Implementation of a Registry Management Component for Mobile Sampling Service

Henrik Toivakka

Bachelor's thesis
October 2015
Business Information Systems
Option of Software Development

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Tietojenkäsittely
Ohjelmistotuotanto

TOIVAKKA, HENRIK:
Implementation of a Registry Management Component for Mobile Sampling Service

Opinnäytetyö 49 sivua, liitteitä 3 sivua
Marraskuu 2015

Opinnäytetyön tavoitteena oli toteuttaa rekisterinhallintakomponentti liikkuvaan näytteenottojärjestelmään Mylab Oy:lla. Yritys valmistaa laboratoriojärjestelmiä Suomessa. Komponentti on osa suurempaa palvelintoteutusta, joka asennetaan asiakkaille ympäri Suomea. Palvelintoteutusta hallinnoidaan rekisterinhallintakomponentin kautta.

Opinnäytetyön tarkoituksena oli parantaa kykyä hallita mobiilipalvelua, luoda hallittava laiterekisteri ja tehdä mahdolliseksi laajentaa palvelua tuleville asiakkaille. Opinnäytetyön toimeksiantona oli mahdollistaa laitteiden pääsynhallinta ja rekistereiden hallinta web-käyttöliittymän kautta. Komponentin on tarkoitus olla luotettava ja käytettävä tuotantoympäristössä.

Toteutus koostuu JAX-RS-standardiin perustuvasta, REST-pohjaisesta, www-sovelluspalvelusta ja yhden sivun -arkkitehtuuriin perustuvasta web-sovelluksesta. Www-sovellus on toteutettu käyttäen tekniikoita Jersey, EclipseLink MOXy ja EclipseLink JPA. Web-sovellus on toteutettu käyttäen tekniikoita AngularJS ja Twitter Bootstrap. Toteutus on paketoitu yhdeksi asennuspaketiksi, joka voidaan asentaa suoraan asiakkaalle.

Projekti täytti kaikki vaatimukset ja on asennettu asiakkaan tuotantoympäristöön syksyllä 2015. Tuotosta käyttää yrityksen sisällä toimivat rekisterin ylläpitäjät. Projekti antoi arvokasta tietoutta ohjelmistokehityksen metodeihin ja kokemusta seuraavien versioiden tuottamiseen palvelusta.

Avainsanat: terveydenhuollon tietojärjestelmä, yhden sivun -arkkitehtuuri, rekisterinhallinta, Jersey, AngularJS

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business Information Systems
Option of Software Development

TOIVAKKA, HENRIK:
Implementation of a Registry Management Component for Mobile Sampling Service

Bachelor's thesis 49 pages, appendices 3 pages
November 2015

The objective of this thesis was to implement a registry management component for a mobile sampling service at Mylab Oy. The company is a major manufacturer of healthcare information systems in Finland. The component is a part of a server implementation that is installed for customers around Finland. The service installed for the customer is administrated through the registry management component.

The purpose of this thesis was to make administration of the mobile sampling service more intuitive, create a manageable device register and make it possible to expand the mobile service for future customers. The requirements set for the registry management component were the ability to control device access and administrate registries through web user interface. The component should be reliable and usable in production environment.

The implementation consists of a JAX-RS RESTful web service and an AngularJS single-page application. The web service is implemented with technologies such as Jersey, EclipseLink MOXy and EclipseLink JPA. The single-page application is implemented with technologies such as AngularJS and Twitter Bootstrap. The implementation is packaged into a single install file ready to be deployed to a customer.

The project met all the requirements given and was installed to the customer's production environment in the autumn 2015. The implementation will be used by the registry administrators from within the company. The project gave valuable knowledge for the future development and experience for implementing the next iterations of the application.

Key words: healthcare information systems, single-page application, registry management, Jersey, AngularJS

CONTENTS

ABBREVIATIONS AND TERMS

TAMK	Tampere University of Applied Sciences
cr	credit
LIS	Laboratory Information System
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
VPN	Virtual Private Network
SOAP	Simple Access Object Protocol
REST	Representational State Transfer
RESTful Service	Web Service built with REST architecture.
Service Component	Service providing interface between mobile applications and the laboratory information system.
CRUD	Create, Read, Update, Delete. Database actions, which define what is being done with the data.
XML	Extensible Mark-Up Language
ORM	Object Relational Mapping
DAO	Data Access Object
DAL	Data Access Layer
AJAX	Asynchronous JavaScript and XML
JAX-B	Java Architecture for XML-Binding
SAX	Simple API for XML
JAX-RS	Java API for RESTful Web Services
JPA	Java Persistence API. Application programming interface for abstracting database queries.
Component	Part of a greater implementation
SPA	Single-Page Application
DOM	Document Object Model
Framework	Reusable library of code. Provides abstracted functionality to make development easier.
Use case	Action performed by someone to achieve a goal. Used to image how the product is used.
Open Source	Source-code under free license.

Duty Sampling	Sampling procedure where the samples are taken in between sampling rounds and in acute cases.
Round Sampling	Sampling procedure where the samplers go through all units in the hospital and take samples from the patients. Happens every few hours.
SSL	Secure Sockets Layer. Cryptographic protocol.
Registry Administrator	The person administrating the registry of devices, services and print label definitions.
Tomcat context	An URL-address where to find the deployed web application in the Tomcat container.

1 INTRODUCTION

The principal of the thesis is Mylab Oy, a major manufacturer of laboratory information systems (LIS) in Finland. The system produced by the company is used in major hospitals and laboratories around Finland.

Mylab also has smaller products and innovations, of which most are developed to support the main business. One of them is the mobile sampling service, an integrated component to the laboratory information system. The mobile sampling service is used to take blood samples from the patients, but technically it is not restricted to only blood samples.

The service presents an accurate information of the samples and the sampling vessels in a mobile device. The information is important, because the sampler needs to know what samples are ordered and how they are taken from the patient. There are different kind of tests that can be done to a sample and they require samples to be taken in specific ways.

The mobile sampling service also makes sampling more accurate as the sample is instantly checked out after the sampling operation, giving an accurate sampling time. Getting an accurate sampling time was not possible before. It helps laboratory workers to get reliable results from the sample.

There is also a Bluetooth thermal printer that is used to print labels on the sampling vessels. The labels are used to identify the samples in the laboratory. Prior to the service, the information of the samples was on paper and the labels had to be printed before the sampling operation could be started. The service speeds up the sampling procedure, thus making the samplers work more efficient.

There are some problems with the mobile sampling service, which are becoming more important to solve as the service is being deployed around Finland. The device registries of the service are managed by hand, which makes it difficult to administrate the service. Updating of the mobile sampling applications is done by hand so distributing the updates to the customers is a burden. Controlling device accesses to the company network was difficult as it was done by modifying a text file. The registry management component brings solution to all of these problems.

The purpose of this thesis was to make administration of the mobile sampling service more intuitive, create manageable device register and make it possible to expand the mobile service for future customers.

The objective of this thesis is to implement a registry management application, which contains a RESTful web service and a web user interface with the technologies used in the company. Also techniques and project tools used are introduced to the reader.

The registry management component is a part of a server implementation that was installed to the customer in the autumn 2015.

2 SERVER ARCHITECTURE

The server architecture is introduced to bring some insight for the reader about the operational environment of the application. The registry management application is only a part of the entire server implementation and it might be hard to understand the concept without introduction to the other components of the server software.

The application is ran in a Linux-based server, which is located in a cloud computing service. With the help of the cloud computing service, the system is more scalable and easier to maintain. Cloud computing service allows taking regular snapshots, by which the server can be recovered if something goes wrong.

2.1 Apache Tomcat

"Apache Tomcat™ is an open source software implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. The Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket specifications are developed under the Java Community Process." (Apache Tomcat Home, 2015.)

Tomcat is a lightweight and reliable HTTP web server software. It is very customizable for different situations and can be used for many different purposes. Tomcat is used as a container for the router, service and registry management components.

Tomcat was mainly chosen because it is lightweight and supports SSL-encryption. There is lots of experience and knowledge of Tomcat in the company. The server contains two Apache Tomcats, which are called **tomcat-priv** and **tomcat-pub**.

2.1.1 Public Tomcat

Tomcat-pub is exposed to the internet and it acts as a gateway to the company's intranet. Most functionality provided by Tomcat container is disabled in tomcat-pub. The purpose is to provide secured connection and a routing service for the devices using mobile sampling service.

The Tomcat is secured with SSL-encryption. Every device has a unique certificate, generated in the company. The Tomcat can't be accessed from the Internet without a valid certificate. By using a secured connection, the communication between the client and the server is always encrypted. It also blocks all unwanted requests to the server.

2.1.2 Private Tomcat

Tomcat-priv is a container for service components and the registry management component, which provide server-side functionality for the mobile sampling applications. It can be accessed from the company's intranet through static VPN-tunnel, therefore it does not require a certificate.

