Title: RESTful API of a browser to browser call service based on WebSockets

Volume: 1/1

Author: Francesc Montserrat Carvajal

Date: June 2012

Supervisor: Ján Murányi

Department: Institute of Computer Systems and Networks

University: Slovak University of Technology in Bratislava

Faculty: Faculty of Informatics and Information Technologies
RESTful API of a browser to browser call service based on WebSockets

Francesc Montserrat Carvajal

June 2012
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.4</td>
<td>Hibernate</td>
<td>34</td>
</tr>
<tr>
<td>5.1.5</td>
<td>JAXB</td>
<td>34</td>
</tr>
<tr>
<td>5.1.6</td>
<td>jWebSocket</td>
<td>34</td>
</tr>
<tr>
<td>5.1.7</td>
<td>Tomcat</td>
<td>34</td>
</tr>
<tr>
<td>5.1.8</td>
<td>Maven</td>
<td>34</td>
</tr>
<tr>
<td>5.1.9</td>
<td>GIT</td>
<td>35</td>
</tr>
<tr>
<td>5.1.10</td>
<td>jQuery</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Server Implementation</td>
<td>36</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Communication Layer: The RESTful API</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Communication Layer: The jWebSocket Plugin</td>
<td>36</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Domain Model Services</td>
<td>38</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Towards automatic model generation</td>
<td>39</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Persistence Layer</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Client Implementation</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Conclusions</td>
<td>42</td>
</tr>
<tr>
<td>6.1</td>
<td>Testing Scenarios</td>
<td>43</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Alice and Bob: Unicast communication</td>
<td>43</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Alice, Bob and Admin: Simultaneous unicast communications</td>
<td>43</td>
</tr>
<tr>
<td>6.1.3</td>
<td>API tests</td>
<td>46</td>
</tr>
<tr>
<td>6.2</td>
<td>Test Results</td>
<td>47</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Single Unicast communication results</td>
<td>47</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Simultaneous unicast communications results</td>
<td>48</td>
</tr>
<tr>
<td>6.3</td>
<td>Conclusions</td>
<td>48</td>
</tr>
<tr>
<td>7.1</td>
<td>Goals</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Future Work</td>
<td>50</td>
</tr>
<tr>
<td>7.3</td>
<td>Used and Acquired Knowledge</td>
<td>51</td>
</tr>
<tr>
<td>7.4</td>
<td>Acknowledgement</td>
<td>51</td>
</tr>
<tr>
<td>A</td>
<td>API Reference Manual</td>
<td>53</td>
</tr>
<tr>
<td>A.1</td>
<td>User Resource</td>
<td>53</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Main URL</td>
<td>53</td>
</tr>
<tr>
<td>A.1.2</td>
<td>Retrieve the current user information</td>
<td>53</td>
</tr>
<tr>
<td>A.1.3</td>
<td>Search user by username</td>
<td>54</td>
</tr>
<tr>
<td>A.1.4</td>
<td>Create a new user</td>
<td>54</td>
</tr>
<tr>
<td>A.1.5</td>
<td>Update the current user information</td>
<td>54</td>
</tr>
<tr>
<td>A.1.6</td>
<td>Delete the current user</td>
<td>55</td>
</tr>
<tr>
<td>A.2</td>
<td>Registration Resource</td>
<td>56</td>
</tr>
<tr>
<td>A.2.1</td>
<td>Main URL</td>
<td>56</td>
</tr>
<tr>
<td>A.2.2</td>
<td>Register the current user</td>
<td>56</td>
</tr>
<tr>
<td>A.2.3</td>
<td>Unregister the current user</td>
<td>56</td>
</tr>
<tr>
<td>A.3</td>
<td>Update Resource</td>
<td>57</td>
</tr>
<tr>
<td>A.3.1</td>
<td>Main URL</td>
<td>57</td>
</tr>
<tr>
<td>A.3.2</td>
<td>Update the current user registration</td>
<td>57</td>
</tr>
<tr>
<td>A.4</td>
<td>Contacts Resource</td>
<td>58</td>
</tr>
<tr>
<td>A.4.1</td>
<td>Main URL</td>
<td>58</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Retrieve the list of contacts of a user</td>
<td>58</td>
</tr>
</tbody>
</table>
CONTENTS

A.4.3 Retrieve a single contact ........................................ 58
A.4.4 Add a contact to the contact list .................................. 59
A.4.5 Delete a contact .................................................. 59
A.5 Call Resource ....................................................... 60
A.5.1 Main URL ......................................................... 60
A.5.2 Invite another user to a call ..................................... 60
A.5.3 Answer an incoming call ......................................... 60
A.5.4 End a call ......................................................... 61

B Installation Manual ..................................................... 63
B.1 Install Tomcat ......................................................... 63
B.2 Configure Tomcat ..................................................... 63
B.2.1 Add User Realm ................................................... 63
B.2.2 Add Tomcat Manager User ....................................... 64
B.3 Install Maven ......................................................... 65
B.4 Configure Maven ...................................................... 65
B.4.1 Define Maven Settings ............................................ 65
B.5 Install MySQL ......................................................... 66
B.6 Configure MySQL ..................................................... 66
B.6.1 Add MySQL Connector to Tomcat ................................. 66
B.7 Install jWebSocket .................................................... 67
B.7.1 Install Server ...................................................... 67
B.7.2 Install Channel PlugIn ............................................ 67
B.7.3 Set Environment Variables ....................................... 67
B.8 Deploy Project ....................................................... 68
B.8.1 Run Tomcat ......................................................... 68
B.8.2 Compile and Deploy Project ..................................... 68
B.8.3 Test it! ............................................................ 68

Glossary ................................................................. 69

Bibliography ............................................................ 74
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>SIP infrastructure example</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Safety, idempotency and use of HTTP methods</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>SIP-REST request mapping</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Message exchange between UAC and UAS</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Message exchange between UAC</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>Specification main flow sequence diagram</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Visual overview of the server</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>SIP-WS token mapping</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Design connection establishment flow sequence diagram</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>Design call rejection alternative flow sequence diagram</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Design call termination flow sequence diagram</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Conceptual data model</td>
<td>31</td>
</tr>
<tr>
<td>5.1</td>
<td>RESTful resource example: GET over User Resource</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>jWebSocket plug-in chain</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>Adding a jWebSocket plug-in to the chain</td>
<td>38</td>
</tr>
<tr>
<td>5.4</td>
<td>User XSD schema</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Client REST request example</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Simple unicast test description</td>
<td>44</td>
</tr>
<tr>
<td>6.2</td>
<td>Simultaneous unicast test description</td>
<td>45</td>
</tr>
<tr>
<td>6.3</td>
<td>XHR Poster screenshot</td>
<td>46</td>
</tr>
<tr>
<td>6.4</td>
<td>Clients after single unicast test</td>
<td>47</td>
</tr>
<tr>
<td>A.1</td>
<td>GET User XML example</td>
<td>53</td>
</tr>
<tr>
<td>A.2</td>
<td>GET user JSON example</td>
<td>53</td>
</tr>
<tr>
<td>A.3</td>
<td>Search User XML example</td>
<td>54</td>
</tr>
<tr>
<td>A.4</td>
<td>Search user JSON example</td>
<td>54</td>
</tr>
<tr>
<td>A.5</td>
<td>Register XML example</td>
<td>56</td>
</tr>
<tr>
<td>A.6</td>
<td>Register JSON example</td>
<td>56</td>
</tr>
<tr>
<td>A.7</td>
<td>Update XML example</td>
<td>57</td>
</tr>
<tr>
<td>A.8</td>
<td>Update JSON example</td>
<td>57</td>
</tr>
<tr>
<td>A.9</td>
<td>GET contacts XML example</td>
<td>58</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

A.10 GET contacts JSON example ............................................. 58
A.11 Call XML example ......................................................... 60
A.12 Call JSON example ......................................................... 60

B.1 server.xml ...................................................................... 63
B.2 tomcat-users.xml ............................................................... 64
B.3 settings.xml .................................................................... 65
B.4 .bashrc ............................................................................. 67
Chapter 1

Introduction

1.1 Motivation

The aim of this thesis is to define a way to asynchronously create a general purpose server-controlled client-to-client-communication using only static HTML5 standards and dynamic client-side scripting.

1.2 Objectives

The main goals of this project are:

- Define a simple RESTful SIP-like communication protocol to negotiate the connection.
- Use the previously defined API to establish a client-to-client connection over standard WebSockets.
- Implement a simple solution of the server side application and a lightweight web client to test it.
- Turn this solution into a framework that could help future research in communications on top of it.
- Define and use a simple illustrative data representation to test the communication between clients once the session is established.

1.3 Organization of this memory

This memory is organized following the classic software development model or 'Waterfall Model' due to present it in a structured lineal order, nevertheless the project has been developed following an iterative and incremental model. The memory may include references to the incremental process in order to document and justify decisions and to modularize the project development into sets of features distributed among the iterations. The sections are distributed as follows:

- Introduction (chapter 1). In this chapter the project motivation, goals and memory structure are defined.
• **State of the art** (chapter 2). This chapter introduces previous work and similar projects in the same scope. Its aim is to set this thesis on context and to help understanding this memory.

• **Analysis** (chapter 3). This chapter includes the requirement analysis and the functional technology-independent Software Engineering technical specification of the System.

