UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Rainer Villido

Semantic Integration Platform for Web Widgets' Communication

Master's thesis (30 ECTS)

Advisor: Peep Küngas

Author: .... "..." May 2010

Advisor: .... "..." May 2010

Approved for defence
Professor: .... "..." May 2010

Tartu 2010

Contents

Introduction

Semantic integration platform for web widgets communication is a framework for providing collaboration capabilities between loosely coupled Web components in a mashup-like Web application.

Mashups are Web applications that allow reuse of excising resources by combining different widgets that use data from various sources in the Web. Current mashup platforms do not support collaboration between widgets if widgets have been developed and maintained by different vendors and are not able to interpret messages sent by other widgets. This limits the creation of sophisticated mashups where Web widgets could interactively share and exchange data between each other and make it possible to have collaboration between independent Web components.

This thesis intends to solve the problem that widgets are not able to share information with each other and collaboration between widgets is limited. Making the data published by a widget on a Web application available to all the other widgets connected to the application is the main goal of this work, which would allow interactively combine data from various sources to enable collaboration between loosely coupled components on a Web application.

The thesis proposes a solution for aggregating data from messages sent by different widgets and reusing the data to generate new messages to other widgets which could use the combined data. The main problem is collecting useful data from the exchanged messages and transforming the collected data into new messages that would be interpretable by widgets that are using different data formats and structures. Integrating and sharing data from various sources is the main research problem in the field of semantic integration and this thesis proposes one solution for sharing data between independent

Web widgets in a mashup.

The solution proposed in the thesis is built on the OpenAjax Hub [2] framework that provides the means for Web widgets to exchange messages between each other. OpenAjax Hub provides a central hub that allows messaging between widgets that are connected to the hub.

The main problem is the use of different data structures in messages exchanged by widgets developed by different vendors. Even though the widgets can use the hub for exchanging messages, the content of the exchanged messages remains unknown for the widgets because they are not able to interpret messages sent by other widgets.

The solution proposed and implemented in this thesis is a JavaScript application that is connected to the OpenAjax messaging hub to transform the exchanged data to be interpretable to all the widgets. The proposed application is a widget called Transformer Widget that uses semantic integration to transform data.

The Transformer Widget listens to the messages exchanged by the widgets connected to the hub and uses preconfigured mappings to identify and combine data elements it receives from messages. Mappings that describe the structure and the semantics of the messages are being used to collect data elements from existing messages and to generate new messages from the collected data elements. Mappings contain descriptions of the atomic data elements in the messages where each atomic data element is matched with a term in an ontology[1] that describes the meaning of that particular data element. This allows automatic understanding of the content of the exchanged messages regardless of which data structures are used in the messages. With the help of mappings, it is possible to collect atomic data elements from the received messages to generate new messages that can be sent to the widgets that can interpret generated messages.

The Transformer Widget allows building of mashups with complex application logic where loosely coupled components (widgets) can collaborate and perform tasks that would otherwise be difficult to implement using widgets.

[1]Ontology is a specification of a representational vocabulary for a shared domain of discourse – definitions of classes, relations, functions, and other objects [17].

The rest of the thesis is organized as follows. The problem is introduced further in the next chapter 1 which discusses the paradigms of Web 2.0, mashups and widgets which are the fundamental concepts underlying the thesis. The chapter 2 gives an overview of the current standards, platforms and problems that relate to mashups. The chapter 2 also introduces the main conventional problems in the field of semantic integration. The proposed solution for semantic integration of data exchanged by widgets is discussed in the chapter 3 where the main ideas of how to transform messages are discussed. The chapter 4 describes the specification of mapping configurations that can be created to integrate data on a mashup. The chapter also gives installation instructions of how to implement and install mashups and widgets using OpenAjax Hub and Transformer Widget. The chapter 5 discusses the technology behind the Transformer Widget and gives a detailed overview of the implementation of the project. The usability of the Transformer Widget is discussed in the chapter 6 which gives an overview of the test application that was used to verify if the Transformer Widget can be used in integrating data to enable collaboration between widgets. The possible improvements of the Transformer Widget are introduced in the chapter 7 where possible future developments are discussed.

The author would like to thank his supervisor Peep Küngas for his insightful guidance and support.

Chapter 1

Mashups

Mashups have emerged from the idea of Web 2.0 [30] which is a concept (but not a new technical specification) of Web pages having more qualities of interactive information sharing, interoperability, user-centered design, and collaboration (like web communities, social networking sites, video sharing sites, wikis, blogs etc). A Web site would allow interaction between users or to change website content by users in contrast to non-interactive websites where users only passively view information that is provided to them by static Web.

Web 2.0 consists of principles like "The Web as Platform" [30] where applications have moved from desktop to Web, "Harnessing Collective Intelligence" [30] where end-users generate content for the applications, where it is hard to distinguish who owns the data in the Web.

The idea of Web 2.0 has resulted in widgets and mashups technologies where end-users mash reusable components (widgets) together to personal Web pages. That breaks the traditional software development model with a system analysis, programming, testing, and deployment of the application. With widgets and mashups the applications can be built more rapidly by combining reusable components.

Widgets (sometimes also called gadgets) are small reusable client-side Web applications meant to be distributed to multiple locations. They can be installed on a client machine or can be embedded into Web pages and run in a Web browser. According to The World Wide Web Consortium (W3C) [41], they range from simple widgets that simply display information (e.g. weather forecast) to complex applications that combine data from multiple sources and allow user to interact with it in a useful way.

Widgets are created using Web technologies which means that they are easier to create than traditional binary applications developed with lower-lever programming languages.

W3C categorizes [42] widgets into desktop widgets, Web widgets and mobile widgets. All of them are built on Web technologies and offer mostly the same functionality, but are packaged differently, have different security models and APIs. Web widgets have the same restrictions [42] as regular Web applications that are limited by the security models of Web browsers (e.g. applications cannot autonomously access resources on a device) but desktop and mobile widgets have less strict security models that can allow more access to devices. No matter in which environment widgets are used, they are still regular Web applications that are mainly composed of HTML, CSS, JavaScript and other Web technologies where HTML and CSS are used in user interface layout and JavaScript is used in application logic and communicating with external services and resources. Widgets are designed to provide a specific function that can be used together with rest of the application. They are meant to be used in different applications and by different users and therefore are usually designed to be customizable and flexible to be adoptable in wide range of applications.

These environments or platforms that allow combination of widgets into a single application are called Mashups [7]. Mashups come from a phrase “to mash up” [7] that literally means an activity when a user putting together existing widgets – the user is “mashing up” a new application. The idea of mashups is to allow end users to combine together existing Web components and resources from different providers to make up new applications that could provide desired functionality. This makes mashups personal and unique to every user.

The traditional software development model with requirements, design, implementation and testing is no longer valid when building mashups.

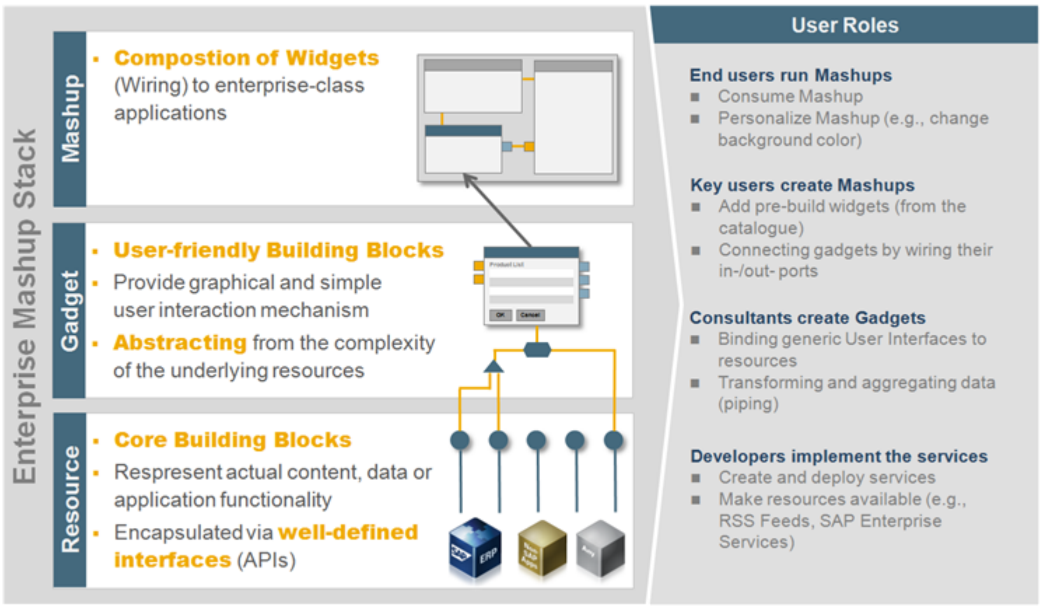

Enterprise Mashup Stack
Mashup
Compostion of Widgets (Wiring) to enterprise-class applications
Gadget
User-friendly Building Blocks
Provide graphical and simple user interaction mechanism
Abstracting from the complexity of the underlying resources
Resource
Core Building Blocks
Respresent actual content, data or application functionality
Encapsulated via well-defined interfaces (APIs)
User Roles
End users run Mashups
Consume Mashup
Personalize Mashup (e.g., change background color)
Key users create Mashups
Add pre-build widgets (from the catalogue)
Connecting gadgets by wiring their in-/out- ports
Consultants create Gadgets
Binding generic User Interfaces to resources
Transforming and aggregating data (piping)
Developers implement the services
Create and deploy services
Make resources available (e.g., RSS Feeds, SAP Enterprise Services)

Figure 1.0.1: Enterprise Mashup Stack [18]

Mashups create a new paradigm [7] for software development where developers create new Web services (the SOA approach) and build new widgets using existing services and resources and end users simply take existing widgets and combine them to perform desired tasks. This is much faster way of building new applications than going through the whole process in building traditional applications.

The Figure 1.0.1 [18] represents mashup architecture. Resources are at the bottom layer and are various Web services and resources that can provide source of information and data for widgets. The resources can be provided by different vendors and are not restricted by only one provider, making them core building blocks for creating functionality for the widgets. The Gadget layer represents widgets that combine various resources from the Resource layer into small reusable Web applications. On top in the Mashup layer which combine the collection of widgets into a complex application. It allows linking and exchanging of information between widgets.

Many software vendors have mashup platforms available for personal and enterprise use. The more popular mashup platforms are IBM Mashup Center [19], Kapow Mashup Server [37], JackBe Presto [6], iGoogle [11], Netvibes

[26], Yahoo! Pipes [46] etc. Mashup providers differentiate between consumer mashups and enterprise mashups. Consumer mashups (iGoogle, Netvibes, Yahoo! Pipes) allow individual users to create their own individual mashups that are hosted on the providers' infrastructure (i.e. on cloud). Enterprise mashup providers (IBM Mashup Center, Kapow Mashup Server, JackBe Presto, etc) provide mashup platforms for enterprises and allow building more sophisticated mashups than personal mashup providers. Most enterprise mashup platforms allow collaboration between widgets whereas only few consumer mashup platforms (Yahoo! Pipes) allow some kind of messaging between widgets which makes it difficult to create more complex applications based on widgets.

There are many shortcomings regarding current mashup platforms. The main problem is lack of collaboration between widgets on mashup platforms. Most widgets today are very simple and made only for displaying information. There is a lack of infrastructure that would support communication and exchange of data between widgets.

There are few mashup platforms available that support collaboration between widgets within their platform, but those solutions are too vendor specific which do not allow communication with external widgets. There are also few data-flow model [48] approaches available (e.g. Yahoo! Pipes) which allow combination of different data sources while a mashup page is being loaded, but they lack event-based [48] messaging which would allow more interactive mashups to be built where widgets could exchange messages and data in real-time without reloading the page.

To allow building of more complex mashups, an infrastructure and standards for collaboration between widgets are needed. If widgets could be combined to exchange messages with each other so that independent Web components could collaborate on a Mashup, then more sophisticated applications can be built using widgets.

Another problem is incompatibility between mashup platforms and widgets of different vendors. All mashup platforms have different packaging formats, API and standards for widgets which restricts the combination of widgets from different vendors. To build a mashup means locking in to the technologies and standards of a particular mashup provider which makes it impossible to use widgets developed for another platform.

A study about mashups and modularity [35] found that in order to accomplish mashups that uses components and other content from all over the Web, improvements are needed in security and modularity which is critical, because it is quite impossible to build maintainable systems unless all the reusable elements have well-defined interfaces. Additionally, according to the article [48], despite of recent advanced in mashup creation still too much manual effort is required to build mashups, and the article [49] adds that the problems developers encounter when creating mashups are the reliability of the API documentation and coding details, and the JavaScript skills needed to integrate the APIs.

Chapter 2

Related Work

The lack of common standards in widgets and mashups reduce interoperability between components made by different providers. This has lead to many initiatives to develop common standards for all widget providers.

The World Wide Web Consortium (W3C) [41] has worked on a standard for widgets' packaging format and the requirements of installable, desktop or mobile widgets. This standard is not meant for distributing widgets in Web applications, but rather in various user agents for devices like mobile phones and other environments like desktop sidebars. The idea is to have one common packaging standard so that no matter for which platform you develop the widget for, it can run in any other platform also (for example, you can run the same widget in Windows Sidebar, Apple Dashboard, and Google Desktop).

OpenAjax Alliance [1] is an organization of vendors, open source projects and companies includes members from IBM, Microsoft, Tibco, Google, JackBe, Adobe, Opera, Oracle and many others. The objective of this standardization effort is to define a metadata format and a lightweight runtime environment that allows developers and users of widgets to easily assemble loosely coupled components from a variety of widget providers [39]. The specification does not provide a new widget model but provides a metadata layer to the widget environment. This metadata layer provides enough information to a widget assembly tool so that it can understand the properties and messaging exchanging capabilities of a particular widget. Widgets retain their original component model and are capable to be easily portable to different platforms [39].

OpenAjax Hub [2] is another initiative from OpenAjax Alliance. The OpenAjax Hub defines a standard for using the JavaScript library of the OpenAjax Hub to isolate widgets into secure containers and allowing them to communicate with each other through a central messaging hub that exchanges messages sent by widgets. It also provides a security mechanism for controlling widgets communication and preventing malicious widgets from accessing and harming rest of the application.

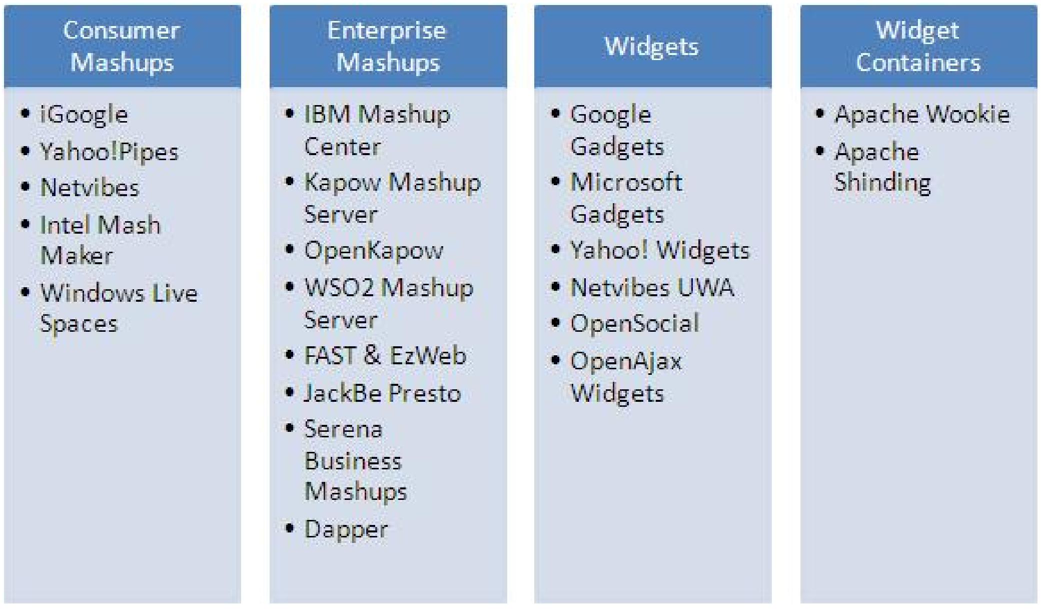
The OpenAjax Hub is currently the only framework for allowing third party widgets to exchange messages between each other. The hub provides necessary infrastructure to build complex mashups using widgets that are built by different vendors.

The Open Mashup Alliance has developed open Enterprise Mashup Markup Language (EMML) [4] that improves mashup portability of mashup designs, and increases the interoperability of mashup solutions. Mashups written in EMML can be deployed to any EMML-compliant application.

OpenSocial [8] is a standard established by Google to access the social data contained in the various social networking sites. It provides a set of common APIs for building social applications across the Web. An advantage of OpenSocial is that it is supported by at least 20 [8] social networking websites.

2.1 Mashup Providers

There is wide range of mashup providers available with each one with their own platforms, technologies and APIs that makes up very diverse landscape of mashup technologies. The most popular consumer mashup platforms are iGoogle [11], Netvibes [26] and Yahoo! Pipes [46] which allow users to create their own simple mashups. Consumer mashup platforms usually have communities that are developing new widgets for everyone to use. Popular enterprise mashup platforms are IBM Mashup Center [19], Kapow Mashup



Consumer Mashups	Enterprise Mashups	Widgets	Widget Containers
• iGoogle • Yahoo!Pipes • Netvibes • Intel Mash Maker • Windows Live Spaces	• IBM Mashup Center • Kapow Mashup Server • OpenKapow • WSO2 Mashup Server • FAST & EzWeb • JackBe Presto • Serena Business Mashups • Dapper	• Google Gadgets • Microsoft Gadgets • Yahoo! Widgets • Netvibes UWA • OpenSocial • OpenAjax Widgets	• Apache Wookie • Apache Shinding

Figure 2.1.1: Mashups and Widgets

Center [37], WSO2 Mashup Server [45], JackBe Presto [6] and many others that provide businesses with more sophisticated mashup platforms, but they usually lack large collection of widgets created by community that is more common for the consumer mashups.

Most mashup providers usually have their own widget packaging formats and APIs used to add widgets to mashups. Google [11] has its own Google Gadgets and Google Wave Gadgets standards and APIs for iGoogle and Google Wave, Microsoft [23] has Microsoft Gadgets standards used in Microsoft Sidebar and Windows Live Spaces, Netvibes [26] has Netvibes UWA and there are many more to choose from. There are also some widget containers available (e.g. Apache Wookie [20] and Apache Shinding [9]) to host widgets in any web page rather than just in a particular mashup platform. In the Figure 2.1.1 there is an overview of the most popular mashup platforms, widget API standards and widget containers.

Netvibes [25, 26] allows assembling widgets, feeds, social networks, email, videos and blogs on one customizable page. It uses the Universal Widget API (UWA) for building widgets for their mashup platform. The UWA supports iGoogle, Apple Dashboard, Windows Live Spaces, Opera and some more as a publishing platform meaning that Netvibes widgets are portable to other platforms not just Netvibes platform. Netvibes has also made their technology publicly available for everyone to download their UWA JavaScript Runtime for running UWA widgets and PHP Libraries for handling server operations rather than relying on Netvibes infrastructure.

Yahoo! Pipes [46, 39] allows mashing up pipes from different sources (e.g. JSON, RSS, Atom feeds) without programming and by drag and dropping components to a workspace. The Visual development environment is based on dragging pipes from a toolbox and dropping them in work space, specifying data inputs, interconnecting widgets through pipes and finally specifying data output formats. Yahoo! Pipes is a quite data-oriented approach of building mashups by directly combining different data feeds, unlike other mashup platforms which combine available widgets.

Intel Mash Maker [21] is a browser extension that capable of learning what information user is interested in and creating personalized mashups. Mash Maker allows building mashups by combining content from multiple sources such as web content, videos, maps, RSS feeds and photos. It adopts mashups to the user behavior and suggests mashups that it thinks the user would like based on the user past behavior and the behavior of other users. Intel Mash Maker has its separate open Widget API for creating widgets. Each widget can see data added by other widgets, allowing composing a collection of widgets into a Mashup. Mash Maker also supports use of Google Gadgets but they cannot do as much as Mash Maker widgets. Nevertheless, Intel Mash Maker's expressive power is limited since it doesn't support RSS and Web APIs

IBM Mashup Center [19, 7] is an enterprise mashup platform allowing creation of widgets and combining them to new mashups. IBM has many different tools like WebSphere sMash, WebSphere Portal, Lotus Widget Factory, MashupHub and Lotus Mashups for creating and storing widgets and mashups. MashupHub is a Web-based editor for creating, storing and transforming feeds from different sources (e.g. XML, SQL queries, spreadsheets) that can be used in IBM mashups. Lotus Mashups (formerly called IBM QEDWiki) is a lightweight mashup environment for assembling personal, enterprise and Web content into simple applications [19] that provides internal communication between widgets.

Kapow Mashup Server [36, 7] is a mashup platform for enterprise mashups and the OpenKapow is a free community version of the mashup platform with some limitations. Kapow widgets are called Robots that are individual mashups with variety of data outputs and with communication capabilities between other Robots. The Kapow Web Data Server [37] is capable of wrapping Web applications into data feeds and RoboMaker [37] is a visual scripting application to build Robots by visual process flow steps.

WSO2 Mashup Server [45] is open source platform for mashups to acquire data from a variety of sources including Web Services, HTML pages and feeds, and process and combine it with other data. The Mashup Server enables recursively mashing up services, meaning that a mashup can be consumed by another mashup.

Google Gadgets [11, 39] are widgets for platforms like iGoogle, Google Apps, Google Desktop, Google Maps, Google Toolbar, Orkut, Blogger, Google Calendar, Google Spreadsheets, Gmail, and Google Sites. There are also third-party platforms for Google Gadgets like MyAOL, IBM websphere portal, Red Hat JBoss portal, SUN portal, and BEA weblogic portal. It is possible to deploy widgets on third party Web pages but with some restrictions as the options available to widget publishing are not standardized.

Apache Shindig [9] provides infrastructure for hosting OpenSocial widgets on third-party websites. It was originally started by Google but it is now an Apache project. It provides necessary JavaScript libraries and an application server to create, store and host OpenSocial compatible widgets.

Apache Wookie [20, 44] is an open source Java server application that allows uploading and deploying widgets to a local server for hosting them in Web applications. It includes a Widget Engine together with a plug-ins for popular Web applications such as Wordpress, Moodle and ELGG. Wookie uses the W3C Widgets Packaging specification and widgets can be imported directly to Wookie Server if they are in this format. It also supports Google Gadgets and OpenSocial widgets. Widgets can have collaborative or social functionality by making use of the Wookie shared data API which provides methods for storing and accessing data that can be shared among all instances of widgets that share a common context.

useKit [32] is a software platform that allows users to add individual selected functionalities to any Web site without installing software. It allows manipulating content, presentation and behavior of a Web site and mixing it with content or functionality coming from other Web sites. It focuses on personalized applications and services that can be applied to any Web site.