2.2 Components of the mobile sampling service

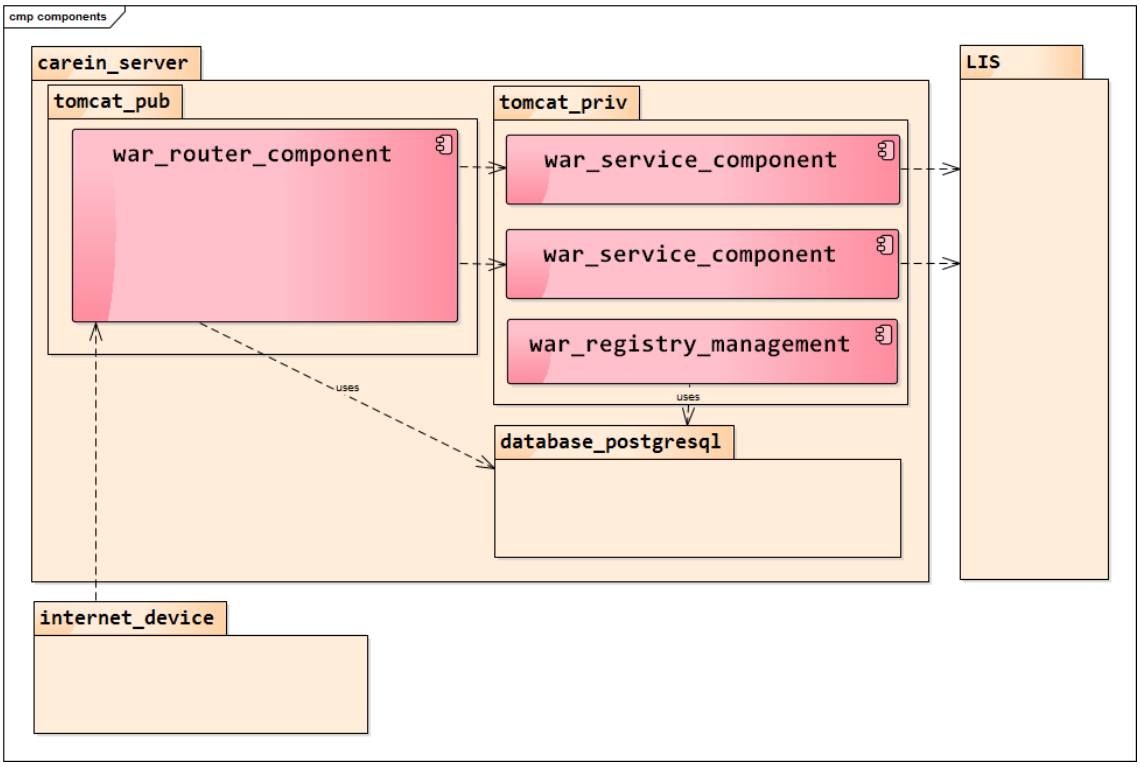


Figure 1. A component diagram of the server architecture.

As can be seen in the Figure 1. the service consists of different components. Most of them are dependent of the **router component**. However the mobile service is designed to be modular in such a manner that the service components are not dependent of each other. There is no limitation which components can be installed on the customer's server.

2.2.1 Mobile sampling applications

A mobile sampling application is a client-side application located in a mobile device. The application is the part of the service that uses the server and presents data to the end user. It is used in clinical environment such as hospitals. There are different sampling applications for different sampling procedures, such as round sampling and duty sampling. The sampling applications are developed on Android platform, but the service is designed to support client applications on different platforms.

2.2.2 Router component

Router component is located in **tomcat-pub** and contains the routing logic of the service. It is the most important component of the server. The purpose of the router is to receive all incoming requests and check if they have access to the service components in the server. A device can either have allowed or not allowed access to a service component. The router redirects the traffic to the desired service component. In a nutshell, all it does is receive a request, check the access status of the device and route the request to its destination.

2.2.3 Service components

The business logic of the request happens in a service component. Every mobile sampling application has a dedicated service component processing the incoming requests and queries the laboratory information system for data. Service components can also read or generate application specific information, which can't be acquired from the laboratory information system. The purpose of the service component is to provide interface and process the data for mobile sampling application, which increases the application performance in the mobile device.

Library components are used to share code between service components. Code reusability is important when creating information systems because they tend to have code overlap. Library components reduce the amount of work needed for implementations of service components, if the same functionality is already implemented elsewhere. Registry management does not use any library components, excluding Maven dependencies, thus they are only briefly introduced.

2.2.4 Registry management component

Registry management component is becoming necessary for administrating the mobile sampling service. The number of installed devices is increasing rapidly and it is becoming very hard to administrate them by hand. Every customer has their own instance of the server software, therefore the registry management is server-specific. It is independent of any other component, even the router, and can be accessed through static VPN tunnel from the company's network. It is used to administrate the server software installed for the customer.

As can be seen in the Figure 2. the registry management component consists of a JAX-RS RESTful web service made with Jersey framework, an AngularJS single-page application and uses PostgreSQL as the database.

The data of the customer's devices, services, print label definitions and laboratory information system interfaces is stored in the database and displayed through the AngularJS application. The component allows the registry administrator to modify the displayed data, distribute sampling application updates to the devices and control the device access.

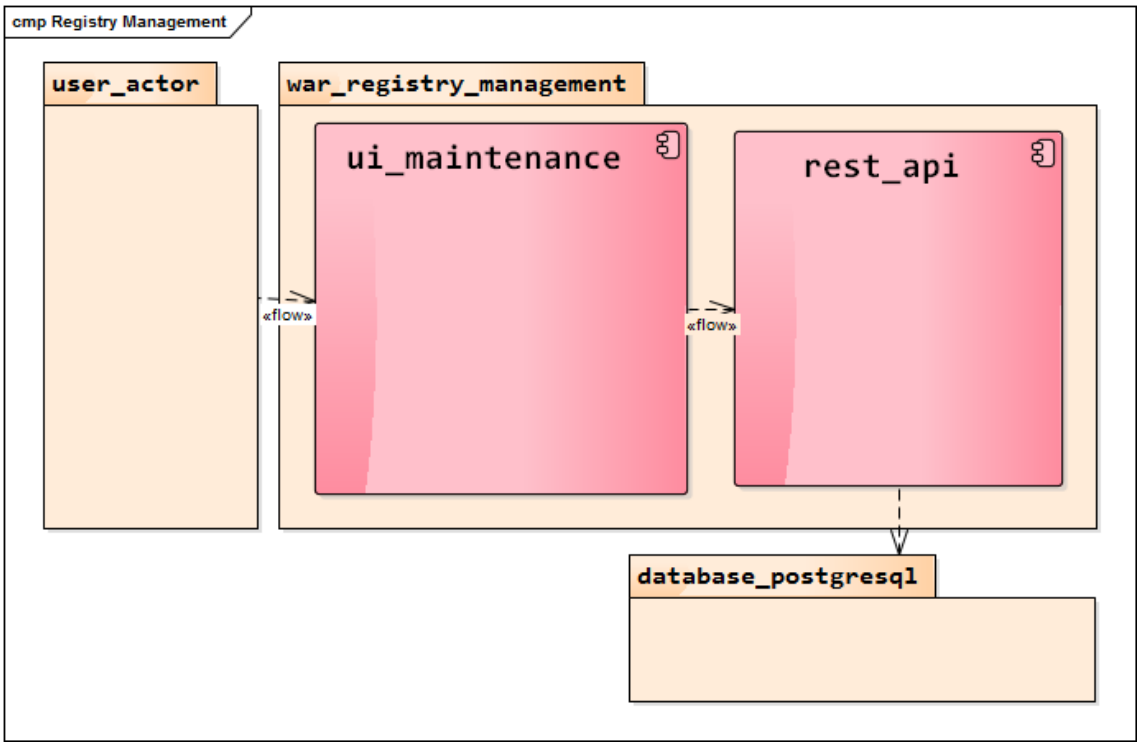


Figure 2. A component diagram of the registry management component.

2.2.5 Registry management domain

The registry management domain model diagram can be found in the Figure 3. The objective of the domain model is to provide device access control, be able to administrate print label definitions, device and service registries and distribute application updates to the mobile devices with auto-update.

Device access control allows the administrator to take action in case of theft or any other abuse of the service. The device access control was done before by modifying a text-based list. However administrating it has become harder as the number of devices in the service is incrementing.

All samples taken in hospital are marked with a label printed according to the print label definition. Sometimes print labels have to be modified. The print labels are customer specific data.

The updating of the mobile sampling applications was done by hand before the existence of the registry management component. The mobile applications can't be found in any app market such as Google Play Store or Apple Appstore. They were installed locally by downloading the Android application package into device's internal storage. The auto-update requires information about the application and service versions. The update can either be mandatory or voluntary depending on the importance. Mandatory updates are distributed by incrementing **minimum application version** in the InternalService-object which affects all devices attached to the service. Voluntary updates are distributed by incrementing **target application version** in the DeviceAccessToService-object and affects only the specific device. The objects are modeled in the Figure 3.