• **Design** (chapter 4). This chapter defines the solution in terms of architectural view, design use case description, modularity and connectivity of System components.

• **Implementation** (chapter 5). This chapter defines and justifies the selected technologies and shows an overview of the software implementation.

• **Testing** (chapter 6). This chapter covers the main testing scenarios and domain use cases including its definition, representation, execution and results.

• **Conclusions** (chapter 7). This chapter includes a summary of the conclusions of the thesis. Every previous chapter includes partial conclusions as well.

• **API Reference Manual** (appendix A). This chapter includes a user guide to the RESTful API of the System.

• **Installation Manual** (appendix B). This chapter includes a manual to install the System in Ubuntu 11.10.
Chapter 2

State of the art

In this section we are going to briefly analyze the properties of Session Initiation Protocol (SIP). We will also explore the Web Real Time Communication (WebRTC) standard which is being drafted by the World Wide Web Consortium (W3C).

2.1 SIP

The SIP is an IETF-defined signaling protocol widely used for controlling communication sessions over IP. The protocol is used for initiating, modifying or terminating sessions between two (unicast) or more (multicast) parties. It is an Application Layer protocol and it is text based. SIP has some similitudes with other Application Layer protocols such as HTTP (Hypertext Transfer Protocol) and reuses most of the header fields, encoding rules and status codes of HTTP, providing a readable text-based format.

The SIP also define server network elements creating its own SIP infrastructure which main element is the User Agent.

User Agent Is the logical network end-point used to create or receive the messages. A SIP UA can perform the role of a User Agent Client (UAC) or a User Agent Server (UAS). This roles only last during the duration of the SIP transaction. The protocol also defines Proxy Servers, Registration Servers, Redirection Servers or Gateways.

To establish a SIP two-party (unicast) session the main messages exchanged -and the most interesting for the purposes of this thesis- are:

- REGISTER: Used by a UA to indicate its location (IP address) and the URLs for which it would like to receive calls.
- INVITE: Indicates a client is being invited to participate in a call session.
- ACK: Confirms that the client has received a final response to an INVITE request.
- CANCEL: Cancels any pending request.
- BYE: Terminates a call and can be sent by either the caller or the callee.
- SUBSCRIBE: Subscribes for an Event of Notification from the Notifier.
• **NOTIFY:** Notify the subscriber of a new Event.

• **PUBLISH:** Publishes an event to the Server.

• **UPDATE:** Modifies the state of a session without changing the state of the dialog.

Figure 2.1: SIP infrastructure example

The figure 2.1 shows an example of how the messages are exchanged in a SIP infrastructure.
2.2 WebRTC

WebRTC is a HTML5 standard being drafted by the W3C that enables applications such as voice calls, video chat, P2P file sharing and so on. It is not complete and while early experimentations are encouraged, it is therefore not intended for implementation. The WebRTC consists of two pieces:

- A protocol specification, done in the IETF (Internet Engineering Task Force)
- A JavaScript API specification, done in the W3C

"Together, these two specifications aim to provide an environment where JavaScript embedded in any page, viewed in any compatible browser, when suitably authorized by its user, is able to set up communication using audio, video and auxiliary data, where the browser environment does not constrain the types of application in which this functionality can be used." IETF

2.2.1 Implementations

Google Chrome, Mozilla Firefox and Internet Explorer have started working on their own implementation/integration of the WebRTC API. There are other organization/projects working on it such as Ericsson Labs or a migration of Google Talk to the WebRTC framework.

2.2.2 The WebRTC Framework

WebRTC

"WebRTC is an open framework for the web that enables Real Time Communications in the browser. It includes the fundamental building blocks for high quality communications on the web such as network, audio and video components used in voice and video chat applications. These components, when implemented in a browser, can be accessed through a JavaScript API, enabling developers to easily implement their own RTC web app." http://www.webrtc.org

The main keys of this project are that is free, standard, complete and already integrates the main voice and video engines; and there is no previous solution working on top of a web browser with all those qualities. It is supported by Google, Mozilla and Opera.

2.3 Conclusions

Real time communication in web browsers is not yet a standard and there is still space for experimentation. This thesis is an experimental approach to define an asynchronous method to establish sessions between browsers using technologies that are already standard and provide a context for further real time communication research.
Chapter 3

Analysis

This section includes the requirement analysis and the functional technology-independent Software Engineering technical specification of the system presented in this thesis.

3.1 SIP and REST

Representational state transfer (REST) is a software architecture style for designing network applications and distributed systems. It is a Web service model much simpler and flexible than other models such as SOAP or WDSL. It was developed in parallel with HTTP/1.1 and was initially described in the context of HTTP but it is not limited to this protocol. REST is an architecture, not a protocol and it may be based on other Application Layer protocols if they already provide a rich and uniform vocabulary for applications based on the transfer of meaningful representational state.

The first goal of this thesis is to define a SIP-like communication protocol to negotiate the connection. REST has been chosen as the architecture to base the project API -or web service- for the following reasons:

- **HTTP reuse**: Both SIP and REST reuse header fields, vocabulary and status codes of the HTTP protocol.
- **Generality of interfaces**: Due to its architecture nature, REST is a very flexible model to send SIP messages in a simple non-verbose style.
- **HTTP/1.1 compatibility**: The main verbs of REST (GET, POST, PUT and DELETE) are natively supported by web browsers since they support HTTP/1.1.
- **Client-side API**: REST interfaces can be accessed natively from scripting client-side languages such as JavaScript due to its HTTP context.
- **Previous experience**: The author of this thesis had had previous experiences designing, implementing and using RESTful APIs.

The API of this thesis will be designed not to completely implement SIP but to use the SIP terminology while following the RESTful principles of **idempotency**, **safety** and **content negotiation**.
Safety

"In HTTP, safe methods are not expected to cause side effects. Clients can send requests with safe methods without worrying about causing unintended side effects. To provide this guarantee, implement safe methods as read-only operations." *Subbu Allamaraju, RESTful Web Services Cookbook*

Idempotency

"Idempotency guarantees clients that repeating a request has the same effect as making a request just once. Idempotency matters most in the case of network or software failures. Clients can repeat such requests and expect the same outcome." *Subbu Allamaraju, RESTful Web Services Cookbook*

Content negotiation

"Content negotiation, or *conneg* as it is sometimes called, is the process of selecting the best representation of a resource for a client when there are multiple representations (or variants) available." *Subbu Allamaraju, RESTful Web Services Cookbook*

The table in the figure 3.1 summarizes the uses and properties of the main HTTP1.1 methods in a RESTful API:

**Table 3.1: Safety, idempotency and use of HTTP methods**

<table>
<thead>
<tr>
<th>Verb</th>
<th>Safe?</th>
<th>Idempotent?</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Yes</td>
<td>Yes</td>
<td>• Safe and idempotent information retrieval</td>
</tr>
<tr>
<td>PUT</td>
<td>No</td>
<td>Yes</td>
<td>• Update resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Create new resources only when clients can decide URIs of resources</td>
</tr>
<tr>
<td>DELETE</td>
<td>No</td>
<td>Yes</td>
<td>• Delete resources</td>
</tr>
<tr>
<td>POST</td>
<td>No</td>
<td>No</td>
<td>• Create new resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Any unsafe or nonidempotent operation when no other method is appropriate</td>
</tr>
</tbody>
</table>

The resource collection of the System follows the previous principles to cover the required SIP requests in the context of this thesis. The table in the figure 3.2 maps a draft of the REST web service to the main SIP request messages:
The **SUBSCRIBE**, **NOTIFY** and **PUBLISH** requests will be transmitted over a different communication channel. Unlike the other SIP request codes, these three methods are used in a context of event-driven communication. REST only support synchronous (in contrast to asynchronous) messaging, because of its reliance on HTTP and HTTPS infrastructure. An asynchronous channel has to be created during the registration (**POST** to the Registration resource) and kept open until the unregistration process (**DELETE** to the Registration resource) ends the session. The figure 3.3 shows how a User Agent Client (UAC) and a User Agent Server (UAS) will exchange messages in the context of this thesis.

The asynchronous communication channel has to stay opened during the session in order to keep the event exchange system functional and ensure correct information transmission between User Agents. According to the objectives of this thesis, the UACs of the system are intended to use web browsers to communicate through both the synchronous and asynchronous channels. An important functional requirement of the System is that these channels have to be accessible by any UAC **using only HTML5 and client-side scripting languages**.
This two channels will be used by UACs to initiate, maintain and terminate any client-to-client session. The two client-to-server channels, used together provide a mechanism to manage the current session or create new sessions with other UAC which can be overlapped in time. The figure 3.4 shows how two UAC can communicate using a server-controlled communication.

The figure 3.5 represents the main flow of the UAC registration use case. In this use case, Alice makes use of the RESTful API of the UAS to request registration. Once the registration is done, Alice can subscribe to the asynchronous channel to receive notifications from the server. At this point the connection between the UAC (Alice) and the UAS is fully functional and Alice can use it to request communication with other UAC. In our use case, Alice invites another UAC (Bob) to establish a session. In the main flow we should assume that Bob is already registered in the System and subscribed to notifications. The UAS will forward Alice’s request to Bob through the asynchronous connection between Bob and the server (see figure 3.4). Now Bob can accept Alice’s request and join the session finishing the client-to-client communication channel establishment. In an alternative flow, Bob could want to dismiss the request by sending the CANCEL message through the synchronous channel (REST API). In this case Alice would receive a notification over the asynchronous channel and the session creation would be interrupted.

