyDashboard et définissant la logique applicative du mashup définissant son implantation est présenté dans le code JAVA commenté suivant :

```
public void defineMashupComponents() {

        ////Mashlet Weahter
        RetrieveWOEIDPlace rwc = new RetrieveWOEIDPlace();
        WoeidPlaceExtractor ewc = new WoeidPlaceExtractor();
        WeatherRetrieve wr = new WeatherRetrieve();

        Sequence ws = new Sequence();
        ws.addActivity(rwc);
        ws.addActivity(ewc);
        ws.addActivity(wr);

        Mashlet weather = new Mashlet("weather",ws);
        this.addMashlet(weather.getName(),weather);

        ////Mashlet TV
        TVProgRetrieve  tvp = new TVProgRetrieve();
        Mashlet myprg = new Mashlet("myprg",tvp);
        this.addMashlet(myprg.getName(),myprg);

}
```

La Figure 64 présente les résultats du mashup MyDashboard avec Grenoble comme paramètre en entrée.

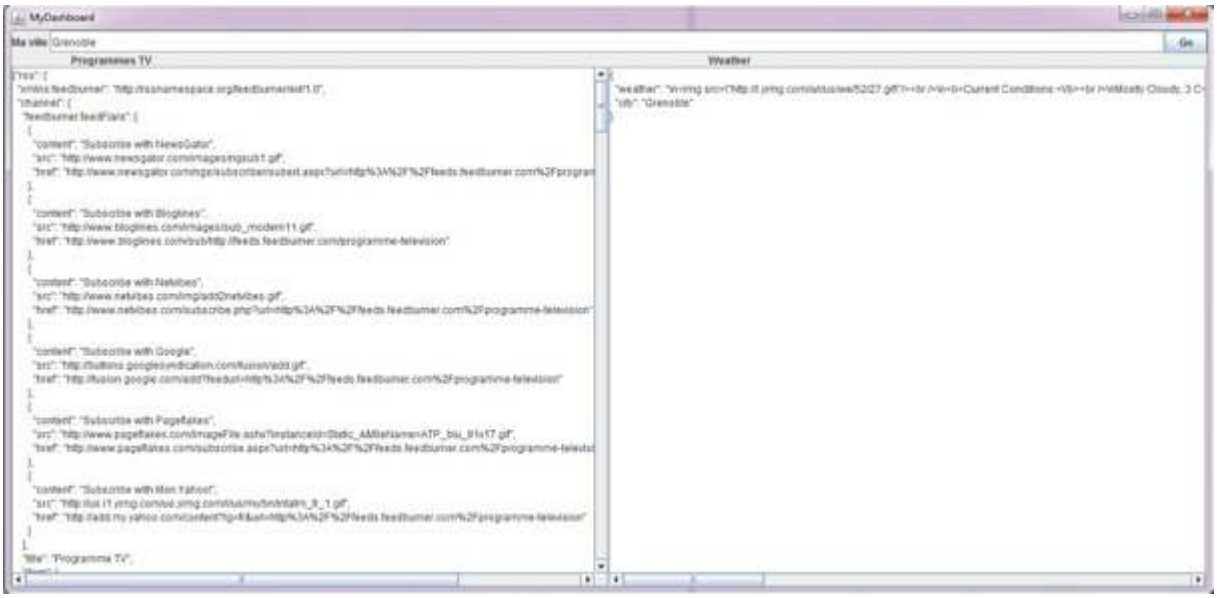

Figure 65 : Résultats du mashup MyDashboard avec Grenoble comme paramètre en entrée.

Le mashup possède une seule entrée : la ville de l'utilisateur de type chaine de caractères. Les données du mashlet « Programmes TV » proviennent d'un flux RSS[47]. Alors que les données du

[47] http://feeds.feedburner.com/programme-television

mashlet « Weather » proviennent du service Yahoo! Weather. Un mapping est nécessaire pour transformer la description de la ville de l'utilisateur du format « nom de ville » au format WOEID. Ce mapping est fait via le service Yahoo! GeoPlanet[48]. Le Tableau 7 donne les durées de vie que nous avons estimées pour les données récupérées de chaque service.

Fournisseur des données	**Durée de vie des données**
Flux TV	1 jour
Yahoo! Weather	5 heures
Yahoo! GeoPlanet	60 jours

Tableau 7 : Durées des vies des données du mashup MyDashboard

5.12.3 Execution des instances des mashups

Nous avons crée plusieurs instances des mashups ItineraryPlanner et MyDashboard, que nous avons lancées avec plusieurs valeurs en paramètres d'entrée (cf. Figure 66). Les exécutions se sont terminées correctement et ont retourné les résultats attendus.

Par injection d'impressions de traces dans le code, nous avons pu observer la gestion des données des mashups. Les données produites par les activités de type RetrievePP sont bien stockées dans le Store. Ces données sont récupérées directement du Store, lors d'une exécution ultérieure des mêmes activités avec les mêmes valeurs de paramètres d'entrées. Ceci fut observé également avec la rapidité d'exécution des mashups et l'obtention rapide des résultats. En plus, ce qui est le plus important est le fait de pouvoir accéder aux données du Store pour exécuter des instances de mashups même en cas de rupture de communication avec les fournisseurs de données (simulée par une déconnexion de la machine du réseau) ce qui valide notre approche implantée avec un cache pour garantir la disponibilité des données dans les mashups.

Le gestionnaire de remplacement évinçait les items jugés les moins importantes par la politique de remplacement choisie. Le gestionnaire de rafraichissement s'est lancé périodiquement pour vérifier l'état de validités des items. Ainsi, les items, correspondant aux données provenant du service Yahoo! Weather, furent mises à jour pendant la journée de test. En plus, lorsque nous avons lancé le mashup ItineraryPlanner, à un moment où les données météorologiques étaient périmées dans le Store (avant le déclenchement du processus périodique de rafraichissement), le Store demandait au gestionnaire de rafraichissement de rafraichir les items avant de les retourner.

5.13 Conclusion

Ce chapitre a présenté l'implantation d'un prototype de MELQART notre système d'exécution de mashups avec disponibilité de données. L'architecture du système fut présentée en premier. Ensuite, nous avons présenté l'implantation des concepts de mashups et des fonctionnalités proposées pour améliorer la disponibilité de données. Pour implanter ces fonctionnalités, Nous avons adapté des gestionnaires d'ACS au contexte des mashups et nous avons introduit le gestionnaire de rafraichissement. La validation de notre implantation nous a montré que les dites fonctionnalités implantées via un cache constituent une solution crédible pour garantir la disponibilité de données dans les mashups.