The purpose of the device groups is solely to make it possible to group devices, allowing to see which unit and group they belong to. The groups are made after the units of a customer, usually a hospital, and provide contact information of the person in charge.

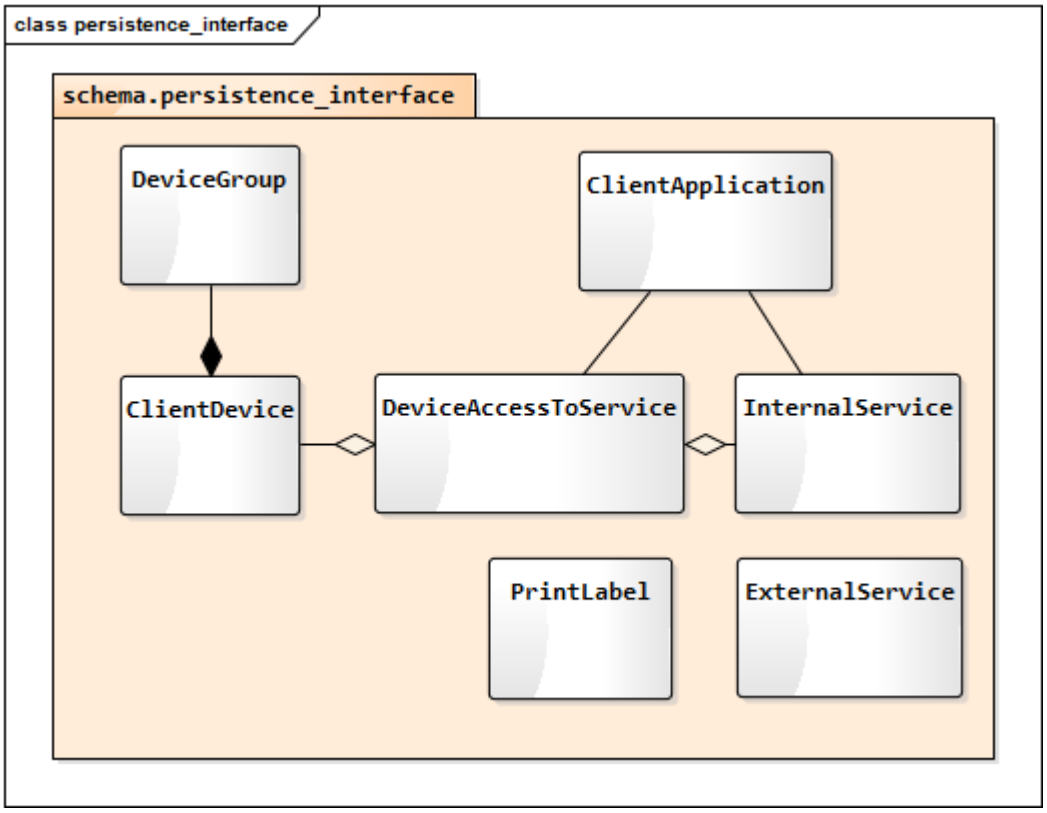


Figure 3. Registry management domain model.

2.3 PostgreSQL

PostgreSQL is open-source database implementation used by many large companies around the world. Being open-source and free is an advantage to look at, because it lowers the maintaining costs. A database must provide security, reliability, stability and extendibility. ACID-transactions, optimistic locking and connection pooling are necessary features and most database providers implement those features by default. (PostgreSQL Advantages. 2015.)

There are two database choices in the company, which are **PostgreSQL** and **Caché**. The reason for choosing PostgreSQL was because it is open-source, free to use and Caché is not generally a good database choice for this kind of a project, because the amount of data is not huge.

The database is the most persistent part of the application environment. A well-designed and established database is important for an enterprise level application, because changing the database is awkward after the application has been deployed into production environment.

2.3.1 Configuring database

Database configuration affects the performance, security and reliability of the whole system. Giving database users access only to the information needed is a good way to protect a database. Performance can be improved by using connection pooling and indexing.

A database should be connectable from trusted hosts or only from localhost, therefore the database cannot be queried without direct access to the server. The purpose is to create a secure database, which can only be accessed from trusted hosts.

2.3.2 Roles

Database roles are generic groups which can be granted privileges to execute actions in the database. Changing a role is reflected to all users under the role. A good practice is to have only the needed privileges on the database role. If the application only reads data, it should only have privileges to do that. If the role had privilege to insert data, it could lead to unexpected security vulnerability.

Roles can prevent connections from unwanted locations. A role for database admin could be assigned to be connectable only from the localhost. Connections from anywhere else than localhost are blocked.

```
CREATE role persistence_interface_user NOINHERIT;
CREATE role user LOGIN PASSWORD 'XXXXXXX' INHERIT;
GRANT persistence_interface_user TO user;
```

Figure 4. Example of creating database roles.

In Figure 4. there are two roles created. A role for accessing database tables and a role for logging in to the database. It is optional to set a password for a role, but it is always recommended for login roles. The role ‘persistence_interface_user’ is granted to the user, so the user can login and access tables in the database.

2.3.3 Schemas, tables and views

A database schema can be thought as a container for logically grouped data. Schemas provide extra level of security and a way to group data more effectively in the database. Roles and users can be granted access to different schemas.

Tables are the structured collection of data in the schema. Tables consist of rows and columns, where the actual data can be found. Table cells may contain any basic type data or references to other database tables.

Views are pre-made queries to the database, which query the data every time they are called. The advantage of views are that they provide extra security as they can be set to read-only mode and consist of the cells from other tables. (PostgreSQL 9.4.4 Documentation. 2015.)

```
CREATE SCHEMA persistence_interface;
GRANT USAGE ON SCHEMA persistence_interface TO persistence_interface_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA persistence_interface GRANT SELECT, UPDATE,
INSERT, DELETE ON TABLES TO persistence_interface_user;
```

Figure 5. Example of creating a database schema.

In Figure 5. database schema is created. Privileges are granted to a role created in Figure 4 in such a manner the users possessing the role can access the schema. The role is also granted with select, update, insert and delete privileges on all tables on the schema. After creating the schema, the tables and views can be created to the database. A good practice for development is to have SQL-scripts for quickly formatting the database and filling it with test data if needed. For this project there is a script for recreating all tables and views. Example shown in Figure 6. only contains script for one table. There is also a database dump file used for populating the tables with test data.

```
-- Drops table, if it exists --
DROP TABLE IF EXISTS persistence_interface.client_application;

-- Creates the table and its unique indexes --
CREATE TABLE persistence_interface.client_application (
            id SERIAL NOT NULL,
            name VARCHAR(30) NOT NULL CHECK (name <> ''),
            version INT NOT NULL,
            download_url VARCHAR(100) NOT NULL,
            CONSTRAINT pk_client_application PRIMARY KEY(id));
CREATE UNIQUE INDEX ui_unique_application_version ON
persistence_interface.client_application (name, version);

-- Grants the database user privileges to the table's sequence --
GRANT USAGE, SELECT ON SEQUENCE persistence_interface.client_application_id_seq TO
xxx;
```

Figure 6. Example script for creating a database table.

2.3.4 Users

It is generally a good practice to have different database users for different purposes in database. A database should have one root user that creates the database, but does not have access to the schemas. There should be a user for creating schemas and tables and it should have privileges to modify data in that particular schema. This is done to prevent harmful actions in the database in case of a security breach. Lastly there are users who can read and modify data, but can't change the table structure. These privileges are generally granted by creating roles for them. For example there could be 3 database users for different service components that all have a role 'persistence_interface_user'.

2.3.5 Transactions

Transactions are a way to make sure the data stays intact between operations in the database. It prevents multiple operations to be done at the same time by locking other users out while executing a transaction. Reading database should be kept as light as possible while in transaction, because it locks other users out while reading. However only inserts, updates and deletes need a transaction since they modify the data. Read operation does not modify the data, so a transaction is not needed. Good practice is to have one complete operation within one transaction as that leaves less space for transaction failure. Transaction usually fails when the given parameters are either out of bounds or invalid.

A failed transaction rolls the database state back to where it was in the beginning of the transaction. After a failure the next transaction in the queue is executed. However there is a problem with transactions, when two users are modifying the same data at the same time. Both users want to change the data, but neither of users know about the other user modifying the data. There is a possibility this could break the application or corrupt the data. (R.Elmasri & S.Navathe 2004, 558-564).

2.3.6 Optimistic concurrency control

Optimistic concurrency control (optimistic locking) is a unique serial column in the database table, which can be used to track the version of the table. After every successful transaction the serial number is incremented to let next transaction know the data has been updated. If there are two transactions changing the same table, the latter gets rejected because the serial number has changed after the first transaction. In this case the data in the client application needs to be updated to reflect the database changes. The purpose of optimistic locking is to ensure that the data of user A is never overwritten by user B without the users being aware of it. (R.Elmasri & S.Navathe 2004, 599-604).

