WEB-BASED DEVICE RESERVATION SYSTEM FOR JYVSECTEC

Flórián Ákos Szabó

Bachelor's Thesis
May 2014

Degree Programme in Software Engineering
School of Technology, Communication and Transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

DESCRIPTION

Author(s) SZABÓ, Flórián Ákos	**Type of publication** Bachelor´s Thesis	**Date** 7.05.2014
	Pages 61	**Language** English
		Permission for web publication (X)
Title Web-based device reservation system for JyvSecTec		
Degree programme Software Engineering		
Tutor(s) SALMIKANGAS, Esa		
Assigned by SILOKUNNAS, Marko Jyväskylä Security Technology		

Abstract

This thesis presents a project for developing a web application to be used for reservation of devices inside Jyväskylä Security Technology. The purpose of the project was to develop the server application and this thesis describes the process of development.

The first chapter of the thesis describes the background of the thesis, and lists the objectives and requirements for the application to be developed. The second chapter covers part of the theoretical knowledge required to implement the project. The remaining chapters deal with the actual implementation and testing of the web server, the database and client side implementation.

The result of the thesis is a functioning web application that can be utilized to make reservations for the devices. The end result was reviewed and accepted by the supervisors of the thesis and practical training, although there is still room for improvement.

Keywords

Go programming, Web development, REST, JavaScript, jQuery, HTTPS, cookie

Miscellaneous

CONTENTS

FIGURES

ACRONYMS

AJAX	Asynchronous JavaScript and XML, is a technique for creating fast and dynamic web pages.
API	Application Programming Interface, is a set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application, or other service.
CA	Certificate Authority, is an organization that issues digital certificates.
CDN	Content Delivery Network, is a large distributed system of servers with the goal to serve content to end-users with high availability and high performance.
CSS	Cascading Style Sheets, a style sheet language used for describing the presentation semantics of a document written in a markup language.
DOM	Document Object Model, is an application programming interface for HTML and XML documents.
HTML	HyperText Markup Language, is a standardized system to transfer information of web pages across the internet.
HTTP	HyperText Transfer Protocol, is an application protocol used most of the time for transferring web pages over the internet.
JS	JavaScript, a scripting language used by browsers.
JSON	JavaScript Object Notation, is a format that uses human-readable text to transmit data objects across the network.
REST	Representational State Transfer, a set of constraints that define a software engineering style popular in web development.
SSL	Secure Sockets Layer, is a protocol providing secure transmission over an unsecure network.
TCP/IP	Transmission Control Protocol/Internet Protocol, is a set of communications protocol used to connect hosts over the internet
TLS	Transport Layer Security, is a protocol that ensures privacy and security over the internet.
URI	Universal Resource Identifier, is a string which is used to identify resources over the internet.

URL	Universal Resource Locator, is a string which is used to locate resources on the internet.
XML	Extensible Markup Language, is a markup language designed to carry data across a network.

1 Introduction

1.1 Jyväskylä Security Technology

Jyväskylä Security Technology (JyvSecTec) is a development project coordinated by JAMK University of Applied Sciences (JAMK). It was founded in September, 2011 to kick-start information security development and research projects in Central Finland. A so called Realistic Global Cyber Environment has been developed, in short RGCE, which enables cyber security development, testing and later the possibility to offer training services in this specific field. (“*Introducing JyvSecTec”, 2013*)

The RGCE environment has been created as a scaled down version of today’s Internet with as similar structure as possible. It contains network operators and Internet Service Providers, in short ISP-s, offering their services offered (e.g. Domain Name System, e-mail system) to the customers. This scaled down version of the Internet enables research and development as if it was carried out in the real world on real equipment using the existing infrastructure, but without the risk of disrupting the public services. It can simulate realistic corporate networks and ISP environments and enable testing these services against real threats and provide possible solutions to existing problems. (“*Introducing RGCE”, 2013*)

1.2 Objectives

The foundation for this thesis came from the time I spent at Jyväskylä Security Technology as part of the practical training period. I joined a project called Proxy which is a very complex project and this thesis only covers one part of it.

The idea for this project came from the possible need to offer training services in the future to students of JAMK University of Applied Sciences as well as companies interested in it.

The project consists of three parts. The Proxy server itself is the main part holding the system together. The other two parts: the Configurator system and the Reservation system. The later one, the Reservation system is the one which was developed during the practical training period. In Figure 1 an overview of the system is presented.

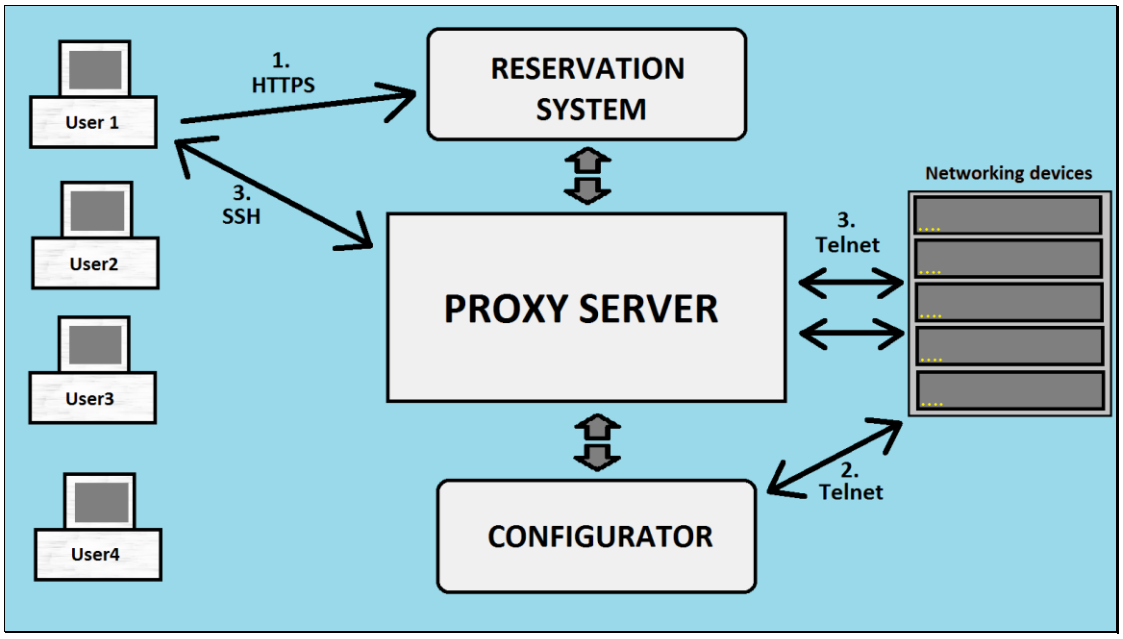


Figure 1: Overview of the structure of the whole system.

A possible user scenario is studied below as follows:

- A user wants to use the training services offered by JyvSecTec. First, the user has to log in to the '*Reservation System*' (nr. 1. on Figure 1.) and make a reservation for the *Networking devices* needed for a specific period of time. This part is done by communication with the *Reservation system's* web server through a web browser. Here the user can make a reservation for a set of devices, each with the selected configuration. If successful, this reservation will be stored in the database.

- After the reservation has been placed, the ‘*Proxy Server*’ has to make sure that the devices are pre-configured in time before the reservation begins. This is the accomplished by communicating with the ‘*Configurator’* (nr. 2. on the Figure 1.), which will then configure the devices before the reservations starts.

- Finally, when everything is ready, and the devices are configured before the reservation begins, the user is able to log in to the Networking devices during the reservation through the ‘*Proxy Server’* (nr. 3. on the Figure 1.).

Although this thesis is only concerned with the implementation of the ‘Reservation system’, the overall functioning is described shortly in the next paragraphs.

The component of the system called ‘*Proxy Server’* deals with connecting users to the ‘*Networking devices’*. This can be done in two ways: using the *Telnet* or the *Secure Shell* (in short SSH) protocols. The first one, Telnet, is not secure, because it sends the traffic across the network without encryption and authentication, so it can only be used internally between the ‘*Proxy Server*’ and the ‘Networking *devices’* where hijacking the connection can only be done if the attacker has physical access to the equipment. Since the users may come from the “outside world”, considering from JyvSecTec’s point of view, a secure protocol has to be used to connect to the ‘*Proxy Server’*, and therefore it is required that SSH is used. After the connection is established between the ‘*User’* and the ‘*Proxy Server*’, it is the server’s responsibility to translate between SSH and Telnet in both directions when the communication is established.

The ‘*Configurator’* is the part which deals with configuring the ‘Networking devices’ in time before a specific reservation starts. It is using the database managed by the ‘Reservation system’ to accomplish its tasks.

The ‘Reservation system’ is the main part that was implemented throughout the practical training period. The end result is a web application which uses the REST API exposed by the server to view, make, update and delete reservations. The overview of the architecture of the Reservation System is presented in Figure 2.

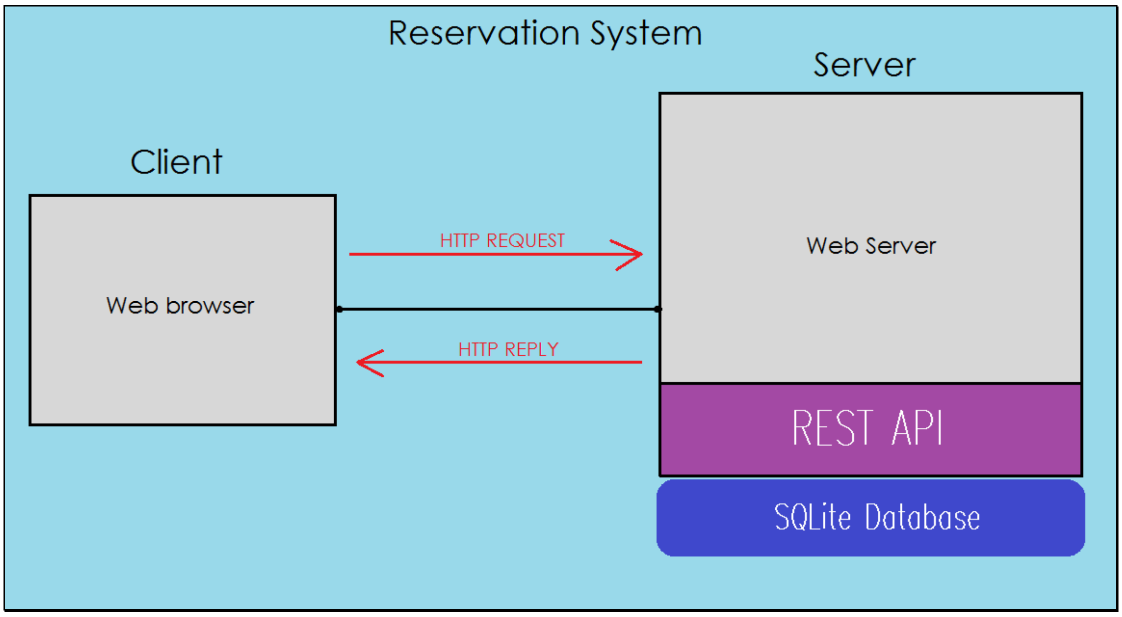


Figure 2. Reservation system architecture.

The system components, like the REST API, or the SQLite Database, are discussed in later chapters. The server-side programming was done in Google’s Go programming language, and the client-side programming is achieved with a combination of HTML, CSS, and Javascript. These topics are also discussed in later chapters.

1.3 Requirements for the application

Before the development of a system can be started, the details and objectives have to be clarified, so that appropriate constrains can be set up for the end result.

Authenticating the user before the system can be accessed was a clear requirement right from the start. This means there has to be user accounts, which can be used to access the Web application. Also, different user levels are needed. There need to be ‘administrator’ and ‘normal’ user level. Administrators can do basically anything, while normal user accounts have restrictions, for example, they can only modify or delete their own reservation.