[48] http://developer.yahoo.com/geo/geoplanet/guide/index.html

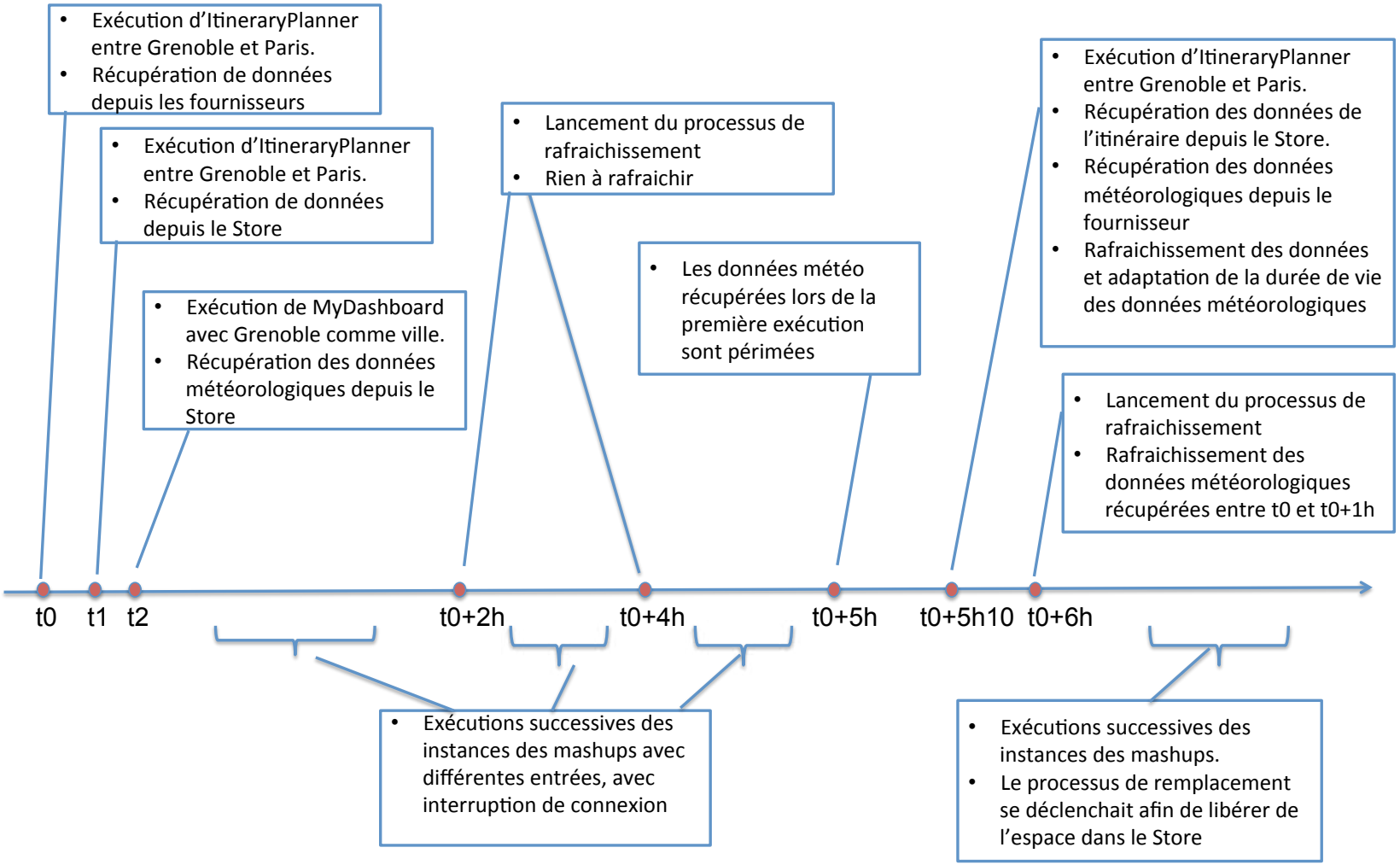

Figure 66 : Chronologie des exécutions

Chapitre 6

CONCLUSION

L'objectif de ce travail était d'améliorer la disponibilité de données fraiches dans les mashups. Nous avons proposé MELQART : un système d'exécution de mashups avec une solution pour améliorer la disponibilité et la fraicheur de données des mashups. Dans la suite nous soulignons nos contributions et les perspectives de notre travail.

6.1 Contribution et bilan

La première contribution de ce travail est un état de l'art sur les modèles des mashups, les systèmes d'exécution de mashups et leurs architectures ainsi que la disponibilité de données fraiches sur le web. Nous avons observé qu'il y a peu de travaux qui proposent des modèles de mashups et que ceux qui existent ne sont pas suffisamment riches pour spécifier les caractéristiques de la disponibilité de données. Nous avons également constaté que les techniques actuelles de disponibilité de données ne sont pas adaptées à des applications comme les mashups. Enfin, parmi le peu de travaux qui se sont intéressés à la disponibilité des données des mashups, le problème de la fraicheur des données disponibles n'est pas abordé.

Nous avons en conséquence proposé MELQART : un système d'exécution de mashups avec une solution pour améliorer la disponibilité et la fraicheur de données des mashups. Cette solution propose des fonctionnalités orthogonales au processus d'exécution de mashups pour améliorer la disponibilité de données récupérées auprès des fournisseurs de données. Dans la suite de cette section, nous présentons nos conclusions au regard de nos contributions dans le cadre de MELQART.

Nous avons défini un modèle de description de mashups basé sur les valeurs complexes. Des activités basiques décrivent la manipulation des données de mashups (le filtrage, la projection, l'extraction, l'élimination de doublons, le tri, l'union et l'annexe). Ces activités sont coordonnées par des activités composites (Séquence, Foreach, Parallel). Le modèle peut être enrichi avec d'autres activités dans le futur selon les besoins qui peuvent apparaître. Elles sont utilisées par des mashlet et des wirings. Les mashlets affichent des données produites par les activités tandis que les wirings coordonnent l'exécution des mashlets. Dans notre modèle, les wirings ne décrivent pas qu'un transfert de données entre mashlets comme c'est le cas dans les modèles existants. Ils peuvent aussi, décrire le traitement des données pour les délivrer sous le bon format aux mashlets destinataires. Notre modèle permet également de spécifier les caractéristiques de la disponibilité et de la fraicheur de données.

Nous avons présenté les phases du processus d'exécution d'un mashup et notre solution pour améliorer la disponibilité de données fraiches des mashups. Celle ci est à base de fonctions orthogonales au processus d'exécution des mashups. Ce processus est basé sur la représentation d'un mashup sous la forme d'un graphe dont le parcours permet de coordonner l'exécution des mashlets et des wirings du mashup. Nous avons décrit les phases des processus d'exécution des mashlets et des wirings et de leurs activités sous-jacentes. Des fonctionnalités permettant l'améliorer la disponibilité des données récupérées auprès des fournisseurs de données et d'assurer leur fraicheur. Les données sont rendues disponibles dans un $Store$. Pour cela, nous avons décrit l'organisation du contenu du $Store$ en un ensemble d'items. Nous avons ensuite décrit la gestion de ce contenu et les fonctions qui y sont relatives : ajouter, modifier, supprimer et rechercher des items. Ensuite, nous avons décrit les fonctions de remplacement et de rafraichissement des items. Pour le remplacement des items, nous avons dressé une liste de facteurs entrant en jeu lors de la définition de la politique de remplacement dans le contexte des mashups. Le rafraichissement des items est exécuté d'une façon incrémentale. Nous avons adopté une stratégie adaptative pour la définition des durées de vie des items ainsi nous avons adapté la stratégie TTL incrémental au contexte des mashups.

Nous avons implanté les concepts de mashups et les fonctionnalités proposées pour améliorer la disponibilité de données. Pour implanter ces fonctionnalités, Nous avons adapté des gestionnaires d'ACS au contexte des mashups et nous y avons introduit le gestionnaire de rafraichissement. La validation de notre implantation, via des tests sur l'exécution de deux scénarios de mashups, nous a montré que les dites fonctionnalités implantées via un cache constituent une solution crédible pour garantir la disponibilité de données dans les mashups.

6.2 Perspectives

Les travaux réalisés dans cette thèse ne sont que le premier pas vers la définition d'un environnement pour la construction et l'exécution de mashups. Des nouveaux défis et améliorations doivent être traités. Les points qui nous apparaissent intéressants pour la suite sont :

- Définir un langage déclaratif permettant à un utilisateur final de définir son mashup. Ce langage peut être aussi bien textuel que graphique. Ceci constitue une amélioration apportée à MELQART en vue d'une évolution vers un environnement de définition et d'exécution de mashups. Actuellement, la définition d'un langage de mashups fait partie des projets RedShine[49] et Clever[50].
- Considérer le cas de figure où les données du $Store$ sont périmées et qu'il est impossible de les rafraichir (dans le cas de rupture de connexion avec le fournisseur de données correspondant). Ceci implique la mise en œuvre d'un traitement d'exceptions et des actions pour palier ces exceptions en assurant un fonctionnement du mashup même détérioré. Une solution possible peut inclure la recherche d'un autre fournisseur de données offrant des données contenant l'information recherchée. Ceci nécessite la mise en place d'un processus de mapping de données pour récupérer des données décrites selon le même format. Ce travail a été démarré dans le cadre des projets Clever et e-Cloudss[51] et dans un des scénarios de validation de la thèse de Javier Espinosa [99]. Ils implantent des stratégies simples de rafraîchissement. Ils doivent évoluer pour prendre en considération des stratégies adaptatives, modifiables et tenant compte des besoins de l'utilisateur.

