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

Karli Kirsimäe

Automated OpenAjax Hub Widget Generation for Deep Web Surfacing

Master's thesis (30 ECTS)

Advisor: Peep Küngas

Author: .... "..." May 2011

Advisor: .... "..." May 2011

Approved for defence
Professor: .... "..." May 2011

Tartu 2011

Contents

Abstract **5**

1 Introduction **6**

2 Architecture **11**

2.1 Client-side . . . 11

2.1.1 OpenAjax Hub . . . 11

2.1.2 Transformer Widget . . . 12

2.1.3 Proxy Widget . . . 13

2.1.4 ProxyWidgetUtils . . . 13

2.2 Server-side . . . 14

2.3 Used Components and Technologies . . . 14

2.3.1 SOAP Web Services . . . 14

2.3.2 WSDL . . . 14

2.3.3 SAWSDL . . . 15

2.3.4 JSON . . . 15

2.3.5 JSON-RPC . . . 15

2.3.6 JSONP . . . 15

2.3.7 JSON Schema . . . 16

2.3.8 SMD . . . 16

2.3.9 Dojo Toolkit . . . 16

3 Implementation **17**

3.1 The Client-side . . . 17

3.1.1 Setting Up The Client-side . . . 17

3.1.2 Creating the Proxy Widget 19
3.1.3 Creating a Consumer for the Proxy Widget 22
3.1.4 Consuming the Proxy Widget 25
3.2 The Server-side 28
3.2.1 Generating Mappings 28
3.2.1.1 Transforming SOAP Message Structure to JSON 30
3.2.1.2 Implementation of the MappingGenerator . . 31
3.2.1.3 Generating Topic Names 31
3.2.1.4 Setting Default Values in Mappings 31
3.2.2 Generating JSON Schemas 33
3.2.3 Generating SMD Documents 36
3.2.4 Proxying Requests to the SOAP Service 37

4 Proof of Concept 41

5 Related Work 48
5.1 Service Composition with Mappings 48
5.2 Graphical Tools for Mashup Generation 48
5.3 Categorizing Mashup Tools and Environments 50
5.4 The Future of Mashup Development 51
5.5 Bypassing Same Origin Policy 52
5.6 Accessing SOAP Services From Browser 53
5.7 Problems With Mashups 54
5.8 Semantic Web Services 54

6 Conclusion 55

7 Future Work 57

Abstract (in Estonian) 59

Bibliography 61

Appendix 65

Abstract

The Deep Web, as the name implies, is typically hidden from a common web user, because the information it contains, is not findable through standard search engines. However, this hidden information is often useful to the web user. The question is, what are the possibilities to surface those resources?

An example of Deep Web resource would be a SOAP web service of Estonian Business Registry. If a developer wants to use this service in a web application, to query data about annual reports, he should create a service client on the server-side and then manually wire together the user interface and the web service. This requires quite a lot of work and knowledge of server-side programming.

Following a current trend where Web application development is geared towards the browser-side implementations[22], what should a developer do in order to create a client-side mashup using Deep Web resources and web widgets to visualize the annual report data? Unfortunately, his possibilities narrow down quite heavily. The creation of SOAP requests on the client-side is not well supported and he should still put up a server-side proxy to request data outside his own domain. And of course, the wiring with visual widgets still requires much work.

This thesis aims to provide a solution that helps a developer to create such client-side mashups. It will provide an infrastructure, that takes care of the cross-domain request problems by creating a common server-side proxy, that anyone could use. It will allow a developer to initiate SOAP requests from within a web browser, by using just JSON request data. Additionally, the solution allows a developer to integrate SOAP web services with visual widgets, by using semantic integration instead of hard-wiring.

Chapter 1

Introduction

There is a growing trend in software industry to move the development toward presentation layer. In the context of Web applications the presentation layer usually consists of browser-interpretable logic and components. At the same time there are efforts to surface the hidden Deep Web resources. The reason for this is the amount of information that is currently kept a way from a Web user.

Part of the Deep Web are the web services, among which the SOAP web services are of particular interest in this thesis. SOAP web services were chosen, because compared to many other web service protocols, these are much more difficult to call from the presentation layer. As an example, imagine the work a developer should currently do, to query a list of annual reports from Estonian Business Registry SOAP service and then visualize this information as a table on a web page. First of all, a developer should set up a server-side proxy, because it is impossible to call business registry service directly from client-side, unless the original service was altered. As typically the potential service consumers cannot ask the service provider to change the service, an obstacle called Same Origin Policy has to be dealt with and the server-side proxy is probably the only way to reasonable solve this problem. Setting up a server-side proxy requires knowledge of some server-side programming language which sets additional requirements to the set of skills a developer should have.

The next problem, that the developer has to face with, is the lack of support for creating SOAP requests using lightweight browser technologies like JavaScript. Even popular JavaScript APIs do not help to greatly ease the calling of those services. There are some libraries that try to solve this problem, but they usually cannot deal with the complexity of SOAP requests.

Even if the developer is able to successfully query the annual report data from the client-side, he would then have to manually set up the visualization of this information. This means that he must know the exact format of the returned list of annual reports, and know, how to extract the information, that he wants to make visual. The hard-wiring, that needs to be done, is not only inconvenient, but also subject to errors, that might come from changes to the interface.

Before proposing any concrete solutions to the above-mentioned problems, it would be interesting to see what are the current trends and predictions in the field of web technologies. An authoritative source of information is the annual analysis of Web and User Interaction Technologies produced by Gartner. The 2010 release [22] enlists some of the technologies that could have potential in relation to the problems with Deep Web surfacing. Gartner predicts, that mashup applications will soon see enterprise adoption and will provide significant value to the enterprise, which means that the scenarios, like the one described before, will become more frequent. Semantic Web is predicted to give high benefit to the enterprise but will likely take more than ten years to be adopted widely. Semantics could be used to automatically visualize the annual report data from Business Registry. Gartner also reports that Citizen Developers are predicted to gain mainstream adoption in about five to ten years and Web Widgets are predicted to be a mature enough technology in about 2-5 years. Web Widgets are exactly the technology that might help creating client-side mashups like the one with Estonian Business Registry. An illustration of the Web and User Interaction Technologies Hype Cycle is shown in Figure 1.1 on the following page.

In regards of web services, the current trend is that most of the more popular consumer-oriented services support more lightweight protocols such as Representational State Transfer or JSON-RPC while on the enterprise level,

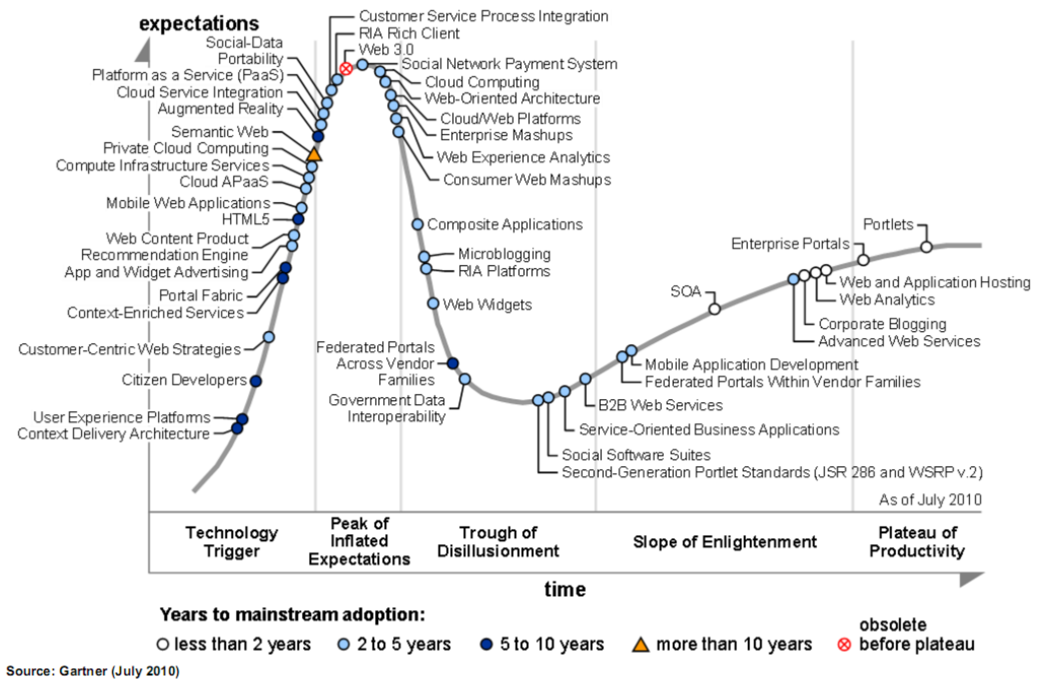


Figure 1.1: Hype Cycle for Web and User Interaction Technologies, 2010 [22]

SOAP is still widely used. At the same time, the popularity of lightweight scripting languages such as JavaScript has been greatly increased in recent years. While the lightweight service protocols are quite well supported by JavaScript APIs, it is rather difficult to involve heavyweight SOAP services into the JavaScript world. Based on that, it makes sense to find ways to transform SOAP services to something more native to the JavaScript world, for example JSON-RPC.

One would wonder, what is the reason, that has kept SOAP services away from JavaScript. Probably one of the main reasons is, that most of the popular services do not support SOAP, so there lacks motivation for the API developers to provide support. Another reason is dealing with XML, that is not so well supported. Good support for XML is needed to really provide seamless integration of SOAP services. JavaScript APIs should have excellent abilities to parse the WSDL documents that are typically used to describe the SOAP services but they currently do not have these abilities.

When the web services are involved in the mashups, they are usually hardwired together and the wiring is most often done on the server-side. Because of hard-wiring, the integration of services in mashups requires a lot of manual work and is subject to the risk of interface changes, as already stated earlier. Another thing is that most of the mashups are typically constructed and deployed on the server of a mashup tool provider. It is desirable to find the possibilities to do the integration on the client-side using JavaScript and other browser-supported technologies.

One way to avoid hard-wiring of the services is to use semantically annotated service descriptions. There is a framework that supports semantic integration of OpenAjax Hub widgets [28] and it might be feasible to exploit that framework to solve the above-mentioned issues - avoid hard-wiring of services and mashing up services on the client-side. There are also other efforts done in the field of unifying web widgets' interfaces and ease their interconnection, like [17] for example. But at the moment, each widget provider has different and not matching interfaces, which makes it hard for them to communicate with each other.

There is one other obstacle that prevents the easy wiring of independent services that are deployed on separate remote sites - this is the Same Origin Policy restriction that modern browsers implement. Same Origin Policy means that a script on a web page can only make requests to the same domain that the web page originates from. This makes it impossible to compose a client-side application with services from remote domains unless a workaround is used.

Another goal with widgets is to minimize the programming effort, that is needed for wiring widgets together, and rely more on the configuration. This enables users with little programming skills to more easily construct widget mashups. This also reduces the potential of errors that can occur.

Keeping in mind the current trends and available technologies, the thesis will propose a solution that helps a developer to overcome the restrictions that were described earlier. The main idea of the solution will be the automated generation of hidden widgets, that can be used to access the SOAP service operations. This widget will use a server-side proxy to bypass the Same Origin Policy. The proxy can be set up once and used by many without any need to care about the set up. The problems with SOAP request creation will be handled by the automatically generated OpenAjax Hub data widgets, that will bridge SOAP services to ease the inclusion of heavyweight web services into client-side mashups. In order to simplify the visualization of data, semantic integration of widgets will be used. For this, the WSDL document of the service should be annotated using SAWSDL attributes. The semantic integration will rely on Transformer Widget that is a special OpenAjax Hub widget that enables widget interconnection.

The proposed solution will provide a simpler way for developers to use many of the SOAP web services, that are available today. By using the automatically generated widgets, they do not have to do any server-side programming or set up the proxy server for communication with services in other domains. This results in a reduced technological barrier to start using the SOAP services in client-side mashups because instead of knowing SOAP, XML and some server-side programming language, understanding pure JavaScript could be enough. The proposed solution is meant to be useful in mashups, that do not involve very complex services or difficult rules for service calling. To survive in more demanding applications, this solution should be developed further.

The rest of the thesis is organized as follows. In chapter 2, the main components and technologies that are used in this thesis, are described. An architectural overview of the solution is also given in this chapter. Chapter 3 goes in detail with the concrete implementation of the main components of the solution. Chapter 3 also gives guidance on how the solution can be used and set up. Chapter 4 explains how the provided infrastructure demonstrated its usefulness in a proof of concept demo application. Related works and alternative solutions are discussed in chapter 5. In chapter 6 a short overview of the achievements are given which are followed by chapter 7 with thoughts on ways to improve the system further.

Chapter 2

Architecture

The proposed solution for automated SOAP widget generation is divided in two separate parts - the client-side and the server-side components. The main responsibility for the client-side is to enable communication with and between SOAP services via hidden web widgets. The main responsibilities for the server-side component are to provide metadata for the client-side and to proxy the service requests from client-side to the actual service endpoints. The client-side itself is additionally divided into OpenAjax Hub, Transformer Widget, Proxy Widget and some helper functions to simplify the generation of proxy widget and setting up the application environment. A conceptual class-diagram of the architecture is shown in Figure 2.1. The components are described in more detail in the following sections.