The communication between the user and the server needs to be secure, which means it has to use Secure HTTP / HTTP over TLS. This means traffic is encrypted between the client and the server.

The server needs to provide a so called REST API which allows the client to be kept as simple as possible, and also it means that if needed new type of clients can utilize this API (e.g.: Android application which was written to use this API).

Once the user has authenticated and logged in to the system, it has to be able to create a reservation for a given period of time for a set of devices with predefined configurations to be loaded to those devices. The server has to make sure that there are no conflicts between two existing reservations.

Since this system is developed to be used by humans, there are some functional requirements as well, like a good looking user interface, and easy to use functionalities.

2 Theoretical background

The following chapters and paragraphs discuss the theoretical knowledge that was necessary in order to make this Web based Reservation system work correctly.

2.1 Web development

Web development or web programming is an area in Software development specifically dedicated to produce content which then can be accessed through the Internet using a web browser.

Web development itself has separate sub-fields like web-design which focuses more on the visual manifestation of a webpage, and web-programming which more or less focuses on the underlying architecture that serves and generates the content to be displayed. For larger companies, the web-development team can consist of hundreds of people, while smaller organizations may only need a single webmaster who oversees the operation of the organization’s websites. (“*Web development”, 2014*)

In the early days of the Internet around the mid 90s, which era is also called Web 1.0, web development was very different. Web servers served static web pages which the user could only view and not contribute to. Information was not dynamic; it was updated only by the webmaster. The users were only consuming the content. (“*Web 1.0*”, 2014)

Then, with the introduction of the so-called Web 2.0 technologies and the commercialization of the Internet, many things have changed. Web development has become a growing industry pushed by businesses trying to sell products and services to consumers online. Web servers no longer served only static web pages. The content became dynamic and now users are able to contribute to the web pages. Social media have also had a huge impact on the web. Web 2.0 sites allow users to communicate and collaborate with each other, it allows users to do much more than just retrieve information. User experience has become much richer and responsive to input. (“*Web 2.0*”, 2014)

For a better understanding of web development, it can be divided into two main areas:

- *Client side coding*: this part is mainly concerned with how the content is presented to the end-user. It uses technologies like HTML, CSS, JavaScript, jQuery, AJAX, Adobe Flash, Microsoft Silverlight etc. Some of these technologies are discussed later in this document. (“*Web development*”, 2014)

- *Server side coding*: this part of web development is used to provide backend services on which client-side programming can rely. The main programming languages are Java, PHP, Python, Ruby, .NET and **Golang.** From these languages Golang will be discussed later because that was the chosen language for implementing this project. (“*Web development*”, 2014)

As the reader can see, there are many options in both categories which can be combined in multiple ways in order to achieve the goals of certain projects. This abundant number of possibilities means that careful planning is needed before deciding which technologies to combine with each other because certain tools work best with certain other tools. This needs the web developer to be up-to-date on a wide range of old and new technologies however hard it can be. Choosing the right technology for our purposes can pose a major challenge in some cases, thus it requires careful consideration of all possibilities before settling on which to use.

2.2 Go programming language

On server-side, a system using the Go programming language was implemented, therefore in this chapter the language is introduced with some detail on what parts of it were specifically used when implementing the server.

Go, its other name Golang, is a programming language developed by a team at Google. The development itself started in September 2007, however the first version was released only in November 2009. Despite the fact that it is a newly developed programming language it has already gained significant attention by the developer community around the world. (“*Go history*”, n.d.)

Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java. (“*Go history*”, n.d.)

The authors of the language are: Robert Griesemer, Rob Pike and Ken Thompson. Ken was already involved in developing B, the predecessor of the C programming language which is still used in many parts of the world today. Ken and Rob worked together in the past at Bell Labs (“*About Ken Thompson*”, 2014). These people joining forces again at Google was bound to mean something great will follow, and it did: the Go programming language was created.

Go was created to make development in large scale more productive and it was designed to address the problems faced in software development at Google, which led to a language that is not a breakthrough research language

; however it is nonetheless an excellent tool for engineering large software projects.

Concurrency is important to the modern computing environment with its multicore machines running web servers with multiple clients, which might be called the typical Google program (“*Go at Google”*, Pike, 2014). Go is specifically designed for web server programming which was one of the reasons why it was chosen for implementing this project. Here is a simple web server which will print “Hello World!” to the browser.

```
package main

import "fmt"
import "net/http"

func helloHandler(wr http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(wr, "Hello World!")
}

func main() {
    http.HandleFunc("/", helloHandler)
    http.ListenAndServe(":8080", nil)
}
```

The first line *“package main”* is mandatory, and it has to be the main package if it is meant to be executed. If the purpose is to build an external library then the package name can be anything other than “*main*”.

The second statement import will fetch the listed libraries from Go’s standard library. The “*fmt*” package deals with formatted Input/Output, while the *“net/http”* package is essential for implementing the “Hello World!” web server.

The remaining parts are examined next, starting with the *“main()”* function. The first line which will call the “*http.HandleFunc()”* will request the “*http*” package to handle all HTTP requests with calling a function called “*handler*” with the parameters of the request and the option to reply to this request. This function call will be executed in parallel and this means it can handle many client requests for the same URL at the same time concurrently. Then the next line will call the *http.ListenAndServe()* function specifying that the server should listen on port “:*8080”* on any existing networking interface.

The mentioned “*handler*” function has a really simple job to do. It takes two parameters, an *http.ResponseWriter* and an *http.Request*. An *http.ResponseWriter* represents the web server’s response to the HTTP client. By writing to it, it can send data directly to the client. The *http.Request* is a data structure which contains the request that was received by the server. It has several fields like ‘*Method’* which can hold the standard HTTP methods, the two most common being *HTTP GET* and *HTTP POST* methods. It also contains the ‘*URL’*, to which the request was issued, and also the ‘*Header’* and ‘*Body’* which are very important parts of an HTTP request.

The only line the “handler” function contains *fmt.Fprintf(w, “Hello World!”)* requests the server to print the message “Hello World!” to the client by writing it to the http.ResponseWriter.

Despite the fact that Go is a fairly new language, it already has a quite large standard library making it easier to use the language to solve an ever widening range of everyday problems in programming.

Some packages are listed here which proved to be very useful in implementing the Reservation system:

- ***time*** – standard library for time management
- ***database/sql*** – standard library, used by the SQLite 3 database driver
- ***github.com/mattn/go-sqlite3*** – external, third party SQLite 3 database driver, needed for the *database/sql* standard library to extend its capabilities to SQLite databases.
- ***encoding/json*** – standard library for JSON parsing
- ***crypto/sha512*** – standard library, used in the process of storing the hash value of passwords instead of clear-text format.

Go was extensively used for creating the server application in two ways. First, the general purpose web-server was created, defining the URL-s and handler functions for those URL-s, then the REST API was written which can be used by the client to interact with the SQLite Database.

2.3 HyperText Transfer Protocol

The HyperText Transfer Protocol, is a stateless application level protocol used in distributed, collaborative, hypermedia information systems. It is the foundation on which the World Wide Web is built. It is a protocol that defines how web browsers can pull the contents of a web page from a web server. It provides an interface between software running on a computer and the network itself that carries the information. HTTP allows basic hypermedia access to resources available from diverse applications. (“*Hypertext Transfer Protocol -- HTTP/1.1*”, 1999)

HTTP communication usually takes place over TCP/IP connections. The default port is TCP 80, but other ports can be used. This does not preclude HTTP from being implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport; any protocol that provides such guarantees can be used.

HTTP functions as a request-response protocol in a client-server model. A web browser, for example may be the client, and an application running on a computer hosting a web site may be the server. The following figure shows the structure mentioned above:

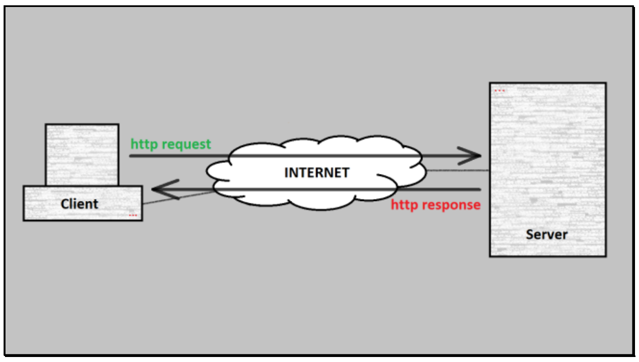


Figure 3: The basic model of requesting a web page using a browser

Most HTTP communication is initiated by the user agent and consists of a request to be applied to a resource on the target server. HTTP defines certain methods to specify the desired action to be performed on the identified resource. The resource if very often corresponds to a file or the output of a process residing on the server side. The following example shows parts of an HTTP GET request sent to www.jamk.fi/ web page using HTTP version 1.1.

```
GET / HTTP/1.1
Host: www.jamk.fi
```

Next is a possible response to this request by the server:

```
HTTP/1.1 200 OK
Date: Sun, 05 Apr 2014 16:38:00 GMT
Server: Apache/2.0.3 (Unix)…
Connection: Keep-Alive
Keep-Alive: timeout=15, max=100
… (more optional header parameters could follow)


<!DOCTYPE html>
<html>
…
</html>
```

To differentiate the results of executing an HTTP method, status codes were defined. Each code is a 3 digit integer which holds the information about the result of the request previously made. The first digit defines the class of response:

- *1xx*: *Informational* – Request received, continuing the process
- *2xx*: *Success* – The request was received, understood and accepted.
- *3xx*: *Redirect* – Further action is needed to complete the request
- *4xx*: *Client side error* – Bad syntax or request cannot be fulfilled
- *5xx*: *Server side error* – Server failed to complete the request which is otherwise valid.

The HTTP version 1.0 defined GET, POST and HEAD, while the currently used HTTP 1.1 specification added OPTIONS, PUT, DELETE, TRACE and CONNECT (*“HTTP - RFC 2616”,* 1999*).* Their effects are well known and can be relied upon. Any client can use any of these methods and servers can be configured to support any of them. Though most of the time only a fraction of the methods are used, namely: GET and POST. Next they are described with some additional methods important for this project implementation:

- **GET**: used to retrieve information identified by the Request-URI. This is a safe method that means it is only used for information retrieval.
- **POST**: used to transfer data to the server, for the purpose or storing or sharing information. POST is unsafe.
- **DELETE**: requests the server to delete the resource identified by the Request-URI. DELETE is unsafe.
- **PUT**: requests the server to update the resource identified by the Request-URI. PUT is unsafe as well.

Implementing the project required clear understanding of these properties of HTTP. It is very important to understand the basics of HTTP because in later chapters the REST architecture is discussed which builds on these principles.

Secure HTTP

Technically, HTTPS is not a standalone protocol, rather it is the result of simply encapsulating HTTP inside the SSL/TLS protocols thus, adding making it secure. The full specification of HTTP over SSL/TLS is out of the scope of this thesis, but it is discussed briefly in the following paragraph(s).

Conceptually, HTTPS is very simple. One key difference when using HTTPS instead of HTTP is the protocol identifier in URLs. The keyword “https” is used instead of “http”.

The security is achieved through simply using HTTP over SSL/TLS as it is used over TCP. SSL provides communication security between two hosts through integrity, authentication and confidentiality.

SSL/TLS uses certificates to achieve the desired security. As a consequence certificate authorities, and public key infrastructure is needed to verify that the certificate belongs to the correct entity. (“*HTTP over TLS (HTTPS)*”, 2000*)*

In order to prepare the web server used in this project to serve content over HTTPS, a public key certificate was necessary to be created using a generating code provided by the Go programming language. This certificate is not issued by a trusted Certificate Authority (CA), therefore browsers display a warning message when first accessing the project web page. But for development purposes, this solution works and the warning can be neglected.

2.4 Session management via HTTP Cookies

In this project, session management is implemented using HTTP Cookies, therefore, in the following paragraphs the details are discussed.