[49] http://red-shine.imag.fr/
[50] http://clever.imag.fr
[51] http://e-cloudss.imag.fr/

- Mettre en cache les résultats de certaines activités intermédiaires dans un arbre d'activités. Toutefois, pour cela, il faut prendre en considération plusieurs facteurs pour choisir l'activité ; comme le nombre d'exécutions (cette différence entre activités d'un arbre provient de la possibilité de qu'une activité peut faire partie de deux arbres ou plus), l'espace restant dans le $Store$ etc.
- Mettre des données en cache sur la machine du client. Le cache, en question, pourrait être utilisé pour stocker les données des mashlets d'un mashup. Ces données sont produites sur le serveur de MELQART et envoyées à l'utilisateur. Cacher ces données permet d'assurer leur disponibilité même en cas d'une déconnexion de la machine du client.
- Considérer le cas des données récupérées avec un mécanisme de pagination ou les données récupérées en continue sous la forme d'un flux. Dans ce cas Comment réorganiser le $Store$? Comment le gérer ? Et Comment réadapter les fonctionnalités de disponibilités de données ?
- Étudier et procéder à l'amélioration d'autres qualités de services QoS. Il existe déjà des travaux comme [100][101] qui ont traité le problème de la confidentialité des données (*data privacy*). D'autres propriétés QoS pour la gestion des mashups et des données ont été identifiées dans [67][102][103][33][104]. Parmi ces propriétés, nous retrouvons, la sécurité des données (*data security*), la précision des données (*data accuracy*) et l'utilisabilité de la présentation (*presentation usability*).
  - La sécurité réfère aux stratégies qui assurent la sureté d'un mashup et de ses données. Comment s'assurer de l'intégrité des données : le fait que les données ne subissent aucune altération ou déformation lors du transfert des données entre les fournisseurs et le mashup ? Comment gérer les données confidentielles dans une application où les données sont mélangées ?
  - La précisons des données réfère à l'exactitude et la cohérence des données du mashup par rapport au monde. La précision est mesurée comme la proximité des données récupérées aux données correctes (dans le monde réelle). La précision des données d'un mashup dépend de la précision des fournisseurs de données. Dans [67], la précision des données est exprimée comme étant la probabilité que les donnés soient correctes :
    $$p(acc) = 1 - p(err)$$
    Où $p(err)$ est la probabilité qu'une erreur se produise. L'erreur de précision peut se produire pour différentes raisons, telles que les fautes de frappe, la représentation erronée ou mises à jour manquantes. Comment un système d'exécution de mashups peut vérifier l'exactitude des données récupérées ? Comment détecter si un service de données n'est plus maintenu à jour par ses propriétaires ? Et comment réagir dans ce cas ?
  - L'utilisabilité de la présentation est une propriété qui caractérise l'expérience de l'utilisateur. Les mêmes attributs de l'utilisabilité de présentation, définis pour les pages web [105][106], peuvent être pris en compte pour l'utilisabilité de la présentation des mashups. En particulier la compréhension du fonctionnement (*learnability*) et la compréhension du contenu (*understandability*) et l'attractivité de la présentation (*attractiveness*). Dans le cas des mashups, est ce qu'il faut revoir ces attributs ? Quelles stratégies faut il mettre en place pour aider le constructeur de mashups à construire des mashups respectant ces attributs ?
- Etudier la prise en compte des préférences de l'utilisateur. Dans [107], les auteurs proposent un système de mashups avec la prise en compte des préférences de l'utilisateur liées aux informations qu'il recherche (par exemple : préférer le moins cher etc.). Nous pensons qu'il faut également développer une étude sur les préférences de l'utilisateur en matière de qualités de service du mashup comme la sécurité, la fraicheur, la précision des données.
- Étudier la répartition de la charge de l'exécution des mashups entre le serveur de MELQART et sur la machine de l'utilisateur. Comment repartir les tâches ? Et dans ce cas, la machine du client peut être faible en ressources (capacité de calcul mémoire, énergie), ainsi il faut mettre une place une stratégie d'optimisation d'exécution des mashups. Cette stratégie doit prendre en

considération l'ordre des exécutions des activités et le choix des algorithmes pour l'exécution des opérations de manipulation de données.

BIBLIOGRAPHIE

[1] D. Merrill, "Mashups: The new breed of Web app," IBM Corporation, CT316, Aug. 2006.

[2] F. Belleau, M.-A. Nolin, N. Tourigny, P. Rigault, and J. Morissette, "Bio2RDF: Towards a mashup to build bioinformatics knowledge systems," *Journal of Biomedical Informatics*, vol. 41, no. 5, pp. 706–716, Oct. 2008.

[3] S. Abiteboul, O. Greenshpan, and T. Milo, "Modeling the mashup space," in *Proceeding of the 10th ACM workshop on Web information and data management*, Napa Valley, California, USA, 2008, pp. 87–94.

[4] Ke Xu, Meina Song, and Xiaoqi Zhang, "Home Appliance Mashup System Based on Web Service," in *Service Sciences (ICSS), 2010 International Conference on*, 2010, pp. 94–98.

[5] C. Safran, D. Helic, and C. Gütl, "E-Learning practices and Web 2.0," presented at the Proc. of the 10th International Conference of Interactive computer aided learning (ICL 2007), Villach Austria, 2007.

[6] M. Eisenstadt, "Does Elearning Have To Be So Awful? (Time to Mashup or Shutup)," in *Advanced Learning Technologies, 2007. ICALT 2007. Seventh IEEE International Conference on*, 2007, pp. 6 –10.

[7] N. Schuster, C. Zirpins, M. Schwuchow, S. Battle, and S. Tai, "The MoSaiC model and architecture for service-oriented enterprise document mashups," in *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, New York, NY, USA, 2010, pp. 5:1–5:8.

[8] M. Essid, Y. Lassoued, and O. Boucelma, "Processing Mediated Geographic Queries: a Space Partitioning Approach," in *The European Information Society*, S. I. Fabrikant and M. Wachowicz, Eds. Springer Berlin Heidelberg, 2007, pp. 303–315.

[9] O. Boucelma, "Spatial Data Integration on the Web: Issues and Solutions," presented at the MoMM, 2006, pp. 5–6.

[10] "Pipes: Rewire the web." [Online]. Available: http://pipes.yahoo.com/pipes/. [Accessed: 12-Jan-2011].

[11] Jin Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding Mashup Development," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 44–52, 2008.

[12] H. Chen, B. Lu, Y. Ni, G. Xie, C. Zhou, J. Mi, and Z. Wu, "Mashup by surfing a web of data APIs," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1602–1605, 2009.

[13] E. M. Maximilien, A. Ranabahu, and K. Gomadam, "An Online Platform for Web APIs and

Service Mashups," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 32–43, 2008.

[14] E. M. Maximilien, H. Wilkinson, N. Desai, and S. Tai, "A Domain-Specific Language for Web APIs and Services Mashups," in *Proceedings of the 5th international conference on Service-Oriented Computing*, Vienna, Austria, 2007, pp. 13–26.

[15] Nan Zang and M. B. Rosson, "Playing with information: How end users think about and integrate dynamic data," in *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, 2009, pp. 85–92.

[16] R. Ennals and D. Gay, "User-friendly functional programming for web mashups," in *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, Freiburg, Germany, 2007, pp. 223–234.

[17] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau, "End-user programming of mashups with vegemite," in *Proceedings of the 13th international conference on Intelligent user interfaces*, Sanibel Island, Florida, USA, 2009, pp. 97–106.

[18] E. Acquaro, V. Manferto de Fabianis, and L. Cohen, *Les Phéniciens trésors d'une civilisation ancienne*. Vercelli (Italie); Paris: White Star, 2009.