There are also other ways to do concurrency control, but the idea behind optimistic locking is to read, validate and write in one phase. It is called optimistic because it is assumed there is not much interference during transactions. However if interference happens often, optimistic locking might not be the best choice.

3 ANGULAR.JS WEB APPLICATION

The user interface is required to be web-based and work on all common browsers. It is supposed to present data in visual form to a registry administrator. The goal is to allow the registry administrator to administrate devices, applications, print label definitions and services. It is also used for device access control and distributing application updates. The user interface is designed for professional use and has high learning curve for some features, like distributing updates.

The user interface is implemented using AngularJS framework and open-source libraries such as UI-Bootstrap and Angular-translate. The styling is done with Bootstrap CSS-library. The implementation relies on frameworks because it reduces the amount of code maintenance.

AngularJS is an open-source framework maintained by Google and community of developers and corporations. AngularJS is used to create **rich internet applications** (RIA). It is quick and simple to develop and very extensible framework allowing to create functionality for web sites. AngularJS was chosen because of its popularity and being open-source framework.

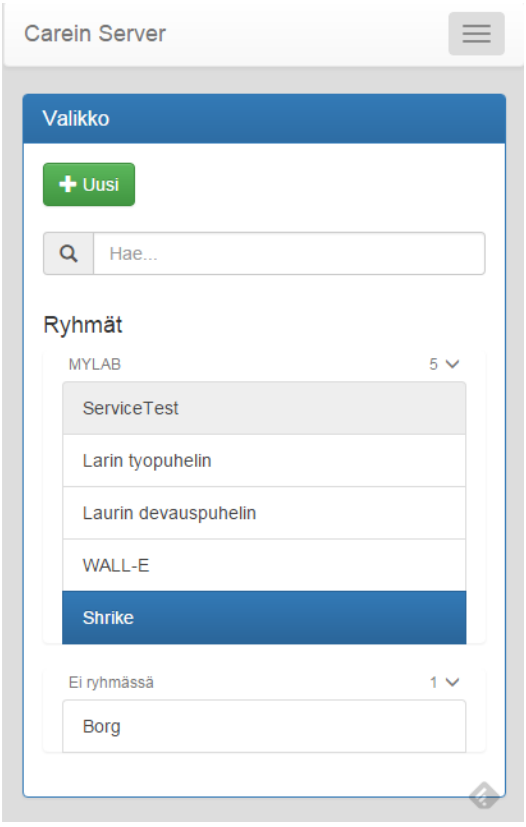


Figure 7. User interface in mobile view.

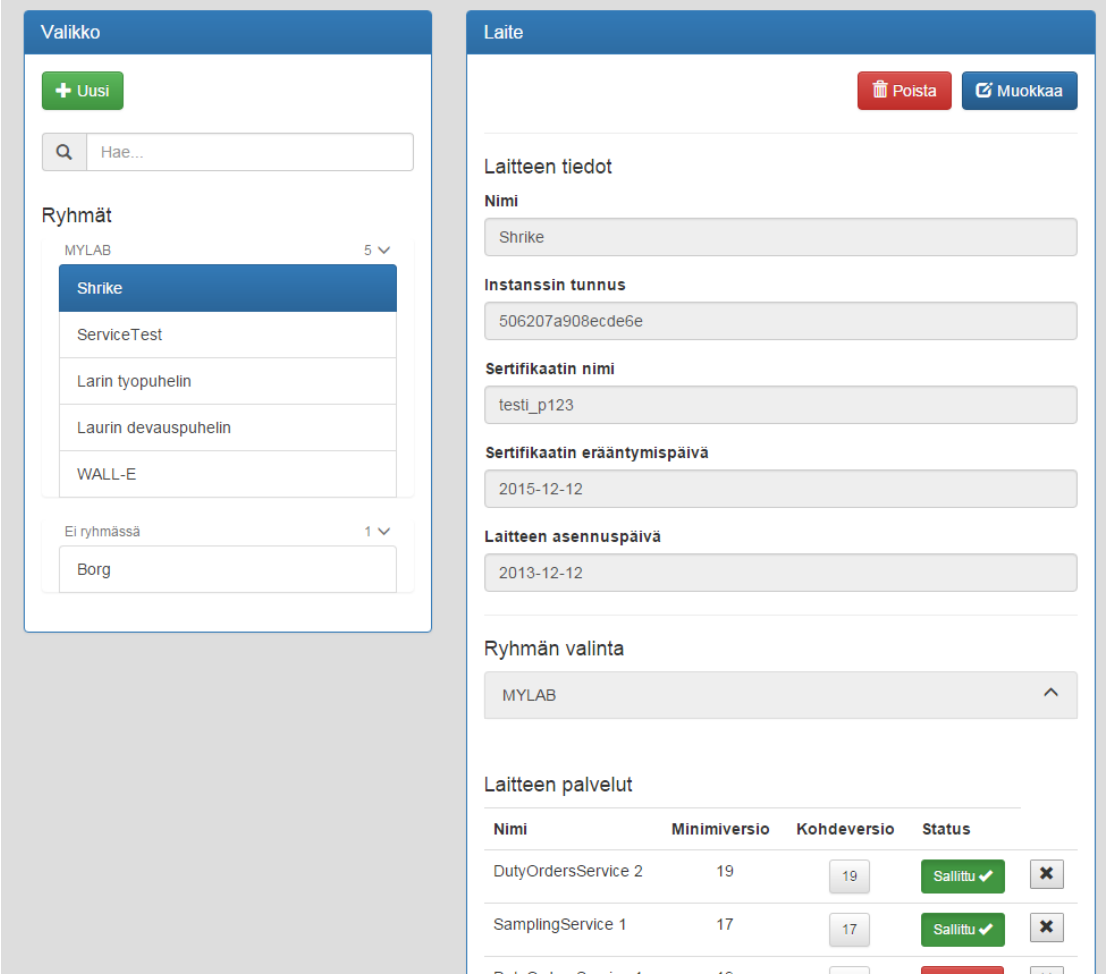


Figure 8. User interface in desktop view.

In Figures 7-8 is shown the user interface in mobile and desktop views.

3.1 AngularJS single-page application

Single-page application (SPA) is a web application with the purpose of providing better user experience than traditional web applications. The difference between a single-page application and a traditional web application is that the single page application is initially loaded from the server and fragments of the application are refreshed with AJAX every time action is executed in the user interface. The initial load of the application is slower, but after that a single-page application works faster than a traditional web application.

The advantage of using single-page application architecture is noticed when using the application in a slower network as the architecture reduces the amount of requests to the server and only fragment of the page is loaded at the time. In large-scale applications,

SPA increases the server performance because most of the computing is done in the client. However in this scenario the server load is not an issue, but it should be taken in consideration for future.

The disadvantage is the need for JavaScript support in the browser. However all modern browsers support JavaScript and have it enabled as default. Navigation in the application breaks the browser history in some cases and that must be taken in consideration as user might want to use forward/back buttons in the browser.

The navigation problem is resolved by **ngRoute**-module of the AngularJS. It provides JavaScript routing for the application, which uses URL-paths for the navigation. The JavaScript routing works inside Tomcat container by using #-symbols in the URL-path after the Tomcat context path changes to the AngularJS application's path.

3.1.1 Utilization of AJAX

AJAX is term for using combination of HTML, DOM, HTTP requests and formatted messaging techniques together. It works in the background, so user will not notice its presence. All most common web browsers support it. It enables dynamic exchange of data with the server in small fragments. The purpose of AJAX is to improve load times, lower network traffic and increase usability as only fragment of the application is loaded when the state changes.

AJAX is used for all HTTP-requests in the AngularJS application. It creates dynamic feeling for the end-user as the data updates every time an action is executed. AJAX is used through an in-built $http-service of the AngularJS.

Every HTTP-request in the application goes through the function shown in Figure 9. The function is used for creating custom requests to the server and handles the outcome with either success or error –functions. On success, the data is resolved and returned to the controller. On error, the data is rejected and an error message is broadcasted to the controllers. The function is called from a service, which implements the actual requests. The function shown in Figure 10. calls the http-function and stores the data to the master store, where it can be loaded to the controllers.

```
function http(url, method, payload, errorMessage, headers) {
    var defer = $q.defer();

    $http({
        method: method,
        url: url,
        data: payload || {},
        headers: headers || {}
    })
    .success(function(success) {
        defer.resolve(success);
    })
    .error(function(error, status, headers, config) {
        defer.reject(error);
        $rootScope.$broadcast('HttpErrorEvent', status, errorMessage);
    });
    return( defer.promise );
}
```

Figure 9. Example function used as the base for all requests going through the client.

```
function getDevices(url) {
    return api.connect(url, 'GET').then(function(response) {
        self.store.devices = response;
    });
};
```

Figure 10. Example function from the HTTP-service.