Because of the stateless nature of HTTP, a client using a web browser must establish a new connection to the web server with each new GET or POST method. The web server, therefore, cannot depend on an already existing network connection for longer than a single HTTP GET or POST operation.

Session management is a technique applied by the web developer to enable the stateless HyperText Transfer Protocol to utilize session state. For instance, once a user has been authenticated to a web service, the user should remain authenticated throughout a certain period of time without the need of authenticating again when new requests are made to the same service. There are some methods available to achieve this and of them is achieved through using HTTP Cookies. (“*HTTP State Management Mechanism”*, 2011)

How it works: The server receives a request by the user (usually GET request), and sends back a response with a special Set-Cookie in the HTTP header which contains the following properties:

- ***Name***: name of the Cookie
- ***Value***: the actual value of the Cookie. This can be random or not, but the important thing is that it is stored also on the server side along with more information about the client.
- ***Domain*** and ***Path***: these two attributes of a Cookie define the scope in which the Cookie is valid. When a new HTTP request is about to be sent the browser will attach every Cookie that is valid in that request's domain to the request.
- ***Expires / Max-Age***: defines a timestamp until the Cookie is valid. It tells the browser when to delete the Cookie.
- ***Secure***: this attribute does not have a value; rather the presence indicates that this behavior is expected by the browser. In case it is present it tells the browser to use the Cookie only via secure/encrypted communication channel.
- ***HttpOnly***: this is also an attribute without an actual value. An HttpOnly Cookie is only accessible via HTTP / HTTPS methods and inaccessible via for example JavaScript, which means that the cookie cannot be accessed by for example JavaScript's 'document.cookie' method, thus it cannot be stolen via cross-site scripting attacks.

After a Cookie has been set on the client side (browser), with every new request inside the Cookies domain, the name and value of the cookie is sent in every request. Next is an example that shows this behavior.

```
GET / HTTP/1.1
Host: www.jamk.fi
Cookie: cookie1=value1; cookie2==value2; …

…
(rest of header parameters)
```

When the server receives the request above it can check if the cookie is present in the request and act accordingly.

A good example of how this works that is also implemented in this project is the login page for accessing the reservation service. The first time a user tries to access the service it will be presented with a login page. If authentication is successful the server can redirect the user to an internal page while setting a cookie on the client's browser at the same time. Here is the details of the cookie that's actually set on the client after successful login.

- **name**: 'session_token'
- **value**: 50 characters long randomly generated string consisting of digits, lower and upper case letters.
- **expires**: a time in the future specified in the format defined by RFC 1123 ("Wdy, DD Mon YYYY HH:MM:SS GMT"). Currently it is set to be 15 minutes from the current time.
- **domain**: this property is not yet defined, because the web server application does not have a public domain registered. Once it does it can be set here, which will mean the cookie is only valid in that domain.
- **HttpOnly**: true
- **Secure**: true.

Now, every time the user navigates to a sub-page inside the service the server will only allow access after checking if the correct (session) Cookie is present in the client's subsequent requests.

2.5 HyperText Markup Language

One of the main technologies used on the client side is the HyperText Markup Language or HTML for short. This is the foundation that provides a place for integrating other useful technologies into the system, therefore HTML is discussed next.

HTML is the main markup language that is meant for creating web pages that are to be displayed in a web browser. The purpose of the web browser is to read HTML code and compose it into visible or audible information, and present it to the user.

HTML code is written in the form of elements consisting of tags enclosed in angle brackets like <*html*>. HTML elements are usually made up of an opening tag, some content and a closing tag. Also there are some exceptions called empty elements like <*br*> which commands the browser to insert a line break into the web page. Here is a general example of HTML code displaying the usual “Hello World!” message when viewed in a browser:

```
<!DOCTYPE html>
<html>
     <head>
     </head>
     <body>
          Hello World!
     </body>
</html>
```

This example demonstrates well the meaning of markup language. It basically encodes information between tags and makes it easy for machines to process the information.

Throughout the implementation of this project, one of the goals was to implement the server in a way that it is able to generate valid and correct HTML by querying the database, generating content based on the queries and returning the document to the user’s browser upon a request.

2.6 Cascading Style Sheets

Cascading Style Sheets (CSS) is the language for describing the presentation of Web pages, including colors, layout, and fonts. It allows one to adapt the presentation to different types of devices, such as large screens, small screens, or printers. CSS is independent of HTML and can be used with any XML-based markup language. The separation of HTML from CSS makes it easier to maintain sites, share style sheets across pages, and tailor pages to different environments. This is referred to as the separation of structure (or: content) from presentation. ("*What is CSS*?", n.d.)

When implementing the reservation system web UI, CSS was used extensively to achieve good design, which is essential for a good user experience. The following figure shows a simple example how CSS was used.

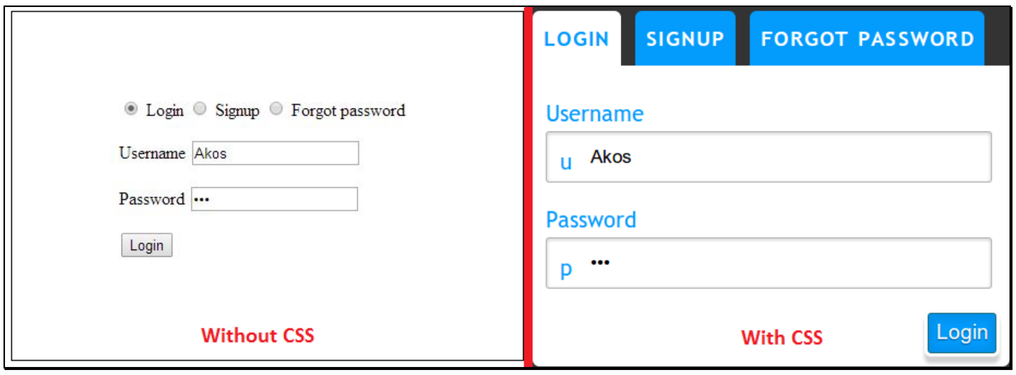


Figure 4. Example demonstrating CSS use case.

On Figure X. a simple use-case of CSS is shown. With its help, HTML source codes can be kept simpler, and less redundant, while also making the life of the developer easier.

2.7 JavaScript

Although, on the server-side Go was used as the primary technology for implementation, to make the client-side work, several technologies were used in cooperation to achieve the implementation goals and create a good user experience. One of these technologies is JavaScript, which was used to utilize the REST API offered by the server, as well as to make an easy to use and intuitive user interface.

JS is a dynamic programming language, most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously and alter the content displayed by the browser. It is also a prototype-based scripting language with dynamic typing and has first-class functions. It was influenced by C, and has similar naming conventions as Java, however, the two languages (JS and Java) are otherwise unrelated and have very different semantics. JS is a multi-paradigm language, supporting object-oriented, imperative and functional programming styles. JavaScript also has other application areas for example in PDF documents, site-specific browsers, and desktop widgets. (*"JavaScript", n.d.*)

The most common use of JS on the client-side is to write functions that are embedded in HTML web pages thus allowing it to interact with the Document Object Model (DOM) of the page. Some simple examples of this usage:

- animation of page elements, fading out, resizing, moving etc.
- interactive page content
- validating user input of a web form
- loading new page content or submitting data to server via AJAX

The JS source code itself can be loaded into pages in two ways. It can be written in a separate file with '.js' extension and then included anywhere in the HTML source code. This way, the when the browser reads this line, it has to make a new request to fetch the file specified by the <script src=""></script> tag's 'src' attribute. The source can be either local on the same server that is serving the HTML code, or another completely independent source, for example a public CDN.

```
<head>
    <!-- inserting JS from separate file -->
    <script src="path/to/code.js"></script>
    <!-- inserting JS code directly into HTML -->
    <script>
          alert("This is JavaScript");
    </script>
</head>
```

Also, the JS source code can also be inserted directly between the opening <*script*> and the closing <*/script*> html tags. Regardless of which method is used, execution of the JS code is going to give the same result.

All JavaScript source codes, required for the implementation of Reservation system, are stored locally (even the external libraries like jQuery, jQuery Datatables) and served by the web server, thus minimizing the latency of fetching the code and allowing the HTML to be kept simple and well organized.

2.8 jQuery

During the implementation of the project, jQuery was used several times to make the client-side programming easier, so in this chapter it is introduced with some examples alongside as well.

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML content. Since it is purely a JavaScript library, it is really helpful to learn JavaScript before starting to work with jQuery. It was released in January 2006 at BarCamp NYC by John Resig. It is currently developed by a team of developers led by Dave Methvin at the jQuery Foundation. Used by over 80% of the 10,000 most visited websites, jQuery is the most popular JavaScript library in use today. (“jQuery”, n.d.)

jQuery is open source software and licensed under the MIT License (http://opensource.org/licenses/MIT). jQuery's syntax was created to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. jQuery also provides the environment for developers to create plug-ins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications.

Before it can be used in the HTML code, it has to be included/imported, which basically means downloading a single JavaScript file. It can either be included locally, or from a public CDN which hosts this file in a compressed production-ready format. These two methods can be seen on the following figure. The option, to store the jQuery source code locally and let the web-server serve it, was chosen when implementing the system.

```
<!-- inserting jQuery from file on local storage -->
<script src="path/to/jquery.min.js"></script>

<!-- inserting jQuery from public Contend Delivery Network -->
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js">
</script>
```

Although jQuery itself is a pretty lightweight JS library it can help the programmer achieve many complex tasks with simple and easy to remember commands. Here are a couple of things jQuery is useful for:

- DOM element selection
- DOM manipulation
- Events, effects and animations
- AJAX
- XML and JSON parsing
- Extensibility through plug-ins

The jQuery library itself can only be used after loading a web-page is completed. There is a built-in way to check this, using the following code.

```
$(document).ready(function() {
     //everything inside these parenthesis will be
     //executed when the DOM has been fully loaded.
});
```

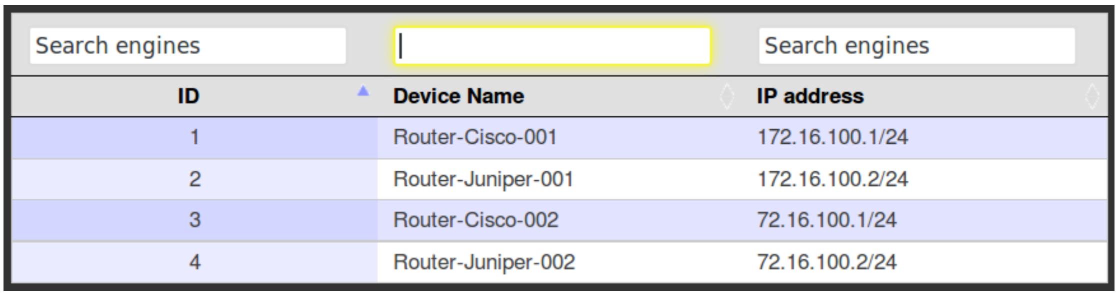
Once the DOM is fully loaded, this function will be called and all the code inside will be executed. Practically this is a way of making sure the webpage will not become broken. For example, there could be a code snippet, which would redesign the looks of a page, but if this is executed before the element it would modify is created, then it could cause problems. Therefore it is a common practice to put jQuery code in the block mentioned above. This method is used in every part of the system that uses jQuery.

One really helpful usage of jQuery was through the plug-ins called Datatables, Datepicker and Timepicker.

The jQuery Datatables plug-in was used to present data from the database in an easy to use and user friendly format. It hallows the data to be loaded from JSON source which is suitable for integration with the server's REST API. Next an example is shown how the plug-in was utilized during the implementation of the system.

```
$(document).ready(function() {
     var table = $('#restTable').dataTable({
          …
          "sAjaxSource": "/app/v1/devices?for=datatable",
          …
     });
});
```

On the figure above, an example is shown about how the jQuery Datatable plug-in was implemented. Inside the '.dataTable()' function the parameter "*sAjaxSource*" is used to specify that the content of the table is to be fetched from the link "*app/v1/reserve?for=datatable*", which happens to be part of the REST API which will return data in JSON format to the client. Upon receiving the data, the plug-in will parse the JSON data and present it to the user in the following format.