2.1 Client-side

2.1.1 OpenAjax Hub

OpenAjax Hub [6] is a set of standard JavaScript functionality that addresses key interoperability and security issues that arise when multiple Ajax libraries and/or components are used within the same web page. The standard specification is developed by OpenAjax Alliance. The standard defines a publish/subscribe engine that includes a ”Managed Hub” mechanism that allows a host application to isolate untrusted components into secure sandboxes.

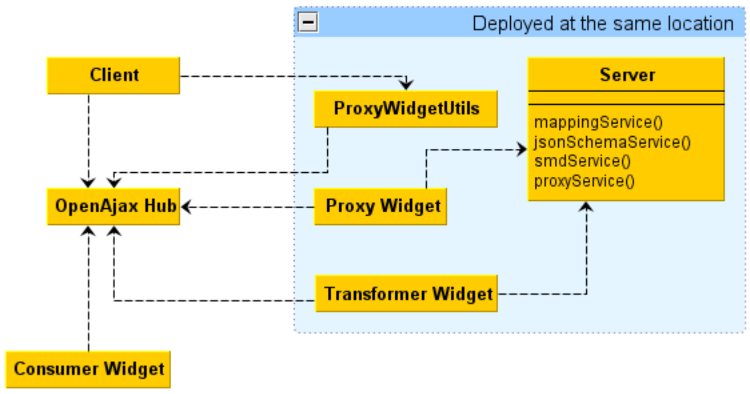
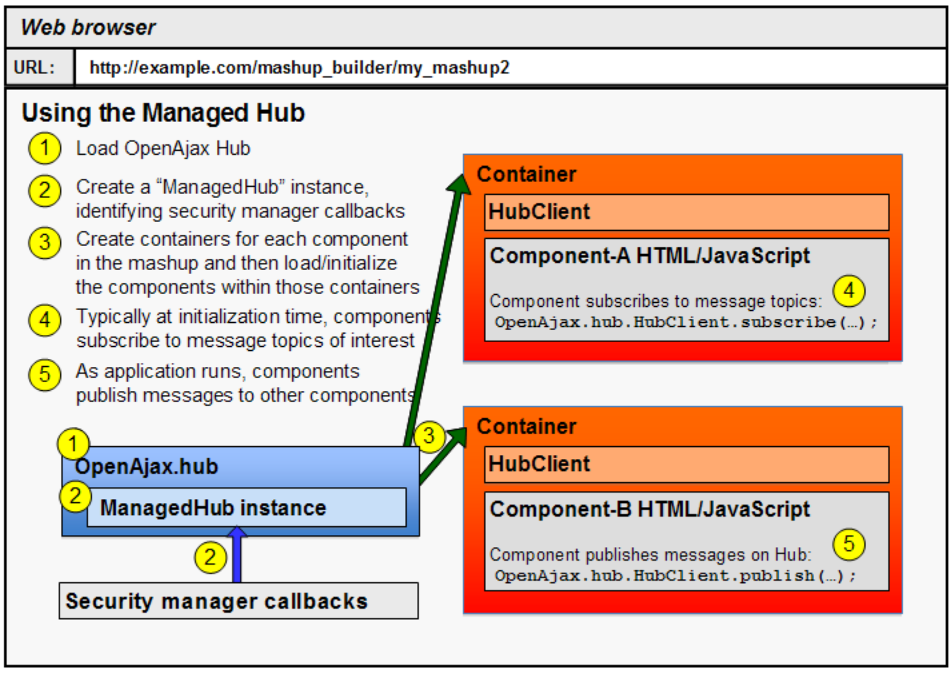


Figure 2.1: A conceptual-view class diagram of client-server architecture.

However, in this solution the primary reason for using OpenAjax Hub is its ability to mediate the inter-widget communication. An illustration of using OpenAjax Hub for mashup assembly is given in Figure 2.2. The way, that OpenAjax Hub is drawn in Figure 2.1, might leave an impression, that the OpenAjax Hub is not grouped together to be deployed with the server-side component. Actually this only shows, that a different implementation of the OpenAjax Hub specification can be used but does not have to. In fact, there is a Tibco implementation bundled with the server-side component and it makes sense to use it.

2.1.2 Transformer Widget

The Transformer Widget [28] is an OpenAjax Hub widget that enables semantic integration of messages exchanged by other OpenAjax Hub widgets. The Transformer Widget uses special mappings of data elements in the exchanged messages that allow linking of data with corresponding terminology in ontologies. The Transformer Widget receives all the messages that are being published by other widgets, uses the mappings to aggregate data from those messages and generate new messages that would be interpretable by widgets interested in the aggregated data. This thesis extended the Transformer Widget in some parts, but most of the development was done in



Web browser
URL:
http://example.com/mashup_builder/my_mashup2
Using the Managed Hub
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
OpenAjax.hub
ManagedHub instance
Security manager callbacks
Container
HubClient
Component-A HTML/JavaScript
Component subscribes to message topics:
OpenAjax.hub.HubClient.subscribe(…);
Container
HubClient
Component-B HTML/JavaScript
Component publishes messages on Hub:
OpenAjax.hub.HubClient.publish(…);

Figure 2.2: Typical usage of the Managed Hub [5]

master's thesis by Rainer Villido [28].

2.1.3 Proxy Widget

The Proxy Widget is an OpenAjax Hub widget that acts as a proxy to a certain operation provided by SOAP service. It uses server-side component to bypass the Same Origin Policy restrictions that prohibit the accessing of resources outside the client domain. The Proxy Widget is entirely created from ground-up in this thesis.

2.1.4 ProxyWidgetUtils

The ProxyWidgetUtils, as drawn in Figure 2.1 on the preceding page, are a set of JavaScript utility functions, that hide the complexity of setting up the mashup environment and help to easily create new Proxy Widgets for a certain SOAP service operation. This component is also the result of work done in this thesis.

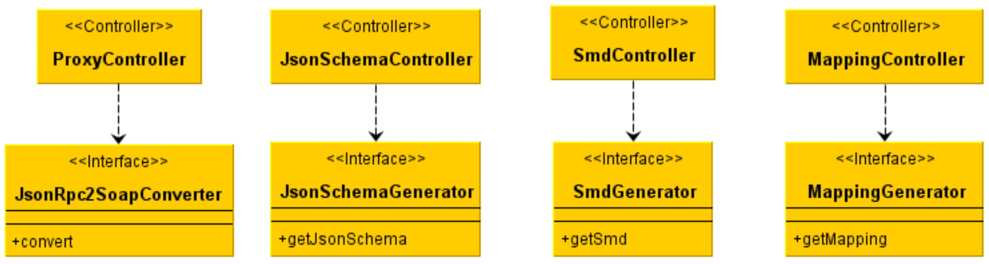


Figure 2.3: A specification view of server-side class diagram

2.2 Server-side

The server-side component provides its services through four controllers as can be seen from Figure 2.3. The controllers should be as lightweight as possible, therefore they each rely on specific components for their core functionality. The server-side uses Spring MVC framework to implement the controllers. Spring is also used for some application configuration and dependency injection.

2.3 Used Components and Technologies

2.3.1 SOAP Web Services

SOAP [10] is a lightweight protocol that uses XML messages to exchange structured information. SOAP is used as a protocol for many heavyweight web services. The proposed solution will enable the calling of SOAP services from within a client browser.

2.3.2 WSDL

WSDL (Web Services Description Language) [11] is an XML format for describing web services. The WSDL document is used to understand the structure of input and output messages of a SOAP service. The WSDL document is also used to extract the semantic annotations of the messages. Based on the structure and semantics, it is possible to generate the mappings document, that the Transformer Widget consumes to mediate communication between widgets.

2.3.3 SAWSDL

SAWSDL [9] defines a set of extension attributes for the Web Services Description Language and XML Schema definition language that allows description of additional semantics of WSDL components. It provides mechanisms by which concepts from the semantic models, typically defined outside the WSDL document, can be referenced from within WSDL and XML Schema components using annotations. The attributes defined by SAWSDL are used to read the semantic vocabulary associated with the structure of SOAP messages.

2.3.4 JSON

JSON [3] is a lightweight open standard for text-based data-interchange format that is both human- and machine-readable. JSON defines a small set of formatting rules for the portable representation of structured data. JSON is the format that is used to send messages between the widgets and to send data to the server component.

2.3.5 JSON-RPC

JSON-RPC [14] is a stateless, light-weight remote procedure call (RPC) protocol that uses JSON as data format. JSON-RPC is the protocol that is used to send messages between the Proxy Widget and the server-side component.

2.3.6 JSONP

JSONP [8] is a way of doing cross-domain requests with the help of using HTML Script tag. JSONP specifies, that the JSON, that is retrieved from the server, should be wrapped inside a callback function, that is specified when initiating the request, so that client-side will be able to respond instantly to the successful response. JSONP is used when service calls are proxied through the server-side component. This enables using the server-side proxy service even without the Proxy Widget. If only the Proxy Widget would be using the proxy service and if they both would be served from the same domain, then using JSONP would not have any effect.

2.3.7 JSON Schema

JSON Schema [13] defines the media type ”application/schema+json”, a JSON based format for defining the structure of JSON data. JSON Schema is generated for each input message that is sent to the server proxy via JSON-RPC. It is required by the Transformer Widget which uses the schema to construct messages with valid structure.

2.3.8 SMD

Service Mapping Description (SMD) [12] is a JSON representation describing web services. It allows the description of JSON-RPC and REST web services. SMD is used by Dojo to automatically generate a JavaScript proxy to call the described service which in this case is actually the JSON-RPC service provided by the server proxy.

2.3.9 Dojo Toolkit

Dojo [1] is a JavaScript toolkit that is widely used by many large companies. Dojo is used in this solution to create JavaScript service wrappers based on SMD documents and to request the SMD document from the server-side.

Chapter 3

Implementation

3.1 The Client-side

The client-side consists of various components as can be seen from Figure 2.1 on page 12. The way that client-side components are structured and served by the server-side can be seen from Table 3.1 on the following page. The client-side depends heavily on services on the server-side. Therefore it is important, that before starting with the client-side, a developer makes sure, that the server-side component is up and running.

3.1.1 Setting Up The Client-side

A client-side application can be served and set up separately from the server-side component, although it depends on it. This means, that the developer of the client-side does not need to have access to the server-side component, or change anything on the server-side, to create the client-side application. The minimal client-side application could consist of the main application HTML page and the tunnel.html file, because most of the dependencies are bundled with and can be included from the deployed server-side component. The tunnel.html file is needed by the OpenAjax Hub to enable messaging between widgets in IFrame containers. The main HTML file is where the client-side mashup application is defined.

The main application HTML file must follow certain rules to use the au-

Path from server-side root	Description
/TransformerWidget.html	The main file for loading the Transformer Widget.
/transformerwidget/	The directory, that contains the compiled browser-specific permutations, that are used based on the used browser.
/widgets/pagebus.js	The TIBCO implementation of the OpenAjax Hub 2.0 standard.
/widgets/ProxyWidget.html	The main file for loading the Proxy Widget
/widgets/ProxyWidget/ProxyWidget.js	Main functionality of the Proxy Widget
/widgets/ProxyWidgetUtils.js	Utility functions to ease the set up of the environment and creation of Proxy Widgets

Table 3.1: Structure of required client-side components

tomatic widget generation functionality. The main page needs to include two JavaScript files - an implementation of the OpenAjax Hub by Tibco (`pagebus.js`) and a JavaScript file `ProxyWidgetUtils.js` that contains utility functions. The Tibco implementation of OpenAjax Hub with pagebus support is needed to enable message caching. The `ProxyWidgetUtils.js` contains functions for generating Proxy Widgets and initializing the client-side application. It also includes some of the boilerplate callback methods needed by the OpenAjax Hub. These methods can be easily overloaded in the main application file, if needed. Both of these files are bundled with the server-side component and can be accessed from there.

When the two required JavaScript files have been included, it is possible to start setting up the application. The main application HTML document must have a container for all the OpenAjax Hub widgets. The container must have an id `"mashupArea"`. All the generated widgets will be placed inside that container. Most common solution is to add a `div` tag with that id inside the `body` tag.

The setting up of the application environment should take place after the page has been loaded. For this, a callback function, that takes care of the set up, should be registered with the page `onLoad` event. The name of the callback could be `loadEventHandler`. Inside that callback, the first call should be to a function `setUpEnvironment`. This function is contained in the `ProxyWidgetUtils.js` file and takes two parameters - a URL of the `tunnel.html` and a URL of the Transformer Widget. The URL of the The Transformer Widget must refer to the same domain as the server-side component. This is due to the Same Origin Policy - because Transformer Widget makes XMLHttpRequests to the server-side, there cannot be any restrictions to accessing this service. The `setUpEnvironment` function does all the common setting up of the OpenAjax Hub environment and assigns the initialized hub to the global variable `managedHub`. The `managedHub` can then be accessible throughout the application. The function also includes the Transformer Widget to the page and adds the widget to the hub.