To end the session, either of the UAC can send a BYE message (DELETE to the Call resource) requesting the UAS to terminate that specific session. In a multicast approach of the system, UAC should be able to leave the session without ending it. To end the client-to-server communication, a UAC has to unsubscribe from the asynchronous channel and finally unregister from the System (DELETE to the Registration resource) to inform the Server that the UAC is not available for client-to-client communications anymore. On conclusion, the registration and unregistration negotiation processes are symmetric.
3.2 System Requirements

3.2.1 Functional Requirements

The following functional requirements have been defined based on the previous analysis, the goals defined in chapter 1 and a generalization of the most common VoIP service functionalities:

Client requirements:

- **Must** be implemented using only static web pages and client-side scripting.
- **Must** be able to communicate with both the synchronous and the asynchronous front ends of the server. "Communicate" includes making requests, processing responses and serving requests from other UC.
- **Must** provide an illustrative example of client-to-client data transmission in a *unicast* session.
- **Must** be able to perform any call required by the main use case of the project.
- **Will be** a simplified sample client of the functionalities of the project. Secondary functionalities will not necessarily be accessible from the client (any secondary REST call can be tested using a third party RESTful client).

Server requirements:

- **Must** define two front ends for clients: a first one for synchronous requests and a second one for asynchronous requests.
- The synchronous front end **will be** a RESTful API.
- The API **will provide** different data representations (Extensible Markup Language (XML) and JavaScript Object Notation (JSON)).
- The requests commands and code names **will try to** follow those defined by SIP and the table in the figure 3.2.
- **Must** be able to initiate, modify and terminate a client-to-client communication channel.
- The server **will not permit** multicast sessions but it is considered as a potential future improvement.
- **Must** be able to persist information in a database.

System user management requirements:

- **Must** be able to create, retrieve, modify and delete user information (creating, deleting and modifying are secondary features).
- **Must** be able to manage user contacts in a asymmetric way (adding and deleting contacts are secondary features).
- **Must** define the status of a user and allow each user to change it: The possible status will be: *not registered, available, busy, away and invisible*. 
CHAPTER 3. ANALYSIS

Security Requirements:
- Security is **not considered a major priority** in the scope of this project.
- User private information **must** be retrieved only by the owner of the information.
- User public information can be retrieved by any user registered in the system’s database.
- Access to API resources **will be** limited by roles.
- Activity of invisible users (users in *invisible* status) **must not be broadcasted** through any of the communication channels.

3.2.2 Non-Functional Requirements

The main non-functional requirements have been defined in the following basis:
- This thesis will be used by others in future research.
- The project is experimental and may need to be extended or modified in the future.

Included non-functional requirements:
- **Open Source**: The project will try to use open source based technologies if possible.
- **Extensibility**: New features can be added to the project (a RESTful API can easily be extended and versioned).
- **Portability (client-side)**: New client implementations of different natures (such as web clients, desktop applications or smartphone apps) can be created consuming the RESTful API and implementing a different (or the same) asynchronous channel.
- **Maintainability**: The project will include automated code generation to avoid repetitive tasks if extended.
- **Documentation**: This memory should be sufficient to understand and help extending the project. The server installation manual should include all the steps to deploy the server in the defined platform.

Excluded non-functional requirements:
- **Security**: Security is considered only by the use of roles, permissions and the right behavior and responses of the server. The project is experimental and analysis of backdoors or know issues of the chosen technologies won’t be a priority.
- **Portability (server-side)**: The server of the system may be portable but won’t be tested. Any future port attempt "will be at your own risk".
- **Scalability**: The System scalability won’t be tested.
3.3 Conclusions

The System will use two communication channels between users and the server, being one of them synchronous (RESTful API) and the other asynchronous. The asynchronous channel will be implemented by a standard technology that will be chosen in design time. The session establishment will follow the process defined in figure 3.5.

Once the session is established, the System will provide a client-to-client asynchronous communication channel that can or cannot be based in the same technology than the asynchronous client-to-server channel.

In this chapter we defined a draft of the REST resources and how they map to the SIP main messages, accomplishing the first of the goals of this thesis.
In the previous chapters we described the goals of this thesis and the state of the art and defined the functional and non-functional requirements of the System. In this chapter we are going to draw the design of the solution in terms of used frameworks, architectural view, modularity and connectivity of System components.

4.1 Server Architecture

4.1.1 Persistence Layer

According to the functional requirements, the System must permit data persistence. The System will be based in a layered architecture using a middleware to map domain objects to a relational database. The use of a middleware provides us DBMS (database management system) Independence and reduces development and debugging time. A layered architecture using middleware is an example of a multi-tier architecture (or n-tier architecture). The figure 4.1 shows a visual overview of the server including the front ends of the two communication channels defined in previous chapters; and the domain layer. The following subsections will describe the components and connections represented in that figure.

4.1.2 Communication and Domain Layers

In the previous chapter we defined a communication channel that would allow AUC and UAS to exchange notifications asynchronously. This channel will be driven by events that can be triggered in the server by internal threads, other events or as a response to API calls performed by other UAC. The System distributes the logic treatment of domain operations in two different Domain Model Services. The first service, represented in the figure 4.1 by the name of Synchronous Service, is responsible to process (only) requests from API resources and must give an immediate answer to any request. On the other side, the Asynchronous Service can be called by both front ends of the Communication Layer (as response to UAC notifications or events triggered by calls to the REST API) or internally by server-generated events. The data flow of this calls and its responses are represented in the figure 4.1.
4.2 Designing the asynchronous channel

According to the System requirements defined in chapter 3, the technology implementing the asynchronous channel must be accessible using only HTML and client-side scripting. The selected technology to implement this channel are **WebSockets**.

**WebSockets:**

- Designed to be implemented in web browsers and web servers.
- The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455.
- Bi-directional, full-duplex communications channels over a single TCP connection.
4.2. DESIGNING THE ASYNCHRONOUS CHANNEL

We will use a WebSocket connection to transmit tokens between the web browser and the server asynchronous front end. This connection will be requested after user registration by what we defined in the previous chapter as subscription and will be ended just before unregistration. The tables in the figure 4.2 include the defined tokens for the purposes of this thesis, trying (as we did with API resources) to use a SIP-like terminology.

Figure 4.2: SIP-WS token mapping

<table>
<thead>
<tr>
<th>Server name</th>
<th>token</th>
<th>SIP Request / SIP Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td></td>
<td>NOTIFY</td>
<td>Identifier of server notification tokens to a subscriber.</td>
</tr>
<tr>
<td>invite</td>
<td></td>
<td>INVITE</td>
<td>Indicates a client is being invited to participate in a call session.</td>
</tr>
<tr>
<td>subscribed</td>
<td></td>
<td>200 OK</td>
<td>Response of a successful subscription attempt.</td>
</tr>
<tr>
<td>unsubscribed</td>
<td></td>
<td>200 OK</td>
<td>Response of a successful unsubscription attempt.</td>
</tr>
<tr>
<td>accepted</td>
<td></td>
<td>-</td>
<td>Forwarded response to an 'invite' request made to another UAC. This response can be positive or negative.</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td>-</td>
<td>Notification that a certain contact has changed the status. Status changes include registration and unregistration.</td>
</tr>
<tr>
<td>cancel</td>
<td></td>
<td>-</td>
<td>Indicates a client that a session termination request has been send by another user.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Client name</th>
<th>token</th>
<th>SIP Request</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>publish</td>
<td></td>
<td>PUBLISH</td>
<td>Identifier of client notification tokens to the server.</td>
</tr>
<tr>
<td>subscribe</td>
<td></td>
<td>SUBSCRIBE</td>
<td>Subscribes to receive event notifications from the server.</td>
</tr>
<tr>
<td>unsubscribe</td>
<td></td>
<td>SUBSCRIBE</td>
<td>Unsubscribes from event notifications from the server.</td>
</tr>
</tbody>
</table>
4.3 Establishing client-to-client Connections

In chapter 1 we remarked that this project has been developed following an iterative and incremental model. The design of the communication channel between two UAC has been redefined during the implementation after selecting a specific WebSocket framework. Later on this chapter we will describe all frameworks used in the system and its advantages and limitations including the WebSocket implementation. Some of this advantages and limitations have forced changes in the design of the System. Before defining the client-to-client channel creation and illustrate the Call flow we need to advance those changes:

- **Plug-In oriented technology**: In our selected WebSocket implementation almost all higher level functionality is implemented in plug-ins. Following this paradigm, we are going to develop a server-side plug-in that will perform the role of our asynchronous communication front end. We will also define a JavaScript client-side plug-in that will make our client cleaner and modularized.

- **Channel Plug-In**: Our selected WebSocket implementation comes with a set of official plug-ins. Among them, the Channel Plug-In provides us the general purpose server-controlled client-to-client-communication we need to implement a call between two UAC.

- **Static User Management**: WebSocket is a very new technology and almost every implementation is in an early stage of development. Our selected framework doesn’t implement dynamic user management. To overcome this obstacle, our system will implement the asynchronous channel between UAC and UAS using an anonymous WebSocket connection. User authentication will be managed by the Asynchronous Service in the Domain Layer and UAC will need to add their credentials to every notification token.