Search engines		Search engines
ID	Device Name	IP address
1	Router-Cisco-001	172.16.100.1/24
2	Router-Juniper-001	172.16.100.2/24
3	Router-Cisco-002	72.16.100.1/24
4	Router-Juniper-002	72.16.100.2/24

Figure 5. jQuery Datatable plug-in example.

Using this jQuery plug-in allows the client interaction to be more intuitive and user-friendly which helps to achieve a great user experience.

2.9 SQLite Database

During the implementation of the project, a local SQLite database was used as storage, so in the following chapter SQLite in general is discussed.

SQLite is a free to use database software library that provides a database engine with the following properties. (“SQLite”, n.d.)

- ***Self-contained***, which basically means it requires only minimal support from external libraries or from the Operating System it is running on.
- **Server-less**, which means unlike most SQL database engines SQLite is not implemented as a separate server process which can only be accessed through TCP/IP. When using SQLite, if the user wants to access the database the only thing required is access to the database files stored locally on the disk. One advantage this property of SQLite is that there is no separate server process which needs to be installed, configured, managed and troubleshot.
- ***Zero-configuration***, which is apparently a consequence of the previous property. Since it is server-less, there is no need for an administrator to manage a complex database server system. This makes SQLite an ideal tool for development purposes and small-scale systems.
- **Transactional**, which guarantees that the database implementation complies with the ACID principles. That is all changes in the database are **A**tomic, **C**onsistent, **I**solated and **D**urable, even if the transaction is interrupted by a program crash, operating system crash or power failure. (“*ACID properties*”, n.d.)

As a consequence of these properties, SQLite is very well suited for development purposes. It is compact and lightweight hence it is easily deployable to any system. Also Go has good supporting libraries to manage

SQLite databases, so the decision to choose SQLite for implementing it into the project was easy to make.

2.10 Representational State Transfer

During the discussion about the implementation of the project, it was a clear requirement that the system needs to use the REST architectural style, therefore it is discussed next.

The term REST, was first mentioned in an academic dissertation by Roy Fielding at the University of California in 2000 with the title "Architectural Styles and the Design of Network-based Software Architectures".

In short, REST is a software architectural style consisting of a coordinated set of architectural constraints applied to components, connectors and data elements, within a distributed hypermedia system. ("*Representational State Transfer*", n.d.)

REST is a resource-based architecture, in comparison to SOAP-RPC for example, which is action based. In a REST environment each resource is identified by an URI (Universal Resource Indicator), which then is used in each request to specify the type of action, like HTTP GET or POST, to perform on the given resource pointed by the URI. The result of the action can be sent back to the client by using HTTP status codes. A couple of examples are "200 – OK", "201 – Created", "401 – Unauthorized".

Most of the time the data representation format is JSON or XML with JSON being more common, but the REST architecture does not define a strict rule for this.

The architecture is most commonly described by the 6 constraints that has been established in Roy Fielding's dissertation. The following 6 constraints are deducted from the dissertation: "*Architectural Styles and the Design of Network-based Software Architectures", 2000.*

- ***Client-Server model***: This is probably the most common architectural model and is a requirement for REST. It basically means that there is a clear separation of concerns, which means clients are not concerned with data storage; that is the server's task, so that clients can be kept simple. On the other hand, servers are not concerned with the user interface or user state, which means they can also be simpler and more scalable. Servers and clients can also be replaced and developed independently so long as the interface between them remains the same.
- ***Stateless***: Which means that no session data is stored on the server side, thus each request has to have every required parameter to be successfully processed by the server. It means that each request is independent from every other request. These requests do not require the server to retrieve any kind of application context or state. One trade-off with this constraint is the fact that it might decrease network performance by increasing the repetitive data sent in a series of requests, because it cannot be left on the server in a shared context.
- ***Cacheable***: This is an effort to try to minimize the network overhead caused by statelessness. What this constraint means is that the server responses must be cacheable whenever it is possible. It can be implicit when the client decides to cache despite the fact that the server did not explicitly set an Expires / Max-age header field when returning a response. This feature has the potential to partially or completely eliminate some interaction, improving efficiency, scalability and performance as perceived by the user, since it reduces latency.
- ***Uniform interface***: This is the fundamental feature which distinguished the REST architectural style from other network-based styles. It describes the interface between the client and the server. Consequentially, what this means is that, HTTP methods are

associated with operations which can be applied on resources identified by URI-s found on the server, and the result of the operation is signaled by the HTTP response status. Though it has to be noted that REST does not specify that HTTP has to be used for this purpose, any other suitable protocol can be used as well.

- **Layered system**: This is a feature which means that the client, when communicating with the server, cannot actually tell whether it is directly communicating with the server or with some intermediary between the client and the server. This feature is connected with the *Cacheable* constraint.
- ***Code-on-demand*** [OPTIONAL]: This feature allows client functionality to be extended by transferring code on demand in the form of applets and scripts to the client which can utilize them. This allows clients to be simpler by reducing the number of features required to be pre-installed.

All these features, except the optional 6th feature, are needed to rightfully state that a system is implemented in the REST architectural style. Compliance with the REST constraints provides: scalability, simplicity, modifiability, visibility, portability and reliability.

However, this cannot be stated about the Reservation system because of two reasons. First, it uses HTTP Cookies to manage session state, which contradicts the statelessness constraint. Second, it does not utilize the caching services offered by HTTP headers, therefore caching is missing from the API.

Despite these two reasons, the most important constraint, the “Uniform interface” is satisfied, therefore it can be considered as a REST-like service, but not a full REST API, because it does not meet all the requirements.

3 The Database

In this chapter, the actual process of planning and implementing the database is detailed.

3.1 Structure and layout

Before the database structure can be implemented, it was necessary to think about what information is needed to make a well functioning system. Then when this information is ready, it needs to be translated into actual database tables with certain fields holding important information, and links have to be defined between these tables which helps to make sense of the data.

The first version of the database structure plan was very simple, but as the development of the system was evolving over time, new requirements arose that required small modifications to be applied to the structure, for example adding new table, or adding new columns to existing tables, or defining new connections between tables. The next figure shows the first version of the database that was created when development of the reservation system started.

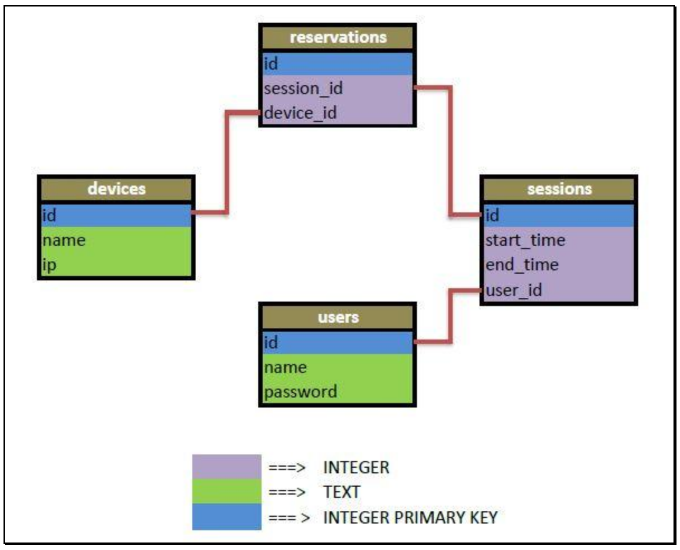


Figure 6. Showing the initial layout of the database.

After some time spent developing and improving the system, as new features were added, modifications were needed to be made to the structure of the database. For example, to enable registering a new user account, and recovering an existing because of a forgotten password, two new tables had to be introduced: ‘recover’ and ‘activate’. The following figure shows the current structure of the database.

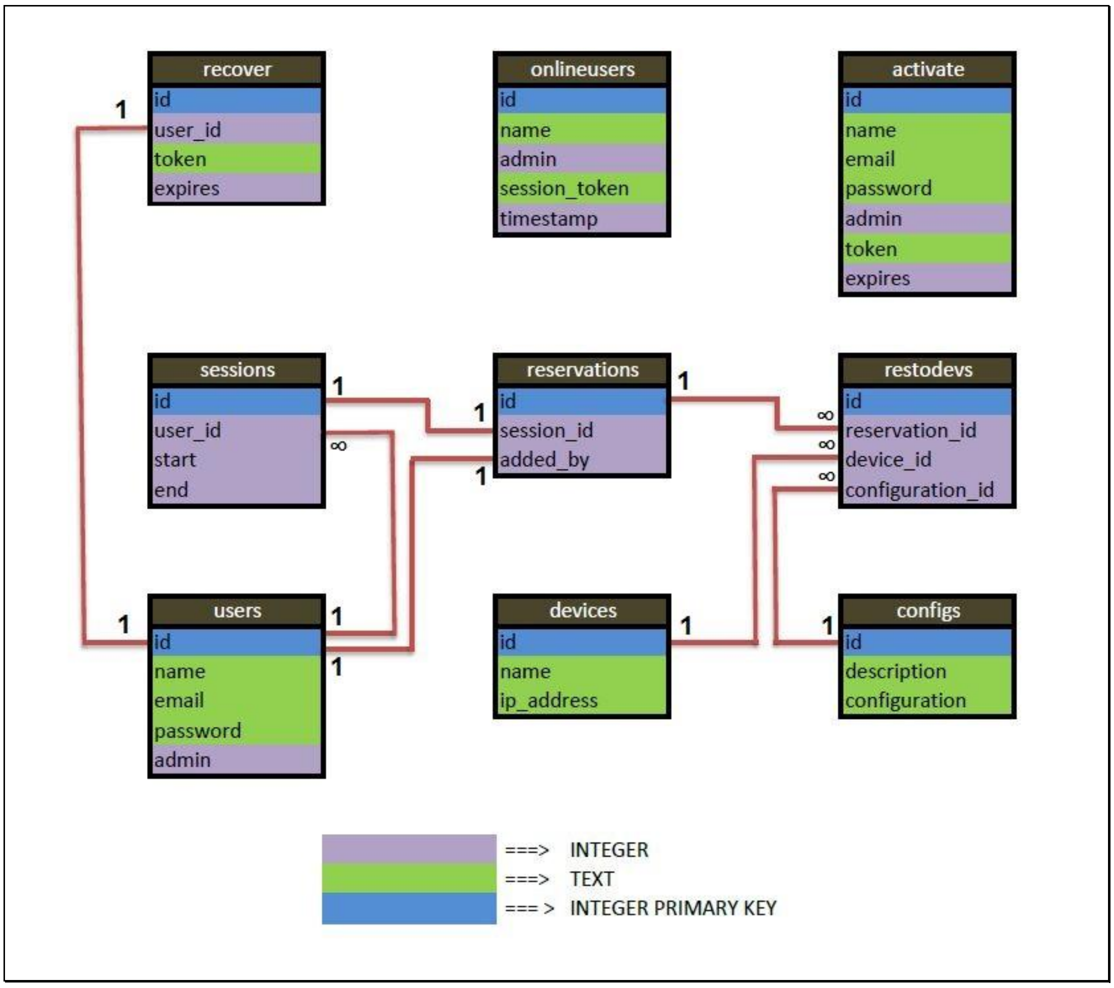


Figure 7. Showing the current layout of the database.

Users table

This table contains information about users that can log in with its ‘name’ and ‘password’ to access the Reservation system. After a successful user registration process, information is stored here. For security reasons the ‘*password’* column contains the hash value of the actual password, which is generated using the *SHA-512* algorithm. The ‘*admin’* column signals whether a user has administrator privileges or not, by containing ‘*1*’ if it does and ‘*0*’ if it does not.