Ousia Weaver [47] provides a visual editor which enables users to create and publish mashups without writing code. Users can visualize a mashup results by using desired visualization widgets. It also involves a simple Web server which automatically publicizes mashup results on the Web. It provides visualization widgets to visualize mashups, data transformation operators to add attributes to data and transform data to visual form, and a mashup server with a simple Web server functionality to automatically publish mashups results on the Web. User can define how a mashup collects, combines, and processes data. Users create mashup data-flows which represent rules of collecting, combining and processing data, and in the visualization phase, users define how to visualize the result obtained by the mashup data-flows. Ousia Weaver is similar to Yahoo! Pipes, but with Yahoo! Pipes it is impossible to publish mashup results as independent Web pages and its operators cannot be extended by users.

Widgets on desktop and mobile devices are similar to Web widgets but are meant to run on widget engines on desktop and mobile environment. The most popular mobile and desktop runtime environments for widgets are Opera Widgets [24] for various mobile devices and operating systems, Apple Dashboard [5] for Mac OS, Windows Sidebar [23] for Windows operating systems, Google Desktop [12] (for Windows, Linux and Mac OS), and BONDI [29] widgets for mobile devices. Widget platforms for desktops and mobiles are not limited to browser's security restrictions (that Web widgets are limited with) and can provide additional functionality that Web widgets cannot (e.g. a widget can access physical resources of a device and show memory and CPU usage or access camera of a mobile phone). Mobile devices set additional requirements on widgets because of smaller screen and difficult navigation of small devices.

Opera widgets [24, 39] can be used in every device (desktop, mobile, TV) where the Opera Widgets runtime is installed. Opera Widgets are compatible with the W3C Widgets 1.0 specification and are not being locked into the Opera specific technology. Opera Widgets' security model prohibits any information sharing between different widget insistences meaning that separate widget instances cannot share any information (e.g. no sharing of settings or cache) and accessing other widgets is not allowed. This means that it is not possible to have any collaboration between widgets to take place on the Opera platform.

BONDI [29] is a widget runtime environment for mobile devices that uses existing W3C widgets 1.0 standard with additional standard for Web interfaces and security. BONDI widgets can be run on all devices that are using BONDI runtime environment. It provides an API for widgets to access resources on a device (e.g. use of the camera, location or contact details) so that a widget could run on every mobile device without modifications.

2.2 Related Work on Semantic Integration

There is a need for integration of exchanged data in mashups, because mashups use data from different sources and widgets communicate using different data structures. This problem is being researched in the field of semantic integration. An introductory article [28] about semantic Integration states that it is a field related to problems that arise with sharing data across different sources which requires solving many problems, such as matching ontologies or schemas, detecting duplicate values, reconciling inconsistent data values, modeling complex relations between concepts in different sources, and reasoning with semantic mappings.

An article [31] about schema matching indicates that a fundamental operation in the integration data is matching, which takes two schemas as input and produces a mapping between elements of the two schemas that correspond semantically to each other. The semantic integration introductory article [28] agrees that one of the main problems in semantic integration is establishing semantic correspondences (also called mappings) between vocabularies of different data sources and asks if two ontologies, two database schemas, or any other structured resources are given, then how to determine which concepts are similar or related? The article continues by naming techniques for integrating different data sources, which are linguistic analysis of terms, comparison of graphs corresponding to the structures, mapping to a common reference ontology, use of heuristics that look for specific patterns in the concept definitions, and machine learning.

Correspondences between two sources must be represented in a machine-interpretable way and according to the article [28], those correspondences must then be used for specific integration tasks, for example representing mappings as instances in an ontology of mappings, defining bridging axioms in first-order logic to represent transformations, and using views to describe mappings from a global ontology to local ontologies. The article [28] then continues by naming integration tasks to use after the resources have been correlated, which are data transformation from one source to another, merging of ontologies and schemas, robust reading of natural text, query and data mediation in peer to peer settings, and data integration.

Although there are many proposed approaches (some of them analyzed in [31, 43, 27]) for automatic integration of data from various sources, then this thesis proposes a solution for a manual matching of data elements that would allow automatic data integration. Ontologies are used in defining mappings between similar data elements where references to ontology concepts (e.g. references to OWL[1] classes and attributes) that encode meaning about data elements are used (i.e. data is linked with ontologies to allow integration of information).

[1]Web Ontology Language (OWL) is a family of knowledge representation languages for describing ontologies [40].

Chapter 3

Semantic Integration of Exchanged Data

Current mashups available are not compatible with each other. There is a lack of standards and platforms that would provide collaboration capabilities between various widgets. Current widgets do relatively well in displaying information from various sources but that is not enough for composing more advanced application logic in mashups. Piping information through many different widgets in a mashup is a difficult task. Current mashup platforms that provide collaboration between widgets within their platform are using vendor-specific solutions that restrict collaboration between widgets provided by third parties.

It would be reasonable to have communication on the presentation layer (using JavaScript in a browser) as shown in the Figure 3.0.1. Presentation layer would have its own hub for handling communication between various widgets which would allow collaboration between widgets that are hosted by various vendors. Communication would not have to rely on a central server because the communications' hub would be based on the presentation layer. That would allow loosely coupled components hosted at different servers to be combined together to collaborate with each other without a central server.

Widgets could be held in secure `IFrame` containers limiting their access to the rest of the Web page which would increase security of mashups and would

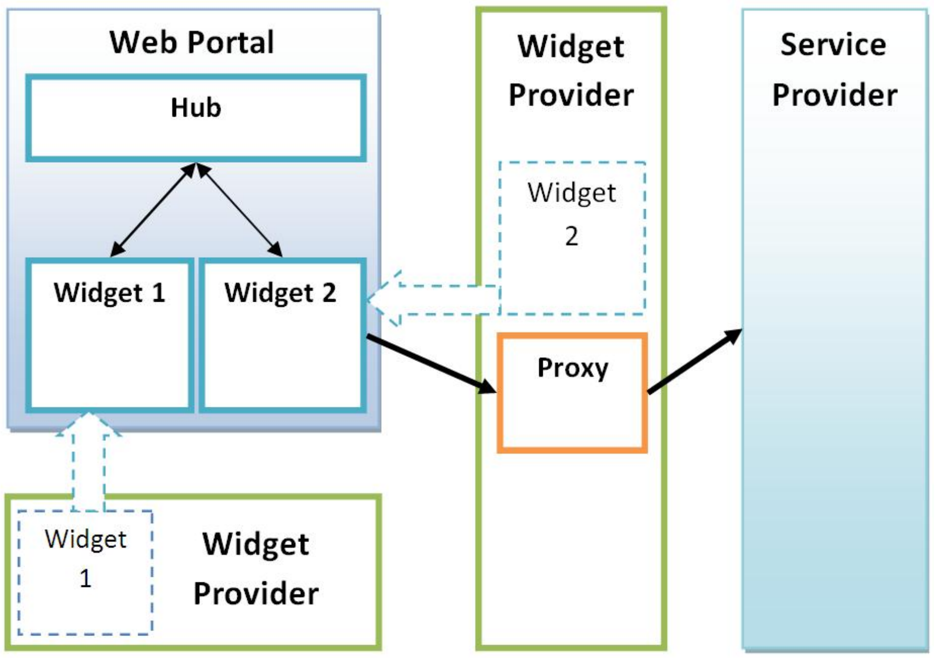


Figure 3.0.1: Widgets communication with other widgets and external services

allow the use of widgets that are not trusted. Widgets would communicate with each other only through the communications' hub and malicious widgets would not be able to access rest of the application.

Widgets that are loaded to the Web portal from different domains would not be able to manipulate properties of the rest of the application. That is due to the browsers' Same Origin Policy [33] that sets additional access restrictions to scripts loaded from other domains. It is also true for document retrieval when the `XmlHttpRequest` method is used [34], meaning that widgets that are loaded from other domains will not be able to retrieve documents that are not in their domains.

The Same Origin Policy can often be too restrictive for widgets that want to combine services from various sources outside their domain. A workaround to this restriction is to have a proxy to mediate information between domains, as shown in the Figure 3.0.1. Widget providers would be able to provide proxies in their domains for their widgets which would be able to access external services through the proxy.

OpenAjax Alliance has released OpenAjax Hub 2.0 [2] which provides architecture and standards for widgets to exchange messages through a hub in a Web application as discussed above. It provides a communication platform for mashups and allows widgets to communicate with each other but does not solve all the problems with collaboration. Widgets, which are not created by the same vendor, may use different data structures and formats and would therefore not be able to communicate directly with each other.

The problem can be illustrated in a following example where we would have a mashup with two widgets which are created by different vendors. Let us say that the first widget would be an event listing widget for showing a list of events and the other widget would be a map widget for showing objects on the map. If we would like to make those widgets to collaborate and share information with each other so that events from the events listing widget would show up as objects on the map in the map widget and, vice versa, the events listing widget would show list of events relevant to the geographic area to where the user has zoomed in on the map, then it would be quite difficult. Since those widgets would have been created by different vendors then they would be unaware of each other and would use different data structures in their messages. Messages sent by one widget would not be interpretable by other widget. The use of different data structures makes it hard for widgets to interact with one another.

A solution to this problem would be to translate a message sent from widget A to a format that would be interpretable by widget B. It would require a component that would listen to all exchanged messages and transform them to formats interpretable by all widgets.

The problem with translating messages directly from one widget to another is that messages from widget A may contain only partial data necessary to form a message for widget B. Widget B may require some data that widget A does not provide but, let us say, widget C provides instead. We would need to aggregate data from widget A and widget C to form a message useful to widget B. Instead of just directly translating messages from one widget to another, we would have to aggregate data from various sources and form new messages by picking useful data from aggregated data and transforming

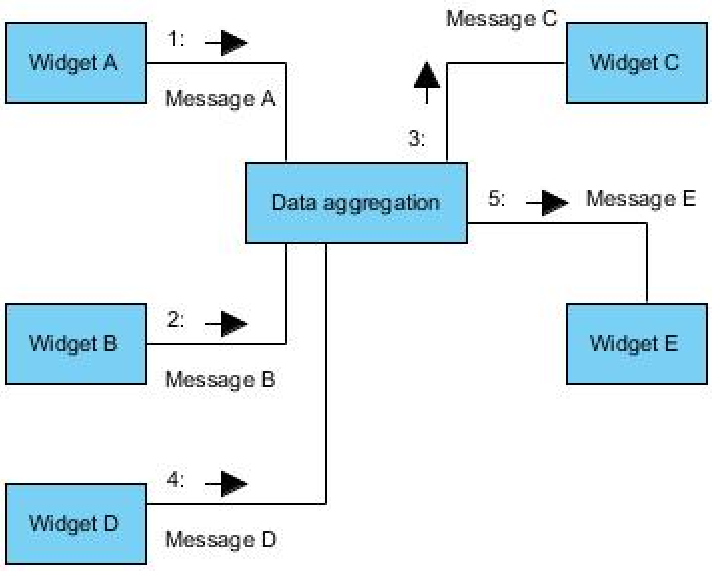


Figure 3.0.2: Data aggregation

it to formats that are interpretable by different widgets.

In the Figure 3.0.2 the messages are transformed in the data aggregation component that receives messages from widget A, widget B and widget D, and, based on aggregated data, composes new messages that are sent to widget C and widget E.

Aggregating data from various sources (from widgets connected to the hub) and transforming this data to messages readable to all widgets means that the component which is responsible for such transformations (let us call it the transformer) would have to understand the semantics and structure of the data that is being exchanged between widgets. The transformer component should be able to link atomic data elements from exchanged messages with other potential messages which are useful to widgets so that if a widget is linked with enough atomic data elements then a new message could be composed to that widget.

In the Figure 3.0.3 the message to widget C is formed from messages from widgets A and B. A message from the widget A contains two atomic elements of data (A1 and A3) that can be used as C1 and C2 when composing a message to the widget C, and a message from widget B contains one atomic

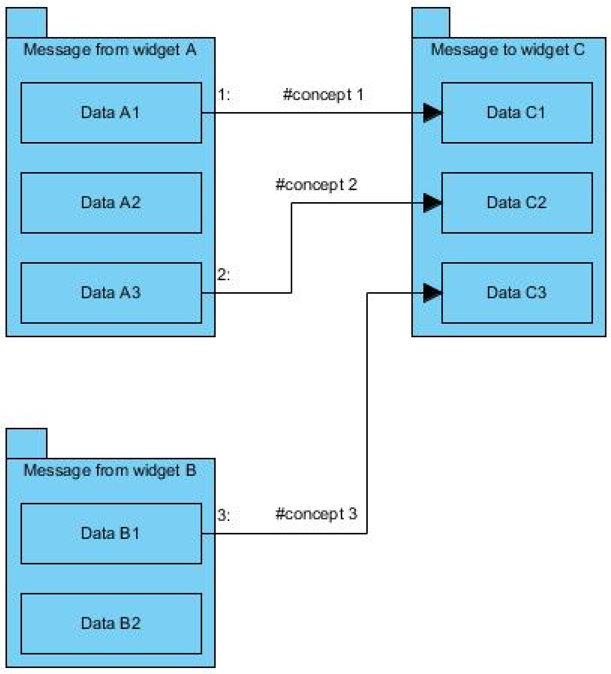
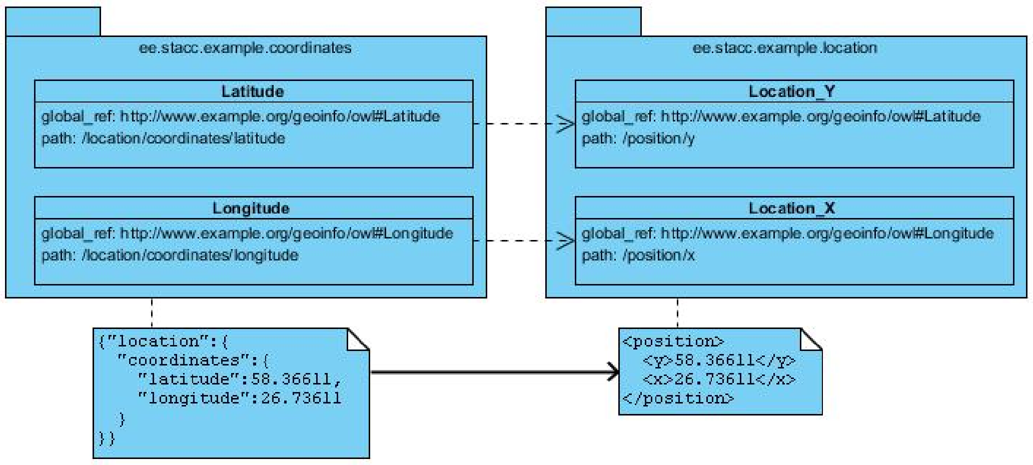


Figure 3.0.3: Message composition

data element B1 that can be used as C3 when composing a message to the widget C.

This means that we would have to describe the semantics of each message that is being exchanged in the hub. It can be done by linking contents of a message with ontologies so that each atomic element of data in the message is given specific meaning in the context of ontology. Atomic elements of data in messages can be linked with specific ontology concepts that carry certain meaning within them. This allows semantic integration of data from different sources by picking atomic data elements according to their related concept in ontology and then forming messages from those data elements. If message A and message C both contain an atomic data element that refers to the same concept in ontology then they both carry a part of the same meaning within them and message C can be formed from the data in message A.

To describe the semantics of each message we have to map the elements in a message to elements of ontologies. Ontologies can be described using Web Ontology Language (OWL) [40] that represents knowledge about


ee.stacc.example.coordinates
Latitude
global_ref: http://www.example.org/geoinfo/owl#Latitude
path: /location/coordinates/latitude
Longitude
global_ref: http://www.example.org/geoinfo/owl#Longitude
path: /location/coordinates/longitude
ee.stacc.example.location
Location_Y
global_ref: http://www.example.org/geoinfo/owl#Latitude
path: /position/y
Location_X
global_ref: http://www.example.org/geoinfo/owl#Longitude
path: /position/x

```
{"location":{
  "coordinates":{
    "latitude":58.36611,
    "longitude":26.73611
  }
}}
```

```
<position>
  <y>58.36611</y>
  <x>26.73611</x>
</position>
```

Figure 3.0.4: Semantic integration

a domain. Atomic data elements can be linked to ontology elements (instances of OWL classes or properties) using URI-s. For example, an URI of `http://www.example.org/geoinfo/owl#Latitude` would represent a reference to an OWL concept of latitude that represents a part of geographical coordinate. If two widgets would both be exchanging geographical coordinates in their messages and if those messages would both have references to the same OWL concept of latitude, then it would be possible to translate those messages interpretable to both of these widgets. This would allow communication to take place between widgets even if widgets are unaware of each other.

Descriptions of messages' semantics can be held in a separate file that maps the atomic data elements of each message with corresponding OWL classes that represent the meaning of those data elements. This approach would allow automatic interpretation of messages that would make aggregation and transformation of data possible.

In the Figure 3.0.4 there is an example of mappings of two messages which both contain geographical coordinates. Let us say that there is widget A that can process coordinates and accepts messages that contain coordinates in an XML document similar to the one in the bottom right. There is also widget B that publishes messages containing coordinates in JSON format similar to the message in the bottom left. They both deal with coordinates but use different data structures in their messages.

We can describe the semantics of those messages similarly to the example above by mapping all the atomic data elements with corresponding OWL classes and their physical locations in messages. In the example, both messages contain two atomic data elements, latitude and longitude. An atomic data element is described with an URI of the corresponding OWL class and path to the location of the data element in the message. Reference to an OWL class (marked as `global_ref`) is used to specify the meaning of a data element, and absolute path (marked as `path`) is used to specify the location of an atomic data element.

In the example above, the message on the left contains two atomic data elements: latitude and longitude. The first atomic data element mapping, latitude, has `global_ref` URI value `http://www.example.org/geoinfo/owl#Latitude,` which is the reference to the corresponding OWL class Latitude that makes it possible to interpret that data element as a part of coordinates' pair. Also, the data element mapping has attribute `path` referring to location `/location/coordinates/latitude,` which points to the exact location where the atomic data element is located in the message, allowing retrieval of the data element from the message. The second data element mapping in the example is longitude which has `global_ref` referring to `http://www.example.org/geoinfo/owl#Longitude` and `path` pointing to `/location/coordinates/longitude`. This shows where the longitude, a second part of coordinates' pair, is located.

This information is sufficient to map messages' content with ontologies and attach meaning to data elements in messages. Data element from one message can be used in forming another message if both of them have the same OWL class referred in their mappings. In the example above both messages use references to the same OWL classes `http://www.example.org/geoinfo/owl#Latitude` and `http://www.example.org/geoinfo/owl#Longitude` which means that one message can be formed from the data in another message.

New messages can be generated using schemas (XSD schemas for XML documents and JSON schemas for JSON documents) that describe the structure of those messages. That makes it possible to generate messages in every format with different structures using mappings and schemas.

Chapter 4

Transformer Widget

Transformer Widget is an application to solve semantic integration problems in mashups that are using OpenAjax Hub 2.0 for exchanging messages between widgets. It is an invisible widget that is connected to the hub like any other widget, but unlike regular widgets, it gathers all the exchanged data sent by widgets through the hub. It aggregates data sent from all the widgets, composes new messages based on the aggregated data and mappings, and sends generated messages to widgets that are interested in particular data.

Widgets exchange data through hub by publishing messages under specific topics. Widgets that are interested in particular messages subscribe to topics where those messages are exchanged. Unfortunately, widget providers do not know all the topics and data structures of messages that could be exchanged in mashups by other widgets. That makes direct collaboration

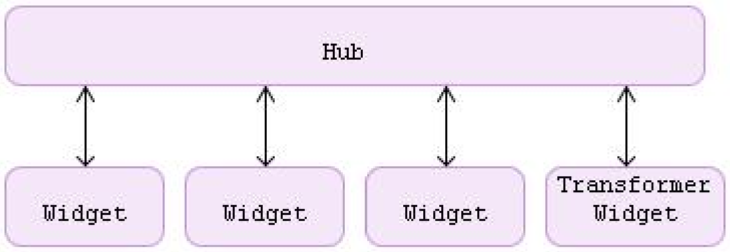


Figure 4.0.1: Widgets connecting to the hub

between widgets impossible unless the whole application and its widgets are developed by the same provider who has knowledge and control over all of his widgets. If third party widgets are used in a mashup and there is a need for collaboration then an additional aggregation component like Transformer Widget is needed to support collaboration. Transformer Widget can be added in the same way as any other widget in a mashup (as shown in the Figure 4.0.1) to make data aggregation and collaboration between widgets possible.

Transformer Widget uses special mappings in an XML file to interpret semantics behind data packages that widgets exchange. The mappings XML file describes the structure and semantics of a data packages in each topic so that the transformer widget can understand which data elements can be useful to other widgets. If the transformer widget has collected enough data elements to form a new data package, then it composes all the necessary data together and forms a new data package in a format that is interpretable to the widget interested in that particular message. The data package is then published through the hub to the widget.

4.1 Mappings Configuration

To enable data aggregation and messaging in the Transformer Widget, a `mappings.xml` file has to be configured to include all the semantic mappings of messages exchanged by the widgets. The configuration file has to include mappings of all the messages each widget publishes or receives.

For example, let us say that there is a widget that is capable of receiving geographical coordinates in JSON format as shown in the Example 4.1 on the following page. The message shown in the Example 4.1 holds one pair of coordinates: latitude of 58.36611 and longitude of 26.73611 meaning that there are two atomic data elements. Let us say that the widget has subscribed to the topic `ee.stacc.coordinates` in order to receive similar messages.

To map the information shown on the Example 4.1 to the mappings configuration file, we would have to add the following lines with XML notation to the `mappings.xml` file as shown in the Example 4.2 on the following page.

Example 4.1 An example message in JSON format of geographical coordinates.

```
{location:{
  coordinates:{
    latitude:58.36611,
    longitude:26.73611
  }
}}
```

Example 4.2 Mappings configuration corresponding to the JSON object shown in the Example 4.1

```
<frame>
  <topic>ee.stacc.coordinates</topic>
  <format>json</format>
  <schema>schemas/coordinates.js</schema>
  <mappings>
    <mapping>
        <global_ref>http://www.example.org/geoinfo/owl#Latitude</global_ref>
        <path>/location/coordinates/latitude</path>
        <default>26.73611</default>
      </mapping>
      <mapping>
        <global_ref>http://www.example.org/geoinfo/owl#Longitude</global_ref>
        <path>/location/coordinates/longitude</path>
      </mapping>
      <constant path="/location/coordinates/srs" value="EPSG:4326" />
    </mappings>
</frame>
```

In the mappings configuration's XML notation one `frame` represents meta-data about messages that are sent under single topic. It is assumed that all messages exchanged under one topic follow the same structure otherwise it would be difficult for widgets to parse those messages. The `topic` represents a name of a channel that widgets use in exchanging particular data and is specified in the topic element in the XML notation.