[19] S. Abiteboul and C. Beeri, "The power of languages for the manipulation of complex values," *The VLDB Journal*, vol. 4, no. 4, pp. 727–794, Oct. 1995.

[20] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases: The Logical Level*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, juin 1970.

[22] F. Bancilhon, "The O2 object-oriented database system," in *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1992, pp. 7–7.

[23] K. R. Dittrich, H. Fritschi, S. Gatziu, A. Geppert, and A. Vaduva, "SAMOS in hindsight: experiences in building an active object-oriented DBMS," *Inf. Syst.*, vol. 28, no. 5, pp. 369–392, juillet 2003.

[24] R. G. G. Cattell, D. K. Barry, M. D. Berler, J. Eastman, D. Jordan, C. Russel, O. Schadow, T. Stanienda, and F. Velez, *The Object Data Satandard: ODMG 3.0*. Morgan Kaufmann, 2000.

[25] E. Griffin, *Foundations of Popfly: Rapid Mashup Development*. Apress, 2008.

[26] S. Abiteboul, P. Buneman, and D. Suciu, *Data on the Web: From Relations to Semistructured Data and Xml*. Morgan Kaufmann, 2000.

[27] S. Abiteboul, I. Manolescu, P. Rigaux, M.-C. Rousset, and P. Senellart, *Web Data Management*. Cambridge University Press, 2011.

[28] C. R. Anderson and E. Horvitz, "Web montage: a dynamic personalized start page," in *Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, 2002, pp. 704–712.

[29] R. J. Ennals and M. N. Garofalakis, "MashMaker: mashups for the masses," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, Beijing, China,

2007, pp. 1116–1118.

[30] "JackBe.com - Real-Time Intelligence Solutions | Presto." [Online]. Available: http://www.jackbe.com/products/. [Accessed: 12-Jan-2011].

[31] A. Bouguettaya, S. Nepal, W. Sherchan, Xuan Zhou, J. Wu, Shiping Chen, Dongxi Liu, L. Li, Hongbing Wang, and Xumin Liu, "End-to-End Service Support for Mashups," *Services Computing, IEEE Transactions on*, vol. 3, no. 3, pp. 250–263, 2010.

[32] "Mashup Server by WSO2 - Open Source Mashup Server for easy Web service composition and aggregation using JavaScript | WSO2." [Online]. Available: http://wso2.com/products/mashup-server/. [Accessed: 12-Jan-2012].

[33] J. Palfrey and U. Gasser, "Case Study: Mashups Interoperability and eInnovation," University of St. Gallen, 2007.

[34] D. Deutch, O. Greenshpan, and T. Milo, "Navigating through Mashed-up Applications with COMPASS," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, 2010, pp. 1117–1120.

[35] V. Hoyer and M. Fischer, "Market Overview of Enterprise Mashup Tools," in *Service-Oriented Computing – ICSOC 2008*, 2008, pp. 708–721.

[36] Amin Andjomshoaa, G. Bader, and A. M. Tjoa, "Exploiting Mashup Architecture in Business Use Cases," presented at the NBIS 2009, Indianapolis, USA, 2009.

[37] V. Hoyer and K. Stanoevska-Slabeva, "Towards a Reference Model for Grassroots Enterprise Mashup Environmentd," in *Proceedings of the 17th European Conference on Information Systems (ECIS 2009)*, Verona, Italy, 2009, p. 10.

[38] M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh, "Damia: a data mashup fabric for intranet applications," in *Proceedings of the 33rd international conference on Very large data bases*, Vienna, Austria, 2007, pp. 1370–1373.

[39] L. Ramaswamy, J. A Miller, and O. Al-Haj Hassan, "The MACE Approach for Caching Mashups," *International Journal of Web Services Research*, vol. 7, no. 4, pp. 64–88, 2010.

[40] J. Crupi, "A Business Guide to Enterprise Mashups," JackeBe Corporation, 2008.

[41] V. Hoyer and K. Stanoevska-Slabeva, "Generic Business Model Types for Enterprise Mashup Intermediaries," *Value Creation in E-Business Management*, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03132-8_1. [Accessed: 11-Dec-2009].

[42] D. Deutch, O. Greenshpan, and T. Milo, "Navigating in complex mashed-up applications," *Proc. VLDB Endow.*, vol. 3, pp. 320–329, Sep. 2010.

[43] A. Thor, D. Aumueller, and E. Rahm, "Data Integration Support for Mashups," 2007.

[44] G. Di Lorenzo, H. Hacid, H. Paik, and B. Benatallah, "Data integration in mashups," *SIGMOD Rec.*, vol. 38, no. 1, pp. 59–66, Jun. 2009.

[45] N. Bidoit, *Bases de données déductives: présentation de Datalog*. Armand Colin, 1991.

[46] P. G. Kolaitis and M. Y. Vardi, *On the Expressive Power of Datalog: Tools and a Case Study*.

University of California, Santa Cruz, Computer Research Laboratory, 1990.

[47] E. Ort, S. Brydon, and M. Balser, "Mashup Styles, Part 1: Server-Side Mashups," Oracle, 2007.

[48] E. Ort, S. Brydon, and M. Balser, "Mashup Styles, Part 2: Client-Side Mashups in the Java EE Platform," Oracle, 2007.

[49] A. Koschmider, V. Torres, and V. Pelechano, "Elucidating the Mashup Hype: Definition, Challenges, Methodical Guide and Tools for Mashups," in *2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web in conjunction with the 18th International World Wide Web Conference*, Madrid, 2009.

[50] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi, "Intel Mash Maker: join the web," *SIGMOD Rec.*, vol. 36, no. 4, pp. 27–33, 2007.

[51] S. Aghaee and C. Pautasso, "Mashup development with HTML5," in *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, New York, NY, USA, 2010, pp. 10:1–10:8.

[52] K. Pfeil, "Data Security and Data Availability in the Administrative Authority," Microsoft TechNet.

[53] F. Piedad and M. W. Hawkins, *High Availability: Design, Techniques, and Processes*, 1st ed. Prentice Hall, 2000.

[54] E. Marcus and H. Stern, *Blueprints for High Availability*. Wiley, 2003.

[55] V. Peralta, "Data freshness and data accuracy: A state of the art," Instituto de Computación, Facultad de Ingeniería, Universidad de la República, URUGUAY, 2006.

[56] M. Bouzeghoub, "A framework for analysis of data freshness," in *Proceedings of the 2004 international workshop on Information quality in information systems*, New York, NY, USA, 2004, pp. 59–67.

[57] B. Shin, "An Exploratory Investigation of System Success Factors in Data Warehousing," *Journal of the Association for Information Systems*, vol. 4, no. 1, Aug. 2003.

[58] X. Ma, S. S. Vazhkudai, and Z. Zhang, "Improving Data Availability for Better Access Performance: A Study on Caching Scientific Data on Distributed Desktop Workstations," *J Grid Computing*, vol. 7, no. 4, pp. 419–438, Dec. 2009.

[59] S. Drapeau, "RS2.7 : un Canvas Adaptable de Services de Duplication," Institut National Polytechnique De Grenoble, Grenoble, 2003.

[60] R. van Renesse and R. Guerraoui, "Replication," B. Charron-Bost, F. Pedone, and A. Schiper, Eds. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 19–40.

[61] R. van Renesse and R. Guerraoui, "Replication Techniques for Availability," in *Replication*, B. Charron-Bost, F. Pedone, and A. Schiper, Eds. Springer Berlin Heidelberg, 2010, pp. 19–40.

[62] L. D'Orazio, "Caches adaptables et applications aux systèmes de gestion de données répartis à grande échelle," Grenoble INP, 2007.

[63] J. Wang, "A survey of web caching schemes for the Internet," *SIGCOMM Comput.*

Commun. Rev., vol. 29, no. 5, pp. 36–46, Oct. 1999.