A connection between two UAC will be implemented using a private WebSocket channel identified with the names of the two end points (or UAC) and secured by a shared key generated by the server and transmitted using the two client-to-server communication channels. The figure 4.3 illustrates the establishment of this connection which conforms the main flow of the main use case of our System. Like the last time we will assume that Bob is already registered and subscribed at the beginning of the flow.

The messages starting by [REST] and its responses are send using the synchronous channel (the RESTful API) and the messages starting by [WS] (and its responses) are transmitted using the asynchronous channel (the WebSocket client-to-server connection). The sequence diagram in the figure 4.3 extends the analysis sequence diagram in figure 3.5 adding the information provided by the tokens described in the previous section. We also included parameters in the calls. This parameters are implemented by **token fields** in the WebSocket communication and **query parameters, path parameters or payloads** in the RESTful API.

Once a UAC (Alice) is registered and subscribed to the System, she can invite registered users (those with a status different than not registered or invisible) to a session. Alice will create the invitation request using the synchronous channel and will get a WebSocket channel id and a shared key with the response. The client must use this data to create a new channel using the **WebSocket Channel plug-in** (in the next chapter we are going to analyze the reasons and implications of giving this responsibility to the client).

The UAS will generate an **invite event** that will be send to Bob’s browser. At this point Bob have three options: accept or reject the request using the RESTful API (PUT to the Call resource) or ignore the request. This last option will leave Alice waiting. Once Bob has accepted the request, he can join Alice’s channel using the channel id and key provided by the **invite event**. At this point both users are subscribed to the same WebSocket channel and
can communicate sending messages to this channel using the WebSocket Channel plug-in publish token. This messages will be broadcasted to all UAC subscribed to the channel. This approach will permit multicast communication in the future. To invite a third UAC to the channel, we have to extend the server to allow invitations to existing calls. Once the UAC has received the invitation, he/she just has to subscribe the channel following the exact steps Bob followed in the unicast use case to be able to publish an receive broadcasted messages.

The sequence diagram in figure 4.3 has been simplified in behalf of readability. The following details have to be considered in the implementation:

- Alice needs to subscribe the channel after creating it. The procedure is the same that Bob follows at the end of the diagram.
- Every single call using either of the server end points requires user credentials.
- Both users have to authenticate themselves in the WebSocket channel before they can publish on it. This can be done using the client-side WebSocket Channel plug-in and the shared key.

4.3.1 Alternative flow: Rejecting connection requests

The figure 4.4 shows an alternative flow were Bob rejects the incoming call. The flow is the same as in figure 4.3 until Bob answers the invitation request.

4.3.2 Terminating a Session

Once the session is created UAC can publish messages in the shared channel. This messages will be broadcasted to any UAC subscribed to the channel. To terminate the session either of the UAC participating on it can send a synchronous request to the RESTful API using DELETE over the Call resource. The server will answer with a 204 NO CONTENT and broadcasting a cancel token to all the participants in the call. The client will unsubscribe the channel after receiving the cancel token and the responsibility to close it lies with the client that created it. The figure 4.5 contains a sequence diagram illustrating the steps performed to end a session on Bob’s request.
Figure 4.3: Design connection establishment flow sequence diagram
Figure 4.4: Design call rejection alternative flow sequence diagram
Figure 4.5: Design call termination flow sequence diagram
4.4 Data Model

The concepts of this thesis are very simple. The figure 4.6 represents the Unified Modeling Language (UML) conceptual data model of the System. The following details may be interesting in order to understand and extend the model:

- Associating a Call to a Contact simplifies the Data Model guaranteeing two restrictions of our System:
  - 
  - Direction of the Call: A Contact is associated with two users but it is not a symmetric relation (User A having User B as a contact doesn’t force User B to have User A as a contact). Associating a Call with a Contact gives us an immediate knowledge of who are the source and target of the Call.
  - Call scope restriction: Users can only Call their own contacts.

NOTE: This model would need to be extended to permit multicast sessions.

- The Call representation is coupling to a channel-based session.
- A Call has currently only two states: started or not started. This have proved to be enough for the scope of this thesis but may need to be extended in the future.
- The System includes Role management which allows the API to define Administrator features. The RESTful API structure has been designed have two domains of resources: http://www.example.com/api/user and http://www.example.com/api/admin. A third role named client will be created to allow anonymous calls such as User creation (POST to the User resource).

Figure 4.6: Conceptual data model
4.5 Conclusions

In this chapter we defined the architecture and data model of the System. We also extended the specification choosing a technology to implement the asynchronous channel and sessions. The SIP event-driven messages have been mapped to WebSocket tokens accomplishing the second goal of this project and the sequence diagram of the previous chapter has been completed with technology-dependant information.
Chapter 5

Implementation

This chapter describes the technologies used on this project and their place in the architecture defined in previous chapters as well as their interconnection and integration in the System. We will also analyze the main issues found during the implementation process and how they were solved.

5.1 Used Technologies

5.1.1 Programming Environment

Java was chosen as the programming language to implement the System server-side for being a simple, object-oriented language with available implementations of all the technologies required by this thesis.

The server was developed using Eclipse and the client was developed on Geany. This memory was written entirely in LaTeX using the package provided to thesis students by Barcelona School of Informatics (FIB). The package has been slightly modified to be translated into English and adapted to the thesis structure defined by the Faculty of Informatics and Information Technologies (FIIT) in Bratislava. The graphics and diagrams of this memory have been created using yEd, Dia and GIMP.

5.1.2 Jersey

"Jersey is the open source, production quality, JAX-RS (JSR 311) Reference Implementation for building RESTful Web services. But, it is also more than the Reference Implementation. Jersey provides an API so that developers may extend Jersey to suit their needs."

Jersey is a powerful Java API for RESTful Web Services (JAX-RS) implementation. JAX-RS is a Java programming language API that provides support in creating web services according to the REST architectural style. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints.

The author of this thesis had previous experience using JAX-RS and found that Jersey more mature and stable than other implementations such as Apache Wink. The learning curve to use this technology was null since is using the same annotation-based API (JAX-RS).
5.1.3 MySQL

MySQL is the world’s most used RDBMS (relational database management system). It is an open source technology with a full-featured database management system.

5.1.4 Hibernate

The selected Middleware to implement the persistence layer is Hibernate. It is the most used Object-relational mapping (ORM) Java library. Together with Maven it provides us an automatic plain old Java object (POJO) generation that will simplify our Domain Layer implementation. We are going to analyze automatic code generation in the next section.

5.1.5 JAXB

We will use Java Architecture for XML Binding (JAXB) to bind XML documents and manage the payloads and responses of our RESTful API. As we will explain in the next section, the purpose of this is to automatize data transfer object (DTO) generations using XSD Schemes. Once XML documents are converted into Java classes, they can be converted to XML again (or even JSON) by Jersey without any additional development cost.

5.1.6 jWebSocket

jWebSocket was selected as both the Java WebSocket server and JavaScript client-side library of the project. It was selected because it is a pure Java/JavaScript open source implementation, that can be integrated with Tomcat and Maven. It is in an early stage of development (version 1.0 was used in this project) and has been the cause of the longest delays in the project implementation due to its incomplete (and sometimes deprecated) documentation and its difficulty to be integrated in the dependency management system. The development issues regarding this library will be analyzed in the next section.

Besides that, it is a complete, functional and flexible implementation with a huge set of extensions (and examples) and a very useful plug-in focused structure in both the server and client-side.

5.1.7 Tomcat

"Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages technologies."

We choose Apache Tomcat to deploy our server because it is a pure Java environment with which the author of this thesis have had previous experience. Tomcat has proved to be easily connected to MySQL to use dynamic user management and authentication. It was also easy to use for Web application deployment using the tomcat-maven-plugin and not so easy but still good enough with the jWebSocket Server.

5.1.8 Maven

"Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project’s build, reporting and documentation from a central piece of information."
Maven will be very helpful to manage the dependencies of all the frameworks described above. Maven does also provide a very wide and useful set of plug-ins that will simplify our development process. We are going to use those plug-ins to compile the project, deploy and undeploy it into our Tomcat, generate and dump the database and generate automatic code (using Hibernate and JAXB). Maven does also provide a modular project structure that will help us to distribute our client and server into two separate webapps and deploy them simultaneously.

5.1.9 GIT
We used Git for revision control and source code management. The project has been developed in a local repository by a single developer but Git was very useful to keep track of changes, keep a history of the development project and rollback changes if necessary.

5.1.10 jQuery
We used jQuery to simplify calls to the RESTful API from the JavaScript client. It also proved to be very useful to manipulate HTML input elements. An extended description of the client implementation can be found later in this chapter.
5.2 Server Implementation

5.2.1 Communication Layer: The RESTful API

The RESTful API implements the synchronous channel of our system using the Jersey JAX-RS API library. Basically every resource is implemented by a single class and HTTP methods, MIME types and parameters are defined using annotations. The following code snippet is an example of how a resource implements the GET method:

```java
@GET
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@RolesAllowed({Roles.ADMIN, Roles.USER})
public JAXBElement<User> getUser(@Context final SecurityContext context) {
    String username = context.getUserPrincipal().getName();
    User result = createUserDto(SyncService.getInstance().getUser(username));
    return factory.createUser(result);
}
```

The path of the resource is defined as an attribute of the class UserResource (which includes the previous method). This method basically obtains the id of the user who performed the request from the security context and delegates the responsibility of retrieving the user information to the Synchronous Service. Once the service has served the response, this resource will convert it into a DTO and then wrap it in a JAXBElement representing the XML or JSON response. The annotations on the method define which MIME types and roles are allowed. Jersey will automatically manage any request not accomplishing this conditions.