The `format` element specifies the data format which is used in those messages. Currently, JSON and string data formats are available to use, but it is possible to add support to many other data formats like XML, CSV etc.

The `schema` element specifies the location of the schema which describes the structure of the messages exchanged under that topic. This schema is used by the Transformer Widget to generate messages on aggregated data. If there are no widgets that are subscribed to that topic and there are some widgets that only publish messages under that topic, then the schema is not necessary and can be left out, because it is not necessary to generate messages if there are no widgets to receive them. But even if widgets are only publishing messages under the topic and no one is listening, then it is still necessary to add the mappings to the configuration file so that the Transformer Widget would know how to aggregate data from those messages.

It is important to note that the physical location of those schemas must be in the same domain with the Transformer Widget due to the Web Browsers' Same Origin Policy, unless a separate proxy is used. This basically means that schema files must be in the same folder (or in a subfolder) where the transformer widget is located.

The `mappings` element contains mappings for each atomic data element in messages exchanged under the topic. Each atomic data element's mapping is specified in a separate `mapping` element in the mappings element group. A `mapping` element contains two mandatory attributes `global_ref` and `path`.

The `global_ref` attribute in a `mapping` element refers to the OWL class that defines the meaning of the atomic data element. All the atomic data elements that contain the same kind of data should refer to the same OWL class. For example, if some messages contain atomic data elements with co-ordinates, e.g. latitude, then they should all refer to the same OWL class

Example 4.3 A simple message in XML format.

```
<location>
  <coordinates>
    <latitude>58.36611</latitude>
  </coordinates>
</location>
```

`http://www.example.org/geoinfo/owl#Latitude` to notate that all those atomic data elements contain information about latitude. This makes it possible for the Transformer Widget to aggregate and distribute data properly.

The `path` attribute in a `mapping` element notates the location (absolute path) of the atomic data element in messages published under the specified topic. The `path` attribute is used to locate atomic data elements in received messages and it is also used to insert atomic data elements to new messages that are created according to schemas. The delimiting character used to distinguish elements in a message is slash (`/`). As an example, a typical path would look like `/location/coordinates/latitude` where the `latitude` would notate the atomic data element that is located in a `coordinates` element which itself is located in a `location` element. A simple message in XML format that would correspond to such structure would look as shown in the Example 4.3.

An additional parameter `default` can be added to a mapping to specify the default value of the atomic data element. The default value is used in message creation when no data has been aggregated which would correspond to the same OWL class. This also means that the message can be created before any data has been aggregated that would correspond to the OWL class specified in the `global_ref` attribute.

Messages in the JSON data format also support separate default values in JSON schemas (based on the JSON schema standard) if not defined in the mappings configuration. When the Transformer Widget generates a new message from a JSON schema and discovers a data value that does not have corresponding data value in the aggregated data that the Transformer Widget could use in generating the message, then it uses the value defined in the default element. The following example illustrates the use of default values in

Example 4.4 Mappings containing repeating element groups.

```
<frame>
  <topic outgoing_only="true">ee.stacc.coordinates.list</topic>
  <format>json</format>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/lang/owl#Name</global_ref>
      <path>/location/placename</path>
    </mapping>
    <repeating_element_group path="/location/coordinates">
      <mapping>
        <global_ref>http://www.example.org/geoinfo/owl#Latitude</global_ref>
        <path>/location/coordinates/latitude</path>
      </mapping>
      <mapping>
        <global_ref>http://www.example.org/geoinfo/owl#Longitude</global_ref>
        <path>/location/coordinates/longitude</path>
      </mapping>
    </repeating_element_group>
  </mappings>
</frame>
```

JSON schemas: `{"z":{"type":"number", "default":1235}}`. In the example, the JSON object contains one element z which is a numeric type with the default value of `1235`. The value `1235` is used when the corresponding data value for z is not found from the collection of atomic data values the Transformer Widget has aggregated.

The element `constant` in the mappings element can be used to specify constant values in messages. It contains an attribute `path` and an attribute `value` where the `path` is used to specify the location of the constant value in the message, and the attribute `value` is used to specify the constant value of the atomic data element in the message.

If messages contain repeatable data elements (e.g. in arrays) than these repeatable elements must be specified in the element `repeating_element_group` that is added inside the mappings element.

An example of a mappings configuration of a message for sending a list of multiple coordinates (i.e. repeating data elements as coordinates) is shown

Example 4.5 JSON object containing an array.

```
{location:
  { placename:"Railway Station",
  coordinates:[
    {latitude:58.36611, longitude:26.73611},
    {latitude:59.45723, longitude:27.24301},
    {latitude:60.54561, longitude:27.53413}
  ]}
}
```

in the Example 4.4 on the preceding page. An example message in JSON format that would correspond to such mapping configuration is shown in the Example 4.5.

In the Example 4.5 is shown an array of three coordinate pairs of latitude and longitude that represent geographical coordinates of three railway stations. There is an array of similar elements (i.e. repeatable data elements), three coordinates' pairs, that has to be represented as a separate element `repeating_element_group` containing mappings of those atomic data elements that are represented in the array. If the message contains more than one array then a separate `element repeating_element_group` must be specified for each array.

The `repeating_element_group` element must contain a `path` attribute that specifies the location (absolute path) of the array in a message (the delimiting character used to distinguish elements in a message is the slash (`/`) symbol). Note that each mapping's path attribute in the repeating element group must begin with the same path as the repeating element group's path because those data elements are contained physically inside the array.

Note that in the Example4.4 there is no schema specified. That is because in the element `topic` has an attribute `outgoing_only` with a value of `true` meaning that there are no widgets that are subscribed to that topic (in this example `ee.stacc.coordinates.list`) and therefore there is no need to generate these messages by the Transformer Widget and there is no need for a schema for the message. Even though there are no widgets receiving those messages it is still important to add the mappings to the mappings

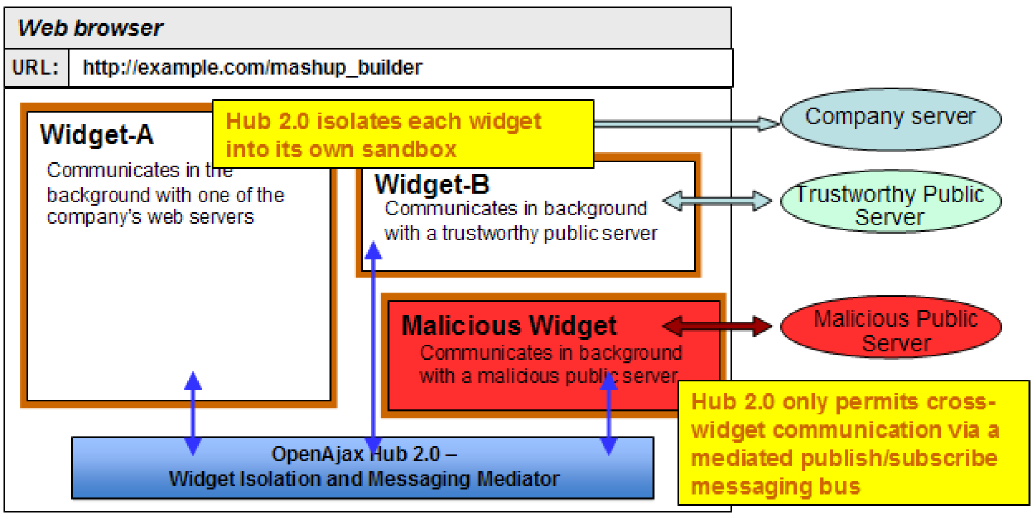

Web browser
URL:
http://example.com/mashup_builder
Widget-A
Communicates in the background with one of the company's web servers
Hub 2.0 isolates each widget into its own sandbox
Widget-B
Communicates in background with a trustworthy public server
Malicious Widget
Communicates in background with a malicious public server
OpenAjax Hub 2.0 –
Widget Isolation and Messaging Mediator
Company server
Trustworthy Public Server
Malicious Public Server
Hub 2.0 only permits cross-widget communication via a mediated publish/subscribe messaging bus

Figure 4.2.1: OpenAjax Hub 2.0, taken from [2]

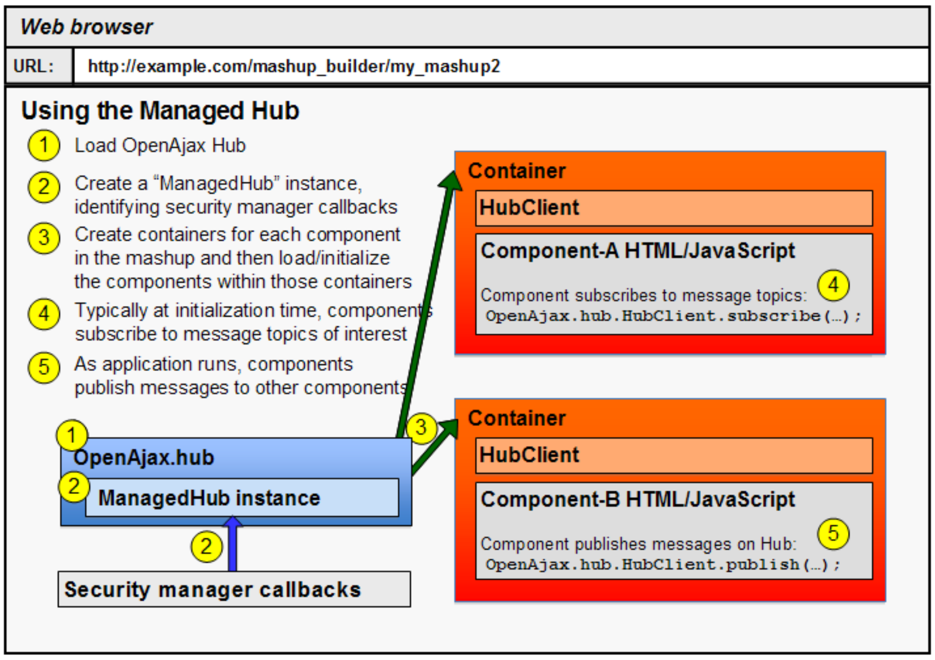
configuration file if there is at least one widget that publishes messages under that topic because then it is possible for the Transformation Widget to aggregate data from those messages and use that data to compose messages to other widgets.

4.2 Installing Widgets

The Transformer Widget uses OpenAjax Hub 2.0 [2] architecture which defines a standard for how widgets can be isolated into secure containers and how widgets can communicate with each other through a messaging hub.

OpenAjax Hub 2.0 is a JavaScript library [2] for mashups that can isolate third party widgets into secure containers and provide messaging capabilities for those widgets (as shown in the Figure 4.2.1 [3]). OpenAjax Hub 2.0 allows widgets to communicate with each other through the hub but isolates widgets to `IFrame` elements to prevent them from accessing other components in the mashup.

The Openajax hub has a security manager [2] within the hub to control messaging between widgets. Every widget attempt to publish or to subscribe to topics is controlled by the security manager's logic [3]. The ap-


Web browser
URL:
http://example.com/mashup_builder/my_mashup2
Using the Managed Hub
Container
HubClient
Component-A HTML/JavaScript
Container
HubClient
Component-B HTML/JavaScript
OpenAjax.hub
ManagedHub instance
Security manager callbacks
1
2
3
4
5

1 Load OpenAjax Hub
2 Create a "ManagedHub" instance, identifying security manager callbacks
3 Create containers for each component in the mashup and then load/initialize the components within those containers
4 Typically at initialization time, components subscribe to message topics of interest
5 As application runs, components publish messages to other components

Component subscribes to message topics:

```
OpenAjax.hub.HubClient.subscribe(…);
```

Component publishes messages on Hub:

```
OpenAjax.hub.HubClient.publish(…);
```

Figure 4.2.2: Managed Hub Initialization, taken from [3].

plication must provide security manager callback methods (`onPublish` and `onSubscribe`) that implement access control policy for messaging between the widgets [3].

The Figure 4.2.2 [3] provides a conceptual overview for how an application initializes the hub (called Managed Hub) and containers for widgets (called Client Applications) in the Web page [3]. First, the Managed Hub instance has to be created along with security manager's callback methods. After the hub has been created, the containers for widgets have to be created. The containers then load and initialize widgets on the page and while the widgets are being initialized they subscribe (and publish) to topics they are interested in.

If an application has finished initializing hub and widgets, then the widgets can start passing messages to each other using the OpenAjax Hub's publish and subscribe API [3]. The Figure 4.2.3 [3] shows how a message from one widget (Component-B) is sent to another widget (Component-A). Component-B is located in an `IframeContainer` and Component-A is lo-

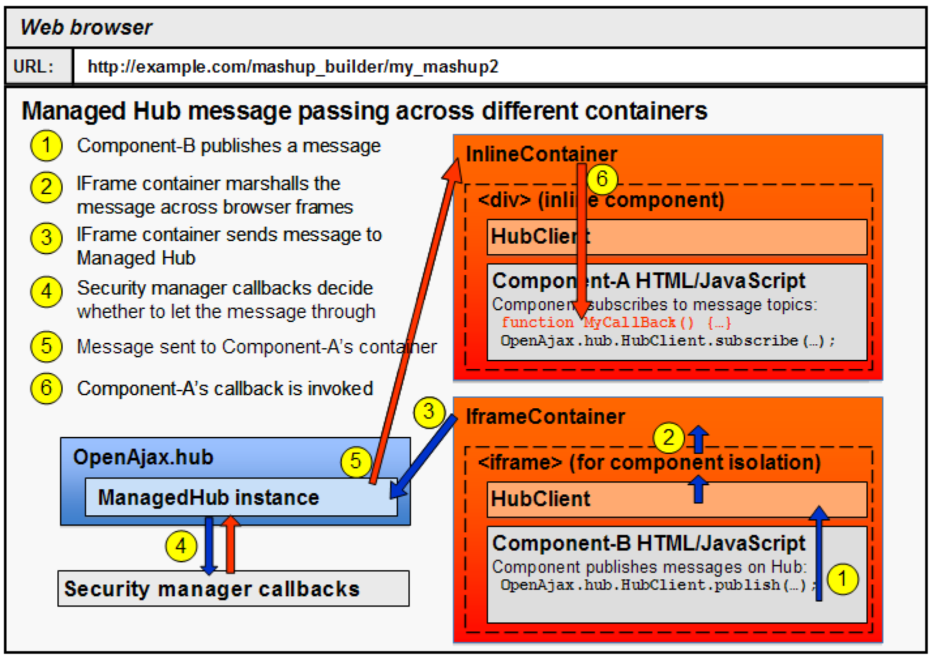

Web browser
URL: http://example.com/mashup_builder/my_mashup2
Managed Hub message passing across different containers
1 Component-B publishes a message
2 IFrame container marshalls the message across browser frames
3 IFrame container sends message to Managed Hub
4 Security manager callbacks decide whether to let the message through
5 Message sent to Component-A's container
6 Component-A's callback is invoked
InlineContainer
<div> (inline component)
HubClient
Component-A HTML/JavaScript
Component subscribes to message topics:
function MyCallBack() {…}
OpenAjax.hub.HubClient.subscribe(…);
IframeContainer
<iframe> (for component isolation)
HubClient
Component-B HTML/JavaScript
Component publishes messages on Hub:
OpenAjax.hub.HubClient.publish(…);
OpenAjax.hub
ManagedHub instance
Security manager callbacks

Figure 4.2.3: Messaging in OpenAjax Hub from, taken from [3].

cated in an `InlineContainer` (presumably because, from the security perspective, the Component-A is more trustworthy widget than the Component-B).

The `InlineContainer` holds widgets in a HTML element (like `div` for example) which tends to be faster and require less memory than the `IFrameContainer` where widgets are placed to an `IFrame` element [3]. However, `IframeContainer` does not isolate widgets and therefore it should be used only with trusted widgets because widgets then have complete access to the Web application [3]. Any widget that may contain malicious script should be isolated from the application by being placed in an `IframeContainer`.

The primary feature of the OpenAjax Hub is publishing and subscribing functionality that allows different components to send messages to each other. It supports broadcasting anonymous messages within the mashup. Widgets that publish messages are unaware of how many widgets have subscribed to the same topic and widgets that receive messages do not know which widgets published those messages.

Components can broadcast messages by invoking the hub's publishing [3] function with topic and data as parameters. The topic parameter is a string specifying the common name for the messages published under the topic, and the data parameter specifies the message (a JavaScript object) that is being sent.

Topic names are expressed by tokens separated by the dot (`.`) character. An example of a topic name is `org.example.location` which could be used to send locations (e.g. coordinates). Widgets can use special characters for more dynamic subscription of topics. The wildcard character, asterisk (`*`), can be used to subscribe to all the subtopics of a particular topic [3]. For example, we could subscribe to all of the subtopics of the topic `org.example` by using the asterisk as in the following example `org.example.*` in which we would receive messages sent both under the topics `org.example.location` and `org.example.coordinates`.

4.2.1 Requirements for Widgets and Portals Using OpenAjax Hub 2.0

OpenAjax Hub supports widgets that are either built as separate HTML pages held in `IFrame` containers, or as snippets of code that are held in inline containers. Regardless of which container is used, the requirements for widgets are largely the same. The widget has to create a client instance (either `IframeHubClient` or `InlineHubClient` instance), define a method for handling security alerts, and connect to the hub.

An example of a widget (based on [3]) in an `IFrame` container will be constructed in the following way. In the Example 4.6 on the following page is shownhow the widget creates an `IframeHubClient` instance `hubClient` with a parameter of a `HubClient` object. The `HubClient` object takes a parameter `onSecurityAlert` with a reference to the method's name that is called when security alerts are raised.

If a widget is held in an inline container, then the client initialization is similar to the `IFrame` container's initialization, but the `InlineHubClient` instance should to be created instead. An `InlineHubClient` object has to

Example 4.6 Initialization of a `IframeHubClient` instance [3].

```
hubClient = new OpenAjax.hub.IframeHubClient({
  HubClient: {
    onSecurityAlert: clientSecurityAlertHandler
  }
});
```

Example 4.7 Initialization of an `InlineHubClient` instance [3].

```
var hubclient = new OpenAjax.hub.InlineHubClient({
  HubClient: {
    onSecurityAlert: clientSecurityAlertHandler
  },
  InlineHubClient: {
    container: container1
  }
});
```

be created and passed as a parameter to the `InlineHubClient` in addition to the `HubClient` object (with the `onSecurityAlert` parameter) that is created similarly when the `IframeHubClient` is initialized. The `InlineHubClient` object should have a parameter called container that should reference to the container instance to which the `HubClient` would connect. In the Example 4.7 is shown a creation of an `InlineHubClient` instance.

The problem with managing widgets in inline containers is that the container's name is known only to the portal creator and not known to the widget developer making inline containers not suitable for hosting third party widgets. In general, widgets should be held in `IFrame` containers unless a widget is developed in-house by the same developers that develop the Web portal that is hosting the widget.

After the hub client instance has been created (regardless of which container was used), it has to connect to the hub as shown in the Example 4.8 on the following page. The optional callback method (called `connectCompleted` shown in the Example 4.9 on the next page) can be provided which would be called after the connection with the hub has been established.

In the Example 4.9 on the following page, there is an optional callback

Example 4.8 Connecting to the hub [3].

```
hubClient.connect( connectCompleted );
```

Example 4.9 An example callback method called when the widget has finished connecting to the hub [3].

```
function connectCompleted ( hubClient, success, error ) {
  if (success) {
    // hubClient.publish(...)
    // hubClient.subscribe(...)
  }
}
```

method that would be called (if the method was specified) after the widget has finished connecting to the hub. The method `connectCompleted` would called when the widget has connected to the hub. The method has parameters `hubClient` which is a reference to the hub client instance, `success` which is a boolean value stating whether the connection was successful, and `error` specifying the error message used if the connection was not successful.

4.2.1.1 Widgets' Messaging in OpenAjax Hub 2.0

Widgets can subscribe topics to start receiving messages that are published under the topic. Widgets can also publish messages (objects) to other widgets under specific topics.

Widgets can publish messages using the method shown in the Example 4.10 on the next page (after the widget has initiated `hubClient` object and has connected to the hub). The parameter `topic` is a string specifying the topic of the message that is being published and the parameter `data` is a message object that is being published. The message can be any JavaScript object like `string`, `boolean` etc, including JSON objects with complex data structures. In the Example 4.11 on the following page is shown a message 'Hello World' which is sent with the topic `org.example.topics.textmessage`.

Widgets can subscribe to topics using the method shown in the Exam-

Example 4.10 Method for publishing messages [3].

```
hubClient.publish(topic, data);
```

Example 4.11 Publishing "Hello World" [3].

```
hubClient.publish('org.example.topics.textmessage','Hello World');
```

ple 4.12 (assuming that the widget has initiated `hubClient` object and has connected to the hub). The parameter `topic` specifies the topic to use when sending a message and the parameter `onData` refers to the method which is called every time the widget receives a message with that topic. The method `subscribe` can have additional optional parameters called `scope`, `onComplete` and `subscriberData` that can be specified when needed. The parameter `scope` refers to the JavaScript keyword `this` when `onData` or on-Complete callback method is called. The parameter `onComplete` refers to the callback method that is called when the subscribe operation has finished, and the parameter `subscriberData` can be used to provide data which is handed back to the `onData` callback function. A practical application to the `subscriberData` parameter is enabling event caching when using TIBCO PageBus extension to OpenAjax Hub. Event caching is explained in the section 4.2.1.2 on the following page.

In the Example 4.13 on the next page is shown where a widget subscribes to the topic called `org.example.topics.textmessage` and will start receiving all the messages that widgets publish under this topic. Second parameter `onData` is the name of the callback method that will be called every time the widget receives a message with that topic.

The callback method header for handling received messages is shown in the Example 4.14 on the following page. The `onData` is the callback method which name is specified in the subscribe method when subscribing to topics. If a widget subscribes to many different topics, then separate callback methods can be defined for each of those topics. In our examples the method name

Example 4.12 Subscribing to topics [3].

```
hubClient.subscribe(topic, onData);
```

Example 4.13 Subscribing to an example topic [3].

```
hubClient.subscribe('org.example.topics.textmessage', onData);
```

Example 4.14 Callback method header for handling received messages [3].

```
function onData(topic, data, subscriberData )
```

is `onData`. The parameter `topic` specifies the topic of the messages and the parameter `data` is the message object that was received by the widget. The parameter `subscriberData` is an object that was specified when the subscribe method was called.