Devices table

This is the table containing information about the devices that can be reserved by the system. Later, when it is needed it can be extended to add new columns if needed to store additional information about each device. The ‘*name*’ attribute has to be unique, as well as the ‘*ip_address*’ which is used to log in remotely to the device and access its services.

Configurations table

This table contains information about configurations that can be applied to certain networking devices. The ‘description’ and the ‘configuration’ columns have to be unique separately and together as well.

Sessions table

This is one of the three tables that are holding the whole system together. It contains 1 row of each reservation that has been made. The start and end time of a particular reservation is stored in *UNIX time* format, defined as the number of seconds that have elapsed since the 1st of January, 1970 (“*UNIX time*”, n.d.). The ‘user_id’ column refers to the user that owns the reservation (it can be different from the user that has placed the reservation, since ‘admin’ users can create reservation for others. Also, there is a constraint defined which assures that the end time of a reservation is always after the start time.

Reservations table

This is another table which is essential to the reservation system to function properly. It ties a session from the ‘*sessions*’ table to devices through another table ‘*restodevs*’ and also stores information about which user made this reservation.

Restodevs table

This is the last of the three main tables needed to properly store a reservation. It specifies which devices with which corresponding configuration selected for the device, are needed for a reservation. It has a couple of constraints to make sure that a device cannot be present twice in the same reservation, and also that a configuration can only be selected once per device per reservation.

Onlineusers table

This table is used in keeping track of the logged in users. Once a user successfully authenticated to the system, a new record is created in this table with the specified columns. ‘Name’ stores the username, ‘admin’ contains ‘1’ if user is admin and ‘0’ is user is not. The ‘session_token’ column stores the randomly generated string which is used when setting the cookie in the client’s browser. The ‘time_stamp’ column holds information about when the user logged in, which is used in limiting the length of a session for a specific amount of time for security reasons. Right now, a user will be automatically logged out 15 minutes after the login.

Recover table

This table is used in the recovery process of accounts whose owner has forgotten the password and is unable log in. This is done after the user has filled in a form containing the username and email which belongs to the same account. After this information has been submitted, the Reservation system will send an email to the specified email address containing a link which can be used to recover the account by creating a new password. This link that is sent contains a randomly generated 50 characters long token which will be stored in this table. When the user clicks the link the server will check if this token exists in this table and if it does it will allow the user to reset the password for the account.

Activate table

This table is used in registering a new user account for the Reservation system. When filling out the form for registering a new account, the information is stored in this table. After the registration is done an email will be sent to the specified email address containing a link which needs to be opened in order to fulfill the registration. The link contains a randomly generated 50 characters long token which is used to identify which account needs to be put to the ‘users’ table from this table. Until this is done, the account cannot be used to log in, as the user account details are only moved to that table upon opening the link from the email.

3.2 Managing the Database in Go

In this chapter the process of integrating the SQLite Database into a Go program is explained in more detail as well as how the Database is utilized in the application.

In order to be able to use a SQLite database two Go software libraries have to be imported to the application:

```
import (
    "database/sql"
    _"github.com/mattn/go-sqlite3"
)
```

The first one is from to Go standard library. Because it is a general purpose database library, it cannot operate on a SQLite Databases on its own. It needs a database driver, which is provided in the second import statement. The ‘go-sqlite3’ library is a third party library developed by a github.com user called ‘mattn’. This library has passed the compatibility test and is recommended for use (see the full list: https://code.google.com/p/go-wiki/wiki/SQLDrivers) by the Go community.

Next when the required packages are imported, to operate on a database, it either has to be created or opened from the computers local storage. The Reservation system is implemented in a way that the creation of the database is decoupled from the actual web server. The code for this is in a separate file which has the following code to create the database.

```
os.Remove("first.db")
db, err := sql.Open("sqlite3", "./jyvsectec.db")
        if err != nil {
                log.Fatal(err)
                return
        }
defer db.Close()
```

After the database is created, it can be manipulated by using functions provided by the third party database driver, in this case the ‘go-sqlite3’ driver by ‘mattn’. A couple of examples follow to demonstrate the functionality.

```
//inserting into the a database table
_, err = db.Exec("INSERT INTO users … ;")
if err != nil {
     return err
}
//running a query on a table in the database
result, err := db.QueryRow("SELECT name FROM users WHERE id=?", id)
if err != nil {
     return err
}
```

As it can be seen in the example above, it is fairly easy to write functions that can create, delete, query tables in the database. There are a couple of functions that can be used for this purpose, like *Exec()*, *Query()* and *QueryRow()*. The *Exec()* function can be used to do “INSERT”, “UPDATE” and “DELETE” operations while *Query()* and *QueryRow() functions* can be used to execute “SELECT” statements, as the name suggests.

Another important thing that is useful in some cases is the ability to execute transactions in the database. This can be done as shown in the following code.

```
//this is the beginning of the transaction
tx, err := db.Begin()
if err != nil {
     return
}
  //then prepared statements are defined with the following code
  stmt, err := tx.Prepare("INSERT INTO ex (id) VALUES (?)")
  if err != nil {
      return
  }
  defer stmt.Close()
  //then these prepared statements can be executed
  result, err := stmt.Exec(userID)
  if err != nil {
      return
  }
//then as the last step the transaction has to be closed
tx.Commit()
```

Using transactions can be very useful in some cases, especially when there is a batch of operations that have to be executed as a block. If the execution of one statement fails the whole transaction is cancelled and the database is rolled back to the state before the transaction was started. This was particularly useful during development and made the implementation of the reservation handler much easier, because a successful reservation required data to be inserted into three separate tables, and in case of failure of an insert, a database rollback was automatically executed.

4 The Web server

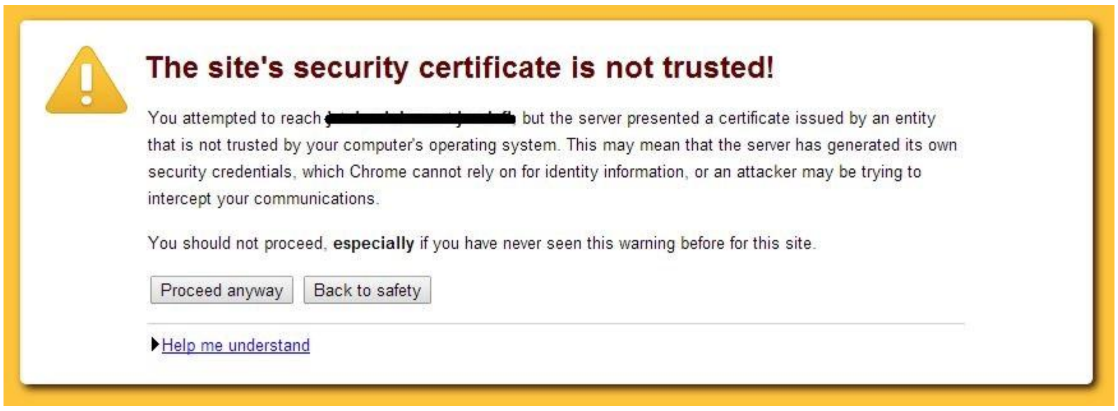
In this chapter the process of implementing the Web server is discussed, highlighting the most important aspects of the server.

4.1 Laying down the base of the Web server

The development of the Web server was started from a single “Hello World!” server as shown in Section 2.2. From that point several aspects of the server were improved. One of these things was switching from an HTTP server to an HTTPS server which makes the communication between the client and the server encrypted, thus the connection becomes secure, which was a requirement for the application. Here is how it is achieved, containing the scenario when a user tries to access the site using HTTP but gets redirected by the server to use HTTPS.

```go
//start the HTTPS server in a separate goroutine
go func() {
      err := http.ListenAndServeTLS(":7777", "cert.pem", "key.pem", nil)
      if err != nil {
            log.Fatal(err)
      }
}()
//start the HTTP server in current goroutine
err := http.ListenAndServe(":7770", http.HandlerFunc(redirectHTTPS))
if err != nil {
        log.Fatal("ListenAndServe error: %v", err)
}
//this part is outside of the main() function body
func redirectHTTPS(w http.ResponseWriter, r *http.Request) {
        http.Redirect(w, r, "https://localhost:7777"+r.RequestURI, 301)
}
```

For this feature to work, a certification and a key had to be generated. Thankfully there is a tool for this in the ‘net/http’ package’s folder, named *‘generate_cert.go’*. It allows the developer to generate a self signed certificate which can be imported by the HTTPS server as shown in the code snippet above. Using this certificate by the server will cause the browsers, which connect to this web server, to display a warning message about the fact that the server’s certificate is not valid. In this case this warning can be safely ignored. Next is an example of such a warning.


The site's security certificate is not trusted!

You attempted to reach [redacted] but the server presented a certificate issued by an entity that is not trusted by your computer's operating system. This may mean that the server has generated its own security credentials, which Chrome cannot rely on for identity information, or an attacker may be trying to intercept your communications.

You should not proceed, **especially** if you have never seen this warning before for this site.

Figure 8. Example of an SSL certificate warning.

4.2 URL handling by writing functions

At this point, when the server is started, it can serve requests, but for that to happen different functions have to be declared which will be invoked when a request arrives for the URL handled by that function. The next example shows how to declare the URL handlers.

```
//these commands have to be executed inside main()

http.HandleFunc("/", defaultHandler)
http.HandleFunc("/main", mainPageHandler)
```

After the web server is deployed, the effect of these two commands will be that the URL '*https://custom.domain/*' will be served by the *defaultHandler()* function while the URL '*https://custom.domain/main*' will be served by the *mainPageHandler()* function.

Here is a list of URLs which are handled by the Web server and their corresponding functionality.

- '/' – This is the default page which is displayed when the client first visits the webpage, and also where the client is redirected in case of running out of the session window.
- '/main' – which is handling the main page from where the client can navigate to different pages.

- '/[users/devices/configs/sessions/reservations/restodevs]' – which are handlers for interacting with individual tables in the database. Only an admin user can access these and make modifications in them.
- '/[recover/activate]' – These two URL-s are used by the account registration and recovery process.
- '/reserve' – This is the URL which will display the Web UI for making a reservation
- '/app/v1/[users/devices/…]' – This is the URL for accessing the REST-ful API. As the last part of the URL, a table name can be inserted and this way the client can interact with the database tables directly using the API.

Next an example function is shown which handles the URL *'/users'*.

```
func usersHandler(w http.ResponseWriter, r *http.Request) {
      user, err := getCurrentUser(r)
      if err != nil {
            http.Redirect(w, r, "/?redirect=session_expired", 302)
            return
      }
      if !isAdmin(user.Name) {
            renderTemplate(w, "error",
                           map[string]string{"Error":"Permission denied!
                           Only the administrator can interact
                           with this URL."})
            return
      }
      w.Header().Set("Content-Type", "text/html")
      renderTemplate(w, "user", user)
}
```

Explanation: First, the server checks if the client, which sent the request, is logged in or not. If not, it will redirect the client to the default page: '/' with a message that the session has expired. Next, the server check if the current used has admin privileges or not. If it does not, it will display an error message in the middle of the screen saying in red: *"Permission denied! Only the administrator can interact with this URL."*

The rest of the URL handler functions work exactly like that, except the function which handles the URL '*/reserve*', because it does not require the client to have administrator privileges to access the page.

5 The REST-like API

In this chapter, one of the most important aspects of the server, the REST-like API is discussed. It is important because it gives the possibility, in case it is needed, to port the Reservation system to other clients like mobile (e.g. Android, iOS, Windows Phone) or desktop clients (e.g. native Windows application).

5.1 Resources and representations.

In the world of REST-ful services, it is important to understand the fact that the system consists of resources rather than actions. Often these resources are database records, or other kinds of information. Another important thing is the representation of these resources which will be applied during the transferring of the resource between the client and the server.