[64] K. S. Ahluwalia and A. Jain, "High availability design patterns," in *Proceedings of the 2006 conference on Pattern languages of programs*, New York, NY, USA, 2006, pp. 19:1–19:9.

[65] G. Attiya and Y. Hamam, "Reliability oriented task allocation in heterogeneous distributed computing systems," in *Ninth International Symposium on Computers and Communications, 2004. Proceedings. ISCC 2004*, 2004, vol. 1, pp. 68 – 73 Vol.1.

[66] N. R. May, H. W. Schmidt, and I. E. Thomas, "Service Redundancy Strategies in Service-Oriented Architectures," in *Proceedings of the 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, Washington, DC, USA, 2009, pp. 383–387.

[67] C. Cappiello, F. Daniel, M. Matera, and C. Pautasso, "Information Quality in Mashups," *Internet Computing, IEEE*, vol. 14, no. 4, pp. 14–22, 2010.

[68] D. Ballou, R. Wang, H. Pazer, and G. K. Tayi, "Modeling Information Manufacturing Systems to Determine Information Product Quality," *Manage. Sci.*, vol. 44, no. 4, pp. 462–484, avril 1998.

[69] G. Di Lorenzo, H. Hacid, H. Paik, and B. Benatallah, "Mashups for Data Integration: An Analysis," UNSW-CSE, 0810, 2008.

[70] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive Push-Pull: Disseminating Dynamic Web Data," *IEEE Trans. Comput.*, vol. 51, no. 6, pp. 652–668, juin 2002.

[71] G. Soundararajan, C. Amza, and A. Goel, "Database replication policies for dynamic content applications," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 89–102, avril 2006.

[72] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[73] C. Pu and A. Leff, "Replica control in distributed systems: as asynchronous approach," in *Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 1991, pp. 377–386.

[74] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," *ACM Trans. Database Syst.*, vol. 25, no. 3, pp. 333–379, Sep. 2000.

[75] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[76] M.-K. Liu, F.-Y. Wang, and D. Zeng, "Web caching: A way to improve web QoS," *Journal of Computer Science and Technology*, vol. 19, no. 2, pp. 113–127, 2004.

[77] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar, "Engineering web cache consistency," *ACM Trans. Internet Technol.*, vol. 2, no. 3, pp. 224–259, août 2002.

[78] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic, "DBToaster: higher-order delta processing for dynamic, frequently fresh views," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 968–979, juin 2012.

[79] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald, "Cache tables: paving the way for an adaptive database cache," in *Proceedings of the 29th international conference on Very large data bases - Volume 29*, 2003, pp. 718–729.

[80] A. Labrinidis and N. Roussopoulos, "Exploring the tradeoff between performance and data freshness in database-driven Web servers," *The VLDB Journal*, vol. 13, no. 3, pp. 240–255, Sep. 2004.

[81] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," in *Proceedings of the 22th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1996, pp. 51–62.

[82] Y. Kotidis and N. Roussopoulos, "DynaMat: a dynamic view management system for data warehouses," *SIGMOD Rec.*, vol. 28, no. 2, pp. 371–382, juin 1999.

[83] W. Ye, N. Gu, G. Yang, and Z. Liu, "Extended derivation cube based view materialization selection in distributed data warehouse," in *Proceedings of the 6th international conference on Advances in Web-Age Information Management*, Berlin, Heidelberg, 2005, pp. 245–256.

[84] I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy, "Second chance: a hybrid approach for dynamic result caching in search engines," in *Proceedings of the 33rd European conference on Advances in information retrieval*, Berlin, Heidelberg, 2011, pp. 510–516.

[85] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge, "A refreshing perspective of search engine caching," in *Proceedings of the 19th international conference on World wide web*, New York, NY, USA, 2010, pp. 181–190.

[86] Sadiye Alici, I. S. Altingovde, R. Ozcan, B. B. Cambazoglu, and Ö. Ulusoy, "Adaptive time-to-live strategies for query result caching in web search engines," in *Proceedings of the 34th European conference on Advances in Information Retrieval*, Berlin, Heidelberg, 2012, pp. 401–412.

[87] J. Tatemura, O. Po, A. Sawires, D. Agrawal, and K. S. Candan, "WReX: a scalable middleware architecture to enable XML caching for web-services," in *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, New York, NY, USA, 2005, pp. 124–143.

[88] V. Ramasubramanian and D. B. Terry, "Caching of XML Web Services for Disconnected Operation," Microsoft Research, 2002.

[89] T. Takase and M. Tatsubori, "Efficient Web services response caching by selecting optimal data representation," in *24th International Conference on Distributed Computing Systems, 2004. Proceedings*, 2004, pp. 188 – 197.

[90] D. D. Terry and V. Ramasubramanian, "Caching XML Web Services for Mobility," *Queue*, vol. 1, no. 3, pp. 70–78, mai 2003.

[91] H. Artail and H. Al-Asadi, "A Cooperative and Adaptive System for Caching Web Service Responses in MANETs," in *International Conference on Web Services, 2006. ICWS '06*, 2006, pp. 339 –346.

[92] J. Huang, X. Liu, Q. Zhao, J. Ma, and G. Huang, "A browser-based framework for data cache in Web-delivered service composition," in *2010 IEEE International Conference on Service-*

Oriented Computing and Applications (SOCA), 2010, pp. 1–8.

[93] Qi Zhao, Gang Huang, Jiyu Huang, Xuanzhe Liu, and Hong Mei, "A Web-Based Mashup Environment for On-the-Fly Service Composition," in *Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on*, 2008, pp. 32–37.

[94] P. Grogono and P. Santas, "Equality in Object Oriented Languages," presented at the EastEurOOPe'93, 1993.

[95] P. Grogono and M. Sakkinen, "Copying and Comparing: Problems and Solutions," in *ECOOP 2000 – Object-Oriented Programming*, E. Bertino, Ed. Springer Berlin Heidelberg, 2000, pp. 226–250.

[96] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction To Algorithms, 3rd Edition*, 3rd Revised ed. Cambridge, Massachusetts - London, England: The MIT Press, 2009.

[97] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," *ACM Comput. Surv.*, vol. 24, no. 4, pp. 441–476, décembre 1992.

[98] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley, 1994.

[99] J. Espinosa-Oviedo, "Coordination fiable de services de données à base de politiques actives," de Grenoble, 2013.

[100] M. Barhamgi, D. Benslimane, C. Ghedira, A. N. Benharkat, and A. L. Gancarski, "PPPDM - a privacy-preserving platform for data mashup," *International Journal of Grid and Utility Computing*, vol. 3, no. 2/3, p. 175, 2012.

[101] R. Hasan, M. Winslett, R. Conlan, B. Slesinsky, and N. Ramani, "Please Permit Me: Stateless Delegated Authorization in Mashups," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, 2008, pp. 182, 173.

[102] C. Cappiello, F. Daniel, and M. Matera, "A Quality Model for Mashup Components," in *Web Engineering*, 2009, pp. 236–250.

[103] A. Koschmider, V. Hoyer, and A. Giessmann, "Quality metrics for mashups," in *Proceedings of the 2010 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, New York, NY, USA, 2010, pp. 376–380.

[104] A. Portilla, V. Hernandez-Baruch, G. Vargas-Solar, J.-L. Zechinelli-Martini, and C. Collet, "Building reliable services based mashups," presented at the IV Jornadas Cienfifico-Técnicas en Servicios WEB y SOA (JSWEB 2008), Sevilla, Spain, 2008.

[105] C. Calero, J. Ruiz, and M. Piattini, "A Web Metrics Survey Using WQM," in *Web Engineering*, vol. 3140, N. Koch, P. Fraternali, and M. Wirsing, Eds. Springer Berlin / Heidelberg, 2004, pp. 766–766.

[106] M. Matera, F. Rizzo, and G. Carughi, "Web Usability: Principles and Evaluation Methods," in *Web Engineering*, E. Mendes and N. Mosley, Eds. Springer Berlin Heidelberg, 2006, pp. 143–180.