In the Example 4.15, there is a callback method `onData` to handle received messages. In the example the method checks if the received message is a string object and then appends the message to the `div` element called `messageArea`.

4.2.1.2 Message Caching with Tibco PageBus

A widget can start publishing messages after it has been initialized. Since loading and initialization of widgets is an asynchronous process then the order of how widgets are being initialized is undetermined. This means that widgets are being loaded in a random order and it is never certain which widget finishes initialization first and which one finishes last.

When a widget finishes initialization and starts publishing messages before other widgets have finished initialization, then these messages are not received by the widgets that finished initialization later. This leads to loss of data exchanged by widgets during a Web page initialization.

Example 4.15 An example callback method for handling received messages [3].

```
function onData(topic, publisherData) {
  if (typeof publisherData === "string") {
    var messageArea = document.getElementById('messageArea');
    messageArea.innerHTML = publisherData;
  }
}
```

Example 4.16 Enabling event cache

```
hubclient.subscribe('org.example',onData,null,null,{PageBus:{cache:
true}});
```

To overcome the problem of loosing exchanged data during Web page initialization, TIBCO has extended OpenAjax Hub with PageBus [38] extension to support event caching so that the widgets would receive messages that were published before they were initialized. That would prevent any data loss if event caching is enabled.

To enable support of event caching with TIBCO PageBus the PageBus library has to be used instead of the native OpenAjax Hub library. To enable event caching in a particular topic the parameter `PageBus: { cache: true }` has to be added [38] to the subscriberData parameter object when subscribing to a topic with the subscribe method call.

In the Example 4.16 is shown subscribing to a topic with enabling event cache.The parameter `'org.example'` is the name of the topic that is being subscribed to and the parameter `onData` is the name of the callback method that is called every time the widget receives a message with that topic. The two parameters which are `null` are respectively `scope` object and `onComplete` callback method which are not used in this example, and the last parameter `{PageBus:{cache: true}}` is used to enable Tibco PageBus event cache.

If event cache has been enabled on a topic and a widget subscribes to the topic with the `PageBus` parameter, then its `onData` callback method is invoked with each cached message. Caching on a topic is maintained until the last cache enabled subscription is destroyed on that topic [38].

4.2.1.3 Security Restrictions to Third Party Widgets

Widgets communication with outside world is limited with the Same Origin Policy [33] enforced by Web browsers. Widgets in can make connections only to the servers in their subdomain they belong to (e.g. widgets can only communicate with the server they are being hosted from). Widgets in cannot make direct connections to servers outside their domain.

The Same Origin Policy can be too restrictive to those widgets that want to use services outside their domain (e.g. accessing RSS feeds from various news channels). Accessing such services is not harmful for mashups when widgets are held in `IFrames`, because `IFrame` prevents widgets from accessing and manipulating other components of the Web application and messaging is done only through the hub.

One option to bypass the restriction which does not allow a widget to use Web services outside the widget's domain is to create a proxy in the widget's domain which would mediate connections between the widget and outside servers. The widget would then connect to the proxy that is in the same domain with the widget and the proxy would mediate messages between the widget and any server outside the widget's domain. This would allow access to services outside widget's domain through the mediating proxy.

For example, let us say that we have a widget in an `IFrame` that is being hosted from the location `http://stacc.ee/~villido/mashup/widgets/hubTestDataFetcher/index.html`. If the widget would want to fetch data from the BBC News RSS feed in the following location `http://newsrss.bbc.co.uk/rss.xml`, then it would be impossible for the widget to do so directly. The problem is that the BBC News feed is hosted at a different domain (`newsrss.bbc.co.uk`) than the widget (hosted at `stacc.ee`). If the widget would try to fetch data directly from the BBC's server then the browser's same origin policy would restrict such an attempt.

To enable the widget to fetch data from the BBC server we would have to set up a proxy service in the stacc.ee domain. We can do that by setting up a proxy service at the location `http://stacc.ee/~villido/proxy.php`. After we have set up a proxy service, the widget can fetch data from BBC server through the proxy by using following URI: `http://stacc.ee/~villido/proxy.php?url_path=http://newsrss.bbc.co.uk/rss.xml`. The proxy would then fetch data from the BBC's server and would then forward it back to the widget. This approach would allow bypassing the browser's Same Origin Policy to allow widgets to make connections to all required servers regardless of their domains.

Example 4.17 Creating a ManagedHub instance [3].

```
var managedHub = new OpenAjax.hub.ManagedHub(
  {
    onPublish: onMHPublish,
    onSubscribe: onMHSubscribe,
    onUnsubscribe: onMHUnsubscribe,
    onSecurityAlert: onMHSecurityAlert
  }
);
```

4.2.2 Installing OpenAjax Hub

To add widgets' collaboration functionality to a Web page an OpenAjax Hub 2.0 has to be initialized [3]. To initialize a hub, first the `ManagedHub` instance has to be created along with particular security callback methods. Then, widgets' containers can be added to the hub after a `ManagedHub` instance has been created.

4.2.2.1 Creating a ManagedHub Instance

A `ManagedHub` instance provides the Hub API for the manager application (Web portal). An example of a `ManagedHub` creation is shown in the Example 4.17. The `ManagedHub` constructor takes in a parameter of an object with references to various security callback methods. When creating a `ManagedHub` instance four methods have to be created and referred to as parameters. The parameter names that refer to the security callback methods are `onPublish`, `onSubscribe`, `onUnsubscribe`, and `onSecurityAlert`. In the Example 4.17, the names of the corresponding implemented method names are `onMHPublish`, `onMHSubscribe`, `onMHUnsubscribe`, and `onMHSecurityAlert`.

The callback method header referred in the `onPublish` parameter for handling publishing requests is shown in the Example 4.18 on the next page.The `onPublishMethod` is the name of an implemented callback method referred in the `onPublish` parameter when the `ManagedHub` instance was created. The parameter topic is the name of the topic of the message that is being pub-

Example 4.18 Callback method for handling publishing requests [3].

```
function onPublishMethod(topic, data, publishContainer, subscribeContainer)
```

Example 4.19 Callback method for handling subscribing requests [3].

```
function onSubscribeMethodName(topic, container)
```

lished, the parameter data is the message object that is being published, the parameter `publishContainer` refers to the widget's container that published the message, and the `subscribeContainer` refers to the widget's container that has subscribed to the topic and is about to receive the message. The method is called every time a message is being published and should return a `boolean` value of either `true` or `false` depending on whether the current message exchange is allowed to take place. The minimum implementation of this method would simply always return `true`, which would allow all messages to be exchanged regardless of which widgets are involved.

The next callback method header is for handling subscribing requests. It is being referred by the `onSubscribe` parameter in the `ManagedHub` constructor and would look like shown in the Example 4.19. The `onSubscribe-MethodName` is the name of the callback method, whereas the parameter `topic` is a string representation of the topic name that is being subscribed to, and the parameter `container` refers to the widget's container that is subscribing to a topic. This method is for controlling which widgets can subscribe to which topics. If a widget is allowed to receive messages from a particular topic then the method should return a `boolean` value of `true`, otherwise, if a widget is not allowed to receive messages from a topic, then the method should return `false` to restrict that topic to the widget. The minimum implementation of this method would simply return `true` to allow every widget to subscribe to every topic.

Similarly to the previous method, which is called each time a widget subscribes to a topic, the following callback method is called every time a widget is unsubscribing from a topic. The method's header is shown in the Example 4.20 on the next page. The `onUnsubscribeMethodName` is the

Example 4.20 Callback method for handling unsubscribing requests [3].

```
function onUnsubscribeMethodName(topic, container)
```

Example 4.21 Callback method for handling security alerts [3].

```
function onSecurityAlertMethodName(source, alertType)
```

name of the callback method which is being referred from the `ManagedHub` constructor's parameter `onUnsubscribe`. The parameter `topic` is the name of the topic which is being unsubscribed from, and the parameter `container` refers to the widget that is unsubscribing. This method is optional and intended for just only providing information about a widget unsubscribing from a topic.

The last callback method referred from the `ManagedHub` instance constructor is to handle security alerts. The method is referred form the `ManagedHub` constructor's parameter `onSecurityAlert`, and the method's header is shown in the Example 4.21. The `onSecurityAlertMethodName` is the name of the callback method to handle security alerts raised when widgets have been blocked from attacking. The parameter `source` refers to the widget that has been misbehaved and been blocked, and the parameter `alertType` specifies the type of the alert that caused the alert.

4.2.2.2 Adding an IFrame Container

An `IFrame` container is used to handling widgets in secure `IFrame` elements. Widget containers (either `IFrame` or `Inline` containers) can be created after the `ManagedHub` object has been created. The constructor for `IFrame` containers is shown in the Example 4.22 on the next page. Here the parameter `hub` is a reference to the `ManagedHub` instance, the parameter `clientID` is a unique container id that identifies a particular client for the `ManagedHub`, and the parameter `params` is an object with two parameters that allows additional information to be used in instantiating the `IframeContainer`. The parameters in the `object` params consist of two objects: `Container` and `IframeContainer`, both with their additional parameters.

The object `Container` in the parameter params consists of parameters

Example 4.22 The constructor of the IframeContainer [3].

```
OpenAjax.hub.IframeContainer (hub, clientID, params)
```

of methods that are called in various events related to the container. The only mandatory parameter of a `Container` object is the `onSecurityAlert` parameter which has to refer to a method that is called if there is a client-side security alert (for example if a widget is misbehaving). The two optional parameters are `onConnect` which refers to the method that is called when the client connects to the hub, and `onDisconnect` which refers to the method that is called when the client disconnects from the hub.

The object `IframeContainer` in the parameter `params` is an object with following parameters: the parameter `parent` is to specify the widget's parent DOM (Document Object Model) element (where the widget is to be located on the Web page), the parameter `uri` is to specify the widget's location URI (that specifies where the widget is located in the Web), and the parameter `tunnelURI` is to specify the URI of the location of the tunnel which tunnels the widget to the `IFrame` (tunnel URI must origin the same domain as the Web page which instantiates the `IframeContainer`). There are also optional parameters like `iframeAttrs` for specifying the `IFrame` attributes (e.g CSS styles) for the widget. More detailed overview of the optional parameters can be obtained from the OpenAjax Hub 2.0 specification [3] wiki.

In the Example 4.23 on the following page is shown adding of a widget to an `IFrame` container in a Web page. In the example, the first `div2` object is created that is a `div` element and used as a parent element for hosting the `IFrame` container. Then an `IframeContainer` is created with various parameters. The first parameter of the `IframeContainer` is the `managedHub` which have been created earlier as a `ManagedHub` instance. The second parameter, "`client2`", is a unique ID that uniquely identifies the widget client container to the `ManagedHub`.

In the same example, the `Container` object contains three parameters (`onSecurityAlert`, `onConnect`, and `onDisconnect`) that refer to the following implemented callback methods `onClientSecurityAlert` (called out when there is a client-side security breach), `onClientConnect` (called when

Example 4.23 An example of a widget being added to an `IFrame` container [3].

```
var div2 = document.createElement( ”div” );
var container2 = new OpenAjax.hub.IframeContainer(managedHub ,
”client2”,
 {
  Container: {
   onSecurityAlert: onClientSecurityAlert,
   onConnect: onClientConnect,
   onDisconnect: onClientDisconnect
  },
  IframeContainer: {
   // DOM element that is parent of this container:
   parent: div2,
   // Container’s iframe will have these CSS styles:
   iframeAttrs: { style: { border:”black solid 1px” }},
   // Container’s iframe loads the following URL:
   uri: ”http://c0.foo.bar.com/samples/ClientApp2.html”,
   // Tunnel URL required by IframeHubClient:
   tunnelURI: ”http://mashup.foo.bar.com/hub20/tunnel.html”
  }
 }
);
```

client connects to the hub), and `onClientDisconnect` (when client disconnects from the hub).

In the same example above, the `IframeContainer` object has four parameters. The parameter parent specifies that the widget is to be created into the `div2` object (which in this example is a `div` element). The parameter `iframeAttrs` specifies `IFrame` attributes which in this example are CSS style parameters (black solid one pixel wide border). The parameter `uri` specifies that the widget is located at `http://c0.foo.bar.com/samples/ClientApp2.html`, and the parameter tunnelURI specifies that the tunnel is located at `http://mashup.foo.bar.com/hub20/tunnel.html`.

Chapter 5

Implementation of the Transformer Widget

Transformer Widget is essentially a widget that can be plugged in to any application that is using OpenAjax Hub 2.0. It is using TIBCO PageBus which is an extension to OpenAjax Hub to support event cache whenever a Web application wants to cache messages while its widgets are being initialized. Transformer Widget is a separate Web application meant to be held in an `IFrame` of a Web page. It is intended to be invisible and it has no graphical output (besides messages log which is shown only for development purposes). Its application logic is built on JavaScript and it uses TIBCO PageBus JavaScript library to communicate with OpenAjax Hub.

Transformer Widget is written in Java but compiled to JavaScript by using Google Web Toolkit (GWT) [15]. GWT and Java was adopted in order to simplify creation of JavaScript code to cope with different browser standards so it would be optimized to work best with all the browsers. Also, GWT provides a debugger and abstraction for browser's DOM manipulation which increases overall development efficiency.

5.1 Directory and Package Layout

The Transformer Widget project is divided into two main folders: `src` and `war`. The `src` folder contains Java source code of the project which is used to generate JavaScript that would be able to run in any browser. The whole Transformer Widget application logic is meant to run fully in a browser and there is no server-side application logic. The `war` folder contains static Web resources like HTML pages, JavaScript libraries, compiled output of the application logic, mappings and schema files. The contents of the `war` directory can be deployed to any Web server to make Transformer Widget available over the web.

In the table 5.2 on the next page is a summary of the contents of the `war` directory. The `TransformerWidget.html` is the main entry point for the widget and is called when the widget is being loaded to a Web page. It loads the `pagebus.js` file which is TIBCO PageBus JavaScript library that is an extension of OpenAjax Hub 2.0.

The `war` directory also includes the `mappings.xml` file which is an XML file to describe the semantic integration logic of the messages that are being exchanged by the widgets in the main Web application. The `mappings.xml file` should be configured according to the Web application where Transformer Widget is being used.

The `schemas` folder in the `war` directory is used to keep schemas that are used by Transformer Widget to generate messages from aggregated data collected from the messages exchanged by the widgets in the main application. Only JSON schemas are currently supported by the Transformer Widget but if more data formats (like XML, CSV etc) will be supported in the future, then other schema types could be held in the folder as well. The schema files are referred from the `mappings.xml` file and loaded by Transformer Widget when the `mappings.xml` file is being parsed while Transformer Widget is being initialized when the main Web application is being loaded.

The compiled output generated by GWT is kept in the `transformer-widget` folder in the `war` directory. The `TransformerWidget.html` loads the GWT generated bootstrap [13] file `transformerwidget.nocache.js` which

Directory	**File**	**Description**
/war	TransformerWidget.html	Contains static Web resources (including schemas, mappings, JS libraries) and compiled output. Can be deployed to any Web server.
/war	pagebus.js	The main HTML file for loading the TransformerWidget.
/war	mappings.xml	The TIBCO PageBus extended version of OpenAjax Hub 2.0 JavaScript library.
/war		The mappings configuration file for defining the semantic integration rules of the Web application.
/war/schemas		Directory containing the schema files used to generate messages by Transformer Widget via semantic integration. The schema files are referred from the mappings.xml file.

Table 5.2: The main directory of the Transformer Widget

is a JavaScript file used to load the correct version of the application logic compiled to the specific browser. GWT compiles the application logic from Java to JavaScript for each browser into separate files and the bootstrap file imports those files according to the browser used by user. GWT supports following browsers [14]: Firefox 1.0, 1.5, 2.0, 3.0, and 3.5; Internet Explorer 6, 7, and 8; Safari 2, 3, and 4; Chromium and Google Chrome; and Opera 9.0. The precompiled application logic files are located in the same directory as the bootstrap file and are named by MD5 sums generated from the code during compilation. This assures good caching so that application logic would be up to date and old files would not be cached in browsers' memory if the files have been updated, because the name of the file containing application logic would change every time the application logic has been updated.

The table 5.4 on the following page contains the contents of the `transformerwidget` directory. The source code of the Transformer Widget is held in the `src` folder and it contains Java classes which are being used to generate JavaScript that would run in browsers. The main class of Transformer Widget is `ee.stacc.transformer.client.TransformerWidget` and it implements the GWT `EntryPoint` interface. It is compulsory for the main class to implement the GWT's `EntryPoint` interface to be compatible with GWT framework. When the application is being initialized, the `onModuleLoad` method is called from the main class implementing the `EntryPoint` interface.

The module XML file `TransformerWidget.gwt.xml` is located in the same package `ee.stacc.transformer` with the Transformer Widget and contains settings for the GWT compiler specifying the main entry class of the project and the modules to be loaded. The table 5.6 on page 56 contains a summary of the project source code files.

The testing files to run Transformer Widget in a simple Web page to test the functionality are located in the `test` folder in the `war` directory. The `test` folder contains the `test.html` file which is a simple Web application that initializes three widgets and Transformer Widget to exchange messages between those three widgets. The `tunnel.html` file is used to support messaging by the widgets in `IFrame` containers with the rest of the application

Directory	**File**	**Description**
/war/ transformerwidget		Directory for the compiled output generated by GWT. Contains browser-specific application JS files loaded depending on the browser used.
/war/ transformerwidget	transformerwidget. nocache.js	The bootstrap file generated by GWT to load browser specific application logic to Transformer Widget when initialized. This file must be included in the TransformerWidget.html file.
/war/ transformerwidget	<MD5 sum>. cache.html files	HTML files containing JavaScript of the TransformerWidget's application logic which are generated by GWT from Java source code.
/war/ WEB-INF	web.xml	Configures the Transformer Widget for a Web server and specifies the main html file which is the TransformerWidget.html.
/war/ WEB-INF/ classes		Compiled Java classes used by GWT to run an application in development mode without compiling source code into JavaScript.

Table 5.4: The directory of the compiled output.

Directory	**File**	**Description**
/src		Folder containing the source code of the project.
/src/ee/stacc/ transformer	Transformer-Widget.gwt.xml	The module XML file with the project configuration for the GWT compiler. Specifies the main class of the project.
/src/ee/stacc/ transformer/client		Java source code of the Transformer Widget.
/src/ee/stacc/ transformer/client	Transformer-Widget.java	The main class of the project implementing the GWT's EntryPoint interface with onModuleLoad method.
/src/ee/stacc/ transformer/client/ data		Contains Java source code for keeping data elements and generating data packages. Subdirectories of this directory contain implementations for specific data types.
/src/ee/stacc/ transformer/client/ mapping		Contains Java source code for handling mappings logic.

Table 5.6: The directory structure of the Java source code

Directory	File	Description
/war/test		For testing purposes only. Contains files to test Transformer Widget.
/war/test	test.css	Style sheet to set layout in the test.html page.
/war/test	test.html	HTML page for running a simple Web application to test Transformer Widget.
/war/test	tunnel.html	/war/test/testWidgets Directory containing three test widgets: widget_a.html, widget_b.html and widget_c.html. Those three widgets exchange simple test messages to test the aggregation and combination of data elements from different messages and generation of new messages based on the aggregated data.

Table 5.7: Directory of the test files

through the OpenAjax Hub. The `tunnel.html` file should also be used in every Web application that wants to allow messaging between widgets in `IFrame` containers using OpenAjax Hub. Table 5.7 summarizes the contents in the test folder.

5.2 Compiling the Project

The Transformer Widget project is heavily dependent on the Google Web Toolkit (GWT) and the compiler of the GWT which transforms Java source code into JavaScript. Therefore it is necessary to configure the project to use GWT.

To help compiling the Transformer Widget and running the test application, a GWT generated `build.xml` file is used to build project files using Apache Ant [10]. Apache Ant is a command-line tool for building and assembling Java applications.

The `build.xml` file needs to be configured with the correct path to the GWT SDK directory by setting the property `gwt.sdk` with the right location value.

It is possible to run the project in Development Mode [13] which does not compile Java classes to JavaScript but instead Java Virtual Machine is executing the application code which makes it possible to debug code while running the application and see the changes made in the source code without recompiling the application. The Development Mode can be run with the Ant target devmode (run '`ant devmode`') which opens the GWT Development Mode window where the application can be launched. To simply compile the Transformer Widget the Ant target `build` must be called (run '`ant build`') which compiles the whole project into JavaScript that can be deployed to a Web server. It is also possible to make a `war` file that can be deployed to any Web server. To make a `war` file, the Ant target `war` must be called (run '`ant war`') which then compiles the project and compresses the files to the `TransformerWidget.war` file.

To see the compilation results, the project files should be deployed to a Web server and the `test.html` file in the test folder should be opened. It should then display three widgets that can exchange messages between each other with help of the Transformer Widget. To add Transformer Widget to any Web application, the `TransformerWidget.html` file must be loaded as a widget.

It is also possible to integrate the Transformer Widget project to Eclipse or to any other IDE (Integrated Development Environment) [13]. In Eclipse the GWT Plug-in and SDK can be directly installed through the software installation feature of Eclipse. The project should then be configured to use the GWT SDK. More detailed instructions of how to set up GWT in Eclipse can be found from the GWT resources page [13].

5.3 Implementation

The main class of the Transformer Widget application logic is `TransformerWidget`, which implements the Google Web Toolkit's `EntryPoint` interface with its `onModuleLoad` method. The `onModuleLoad` method is called when the widget is being initialized. The `onModuleLoad` method fetches mappings from the `mappings.xml` file and loads them to memory, and then it connects to the OpenAjax hub.

5.3.1 Fetching Mappings

To fetch mapping from the `mappings.xml` file, the `TransformerWidget` object makes an asynchronous RPC call from the private method `fetchMappings` (called from the `onModuleLoad` method) which sends a HTTP request to the server where the `mappings.xml` file is located. The GWT's `RequestBuilder` class is then used to send HTTP requests and the `MappingsRPCResponseHandler` class is used to handle asynchronous responses. When the response is received from the server where the HTTP request was sent, the `onResponseReceived` method is called from the `MappingsRPCResponseHandler` class. The `onResponseReceived` method then calls the `processMappings` method with the XML string of the mappings received from the response. The `processMappings` method then creates an XML document object from the XML string and calls the `loadDataFrames` method from the `DataFrame` class which calls the `loadDataFrames` method from the `MappingsXmlParser` class. The `loadDataFrames` method takes the XML document created from the mappings XML string and parses through the document to create a list of data frames (`DataFrame` objects) based on the received mappings. The class diagram with classes to fetch mappings is shown in the Figure 5.3.1 on the next page.