For example, in the Reservation a User from the 'users' table is a resource, and the representation can be the particular User's name, email address, password, admin rights represented in JSON format, like it is implemented in the system, however it can also be in XML format. Next is an example of that resource represented being sent to the server in JSON format.

```json
{
    "id": 1,
    "name": "Florian",
    "email": "test@example.com",
    "password": "test",
    "admin": 1
}
```

This JSON representation describes a User account in the reservation system. Each resource like this has a clearly defined URI which can be used to gain access to the resource with a protocol like HTTP. For example, the above mentioned resource has the following URI in the Reservation system.

```
/app/v1/users/1
```

When the resources are clearly described, HTTP can be used to define operations on these resources using the URI. Next the 4 operations, that are needed to manage resources of the Reservation system, are shown.

- **Querying a User resource**: To get the details of a specific user though the REST API, the following HTTP request has to be sent.

  ```
  GET /app/v1/users/1 HTTP/1.1
  Host: reservation.com
  Accept: application/json
  ```

  The method is HTTP GET, the URI has to specify the Users's ID that we want to query, and the 'Accept:' header can specify that the sender of the request is expecting the resource in JSON format.

- **Creating a User resource**: In order to create a new user in the database through the REST API, the following HTTP request has to be sent to the server:

  ```
  POST /app/v1/users HTTP/1.1
  Host: reservation.com
  Accept:  application/json
  Content-type:  application/json

  {"id": 1, "name": "Florian", "email": "test@example.com",
  "password": "test", "admin": 1}
  ```

  The request has to be HTTP POST to the URI which does not specify a particular user, rather the collection of users (*/app/v1/users*). In the 'Content-type' header the sender specifies that the body of the request is in JSON format and 'Accept' header specifies that the response it is expecting should also be in JSON format.

- **Updating a User resource**: In order to update an existing User in the database through the REST API, the following HTTP request is sent.

  ```
  PUT /app/v1/users /1 HTTP/1.1
  Host: reservation.com
  Accept:  application/json
  Content-type:  application/json

  { "id": 1, "name": "Florian", "email": "test@example.com",
  "password": "test", "admin": 1 }
  ```

In this case, the request is very similar to the one before, when creating a new user, but two changed are needed. The HTTP method is PUT instead of POST, and the URI has to point to a specific user by its ID, which already exists in the database.

- **Deleting a User resource**: To delete an existing User in the database the following HTTP request has to be sent.

```
DELETE /app/v1/users/1 HTTP/1.1
Host: reservation.com
```

  This action is very simple. The HTTP method is DELETE and the URI has to point to an existing User resource in the database to perform the delete operation.

These operations are implemented in the same manner for the following tables in the database: ‘*users*’, ‘*devices*’, ‘*configs*’, ‘*sessions*’, ‘*reservation*’, ‘*restodevs*’.

5.2 Implementing the REST handler functions

After the URI structure of the REST API has been constructed, developing the functions which serve the HTTP requests, arriving to those URI, can be started. Next an example is shown, which is the handler function that processes the actual reservation requests using the URI ‘/app/v1/reserve’.

The function itself is very similar to the ones that serve normal web pages. The whole code for this one function is too long to be copied here, but the basic structure is shown next.

```
//handler function for the URI: "/app/v1/reserve"
func resRESTHandler(w http.ResponseWriter, r *http.Request) {
    loggedInUser, err := getCurrentUser(r)
    if err != nil {
        sendJSON(w, 401, map[string]string{
                        "Status": "401 - Unauthorized",
                        "Message": "This request needs
                        authentication. Please log in!"})
        return
    }
    switch r.Method {
        case "GET":
            //code to handle HTTP GET
        case "POST":
            //code to handle HTTP POST
    }
}
```

The first part of the function decides whether the user is authorized to access this page or not, based on the cookie that it provides in its request. If the user has already logged in to the reservation system, it will be allowed to view the reservations or make a new reservation. First one will be handled in the "*case "GET":"* branch, the second in the "*case "POST":"* branch of the function.

- In case the request was a *GET*, the only thing the server has to do, is fetch the reservations from the database by querying the tables, then construct the JSON response and send it.
- In case the request was a *POST* request, the server has to make some checks first to make sure the parameters of the reservation are valid and make sure that after inserting the reservation to the database there will be no conflict between two reservation trying to own the same device.

Deleting and updating an existing reservation is handled by another function, since it is using a different URI, '/app/v1/reserve/{id}'. Next the function is shown with some explanation.

```
//handler function for the URI: "/app/v1/reserve/{id}"
func resByIdRESTHandler(w http.ResponseWriter, r *http.Request) {
   user, err := getCurrentUser(r)
   if err != nil {
         sendJSON(w, 401, map[string]string{
                              "Status": "401 - Unauthorized",
                              "Message": "This request needs
                              authentication! Please log in!"})
         return
   }
   urlParam := r.URL.Path[16:]
   var regexID = regexp.MustCompile("^[1-9][0-9]*$")
   switch r.Method {
         case "GET":
                 //code to handle HTTP GET
         case "PUT":
                 //code to handle HTTP PUT [UPDATE]
         case "DELETE":
                 //code to handle HTTP DELETE
   }
}
```

This function is constructed in the same way that the one shown before. One difference is that it needs to extract the ID specified in the URL and use it when making queries, updates or deletes. This is accomplished with the help of the ‘regexp’ standard package. Using regular expressions provides a powerful tool for pattern matching with very little effort. The example above shows an example of using the ‘regexp’ package to make sure the ID contained in the URL is a valid ID, not zero, or a number with leading zeros, etc…

6 Client side implementation

In this chapter, the client side implementation of the Reservation system is detailed. On the client side, HTML, CSS and JavaScript were used extensively to achieve a good user experience.

6.1 Login page

On the next figure the login page is shown, which is used to authenticate the users before accessing the internal services offered by the server. Upon successful authentication, a cookie named ‘*session_token*’ will be set on the browser which will expire in 15 minutes, requiring the client to re-authenticate.

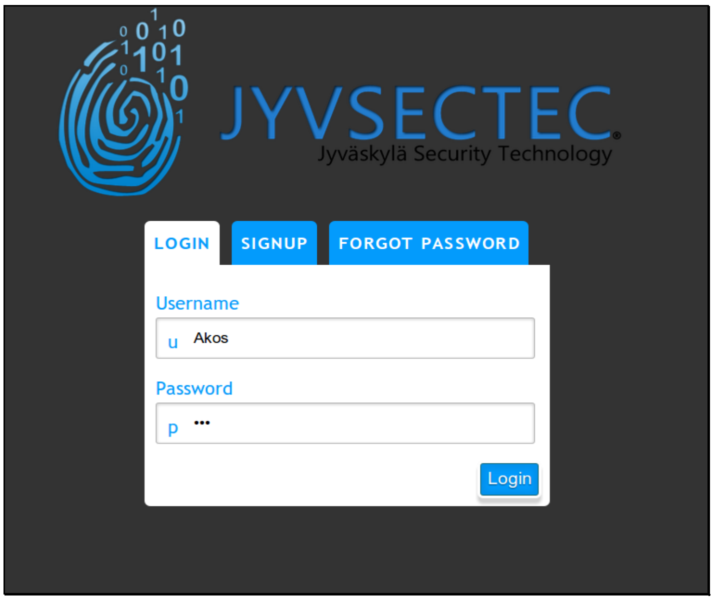
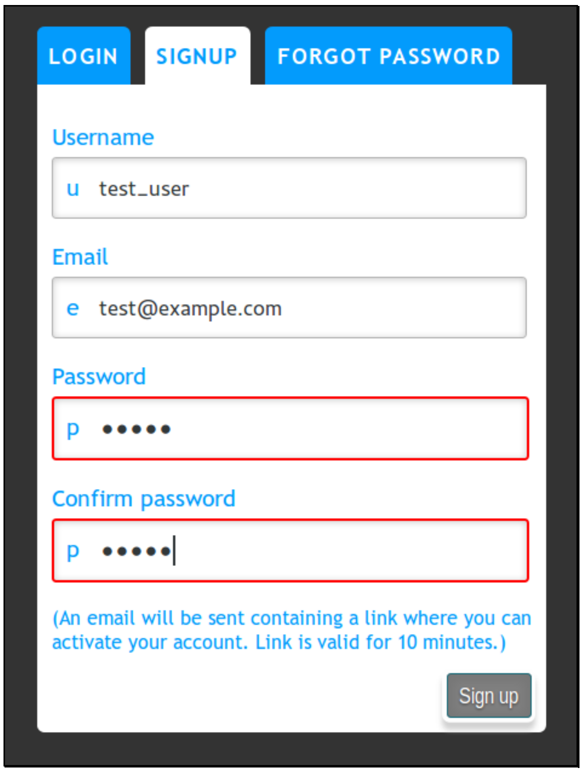


Figure 9. Reservation system login page.

On the second and third tabs of the login box, the client can sign up for a new account, or recover one that cannot be accessed because of a forgotten password. These features are implemented by sending an email to the specified address with a link to either activate or recover an account. With the help of jQuery, it is checked if the given password and the repeated password were matching, in case they were not, the border of the input fields will change to red color, until the problem is fixed, as shown in the following figure. However, this still does not eliminate the need to check every detail of the user input at server side. Next is a figure showing the password checking feature in action.



Username

u test_user

Email

e test@example.com

Password

p •••••

Confirm password

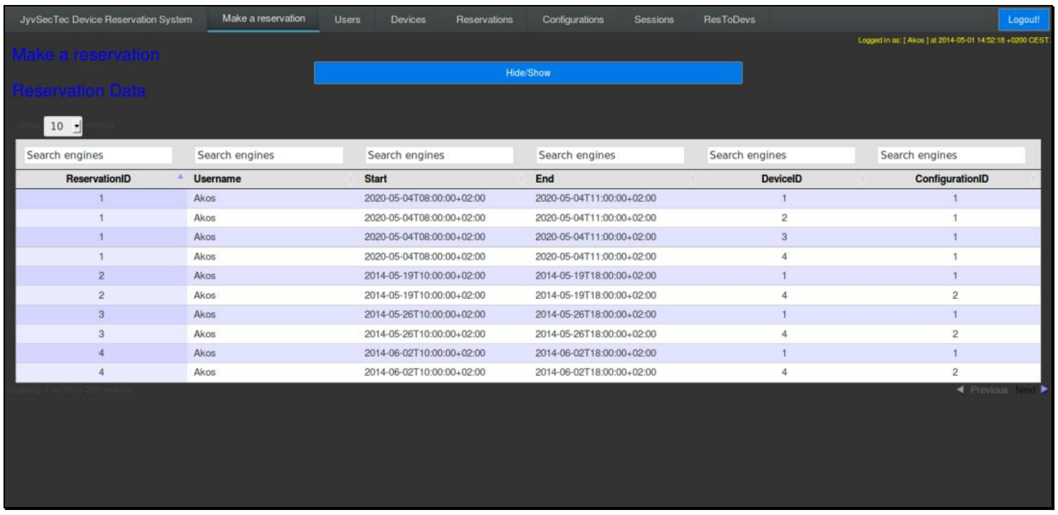
p •••••

(An email will be sent containing a link where you can activate your account. Link is valid for 10 minutes.)

Figure 10. Checking passwords using jQuery.

6.2 The page for making reservations

The webpage contains two main areas, one for a set of inputs that are used for adding, deleting or updating a reservation, and another for the jQuery Datatables plug-in, which will display all the reservations that exist in the database in a sort-able and searchable way. Here is the overall look of the reservation page opened in a Firefox browser.



Logged in as: [Akos] at 2014-05-01 14:52:18 +0200 CEST.