As we can see in the code, in our implementation we included object transformation (from DTO to domain POJO and POJO to DTO) at resource level. In the section Towards automatic model generation we are going to see the difference between POJOs and DTOs in our System. The appendix A includes a complete description of all the resources, its use, parameters and responses.

5.2.2 Communication Layer: The jWebSocket Plugin

"In jWebSocket almost all higher level functionality is implemented in plug-ins. Therefor the jWebSocket Server provides the required managing capabilities. One of these capabilities is the plug-in chain. In this chain multiple plug-ins are arranged in a certain order. The order of the plug-ins can be arbitrarily changed and new plug-ins can easily be appended or inserted. Basically a plug-in can process an almost unlimited number of client commands by interpreting the various dynamic data fields of the tokens. A complex plug-in can consist of multiple classes in a separate package which can either be embedded or dynamically linked at run-time as a separate and even distributable .jar file.

This concept allows you to easily implement and distribute new WebSocket services according to your and your customers needs based on what jWebSocket already provides."

To add our own plugin to the chain during execution time we will create a context listener in the servlet and instantiate our plug-in during the context initialization (as shown in figure 5.3).
In the code of figure 5.3 we can also see that we are linking our jWebSocket Token Server to the Asynchronous Domain Model Service. This will allow the service to implement high level domain functionalities of our asynchronous channel delegating low level calls to the jWebSocket server.

The Channel Plug-In that we are going to use to implement client-to-client sessions is automatically added to the plug-in chain by default in the jWebSocket configuration file.

In previous sections we introduced that jWebSocket is a new technology and its use has delayed the development process. The following list summarizes the main difficulties found while working with this library:

- **Browser Compatibility**: The last release of the library (1.0) is not compatible with the latest version of the main browsers. A newer night build (1.0-nb11024) should be used instead.

- **Limited Repository (versions)**: The jWebSocket official repository only includes major releases. Due to the previous issue, including jWebSocket in Maven’s dependency management was not possible. After testing, researching and some try and error, our final approach was using a Bundle (containing the jWebSocket Server and all its dependencies) provided in the library’s website. The bundle was used in our webapp during compilation time and also moved into Tomcat common lib folder. The inclusion of external dependencies in the bundle created new dependency conflicts with other libraries in the project that had to be fixed using exclusions.
CHAPTER 5. IMPLEMENTATION

Figure 5.3: Adding a jWebSocket plug-in to the chain

```java
public class JWSContextListener implements ServletContextListener {
    @Override
    public void contextInitialized(final ServletContextEvent sce) {
        // start the jWebSocket server sub system
        JWebSocketFactory.start();
        // add b2b plug-in to the plug-in chain of the jWebSocket Server
        TokenServer ts = (TokenServer) JWebSocketFactory.getServer(DEFAULT_SERVER);
        B2BPlugIn b2bPlugin = new B2BPlugIn();
        ts.getPlugInChain().addPlugIn(b2bPlugin);
        // share JWebSocketServer with Service
        AsyncService.getInstance().setTokenServer(ts);
    }

    @Override
    public void contextDestroyed(final ServletContextEvent sce) {
        JWebSocketFactory.stop();
    }
}
```

- **Limited Repository (plug-ins):** The plug-in implementations are not available in the official repository. The Channel Plug-In is automatically added to the plug-in chain using the default jWebSocket configuration file but its implementation is not provided in the bundle and has to be manually copied to the Tomcat common lib directory.

- **Documentation:** The project documentation is very structured and well defined but it is not finished and has important gaps. It has numerous code examples but some of them are implemented using previous versions and that turned out to be very confusing since the library is evolving and has significant changes between versions.

- **Static user management:** That was not actually a problem but a missing feature. We described our approach to solve this gap in previous sections.

- **Java Client not released:** The Java Script client is fully implemented and perfectly suitable for our System but sometimes we require native use from Java code. This would allow us to restrict channel creation/termination to a privileged user and avoid a security issue: With the current implementation any user can block potential sessions by creating channels (without performing the call creation) and giving them a legit name in the server name code. This practice would make impossible for other users to create those channels blocking communication. The Java client has announced to be released with the following versions.

5.2.3 Domain Model Services

in Domain-Driven Design, Domain Services are singleton classes including any domain operation that does not conceptually belong to any object. In our System, we defined two separate services to encapsulate those domain operations. As we commented in previous chapters, the Synchronous Service will include treatment of requests from API resources and the Synchronous Service will treat requests from the jWebSocket plug-in as well as events generated from synchronous calls (see figure 4.1 in chapter 4).
5.2. SERVER IMPLEMENTATION

5.2.4 Towards automatic model generation

Using the hibernate3-maven-plugin and defining several actions during the source-generation phase of the Maven Build Lifecycle we can generate all the Domain Objects and data access object (DAO) of the System based on our database schema by running:

```
mvn generate-sources
```

A DAO is an object that provides an abstract interface to the database or persistence mechanism, providing some specific operations without exposing details of the database. This objects are generated from a template (named daohome.ftl in our project) that we slightly modified to fit our System requirements.

In the same phase of the Maven Build Lifecycle we are using the maven-jaxb2-plugin to generate DTO objects automatically from XSD schemes. Figure 5.4 shows the schema to generate and validate the user (and user list) DTO.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
jxb:version="1.0">
  <xsd:element name="user" type="user"/>
  <xsd:element name="userList" type="userList"/>
  <xsd:complexType name="user">
    <xsd:all>
      <xsd:element name="username" type="xsd:string"/>
      <xsd:element name="email" type="xsd:string"/>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="surname" type="xsd:string"/>
      <xsd:element name="locale" type="xsd:string"/>
      <xsd:element name="password" type="xsd:string" minOccurs="0"/>
      <xsd:element name="registrationDate" type="xsd:date" minOccurs="0"/>
      <xsd:element name="status" type="xsd:string"/>
    </xsd:all>
  </xsd:complexType>
  <xsd:complexType name="userList">
    <xsd:sequence>
      <xsd:element name="user" type="user" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Adding this two code generation mechanisms simultaneously has not been easy. We had to deal with extensive configuration processes, dependency conflicts, template modification and mappings. Nevertheless it has proved to be very useful to speed the development process. This is the process that was followed to add a new domain object to our System:

- Add a table definition to the SQL schema of the project. Here we should remark that the database is also created in the same Maven Build Lifecycle phase.
- Create a XSD schema defining the same concept.
- Run mvn generate-sources.
- Now we have the POJO, DAO and DTO Java classes of our conceptual object in the project’s classpath.
• To connect the domain with the other two layers we would proceed creating a Repository (see next section) to link Service and DAO and define methods in the communication layer to convert from POJO to DTO and vice-versa.

5.2.5 Persistence Layer

In Domain-Driven Design, Repositories are encapsulated centralizations of CRUD functions. Our Persistence Layer is composed by only two kinds of objects: DAOs and repositories. DAOs encapsulate Hibernate calls as we commented in the previous section. Repositories manage database sessions and transactions while encapsulating CRUD operations and customized SQL statements.
5.3. CLIENT IMPLEMENTATION

The client of our System is implemented using pure HTML5 and JavaScript code. As in the server, it separates the synchronous and asynchronous channels into different modules. The file synCalls.js depends on jQuery to perform calls to the RESTful API.

All the calls are made by the callService() method using global variables to define the state of every call. The figure 5.5 shows an example of how the client performs a request to the RESTful web service.

Figure 5.5: Client REST request example

```javascript
function callService()
{
    log("[REST Call] " + varType + " " + varUrl, '#00FF00');
    if (varData) log("[REST Call Payload] " + varData, '#0000AA');
    $.ajax(
        type: varType, //GET or POST or PUT or DELETE verb
        url: varUrl, //Location of the service
        data: varData, //Data sent to server
        contentType: varContentType, // content type sent to server
        processData: varDataProcess, //Expected data format from server
        success: varCallFunc, //On Successful service call
        error: function (request, status, error){ //On Error
            if (request.status != 401)
                log("[Response error] " + request.responseText + " (" + request.status + " " + error + ")", '#FF0000');
            else
                {
                    jws.$('register').value = 'Register';
                    alert('Invalid username or password');
                }
            if (request.responseText.length < 50) alert(request.responseText);
        }
    ) ;
}

function unregister()
{
    //Unregister from API
    varType = 'DELETE';
    varUrl = rootURL + '/user/register';
    varData = null;
    varDataProcess = false;
    varCallFunc = function(msg) {
        unsubscribeCurrentUser();
        jws.$('statusViewer').src = 'images/status_notRegistered.png';
        jws.$('register').value = 'Register';
        jws.$('apiUser').innerHTML = '';
        jws.$('contactList').innerHTML = '';
    }
    callService();
}
```

The function stored in the variable varCallFunc defines the expected behavior if the calls ends successfully. If the API responds with an error, the callService defines the default behavior on a different function.

The file asynCalls.js depends on the jWebSocket library, the client-side ChannelPlugIn and our own client-side plug-in to send requests and process tokens coming through the
asynchronous channel of the System. The system client-side plug-in is implemented in the file\
\textit{jwsB2BPlugIn.js} and defines high level methods to send requests to the server using \textit{tokens}\
and the default behavior of the client processing incoming tokens.