`DataFrame` objects (also called as data frames) represent messages being exchanged between the widgets in the Web application. Each `DataFrame` object represents one `frame` element in the `mappings.xml` file. The frames describe the information and structure of the messages. Each data frame has a topic which is a unique name identifying the instances of messages being sent with that topic. Messages with the same topic have similar internal structure, data type and deliver data, which conforms to a specified schema. That makes it possible for widgets to interpret similarly all the messages with the same topic. Therefore, each topic and its messages structure is represented by one data frame (`DataFrame` object).

In addition to topic, the `DataFrame` object keeps information about schema

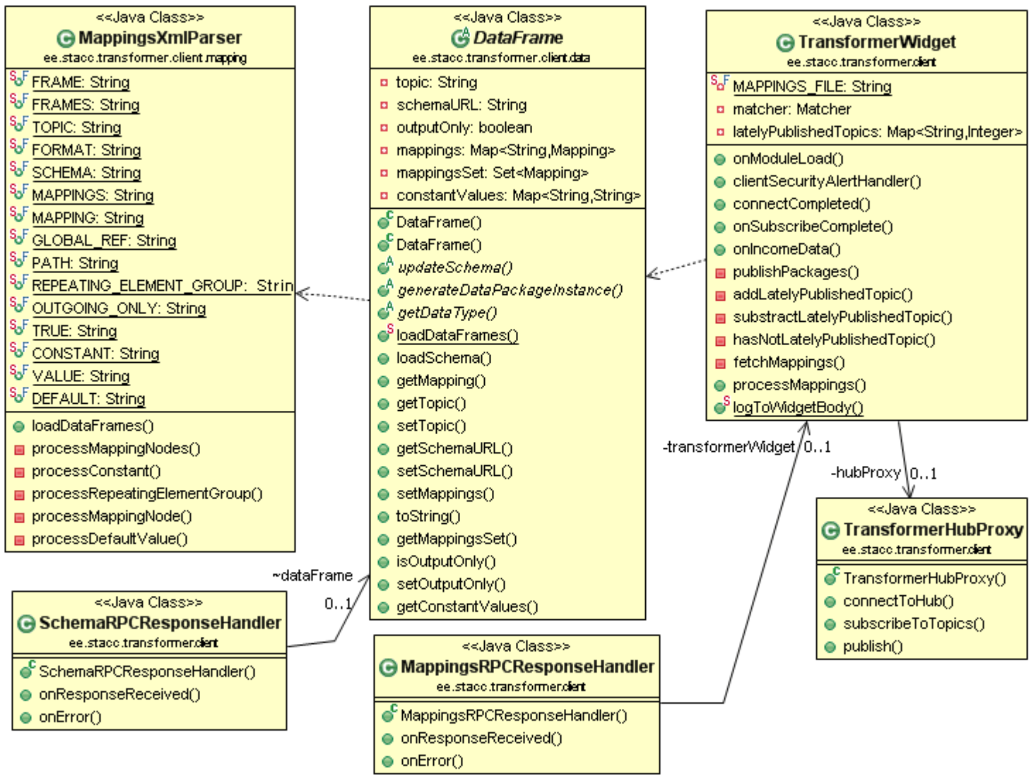


Figure 5.3.1: Loading Mappings.

which is used to generate new messages from aggregated data, mappings which describe the structure and semantics of the messages, and constant values which are used when new messages are being generated. The `DataFrame` object keeps information whether the messages are "outgoing only" which means that those messages are being used only for aggregating data and new messages are not being generated by the Transformer Widget. Some widgets can send messages with topics that no one is subscribed to receive, therefore there is no reason for the Transformer Widget to generate new messages with that topic, but it is still necessary to describe the mappings of those messages so that it would be possible to aggregate data from those messages. If messages are marked as "outgoing only" then it is not necessary to specify schemas for those messages because no messages are being generated with those topics.

The `DataFrame` object is an abstract class that is extended by implementations of different data type specific classes (whether the related messages are in JSON format or in plain string text). The data type specific implementations of the `DataFrame` class contain methods to handle schemas (with the `updateSchema` method) loaded from schema files that can later be used in generating new messages and to generate new data package instances used to store aggregated data to generate new messages. The abstract `DataFrame` class is currently extended by `StringDataFrame` and `JsonDataFrame` classes. The `StringDataFrame` class implements string data type specific data frame functionality and the `JsonDataFrame` class implements JSON data type specific data frame functionality. If support for another data type is to be implemented, then a new class with the new data type specific functionality has to be created to extend the `DataFrame` object. Overview of the data frame classes can be seen from the Figure 5.3.2 on the following page.

After the mappings have been loaded to `DataFrame` objects in the `MappingsXmlParser` class, the schemas of each data frame are fetched using HTTP requests. In each `DataFrame` object the `loadSchema` method is called which uses the schema URL specified in the mappings file to fetch the schema file. If the schema URL is not specified in the mappings file then the schema is not fetched. The GWT's `RequestBuilder` class is used to send HTTP

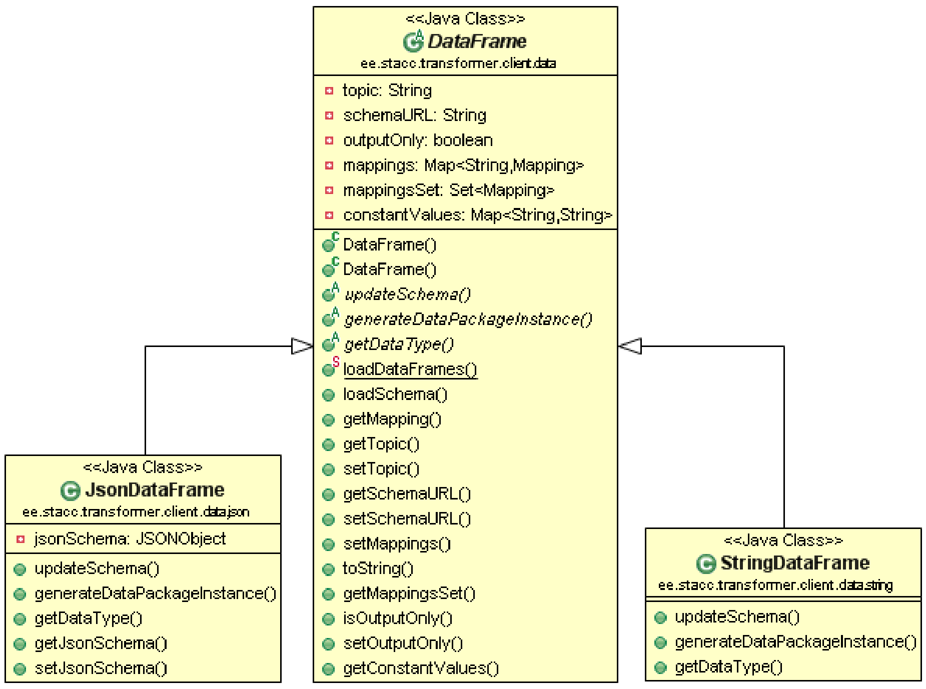
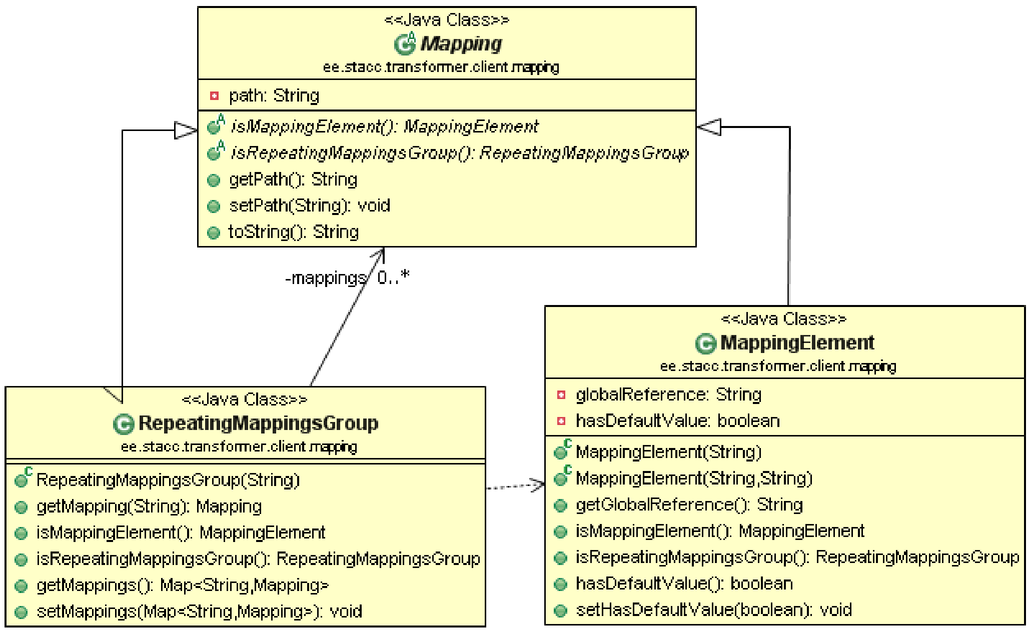


Figure 5.3.2: Data frames.

requests and the `SchemaRPCResponseHandler` class is used to handle the responses of the HTTP requests. The HTTP requests fetch schema files which are text files containing descriptions of the messages' structures.

5.3.2 The Structure of Mappings

`DataFrame` objects keeps a list of `Mapping` objects loaded from the mappings' XML file which describe the structure and semantics of those messages. The `Mapping` class is an abstract class that is extended by classes `MappingElement` and `RepeatingMappingsGroup`. A `MappingElement` object represents one atomic data element in a message represented by a mapping element in the `mappings.xml` file. A `RepeatingMappingsGroup` object represents one group of repeatable data elements (i.e. an array of objects) in a message, represented by a `repeating_element_group element` in the mappings file. The `RepeatingMappingsGroup` contains a list of `Mapping` objects which are the elements in the repeatable data elements group. These `Mapping` objects can



<<Java Class>>
Mapping
ee.stacc.transformer.client.mapping
path: String
isMappingElement(): MappingElement
isRepeatingMappingsGroup(): RepeatingMappingsGroup
getPath(): String
setPath(String): void
toString(): String
-mappings 0..*
<<Java Class>>
RepeatingMappingsGroup
ee.stacc.transformer.client.mapping
RepeatingMappingsGroup(String)
getMapping(String): Mapping
isMappingElement(): MappingElement
isRepeatingMappingsGroup(): RepeatingMappingsGroup
getMappings(): Map<String,Mapping>
setMappings(Map<String,Mapping>): void
<<Java Class>>
MappingElement
ee.stacc.transformer.client.mapping
globalReference: String
hasDefaultValue: boolean
MappingElement(String)
MappingElement(String,String)
getGlobalReference(): String
isMappingElement(): MappingElement
isRepeatingMappingsGroup(): RepeatingMappingsGroup
hasDefaultValue(): boolean
setHasDefaultValue(boolean): void

Figure 5.3.3: Class diagram of the mapping classes

again be either `MappingElement` objects or `RepeatingMappingsGroup`. This makes it possible for the `RepeatingMappingsGroup` to have other `RepeatingMappingsGroup` in the list of `Mapping` objects. In other words, `RepeatingMappingsGroup` is an array for holding either atomic data elements or other arrays. The Figure 5.3.3 shows a class diagram of the mapping classes.

Two main components of mapping elements are path and global reference (with the variable `globalReference`) where path specifies the physical location of an atomic data element in a message and global reference specifies the OWL class which describes the meaning of the atomic data element. This information is enough to be able to aggregate data elements from messages and generate new messages based on aggregated data. With global reference it is possible to identify messages that carry the same kind of data and with path it is possible to locate those data elements from those messages.

The `RepeatingMappingsGroup` class does not contain a global reference value because it represents arrays which contain elements that themselves contain global references. To find out which global references are stored in the elements of a mapping group, an iterator is needed that would iterate through all the elements of a mapping group. Since one mapping group can have elements that are also mapping groups (an array can contain objects that are also arrays), then recursive iteration through all the elements would be computation-intensive process.

To optimize the speed of the Transformer Widget's algorithms, a map data structure (with the `HashMap` object) is used instead of a regular list to allow fast access to the elements stored in the map without iterating through the list of elements every time a new element is needed to be found. In the `RepeatingMappingsGroup` class the `Mapping` objects are held in a map data structure to allow fast look up of elements using a global reference key.

Atomic data elements can have default values so that new messages can be created with default values if no data with corresponding global reference keys have been found, but are still needed for aggregation. This makes it possible to generate new messages even if not all the necessary data has been aggregated. If an atomic data element contains a default value then the `hasDefaultValue` in the `MappingElement` object is set to true and the corresponding default value is stored in the map of constant values in the `DataFrame` object. The constant values are kept in a map data structure and values can be accessed with a path key, so that when a message is being generated, the algorithm can find the corresponding default value by using the path of an atomic data element that is being inserted to the newly generated message body.

5.3.3 Communication with the Hub

Communication with the hub is implemented through the `Transformer-HubProxy` class, which is a proxy class for communicating with the OpenAjax Hub library. The `TransformerHubProxy` class uses GWT's JavaScript Native Interface (JSNI) [16] to integrate third party JavaScript libraries within Java source code by allowing handwritten JavaScript to be mixed with Java code. The `TransformerHubProxy` acts as a wrapper for JavaScript methods that are needed to be called from JavaScript.

During the initialization of the Transformer Widget the `onModuleLoad` method of the TransformerWidget class connects to the hub through the TransformerHubProxy class using its method connectToHub. The connect-ToHub method uses JavaScript Native Interface (JSNI) to create an Iframe-HubClient instance from the OpenAjax hub library where the IframeHub-Client is being used by widgets to communicate with the hub. The Iframe-HubClient instance is then used to connect the Transformer Widget to the hub with the connectCompleted callback method which is called from the TransformerWidget class when the widget has successfully connected to the hub. The TransformerHubProxy class can be seen in the Figure 5.3.1 on page 60.

The connectCompleted method in the TransformerWidget class then calls the subscribeToTopics method from the TransformerHubProxy class to subscribe to all of the topics that are exchanged through the hub. To subscribe to all of the topics, the widget subscribes to the topic "**" which matches the pattern of all the topic names. The subscribeToTopics method is also using a JavaScript Native Interface to use custom JavaScript necessary for integrating with the hub library. The widget subscribes to the topics with the callback method onIncomeData which is called from the Trans-formerWidget class every time a message is being exchanged through the hub.

5.3.4 Handling Incoming Messages

Every time the hub sends a message to the Transformer Widget the on-IncomingData method is invoked from the TransformerWidget class. The onIncomingData method is called with parameters topic and publisher-Data which are the topic and the message object that a widget that sent the message used to pass the message through the hub. A related class diagram can be seen in the Figure 5.3.4 on the next page.

The onIncomeData method then uses method getUpdatedPackages of the Matcher class to aggregate data from the incoming message and generate new messages based on the mappings and aggregated data. The getUpdat-edPackages returns a list of generated messages which are then published

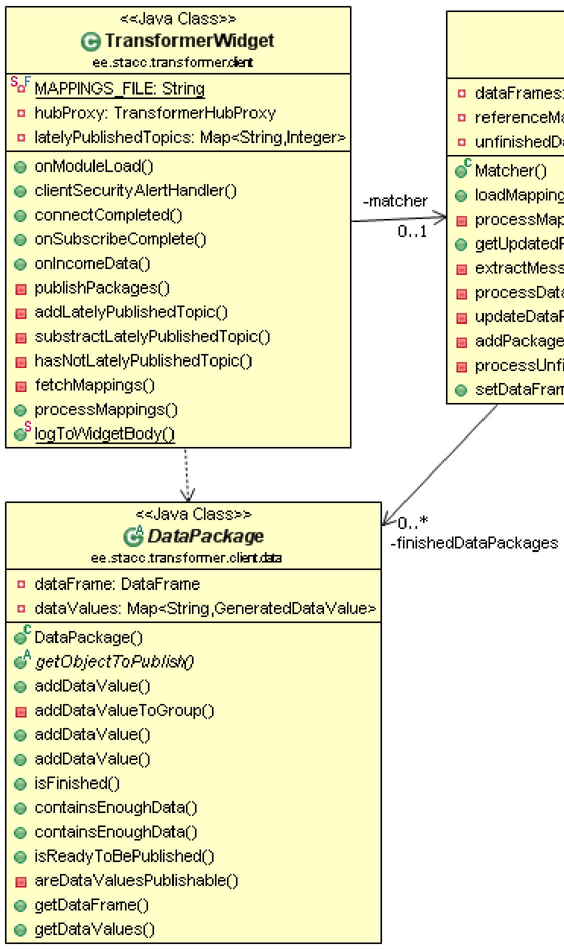

<<Java Class>>
TransformerWidget
ee.stacc.transformer.client
MAPPINGS_FILE: String
hubProxy: TransformerHubProxy
latelyPublishedTopics: Map<String,Integer>
onModuleLoad()
clientSecurityAlertHandler()
connectCompleted()
onSubscribeComplete()
onIncomeData()
publishPackages()
addLatelyPublishedTopic()
substractLatelyPublishedTopic()
hasNotLatelyPublishedTopic()
fetchMappings()
processMappings()
logToWidgetBody()
-matcher
0..1
<<Java Class>>
Matcher
ee.stacc.transformer.client
dataFrames: Map<String,DataFrame>
referenceMappingsToFrames: Map<String,Set<DataFrame>>
unfinishedDataPackages: Map<String,List<DataPackage>>
Matcher()
loadMappingsToMap()
processMappingsForFrame()
getUpdatedPackets()
extractMessageData()
processDataValue()
updateDataPackagesWithNewData()
addPackageToUnfinishedList()
processUnfinishedDataPackages()
setDataFrames()
0..*
-finishedDataPackages
<<Java Class>>
DataPackage
ee.stacc.transformer.client.data
dataFrame: DataFrame
dataValues: Map<String,GeneratedDataValue>
DataPackage()
getObjectToPublish()
addDataValue()
addDataValueToGroup()
addDataValue()
addDataValue()
isFinished()
containsEnoughData()
containsEnoughData()
isReadyToBePublished()
areDataValuesPublishable()
getDataFrame()
getDataValues()

Figure 5.3.4: Handling incoming messages.

through the hub to other widgets connected to the hub that are interested in those messages. The messages returned from the `getUpdatedPackages` method are `DataPackage` objects which contain necessary information about the structure and semantics of the message so that message object can be extracted from them that can be published to the hub. The message objects extracted from the `DataPackage` objects are then published to the hub using the publish method of the `TransformerHubProxy` class. The publish method uses GWT's JavaScript Native Interface to integrate with the hub and can be used to publish any objects with any topic.

The OpenAjax hub then forwards the message to every widget that has subscribed to the topic that the message was published with. Since the Transformer Widget has subscribed to all of the topics then the hub forwards the Transformer Widget all the messages it receives, even the ones that the Transformer Widget itself sends to the hub. This means that the Transformer Widget receives back every message it publishes to the hub. That could cause recursive loop because the Transformer Widget would start aggregating data from the messages it has generated and sent by itself. To avoid those endless recursive loops, the Transformer Widget discards messages that were sent by itself. It keeps the list of topics (in the `latelyPublishedTopics` list) it lately used to send messages to the hub and when the Transformer Widget receives a message, then it checks if the topic of the incoming message is in the list. If the topic is in the list, then it means that the message belongs to the Transformer Widget and can be discarded. After the message has been discarded, then the topic is removed from the list of lately published topics so that the Transformer Widget could process messages sent by other widgets with that topic.

5.3.5 Data Model

Single atomic values (a single number, string etc) in a message are called atomic data elements. A message may contain many atomic data elements (e.g. a person's first name, last name, birth date etc) and they are held as `AtomicDataValue` objects. The `AtomicDataValue` class is an abstract class

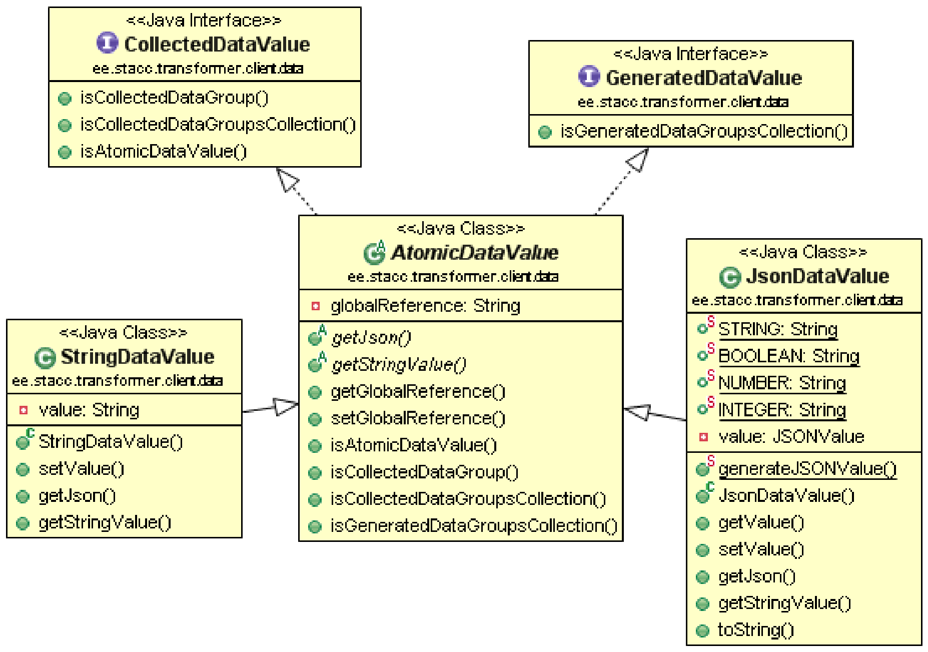
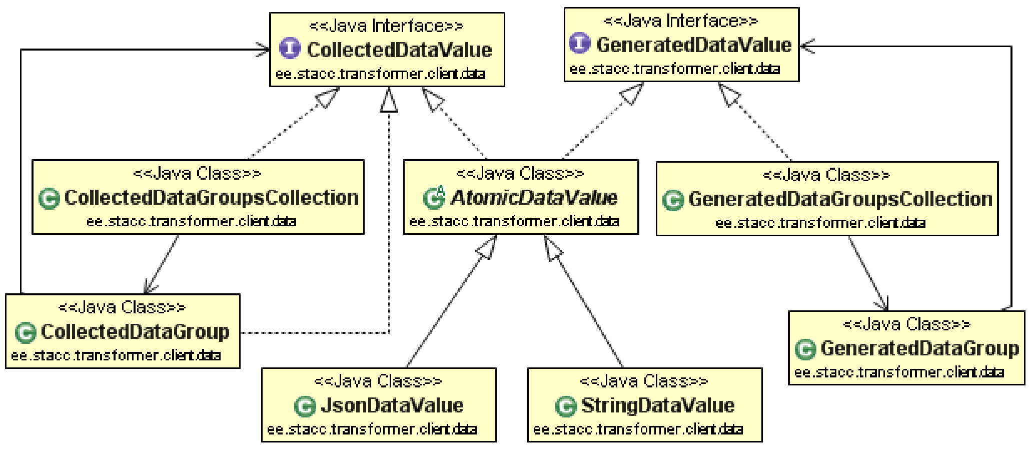


Figure 5.3.5: Atomic Data Values.

which is currently implemented by `JsonDataValue` class for holding atomic data values in JSON format and `StringDataValue` class for holding atomic data values in String data format. If support for additional data formats (XML for example) is to be implemented in the future then a new class that extends the `AtomicDataValue` class has to be created to manage the atomic data values in that new data format. The atomic data value classes can be seen in the Figure 5.3.5.