3.1.2 MVW architecture

Model-View-Whatever is a paradigm used by the AngularJS community. MVW is an architectural pattern for implementing user interfaces. The great thing about MVW is that it does not limit the developer to one pattern, but allows using whatever suits the situation. The purpose of architectural patterns is to make code easier to maintain and develop by keeping business and presentation logic separate.

In this project the approach used is **Model-View-Controller**. The model is located in the RESTful service component and it is retrieved to the StoreService. Controller loads the model and it is shown in the view. Actions in the view trigger actions in the controller, which manipulate the data and send it back to the RESTful service. The data can also be manipulated with custom directives injected to the view.

3.1.3 Bidirectional data binding

Bidirectional data binding is one of the most notable features of AngularJS. It reflects the data instantly to the model from the view and likewise, when the data in model is modified, it is reflected back to the view. Bidirectional data-binding creates dynamic feeling to the web application as input and actions are instantly noticed in the application state.

It brings some problems for usability as it may confuse the user and make room for errors. Sometimes users are confused when the model updates instantly, because other websites do not function that way. However it is hard to change built-in features in a framework, so the restrictions were taken in consideration when choosing the framework.

3.2 The principles of the AngularJS implementation

AngularJS application consists of different modules that have different purposes. Some of the most basic modules are **controllers**, **directives** and **services**.

3.2.1 Controllers

In the Figure 11. is shown a simple AngularJS controller. The controller inherits from the ParentController that has functions and attributes shared between all controllers. The ParentController was found as a good practice to keep the code overlap in minimum and to reduce the overall size of the controllers.

The controller has native AngularJS providers as dependencies. There are also a custom services as dependency that contain application business logic. The dependencies are injected to the controller by the AngularJS $injector service. It is called **dependency injection**. (AngularJS Dependency Injection, 2015.)

The controller contains mostly user interface logic and the injected services contain the logic for communicating with the RESTful web service. In the last lines of the Figure 11. the application data is retrieved from the StoreService when initializing the controller. The purpose of StoreService is to store all application data temporarily, so the views can be loaded instantly for the user. The data is updated when the user performs an action in

the user interface. In the Figure 12. is shown an example of using functions and attributes from the controller in the HTML-view.

```
app.controller('AccessMonitoringController', ['$scope', '$controller',
'StoreService', 'URLFactory', function($scope, $controller, StoreService, urls) {

    // Extends from the ParentController.
    // ParentController contains functionality shared between all controllers.
    $controller('ParentController', {$scope: $scope});

    // Initialize variables

    $scope.selectGroup = function(group) {
        $scope.selectedGroup = group;
    };

    $scope.isSelectedGroup = function(group){
        return $scope.selectedGroup === group;
    };

    $scope.selectAll = function() {
        $scope.selectedGroup = {};
    };

    // Get data from StoreService for populating the views.
    $scope.store.clientAccess = StoreService.store.clientAccess;
    $scope.store.devices = StoreService.store.devices;
    $scope.store.groups = StoreService.store.groups;

}]);
```

Figure 11. Example of simple AngularJS controller.

```
<ul class="list-group">
    <li ng-click="selectAll()"
        class="list-group-item"
        ng-class="{ active : isSelectedGroup(group) }">
        {{ "menuItems.allItems" | translate }}
    </li>
    <li ng-repeat="group in store.groups"
        ng-click="selectGroup(group)"
        class="list-group-item"
        ng-class="{ active : isSelectedGroup(group) }">
        {{ group.groupName }}
    </li>
</ul>
```

Figure 12. Example of using controller functions in the view.

3.2.2 Directives

A directive is an AngularJS specific marker on a DOM element. The directive tells AngularJS HTML compiler to attach functionality of the directive into the DOM element.

AngularJS code stays clean when hiding reusable code behind a directives. Directives should be designed as reusable components because they can be used many times in an application.

In Figure 13. is search component, used in multiple views. The directive returns a rendered component to the view. The directive is given the information about the list the search is made from, the property being searched and a callback function as an attribute. In Figure 14 is shown the template, which is rendered to the actual view in the place of mark-up shown in Figure 15.

```
app.directive('searchBar', function() {
    return {
    // Restricts the directive to Element-only
    restrict: "E",
    // Directive template
    templateUrl: "backend/templates/components/search/component.search.bar.html",
    // Get attributes from the view
    scope: {
        list: "=", // Gets the list where to search from
        property: "@", // Property to search
        callback : "&callback" // Callback function
    },
    link: function(scope, element, attrs) {
        // Initialize variables
        scope.isVisible = false;
        scope.search = '';

        scope.selectItem = function(item) {
            scope.callback({ listObject : item });
            scope.search = undefined;
        };

        scope.$watch('search', function(value) {
            if(!value) {
                scope.isVisible = false;
            } else {
                scope.isVisible = true;
                scope.search = value;
            }
        });
    }
});
```

Figure 13. Example of AngularJS search bar directive.

```
<div class="input-group" id="search">
<!-- Double <span> because of a bug in the Bootstrap.css -->
<span class="input-group-addon"><span class="glyphicon glyphicon-
search"></span></span>
    <input type="text"
            maxlength="30"
            class="form-control"
            placeholder="{{ 'menuItems.searchPlaceholder' | translate }}"
            ng-model="$parent.search"
            popover="{{ 'menuItems.searchPopOver' | translate }}"
            popover-trigger="mouseenter">
</div>
<ul class="list-group search-overlay" ng-style="elementWidth" ng-show="isVisible">
    <li ng-show="results.length == 0" class="list-group-item">
        <span class="glyphicon glyphicon-search"></span>
        <span>
            {{ "searchComponent.noItemFound" | translate }}: {{ search }}
        </span>
    </li>
    <li class="list-group-item"
        ng-click="selectItem(device)"
        ng-repeat="device in ( results = (list | filter : search ))">
        {{ property !== undefined ? device[property] : device }}
    </li>
</ul>
```

Figure 14. Search bar template

```
<search-bar
    list="store.labels"
    property="name"
    callback="select(listObject)">
</search-bar>
```

Figure 15. Example of using the search bar directive.

3.2.3 Services

AngularJS services are used to share functionality across the application. They are injected to the application by using dependency injection. The service object is derived from **provider** recipe. There are different recipes on top of the provider recipe, but only **service** and **factory** recipes are used in the project. Services are singletons, which means that there is only one instance of the object in the lifetime of the application. (AngularJS Providers, 2014.)

The StoreService is shared between all controllers and delivers the data retrieved from the RESTful web service to the controllers. Difference between StoreService and the ParentController is that ParentController extends the controller with its functions and at-

tributes. However the functions of the StoreService are only usable from within the controller, they are not defined in the same context as the controller. In Figure 10. is shown a function for retrieving device data that is located in the StoreService. The function can be called by any controller the StoreService is injected in.

In Figure 9. the http() –function is the base function for all HTTP-requests in the application. The function is called from the StoreService, which creates custom requests with the function.

The difference between service and factory recipes is that service returns instance of the invoked function. Factory recipe returns the value returned by invoking the function.

3.3 Angular-translate

Angular-translate is a library component used for localization in AngularJS applications. Localization is not a requirement, but it is a thing that should be taken in consideration in the early development. Using localization enables usage of different languages within the application, which is important in Finland because of its bilingualism.

Implementing localization with angular-translate happens by using angular-translate directive in the controller or in the view. In the Figure 16. error message is translated with the AngularJS directive. The directive translates the text automatically, when the language is changed.

In the Figure 17. error message is translated by using the directive within the controller. In this scenario the translated string is used to inject custom error messages, if any errors occur on the server. Both ways of using the directive are applicable in certain situations. The actual translations are located in translation files containing key-value pairs for the translations. Translation files are in JSON-format.

```
<span class="help-block red-text"
    ng-show="forms.deviceForm.deviceid.$error.unique">
    {{ "formValidation.uniqueError" | translate }}
</span>
```

Figure 16. Example of using angular-translate directive in the view.

```
var errorMessage = $translate.instant('errorMessages.genericDeleteMessage');
```

Figure 17. Example of using angular-translate directive in the controller.

3.4 UI-Bootstrap

UI-Bootstrap is a library for AngularJS and it contains native AngularJS components for Bootstrap. Some of the Bootstrap components require Bootstrap's JavaScript-library, which is not used in this project because there is AngularJS already.

The library has little use in the project, because the generic components it provides do not work in the desired way. UI-Bootstrap components have been used as placeholders in the development and they have been replaced with tailored components over time.

3.5 Testing

AngularJS is designed with testability as one of the design philosophies. JavaScript code has really good expression power, but is also challenging to debug as the compiler does not give any exact information about errors. JavaScript code should be tested carefully before deployment into production environment.

Testing can be either done by having a person to test the use cases. It can be a burden after the application has been tested a few times. There is a way to implement unit testing for AngularJS application by using the Jasmine framework. Jasmine is a behaviour-driven JavaScript testing framework used to test individual actions. (AngularJS Unit Testing, 2015.)