After calling the `setUpEnvironment` function, it is possible to add custom widgets to the hub. These can be hidden or visible widgets. Adding new widgets should conform to the specification of OpenAjax Hub. An example of a simple application HTML page, that has followed the instructions above, has been given in the Example 3.1 on the next page.

3.1.2 Creating the Proxy Widget

When the application environment has been set up as required, it is possible to start adding the Proxy Widgets to the application. Generating and adding those widgets has been made easy by the inclusion of `ProxyWidgetUtils.js` file. This file defines a function `generateWidget`, that is used to generate a new Proxy Widget. This function assumes that the previously initialized hub can be referred to by the global variable `"managedHub"` - this is an assumption, which is met by default, if the initialization process has been followed as described previously, because the `ProxyWidgetUtils.js` file created this variable and the `setUpEnvironment` function initialized it. The `generateWidget` is called by passing three arguments - URL of the WSDL document, operation name and the URL of the Proxy Widget HTML page. The Proxy Widget main

Example 3.1 An example of minimal setup needed on the client-side HTML page

```
<html>
  <head>
    <script type="text/javascript"
            src="http://www.proxy.com/widgets/pagebus.js"></script>
    <script type="text/javascript"
            src="http://www.proxy.com/widgets/ProxyWidgetUtils.js"></script>
    <script type="text/javascript">
      var tunnel = "./tunnel.html";
      var transformerWidget = "http://www.proxy.com/TransformerWidget.html";
      function loadEventHandler() {
        setUpEnvironment(tunnel, transformerWidget);
        // set up additional widgets
      }
    </script>
  </head>
  <body onload="loadEventHandler()">
    <div id="mashupArea"></div>
  </body>
</html>
```

page must be served from the same location as the server-side proxy. Again, this is because of the Same Origin Policy restriction.

When the function `generateWidget` is called, it makes sure that there is no existing widget for the same WSDL operation description. It does this by checking the global variable `proxyWidgets`, which is a map, where all the generated Proxy Widgets are registered by using a unique widget URL as the key. If no existing widgets are found, then the new widget is registered in the initialized hub instance.

The functionality, that each Proxy Widget will provide, is controlled by two parameters, that are appended to the Proxy Widget URL. These are the location of the web service description and the name of the operation. An example of how the URL is constructed is given in Example 3.2 on the following page. When the hub initializes the Proxy Widget, then those parameters are extracted from the URL and stored in the Proxy Widget.

The root URL of the server-side component is also extracted from the URL of the widget. This is possible because the widget and the server-side component must be served from the same location. Locations of two

Example 3.2 An example of constructing the unique Proxy Widget URL in JavaScript code

```
var soapProxyWidgetURL = "http://proxy.com/widgets/SoapServiceWidget.html";
var wsdlDocumentURL = "http://myservice.com?wsdl";
var operationName = "foo";
var uniqueWidgetURL = soapProxyWidgetURL + "?wsdl=" + wsdlDocumentURL +
"&operation=" + operationName;
```

Example 3.3 A JavaScript example of generating URLs for services that return mappings and SMD documents

```
var soapProxyRootUrl = "http://proxy.com/";
var wsdlDocumentURL = "http://myservice.com?wsdl";
var operationName = "foo";
var mappingServiceURL = soapProxyRootURL + "mapping?wsdl=" + wsdlDocumen-
tURL + "&operation=" + operationName;
var smdServiceURL = soapProxyRootURL + "smd?wsdl=" + wsdlDocumentURL +
"&operation=" + operationName;
```

important services can be determined by using the root URL, URL of the WSDL document and the operation name. These are the locations for getting the mappings for the Proxy Widget and for getting the SMD document that is used by Dojo to generate the JavaScript service wrapper. Those locations are generated as can be seen in the Example 3.3.

If the Proxy Widget is successfully connected to the OpenAjax Hub, there are a few actions that need to be carried out. Firstly, the Proxy Widget publishes a message to the hub with topic `"ee.stacc.transformer.mapping.add.url"` with the URL of the mappings file as message content for registering the mappings at the Transformer Widget. This is a special topic that Transformer Widget listens, to add new mappings to its internal repository. When Transformer Widget receives this message, it initiates a XMLHttpRequest to load the mappings from the given URL. The service replies with a XML document, that contains all the mappings that Transformer Widget uses to interpret, construct and route messages between widgets.

The mappings document is divided into two frames, one for the input and one for the output message. The input message is the one, that the Proxy Widget receives from the OpenAjax Hub and that it eventually uses to construct the SOAP input message for a particular operation. The output message is the one, that the Proxy Widget publishes back to the hub, once it receives the results of the SOAP request from the server-side proxy service.

Each frame has its unique topic name, that is generated by using the algorithm as described in Example 3.9 on page 32. The topic name is used when publishing messages, that conform with the frame, to the hub. When a message is published with a certain topic name, it is passed on to the widget, that has subscribed to that topic.

Transformer Widget also checks for the JSON schema location of each frame in the mappings document. In mappings document, that is generated by the server-side mapping generator service, there is a JSON schema defined only for the input frame. This schema is loaded by the Transformer Widget and persisted internally for later use. Transformer Widget uses JSON schemas to generate the input message it publishes to the Proxy Widget. The schema is also used to check if the message is complete and ready to be published - each element that is marked as required in the schema, must have a value in the message, otherwise it is not published at all.

The Proxy Widget subscribes itself to receive all the messages that OpenAjax Hub publishes under the topic that was used in the mappings document input frame. When subscribing to that topic, the widget specifies a callback method, that is called when it receives a message with that topic. This callback function is used to pass the data in the received message on to the server-side proxy.

The above actions are summarized in a sequence diagram on Figure 3.1 on the next page.

3.1.3 Creating a Consumer for the Proxy Widget

When the Proxy Widget is successfully initialized and registered in the hub and in the Transformer Widget, it is possible to start using it to request information from that particular SOAP operation, that the widget was created for. In order to use the widget, there are two possibilities - either to publish data straight into the hub and use the correct topic or to rely on the Transformer Widget to route the information to the correct widget based on

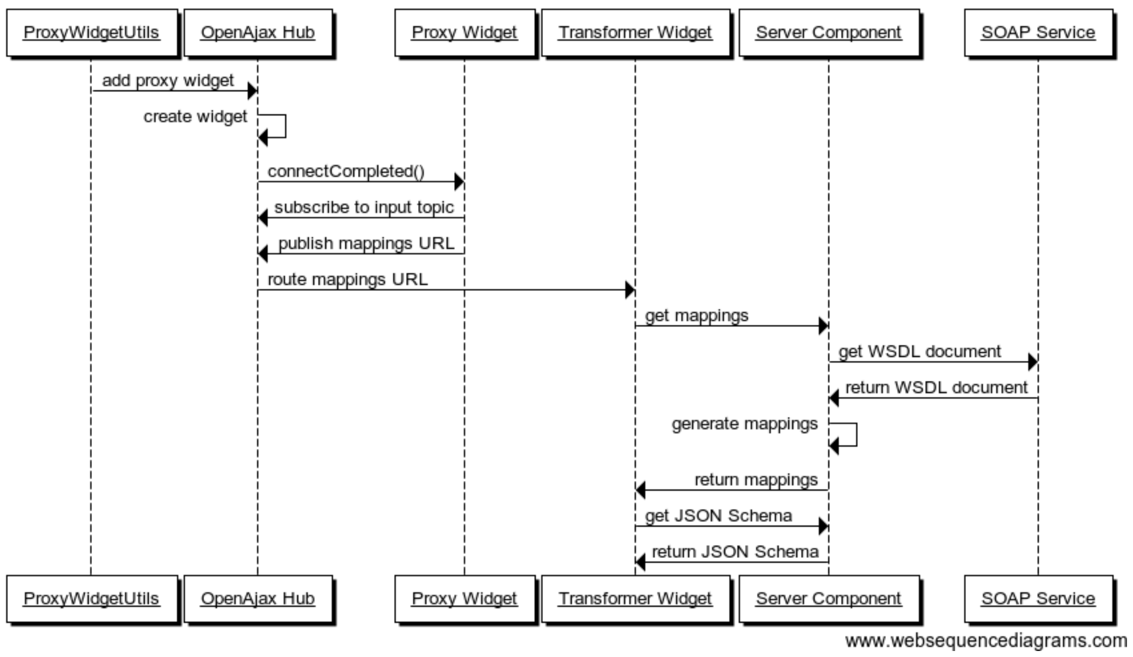


Figure 3.1: Sequence diagram of Proxy Widget creation

semantic metadata. It is encouraged to not use the first option, because this introduces a lot of possibilities to make errors. The reason is, that the user must know all the details and the exact structure for the input message. The user must also change the message it publishes, whenever the structure of the SOAP service changes. For example, if the operation changes names of its input parameters, user must also update corresponding element names in its application implementation.

The preferred way to use the Proxy Widget is through the OpenAjax Hub and Transformer Widget. For this, a consumer widget should be created and registered in the hub and in the Transformer Widget. To allow semantic integration between the consumer widget and the Proxy Widget, the consumer must know the semantic annotations that the service operation uses in its input and output messages. Using those annotations, the consumer of the Proxy Widget must create a mappings document that annotates the message, that the consumer publishes, with the same global semantic references, that the Proxy Widget input message uses. It should also create mappings to receive a response of the service operation. It is critical, that the exact same semantic vocabulary is used, otherwise the Transformer Widget is unable to transform the messages between the widgets.

The created mappings for the consumer widget must be registered in the Transformer Widget. There are several methods to do this. First one is similar to the way, that Proxy Widget registers the mappings - the widget publishes a message with a topic `"ee.stacc.transformer.mapping.add.url"` and URL of the mappings location as the data object. Transformer Widget then loads the mappings from specified URL. However, there is one downside, the URL must refer to location in the same domain as the Transformer Widget itself, otherwise it cannot access the data. Or in other words, the consumer widget must be bundled with the server-side component. In most cases this is impossible to do, unless the developer has full control of the server-side component. This violates with the idea, that the infrastructure can be used by many but without changes to any parts of the system.

Second way is to add mappings statically to the `mappings.xml` document - this document has to reside in the same path as the Transformer Widget itself (in other words, its relative path from Transformer Widget must be `./mappings.xml`). The main problem with this approach is again, that the user must have access to the actual server where the server-side component is deployed. This definitely should not be the case for most developers.

The third and preferred way is to add mappings by publishing raw XML to the OpenAjax Hub. The consumer widget should publish the raw XML data under the topic `"ee.stacc.transformer.mapping.add.raw"`. The Transformer Widget listens to this topic, parses the raw XML and extracts mappings information from it. The main advantage of this method is, that the mappings can be added dynamically and that information does not have to be in the same domain with Transformer Widget. Instead, the consumer widget could save this information in a XML file in its own domain, read it, when it has connected to the hub, and then publish it to be registered in the Transformer Widget.

When a user uses the third method, it cannot use the JSON schema in the mappings, because the Transformer Widget cannot access it unless the

Example 3.4 An example of mapping that uses embedded schema definition

```
<frames>
  <frame>
    <topic>example.topic</topic>
    <format>json</format>
    <schema_data>
      {"type":"object","properties":{"name":{"type":"string"}}}
    </schema_data>
    <mappings>
      <mapping>
        <global_ref>http://www.example.org/person/owl#Name</global_ref>
        <path>/name</path>
      </mapping>
    </mappings>
  </frame>
</frames>
```

URL points to same domain as Transformer Widget is served from. Because the idea is to enable separation of consumer widgets from the whole proxying infrastructure, the schema data should stay together with the consumer widget. To overcome the limitations of schema loading from provided URL, the Transformer Widget was extended so that it is also possible to embed the schema definition inside the mappings. The Transformer Widget looks for an element "schema_data" inside the "frame" element, to read the schema. An example of how to use embedded schema definition is given in Example 3.4.

When the consumer widget registers itself in the hub, it must subscribe itself to the topic that it uses for its input frame. This way the hub can route the messages to the consumer widget. Input frames are the ones, where the topic is not marked as `outgoing_only`. Input frames must always include the schema definition, because it is used to construct messages, while output frames are not required to have schema specified, as this is not used anywhere.

3.1.4 Consuming the Proxy Widget

When the consumer widget is fully initialized, it can start using the Proxy Widget to call the service it needs. To call the service, it must first construct a message to be published to the OpenAjax Hub. The message data must conform to the mapping description that goes with the output frame in the mappings of the consumer widget. The constructed message will then have to be published to the hub using the output frame topic name.

Because Transformer Widget listens to all topics, that are published to the hub, it will receive the consumer widget's output message. Transformer Widget will then find the correct mapping that goes with the received topic. Using the information from mappings, it is able to map the data in the message with correct semantic references. If all the data is semantically annotated, the Transformer Widget will start looking for input frames that have used the same semantic annotations for their data. Because Proxy Widget had registered its input frame in the hub and that input frame used the same semantic references, the Transformer Widget will start constructing the message that conforms to the Proxy Widget input frame. It uses the received data in suitable placeholders, to create the message. When the message is created, the Transformer Widget checks if it is ready to be published - this means that all the required fields must be filled. The Transformer Widget will then publish the data with the topic that the Proxy Widget subscribed to (the one that Proxy Widget used in its input frame).