[107] S. Amdouni, D. Benslimane, M. Barhamgi, A. Hadjali, R. Faiz, and P. Ghodous, "A Preference-aware Query Model for Data Web Services," in *Proceedings of the 31st International Conference on Conceptual Modeling*, Berlin, Heidelberg, 2012, pp. 409–422.

[108] O. Beletski, "End User Mashup Programming Environments," in *T-111.5550 Seminar on Multimedia*, 2008.

[109] "Microsoft Research FUSE Labs - Project Montage." [Online]. Available: http://fuse.microsoft.com/page/montage. [Accessed: 27-Jun-2012].

[110] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi, "Intel Mash Maker: join the web," *SIGMOD Rec.*, vol. 36, no. 4, pp. 27–33, 2007.

[111] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: data mashups for intranet applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, Vancouver, Canada, 2008, p. 1171.

ANNEXE A :

SYSTEMES D'EXECUTION DE MASHUPS ETUDIES

Pendant ces dernières années, plusieurs environnements de création de mashups ont vu le jour. Chacun de ces environnements offre un ensemble de fonctionnalités et caractéristiques qui répondent à des besoins spécifiques d'un certain public. Ainsi, certains environnements sont dédiés aux constructeurs novices, alors que d'autres s'adressent aux développeurs. Dans cette section, nous présentons une sélection d'environnements de création et d'exécution de mashups. Pour des études poussées sur ces outils, le lecteur peut se référer à [35][11][108].

Montage

Montage [109] est un outil de création de mashups développé par Microsoft. Il permet aux utilisateurs novices de créer des pages web personnalisées, qui peuvent afficher des données provenant de différentes sources, comme le montre la Figure 67[52]: vidéos, flux RSS, images, tweets …

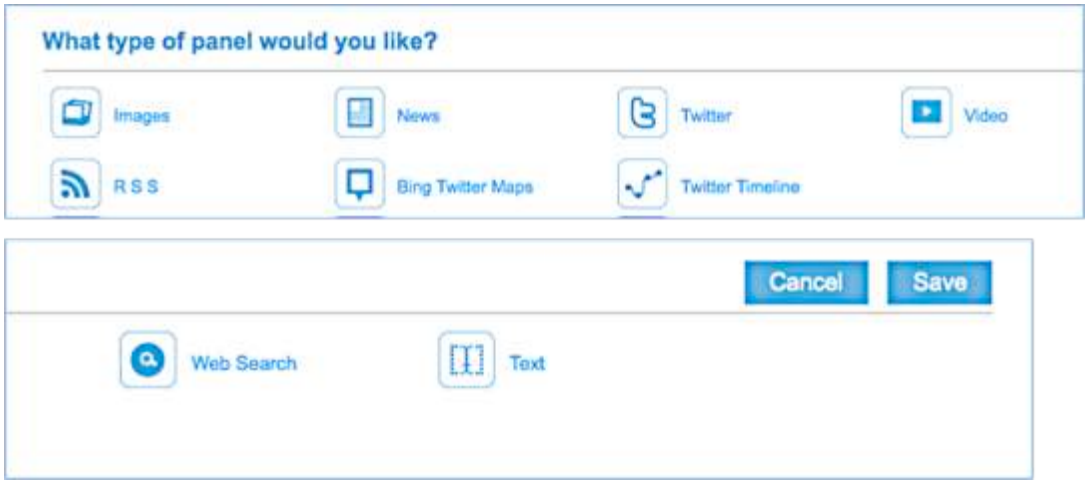

Figure 67 : Sources de données dans Montage

Le contenu est disposé sur la page sous la forme d'une grille. Celle-ci peut être décomposée horizontalement et verticalement. Chaque rectangle de la grille représente un mashlet et affiche les données choisies par l'utilisateur. Il n'y a aucune communication entre les mashlets.

[52] La figure est scindée en deux, pour une raison d'espace.

Yahoo! Pipes

Yahoo! Pipes [10] permet aux utilisateurs d'agréger des données provenant de différents sources comme des flux ou des pages web via un éditeur graphique, comme le montre la Figure 68.

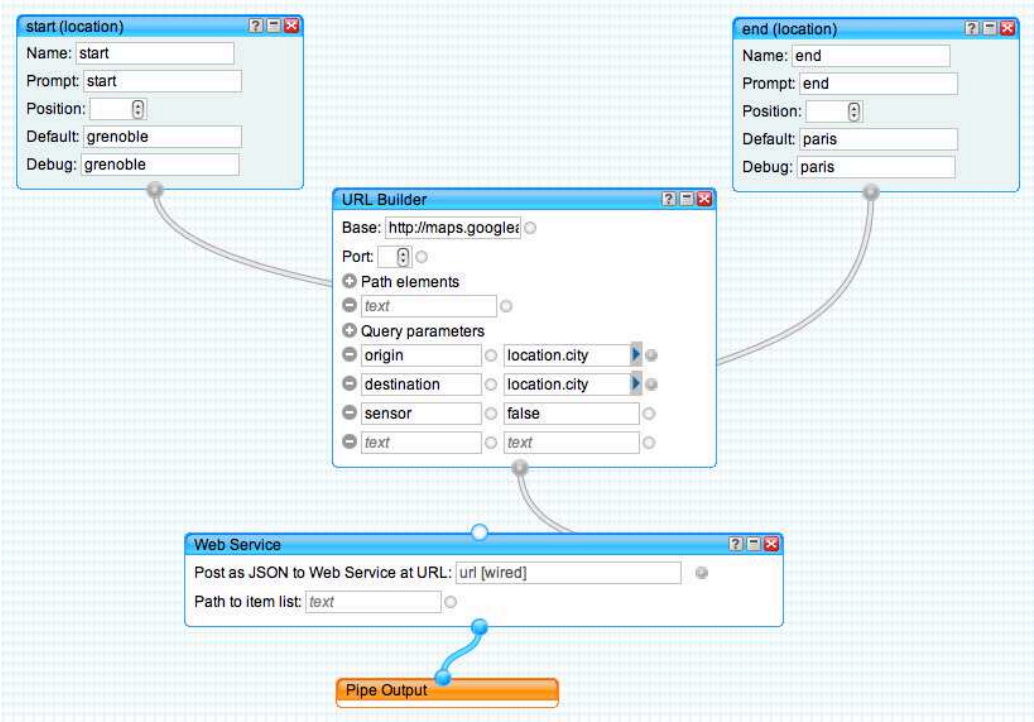

Figure 68 : Construction de mashlets dans Yahoo! Pipes

Il est principalement destiné aux utilisateurs ayant quelques notions de programmation. En effet, les mashups sont créés via un mécanisme de glisser-déposer. Un « Pipe » est un ensemble de modules connectés. Un module correspond à un opérateur qui exécute une tâche spécifique. L'ensemble des opérateurs disponibles comprend les opérateurs de transformation de données (tri, filtrage ...), les opérateurs de contrôle (boucle), les opérateurs arithmétiques (comptage…) et les opérateurs sur les expressions régulières.

Intel MashMaker

MashMaker [110][16][29] est un outil de construction de mashups proposé par Intel en version bêta, mis hors service en 2011. Il est fourni sous la forme d'une extension pour les navigateurs Firefox et Internet Explorer. L'outil offre des fonctionnalités qui permettent de personnaliser des pages web en ajoutant des informations supplémentaires et des widgets. L'utilisateur peut définir des extracteurs (*scrapers*) pour extraire des données des pages web. Les données extraites sont organisées sous la forme d'une structure arborescente qui peut être lue et enrichie par d'autres widgets. Les mashups sont construits par un mécanisme de copier-coller de widgets (qui affichent des données extraites d'autres sites) et par l'ajout de widgets qui exécutent des requêtes de transformations de données.

WSO2 Mashup Server

WSO2 Mashup Server [32] est une plateforme open source qui est dédiée aux développeurs. Elle offre un environnement pour créer, déployer et exécuter les mashups. La plateforme est basée sur les technologies XML et JavaScript. Elle permet d'agréger des données issues de différentes sources comme les pages web, les services web, les flux ou les bases de données. Le résultat peut être exposé sous la forme d'un flux, d'un service web, d'une page web, ou d'un widget à ajouter dans iGoogle.