5.4 Conclusions

We selected and described the technologies and the main details of the implementation process\
together with any decision taken during it. The implementation was the \textbf{third of the goals of this project}. The \textbf{forth goal} (creating a \textit{Framework}) depends on the System API, document-

ation and the way it is used in the future. The \textbf{last goal} will be faced using the simplest data representation to test sessions: \textit{plain text}. 
Chapter 6

Testing

One of the goals of this thesis was to define and use a simple illustrative data representation to test the communication between clients once the session is established. To accomplish this goal we added a chat-like area to our client. This window will show the messages extracted from any token coming from a channel where the user is subscribed. This means showing broadcasted messages only. In the testing scenarios we are going to use plain text, leaving the responsibility to identify users or sending complex data to any protocol working on top of our communication channel.

6.1 Testing Scenarios

We are going to perform two different tests using the web client and following the steps as a real user would. Our first test will be based on a single unicast communication and will test how to create a call, accept it, reject it and finally how to terminate the call and create it again. A second test will include several simultaneous unicast communications.

We are also going to perform some direct calls to the API to test the secondary functions (those which are not covered in the client).

6.1.1 Alice and Bob: Unicast communication

In our first test, Alice wants to send an important message to Bob. Bob is a very busy person and will not be able to answer the phone the first time. After this, Alice will try again, they will exchange a few messages and Bob will excuse himself and end the conversation. The figure 6.1 summarizes the test.

6.1.2 Alice, Bob and Admin: Simultaneous unicast communications

In this second test, Bob will call Admin to ask some questions about the service. Meanwhile, Alice will call Bob and, he will tell her that he is talking to Admin. Alice surprised by the service decides to call Admin too. The figure 6.2 summarizes the communication.
Figure 6.1: Simple unicast test description

Call

Second attempt

Say: ‘Hi, I'm Alice’

Say: ‘Hi!’

Say: "I just wanted to remind you of our appointment tomorrow..."

Say: "I won't forget, thank you Alice ;)"

End Call

Call

Say: "Remember the flowers!"

End Call
6.1. TESTING SCENARIOS

Figure 6.2: Simultaneous unicast test description
6.1.3 API tests

The RESTful API will be tested using **curl** (a command line tool for transferring data with URL syntax) and **XHR POSTER** (a Google Chrome RESTful client extension). This tests won’t be described in this memory because of its extension but can be easily reproduced following the *API Reference Manual* in appendix A. We will test the following actions with correct and incorrect values:

- Creating, searching, updating and deleting **users**.

- Adding, removing and retrieving one or all the **contacts**.

---

**Figure 6.3: XHR Poster screenshot**
6.2 Test Results

6.2.1 Single Unicast communication results

Following the steps defined in figure 6.1 we obtained successful results. Figure 6.4 shows the state of both clients at the end of the test.

Figure 6.4: Clients after single unicast test
6.2.2 Simultaneous unicast communications results

This test, illustrated in figure 6.2 consists in a triangulation of unicast connections between Alice, Bob and Admin. The test worked successfully since each user is only receiving the messages from its own connections. Ending, rejecting and restarting sessions worked successfully too.

6.3 Conclusions

After some smoke testing, the System seems functional and stable. It also proved to be compatible with different web browsers. The System was tested using Mozilla Firefox, Google Chrome (and Chromium) and Opera. Using different browsers helped us fixing minor missing details in both the client and the API giving the System a little bit of robustness.
Conclusions

This chapter exposes the conclusions regarding the goals defined in chapter 1. It also includes comments about future work, the current state of the project and acquired knowledge.

7.1 Goals

The original goals of the thesis were:

- Define a simple RESTful SIP-like communication protocol to negotiate the connection.
- Use the previously defined API to establish a client-to-client connection over standard WebSockets.
- Implement a simple solution of the server side application and a lightweight web client to test it.
- Turn this solution into a framework that could help future research in communications on top of it.
- Define and use a simple illustrative data representation to test the communication between clients once the session is established.

At this point we can observe:

- The protocol has a SIP-like notation. It is distant from following the SIP protocol but should be easy to understand by anyone familiarized with SIP.
- The API plus an extra asynchronous communication channel permit users to negotiate sessions.
- The solution pretends to be simple and proved to be functional but session establishment may be considered "tricky" using two different server front ends and delegating to much responsibility in the client (improving this is considered the most critical potential future task).
- The solution may or may not be used for future research depending on its usability, quality and evaluation.
• The web client uses a visual frame representing the information within the tokens as plain text. This allows us to test the System using a chat-like interface.

The main goals have been accomplished successfully.

7.2 Future Work

We can distribute future work into two categories: improvements and new features. The improvements were commented during this memory:

• **Delegate channel creation and deletion to the Server.** This may be done extending the WebSocket server or using the jWebSocket Java Client (not released yet) with special credentials and restricting regular users.

• **Delegate user management in asynchronous channel to the WebSocket Server.** This may be possible if jWebSocket is extended to use dynamic user management from a database.

• **Allow multicast sessions.** This requires vertical design changes but is technically possible using the chosen framework and performing some domain refactoring.

Possible new features based on VoIP common uses may be:

• Call history

• Tunes

• Multimedia streams

• Admin functions
7.3 Used and Acquired Knowledge

Previous knowledge in some of the used technologies has been a critical factor to finish this thesis in its short time frame and has been a deciding factor to choose them. This technologies include Java, Hibernate, MySQL, REST or Tomcat. jWebSocket on the other side has delayed the development since it required extra research. The reasons of such delays are explained in detail in chapter 5.

Acquired knowledge includes automated model generation, use of WebSocket, web development (mainly use of JavaScript and jQuery). We have to remark the acquired theoretical knowledge about SIP and WebRTC.

We can also remark that this project required a low charge of programming and a big focus on design, framework integration, configuration and documentation.

The project couldn’t have been tested until both the synchronous and asynchronous channel were implemented. This delayed the testing phase because of the previously commented details about jWebSocket integration.

The design of the session establishment without pooling the server has been an interesting challenge.

This memory and the final presentation have been completely created using LaTeX which required extra learning but has proved to be very useful and given good results in short time.

This is the author’s first thesis and has helped to learn about project planning and management of time and resources; and use of written English.

7.4 Acknowledgement

It would not be fair to end this memory without thanking everybody who had made it possible.

First of all I would like to thank the thesis Supervisor, Ján Murányi who has guided me during all the phases of the project, suggesting paths, helping to take decisions and filling my lack of knowledge in the area.

I would also like to thank Dr. Ivan Kotuliak and the Institute of Computer Systems and Networks of the Slovak University of Technology in Bratislava for giving me the opportunity to present my thesis in the frame of their department and research.

I want to thank the city of Bratislava and its people for their kindness. They provided a very pleasant and agreeable environment to work.

Thanks also to my roommates Quentin Bourrely and Vincent Bohly who daily listened and opined about my doubts; and some others who did it from Spain.

Special thanks to my parents who have always supported me on the choice of this profession.
CHAPTER 7. CONCLUSIONS
Appendix A

API Reference Manual

A.1 User Resource

A.1.1 Main URL

This class operates mainly over the Resource URI: /api/user

A.1.2 Retrieve the current user information

Synopsis: GET http://example.com/api/user

Roles Needed: user or admin

Request Headers: Accept:application/xml, Accept:application/json

Request Parameters: -

Request Body Entity: -

Response Message Entity: User

Response Message Examples:

Figure A.1: GET User XML example

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <username>alice</username>
  <email>alice@b2b.sk</email>
  <name>Alice</name>
  <surname>Test</surname>
  <locale>en_US</locale>
  <password>6384e2b2184bcbf58eccf10ca7a6563c</password>
  <registrationDate>2012-05-17T02:00:00</registrationDate>
  <status>notRegistered</status>
</user>
```

Figure A.2: GET user JSON example

```json
{
  "username": "alice",
  "email": "alice@b2b.sk",
  "name": "Alice",
  "surname": "Test",
  "locale": "en_US",
  "password": "6384e2b2184bcbf58eccf10ca7a6563c",
  "registrationDate": "2012-05-17T02:00:00",
  "status": "notRegistered"
}
```
APPENDIX A. API REFERENCE MANUAL

A.1.3 Search user by username

Synopsis: GET http://example.com/api/user/search
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required?</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>Yes</td>
<td>No</td>
<td>Id of the user to search</td>
</tr>
</tbody>
</table>

Request Body Entity: -
Response Message Entity: User
Response Message Examples:

Figure A.3: Search User XML example

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <username>bob</username>
  <email>bob@b2b.sk</email>
  <name>Bob</name>
  <surname>Test</surname>
  <registrationDate>2012-05-17+02:00</registrationDate>
  <status>notRegistered</status>
</user>
```

Figure A.4: Search user JSON example

```json
{"username":"bob","email":"bob@b2b.sk","name":"Bob","surname":"Test","registrationDate":"2012-05-17+02:00","status":"notRegistered"}
```

A.1.4 Create a new user

Synopsis: POST http://example.com/api/user
Roles Needed: client
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Request Body Entity: User
Response Message Entity: User
Required Fields: username, password, email, name, surname.
Restrictions: email format, locale format, status cannot be set, registration date should not be defined (in this case, the system will not raise error but ignore the field).