OpenAjax Hub passes the message to the Proxy Widget where `onData` function is called. The data package, that Transformer Widget constructed, is passed as a parameter to the `onData` function. The function delegates to another function `onSoapServiceData` in `ProxyWidget.js` file. This function only then loads the SMD document from the server by using the SMD service URL, that the Proxy Widget had generated earlier. SMD is loaded by using `dojo.io.script.get` function which creates a Dojo specific Deferred object, which is used to register callbacks on different events. A `callService` callback function is registered on an event of successful retrieval of the SMD document. When SMD document is retrieved and the `callService` function is called, a Dojo service wrapper is created by using `dojox.rpc.Service`. The Dojo service wrapper will allow the calling of services, that are defined in the SMD, like any other JavaScript function. Because the SMD, that is generated in the server-side, includes only one service, there is only one wrapper function created. This is function allows calling of the proxy service on the server-side.

When the Dojo service wrapper for the proxy service is called, another

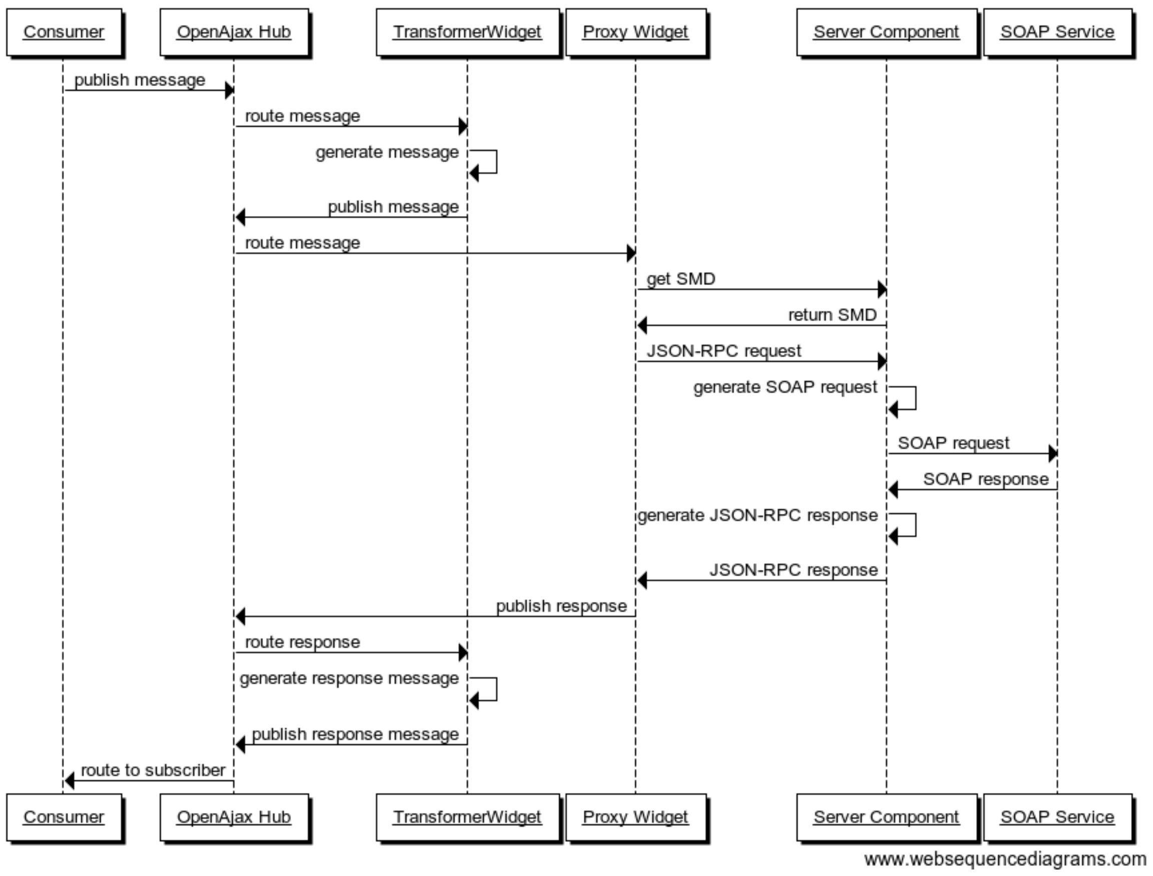


Figure 3.2: Sequence diagram of proxy widget consumption

Deferred object is created and a callback is registered, that on successful retrieval of response data, publishes the received data back to the hub. The data is published using the topic name, that matches the one used for outgoing frame in mappings document. From here on the responsibility goes to Transformer Widget, which reads the mapping for the received topic, finds the semantics for the incoming data, finds other frames that use data fields with the same semantics and creates new data packages, if possible. If any of the data packages are finished and ready to be published, they are published to the hub and the response data, that was generated by the server-side proxy, reaches the consumer widget.

The process of consuming the Proxy Widget is illustrated on Figure 3.2.

3.2 The Server-side

The server-side component provides important services to the client-side Proxy Widget and the Transformer Widget. Its main responsibilities are to create the mappings document for specified service operation, to generate the JSON schema definition that maps with the service operation input message structure, to generate the SMD document that is used to create the Dojo service wrapper and to provide a JSON-RPC service that enables to proxy the requests to the actual SOAP service.

3.2.1 Generating Mappings

The mappings document is required by the Transformer Widget to map the structure of data with the semantic global references that each data field represents. When the Proxy Widget is generated, a certain URL is sent to the Transformer Widget. This URL points to the service location that is used to generate mappings for that exact service operation that the Proxy Widget is generated for. Inside that URL, there are two parameters passed to the mappings service - the URL of the WSDL document and the name of the operation. Those parameters are read by the mappings service and are used to get the structure of the message that the SOAP service uses for the input and output operations.

The mappings service is a Spring MVC controller that internally uses an implementation of the `MappingGenerator` interface to generate the mappings. The `MappingGenerator` interface describes only one method - `getMapping` - as can be seen in Example 3.5 on the following page. The `getMapping` method is passed three parameters. Two of them - `wsdlUri` and `operation` - are request parameters passed to the `MappingController`. Third is a URL to the service that generates JSON schemas. The JSON schema URL is generated inside the controller, using an algorithm like in Example 3.6 on the next page.

Most of the classes and interfaces that are related to mappings generation, are shown in Figure 3.3 on the following page.

Example 3.5 MappingGenerator interface

```
public interface MappingGenerator {
  String getMapping(String wsdlUri,
                    String operation,
                    String jsonSchemaUrl) throws Exception;
}
```

Example 3.6 Generating the URL for JSON Schema service

```
String baseUrl = "http://proxy.com/";
String wsdlDocumentURL = "http://myservice.com?wsdl";
String operation = "foo";
String jsonSchemaServiceURL = soapProxyRootURL + "mapping?wsdl="
                              + wsdlDocumentURL + "&operation="
                              + operation + "&message=input";
```

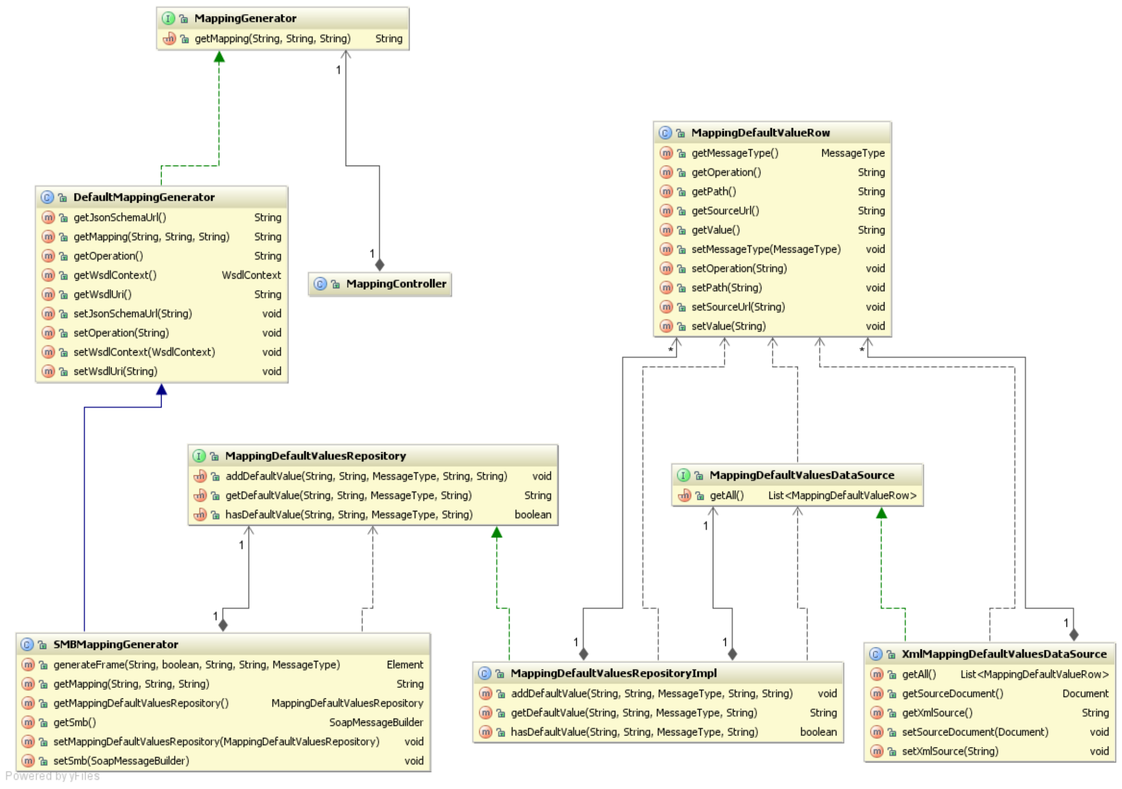

MappingGenerator
getMapping(String, String, String) String
DefaultMappingGenerator
getJsonSchemaUrl() String
getMapping(String, String, String) String
getOperation() String
getWsdlContext() WsdlContext
getWsdlUri() String
setJsonSchemaUrl(String) void
setOperation(String) void
setWsdlContext(WsdlContext) void
setWsdlUri(String) void
MappingController
MappingDefaultValueRow
getMessageType() MessageType
getOperation() String
getPath() String
getSourceUrl() String
getValue() String
setMessageType(MessageType) void
setOperation(String) void
setPath(String) void
setSourceUrl(String) void
setValue(String) void
MappingDefaultValuesRepository
addDefaultValue(String, String, MessageType, String, String) void
getDefaultValue(String, String, MessageType, String) String
hasDefaultValue(String, String, MessageType, String) boolean
MappingDefaultValuesDataSource
getAll() List<MappingDefaultValueRow>
SMBMappingGenerator
generateFrame(String, boolean, String, String, MessageType) Element
getMapping(String, String, String) String
getMappingDefaultValuesRepository() MappingDefaultValuesRepository
getSmb() SoapMessageBuilder
setMappingDefaultValuesRepository(MappingDefaultValuesRepository) void
setSmb(SoapMessageBuilder) void
MappingDefaultValuesRepositoryImpl
addDefaultValue(String, String, MessageType, String, String) void
getDefaultValue(String, String, MessageType, String) String
hasDefaultValue(String, String, MessageType, String) boolean
XmlMappingDefaultValuesDataSource
getAll() List<MappingDefaultValueRow>
getSourceDocument() Document
getXmlSource() String
setSourceDocument(Document) void
setXmlSource(String) void
1
*
Powered by yFiles

Figure 3.3: Classes and interfaces related to mapping generation.

Example 3.7 XML with attributes

```
<elem foo="bar">my value</elem>
```

Example 3.8 JSON translated from XML with attributes

```
{"elem":{"_attr_foo":"bar","_value_":"my value"}}
```

3.2.1.1 Transforming SOAP Message Structure to JSON

The mappings document actually represents the structure of the JSON data that is exchanged between the Proxy Widget and the server-side proxy service. This means that the structure of SOAP request has to be transformed to the structure of JSON-RPC request. Unfortunately, transforming XML structure to JSON is not that straightforward. One of the main problems is that there is no equal counterpart for XML attributes in JSON. The only possible solution is to transform XML attributes to JSON elements. In this particular solution, the XML attributes were translated so, that each attribute name is added a prefix `"_attr_"` and added as a child element. Translating XML attributes to JSON elements creates another problem - the value of the element, whose attributes were translated, has to be given to another special JSON element. This is because in JSON it is not possible to have mixed content value like in XML. Therefore the actual value of the element is given through a special element `"_value_"` which is added as a child to the original element. An example of dealing with attributes can be seen in generating XML as in Example 3.7 to JSON as in Example 3.8.

Another problem is with XML namespaces. This problem is solved by simply dropping the namespaces when transforming XML element and attribute names to JSON element names. This might create problems in some rare occasions, but the risk should be relatively small.

Yet another XML construction that cannot be simply translated to JSON is when a parent element has more than one child element with the same name. In that case the JSON representation will make the child element an array and add all the values inside the array.