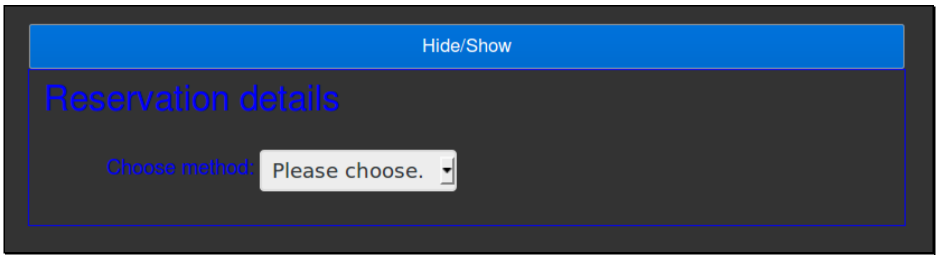
Make a reservation

Reservation Data

ReservationID	Username	Start	End	DeviceID	ConfigurationID
1	Akos	2020-05-04T08:00:00+02:00	2020-05-04T11:00:00+02:00	1	1
1	Akos	2020-05-04T08:00:00+02:00	2020-05-04T11:00:00+02:00	2	1
1	Akos	2020-05-04T08:00:00+02:00	2020-05-04T11:00:00+02:00	3	1
1	Akos	2020-05-04T08:00:00+02:00	2020-05-04T11:00:00+02:00	4	1
2	Akos	2014-05-19T10:00:00+02:00	2014-05-19T18:00:00+02:00	1	1
2	Akos	2014-05-19T10:00:00+02:00	2014-05-19T18:00:00+02:00	4	2
3	Akos	2014-05-26T10:00:00+02:00	2014-05-26T18:00:00+02:00	1	1
3	Akos	2014-05-26T10:00:00+02:00	2014-05-26T18:00:00+02:00	4	2
4	Akos	2014-06-02T10:00:00+02:00	2014-06-02T18:00:00+02:00	1	1
4	Akos	2014-06-02T10:00:00+02:00	2014-06-02T18:00:00+02:00	4	2

Figure 11. Reservation page overall look.

The first block can be shown or hidden by pressing the big blue button in the middle of the page. If pressed for the first time, it will drop down and display the following contents as shown on the figure.

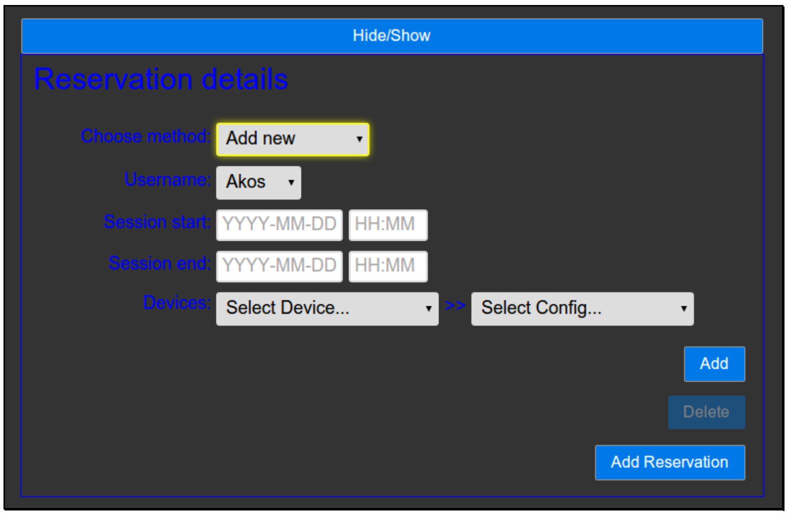


Reservation details

Choose method: Please choose.

Figure 12. Selecting the method to be performed.

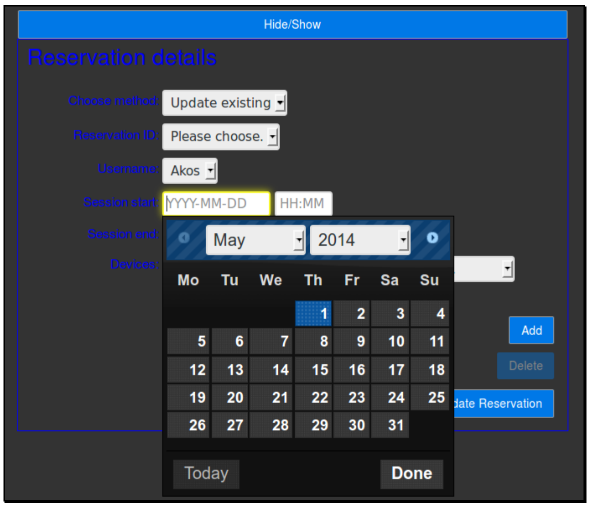
Next, the action which the client wants to perform can be selected. There are 3 options: ‘Add new’, ‘Update existing’ or ‘Delete existing’. Upon selecting one, the appropriate fields for that option will appear.



Choose method: Add new
Username: Akos
Session start: YYYY-MM-DD HH:MM
Session end: YYYY-MM-DD HH:MM
Devices: Select Device... >> Select Config...

Figure 13. Example of how a reservation can be added.

This option will require the client to specify a *username* for which the reservation will be assigned, a *starting* and *ending date* and *time* for the reservation and the *devices* with corresponding configurations. To make sure the dates are in the correct format, a JavaScript plug-in was used which will display a calendar to help choosing dates for the reservation. Next an example of that is shown. This way editing the input field can be disabled, and this will help to make sure the correct date format is maintained when making subsequent reservations (while this also does not eliminate the need to validate these inputs at server side).



Choose method: Update existing
Reservation ID: Please choose.
Username: Akos
Session start: YYYY-MM-DD HH:MM
Session end:
Devices:

Figure 14. Example of jQuery Datepicker user interface.

6.3 Utilizing the REST API through JavaScript

When all input fields are filled in, pressing the Add, Update or Delete reservation button will make an AJAX call to the REST API communicating the parameters through JSON. Next and example code is shown for adding a new reservation by sending an AJAX request.

```
var jsonObject = {};
jsonObject["userid"] = parseInt($("#user").val());
jsonObject["start"] = $("#date_start").val() + "T" +
                      $("#hour_start").val() + ":00Z";
jsonObject["end"] = $("#date_end").val() + "T" +
                    $("#hour_end ").val() + ":00Z";
jsonObject["devtoconf"] = deviceToConfig;

$.ajax({
     type: 'post',
     url: '/app/v1/reserve',
     dataType: "json",
     data: "[" + JSON.stringify(jsonObject) + "]",
     success: function (result) {
            //display some message about success of reservation
     },
     error: function (result) {
            //display some error message on the screen in case of error
     }
});
```

For this to work, first a new *object* needs be defined with fields named the same as the required JSON parameters. Next, these fields have to be filled populated, before the object can be turned into JSON using the built in *JSON.stringify(object)* JavaScript function.

When the JSON string is ready the AJAX request can be send with the help of jQuery library's AJAX function. Inside the request several fields have to be filled in like the request's type: '*POST'* and the URL which has to be the same one that REST API is programmed to listen on: *'/app/v1/reserve'*. As the *data* parameter inside the AJAX request, the result of calling the *JSON.stringify()* function of the newly generated object can be used enclosed inside '[..]' parenthesis. In the request, it can also be defined what action should be taken in case of success or failure of the sent request.

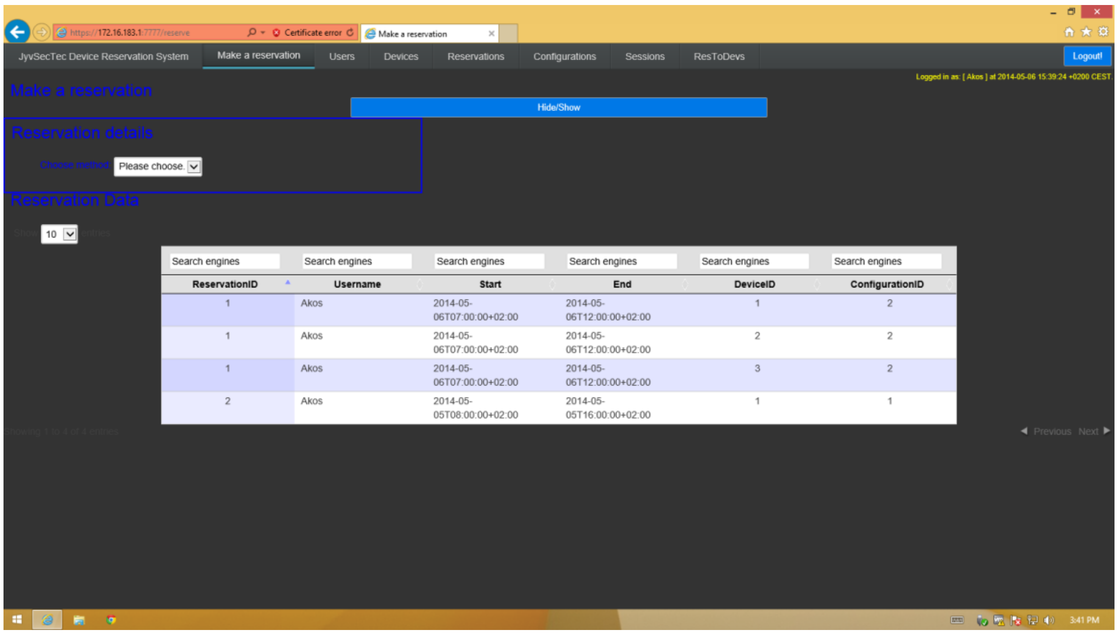
7 Testing

Testing of an application is a very complex but important process. It has to be carefully planned and executed on different levels. During and after the implementation of the system, the following tests were carried out.

7.1 Browser compatibility testing

Users are very likely to access the web page, that is being developed, using a number of different browsers, so taking this fact into consideration already in the development phase, and focusing on adapting browser specific settings for dealing with different browsers is considered a good practice.

During the implementation the project was constantly tested using a number of different web browsers on both Linux and Windows operating systems, including browsers like Firefox, Chrome and Internet Explorer. One problem was found with the use of Internet Explorer, not recognizing certain CSS settings, which in return breaks some animations on the web pages. For this it is recommended to use Firefox or Chrome, until a fix can be found for this.



Logged in as: [Akos] at 2014-05-06 15:39:24 +0200 CEST

Make a reservation

Reservation details

Choose method: Please choose.

Reservation Data

Show 10 entries

ReservationID	Username	Start	End	DeviceID	ConfigurationID
1	Akos	2014-05-06T07:00:00+02:00	2014-05-06T12:00:00+02:00	1	2
1	Akos	2014-05-06T07:00:00+02:00	2014-05-06T12:00:00+02:00	2	2
1	Akos	2014-05-06T07:00:00+02:00	2014-05-06T12:00:00+02:00	3	2
2	Akos	2014-05-05T08:00:00+02:00	2014-05-05T16:00:00+02:00	1	1

Showing 1 to 4 of 4 entries

Figure 15. Example of the UI glitch found when testing in Internet Explorer 11.

7.2 Security testing

Security testing of web applications is a very big topic in itself, and has to be applied extensively.

Two very important things to consider, is to prepare the server against HTML and SQL injection attacks, when there is a possibility to submit input through forms or AJAX requests. This is one of the reasons why every input coming from clients needs to be handled carefully and safely inside the server. One example of SQL injection attack is shown next.

```
SELECT count(*)  FROM users WHERE username='test' AND password='test';
```

This is a query that might be executed when a user tries to log in with the username ‘test’ and password ‘test’ entered in input fields of a login page. This information is then sent to the server and possibly inserted to a query like the above example, giving back the number of matches found. An SQL injection attack could be executed by writing this to the password field “test’; DROP TABLE users; --“ which then inserted into the query looks like this.

```
SELECT count(*)  FROM users WHERE username='test' AND password='test'; DROP  TABLE users;
```

With this command, an attacker can effectively delete an entire table which is very unfortunate. This is a very powerful security hole in systems that use SQL databases, and needs the developers to take actions to defend the system against these kinds of attacks.