A.1.5 Update the current user information

Synopsis: UPDATE http://example.com/api/user
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Request Body Entity: User
Response Message Entity: User
Required Fields: username, password, email, name, surname.

Restrictions: email format, locale format, username cannot be changed, status cannot be changed, registration date cannot be changed (in this case, the system will not raise error but ignore the field).

A.1.6 Delete the current user

Synopsis: DELETE http://example.com/api/user

Roles Needed: user
Request Headers: -
Request Parameters: -
Request Body Entity: -
Response Message Entity: -
APPENDIX A. API REFERENCE MANUAL

A.2 Registration Resource

A.2.1 Main URL
This class operates mainly over the Resource URI: /api/user/register

A.2.2 Register the current user
Synopsis: POST http://example.com/api/user/register
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required?</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>No</td>
<td>&quot;available&quot;</td>
<td>Original user status on registration</td>
</tr>
</tbody>
</table>

Request Body Entity: -
Response Message Entity: User
Response Message Examples:

Figure A.5: Register XML example

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<user>
  <username>alice</username>
  <email>alice@b2b.sk</email>
  <name>Alice</name>
  <surname>Test</surname>
  <locale>en_US</locale>
  <password>6384e2b2184bcbf58eccf10ca7a6563c</password>
  <registrationDate>2012-05-17+02:00</registrationDate>
  <status>notRegistered</status>
</user>
```

Figure A.6: Register JSON example

```json
{ "username":"alice", "email":"alice@b2b.sk", "name":"Alice", "surname":"Test", "locale":"en_US", "password":"6384e2b2184bcbf58eccf10ca7a6563c", "registrationDate":"2012-05-17+02:00", "status":"notRegistered" }
```

Restrictions: status cannot be not_registered or invalid value.

A.2.3 Unregister the current user
Synopsis: DELETE http://example.com/api/user/register
Roles Needed: user or admin
Request Headers: -
Request Parameters: -
Request Body Entity: -
Response Message Entity: -
A.3 Update Resource

A.3.1 Main URL
This class operates mainly over the Resource URI: /api/user/update

A.3.2 Update the current user registration

Synopsis: PUT http://example.com/api/user/update
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required?</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>No</td>
<td>'available'</td>
<td>New user status on update</td>
</tr>
</tbody>
</table>

Request Body Entity: -
Response Message Entity: User
Response Message Examples:

Figure A.7: Update XML example

```xml
<user>
    <username>alice</username>
    <email>alice@b2b.sk</email>
    <name>Alice</name>
    <surname>Test</surname>
    <locale>en_US</locale>
    <password>6384e2b2184bcbf58eccf10ca7a6563c</password>
    <registrationDate>2012-05-17+02:00</registrationDate>
    <status>notRegistered</status>
</user>
```

Figure A.8: Update JSON example

```json
{
    "username":"alice",
    "email":"alice@b2b.sk",
    "name":"Alice",
    "surname":"Test",
    "locale":"en_US",
    "password":"6384e2b2184bcbf58eccf10ca7a6563c",
    "registrationDate":"2012-05-17+02:00",
    "status":"notRegistered"
}
```

Restrictions: status cannot be not_registered or invalid value.
A.4 Contacts Resource

A.4.1 Main URL
This class operates mainly over the Resource URI: /api/user/contacts

A.4.2 Retrieve the list of contacts of a user
Synopsis: GET http://example.com/api/user/contacts
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Response Message Entity: UserList
Response Message Examples:

Figure A.9: GET contacts XML example

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<userList>
  <user>
    <username>admin</username>
    <email>admin@b2b.sk</email>
    <registrationDate>2012-05-17+02:00</registrationDate>
    <status>notRegistered</status>
  </user>
  <user>
    <username>bob</username>
    <email>bob@b2b.sk</email>
    <name>Bob</name>
    <surname>Test</surname>
    <registrationDate>2012-05-17+02:00</registrationDate>
    <status>notRegistered</status>
  </user>
</userList>
```

Figure A.10: GET contacts JSON example

```json
{ "user" : [ { "username" : "admin" , "email" : "admin@b2b.sk" , "name" : "admin" , "surname" : "admin" , "registrationDate" : "2012-05-17+02:00" , "status" : "notRegistered" } , { "username" : "bob" , "email" : "bob@b2b.sk" , "name" : "Bob" , "surname" : "Test" , "registrationDate" : "2012-05-17+02:00" , "status" : "notRegistered" } ] }
```

A.4.3 Retrieve a single contact
Synopsis: GET http://example.com/api/user/contacts/{username}
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Request Body Entity: -
Response Message Entity: User
Restrictions: valid username.
A.4.4 Add a contact to the contact list

Synopsis: POST http://example.com/api/user/contacts
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Request Body Entity: User
Response Message Entity: -
Restrictions: User in request body must be a valid user (use User Resource to search it).

A.4.5 Delete a contact

Synopsis: DELETE http://example.com/api/user/contacts/{username}
Roles Needed: user or admin
Request Headers: Accept:application/xml, Accept:application/json
Request Parameters: -
Request Body Entity: -
Response Message Entity: -
Restrictions: valid username.
A.5 Call Resource

A.5.1 Main URL

This class operates mainly over the Resource URI: /api/user/call

A.5.2 Invite another user to a call

Synopsis: POST http://example.com/api/user/call

Roles Needed: user or admin

Request Headers: Accept:application/xml, Accept:application/json

Request Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required?</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contact</td>
<td>Yes</td>
<td>No</td>
<td>Username of the contact to call</td>
</tr>
</tbody>
</table>

Request Body Entity: -

Response Message Entity: Call

Response Message Examples:

Figure A.11: Call XML example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<call>
  <username>alice</username>
  <contactname>bob</contactname>
  <channelname>b2b:alice2bob</channelname>
  <channelKey>secret</channelKey>
</call>
```

Figure A.12: Call JSON example

```
{"username":"alice","contactname":"bob","channelname":"b2b:alice2bob","channelKey":"secret"}
```

Restrictions: contact must be registered, call not started yet.

A.5.3 Answer an incoming call

Synopsis: PUT http://example.com/api/user/call/{id}

Roles Needed: user or admin

Request Headers: -

Request Parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Required?</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>accept</td>
<td>Yes</td>
<td>No</td>
<td>Boolean value accepting/rejecting the call</td>
</tr>
</tbody>
</table>

Request Body Entity: -

Response Message Entity: -

Restrictions: call must exist.
A.5.4 End a call

**Synopsis:** DELETE http://example.com/api/user/call/{id}
- **Roles Needed:** user or admin
- **Request Headers:** -
- **Request Parameters:** -
- **Request Body Entity:** -
- **Response Message Entity:** -
- **Restrictions:** call must exist.
Appendix B

Installation Manual

In this appendix we will list the required steps to build the project in Ubuntu 11.10.

**NOTE:** The project requires a previous installation of Java 6 in the system.

**B.1 Install Tomcat**

1. $ wget http://tux.rainside.sk/apache/tomcat/tomcat-6/v6.0.35/bin/apache-tomcat-6.0.35.tar.gz
2. $ tar xzvf apache-tomcat-6.0.35.tar.gz
3. $ sudo mv apache-tomcat-6.0.35 /usr/local/tomcat

Change user for your username (e.g. fmontserrat:fmontserrat):

1. $ sudo chown -R user:user /usr/local/tomcat

**B.2 Configure Tomcat**

**B.2.1 Add User Realm**

1. $ gedit /usr/local/tomcat/conf/server.xml

Replace default realm if any.

Figure B.1: server.xml

```
<!---- Realm className="org.apache.catalina.realm.UserDatabaseRealm" resourceName="UserDatabase"/ -->
<Realm className="org.apache.catalina.realm.JDBCRealm"
      driverName="com.mysql.jdbc.Driver"
      connectionURL="jdbc:mysql://localhost:3306/b2b"
      connectionName="root" connectionPassword="root"
      userTableName="user" userNameCol="username"
      userCredCol="password" digest="MD5"
      userRoleTable="role" roleNameCol="rolename"
      resourceName="b2b"/>
```
B.2.2 Add Tomcat Manager User

```bash
$ gedit /usr/local/tomcat/conf/tomcat-users.xml
```

Figure B.2: tomcat-users.xml

```xml
<tomcat-users>
  <role rolename="manager"/>
  <user username="b2b" password="b2b" roles="manager"/>
</tomcat-users>
```
B.3 Install Maven

$ sudo apt-get install maven2

B.4 Configure Maven

B.4.1 Define Maven Settings

$ mkdir ~/.m2
$ gedit ~/.m2/settings.xml

Figure B.3: settings.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <pluginGroups>
  </pluginGroups>
  <proxies>
  </proxies>
  <servers>
    <server>
      <id>mytomcat</id>
      <username>b2b</username>
      <password>b2b</password>
    </server>
  </servers>
</settings>
```
B.5 Install MySQL

```bash
$ sudo apt-get install mysql-server libmysql-java
```

The installer will ask for a root password. Set it as `root` to skip future configuration steps.