When SOAP message is transformed to JSON, then also the body and header parts of the message must be added to JSON and mappings. This is because the header part can include important parameters like license key, that must also be possible to specify when using the proxy service.

3.2.1.2 Implementation of the MappingGenerator

The `MappingGenerator` is a Spring bean which is injected to the `MappingController` with dependency injection. This makes it easy to change the implementation of the actual `MappingGenerator` interface. The current implementation `SMBMappingGenerator` heavily uses SoapUI API to generate the mappings. The `SMB` prefix stands for `SoapMessageBuilder`, which is a class in SoapUI library. The use of SoapUI makes it easy to generate request templates for the SOAP input and output requests. The request templates can then be analyzed to get the actual structure of the message. Some parts of the SoapUI - in particular the `SoapMessageBuilder` class and the `SampleXmlUtil` class - were extended to enable the inclusion of semantic annotations to the elements of SOAP message body and header. Also, there were some changes due to the fact, that XML attributes have to be translated to JSON elements.

3.2.1.3 Generating Topic Names

For input and output frames of each operation, a unique topic name must be generated. Topic name is used by Transformer Widget to find correct mapping for information that is passed from publishers to subscribers. Topic names should usually be in the reverse domain name format. An example of how topic names are generated is given in Example 3.9 on the next page.

3.2.1.4 Setting Default Values in Mappings

In some cases it might be needed to set default values for some fields in the SOAP request. For instance for case where the value cannot be expected to be provided by an application or is a constant. In this solution this need is met by a special XML document, where all the default values can be configured. Each field is identified by four attributes: the URL of the WSDL document, the name of the operation, the path to the field in input or output message and the type of the message (input or output). It is important to note that

Example 3.9 Generating the topic name for input and output frames in the mappings document

```
String wsdlDocumentURL = "http://myservice.com?wsdl";
String operationName = "foo";
String commonTopicPart = "ee.stacc.soapwidgetgenerator.";
commonTopicPart += wsdlDocumentURL.replaceAll("\\W", "-");
commonTopicPart += "." + operationName;
String inputTopic = commonTopicPart + ".input";
String outputTopic = commonTopicPart + ".output";
// inputTopic:  "ee.stacc.soapwidgetgenerator.http---myservice.com-
wsdl.foo.input"
// outputTopic:  "ee.stacc.soapwidgetgenerator.http---myservice.com-
wsdl.foo.output"
```

Example 3.10 XML for specifying default values in SOAP requests

```
<?xml version="1.0" encoding="UTF-8"?>
<defaults xmlns="http://www.cs.ut.ee/schema/soapproxywidget/mappingdefaults"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.cs.ut.ee/schema/soapproxywidget/mappingdefaults
mapping-defaults.xsd">
  <value sourceUrl="http://myservice.com?wsdl" operation="foo"
path="/my/path" messageType="input">myDefaultValue</value>
</defaults>
```

the path must represent the structure of a JSON message. So one must keep in mind the rules that apply when transforming a message structure from XML to JSON. An example of of this XML is given in Example 3.10.

The default values are added to the mappings for each field that the apply. Because the mappings are used by the Transformer Widget, the default values are also set in there. The default value is used in message creation when no data has been aggregated which would correspond to the same metamodel element identifier. There are two reasons to specify the default values in the mappings document. Firstly, the Transformer Widget already has support for dealing with the default values. Secondly, if a required field is left empty while constructing the outgoing message, the message will not be published at all. So it would not be reliable to add default values on the server side, before creating the actual SOAP request. An example of mapping with a default value is given in Example 3.11 on the next page.

The location of the XML document with default values can be configured on the server side. The configuration is in the following file: `war/WEB-`

Example 3.11 An example of mapping with a default value

```
<mapping>
  <global_ref>http://www.example.org/owl#Example</global_ref>
  <path>/my/path</path>
  <default>myDefaultValue</default>
</mapping>
```

`INF/application-config.properties` and the configuration property is `mappingdefaults.xmldatasource.url`. The XML file could be stored in any location accessible to the server-side component.

The classes that are related to default values in mappings can also be seen in Figure 3.3 on page 29. From that figure, it can be seen, that the SMBMappingGenerator uses an implementation of `MappingDefaultValuesRepository`. This where all the default values are loaded to from an implementation of `MappingDefaultValuesDataSource`. The current data source implementation only supports XML.

3.2.2 Generating JSON Schemas

When the Transformer Widget has received a mappings document, it will look for a `schema` tag for each frame that is used for an input message. The `schema` tag specifies the URL, where the schema definition document can be located. If a schema tag contains a valid URL, this schema definition is loaded by Transformer Widget and saved for further usage. The schema locations, that are used for SOAP Proxy Widget's input messages are generated by the server-side component. The URL, that is used for schema retrieval, is generated by mapping generator as can be seen in Example 3.6 on page 29. The URL shows that the JSON schema generation service will get the name of the operation and the location of the WSDL document as parameters.

The `JsonSchemaController` uses an implementation of the `JsonSchemaGenerator` interface to get the correct schemas. The classes that are related to JSON schema generation can be seen from Figure 3.4 on the following page. The `DefaultJsonSchemaGenerator` uses the custom extension of SoapUI `SoapMessageBuilder` to create template message for the actual SOAP request. This way it is not necessary to parse and analyze the XML schema definition

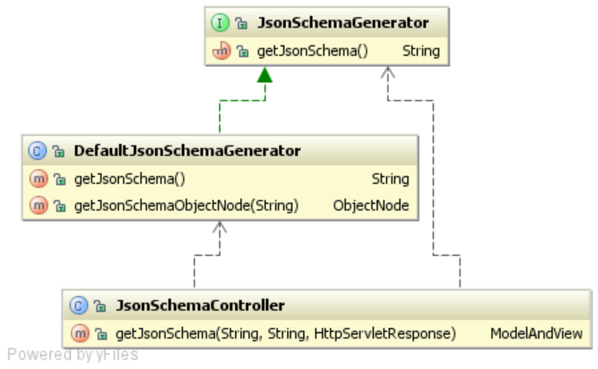

JsonSchemaGenerator
getJsonSchema() String
DefaultJsonSchemaGenerator
getJsonSchema() String
getJsonSchemaObjectNode(String) ObjectNode
JsonSchemaController
getJsonSchema(String, String, HttpServletResponse) ModelAndView
Powered by yFiles

Figure 3.4: Class diagram of JSON schema generation related classes

anymore, as this is already done by SoapUI. The message template is used as a source for the JSON schema generation. Because the schema must represent the same structure that is used in the mappings document, the rules, that were used in generating SOAP message structure to JSON, should be taken into account when generating the corresponding schema. Those rules are described in section 3.2.1.1 on page 30. It must be considered, that the JSON schema generation algorithm currently lacks support for attributes. This means that input messages with attributes cannot be used.

The JSON schema definition will set each element that is not an array or an object as being type of string. Although JSON schema has support for other types as well, this is not needed, because no validation is done anywhere. It is just important to know what the message structure looks like. Besides the message structure, the JSON schema will also specify if a field is required or not. This information is used by Transformer Widget, that makes sure that each required field is present before sending out any constructed messages.

In Examples 3.12 on the following page and 3.13 it can be seen how a SOAP request message template translates to JSON schema.

Example 3.12 An example of SOAP input message template produced by SoapUI.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:ws="http://ws.soatrader.com/"
                  xmlns:eer="http://eer.soatrader.com/">
<soapenv:Header>
  <ws:SOATraderLicense>?</ws:SOATraderLicense>
</soapenv:Header>
<soapenv:Body>
  <eer:getListOfAnnualReports>
    <!--Optional:-->
    <registryCode>?</registryCode>
    <languageId >?</languageId>
  </eer:getListOfAnnualReports>
</soapenv:Body>
</soapenv:Envelope>
```

Example 3.13 An example of JSON schema definition that is generated from input message template given in Example 3.12

```
{"type":"object","properties":{
  "Header":{
    "required":true,
    "type":"object",
    "properties":{
      "SOATraderLicense":{"required":true,"type":"string"}}},
  "Body":{
    "required":true,
    "type":"object",
    "properties":{
      "getListOfAnnualReports":{
        "required":true,
        "type":"object",
        "properties":{
          "registryCode":{"type":"string"},
          "languageId":{"required":true,"type":"string"}
}}}}}}
```

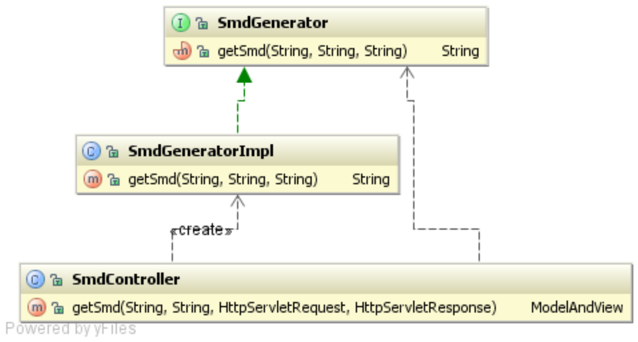

SmdGenerator
getSmd(String, String, String) String
SmdGeneratorImpl
getSmd(String, String, String) String
«create»
SmdController
getSmd(String, String, HttpServletRequest, HttpServletResponse) ModelAndView
Powered by yFiles

Figure 3.5: Class diagram of classes related to SMD generation

3.2.3 Generating SMD Documents

An SMD document is used by the Proxy Widget to automatically generate JavaScript wrappers for a JSON-RPC proxy service that the server-side component provides for SOAP operations. Dojo Toolkit is the actual consumer of the SMD document. The URL for SMD generation service is constructed the way that can be seen in Example 3.3 on page 21. So once again the server-side component gets the URL of the WSDL document and the name of the operation as parameters. This time, however, the WSDL document is not parsed anymore. Those parameters are simply required to construct the URL for the JSON-RPC proxy service for a certain SOAP service operation. This URL is added to the SMD document to refer to the target service. The SMD service also supports JSONP, therefore it is possible to specify the callback function name in a request parameter `callback`. This way the proxying abilities of the server-side component can be used by anyone even without using the Proxy Widget itself.

The `SmdController` uses an implementation of `SmdGenerator`, as can be viewed from Figure 3.5. The generated SMD document is a very simple one. An example of an SMD document that is generated by this service is given in Example 3.14 on the next page.

Example 3.14 An example of a SMD document.

```
{ transport:"JSONP",
  envelope:"JSON-RPC-2.0",
  SMDVersion:"2.0",
  services:{
    getListOfAnnualReports:{
      target:
        "http://localhost:8080/proxy?
          wsdl=http://localhost/EstonianBusinessRegistryService_v2.wsdl
          &operation=getListOfAnnualReports"
    }
  }
}
```

3.2.4 Proxying Requests to the SOAP Service

The JSON-RPC requests, that are created by the Proxy Widget on the client-side, are translated to SOAP requests and forwarded to the actual SOAP service endpoint by the server-side proxying service. Internally the Proxy Widget uses the Dojo service wrapper that was generated earlier with the help of an SMD document. As can be seen from Example 3.14, the SMD document contains URL to the target service. Because the transport method is specified as "JSONP", Dojo will use `script` tag to initiate the request. This also means that Dojo passes the JSON-RPC request data as a URL encoded parameter key. An example of a valid query string, that is generated with SMD in Example 3.14 and where the input data matches the schema that was given in Example 3.13 can be seen in Example 3.15.

From Example 3.15 it can be seen, that the JSON-RPC request contains four parameters: request id, method name, parameters and the version of JSON-RPC that is used. The `"params"` parameter is an array that contains all the parameters that are passed to the Dojo service wrapper. Proxy Widget only passes one parameter - the JSON formatted message that was constructed by the Transformer Widget - therefore the array contains only that message object. The request id is also used in response message for referencing purposes. The `"method"` parameter from JSON-RPC data is not actually used by server-side, because the operation name is read from `"operation"` request parameter.

Example 3.15 An example of valid JSON-RPC request that is sent using JSONP transport.