Sometimes it is necessary to have an array of complex objects in a message (e.g. an array containing personal data of many persons). In case of arrays it is necessary to group data elements that belong together (e.g. we do not want to mix up first and last names of different persons). In the mappings configuration file the data elements are grouped together in a repeating element group (`repeating_element_group element`). The `CollectedDataGroup` is a class for holding data values that are part of the same repeatable element group. It is necessary to group the data values together that are located in the same repeatable element group, because that allows relating to the data



<<Java Interface>>
CollectedDataValue
ee.stacc.transformer.client.data
<<Java Interface>>
GeneratedDataValue
ee.stacc.transformer.client.data
<<Java Class>>
CollectedDataGroupsCollection
ee.stacc.transformer.client.data
<<Java Class>>
AtomicDataValue
ee.stacc.transformer.client.data
<<Java Class>>
GeneratedDataGroupsCollection
ee.stacc.transformer.client.data
<<Java Class>>
CollectedDataGroup
ee.stacc.transformer.client.data
<<Java Class>>
GeneratedDataGroup
ee.stacc.transformer.client.data
<<Java Class>>
JsonDataValue
ee.stacc.transformer.client.data
<<Java Class>>
StringDataValue
ee.stacc.transformer.client.data

Figure 5.3.6: Data Values.

elements that are in the same repeatable element group. For example, if a message contains an array of persons, each containing data about a person's first name and last name, then a separate instance of the `CollectedData-Group` class has to be made for each person for containing the person's first and last name. The `CollectedDataGroup` would store the first name and last name of a person in an array. Overview of the data value classes can be seen in the Figure 5.3.6.

The `CollectedDataGroup` class keeps a collection of data values that are grouped together (e.g. first name and last name) in a map data structure (`HashMap` object) and the values are stored according to their global references as keys for faster retrieval of those values. The `CollectedDataGroup` objects are stored in the `CollectedDataGroupsCollection` class that keeps a collection of data element groups. For example, if the `CollectedData-Group` class is for keeping first names and last names of persons then the `CollectedDataGroupsCollection` class is for keeping all the persons in the array. Overview of the data value groups can be neen in the Figure 5.3.7.

It is necessary to group together certain data elements that are aggregated from a message (e.g. collected data value groups), but it is also necessary to group together data elements when creating new messages. A separate grouping of data elements is necessary based on repeating data element groups of messages that are to be generated from aggregated data. The `Generated-`

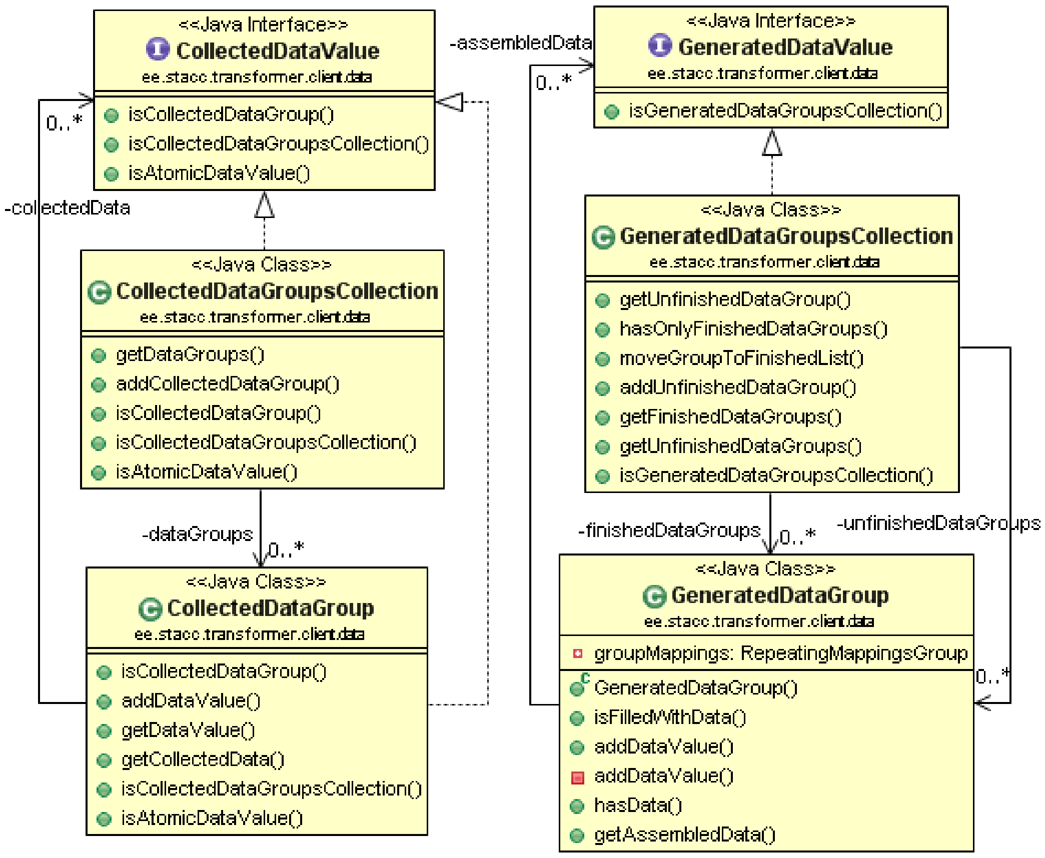


Figure 5.3.7: Data Value Groups.

`DataGroup` class groups data values together (e.g. a person's first name and last name) that are grouped together in the mappings configuration file to form a new message. The data values are stored in a map data structure (`HashMap` object) for fast retrieval of data values. The `GeneratedDataGroup` objects are stored in the `GeneratedDataGroupsCollection` class that stores groups of data values necessary to form new messages. For example if the `GeneratedDataGroup` class is for storing first name and last name of a person, then the array of person objects are stored in the `GeneratedDataGroupsCollection` class.

The collected data values (e.g. `CollectedDataGroup` objects) and generated data values (e.g. `GeneratedDataGroup` objects) look similar but perform different tasks. Data values that are aggregated (or collected) from a message are called collected data values and data values that are used to generate (or to assemble together) new messages are called generated data values. There is no difference between collected data values and generated data values in the level of atomic data elements where a single atomic data element (a number, string or something else) can be used in forming a new message after it has been aggregated from a received message without any further structural manipulations. The difference between collected data values and generated data values is in grouping of data values where generated data values can be grouped together from data values aggregated from different messages.

The `CollectedDataValue` is an interface for all the classes that deal with data values that are collected (or aggregated) from messages and `GeneratedDataValue` is an interface for classes that deal with data values that are used to generate new messages. The abstract `AtomicDataValue` class implements both the `CollectedDataValue` and the `GeneratedDataValue` interfaces because atomic data values do not need any restructuring after they have been aggregated from a received message to generate a new message. Since the abstract `AtomicDataValue` class implements both interfaces (the `CollectedDataValue` and the `GeneratedDataValue` interfaces), then the `JsonDataValue` and `StringDataValue` classes that extend the `AtomicDataValue` class can be used either as collected data values or as generated data values. The `CollectedDataValue` interface is implemented also by the

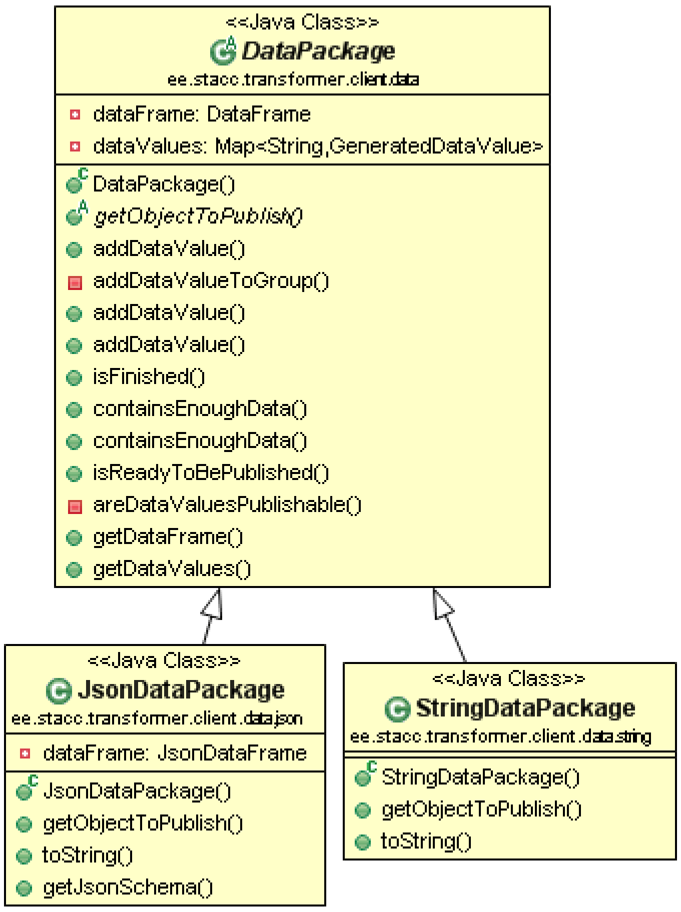


Figure 5.3.8: Data Packages.

`CollectedDataGroup` class and by the `CollectedDataGroupsCollection` class (in addition to the `AtomicDataValue` class) but those classes do not implement the `GeneratedDataValue` interface and cannot be used as generated data values. The `GeneratedDataValue` interface is implemented by the `GeneratedDataGroupsCollection` class (in addition to the `AtomicDataValue` class) and the `GeneratedDataGroupsCollection` class can be used only as generated data value.

Data values are held in data packages (`DataPackage` objects) that represent messages that are being generated from aggregated data. Each data package relates to a corresponding data frames (`DataFrame` objects) that define the structure of a message. Data values are collected to the data packages according to the mappings of the messages. When a new message is received, then data elements are collected from the message and added to the data packages that can use those data elements in generating new messages. When enough data has been collected for a data package, then a new message can be generated from the data in the data package based on the schema and mappings in the corresponding data frame. When a new message is generated from a data package, then the data package is discarded and erased from the memory. Overview of the data package classes can be seen in the Figure 5.3.8 on the preceding page.

A new data package is created when the Transformer Widget receives a message that contains data elements that can be used in generating a message that does not have a corresponding data package. In this case, a new package is created and the data element is added to the data package. One data frame can have many data packages meaning that if messages are generated from aggregated data collected from different sources and one source produces more data than other source then data packages are stacked and filled partially with data from the more productive source. When the Transformer Widget receives a message from a less productive source then the data elements from the message are sent to the older data packages in the stack.

The `DataPackage` class is an abstract class extended by the `JsonDataPackage` class for generating messages in JSON format and the `StringDataPackage` class for generating messages in String format. Most of the logic regarding adding new data values and checking if a data package contains enough data for a message to be generated is situated in the abstract `DataPackage` class. The extending classes implement the final message generation logic where the collected data is put together using mappings and schema to generate the final message object that can be published to other widgets.

To generate a message object from the data in a data package, the abstract method `getObjectToPublish` must be implemented by a class that extends the abstract `DataPackage` class. If support for additional data formats (XML for example) is to be implemented in the future then a new class that extends the abstract `DataPackage` class must be created.

The `addDataValue` method is used to add any collected data value to the data package. If an atomic data value is added, then the data value is added directly as a generated data value because the `AtomicDataValue` class implements both `CollectedDataValue` and `GeneratedDataValue` interfaces and atomic data values do not need any structural rearrangements as data value groups need. If a collected data group (`CollectedDataGroup`) object is added, then all the data values that can be used in forming the message are taken from the collected data group object and added to the data package. If the mappings corresponding to a data package contain a repeatable element group (i.e. an array), then the data values are added according to the mappings of the repeating element groups as generated data values (`GeneratedDataValue` objects) which are held in a generated data groups collection (a `GeneratedDataGroupsCollection` object). This assures that the data values that are added are grouped together according to the mappings configuration and individual values are not mixed up during the adding of elements.

The `isReadyToBePublished` method is used to check if a data package has collected all the necessary data values so that it would be possible to generate a new message to be published to the hub. The method iterates through every mapping of the data package and checks if corresponding data values are stored in the data package. In case of repeating element groups, the method checks integrity of every generated data group (`GeneratedDataGroup`) object related with the repeating element group. Every generated data group object must contain all the necessary data values defined in the repeating element group mapping before the data package is ready to be published to the hub. If there is at least one group that does not contain all the necessary data values, then the data package is not yet ready to be published.

5.3.6 Data Aggregation

The main data aggregation and messages generation is done in the `Matcher` class where most of the messages transformations' logic is located. The `Matcher` class has the collection of all the data frames (`DataFrame` objects) that represent the structures of the messages that are defined in the mappings XML file. The data frames are stored in a map data structure (in a variable called `dataFrames` that is a `HashMap` object) to avoid iterating through all the data frames and being faster when looking up data frames by their topic name.

The data frames are also kept in another map data structure (in a variable called `referenceMappingsToFrames` which is a `HashMap` object) by the global references to allow fast look up of data frames that contain mappings with particular global reference. This allows finding of all the data frames that contain data values with references to the same OWL class or attribute. The two map objects with data frames allow finding of data frames either by their topic names or by their global references. The reason for keeping two map objects with redundant data is solely for better optimization for speed of the data aggregation algorithm by keeping the necessary iterations over collections minimal.

The method that loads the data frames to a map according to the global references associated with the data frames is called `loadMappingsToMap`. The method goes through every mapping in each data frame and adds data frames to the map to the global references.

The `getUpdatedPackets` method in the `Matcher` class is the main method, which is called from the `TransformerWidget` class every time a new message is received from the hub, to aggregate data from the message sent by another widget. The method aggregates data from the received message and generates new messages based on the data it has aggregated. The messages that are being generated are held as data packages (`DataPackage` objects) which refer to the corresponding data frames (`DataFrame` objects) that define the structure of the messages. If not enough data has been aggregated for a data package to generate new message, then the data package is kept in a list of unfinished data packages until the necessary data elements have been collected from aggregating further messages. When enough data has been collected for a data package and it is considered to be finished, then a new message can be generated from the data in the data package based on the schema and mappings in the corresponding data frame. The finished data package is then moved to the list of finished data packages which is used later to generate and publish new messages based on those finished data packages after the received message has been processed.

The `getUpdatedPackets` method's parameters are topic and data where the topic is the name of the topic of the received message and the data is the received message object. The `getUpdatedPackets` method first locates the data frame (data type specific implementation of an abstract `DataFrame` object) which represents the structure and semantics of the received message. If the corresponding data frame was found, then the message object is processed in the `extractMessageData` method using the mappings from the data frame. If the data frame was not found then the mappings for the messages with that topic were not defined in the mappings XML configuration file and the message object is not processed any further.

The `extractMessageData` method which is called from the `getUpdatedPackets` method processes the received message object according to the mappings from the data frame that correspond to the message to aggregate all the data elements from the message object. The method iterates through every mapping and extracts the data values in the message based on the mappings. Every data value is extracted in the `getDataValue` method according to the location specified in the path of a mapping and is processed in the `processDataValue` method.

The `getDataValue` method in the `InstanceFactory` class called from the `extractMessageData` method extracts the data value from the raw data object according to the mapping that describes the location and the meaning of the data value in a message. If the mapping is a `RepeatingMappingsGroup` object, meaning that the data value is an array of collected data groups (in case of a repeating element group mapping), then the whole array of collected data groups that contain data values are extracted from the message to keep the data values in repeating element group grouped together (e.g. so that persons first and last names would not get mixed up). The array of collected data groups extracted from the message is held in the `CollectedDataGroupsCollection` object that represents a single data value containing groups of data values (i.e. a data value containing an array of `CollectedDataGroup` objects that each contain `AtomicDataValue` objects with the data values stored in the repeating element group). The `CollectedDataGroupsCollection` implements the `CollectedDataValue` interface and can be handled as a regular collected data value object. If the data value is atomic, then it is extracted as a single atomic data value as an `AtomicDataValue` object.

If support for another data format is to be implemented, then the `getDataValue` method must be updated for it to be able to extract data values from messages in the new data format. The method must be able to extract a data value according to a path from a mappings configuration.

The `processDataValue` method, which processes data values, checks if the collected data value extracted from the message is an atomic data value or is it an array of collected data groups (a `CollectedDataGroupsCollection` object). If the extracted data value is an atomic data value, then it is passed directly to the `updateDataPackagesWithNewData` method which updates data packages with the new data value. If the extracted data value is an array of collected data groups, then the method iterates through every collected data group and passes those groups to the `updateDataPackagesWithNewData` method.

The `updateDataPackagesWithNewData` method updates all the data packages with an extracted data value (either atomic data value or data value group). First, the method finds all the data frames that contain mappings with the same global reference as the global reference of the data value. If the data value is a group of data values, then all the global references of each data value in the group are taken into account when the data frames with the same global references are being looked up. The method then iterates through those data frames and updates the related data packages with the data value.

The data packages that are not finished and do not contain enough data to be able to generate new messages are kept in the list of unfinished data packages (`unfinishedDataPackages`). If the list of unfinished data packages does not contain a data package related to a data frame, then a new data package is created and added to the list of unfinished data packages for further updates with data values.

Both an atomic data value (`AtomicDataValue` object) and a group of data values (`CollectedDataGroup` object) can be added to a data package that is not finished. When a group of data values is added to a data package, then those data values from the group are picked that have the same global reference than those that are defined in the mappings of the related data frame. This assures that the data values that are grouped together (e.g. first and last name of a person) would not get mixed up with other data values in an array of repeating element groups when added to a data package.

If a data package has collected enough data values necessary to form a new message, then it is moved from the list of unfinished data packages to the list of finished data packages (`finishedDataPackages`). Data packages that generate messages containing arrays are not moved to the list of finished data packages, because infinite number of elements can be added to an array. After all the data values extracted from a message have been processed and added to the corresponding data packages, then the list of unfinished data packages are iterated through and data packages containing arrays are checked if they contain enough data to generate new messages. Those data packages that are ready to generate messages to be published through the hub to other widgets are then also moved to the list of finished data packages. The list of finished data packages are then returned back to the `TransformerWidget` class which generates publishable messages (usually JSON objects) from the data packages and publishes those messages to the hub using topics from related data frames.

5.4 Adding Support for a New Data Format

It is possible to extend the Transformer Widget to add support for a new data format so that it would be possible to aggregate data from messages and form new messages in that data format. To add support for a new data format extensions to three abstract classes (`AtomicDataValue`, `DataFrame` and `DataPackage`) have to be implemented with the data format specific functionality. Additionally, the `InstanceFactory` class has to be extended to support the creation of the newly implemented classes.

The abstract `AtomicDataValue` class has to be extended with the functionality for handling atomic data values in the new data format. There are currently two abstract methods needed to be implemented: `getStringValue` and `getJson`. The `getStringValue` method should return atomic data values of the new data format as String objects (i.e. in plain text) so it would be possible to transform the atomic data values of the new data format into string data format. Similarly, the `getJson` method should return the atomic values of the new data format in JSON objects so it would be possible to transform the atomic data values of the new data format into JSON data format. That makes it possible to transform messages in the new data format to another data format.

Additionally, a new abstract method should be added to the abstract `AtomicDataValue` class to transform atomic data values in every data format into the new data format (similarly to the `getStringValue` and `getJson` methods). The new abstract method should return object instances of the newly implemented class. The new abstract method should then be implemented by every other atomic data value class that extends the abstract `AtomicDataValue` class. That makes it possible to transform atomic data values to the new data format and compose new messages in the new data format.

The `DataPackage` class has to be extended with the functionality specific to the new data format and the abstract method `getObjectToPublish` has to be implemented. The method `getObjectToPublish` must return a message object in the new data format that can be published through the hub to other widgets.

The message object returned by the `getObjectToPublish` method must be transformable into JavaScript by Google Web Toolkit (GWT) compiler, because the Java source code of the Transformer Widget is compiled into JavaScript and used in Web browsers. GWT is capable of transforming following Java objects into JavaScript [16]: `String`, `boolean`, any numeric value (`int`, `double`, `float` etc) except for `long` numeric type which is not allowed, and `JavaScriptObject` which is GWT's class for handling native JavaScript objects (e.g. JSON objects which can be generated from the

GWT's `JSONObject` class).

The abstract `DataFrame` class has to be extended also with the functionality specific to the new data format and three abstract methods `generateDataPackageInstance`, `getDataType` and `updateSchema` have to be implemented. The `generateDataPackageInstance` method has to return an instance of the `DataPackage` class that was extended with the functionality specific to the new data format and the `getDataType` method has to return the name of the new data type. The `updateSchema` method has to handle the schema content that the Transformer Widget loads from the schema files specified in the mappings configuration file. If the new data format does not contain schemas, then the updateSchema method can be left empty. The schemas can be used in generating new messages but they are not required if the new data format does not use schemas.

After all three abstract classes have been extended with the new data format specific implementation, the two methods in the `InstanceFactory` class have to be extended. The two methods are `getDataFrame` and `getDataValue`.

The `getDataFrame` method should return an instance of the extended `DataFrame` class that implements the functionality specific to the data format specified as a parameter (called `dataType`) of the method. If the parameter specifying the data format equals with the name of the new data format, then an instance of the class that extends the `DataFrame` and implements the functionality specific to the new data format must be returned.

The `getDataValue` method should return a collected data value instance extracted from a message object (parameter called `data`) according to a data format (parameter called `dataType`) and a mapping object (parameter `mapping` that specifies the OWL class and the path with the location of the data value in a message). To add support to a new data format, the method must be extended to return a collected data value (`CollectedDataValue`) object from a message in the new data format based on the mapping given as a parameter. If the data value referred in the mapping is an atomic data value, then the atomic data value should be returned as an instance of the class that extends the abstract `AtomicDataValue` class and implements the functionality of the new data format. If the data value referred in the mapping is not an atomic data value, but is an array of repeating element groups, then the whole array must be returned as a collected data value, because it is necessary to keep related data values grouped together if they are in a repeating element group. In case of repeating data element group, a `CollectedDataGroupsCollection` object must be returned that contains an array of groups (`CollectedDataGroup` objects) of data elements (`CollectedDataValue` objects) to store the grouping of data values.