B.6 Configure MySQL

B.6.1 Add MySQL Connector to Tomcat

```bash
$ cp ~/.m2/repository/mysql/mysql-connector-java/5.1.18/mysql-connector-java-5.1.18.jar
/usr/local/tomcat/lib/
```

**NOTE:** if the file is not in repository already, compile the project with:

```bash
$ cd {B2BPATH}
$ mvn clean compile
```

This will automatically download all dependencies (it may take several minutes).
B.7  INSTALL JWEBSOCKET

B.7 Install jWebSocket

B.7.1 Install Server

```bash
$ cp {B2BPATH}/b2b-api/src/main/provided/jWebSocketServer-Bundle-1.0.jar /usr/local/tomcat/lib
$ mv {B2BPATH}/b2b-api/src/main/provided/jwebsocket /usr/local/tomcat/conf/
$ cp {B2BPATH}/b2b-api/src/main/provided/servlet-api-3.0.20100224.jar /usr/local/tomcat/lib
```

B.7.2 Install Channel PlugIn

```bash
$ mv {B2BPATH}/b2b-api/src/main/provided/jWebSocketChannelPlugin-1.0.jar /usr/local/tomcat/lib
```

B.7.3 Set Environment Variables

```bash
$ gedit ~/.bashrc
```

```bash
export CATALINA_HOME=/usr/local/tomcat
export JWEB_SOCKET_HOME=$CATALINA_HOME/conf/jwebsocket
```

```bash
$ source ~/.bashrc
```

Figure B.4: .bashrc
B.8 Deploy Project

B.8.1 Run Tomcat

Make sure the database schema (which includes default users) is created before starting Tomcat.

1 \$ mvn generate-sources
2 \$ sudo /usr/local/tomcat/bin/catalina.sh run

B.8.2 Compile and Deploy Project

If you are developing make sure automatic generated sources are up to date:

1 \$ mvn generate-sources

First deploy:

1 \$ mvn clean tomcat:deploy

Redeploy:

1 \$ mvn clean tomcat:redeploy

B.8.3 Test it!

1 \$ firefox http://localhost:8080/api/user

This should ask for Basic Authentication. Write ‘alice’ as username and password and the API should be returning the user information in XML.

Or opening the client:

1 http://localhost:8080/client/
Glossary

CRUD
Create, read, update and delete (CRUD) are the four basic functions of persistent storage. 36

DAO
A data access object (DAO) is an object that provides an abstract interface to some type of database or persistence mechanism, providing some specific operations without exposing details of the database. It provides a mapping from application calls to the persistence layer. This isolation separates the concerns of what data accesses the application needs, in terms of domain-specific objects and data types (the public interface of the DAO), and how these needs can be satisfied with a specific DBMS, database schema, etc. (the implementation of the DAO). 34–36

DBMS
A database management system (DBMS) is a software package with computer programs that control the creation, maintenance, and use of a database. It allows organizations to conveniently develop databases for various applications by database administrators (DBAs) and other specialists. 19

DELETE
In HTTP, deletes the specified resource. 13, 14, 23

Domain Model Services
In DDD, a Service is used when an operation does not conceptually belong to any object. Following the natural contours of the problem, you can implement these operations in services. 19

Domain-Driven Design
Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. 34, 36

DTO
Data transfer object (DTO), is a design pattern used to transfer data between software application subsystems. DTOs are often used in conjunction with data access objects to retrieve data from a database. The difference between data transfer objects and business
objects or data access objects is that a DTO does not have any behavior except for storage and retrieval of its own data (accessors and mutators). 30, 32, 35

**GET**

In HTTP, GET Requests a representation of the specified resource. Requests using GET should only retrieve data and should have no other effect. (This is also true of some other HTTP methods.) The W3C has published guidance principles on this distinction, saying, "Web application design should be informed by the above principles, but also by the relevant limitations.". 13

**HTML5**

HTML5 is a markup language for structuring and presenting content for the World Wide Web, and is a core technology of the Internet originally proposed by Opera Software. It is the fifth revision of the HTML standard (created in 1990 and standardized as HTML4 as of 1997) and, as of May 2012, is still under development. Its core aims have been to improve the language with support for the latest multimedia while keeping it easily readable by humans and consistently understood by computers and devices (web browsers, parsers, etc.). HTML5 is intended to subsume not only HTML 4, but XHTML 1 and DOM Level 2 HTML as well. 7, 11, 37

**HTTP**

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web. Hypertext is a multi-linear set of objects, building a network by using logical links (the so called hyperlinks) between the nodes (e.g. text or words). HTTP is the protocol to exchange or transfer hypertext. 9, 13

**IETF**

The Internet Engineering Task Force (IETF) is a large open international community of network designers, operators, vendors, and researchers concerned with the evolution of the Internet architecture and the smooth operation of the Internet. It is open to any interested individual. 11

**JavaScript**

JavaScript (sometimes abbreviated JS) is a prototype-based scripting language that is dynamic, weakly typed, general purpose programming language and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. 11, 37

**JAXB**

Java Architecture for XML Binding (JAXB) allows Java developers to map Java classes to XML representations. JAXB provides two main features: the ability to marshal Java objects into XML and the inverse, i.e. to unmarshal XML back into Java objects. In other words, JAXB allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program’s class structure. 30

**JAX-RS**

JAX-RS: Java API for RESTful Web Services is a Java programming language API that
provides support in creating web services according to the Representational State Transfer (REST) architectural style. JAX-RS uses annotations, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints. 29, 32

**JSON**

JSON, or JavaScript Object Notation, is a lightweight text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages. 17, 32

**middleware**

In its most general sense, middleware is computer software that provides services to software applications beyond those available from the operating system. Middleware can be described as 'software glue'. Thus middleware is not obviously part of an operating system, not a database management system, and neither is it part of one software application. Middleware makes it easier for software developers to perform communication and input/output, so they can focus on the specific purpose of their application. 19

**MIME type**

An Internet media type is a two-part identifier for file formats on the Internet. The identifiers were originally defined in RFC 2046 for use in email sent through SMTP, but their use has expanded to other protocols such as HTTP, RTP and SIP. These types were originally called MIME types, after MIME (Multipurpose Internet Mail Extensions), and are sometimes referred to as Content-types, after the name of a header in several protocols whose value is such a type. 32

**ORM**

Object-relational mapping (ORM, O/RM, and O/R mapping) is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a 'virtual object database' that can be used from within the programming language. 30

**POJO**

POJO is an acronym for Plain Old Java Object. The name is used to emphasize that a given object is an ordinary Java Object, not a special object. The term was coined by Martin Fowler, Rebecca Parsons and Josh MacKenzie in September 2000. 30, 32

**POST**

POST is one of many request methods supported by the HTTP protocol used by the World Wide Web. The POST request method is used when the client needs to send data to the server as part of the request, such as when uploading a file or submitting a completed form. 13, 27

**PUT**

In HTTP, Uploads a representation of the specified resource. 13, 22

**RDBMS**

A relational database management system (RDBMS) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd. Most popular databases currently in use are based on the relational database model. 29
REST

Representational state transfer (REST) is a style of software architecture for distributed systems such as the World Wide Web. REST has emerged over the past few years as a predominant Web service design model. REST has increasingly displaced other design models such as SOAP and WSDL due to its simpler style. The term representational state transfer was introduced and defined in 2000 by Roy Fielding in his doctoral dissertation. Fielding is one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification versions 1.0 and 1.1.

RESTful

Conforming to the REST constraints. 7, 16–18, 22, 23, 27, 29, 31, 32, 37

SIP

The Session Initiation Protocol (SIP) is an IETF-defined signaling protocol widely used for controlling communication sessions such as voice and video calls over Internet Protocol (IP). The protocol can be used for creating, modifying and terminating two-party (unicast) or multiparty (multicast) sessions. Sessions may consist of one or several media streams. 9, 13, 17

UML

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering. The standard is managed, and was created, by the Object Management Group. It was first added to the list of OMG adopted technologies in 1997, and has since become the industry standard for modeling software-intensive systems. 27

W3C

The World Wide Web Consortium (W3C) Is an international community where Member organizations, a full-time staff, and the public work together to develop Web standards. Led by Web inventor Tim Berners-Lee and CEO Jeffrey Jaffe, W3C’s mission is to lead the Web to its full potential. 9, 11

webapp

A web application is an application that is accessed over a network such as the Internet or an intranet. The term may also mean a computer software application that is coded in a browser-supported language (such as JavaScript, combined with a browser-rendered markup language like HTML) and reliant on a common web browser to render the application executable. 30, 33

WebRTC

WebRTC (Web Real Time Communication) is a HTML5 standard that is being drafted by the World Wide Web Consortium (W3C), the mailing list was created in April 2011. It is also considered as a framework that was open sourced on June 1, 2011 that allows web browsers to conduct real-time communication. This enables applications such as voice calls, video chat, P2P file sharing and so on. 9, 11

WebSockets

WebSocket is a web technology providing for bi-directional, full-duplex communications channels over a single TCP connection. The WebSocket API is being standardized by the W3C, and the WebSocket protocol has been standardized by the IETF as RFC 6455. 7
XML
Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all gratis open standards. 17, 30, 32

XSD
XML Schema, published as a W3C recommendation in May 2001, is one of several XML schema languages. It was the first separate schema language for XML to achieve Recommendation status by the W3C. Because of confusion between XML Schema as a specific W3C specification, and the use of the same term to describe schema languages in general, some parts of the user community referred to this language as WXs, an initialism for W3C XML Schema, while others referred to it as XSD, an initialism for XML Schema Document. 30, 35
Bibliography