On the AngularJS side of the project unit testing is not implemented and the testing the user interface is done by hand. This decision was made because of limited time resources. However, the unit tests are focused on the REST interface. Testing the server ensures the

database will stay intact because it will not permit any faulty actions. It prevents the possible bugs from user interface to have any impact on the database.

3.6 Twitter Bootstrap

Twitter Bootstrap is Twitter's HTML, CSS and JavaScript –framework for mobile first responsive web pages. In this project the HTML and CSS libraries are used. Responsivity on the web application is not required, but it is still implemented for future purposes.

Bootstrap was chosen because it is popular and widely used framework in the industry. It has consistent theme and can be modified with the Bootstrap customize tool (Bootstrap Customize, 2015). Components can be brought to the AngularJS with the module called UI-Bootstrap or they can be refactored to support AngularJS.

Best features of Bootstrap are the **grid system, consistent theme** and the **component library**. It enables ability to rapidly develop user interface using placeholders that also work on mobile devices.

The grid system is a powerful tool for creating responsive layouts. The system uses rows and columns for dividing the content for different screen sizes. One row may contain up to 12 columns, which can be stacked for smaller screens. The rows and columns are controlled with predefined classes, and mixins for more semantic layouts. (Bootstrap CSS, 2015.) Examples of mobile and desktop user interfaces made with Bootstrap can be found in Appendices 4-7.

Consistent theme and component library enable development without touching the CSS-styles. The developer can focus on the functionality of the application and the layout can be polished and customized later. The purpose of focusing on functionality is to get fast results while creating prototypes or pushing new features out.

3.7 Discussion of AngularJS development

AngularJS is simple and rapid to develop in the beginning of the project. Developing becomes considerably harder when doing something not in tutorials or the official documentation. The AngularJS documentation is comprehensive, but there are a lot of poor examples and the naming conventions are sometimes confusing.

Whole application can be made modular by using directives and providers. Using components will reduce the amount of bugs as the bugs have to be solved only once from the component's code. There are a lot of community created libraries for AngularJS, making development simple, but many of them are not production quality.

Most problematic part of the AngularJS is the design principal, where developer has to maintain the HTML and the JavaScript code at the same time. Having the code in two places makes maintaining difficult.

MVW architecture is useful, however the disadvantage it creates is pretty obvious. There needs to be certain coding conventions so the code stays readable to everyone. Every feature implemented should be done with same syntax and function structure. AngularJS syntax is basically JavaScript, but it has little do with native JavaScript. Even though AngularJS has some problems, it can still be currently considered as one of the best JavaScript frameworks.

4 JAX-RS RESTFUL WEB SERVICE

The RESTful service is implemented with the Jersey framework. It also uses EclipseLink MOXy to generate JSON-formatted messages and EclipseLink JPA to access the PostgreSQL database. In nutshell, the service receives a request, executes the database actions and generates a response.

4.1 Web Service techniques

Generally there are two popular approaches for implementing an application programming interface for feeding data to a client-side application. **Simple Object Access Protocol** is a protocol for applications to exchange data with each other over a network. SOAP is used for sending messages through Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP) protocols into a single address, and the receiving web service processes the message. (Simple Object Access Protocol (SOAP) 1.1. 2000.)

There is a modern alternative for SOAP called **Representational State Transfer** (REST), which is a better choice for a single-page web application. REST is a set of conventions on how to use HTTP. (Bill, B. 2010, 4-6). REST uses URL-paths and HTTP-methods for navigating in the API, which makes the API cleaner and easier to understand. In this project REST architecture is used.

4.2 Message formatting

The most common message formats are **XML** and **JSON**, both widely used in web services. Difference between them is that XML is a markup language syntax and JSON is a way to present objects. JSON is specifically developed to be used with JavaScript and is a more lightweight option than XML. Advantages of XML is that it represents the data better since it has attributes and is more structured, but comes with a downside of being more complex and needs more resources to be manipulated. In Figures 18 and 19 there is the same array of users in different syntaxes to show the contrast.

In the registry management component only JSON is used because JavaScript can parse it natively. The AngularJS client consumes the data as JSON. However JAX-B can handle both formats without additional code.

```
{
     users: [
          {
               "name" : "foo",
               "age" : 21
          },
          {
               "name" : "bar",
               "age" : "35"
          }
     ]
}
```

Figure 18. Example of JSON syntax. An array of users.

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
     <user>
          <name>foo</name>
          <age>21</age>
     </user>
     <user>
          <name>bar</name>
          <age>35</age>
     </user>
</users>
```

Figure 19. Example of XML syntax. Same array of users as in Figure 18.

4.2.1 EclipseLink MOXy

MOXy is the EclipseLink's implementation of the JAX-B specification. JAX-B specification stands for Java Architecture for XML-Binding. It provides object-mapping for XML and JSON documents, which can be used to **marshal** or **unmarshal** formatted messages. Marshalling stands for generating a document from the object-mapping and unmarshalling stands for consuming the message into the object-mapping. The object-mapping presents a message as tree-based organization and content of the document.

Meet-in-the-middle is a technique offered by MOXy which enables mapping a document into Java-classes and avoid static coupling with a single schema. It brings flexibility to generating messages in the server.

MOXy uses SAX as its XML parsing method, which is one of the most efficient ways to parse XML-documents in Java. Being reliable, efficient and flexible JAX-B fits perfectly into server-side application as a message formatter. It also works perfectly well with other libraries such as EclipseLink JPA as the same classes can be used for message unmarshalling and database operations. (EclipseLink MOXy Documentation, 2013).

In Figure 20. there is an example of configuring MOXy into Maven's project object model –file (pom.xml). As shown in Figure 23. MOXy uses JAX-B's annotations to produce JSON/XML. **@XmlRootElement** –annotation defines the root element of the generated document. **@XmlElement** –annotation maps another object into the produced document. The object may be a POJO (Plain Old Java Object) annotated with JAX-B annotations or any basic data type.

```
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-moxy</artifactId>
    <version>2.3.1</version>
</dependency>
```

Figure 20. Adding MOXy dependency to the project.

4.3 Jersey

Jersey is the reference implementation of the Oracle's JAX-RS specification. JAX-RS is the Java API specification for creating RESTful Web Services. The purpose of the RESTful Web Service is to expose objects in the database for a client-side application. This can be done with Jersey, by exposing URL-paths that can be consumed by the AngularJS client.

The content is retrieved from the API by sending HTTP-request to an exposed URL-path. HTTP-request must be defined with HTTP request methods. For example sending a GET-request could retrieve all data from the context path or only one object by id.

As shown in Figure 21. the methods contain the business logic of the application. In this scenario the method is invocating **GenericDaoImpl**-class to query the entity manager for data. The **@Produces**-annotation specifies what format the data is returned in. There can be multiple formats configured and then the client can decide what is returned by using HTTP headers. **@Path**-annotation specifies the context path the resource can be found

from. The data is wrapped into **Response**-object, which adds HTTP-headers, a status code and a link to the object into the server's response. (Jersey Representations and Responses.)

```
@GET
@Path("applications/{id: \\d+}")
@Produces(MediaType.APPLICATION_JSON)
public Response getApplication(@PathParam("id") int id) {

    EntityManager entityManager = factory.createEntityManager();

    GenericDaoImpl<ClientApplication> impl =
    new GenericDaoImpl<ClientApplication>(entityManager,ClientApplication.class);

    ClientApplication application = impl.findOne(id);

    UriBuilder uri = uriInfo.getAbsolutePathBuilder();

    return Response.ok()
                   .link(uri.build(application), LINK_PARENT)
                   .entity(application)
                   .build();
}

@GET
@Produces(MediaType.APPLICATION_JSON)
@Path("applications")
public Response getAllApplications() {

    EntityManager entityManager = factory.createEntityManager();

    GenericDaoImpl<ClientApplication> impl =
    new GenericDaoImpl<ClientApplication>(entityManager, ClientApplication.class);

    List<ClientApplication> clientApplications = impl.getAll();

    UriBuilder uri = uriInfo.getAbsolutePathBuilder();

    return Response.ok()
                   .link(uri.build(clientApplications), LINK_PARENT)
                   .entity(clientApplications)
                   .build();
}
```

Figure 21. Example of exposed GET-paths in Jersey-class.

4.4 Accessing Database

The HTTP-verbs trigger CRUD-actions in the **Data Access Layer** of the RESTful service. The data access layer simplifies the access to the database by having a layer of application logic abstracting raw SQL-queries (Bill, B. 2010, 19-21). Each of the HTTP

verbs has its own meaning, for example sending a HTTP-request with POST method invocates *insert()* in the GenericDaoImpl. Data access layer can also abstract other functionality like *findByName* or simply *register*.

Data Access Object (DAO) is an object handling the database query logic. DAO is an implementation of the data access layer. It uses JPA objects mapped to the relational database as a container for the data. This is called **Object-Relational Mapping** (ORM). It protects the database from SQL-injections, vulnerabilities and allows the developer to handle database contents in the code.

In Figure 22. is shown an interface which all data access objects in the project have implemented. There is one generic implementation of the interface, but if more complex functionality is needed in the interface, it must be overwritten. Example usage of the implemented class is found in Figure 21.