```
http://localhost:8080/proxy
  ?wsdl=http://localhost/EstonianBusinessRegistryService_v2.wsdl
  &operation=getListOfAnnualReports
  &{"id":1305550122389,
    "method":"getListOfAnnualReports",
    "params":[
      { "Header":{"SOATraderLicense":"licenseKey123"},
        "Body":{
          "getListOfAnnualReports":{
            "registryCode":"10283074",
            "languageId":"1"
          }
        }
      }],
    "jsonrpc":"2.0"
  }
  &callback=dojo.io.script.jsonp_dojoIoScript2._jsonpCallback
```

The URL of the WSDL document, the name of the operation and the JSON formatted message data are used to create the actual SOAP request. The `ProxyController` passes these parameters on to the `convert` method of the implementation of the `JsonRpc2SoapConverter` interface. From here on, once again the request template for the actual SOAP service endpoint is created with the help of SoapUI. Then the actual message parameters are read from the JSON message and injected to the corresponding field in the SOAP request template. This can be done quite easily because the JSON message matches almost exactly the SOAP request template. Only exception is, that there are no namespaces used in JSON message. In case of arrays, the fields in SOAP request template are duplicated the exact number of times as there are elements in the array. After that the value injection takes place. In the end, all of the values, that have a matching counterpart in SOAP message template, are injected. The classes that are related to proxying service are shown on a class diagram on Figure 3.6 on the following page.

The resulting SOAP message is then sent to the actual endpoint. The response message is translated from XML to JSON using the `Xml2JsonConverter` class where the same rules, that were described earlier in Section 3.2.1.1 on page 30, are followed. Using the generated response data in JSON, a valid

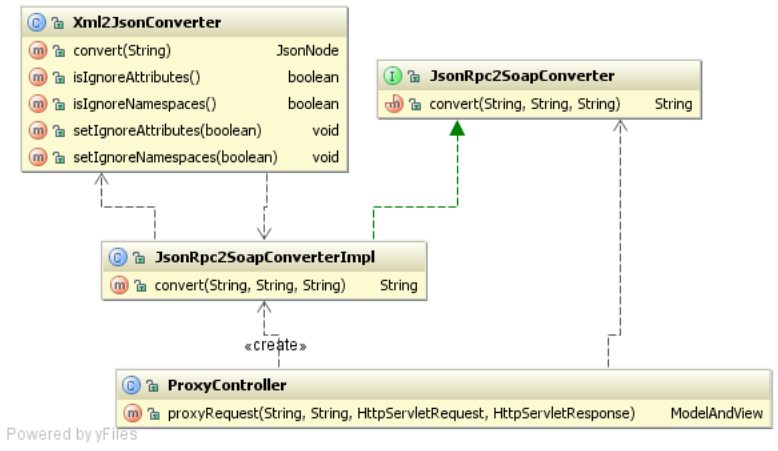

Xml2JsonConverter
convert(String) JsonNode
isIgnoreAttributes() boolean
isIgnoreNamespaces() boolean
setIgnoreAttributes(boolean) void
setIgnoreNamespaces(boolean) void
JsonRpc2SoapConverter
convert(String, String, String) String
JsonRpc2SoapConverterImpl
convert(String, String, String) String
«create»
ProxyController
proxyRequest(String, String, HttpServletRequest, HttpServletResponse) ModelAndView
Powered by yFiles

Figure 3.6: Class diagram of classes related to proxying service

JSON-RPC response is created and sent back to the Proxy Widget. An example of such response is given in Example 3.16 on the following page.

The response is wrapped inside a callback function name, that was passed as a request parameter. This allows this function to be called right after the client-side gets the response. The message data, that was transformed from XML to JSON is given as a value to parameter ”result”. The JSON-RPC response also contains the request id, the version number of JSON-RPC protocol and an `"error"` parameter. The `"error"` parameter can be used to pass the client side information about any errors that occurred. If there are no errors, the parameter must be `null`.

Example 3.16 An example of JSON-RPC response.

```
callbackFunction({
  "result":{"Body":{"getListOfAnnualReportsResponse":{
            "ListOfAnnualReports":{"report":[
              { "reportName":"Name of a report",
                "reportYear":"2010",
                "periodStartDate":"2010-01-01",
                "periodEndDate":"2010-12-31"},
              { "reportName":"Name of a report",
                "reportYear":"2009",
                "periodStartDate":"2009-01-01",
                "periodEndDate":"2009-12-31"}
  ]}}}},
  "id":"1305550122389",
  "error":null,
  "jsonrpc":"2.0"})
```

Chapter 4

Proof of Concept

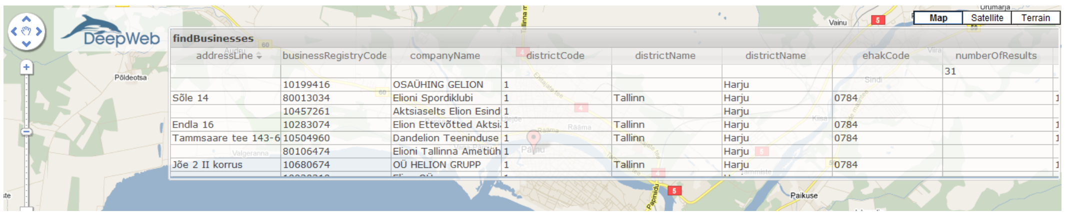
To validate that the provided solution works in a real-life scenario, a proof of concept application was built. This application uses the solution to query three operations from Estonian Business Registry SOAP service. The operations, that are involved, are `findBusinesses`, `getListOfAnnualReports` and `getAnnualReportData`. Besides the three Proxy Widgets, that are generated for each of the SOAP operations, there are three visual widgets to show the output of the SOAP operations. Let them be named by adding word "Visual" as the suffix to each name of the operation, that the widget is meant to visualize. The demo application also additionally uses Google Maps widget.

The application scenario is as follows. When the application page loads, all the required set up for the environment is done as described in section 3.1.1 on page 17. When the page has finished loading, all the widgets are initialized and added to the OpenAjax Hub. The three Proxy Widgets are added with the help of `generateWidget` function inside the `ProxyWidgetUtils.js` file, while the adding of other widgets has to be done manually. When the Proxy Widgets are added, they all register themselves in the Transformer Widget as well.

After all the widgets are created, the first query is made. The first query is sent to the Proxy Widget of operation `findBusinesses`. The parameter, that is used, is "Elion". The JSON message, that is published looks like in Example 4.1 on the following page.

Example 4.1 JSON data, that is published to `findBusinesses` Proxy Widget.

```
{
  "Body":  {
    "findBusinesses":  {
      "registryCode":"Elion",
      "languageId":"1" }
  },
  "Header":  {
    "SOATraderLicense":"soaTraderLicense123"
  }
}
```



findBusinesses

addressLine	businessRegistryCode	companyName	districtCode	districtName	districtName	ehakCode	numberOfResults	
							31	
	10199416	OSAÜHING GELION	1		Harju			
Sõle 14	80013034	Elioni Spordiklubi	1	Tallinn	Harju	0784		1
	10457261	Aktsiaselts Elion Esind	1		Harju			
Endla 16	10283074	Elion Ettevõtted Aktsi	1	Tallinn	Harju	0784		1
Tammsaare tee 143-6	10504960	Dandelion Teeninduse	1	Tallinn	Harju	0784		1
	80106474	Elioni Tallinna Ametiüh	1		Harju			
Jõe 2 II korrus	10680674	OÜ HELION GRUPP	1	Tallinn	Harju	0784		1

Figure 4.1: List of results from `findBusinesses` operation with ”Elion” as input

When the Proxy Widget has gone through all the steps, as described in Figure 3.2 on page 27, it has published the results back to the hub. The list contains all the businesses that have a word ”Elion” anywhere in their name. The list also contains other information about the businesses. The `findBusinessesVisual` widget is then made visible and the result set is shown as a list. The JSON message, that the `findBusinesses` publishes as a result, looks like in Example 4.2 on the following page. The situation that appears to the user, is shown on a screen-shot on Figure 4.1.

The user can then select a business from the list, that was made visible. When the user selects ”Elion Ettevõtted Aktsiaselts”, the next query is initiated. This time a Proxy Widget for the `getListOfAnnualReports` SOAP operation is used. This operation takes the registry code as the main input parameter. The message that is published to the hub, looks like in Example 4.3 on the following page.

The results include all the available annual reports for that particular company. The result, that is published to the hub by `getListOfAnnualReports` Proxy Widget, looks like in Example X. When the results are retrieved on the client-side, the `getListOfAnnualReportsVisual` widget is made visible. The

Example 4.2 JSON data that `findBusiness` publishes as a result.

```
{
  "Body":  {
    "findBusinessesResponse":  {
      "ListOfBusinesses":{
        "recordCount":"31",
        "business":[
        // ...other businesses
        {"businessName":"Elion Ettevõtted Aktsialselts",
         "businessRegistryCode":"10283074",
         "statusCode":"R",
         "statusDescription":"Entered into the register",
         "registryDistrictCode":"1",
         "registryDistrictName":"Tallinn",
         "postalCode":"15033",
         "districtName:"Harju",
         "streetField":"Endla 16",
         "ehakCode":"0784",
         "registrationDate":"1997-10-09T00:00:00.093+03:00",
         "registrationDateInDistrict":"1997-10-09T00:00:00.093+03:00"
        },
        // other businesses...
        ]
      }
    }
  },
  "Header":  {
    "SOATraderUsageStatistics":{
      "HitsMade":"1", "HitsLeft":"9"
    }
  }
}
```

Example 4.3 JSON data that is published to `getListOfAnnualReports` Proxy Widget as input

```
{
  "Body":  {
    "getListOfAnnualReports":  {
      "registryCode":"10283074",
      "languageId":"1" }
  },
  "Header":  {
    "SOATraderLicense":"soaTraderLicense123"
  }
}
```

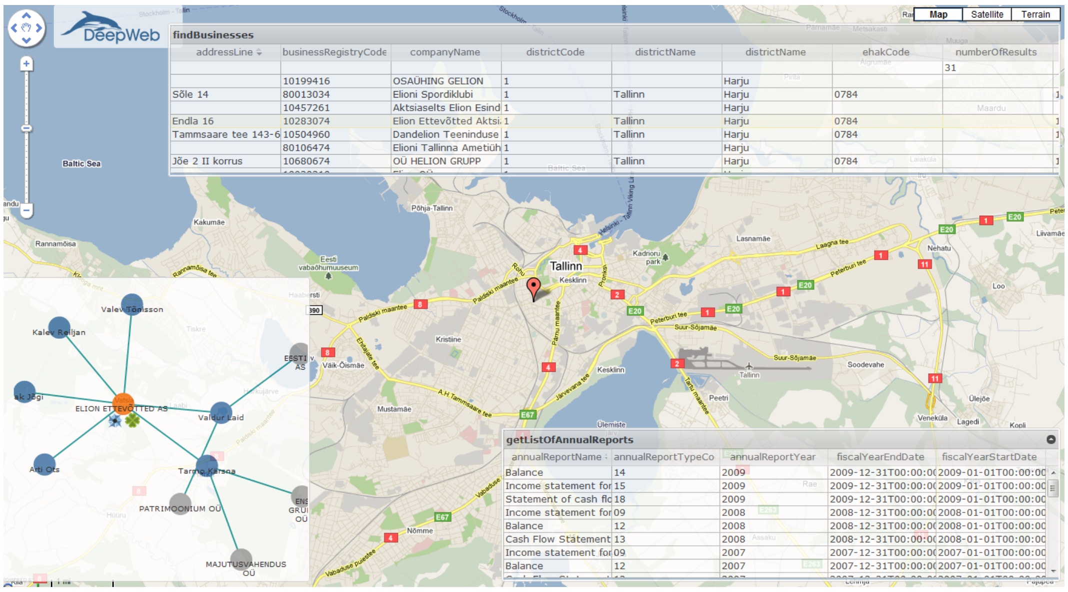
Example 4.4 JSON data that is published as result by `getListOfAnnualReports` Proxy Widget

```
{
  "Body":  {
    "getListOfAnnualReportsResponse":  {
      "ListOfAnnualReports":{
        "report":[
        {"reportTypeCode":"14",
         "reportName":"Balance",
         "reportYear":"2009",
         "periodStartDate":"2009-01-01T00:00:00.339+02:00",
         "periodEndDate":"2009-12-31T00:00:00.339+02:00",
        },
        // other reports...
        ]
      }
    }
  },
  "Header":  {
    "SOATraderUsageStatistics":{
      "HitsMade":"2", "HitsLeft":"8"
    }
  }
}
```

resulting screen-shot can be seen on Figure 4.2 on the next page. In addition to showing the list of annual reports, the address of the selected company is placed to the map. Also, another widget is made visible - the widget, that shows the graph of people and companies that are related to the selected company.

The rows in the table of `getListOfAnnualReportsVisual` are also possible to click. When the user wants to see the balance report from year 2009, she clicks the first row. This triggers the query of operation `getAnnualReportData`. The message, that is published to this operation Proxy Widget looks like in Example X.

The resulting JSON data, that the Proxy Widget of `getAnnualReportData` publishes, looks like in Example 4.6 on page 46. This time the resulting data is not shown as a table. Instead, a more visually appealing type of widget is used, where the balance rows are shown as a sector diagram. The screen-shot of this is given in Figure 4.3 on page 47. As it can be seen, the diagram has



findBusinesses

addressLine	businessRegistryCode	companyName	districtCode	districtName	districtName	ehakCode	numberOfResults	
							31	
	10199416	OSAÜHING GELION	1		Harju			
Sõle 14	80013034	Elioni Spordiklubi	1	Tallinn	Harju	0784		1
	10457261	Aktsiaselts Elion Esind	1		Harju			
Endla 16	10283074	Elion Ettevõtted Aktsi	1	Tallinn	Harju	0784		1
Tammsaare tee 143-6	10504960	Dandelion Teeninduse	1	Tallinn	Harju	0784		1
	80106474	Elioni Tallinna Ametiüh	1		Harju			
Jõe 2 II korrus	10680674	OÜ HELION GRUPP	1	Tallinn	Harju	0784		1

getListOfAnnualReports

annualReportName	annualReportTypeCo	annualReportYear	fiscalYearEndDate	fiscalYearStartDate
Balance	14	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Income statement for	15	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Statement of cash flo	18	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Income statement for	09	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Balance	12	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Cash Flow Statement	13	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Income statement for	09	2007	2007-12-31T00:00:0(	2007-01-01T00:00:00
Balance	12	2007	2007-12-31T00:00:0(	2007-01-01T00:00:00

Figure 4.2: Screen-shot after the `getListOfAnnualReports` operation has been queried.

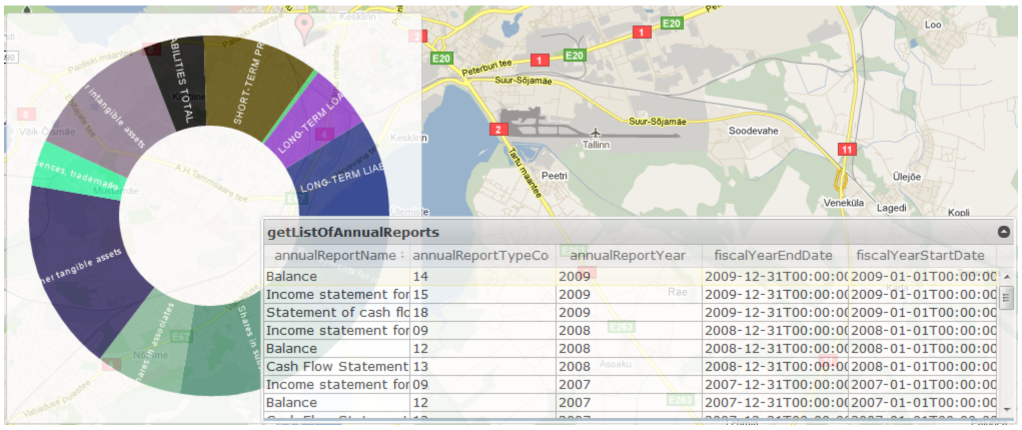
Example 4.5 JSON data, that is sent to Proxy Widget of operation `getAnnualReportData`