Chapter 6

Validation of the Transformer Widget

In order to validate if the Transformer Widget would be usable in Web applications to provide collaboration between widgets that would not be able to communicate with each other directly, a test Web application called Portal[1] with three test widgets was created.

The test Web application called Portal shown in the Figure 6.0.1 is a simple Web page containing OpenAjax Hub and three widgets that are connected to the Hub. Those three widgets can all send and receive various messages, but they use different topics and data formats which make it impossible for

[1]Test Web application for validating the Transformer Widget is located at `http://stacc.ee/~villido/Portal/Portal.html`

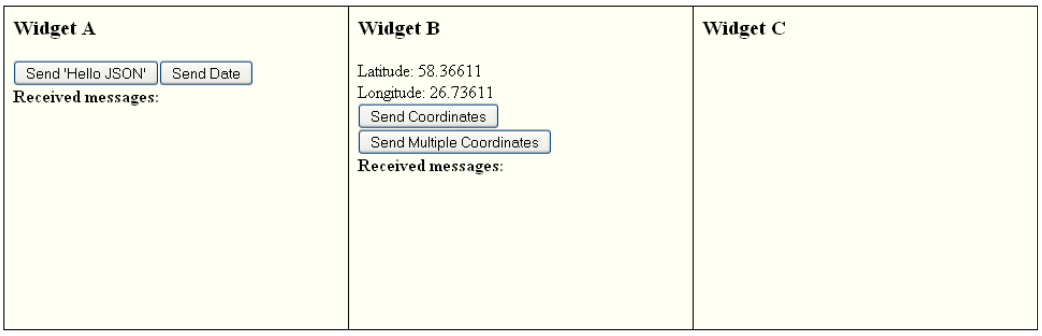


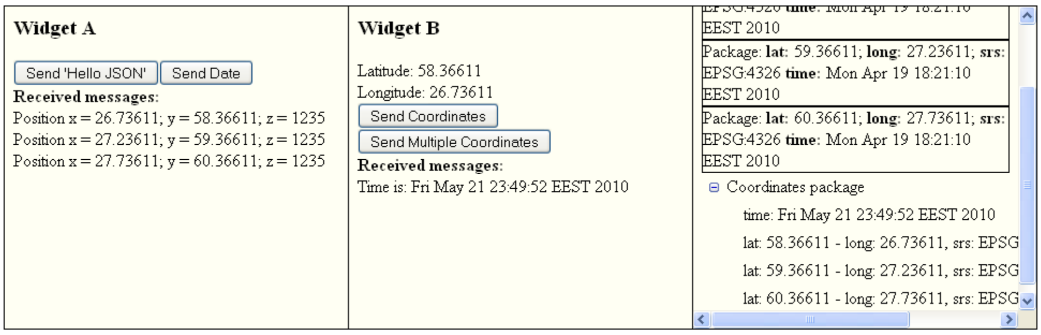
Figure 6.0.1: The Web application for testing the Transformer Widget.

them to communicate directly with each other.

That is a perfect scenario for using the Transformer Widget for aggregating data from the messages sent by those widgets and generating new messages based on the aggregated data that the widgets would be able to receive and interpret. To validate the usability of the Transformer Widget in such scenarios, we would have to create the necessary mappings and schemas that correspond to the messages that the widgets exchange. If the widgets can receive data exchanged between the widgets, then it means that the Transformer Widget works as intended and can be used in other Web applications that require aggregation and transformation of messages the widgets exchange.

In the Portal, four widgets are loaded: Widget A, Widget B, Widget C and Transformer Widget. Widget A is capable of sending messages containing timestamps of current time and capable of receiving messages containing x, y and z coordinates. Widget B is capable of receiving messages with timestamps and capable sending messages containing coordinates. Widget C is capable of receiving messages which contain both coordinates and timestamps in the same message. None of those widgets use the same topics and data structures so they are not able to communicate with each other directly without Transformer Widget. The only message that can be exchanged directly for testing purposes is the “Hello JSON” message that can be sent from Widget A directly to Widget B. The “Hello JSON” message is to test whether the OpenAjax Hub works in case the Transformer Widget has failed to load and transformations do not work. The Figure 6.0.2 on the following page illustrates the messages exchanged between the widgets with the help of the Transformer Widget in the test application.

Widget A has a button “Send Date” which sends a message with a timestamp of the current time when clicked. The timestamp is converted to string and sent with the topic `ee.stacc.date` as plain text. This message is to test whether the Transformer Widget is capable of interpreting messages in a string format. To make the message interpretable by the Transformer Widget, the mappings describing the message must be added to the mappings configuration file. The mappings that describe the message are shown in the


Widget A

Send 'Hello JSON' | Send Date

Received messages:
Position x = 26.73611; y = 58.36611; z = 1235
Position x = 27.23611; y = 59.36611; z = 1235
Position x = 27.73611; y = 60.36611; z = 1235

Widget B

Latitude: 58.36611
Longitude: 26.73611

Send Coordinates
Send Multiple Coordinates

Received messages:
Time is: Fri May 21 23:49:52 EEST 2010

EEST 2010
Package: **lat:** 59.36611; **long:** 27.23611; **srs:** EPSG:4326 **time:** Mon Apr 19 18:21:10 EEST 2010
Package: **lat:** 60.36611; **long:** 27.73611; **srs:** EPSG:4326 **time:** Mon Apr 19 18:21:10 EEST 2010
⊟ Coordinates package
time: Fri May 21 23:49:52 EEST 2010
lat: 58.36611 - long: 26.73611, srs: EPSG
lat: 59.36611 - long: 27.23611, srs: EPSG
lat: 60.36611 - long: 27.73611, srs: EPSG

Figure 6.0.2: Data exchanged between the widgets in the test application.

Example 6.1 Mappings corresponding to the timestamp messages (with topic `ee.stacc.date`).

```
<frame>
  <topic outgoing_only="true">ee.stacc.date</topic>
  <format>string</format>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/lang/owl#Date</global_ref>
    </mapping>
  </mappings>
</frame>
```

Example 6.1.

In those mappings, the `frame` element notates the messages that are sent with topic `ee.stacc.date`. The topic of the messages is specified in the `topic` element with the parameter outgoing_only which is set as `true`. The `outgoing_only`, if set as `true`, notates that those messages are not being received by any widgets and are only being published by a widget. The only reason the message is defined in the mappings configuration file is to enable the Transformer Widget to interpret those messages. The format element in those mappings is set with a value string which means that those messages are all plain text (string) messages. This message has only one mapping element with the global reference (`global_ref`) of `http://www.example.org/lang/owl#Date` which indicates that the atomic data values in the messages encode information about date (timestamp in

Example 6.2 Mappings of the messages containing coordinates.

```
<frame>
  <topic>ee.stacc.location</topic>
  <format>json</format>
  <schema>schemas/location.js</schema>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Latitude</global_ref>
      <path>/position/y</path>
    </mapping>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Longitude</global_ref>
      <path>/position/x</path>
    </mapping>
  </mappings>
</frame>
```

this case). The global reference refers to an example OWL class that does not exist but the reference to the OWL can be used to distinguish different meanings in atomic data values. Since the message format is string, then only one mapping element is allowed and the messages can contain only one atomic data value at the time. In the example, the atomic data value is a timestamp.

In addition to sending timestamp messages, the Widget A is capable of receiving messages with x, y and z coordinates in the JSON format with the topic `ee.stacc.location` as in the following example: `{"position":{"x":26.73611, "y":58.36611, "z":1235}}`. If the Widget A has received a similar message, then it outputs the x, y and z parameters (i.e. coordinates) from the message. To make those messages transformable by the Transformer Widget, then the mappings must be described that specify the content of those messages. The mappings configuration that describes those messages is shown in the Example 6.2.

The `topic` element is set to specify that the topic of those messages is `ee.stacc.location` and the `format` element specifies that the data format is JSON. The schema element specifies the location of the schema file in the local server so that the Transformer Widget could fetch the schema file from

Example 6.3 A message containing geographical coordinates.

```
{"type":"object",
  "properties":{
    "position":{
      "type":"object",
      "properties":{
        "x":{"type":"number"},
        "y":{"type":"number"},
        "z":{"type":"number", "default":1235},
      }
    }
  }
}
```

the server. The schema file is used by the Transformer Widget to generate new messages according to the specified mappings.

The configuration contains two mappings which specify the coordinates in the messages. The first mapping is a y coordinate with the global reference of `http://www.example.org/geoinfo/owl#Latitude` and path of `/position/y`. The path indicates the location of the atomic data element in the message. In the example, the atomic data value of a y coordinate is located in the `y` element which is in the position element of the JSON object. The second mapping is an x coordinate with the global reference of `http://www.example.org/geoinfo/owl#Longitude` and path of `/position/x`. To test the support for default values in JSON schemas, the z coordinate was not specified in the mappings, but only in the JSON schema with a default value that would always be used when new messages are generated, because no other widget provides data values with z coordinate.

After the mappings have been defined for those messages, then a JSON schema must be specified which describes the structure of the messages indicated in the mappings that the Transformer Widget must generate. A JSON schema that corresponds to the messages that the widget A can receive is shown in the Example 6.3:

The above JSON schema corresponds to messages that contain x, y and z coordinates which the widget A can interpret. The z coordinate has the

Example 6.4 Mappings corresponding to a message receiving timestamps.

```
<frame>
  <topic>ee.stacc.time</topic>
  <format>string</format>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/lang/owl#Date</global_ref>
    </mapping>
  </mappings>
</frame>
```

default value of 1235. The schema is located in the file called `location.js` which is located in the schemas directory as indicated in the mappings configuration. This schema along with the mappings above is enough to configure messages from the Widget A "understandable" to the Transformer Widget.

The Widget B is capable of receiving messages with timestamps and is capable of sending geographical coordinates. The topic the Widget B is subscribed to receive timestamp messages is `ee.stacc.time`. Note that the Widget A publishes timestamp messages with the topic `ee.stacc.date` which differs from the one that the Widget B is subscribed. The Transformer Widget can forward messages from the Widget A to Widget B. The mappings configuration for the timestamp messages in the Widget B is similar to the one in the Widget A and is shown in the Example 6.4.

In the mappings configuration shown in the Example 6.4., it is specified that the topic of the messages is `ee.stacc.time`, the data format is string, and the global reference referring to the OWL class indicating the characteristics of the data value in the messages is `http://www.example.org/lang/owl#Date`. Note that the global reference of a timestamp is the same with the global reference of the mappings of the timestamp messages in the Widget A. This makes it possible for the Transformer Widget to collect atomic data values and put them into the messages where needed. The global reference is like an ID which allows binding of data. Since the messages in the example are in string data format, then schema definition is not necessary.

Example 6.5 Message for sending coordinates.

```
{"location":{
  "coordinates":{
    "latitude":58.36611,
    "longitude":26.73611
  }
}}
```

If those mappings are specified in the mappings configuration file of the Transformer Widget, then the Transformer Widget is now capable of forwarding timestamp messages from the Widget A to Widget B. This can be tested by clicking on the "Send Date" button on the Widget A and If the Widget B output something like "Time is: Tue May 18 14:52:44 GMT+300 2010", then the Widget B has received a message from the Widget A. This means that the Transformer Widget has done the transformations correctly.

The Widget B has also two buttons for sending coordinates. The "Send Coordinates" button sends one pair of coordinates: latitude and longitude, and the "Send Multiple Coordinates" button sends three pairs of coordinates (also latitude and longitude) in an array which is useful for validating transformations of repeatable element groups (like an array of coordinates).

The message in a JSON format that the Widget B publishes with the topic `ee.stacc.coordinates` when the button "Send Coordinates" is clicked is shown in the Example 6.5. That message encodes one pair of coordinates: latitude of 58.36611 and longitude of 26.73611. To make this message interpretable by the Transformer Widget, a following mappings configuration must be defined as shown in the Example 6.6 on the next page.

In the mappings shown in the Example 6.6 on the following page, the topic of the messages is marked as `ee.stacc.coordinates` and the `true` value of the `outgoing_only` parameter in the `topic` element indicates that there are not any widget that are able to receive those particular messages and therefore those messages are not needed to be generated. The data format is JSON and because the `outgoing_only` parameter is `true`, then schema is not needed to be defined. It contains two `mapping` elements, one for latitude and other for longitude atomic data elements in the message. The global reference

Example 6.6 Mappings configuration corresponding to the message shown in the Example 6.5.

```
<frame>
  <topic outgoing_only="true">ee.stacc.coordinates</topic>
  <format>json</format>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Latitude</global_ref>
      <path>/location/coordinates/latitude</path>
    </mapping>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Longitude</global_ref>
      <path>/location/coordinates/longitude</path>
    </mapping>
  </mappings>
</frame>
```

of the first mapping element is `http://www.example.org/geoinfo/owl#Latitude` and path is `/location/coordinates/latitude` which indicate the location of the atomic data element of a latitude value in the message. The global reference of the second mapping element is `http://www.example.org/geoinfo/owl#Longitude` and the path is `/location/coordinates/longitude` which indicate the location of the atomic data element of a longitude value in the message.

Note that the global references of the latitude and longitude atomic data elements match with the global references of the latitude and longitude atomic data elements in the x, y and z coordinates message that the Widget A is able to receive. That makes it possible for the Transformer Widget to transform the message from the Widget B to interpretable to the Widget A. If those mappings (shown in the Example 6.6) are added to the mappings configuration file, then the Transformer Widget is capable of transforming those messages. It can be tested by clicking on the "Send Coordinates" button on the Widget B which publishes a message containing coordinates and if the Widget A receives a message containing x, y and z coordinates and outputs something like "Position x = 26.73611; y = 58.36611; z = 1235" under

Example 6.7 Message containing an array of three pair of coordinates.

```
{"location":{
  "placename":"Railway Station",
  "coordinates":[
    {"latitude":58.36611, "longitude":26.73611},
    {"latitude":59.36611, "longitude":27.23611},
    {"latitude":60.36611, "longitude":27.73611}
]}}
```

the received messages section, then it means that the Transformer Widget has transformed the message from the Widget B interpretable to Widget A, and the Transformer Widget can be used in transforming messages in similar scenarios.

A message sent when clicking on the "Send Multiple Coordinates" button on the Widget C is similar to the message when clicking on the "Send Coordinates" button, but the first one sends out an array of three pair of coordinates with the topic `ee.stacc.coordinates.list` as shown in the Example 6.7.

This message is sent to test the usability of repeating element groups in the mappings and arrays in messages. The JSON object above consists of two elements within the location element, one called `placename` and other `coordinates`. The `placename` element is just for noise in this example and is not used by any other widgets. The `coordinates` element is an array which contains coordinates (latitude and longitude). To make this message interpretable to the Transformer Widget, we would have to add following mappings to the mappings configuration file that are shown in the Example 6.8 on the next page.

In the mappings configuration shown in the Example 6.8 on the following page, the topic is set as `ee.stacc.coordinates.list` with the `outgoing_only` parameter as `true` which means that there are not any widgets that could receive messages with that topic and therefore these messages are not needed to be generated by the Transformer Widget and no schemas are therefore necessary. It contains three `mapping` elements and two of them are in the `repeating_element_group` element which means that those two mappings indicate that those atomic data values repeat together in an array.

Example 6.8 Mappings containing a repeating element group.

```
<frame>
  <topic outgoing_only="true">ee.stacc.coordinates.list</topic>
  <format>json</format>
  <mappings>
    <mapping
      global_ref="http://www.example.org/lang/owl#Name"
      path="/location/placename" />
    <repeating_element_group path="/location/coordinates">
      <mapping
        global_ref="http://www.example.org/geoinfo/owl#Latitude"
        path="/location/coordinates/latitude" />
      <mapping
        global_ref="http://www.example.org/geoinfo/owl#Longitude"
        path="/location/coordinates/longitude" />
    </repeating_element_group>
  </mappings>
</frame>
```

The first `mapping` element represents the `placename` element in the message with the path of `/location/placename`, indicating the location of the atomic data element in the message, and with the global reference attribute, referring to `http://www.example.org/lang/owl#Name`, which is an example OWL class that does not exist but indicates that the atomic data element represents a name.

The two mappings in the repeating element group element represent the atomic data values of latitude and longitude. Both of them have the same global references (`http://www.example.org/geoinfo/owl#Latitude` and `http://www.example.org/geoinfo/owl#Longitude`) referring to latitude and longitude that other messages use in their messages. They both also have `path` attribute (`/location/coordinates/latitude` and `/location/coordinates/longitude`) indicating to the location of the atomic data values in the message. This information can be used to extract the atomic data values from the message and use those atomic data values in creating other messages. Note that the `repeating_element_group` element has also the path attribute which indicates to the location of the array where the repeating data values are located.

To test whether the Transformer Widget can transform that message, the "Send Multiple Coordinates" can be clicked which publishes the message. If the transformations were successful, then three messages with coordinates should appear to the Received Messages area on the Widget A (and also messages should appear on the Widget C). Note that since the published message contained an array of three coordinates, then three separate messages were generated for the Widget A because it cannot receive arrays, therefore for each element in the array, a separate message was generated and sent to the Widget A. This proves that the Transformer Widget can be used in similar scenarios where messages are exchanged that contain arrays.

The Widget C has receives messages based on the data published by the Widget A and Widget B. The Widget C is for testing whether the Transformer Widget is capable of generating messages from data elements collected from different sources, i.e. if the Transformer Widget is capable of aggregating data from different widgets and using it to generate new messages (not just doing plain translation from one data format to another). The Widget C has subscribed to receive messages with the topic `ee.stacc.timespace.json` that contain a timestamp and a pair of coordinates and with the topic `ee.stacc.places.list` that contain a timestamp and an array of coordinates. To allow the Transformer Widget to generate messages for the Widget C, a mappings configuration must to be added to the mappings configuration file that would specify the content of those messages.

An example of messages with the topic `ee.stacc.timespace.json` that the Widget C is able to interpret is shown in the Example 6.9 on the next page. The message above contains four atomic data values: `latitude`, `longitude`, `srs` (coordinate reference system) and `timestamp`. The `srs` has a constant value of `EPSG:4326` which is added to every message that the Transformer Widget generates. In our example, it is used to test whether the Transformer Widget is capable of generating messages with constant values.

The mappings configuration corresponding to that message would be as shown in the Example 6.10 on the following page. The mappings configura-

Example 6.9 A message containing aggregated data.

```
{"location":
  {"coordinates":
    {"latitude":58.36611, "longitude":26.73611, "srs":"EPSG:4326"},
    "timestamp":1271690470671}}
```

Example 6.10 Mappings of the message shown in the Example 6.9.

```
<frame>
  <topic>ee.stacc.timespace.json</topic>
  <format>json</format>
  <schema>schemas/timespace.js</schema>
  <mappings>
    <mapping>
      <global_ref>http://www.example.org/lang/owl#Date</global_ref>
      <path>/location/timestamp</path>
      <default>1271690470671</default>
    </mapping>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Latitude</global_ref>
      <path>/location/coordinates/latitude</path>
    </mapping>
    <mapping>
      <global_ref>http://www.example.org/geoinfo/owl#Longitude</global_ref>
      <path>/location/coordinates/longitude</path>
    </mapping>
    <constant path="/location/coordinates/srs" value="EPSG:4326" />
  </mappings>
</frame>
```

tion contains three mappings and one constant. The three mappings specify the timestamp, the latitude and the longitude atomic data values in the message. The global references of those mappings correspond to the same global references in the other messages in the Web application that use timestamp, latitude and longitude.

The global reference with the value of `http://www.example.org/lang/owl#Date` which identifies the `timestamp` element is the same global reference as in the mappings configuration of the message that the Widget A is publishes when sending timestamps of the current time. The `path` of the timestamp atomic data value is `/location/timestamp` that indicates the location of that element in the message. The timestamp mapping has a default value of `1271690470671` which is defined in the `default` element. The default value is used when a corresponding timestamp value has not been collected by the Transformer Widget. In our example application, if the Widget A has not published a timestamp message, then this message for the Widget C is generated with the default value.

The mappings of latitude and longitude elements contain also the same global references (`http://www.example.org/geoinfo/owl#Latitude` and `http://www.example.org/geoinfo/owl#Longitude`) as the other mapping configurations of messages that use latitude and longitude. The `path` elements (`/location/coordinates/latitude` and `/location/coordinates/longitude`) indicate the location of the latitude and longitude atomic data values in the message.

The constant in the mappings configuration defines the value of `EPSG:4326` which is inserted to every generated message. The `path` attribute specifies the location of the element where the constant value is to be used.

The JSON schema `schemas/timespace.js` referred in the mapping configuration is used to generate those messages in JSON and is shown in the Example 6.11 on the next page.

To test whether the Transformer Widget is capable of transforming messages to the Widget C, then one of the buttons on the Widget B can be clicked which publish messages containing coordinates. If the widget C has successfully received a message that correspond to the mappings shown in

Example 6.11 JSON schema corresponding to the message shown in the Example 6.8.

```
{"type":"object",
  "properties":{
    "location":{
      "type":"object",
      "properties":{
        "coordinates":{
          "type":"object",
          "properties":{
            "latitude":{"type":"number"},
            "longitude":{"type":"number"},
            "srs":{"type":"string"}
          }
        },
        "timestamp":{"type":"integer"}
      }
    }
  }
}
```

the Example 6.10 on page 93, then a similar text is outputted by the Widget C surrounding a black border: “Package: lat: 58.36611; long: 26.73611; srs: EPSG:4326 time: Mon Apr 19 18:21:10 GMT+300 2010”. If the Widget A has previously sent a message containing a timestamp value, then the message sent to the Widget C contains the same timestamp value, otherwise the default timestamp value is used. This proves that the Transformer Widget is capable of aggregating data from different widgets and using data from different locations to generate new messages.

The Widget C is also capable of receiving messages with the topic `ee.stacc.places.list` that contain timestamp values and arrays of coordinates. In the Example 6.12 on the next page is shown such a message. The message contains a `time` element containing a timestamp value and a `places` element containing an array of objects containing latitude (`lat` element), longitude (`long` element), `srs` (a coordinate reference system) and `height` elements. The Widget C is receiving those messages to test whether

Example 6.12 Message containing an array.

```
{"route":{
  "time":1274273219312,
    "places":[
      {"lat":58.36611, "long":26.73611, "srs":"EPSG:4326", "height":800},
      {"lat":59.36611, "long":27.23611, "srs":"EPSG:4326", "height":800},
      {"lat":60.36611, "long":27.73611, "srs":"EPSG:4326", "height":800}
]}}
```

the Transformer Widget is capable of generating arrays of data values which have been aggregated from messages of other widgets.