```
public interface GenericDao<E> {
    public E insert(E e);
    public E update(long id, E e);
    public void delete(long id);
    public List<E> getAll();
    public E findOne(long id);
}
```

Figure 22. Interface specifying CRUD-functions.

4.4.1 EclipseLink JPA

EclipseLink JPA is the reference implementation of JPA 2.0 specification. JPA is used to create link between the Java application and the database, creating a **object-relational mapping**. It also supports file and NoSQL databases, but in this project PostgreSQL is used as a relational database. (EclipseLink JPA Wiki, 2015.)

The benefit of using JPA is the ability to abstract database from the application and expose the data as Java objects. The queries can be done with SQL, but the preferred way to use JPA is to either use Java Persistence Query Language (JPQL) or prepared statements. In prepared statements the query is predefined, and the query is executed when the prepared statement is called (EclipseLink JPA Wiki, 2015).

In Figure 23. is shown a configured JPA object. The @**Entity**-annotation specifies that the object is an entity. The @**Table**-annotation specifies the queried table. The @**PrimaryKey**-annotation configures primary key handling of the database. In this case the primary key is generated from sequence, which is an incrementing value in the database. Database columns are specified with the @**Column**-annotation. All existing columns do not have to be annotated if they are not used within the application, which makes JPA very flexible to use. JPA uses accessors and mutators to access the class attributes and does not work without them. In this example there are also junction tables annotated with @**OneToMany** and @**OneToOne**. The junctions are references to other tables. The application-entity is attached to a service entity and an access entity.

```
@Entity
@Table(name = "persistence_interface.client_application")
@PrimaryKey(validation=IdValidation.ZERO)
@XmlRootElement
public class ClientApplication implements Serializable {

    @Id
    @GeneratedValue(generator="clientApplicationSeq")
    @SequenceGenerator(
    name="clientApplicationSeq",
    sequenceName="persistence_interface.client_application_id_seq",
    allocationSize=1)
    private long id;

    @Column(name = "name")
    private String name;

    @Column(name = "version")
    private int version;

    @Column(name = "download_url")
    private String url;

    @OneToMany(mappedBy="application")
    private List<ServiceAccess> access;

    @OneToOne(mappedBy="application",
              cascade={CascadeType.MERGE,   CascadeType.DETACH})
    private InternalService externalService;

    @XmlElement
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    // More setters/getters
}
```

Figure 23. Example of a JPA Object annotated with JAX-B & JPA annotations.

It is also possible to use the same class to query data and produce XML/JSON with JAX-B. This reduces code overlap, since everything can be found within one class.

4.4.2 Usage of EntityManager

The core of the EclipseLink JPA is the **EntityManager**-class. The purpose of the class is to implement the ORM-layer into the application that holds the information of the objects in the database. The EntityManager can then be queried for data in the application layer.

EntityManager is instantiated from the class EntityManagerFactory. EntityManagerFactory should be instantiated only once in the lifetime of the application, because the operation is database specific. EntityManager is equivalent to a database connection and can be instantiated for every operation separately.

The greatest problem of using JPA is keeping the EntityManager-class up to date. When queries are made through JPA, only the query data is updated and the rest of the object-mapping stays unchanged. Sometimes when handling objects with references to other tables, the data in the EntityManager's data may differ from the actual data located in the database. This must be kept in mind when writing the application code and handling transactions.

The EntityManager can be kept in sync with the database by persisting the data every time an action in data access object is executed by using the flush() -method. Nonetheless the operation takes more resources, but ensures every change is stored in the database.

In the Figure 24. is shown a method for updating data. Reference of the object that is going to be updated is queried from the entity manager. The new object is merged into the reference of existing object. The changes are persisted to the database. The updated object is returned so it can be sent back to the client as response. The action happens within a transaction, which blocks all other users from modifying the data at the same time.

```
@Override
public E update(long id, E e) {

    // Begin transaction
    entityManager.getTransaction().begin();

    // Get reference of the object
    E referenceToExistingObject = entityManager.getReference(type, id);

    // Merge incoming object to the existing object
    referenceToExistingObject = entityManager.merge(e);

    // Persist the changes to the database
    entityManager.flush();

    // Commit transaction
    entityManager.getTransaction().commit();

    // Close the entity manager
    entityManager.close();

    return referenceToExistingObject;
}
```

Figure 24. Example use of the EntityManager.

4.4.3 Connection Pooling

Connection pooling is a term used for maintaining a cache of database connections, which can be reused when a database connection is needed. It enhances the performance of database queries as opening and maintaining a connection for each request is very resource and time consuming operation. If all connections are being used, a new connection instance is made and added to the pool. (Tomcat JDBC Pool, 2015.)

In a nutshell, it decreases the time to get the query done and reduces the amount of computing power needed to perform an operation. In a web application, connection pooling increases performance slightly as every request to the RESTful service needs individual database connection.

The connection pool is configured in Tomcat's context configuration file as shown in the Figure 25. The pool is given a certain size and it is configured to release abandoned connections. The connections are validated before using, just to ensure they work correctly.

Connection pool is taken to use in the JPA's configuration file, shown in the Figure 26. The application needs to know which transaction API and JPA providers are used as well as which JPA objects are available.

```
<Resource
     name="jdbc/postgres" <!-- resource name -->
     auth="Container"
     type="javax.sql.DataSource"
     factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
     initialSize="34" <!-- initial size of the pool -->
     maxActive="377"
     maxIdle="233"
     minIdle="89"
     timeBetweenEvictionRunsMillis="34000"
     minEvictableIdleTimeMillis="55000"
     validationQuery="SELECT 1"
     validationInterval="34000"
     testOnBorrow="true"  <!--  Validates the connection when it's retrieved  -->
     removeAbandoned="true"
     removeAbandonedTimeout="55"
     username="yyy"
     password="xxx"
     driverClassName="org.postgresql.Driver"
     url="jdbc:postgresql://127.0.0.1:5432/carein_server"
/>
```

Figure 25. Example of connection pool configuration in Tomcat's context.xml.

```
<persistence-unit name="response" transaction-type="JTA">
  <jta-data-source>java:/comp/env/jdbc/postgres</jta-data-source>
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>fi.mylab.server.system.model.label.Label</class>
  <!-- List of JPA objects in the project -->
  ….
</persistence-unit>
```

Figure 26. Example of connection pool configuration in JPA's persistence.xml

4.5 Logging

In case of an error, logging is an important feature in the server software that can let the server administrator know what caused the problem.

Disk space is an important thing to keep in mind as the log files might grow in size over the time and fill the server's hard drive. If the disk space suddenly runs out, the server crashes. A good solution is to set a maximum size for the log files and append the files after the limit has been reached. Writing logs to a different disk partition is also a good practice.

In healthcare systems, saving patient data is illegal without a permission from government, so obviously any request containing patient data must be carefully examined before logging any information. Timestamps, device identifiers and any exceptions that might have been resolved during the request may be logged without special permissions.

4.5.1 Apache LOG4J2

LOG4J2 is a logging framework for Java applications. It provides a good performance, support for multiple APIs, automatic configuration reloading, filtering the log events, support for plugins and property support (Apache LOG4J2 User's Guide, 2015).

By using a logging framework, it is easy to keep log files consistent and clean. The framework also takes care of performance and disk space issues. The ability to configure logging at class and event level ensures that only the important things are logged at the given time. The logger can be configured to debug-mode in the run time to display more detailed logs. It is also possible to log different things into different files or a database. (Apache LOG4J2 User's Guide, 2015.)

In this project, info and debug-logs are written into separate files. Logging is turned off as default. It is used mostly for development purposes as it slightly reduces the performance of the application.

4.6 Testing

Testing a server-side application is sometimes difficult, because the application needs the server infrastructure around it to work properly. Running tests in the project build phase is impossible, if the application does not work without a database and a web container. It is possible to mock them, but mocking means that the software will not be tested in production-level environment.

This problem was solved by creating a tester application, which is executed against a deployed web application. This way, the application can be tested in the environment it is supposed to be in without any mock objects, and the real functionality is exposed to the tester.

Benefits of a well implemented test suite are huge. The tests define how the code must work, so there will be less bugs. The code can be tested at any time because the testing is automatized. Refactoring code becomes easy, because the functionality of the software can always be tested with the test suite.

Writing a good test suite requires lots of planning and precise use cases that can be turned into writable tests. The difference between a use case and a test is quite simple. Creating a use case starts from the top-level of abstraction and writing a test starts from the bottom-level abstraction. Abstraction in this context refers to the viewpoint how the software is being looked at. It could be looked from the end users view (top-level abstraction) or the technical point of view (bottom-level abstraction).

When writing a test, there must be a defined use case. The use case is split into smaller aggregates until the needed level of abstraction is achieved. Writing tests can start when the aggregates are simple enough to be expressed in the code.

For example, there is a use case for installing a device, which requires many actions in the REST interface. A use case has to be split into single operations which can be implemented in the code, like inserting and updating a device object. After the functionality to insert and update have been implemented, the test for installing a device can be written completely. The created case could be used in higher level abstraction then.

The purpose of the tester application is to make testing simple and efficient. Every time code is changed, the tests can be run without an effort. As shown in Figure 27. the methods are quite thin and test they the functionality of the REST interface. Everyone should be able to contribute to testing and be able to write tests.

Writing and contributing in testing was made easy by writing a framework, which contains built-in tools like object factories. The utility class contains CRUD-actions for every exposed REST-path. The core of the test framework is Jersey Client API, which is used to create HTTP-requests to the server. The assertion and running is done with JUnit framework.