```
{
  "Body":  {
    "getAnnualReportData":  {
      "registryCode":"10283074",
      "reportType":"14",
      "year":"2009",
      "languageId":"1"
    }
  },
  "Header":  {
    "SOATraderLicense":"soaTraderLicense123"
  }
}
```

Example 4.6 JSON data, that `getAnnualReportData` publishes as a result

```
{
  "Body":  {
    "getAnnualReportDataResponse":  {
      "ListOfAnnualReportData":{
      "report":[
      {"reportDesc":{
        "reportTypeCode":"14",
        "reportName":"Balance",
        "reportYear":"2009",
        "periodStartDate":"2009-01-01T00:00:00.673+02:00",
        "periodEndDate":"2009-12-31T00:00:00.673+02:00"},
      "reportRow":[
        {"rowNumber":"10","rowName":"CASH AND BANK",
        "column":[
          {"code":"A1",
           "name":"Current financial year",
           "value":"2.091E7"},
          {"code":"A2",
           "name":"Previous financial year",
           "value":"1.06798E8"}
        ]}
      ]},
      // ...  other report rows
      ]}
    }
  },
  "Header":  {
    "SOATraderUsageStatistics":{
      "HitsMade":"3", "HitsLeft":"7"
    }
  }
}
```

sectors, that represent the rows in the balance report.



getListOfAnnualReports

annualReportName	annualReportTypeCo	annualReportYear	fiscalYearEndDate	fiscalYearStartDate
Balance	14	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Income statement for	15	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Statement of cash flo	18	2009	2009-12-31T00:00:0(	2009-01-01T00:00:00
Income statement for	09	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Balance	12	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Cash Flow Statement	13	2008	2008-12-31T00:00:0(	2008-01-01T00:00:00
Income statement for	09	2007	2007-12-31T00:00:0(	2007-01-01T00:00:00
Balance	12	2007	2007-12-31T00:00:0(	2007-01-01T00:00:00

Figure 4.3: A screen-shot after showing results from operation `getAnnualReportData`.

Chapter 5

Related Work

In order to find what other approaches have been proposed to resolve similar problems, one has to look in the fields of mashups, service composition at the presentation layer and automatic front-end generation for web services.

5.1 Service Composition with Mappings

The idea of using special mappings to interconnect services inside client browser has been presented in [23]. In this article, a special Web Mashup Scripting Language that should ease the writing of mashups, was introduced. The end-user accomplishes this by writing a web page that combines HTML, metadata in the form of mapping relations, and small piece of code, or script. This idea is similar to the one presented in this thesis but there are also differences. In current thesis semantic annotations are used to map fields in different data structures and also mapping generation is done automatically. Another dissimilarity is that in the referred article, there were no hints on using services from other domains, that the presented solution supports.

5.2 Graphical Tools for Mashup Generation

One of the main targets for many developers of mashup related tools or systems is to provide end users with graphical interface, that would dramatically ease the composition and consumption of web services and visual widgets. There are numerous tools already available that target the average web user allow visual composition.

A programming tool called Marmite [29] lets end-users create so-called mashups that re-purpose and combine existing web content and services. The main idea of Marmite is to allow using data-flow-like view to construct the transformation and combination of data and data sources using different operators. It also provides the possibility to look the result of each data-flow step in a spreadsheet view. The tool has no use of semantics, if two services have semantically the same information but is labeled differently, then the tool cannot correlate the information from one to the other. Marmite also relies on the owners of web services or other programmers to create the operators that the tool could use. The solution in this thesis differs in many parts - the targeted user segment is not an end-user, semantic annotations are used to integrate services, no additional involvement of the original service provider is required and there is no graphical support to create the mashups.

The Marmite tool was developed as a Firefox plug-in. There are also many other tools that follow the same pattern because browser is usually a very natural environment to construct the mashups. For example the Intel MashMaker [18] is also implemented as a Firefox plug-in.

The MashMaker works by augmenting live data with user specified formulae. Like a spreadsheet, MashMaker allows users to mix computed values with their data, including editing "live" (i.e., continuously-updated) data assembled through the web and/or user queries.

The Yahoo Pipes [7] has been given credit for a very usable user interface. A study of usability of mashup tools in 2009 [15] says that Yahoo Pipes provides the most interactive, intuitive and user friendly data aggregator and manipulator available at that time. They also describe the Pipes as a tool that has a large community library of data mashups to select and learn from and it is also possible to clone a mashup to suit your own needs.

There are some previously quite popular mashup tools whose development has been discontinued. For example Microsoft Popfly or Google Mashup Editor were both shut down in 2009. One might wonder what could have been the cause for their early closure. Possible reasons could be that the community was too immature or that those technology giants could not find feasible business case to be built around those tools.

5.3 Categorizing Mashup Tools and Environments

To categorize and compare mashup tools and environments, certain attributes are suggested in an article [30] that help to point out the differences. The same article also describes some of the existing mashup tools but as the research was done some time ago, these description might not be relevant anymore. Regardless of that, the comparison model is still useful and enables to position the developed solution in context of other tools and environments.

1. Manual or tool assisted development - Developing mashups with the help of OpenAjax Hub, Transformer Widget and SOAP Proxy is somewhat simplified but still a manual not tool assisted process.

2. Component model - Basically all component types are supported (data, application logic and user interface); existing components provide access via APIs and with the help of mappings in XML format; and end user can freely add new components to the application.

3. Composition model - The output type is an UI; the orchestration style is event-based (publish-subscribing handled by OpenAjax Hub); data passing style is using data-flow approach; compositions are instance-based - the composition is instantiated upon the opening of an application web page; exception and transaction handling is not supported.

4. Development environment - Basic text-based development tools can be used, there is no special development tool provided, therefore the target user is a person with JavaScript programming skills; no special system requirements are needed to run the mashup application - a typical Web browser is sufficient.

5. Run-time environment - The mashup can be deployed in any Web server; the run-time location is the client side; there are no additional requirements for the browser to run the mashup; the scalability depends mainly on the amount of data sources that create traffic with the SOAP Proxy server side component.

5.4 The Future of Mashup Development

Taivalsaari and Mikkonen[25] argue that the Web application developments should occur in a collaborative, social fashion. The article finds that security and modularity are the two areas that currently make the Web an anti-social environment for developers. Some of the problems - secure interaction between content from different sites and overcoming the Same Origin Policy limitations - are very well related to SOAP Proxy solution. The Same Origin Policy is overcome with the help of a server-side proxy, that also does not need to be configured for any additional services. The secure interaction is resolved by using OpenAjax Hub. The modularity issue also finds a solution in the current work, because SOAP Proxy allows the usage of SOAP services, that have well defined interfaces (WSDL), inside the OpenAjax Hub infrastructure.

Ankolekar et al. [16] argued in 2007 that two "rivaling" directions, the Web 2.0 and Semantic Web should gain from each other's strengths. In this article they set three hypotheses. Two of them - "the Semantic Web will be World Wide Web" and a "bottom up user-centered approach is required for the Semantic Web to take hold" - are actually quite relevant in context of this paper. Using the SOAP Proxy with OpenAjax Hub and Transformer Widget does bring the Semantic Web much closer to the end users while at the same time the mashups will live in the World Wide Web.

The same article lists three infrastructure issues that need to be solved in regards of wider spread of Semantic Web: the creation of semantic data, exchanging generated data and reusing the data. Mashups are mentioned as one way to reuse the semantic data.

Hornung et al. have also been interested in surfacing the Deep Web through [20].Their approach involves the use of graphical tools to compose the mashups and to access the Deep Web, they offer the ability to use web forms. By using the web forms, they have to tackle of several problems like form interaction, data record extraction and cleaning, fuzzy result lists and data cleaning. Those problems are not needed to be dealt with when using well defined interfaces of web services. One issue is relevant for both cases - by chaining data sources, each subsequent data source needs to be queried with all (meaningful) combinations found so far. In their paper they show that this can lead to combinatorial explosion in possible value combinations.

In the paper about Semantic Web and Web 2.0 by Christopher Thomas and Amit Sheth [26], it is discussed how can those technologies be combined to help provide the platform for solving complex problems. They conclude that the Semantic Web must provide platforms that facilitate the use of semantics, that hide the formalisms from those who do not want and do not need to see them, that connect the things that are interesting to everyone to those that are interesting only to Semantic Web visionaries. The solution, that this thesis provides, does not currently meet this vision, as while describing the widgets' interfaces a user should still know the correct semantic vocabulary to use.

5.5 Bypassing Same Origin Policy

Salminen et al. [24] discuss several possibilities to bypass the Same Origin Policy restriction. First workaround is using the HTML script tag in conjunction with JSONP transport protocol or using server-side proxy. In this case the remote server must have a support for JSONP (this does not come out-of-the-box). They also stress, that usage of JSONP involves some risks in the form of man-in-the-middle attacks and cases when the remote site is untrustworthy. Second way is to use Adobe Flash object based proxy, that is usable only if the remote server provides a special file that grants access to remote domain flash objects. Of course, user must also have the Flash plug-in installed. Third possibility is to use a server-side proxy, which is good because the proxy does not set any additional requirements to the remote server.

There is a W3C working draft document Cross-Origin Resource Sharing [27] that provides guidelines on how the cross-domain issues could be possibly solved in the future. However, this approach still seem to introduce the need for updates in the remote site.

HTML5 standard is expected to introduce a new specification - HTML5 Web Messaging [19] - that defines mechanisms for communicating between browsing contexts in HTML documents. By using this specification there are no changes needed in the remote site anymore.

5.6 Accessing SOAP Services From Browser

There exist JavaScript libraries that enable sending requests to SOAP endpoints straight from within the browser. One of them is JavaScript SOAP Client [4]. While the presented solution could have embraced the library and not constructed the SOAP messages in the server-side it would still have been needed to use proxy to overcome the cross-site restrictions. Also, the Transformer Widget mappings need to be constructed somewhere and it is doubtful that there exist JavaScript libraries as powerful as SoapUI is.

There is also the IBM's SOAP extension for Dojo Toolkit [2] which is part of IBM WebSphere Application Server Feature Pack for Web 2.0. The Dojo extension comes with functionality that makes it possible to call SOAP services with automatically generated service wrappers. The extension parses WSDL documents to generate the SMD descriptions that Dojo natively supports. The extension lacks support for more complex data structures and the SOAP service result is returned in XML not in JavaScript native JSON format. The WebSphere server can be used to bypass the Same Origin Policy restrictions. The license information was not studied so it is unknown if the Dojo extension can actually be used separately from the WebSphere application server.

5.7 Problems With Mashups

[24] points out that changes in service interfaces causes lots of trouble to the mashup developers. This is one of the problems that using semantically annotated services can minimize to some extent. The same article also discusses the legal issues that arise with mashups when various service providers have different and sometimes conflicting terms of usage or when other rules have to be followed.

A survey [31] was conducted among mashup developers to analyze the participants in the community and also detect problems that occur when developing mashups. The three more problematic areas were: the reliability of the API, documentation, and coding details. Latter being mainly in the form of JavaScript skills needed to integrate APIs.

5.8 Semantic Web Services