Normally, SQL statements are created by joining strings together and executing the result, like in the following example.

```
str := fmt.Sprintf("SELECT * FROM users WHERE id=%d", user_id)
result := db.QueryRow(str)
```

Creating the query string by inserting the received content like in the example above is dangerous, because the content can come from an attacker trying to insert malicious SQL statement. With prepared (parameterized) SQL statements this can be avoided. Here is an example of prepared statements.

```
pstmt, _ := db.Prepare("SELECT * FROM users WHERE id=?")
result := pstmt.QueryRow(user_id)
```

Using this method will mean the query is not executed by inserting the string in the place where the question mark is, but rather taking the parameter of the ‘*pstmt.QueryRow()’* function and checking the database against this value.

The reservation system is tested against both HTML and SQL injection attacks. SQL injections were eliminated by using prepared statements. Protection against HTML injection is achieved through *escaping* all user input.

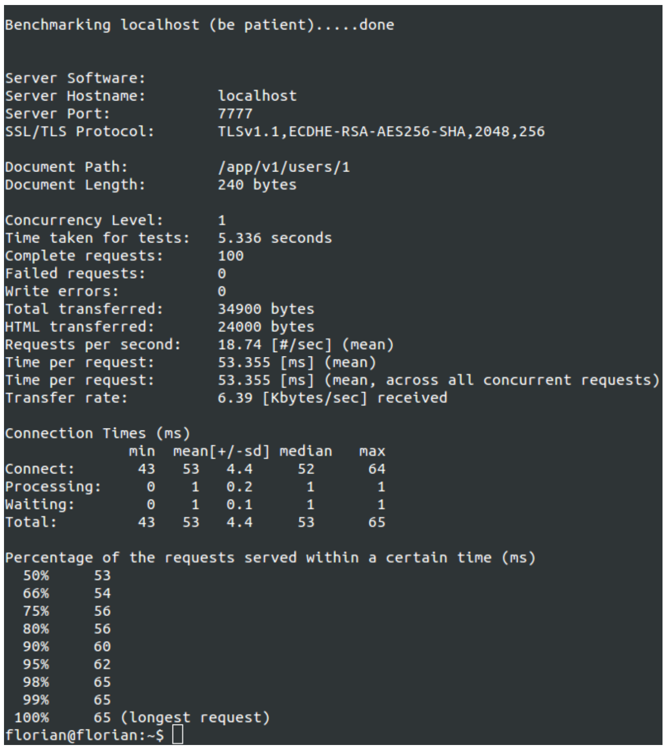
7.3 Load testing

To test how the server can handle large amounts of data, two different methods were used.

First one is a built-in tool in Linux system called ‘ab’, which can be used through the command line with the following command:

```
ab -n 100 -c 1 -C session_token='…' https://localhost:7777/app/v1/users/1
```

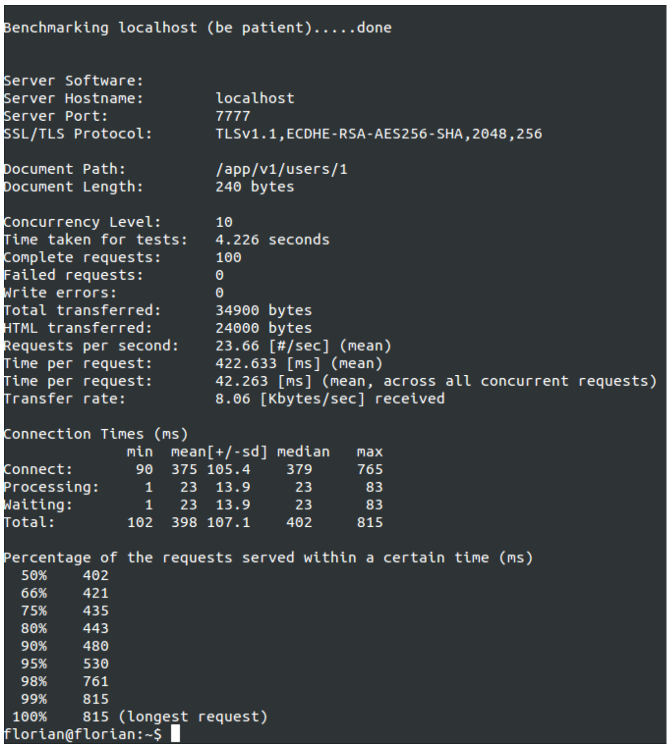
Executing the above command will make generate 100 HTTP GET requests to the given URL with all those requests being executed one after the other (no concurrency). This way the following results were received.



```
Benchmarking localhost (be patient).....done


Server Software:
Server Hostname:        localhost
Server Port:            7777
SSL/TLS Protocol:       TLSv1.1,ECDHE-RSA-AES256-SHA,2048,256

Document Path:          /app/v1/users/1
Document Length:        240 bytes

Concurrency Level:      1
Time taken for tests:   5.336 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      34900 bytes
HTML transferred:       24000 bytes
Requests per second:    18.74 [#/sec] (mean)
Time per request:       53.355 [ms] (mean)
Time per request:       53.355 [ms] (mean, across all concurrent requests)
Transfer rate:          6.39 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       43   53   4.4     52      64
Processing:     0    1   0.2      1       1
Waiting:        0    1   0.1      1       1
Total:         43   53   4.4     53      65

Percentage of the requests served within a certain time (ms)
  50%     53
  66%     54
  75%     56
  80%     56
  90%     60
  95%     62
  98%     65
  99%     65
 100%     65 (longest request)
florian@florian:~$
```

Figure 16. Result of first ‘ab’ benchmark.

For the second time, a concurrency level of 10 was set. Next is the result.



```
Benchmarking localhost (be patient).....done


Server Software:
Server Hostname:        localhost
Server Port:            7777
SSL/TLS Protocol:       TLSv1.1,ECDHE-RSA-AES256-SHA,2048,256

Document Path:          /app/v1/users/1
Document Length:        240 bytes

Concurrency Level:      10
Time taken for tests:   4.226 seconds
Complete requests:      100
Failed requests:        0
Write errors:           0
Total transferred:      34900 bytes
HTML transferred:       24000 bytes
Requests per second:    23.66 [#/sec] (mean)
Time per request:       422.633 [ms] (mean)
Time per request:       42.263 [ms] (mean, across all concurrent requests)
Transfer rate:          8.06 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       90  375 105.4    379     765
Processing:     1   23  13.9     23      83
Waiting:        1   23  13.9     23      83
Total:        102  398 107.1    402     815

Percentage of the requests served within a certain time (ms)
  50%    402
  66%    421
  75%    435
  80%    443
  90%    480
  95%    530
  98%    761
  99%    815
 100%    815 (longest request)
florian@florian:~$
```

Figure 17. Results of second ‘ab’ benchmark with increased concurrency.

It can be seen clearly that as the number of concurrent requests increases the time it takes for the server to process the requests and send back the response increases. This can be accepted until the system is not constantly receiving big amounts of client requests concurrently. At some point it might be feasible to deploy the application to a cluster of servers and use a load balancer to balance client requests so no single instance of the server gets overloaded.

The second method used in testing was by writing an automated application that makes a large amount of reservation. The idea for this test was to see how the serve can handle it, and also to see how the client interface can visualize those reservations. The specially written application for this testing purpose made a hundred reservations, each one being a week apart from one another. The server was able to handle the load, and when viewed it displayed the existing reservations in a nice and compact way, thanks to the Datatables jQuery plug-in which is used to display the reservations.

8 Results

The result of the practical training is a working web application which can be used for reserving devices. All the requirements of the application have been fulfilled. The system developed is ready to be used for reservation. It is using secure HTTP connection as required and user authentication as well. The user interface is intuitive and easy to use.

One of the main goals of the project, the development of the REST API has been successfully completed. It can be used for querying data from the database, and also to create, update and delete data in the database. Therefore, in the future it can be used to implement the system on other platforms like mobile devices.

However there are still some areas where the system can be improved. Here are some examples.

- Making the reservation web page mobile friendly by using responsive CSS design.
- Implementing an intuitive graphical representation of the reservations that are already in the database, so users can check more easily if the devices they wish to reserve are free or already reserved for a given period of time.
- Implementing the system on another platform, for example native mobile applications (Android, iOS, Windows Phone), or desktop applications on different operating systems (Windows, Linux, Mac).

9 Conclusion

During the time spent on implementing the system I have learnt many new and useful technologies, of which I had no knowledge before. Before I started working on it, I have never done web development or web application programming so this was all new to me, but my experience has been very positive.

Starting the development of the application was quite challenging, because I had to learn a lot of new and different technologies in a short amount of time, so that I can start writing the application. But in the end as I looked back I realized how much I learnt.

I really appreciate the fact that I got into developing this system, and learnt so much about things I thought I knew as a networking student, but in reality I did not. I would never have thought that web development can teach me so much about networking. I believe this knowledge is going to be very useful later in my career.

REFERENCES

Introducing JyvSecTec. N.d. Page on JyvSecTec's website. Accessed on 03.03.2014. Retrieved from http://jyvsectec.fi/en/presentation/

Introducing RGCE. N.d. Page on JyvSecTec's website. Accessed on 03.03.2014. Retrieved from http://jyvsectec.fi/en/rgce/

Web development. N.d. Article on Wikipedia's website. Accessed on 25.03.201. Retrieved from http://en.wikipedia.org/wiki/Web_development

Web 1.0. N.d. Article on Wikipedia's website. Accessed on 25.03.2014. Retrieved from http://en.wikipedia.org/wiki/Web_1.0

Web 2.0. N.d. Article on Wikipedia's website. Accessed on 25.03.2014. Retrieved from http://en.wikipedia.org/wiki/Web_2.0

Go history. N.d. Page on Golang's website. Accessed on 20.03.2014. Retrieved from http://golang.org/doc/faq#history

About Ken Thompson. N.d. Article on Wikipedia's website. Accessed on 20.03.2014. Retrieved from http://en.wikipedia.org/wiki/Ken_Thompson

Pike, R. Go at Google. N.d. Page in Golang's website. Accessed on 25.03.2014. Retrieved from http://talks.golang.org/2012/splash.article#TOC_13

Fielding, R., Gettys, J., Mogul, J. C., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. *Hypertext Transfer Protocol -- HTTP/1.1*. 1999. Pdf document at IETF's website. Accessed on 05.04.2014. Retrieved from http://tools.ietf.org/pdf/rfc2616.pdf

Rescorla, E. *HTTP over TLS*. 2000. Pdf document at IETF's website. Accessed on 06.04.2014. Retrieved from http://tools.ietf.org/pdf/rfc2818.pdf

Barth, A. *HTTP State Management Mechanism*. 2011. Pdf document at IETF's website. Accessed on 06.04.2014. Retrieved from http://tools.ietf.org/pdf/rfc6265.pdf

HTML. N.d. Article on Wikipedia's website. Accessed on 04.04.2014. Retrieved from http://en.wikipedia.org/wiki/HTML

What is CSS? N.d. Page on W3's website. Accessed on 01.05.2014. Retrieved from http://www.w3.org/standards/webdesign/htmlcss#whatcss

JavaScript. N.d. Article on Wikipedia's website. Accessed on 25.03.2014. Retrieved from http://en.wikipedia.org/wiki/JavaScript *jQuery*. N.d. Article on Wikipedia’s website. Accessed on 04.04.2014. Retrieved from http://en.wikipedia.org/wiki/JQuery

SQLite. N.d. Page on SQLite’s website. Accessed on 05.04.2014. Retrieved from https://sqlite.org/

ACID properties. N.d. Article on Wikipedia’s website. Accessed on 05.04.2014. Retrieved from http://en.wikipedia.org/wiki/ACID

Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures.* 2000. Pdf document on University of California’s website. Accessed on 02.04.2014. Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Representational State Transfer. N.d. Article on Wikipedia’s website. Accessed on 02.05.2014. Retrieved from http://en.wikipedia.org/wiki/Representational_state_transfer

UNIX time. N.d. Article on Wikipedia’s website. Accessed on 03.05.2014. Retrieved from http://en.wikipedia.org/wiki/Unix_time