The mappings configuration corresponding to the above message is shown in the Example 6.13 on the following page. In the mappings it is specified that the topic of the messages is `ee.stacc.places.list`, the data format is JSON and the schema is located at `schemas/placesList.js`. The mappings configuration contains three mappings and one constant value.

The first `mapping` element represents a timestamp atomic data value with the global reference of `http://www.example.org/lang/owl#Date` (same as in other messages containing timestamps) and with the path of `/route/time`. That defines the time element in those messages.

The two other mapping elements are located in the `repeating_element_group` element which means that they represent data values in an array. The `repeating_element_group` element has a `path` attribute `/route/places` which indicates to the location of the array in those messages. The two mappings inside the `repeating_element_group` element represent the latitude (`lat` element) and longitude (`long` element) atomic data values. Again, they have the same global reference values (`http://www.example.org/geoinfo/owl#Latitude` and `http://www.example.org/geoinfo/owl#Longitude`) as other mapping configurations that represent messages containing coordinates. The corresponding `path` elements with values of `/route/places/lat` and `/route/places/long` indicate the location of latitude and longitude in the messages.

The element `constant` in the mappings configuration defines the constant

Example 6.13 Mappings corresponding to the message shown in the Example 6.12.

```
<frame>
  <topic>ee.stacc.places.list</topic>
  <format>json</format>
  <schema>schemas/placesList.js</schema>
  <mappings>
    <mapping
      global_ref="http://www.example.org/lang/owl#Date"
      path="/route/time"
    />
    <repeating_element_group path="/route/places">
      <mapping
        global_ref="http://www.example.org/geoinfo/owl#Latitude"
        path="/route/places/lat"
      />
      <mapping
        global_ref="http://www.example.org/geoinfo/owl#Longitude"
        path="/route/places/long"
      />
      <constant path="/route/places/height" value="800" />
    </repeating_element_group>
  </mappings>
</frame>
```

Example 6.14 JSON schema corresponding to the message shown in the Example 6.12 on page 96.

```
{"type":"object",
  "properties":{
    "route":{
      "type":"object",
      "properties":{
        "time":{"type":"number"},
        "places":{
        "type":"array",
        "items":{
          "type":"object",
          "properties":{
            "lat":{"type":"number"},
            "long":{"type":"number"},
            "srs":{"type":"string", "default":"EPSG:4326"},
            "height":{"type":"number"}
          }
    }}}}
}}
```

value that is used in the messages. The value is `800` and the path indicating the location of the value is `/route/places/height`.

The JSON schema at `schemas/placesList.js` referred in the `schema` element that represents the structure of those messages is shown in the Example 6.14. Note that the schema contains the element `srs` which was not referred in any `path` element in the mappings configuration. The `srs` element has a default value of `EPSG:4326` which is defined directly in the schema and used when new messages are generated.

To test if the Transformer Widget is capable of generating such messages, then the "Send Data" on the Widget A that publishes timestamps and the "Send Multiple Coordinates" or the "Send Coordinates" button which publishes coordinates must be clicked. If then the Widget C outputs text "Coordinates package" which contains a timestamp (for example a row with the text "time: Wed May 19 16:56:32 GMT+300 2010") and a list of coordinates (for example a row with the text "lat: 58.36611 - long: 26.73611, srs: EPSG:4326, height: 800"), then the Transformer Widget has successfully composed a message as specified in the mappings and can be used in similar scenarios where messages containing arrays and constant values are needed to be generated from aggregated data.

The example application validates the usability of the Transformer Widget that can be used in transforming messages exchanged by widgets connected to a hub in a Web application. It also proves that the semantic integration approach proposed in this work is valid and sound. The example application has proven that it is enough for transforming messages by specifying the mappings configuration and the schemas (when necessary) of the exchanged messages.

Chapter 7

Future Work

It is possible to improve the Transformer Widget by adding various features that could make the collaboration of widgets and building of mashups better. The Transformer Widget can be improved by adding support to new data formats (XML, CSV etc) that widgets can use in exchanging messages. That would make the Transformer Widget useful for boarder range of widgets that may use various data formats.

One more immediate extension would be to add support for recursive use of arrays (i.e. arrays inside arrays or repeating element groups within repeating element groups) which would allow using of more complex data structure where an array can contain other arrays as elements which again can contain arrays inside those arrays and so on.

It is also possible to extend the Transformer Widget by implementing support for additional aggregation rules in the mappings configuration file that would allow more complex control over combining data from messages exchanged between different widgets.

Aggregated data values are not stored separately from data packages, so if the message has been generated and published from a data package, then the data package is removed along with the data it has stored. One possible option of extending the Transformer Widget is storing aggregated data values separately from data packages so if a new data package is generated with a newly received data values, then the previously stored data values can be used in filling the rest of the data package with data. That would make it possible to publish a new message every time at least one data value in the message has been updated.

That would raise new questions regarding storing and aggregating data values. It should be then specified how many similar data values with the same global reference should be stored before the older data values are replaced with newer ones when messages that are being generated contain arrays. The number of data values to be stored separately can be constrained either with a maximum number to be stored or with a time limit of how much time are data values kept. Another option would be to specify that the same data is not sent twice to the same widget so that if an array of data values is sent to a widget, then the message does not contain any data values that the widget has already received, but this is how the Transformer Widget currently works. One option would be to store separately only the latest data value with the same global reference and then to specify which messages should use previously stored data values to fill in missing data when a new data package is generated.

In the other way around, one mapping rule that can be added could be used to specify the size of arrays to be created, i.e. how many elements should an array contain. For example, if there is a widget that receives messages that contain an array of coordinates, then how many coordinates should be collected for the array before the message is sent out. Again, there can be time constrains and numeric constrains, i.e. it could be specified what is the minimum number of elements the array should contain, or it could be specified what is the maximum time limit for collecting elements for the array (e.g. there should be minimum of five seconds before a new message is generated). A time constraint could be added not only for constructing arrays, but in generating messages in general, so that a time limit could be specified for how frequently new messages are generated for a widget (e.g. do not send messages with a topic more frequently than in every five seconds).

In addition to specifying the number of data values stored, it would also be possible to extend the Transformer Widget to specify the number of data packages stored in memory. For example, if there are two widgets, one capable of sending an array of coordinates, but other widget capable of receiving (and processing) only one pair of coordinates at the time, so if the first widget sends an array of coordinates with three locations, then which of those three should be sent to the other widget? Currently the Transformer Widget would generate, in this example, three separate message of each location and send it to the second widget. The Transformer Widget could be extended to allow specifying rules to either send only the latest generated data package, only the first generated data package, or a number of latest data packages. For example, if a widget sends an array of five coordinates and the receiving widget accepts only one coordinate at the time, then it would be possible to specify to generate a message to the receiving widget either with the first coordinate from the array, with the last coordinate from the array, or with the last three coordinates with each in a separate message. The package count limitation could also be applied in cases a message is composed of data from two sources where one widget publishes messages more frequently than the other one and if the maximum number of data packages stored would be set to one, then only the latest data values from the more frequent source would be used in composing a message when the less frequent data values are received. If this rule would be applied then some data values from the more frequent source would be lost and not used in any generated message.

The aggregation rules can be extended to be able to specify topics of messages that can be used in collecting data values for generating a certain messages. This would limit by topics of where widgets can receive information. This means that not only the global reference of a data value must match, but also the topic where the data value was extracted. For example, if a set of widgets exchange coordinate in their messages, but half of the widgets exchange coordinates of railway stations and other widgets exchange coordinates of court houses, then if we would like to keep those concerns separates so that the coordinates of railway stations and the coordinates of court houses would not get mixed up in messages, then a more context aware aggregation rules are needed to be implemented.

The Transformer Widget could also be extended by adding various data operations to the mappings rules which would allow more complex aggregation and manipulation of data. The data operations could be sorting, filtering, looping, regular expressions, counting, keyword extraction etc which is similar to the data-flow model of how Yahoo! Pipes [46] allows building mashups from aggregated data.

A similar approach is introduced in a complex event processing system in distributed systems [22] where causal event histories, event patterns, event filtering, and event aggregation are introduced. Many of the ideas like casual event history visualization which can be used for diagnostics in more abstract level can also be used in this project since events (messages) in distributed systems (independent widgets) are dealt in the Transformer Widget as well.

The Transformer Widget could also be extended with a graphical user interface that would help developers in creating configurations of mappings for mashups. A graphical user interface would allow visualizing configurations of mappings in a more user friendly manner and would reduce the time of learning to understand mappings and create new configurations. For example, the user interface could allow using a drag-and-dropping of data values between different messages to imply connection and similarity of those data values which can be then used in message generation.

A graphical user interface would make it easier for a user to configure mappings, but it may still be tedious if exchanged data and schemas become more complex where manual configuration of mappings would take too much effort. A possible extension to the Transformer Widget would be to add support for automatic schema matching instead of manual creation of mappings. There are numerous approaches (some of them reviewed in [31, 43, 27]) for automatic semantic integration which could then be considered further if automatic matching is to be implemented.

Conclusion

This thesis focuses on the semantic integration of data exchanged in messages sent by independent widgets developed by different vendors to enable collaboration between widgets on a Web application. The Transformer Widget, which was implemented in the thesis, allows building of mashups with complex application logic where loosely coupled components (widgets) can collaborate and perform tasks that would otherwise be difficult to implement using widgets.

The thesis fist gave an overview of current mashup providers and related standards to introduce the problems current mashup platforms and widgets have with content collaboration. It was found that there are no platforms available that would allow Web widgets (developed by different vendors) to properly collaborate with each other, which poses a severe limitation to creating sophisticated Web applications that require integration of different data sources.

A solution for a semantic integration of messages exchanged by widgets was proposed to overcome the limitations current mashup platforms have. The thesis proposed a solution for defining mappings of data elements in the exchanged messages that would allow linking of data with corresponding terminology in ontologies. That would allow automatic understanding of content in messages and data elements could be collected from the data published by widgets. New messages could be generated from the collected data and sent to widgets that could interpret those messages. This would allow integration of data from different sources so that interactive collaboration could be supported.

The proposed solution was implemented as an independent widget called

Transformer Widget. It uses OpenAjax Hub that provides a central hub where all the widgets on the Web application can connect and exchange messages. The Transformer Widget receives all the messages that are being published by other widgets and uses preconfigured mappings to aggregates data from those messages and generate new messages that would be interpretable by widgets interested in the aggregated data.

The thesis gives an overview of the implementation of the Transformer Widget and validates the usability of the Transformer Widget with the help of another Web application that is using three widgets that cannot communicate with each other directly. The thesis proposes a solution for enabling communication between those widgets with the help of the Transformer Widget by configuring the mappings corresponding to the messages exchanged by those three widgets and adding the Transformer Widget to integrate the data exchanged between the widgets. The test results proved the solution and the Transformer Widget to be valid. The successful transformations of the messages by the Transformer Widget allowed the widgets in the test application to exchange data between each other and proved that the solution of integrating data proposed by the thesis is indeed valid and can be used in enabling collaboration between widgets.

Semantiline Integratsiooni Platvorm Veebividinate Suhtlemiseks

Magistritöö (30 EAP)

Rainer Villido

Resümee

Semantiline integratsiooni platvorm veebividinate suhtlemiseks on raamistik mashup-tüüpi veebirakendusel suhtlemise võimaldamiseks erinevate lõdvalt seotud veebikomponentide (veebividinate) vahel.

Antud magistritöö pakub välja lahenduse integreerimaks semantiliselt erinevate vidinate poolt saadetud andmed, nii et vidinad oleks võimelised omavahel andmeid jagama, kui nad vahetult üksteise poolt saadetud sõnumeid tõlgendada ei oska. Kõikide vidinate poolt vahetavate sõnumite andmeelemendid seotakse erinevate ontoloogia terminitega, mis võimaldab sõnumite sisu masinloetavaks ja -arusaadavaks muuta, nii et saadetud sõnumitest oleks võimalik korjata kokku kõik vajalikud andmeelemendid, milleks oleks võimalik koostada uusi sõnumeid. See võimaldab kombineerida erinevate vidinate poolt saadetud andmeid ja luua uusi sõnumeid erinevatest allikatest kombineeritud andmetest ning seejärel saata loodud sõnumid edasi nendele vidinatele, kelle jaoks on need andmed kasulikud.

Lahendus vidinatevahelise koostöö hõlbustamiseks realiseeriti kasutades raamistikku OpenAjax Hub [2], mis on keskne sõnumite jaotur (*hub* ing. k.) lubamaks vidinatel jaoturi kaudu sõnumeid vahetada. Jaoturi kasutamine võimaldab küll vidinatel omavahel sõnumeid vahetada, kuid ei lahenda probleemi, kui vidinad kasutavad sõnumite vahetamiseks erinevaid andmeformaate ja -struktuure. Lahendusena realiseeriti magistritöö raames eraldiseisev vidin nimega Transformatsioonividin (*Transformer Widget* ing. k.), mis kogub andmeelemente kõikidest sõnumitest, mida vidinad publitseerivad. Seejärel genereerib Transformatsioonividin uusi sõnumeid varemkogutud andmeelementidest ja saadab need teistele vidinatele, mis oskavad kogutud andmeid kasutada. Magistrtöö raames defineeriti eeskirjad sõnumite sisu kirjel-

damiseks, kus sõnumis esinevad andmeelemendid vastandatakse ontoloogia terminitega, mille põhjal oskab Transformatsioonividin erinevate vidinate poolt saadetud sõnumeid interpreteerida ja uusi sõnumeid genereerida.

Transformatsioonividina kasutatavust testiti kolmest vidinast koosneva näidisrakenduse peal, mille vidinad omavahel otse suhelda ei osanud. Testi eesmärk oli selgitada, kas transformatsioonividinat saab kasutada sellelaadsete juhtumite puhul vidinatevahelise suhtlemise tagamiseks, kus olemasolevad vidinad ei saa üksteiste poolt saadetud sõnumitest aru. Testi käigus kirjeldati vidinate poolt saadetud sõnumite semantika ja struktuur Transformatsioonividinale arusaadavale kujule, mis võimaldas vidinate poolt saadetud andmeid transformeerida nii, et andmed, mida vidinad publitseerisid, muudeti arusaadavaks ka teistele vidinatele. Näidisrakenduse test oli edukas ja kinnitas Transformatsioonividina kasulikkust selliste probleemide lahendamisel.

Bibliography

[1] OpenAjax Alliance. Openajax alliance. `http://www.openajax.org`, 2010. Cited: 9 May, 2010.

[2] OpenAjax Alliance. Introducing openajax hub 2.0 and secure mashups. `http://www.openajax.org/whitepapers/Introducing%20OpenAjax%20Hub%202.0%20and%20Secure%20Mashups.php`, 2009. Cited: November 13, 2009.

[3] OpenAjax Alliance. Openajax hub 2.0 specification - managed hub overview at openajax alliance wiki. `http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification_Managed_Hub_Overview`, 2009. Cited: December 15, 2009.

[4] The Open Mashup Alliance. The open mashup alliance for enterprise mashups. `http://www.openmashup.org/`. Cited: November 13, 2009.

[5] Apple. Apple dashboard. `http://www.apple.com/downloads/dashboard/`, 2010. Cited: May 14, 2010.

[6] JackBe Corporation. Presto: An enterprise-ready mashup solution. `http://www.jackbe.com/products/`, 2009. Cited: May 17, 2010.

[7] Miguel Carrillo Pacheco et al. Morfeo fast (fast and advanced storyboard tools). major scientifc objects of the fast project. Technical report, Morfeo, 30 July 2008. `https://files.morfeo-project.org/fast/public/8.2_major_scientific_objectives.pdf`.

[8] OpenSocial Foundation. Opensocial. `http://www.opensocial.org/`, 2010. Cited: May 9, 2010.

[9] The Apache Software Foundation. Overview of apache shindig. apache incubator. http://incubator.apache.org/shindig/overview.html, 7 July 2009. Cited: November 16, 2009.

[10] The Apache Software Foundation. Apache ant. the apache software foundation. http://ant.apache.org/, 2010. Cited: May 13, 2010.

[11] Google. Api overview - gadgets api. google code. http://code.google.com/apis/gadgets/docs/overview.html, 2009. Cited: November 10, 2009.

[12] Google. Google desktop. http://desktop.google.com/, 2009. Cited: May 14, 2010.

[13] Google. Compile & debug - google web toolkit. google code. http://code.google.com/webtoolkit/doc/latest/DevGuideCompilingAndDebugging.html, 2010. Cited: April 21, 2010.

[14] Google. Google web toolkit - get started - faq. google code. http://code.google.com/webtoolkit/doc/latest/FAQ_GettingStarted.html, 2010. Cited: May 16, 2010.

[15] Google. Google web toolkit overview. google code. http://code.google.com/webtoolkit/overview.html, 2010. Cited: April 21, 2010.

[16] Google. Javascript native interface (jsni). google web toolkit. http://code.google.com/webtoolkit/doc/latest/DevGuideCodingBasicsJSNI.html, 2010. Cited: April 28, 2010.

[17] Thomas Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5:199–220, April 1993.

[18] Volker Hoyer. Enterprise mashups. http://sites.google.com/site/fastonlinecontest/what-are-enterprise-mashups, 2009. Cited: November 2, 2009.

[19] IBM. Ibm mashup center. http://www-01.ibm.com/software/info/mashup-center/. Cited: November 4, 2009.

[20] The Apache Incubator. Wookie proposal. apache incubator whiteboard. http://wiki.apache.org/incubator/WookieProposal, 20 September 2009. Cited: November 20, 2009.

[21] Intel. Intel mash maker. http://mashmaker.intel.com/web/learnmore.html, 2009. Cited: November 6, 2009.

[22] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical report, Stanford University, 1998.

[23] Microsoft. Windows sidebar and gadgets. http://www.microsoft.com/windows/windows-vista/features/sidebar-gadgets.aspx, 2010. Cited: May 14, 2010.

[24] Chris Mills. Opera widgets specification 1.0 fourth edition. opera development community. http://dev.opera.com/articles/view/opera-widgets-specification-fourth-ed/, 15 September 2009. Cited: November 11, 2009.

[25] Netvibes. Netvibes download. http://netvibes.org/download/. Cited: November 11, 2009.

[26] Netvibes. Netvibes developers network. http://dev.netvibes.com/doc/, 3 July 2009. Cited: November 12, 2009.

[27] Natalya F. Noy. Semantic integration: A survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, 2004.

[28] Natalya F. Noy, AnHai Doan, and Alon Y. Halevy. Semantic integration. *AI Magazine*, 21(1):7–9, 2005.

[29] OMTP. Bondi. http://bondi.omtp.org, 2009. Cited: January 11, 2010.

[30] Tim O'Reilly. What is web 2.0? o'reilly media. http://oreilly.com/web2/archive/what-is-web-20.html, 30 September 2005. Cited: January 6, 2010.

[31] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[32] Sven Rizzotti and Helmar Burkhart. usekit: a step towards the executable web 3.0. In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 1175–1176, Raleigh, North Carolina, USA,, 2010. ACM.

[33] Jesse Ruderman. Same origin policy for javascript. mozilla developer center. https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript, June 2009. Cited: January 4, 2010.

[34] Justin Schuh. Same-origin policy part 1: Why we're stuck with things like xss and xsrf/csrf. the art of software security assessment. http://taossa.com/index.php/2007/02/08/same-origin-policy/, February 2007. Cited: January 10, 2010.

[35] Antero Taivalsaari and Tommi Mikkonen. Mashups and modularity: Towards secure and reusable web applications. In *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 25–33, L'Aquila, 2008. IEEE.

[36] Kapow Technologies. Getting started : Learn more about openkapow robots. openkapow. http://openkapow.com/blogs/getting_started/archive/2007/02/15/Learn-more-about-openkapow-robots.aspx, 15 February 2007. Cited: November 6, 2009.

[37] Kapow Technologies. Kapow web data server. http://kapowtech.com, 2009. Cited: November 6, 2009.

[38] TIBCO. Tibco pagebus developer's guide. `http://developer.tibco.com/resources/gi/pagebus2/tib_pgbs_dev_guide.pdf`, October 2009. Cited: March 3, 2010.

[39] Ivan; Hoyer Volker; Janner Till; Rivera Ismael; Aschenbrenner Nina; Fradinho Manuel; Lizcano David Urmetzer, Florian; Delchev. State of the art in gadgets, semantics, visual design, sws and catalogs. Technical report, Morfeo, 27 February 2009.

[40] The World Wide Web Consortium (W3C). Owl web ontology language – w3c recommendation. `http://www.w3.org/TR/owl-features/`, 10 February 2004. Cited: May 20, 2010.

[41] The World Wide Web Consortium (W3C). Widgets 1.0: Packaging and configuration. `http://www.w3.org/TR/widgets/`, 23 July 2009. Cited: October 27, 2009.

[42] World Wide Web Consortium (W3C). Widgets 1.0: The widget landscape (q1 2008), 2008.

[43] H. Wache, T. Vogele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hubner. Ontology-based integration of information - a survey of existing approaches. In *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing*, pages 108–117, Seattle, WA, 2001.

[44] Wookie. Wookie widget develope's guide. `http://getwookie.org/Widgets_files/widget_dev_guide.pdf`, 8 December 2008. Cited: November 20, 2009.

[45] WS02. Mashup server data sheet. `http://wso2.com/wp-content/themes/wso2ng/images/wso2_mashup_product_data_sheet.pdf`. Cited: November 13, 2009.

[46] Yahoo! Yahoo! pipes. `http://pipes.yahoo.com/pipes/`, 2010. Cited: May 9, 2010.

[47] Ikuya Yamada, Wataru Yamaki, Hirotaka Nakajima, and Yoshiyasu Takefuji. Ousia weaver: A tool for creating and publishing mashups as impressive web pages. In *MEM 2010: 3rd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web, in WWW 2010: Proceedings of the 18th International World Wide Web Conference*, Raleigh, North Carolina, USA,, 2010. WWW 2010.

[48] Jin Yu, Boualem Benatallah, Fabio Casati, and Florian Daniel. Understanding mashup development. *IEEE Internet Computing*, 12:44–52, 2008.

[49] Nan Zang, Mary Beth Rosson, and Vincent Nasser. Mashups: who? what? why? In *CHI '08: CHI '08 extended abstracts on Human factors in computing systems*, pages 3171–3176, Florence, Italy, 2008. ACM.

Appendix

Source Code

The source code of the Transformer Widget and the test application used to test transformations, which were discussed in the thesis, is available upon request from Software Technology and Applications Competence Center.