Damia

Damia [38][111] est un outil de construction de mashups proposé par IBM. Il permet d'agréger des données issues de différentes sources comme Notes, Excel, des pages web, … Les données sont agrégées par le moyen d'opérateurs comme l'union, le tri, le filtrage. La création du mashup se fait au sein d'un navigateur. Comme dans Yahoo! Pipes, le constructeur glisse les sources et les opérateurs et les branches via des connecteurs.

Presto Enterprise Mashups

Presto Enterprise Mashups [40] est une plateforme de création de mashups proposée par JackBe Corporation. La plateforme est dédiée à la création de mashups d'entreprises. Elle permet d'agréger des données issues de sources internes et d'autres issues de sources externes. La plateforme fournit également des outils pour définir les échanges entre mashlets ainsi que l'interface graphique du mashup et des mashlets.

L'agrégation de données peut se faire soit d'une manière graphique, soit par programmation. Dans le premier cas, la construction est similaire à celle de Yahoo! Pipes, et se fait dans un navigateur web. Dans le second cas, le développeur est amené à écrire son code en langage EMML, via une extension Eclipse : Presto Mashup Studio.

La plateforme fournit également des extensions qui permettent d'importer des données et/ou d'exporter les mashlets vers d'autres plateformes comme :

- Microsoft SharePoint
- Des produits Oracles.
- Des terminaux mobiles
- Des portails
- Des fichiers Excel

Mashup Service System

Mashup Service System MSS [31] est une plateforme conçue pour les utilisateurs novices, qui leur permet de construire des mashups personnalisés qui répondent à des besoins spécifiques. Comme illustré dans la Figure 69, les données sont accessibles via des services. Ces services sont décrits selon un certain modèle sémantique de services (basé sur une certaine ontologie définie par « une communauté »), qui permet d'avoir un raisonnement automatique sur les services. L'utilisateur est invité à exprimer ses besoins, qui sont traduits en requêtes MSQL (Mashup Service Query Language, langage de type SQL). MSS recherche alors, automatiquement, les services

appropriés et crée le mashup en composant automatiquement les services. Le mashup est exécuté et le résultat est retourné à l'utilisateur comme réponse à sa requête.

Dans [107], les auteurs proposent une approche similaire. Elle est basée également sur une description sémantique des services web et de l'expression des besoins de l'utilisateur. Ils permettent de surcroit, à l'utilisateur d'exprimer des préférences dans sa requête. Celles ci sont prises en considération lors du calcul et du tri du résultat.

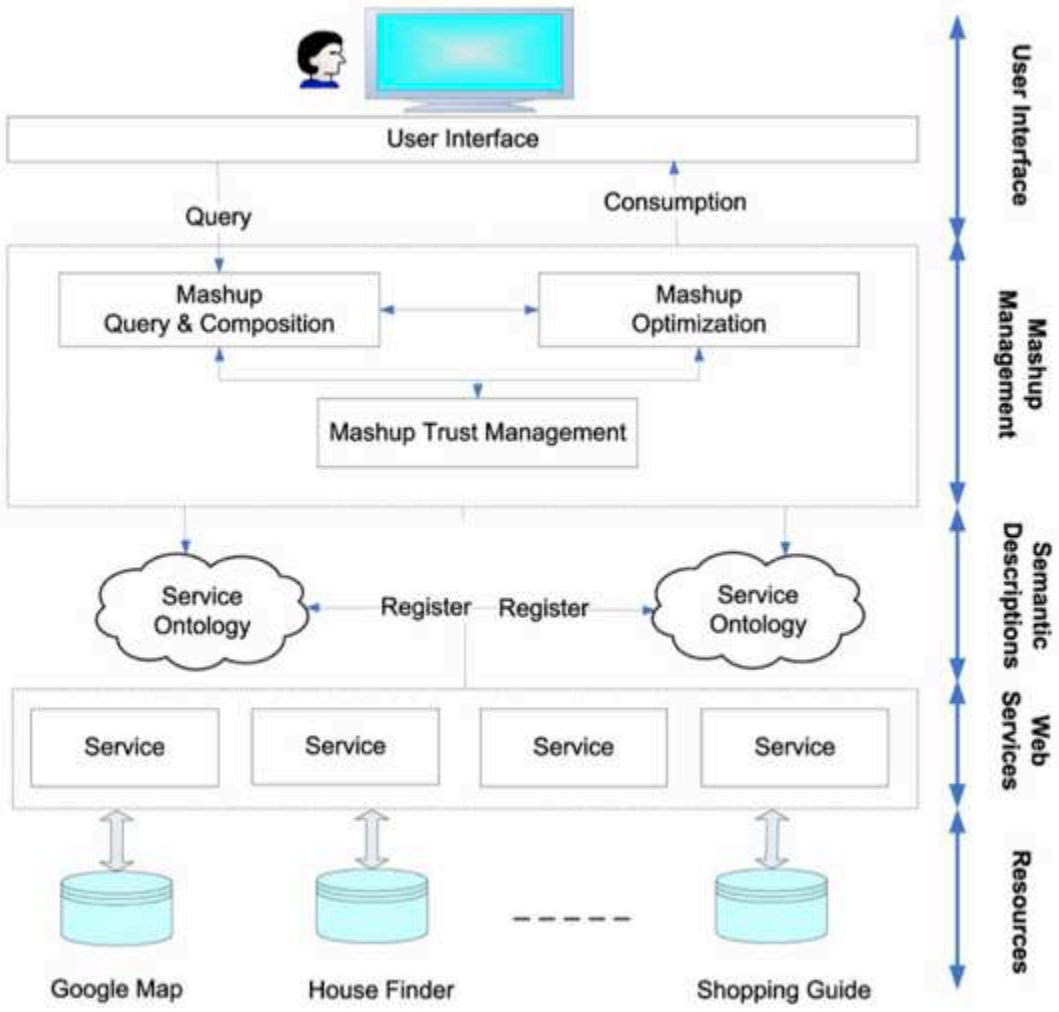

Figure 69 : Architecture de MSS

ANNEXE B :

SYNTAXE DE JSON

La syntaxe de JSON est présentée dans la Figure 70[53]. JSON se base sur deux structures : les **objets** (une collection de paires nom : valeur) et les **tableaux** (une liste de valeurs ordonnées). Pratiquement tous les langages de programmation proposent ces structures de données sous une forme ou une autre. Les **valeurs** peuvent être atomiques de type chaine de caractère, nombre, booléen ou des objets ou des tableaux.