It is interesting to find out how many SOAP web services a potential user can find to unleash the possibilities of presented solution. A research from 2008 [21] included all semantic Web services that could be found in the surface Web by using a special meta-search engine Sousuo does not give good results. At that time they found just around 1500 indexed semantic service descriptions. Of course time has passed, and this number has probably increased. But still the same research concluded that, at that time, this number was expected to be bigger.

Chapter 6

Conclusion

This thesis investigated the ways to ease the surfacing of SOAP web services that are part of the Deep Web. The main focus was to solve the problems related to creating mashup applications on the presentation layer.

At first a possible application scenario was introduced, where a developer wants to use a Estonian Business Registry SOAP service, to build a client-side mashup. The application would query the list of annual reports for a specific company and then visualize the list.

Then the problems, that make developing such applications difficult, were introduced. It was found out, that a developer has to struggle with Same Origin Policy and the lack of support for creating SOAP requests in JavaScript. It was also apparent that visualizing output of the services requires knowledge of the SOAP message structure and lots of manual work.

In order to resolve these problems, a solution was provided, that includes the server-side and the client-side components. On the client-side, an approach of using visual and hidden widgets was chosen. This means that for each SOAP operation, a hidden widget will be created. A hidden widget acts like a proxy to the actual SOAP service operation and provides data to the visualization widget.

The communication infrastructure between widgets was established with the help of OpenAjax Hub and Transformer Widget. OpenAjax Hub provides a publish/subscribe mechanism to route the messages between the hub clients. Transformer Widget takes care of transforming messages from one client to another based on the mappings, where the input and output messages of widgets are mapped with related semantic vocabulary. Transformer Widget also needs JSON schemas when it deals with transforming JSON data from one widget to the other.

The Proxy Widget was introduced to automatically generate hidden widgets. The Proxy Widget is a OpenAjax Hub widget, that proxies requests to the SOAP endpoints. When the Proxy Widget connects to the hub, it needs to provide mappings and JSON schema for the Transformer Widget. Those are generated by server-side services, that take the URL of a WSDL document and the name of the operation as input parameters. Then they use the structure of the actual SOAP messages to generate the needed documents.

The actual proxying of requests was done by first creating JSON-RPC requests on the Proxy Widget and then sending them to the server-side. The server-side translates this request to a SOAP request and sends it forward to the actual endpoint. The response is translated back to the JSON-RPC and forwarded to the Proxy Widget.

The provided solution was tested in a proof of concept application that built upon the scenario that was used in the introduction. The purpose of the demo applications was to access Estonian Business Registry SOAP services in order to search and display information about companies. Three Proxy Widgets were created to enable querying for the list of businesses, the list of annual reports and the data for each report. The output of each operation was visualized by a special widget. The demo proved that the solution enables developers to create client-side mashups using SOAP services.

Chapter 7

Future Work

The presented solution could be improved in various aspects. One of those improvements could be the possibility to control the flow of service calls. This responsibility could be part of the Transformer Widget and would require a development of how to describe the flow and how to process this description inside the Transformer Widget.

Another limitation of current solution is the inability to cache mappings or other generated metadata. By introducing caching, the responsiveness of the server-side component would become much better. The cached metadata could then be used by anyone, who asks for mappings from the server-side. Because the service interfaces do not change very often, the cache time-to-live could be quite high.

The system could also benefit from the ability to automatically detect services that can operate on certain semantic data. Probably this would need to be a separate service that can then be called to get the list of usable services. Those services could then be used to automatically generate widgets for the user. Using Estonian Business Registry (EBR) as an example, imagine, that user asks what services can she use, if she only has a name of a company. The service would then automatically look for operations, that can take the name of the company as an input. The service would detect that there is an operation ”findBusiness” described in the WSDL of the EBR SOAP service. It would then pass on that information to the mashup application, which would then automatically create the Proxy Widget for that service.

Current solution also lacks support of exception handling. There is no easy way to recover from cases when a web service is down, a specified WSDL document cannot be accessed or the operation is not found inside the WSDL document. Luckily the JSON-RPC protocol has support for passing information about errors to the client-side.

It would also be great, if not only SOAP services would be supported by the Proxy Widget. There could be applied some virtualization mechanisms, that would transform the information in the JSON-RPC request to a general request. This general request could then be converted to any other request type, like SOAP or REST and passed on to the actual endpoint. The results would then be transformed back to format suitable for JSON-RPC.

This thesis could also have some positive effect on adding semantic annotations to SOAP web services. At the moment, there are very little services, that are semantically annotated. If the owners of services see, that the presented solution eases the consuming of their services, they might be more motivated to increase the priority of this task.

It is currently possible to use the infrastructure without semantically annotated services. In this case, of course, it is not possible to use the Transformer Widget to do the semantic integration between widgets. However, as it still makes the using of SOAP services a lot easier, it would make sense to define separate interfaces for the usage without semantics. Perhaps even a two separate widgets could be used - Semantic Proxy Widget and just Proxy Widget, which would not have support for semantics.

Süvaveebi kuvamine automaatsete OpenAjax Hub vidinate genereerimise abil

Magistritöö (30 EAP)

Karli Kirsimäe

Resümee

Antud magistritöö uurib, kuidas lihtsustada esitluskihil SOAP protokolli kasutavate veebiteenuste, mis on osa süvaveebist, kasutamist. Sellise teema valimist motiveerib asjaolu, et rakenduste kompositsiooniline raskuskese liigub üha enam esitluskihi suunas, kuid hetkel ei ole veebilehitsejale omaste tehnoloogiatega võimalik väliste domeenide teenuseid kasutada, nende väljundit kuvada ja teenuseid omavahel siduda.

Et välja selgitada, kuidas antud probleemi lahendada, uuriti, mis on hetkel sellise lähenemise kasutusse võtmisel peamised pidurdavad tegurid. Selgus, et põhilisi raskusi tekitavad asjaolud, et veebilehitsejad ei võimalda teha päringuid rakenduse suhtes välistesse domeenidesse ja et JavaScriptis on SOAP päringute koostamise tugi võrdlemisi limiteeritud. Lisaks tõdeti, et teenustest saadava info visualiseerimine nõuab teenuse väljundi ja kuvamisloogika manuaalset kokku-traageldamist (*hard-wiring* ing k).

Probleemi lahendamiseks otsustati kasutada nö veebividinapõhist lähenemist, kus iga teenuse operatsiooni jaoks genereeritakse nähtamatu JavaScripti vidin, millelt saadav info muudetakse nähtavaks mõne teise vidina poolt. Sellise lähenemise rakendamiseks loodi kaheosaline raamistik, mis koosneb kliendikihist ja serverikihist. Vidinate suhtlemise võimaldamiseks võeti kasutusele OpenAjax Hub raamistik [6], mis toimib vidinatevaheliste sõnumite vahendajana. Selleks, et vidinad ei oleks tihedalt kokku traageldatud, võeti appi Transformer Widget [28]. Transformer Widget lisab OpenAjax Hub vidinatele võimaluse omavahel suhelda, kasutades semantilist integreerimist.

Nähtamatute vidinate genereerimiseks loodi eraldi OpenAjax Hub vidin - Proxy Widget. See toimib teenuseid tarbivate vidinate ja tegeliku teenuse vahelise puhvrina ning lisaks hoolitseb selle eest, et vidin oleks korrektselt

Transformer Widgetis registreeritud. Transformer Widgetis registreerimiseks pakub tuge ka serveripool. Serveris genereeritakse selle jaoks dokument, mis kirjeldab vidinate struktuuri ja semantikat ning lisaks ka skeem JSON vormingus andmete kirjeldamiseks. Serveripool kasutab selle jaoks teenuse semantiliselt annoteeritud WSDL keeles kirjeldust, kust saadakse kõik vajalik informatsioon.

Proxy Widgeti puhverdamisloogika toimib nii, et esitluskihis võetakse sisendisse JSON vormingus andmed, mille abil luuakse JSON-RPC päring. See saadetakse edasi serveripoolele, mis omakorda transformeerib päringu SOAP päringuks ning saadab lõppteenusele. Lõppteenuselt saadud vastus teisendatakse tagasi JSON-RPC päringuks ning edastatakse Proxy Widgetile.

Välja pakutud lahenduse toimimist testiti näidisrakendusega, kus esitluskihi tasemel võimaldati tarbida kolme Äriregistri teenust - firmade leidmine nime järgi, firma aastaaruannete leidmine ning aastaaruannete andmete leidmine. Näidisrakendus tõestas, et teenuste tarbimine ning andmete kuvamine osutus antud lahendusega oluliselt lihtsamaks. Lisaks oli see tõestuseks, et teenuste tarbimine oli võimalik vaid veebilehitsejale omaste tehnoloogiate kasutamisega.

Bibliography

[1] http://dojotoolkit.org/. Cited: May 15, 2011.

[2] IBM's SOAP extension for Dojo. http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/web20-mobile/. Cited: May 9, 2011.

[3] Introducing JSON. http://www.json.org/. Cited: May 14, 2011.

[4] JavaScript SOAP Client. http://javascriptsoapclient.codeplex.com. Cited: May 7, 2011.

[5] OpenAjax Hub 2.0 and Mashup Assembly Applications.

[6] OpenAjax Hub 2.0 Specification. http://www.openajax.org/member/wiki/OpenAjax_Hub_2.0_Specification. Cited: May 14, 2011.

[7] Yahoo pipes. http://pipes.yahoo.com/pipes/. Cited: May 13, 2011.

[8] JSONP: JSON With Padding. http://ajaxian.com/archives/jsonp-json-with-padding, December 2005. Cited: May 14, 2011.

[9] Semantic Annotations for WSDL and XML Schema. W3C Recommendation. Technical report, August 2007. Cited: May 14, 2011.

[10] SOAP Version 1.2. W3C Recommendation. Technical report, April 2007. Cited: May 14, 2011.

[11] Web Services Description Language (WSDL) Version 2.0. Technical report, June 2007. Cited: May 14, 2011.

[12] Service Mapping Description Proposal. http://groups.google.com/group/json-schema/web/service-mapping-description-proposal, November 2008. Cited: May 14, 2011.

[13] A JSON Media Type for Describing the Structure and Meaning of JSON Documents. http://tools.ietf.org/html/draft-zyp-json-schema, November 2010. Cited: May 14, 2011.

[14] JSON-RPC 2.0 Specification. http://groups.google.com/group/json-rpc/web/json-rpc-2-0, March 2010. Cited: May 14, 2011.

[15] Tanya Ahmed. Usability review of mashup tools. May 2009.

[16] Anupriya Ankolekar, Markus Krötzsch, Thanh Tran, and Denny Vrandecic. The two cultures: mashing up web 2.0 and the semantic web. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 825–834, New York, NY, USA, 2007. ACM.

[17] Marcos Caceres. Widget packaging and configuration. w3c working draft. Technical report, March 2010. Cited: May 12, 2011.

[18] Robert J. Ennals and Minos N. Garofalakis. Mashmaker: mashups for the masses. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1116–1118, New York, NY, USA, 2007. ACM.

[19] Ian Hickson. HTML5 Web Messaging. W3C Working Draft 17 March 2011. http://www.w3.org/TR/webmessaging/. Cited: May 7, 2011.

[20] T. Hornung, K. Simon, and G. Lausen. Mashups over the Deep Web. In J. Cordeiro, S. Hammoudi, & J. Filipe, editor, *Web Information Systems and Technologies*, pages 228–+, 2009.

[21] M. Klusch and X. Zhing. Deployed semantic services for the common user of the web: A reality check. In *Semantic Computing, 2008 IEEE International Conference on*, pages 347 –353, aug. 2008.

[22] Jim Murphy Eric Knipp David Mitchell Smith David W. Cearley Ray Valdes, Gene Phifer. Hype cycle for web and user interaction technologies, 2010. 2010.

[23] Marwan Sabbouh, Jeff Higginson, Salim Semy, and Danny Gagne. Web mashup scripting language. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1305–1306, New York, NY, USA, 2007. ACM.

[24] Arto Salminen, Tommi Mikkonen, Feetu Nyrhinen, and Antero Taivalsaari. Developing client-side mashups: experiences, guidelines and the road ahead. In *Proceedings of the 14th International Academic MindTrek Conference: Envisioning Future Media Environments*, MindTrek '10, pages 161–168, New York, NY, USA, 2010. ACM.

[25] A. Taivalsaari and T. 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, sept. 2008.

[26] Christopher Thomas and Amit Sheth. Web wisdom: An essay on how web 2.0 and semantic web can foster a global knowledge society. *Computers in Human Behavior*, 27(4):1285 – 1293, 2011. Social and Humanistic Computing for the Knowledge Society.

[27] Anne van Kesteren. Cross-Origin Resource Sharing. W3C Working Draft. `http://www.w3.org/TR/cors/`, July 2010. Cited: May 7, 2011.

[28] Rainer Villido. Semantic Integration Platform for Web Widgets' Communication. Master's thesis, Tartu Ülikool, Estonia, 2010.

[29] Jeffrey Wong and Jason I. Hong. Making mashups with marmite: towards end-user programming for the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '07, pages 1435–1444, New York, NY, USA, 2007. ACM.

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

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

Appendix

Source Code

The source code of the provided solution can be downloaded from Github: https://github.com/karli/Automatic-soap-widget-generator.