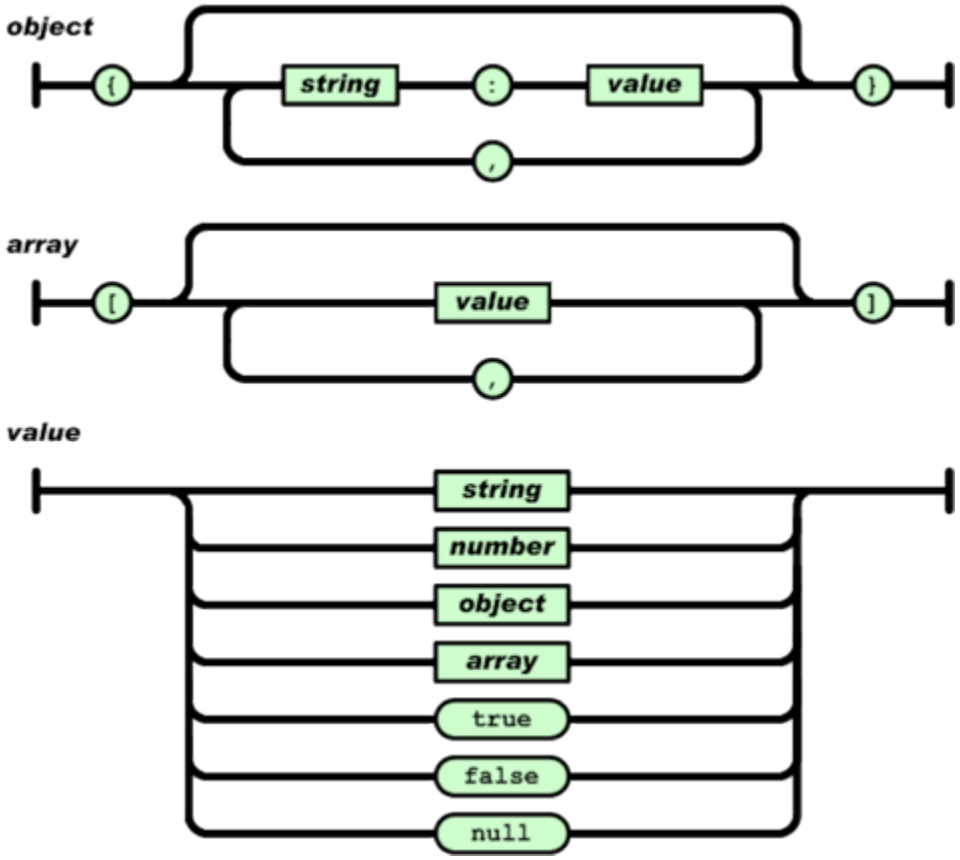

Figure 70 : Syntaxe de JSON

[53] Images obtenues de http://www.json.org/json-fr.html

Un **objet JSON** est représenté par un ensemble de paires **nom : valeurs** séparées par des virgules entre deux accolades. Par exemple une valeur de l'entrée du mashlet Map du mashup ItineraryPlanner peut être représentée par un l'objet JSON `entreeMashup` suivant :

```
entreeMashup = {“depart” : “Grenoble”, “destination” : “Paris”, “note” : 4}
```

Un **tableau JSON** est une liste de valeurs ordonnées séparées par des virgules et délimitées par deux crochets. Par exemple, la liste de valeurs de villes intermédiaires entre Grenoble et Paris sous format WOEID[54] peut être représentée par le tableau JSON `listeVilles` suivant :

```
listeVilles = [{"woeid":12724717}, {"woeid":12724728}, {"woeid":12724752},
{"woeid":12726960}, {"woeid":582081}, {"woeid":12726917},
{"woeid":12726920},{"woeid":12726958}, {"woeid":12723642},
{"woeid":12728226}, {"woeid":12728378}, {"woeid":12728381},
{"woeid":20068149}, {"woeid":20068148}, {"woeid":20068141}]
```

Un **document JSON** est un objet JSON. Il peut être constitué d'imbrications d'un nombre non limité d'objets et tableaux [27]. L'itinéraire entre Grenoble et Paris est décrit par le document JSON `routeGP` dont une partie est donnée ci dessous.

```
{"routes" : [
      {
         "bounds" : {
            "northeast" : {
               "lat" : 48.857240,
               "lng" : 5.724690000000001
            },
            "southwest" : {
               "lat" : 45.18839000000001,
               "lng" : 2.306130
            }
         },
         "copyrights" : "Données cartographiques ©2012 GeoBasis-DE/BKG
(©2009), Google",
         "legs" : [
            {
               "distance" : {
                  "text" : "574 km",
                  "value" : 573623
               },
               "duration" : {
                  "text" : "5 heures 21 minutes",
                  "value" : 19284
               },
               "end_address" : "Paris, France",
               "end_location" : {
                  "lat" : 48.857240,
                  "lng" : 2.35260
               },
               "start_address" : "Grenoble, France",
               "start_location" : {
                  "lat" : 45.18839000000001,
                  "lng" : 5.724690000000001
               },
```

[54] Un identifiant WOEID (Where on Earth IDentifier) est un identifiant de référence assigné par Yahoo! Pour identifier des points géographiques sur la terre.

```
                "steps" : [
                   {
                      "distance" : {
                         "text" : "92 m",
                         "value" : 92
                      },
                      "duration" : {
                         "text" : "1 minute",
                         "value" : 7
                      },
                      "end_location" : {
                         "lat" : 45.188890,
                         "lng" : 5.725610000000001
                      },
                      "html_instructions" : "Prendre la direction
\u003cb\u003enord-est\u003c/b\u003e sur \u003cb\u003ePl. Victor
Hugo\u003c/b\u003e vers \u003cb\u003eRue Paul Bert\u003c/b\u003e",
                      "polyline" : {
                         "points" : "mzxrGib}a@g@gA{@oB"
                      },
                      "start_location" : {
                         "lat" : 45.18839000000001,
                         "lng" : 5.724690000000001
                      },
                      "travel_mode" : "DRIVING"

……
      }
    ],
 "status" : "OK"
 }
```

L'accès à une valeur dans un tableau se fait classiquement par une notation indexée. Par exemple `listeVilles[1] ={"woeid":12724717}`. Tandis que l'accès à une valeur d'une paire d'un objet JSON se fait en indiquant le nom de la paire. Par exemple `entreeMashup.depart = "Grenoble"`. En combinant ces 2 notations, on peut spécifier des chemins d'accès à des valeurs à l'intérieur d'un document JSON. Par exemple, le chemin d'accès suivant :

```
routeGP.routes[0].legs[0].steps[0].start_location
```

permet d'accéder à la donnée qui représente les coordonnées géographiques du point de départ de la première étape de l'itinéraire.

```
 {"lat" : 45.18839000000001,"lng" : 5.724690000000001}.
```

Cette thèse présente MELQART, un système d'exécution de mashups avec disponibilité de données. Un mashup est une application web qui combine des données provenant de fournisseurs hétérogènes (web services). Ces données sont agrégées pour former un résultat homogène affiché dans des composants appelés mashlets. Les travaux dans le domaine des mashups, se sont principalement intéressés au fonctionnement des mashups, aux différents outils de construction et à leur utilisation et interaction avec les utilisateurs.

Dans cette thèse, nous nous intéressons à la gestion de données dans les mashups et plus particulièrement à la disponibilité et la fraîcheur de ces données. L'amélioration de la disponibilité tient compte du caractère dynamique des données des mashups. Elle garantit (1) l'accès aux données même si le fournisseur est indisponible, (2) la fraicheur de ces données et (3) un partage de données entre les mashups afin d'augmenter la disponibilité de données.

Pour cela nous avons défini un modèle de description de mashups permettant de spécifier les caractéristiques de la disponibilité des données. Le principe d'exécution de mashups est défini selon ce modèle en proposant d'améliorer la disponibilité et la fraicheur des données du mashup par des fonctionnalités orthogonales à son processus d'exécution. Le système MELQART implante ce principe et permet de valider notre approche à travers l'exécution de plusieurs instances de mashups dans des conditions aléatoires de rupture de communication avec les fournisseurs de données.

Mots clés : Mashups, Web 2.0, services web, disponibilité de données, fraicheur de données

---

This thesis presents MELQART: a mashup execution system that ensures data availability. A mashup is a Web application that application that combines data from heterogeneous provides (Web services). Data are aggregated for building a homogenous result visualized by components named mashlets. Previous works have mainly focused, on the definition of mashups and associated tools and on their use and interaction with users.

In this thesis, we focus on mashups data management, and more specifically on fresh mashups data availability. Improving the data availability take into account the dynamic aspect of mashups data. It ensures (1) the access to the required data even if the provider is unavailable, (2) the freshness of these data and (3) the data sharing between mashups in order to avoid the multiple retrieval of the same data.

For this purpose, we have defined a mashup description formal model, which allows the specification of data availability features. The mashups execution schema is defined according to this model with functionalities that improve availability and freshness of mashed-up data. These functionalities are orthogonal to the mashup execution process. The MELQART system implements our contribution and validates it by executing mashups instances with unpredictable situations of broken communications with data providers.

Keywords : mashups, Web 2.0, web services, data availability, data freshness