```
@Test
public void addDeviceTest() {

    ClientDevice request = DeviceFactory.getDeviceInstance();

    ClientDevice response = utility.addDevice(request);

    assertEquals(request.getAndroidId(), response.getAndroidId());
    assertEquals(request.getName(), response.getName());
    assertEquals(request.getCertificateName(), response.getCertificateName());
    assertEquals(request.getCertificateExpiration(),
                 response.getCertificateExpiration());
    assertEquals(request.getInstallDate(), response.getInstallDate());
}
```

Figure 27. Example of test method in the tester framework.

Testing can also be done with any HTTP-client capable of sending customized requests to the server.

5 PROJECT TOOLS

Repeating same steps or using inefficient tools might slow down the project workflow. Good project tools improve the ability to work and help with documenting the project and communicating with the team.

5.1 Maven

Maven is a tool for project management and comprehension. It is mature tool and has support for almost any kind of Java project. Maven can generate documentation and reports. It also manages dependencies, builds the project and runs tests on the project from central piece of information, called project object model (Maven documentation, 2015).

Maven was chosen as the build tool, because it has plugins to work with the Tomcat. With the Tomcat plugins it is possible to deploy an application to the Tomcat container, which creates a lean workflow for building and testing the project.

5.1.1 Dependency management

The purpose of dependencies is to make sure every resource used in project is present when building or running the application. Otherwise the application would crash, because the called code does not exists in the application's context. Dependencies can either be from frameworks or independent libraries. Dependencies are important because they eliminate the need of checking every resource between builds.

Maven has global repository containing tons of different dependencies. A developer can also create local repository for private libraries, which can be then managed across different projects. This encourages to write reusable code, which can be used in different projects. Dependencies are managed in POM-file of the project.

5.1.2 Deploying AngularJS application

One of the greatest concerns of the project was where to place the AngularJS code. Tomcat natively supports JavaServer Pages technology, but that was not an option, since AngularJS was chosen as the user interface framework.

The solution to this problem was to place AngularJS code to the project's /webapps/-directory, so it is found inside the war-package after building the project. When deployed, the AngularJS application can be found in the web application's root context path. This way the application stays in one package and is easier to manage. Other solution would have been to deploy the AngularJS application to another context path in the Tomcat, but the installation would have not been as simple.

5.1.3 Installing the application

Compiling Java web application differs from normal Java application. The file format used is WAR instead of JAR. Web application must be deployed to the Tomcat container before it can be ran. The application can be deployed either manually from Tomcat Manager or by copying it into the Tomcat's /webapps/ -directory. Deploying can also be automated with Maven plugin, shown in Figure 28, which deploys the web application automatically after the build is done. After the deployment the web application can be accessed with any web browser or an HTTP-client.

```
<plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.1</version>
    <configuration>
        <url>http://localhost:8080/manager/text</url>
        <server>localhost</server>
        <path>/path</path>
     </configuration>
</plugin>
```

Figure 28. Example of configuring Tomcat 7 Maven-plugin.

5.2 Mercurial

Version control is very important in enterprise-level software development. The code must be safe and recoverable in case of emergency. Also the ability to view the past revisions of the code can be useful. Mercurial is used from the command line, because commands can be given much more precisely and it is a plugin-free environment.

5.2.1 Version control workflow

```
// Make sure we are in default branch
C:\Development\projects\project_A>hg update default

// Check updates from the server
hg inc

// Pull incoming changes to the local master branch
hg pull

// Update the working files
hg update

// Create new branch for JIRA issue
hg branch JIRA01

// Initial commit on branch
hg commit -m "JIRA01 branch created"
```

Figure 29. Begin working with a JIRA issue.

Implementing an issue starts with checking version control and creating new branch for the issue shown in Figure 29.

```
// Changes in the code


// Check the current status of the files
hg status

// Adds new files and removes removed files in single command.
hg addremove

// Can be used instead of addremove, to add a single file
hg add

// Commit changes
hg commit -m "JIRA01 [MESSAGE]"
```

Figure 30. Working with the project.

In Figure 30. is shown working with branch. Keep track of the changed files and commit the changes, when ready.

```
// Move to default branch
hg update default

// Check if the default branch has been updated while working
hg inc


If there are any changes:
// Pull the changes into local master
hg pull

// Update working files
hg update

Merge the default branch to the issue's branch:
// Move to the issue branch
hg update JIRA01

// Merge the default branch
hg merge default

/**** Resolve conflicts if there are any ****/

// Commit merging the branches
hg commit -m "default merged to JIRA01"


Merge development branch to the default branch:
// Update back to the default branch
hg update default

// Merge changes
hg merge JIRA01

/**** Resolve conflicts if there are any ****/
hg resolve --list

hg resolve -m [conflicting_file]

// Commit merge
hg commit -m "JIRA01 merged to default"

// Push new files to the remote repository
hg push --new-branch
```

Figure 31. Finishing an issue. Commit changes to the upstream.

Finishing an issue is bit more complicated because conflicts might occur when there are multiple developers working on the same application. Workflow shown in Figure 31. is designed to prevent conflicts by checking the version control status before pushing or

pulling anything. The updates are pulled from the repository and merged in the local default branch. In this stage the conflicts are resolved in the local branch, so they do not spread out to the Mercurial's upstream. After the changes from the default branch and from the local branch are successfully merged, the changes can be pushed to the upstream successfully.

DISCUSSION

The requirements set for the implemented component should be capable of controlling device access, be able to distribute updates and administrate device, print label definition and service registries. The software met the requirements and all features were successfully implemented.

The registry management component will be published as a part of the server implementation and installed to the customer in the autumn 2015. The new server software will replace the old one, but they are ran concurrently until the new software is completely verified. The registry management component is only a small part of whole server software, but I found it as a good subject for a thesis, because it was simple enough to expose from the rest of the server software.

The component helps other employees of the company in their work. The auto-update feature reduces the amount of visitations to the customers, because it removes the need for physically installing new versions of the sampling applications. Being able to check devices and services from registry improves the quality of life for everyone, because the text files containing the data are not needed anymore. Device access control creates extra layer of security for the service.

There are plans to improve the registry management component. In the next version the registry is expanded to include more peripherals and user data. More features are added to sort and control the data in the user interface. The component is intended for internal use only at the company, however there are plans to let customers have access to the user interface and control their own devices.

Most difficult part of the work was to learn the technology stack the company uses. The technology stack might need some reconsideration in the future. The mobile service is capable of working with components implemented with different technologies, so that leaves room for testing other solutions.

REFERENCES

AngularJS Dependency Injection, 2015. Read 07.07.2015
https://docs.angularjs.org/guide/di

AngularJS Providers, 2014. Read 07.07.2015
https://code.angularjs.org/1.2.28/docs/guide/providers

AngularJS Unit Testing, 2015. Read 07.07.2015
https://docs.angularjs.org/guide/unit-testing

Apache LOG4J2 User's Guide, 2015. Read 08.07.2015
http://logging.apache.org/log4j/2.x/log4j-users-guide.pdf

Apache Tomcat Home, 2015. Read 07.08.2015
http://tomcat.apache.org/

Bill, B. 2010. RESTful Java with JAX-RS. O'Reilly Media, Inc. Read 01.07.2014

Bootstrap Customize, 2015. Read 07.08.2015
http://getbootstrap.com/customize/

Bootstrap CSS, 2015. Read 21.08.2015
http://getbootstrap.com/css/

EclipseLink MOXy Documentation, 2013. Read 12.05.2015
https://wiki.eclipse.org/EclipseLink/UserGuide/MOXy

EclipseLink JPA Wiki, 2015. Read 07.05.2015
https://wiki.eclipse.org/EclipseLink/Examples/JPA

Jersey Representations and Responses. Read 10.07.2015
https://jersey.java.net/documentation/latest/representations.html#d0e6616

Maven Documentation, 2015. Read 10.09.2015
https://maven.apache.org/guides/index.html

PostgreSQL Advantages. 2015. Read 10.10.2015
http://www.postgresql.org/about/advantages/

PostgreSQL 9.4.4 Documentation. 2015. Read 08.10.2015
http://www.postgresql.org/docs/9.4/interactive/index.html

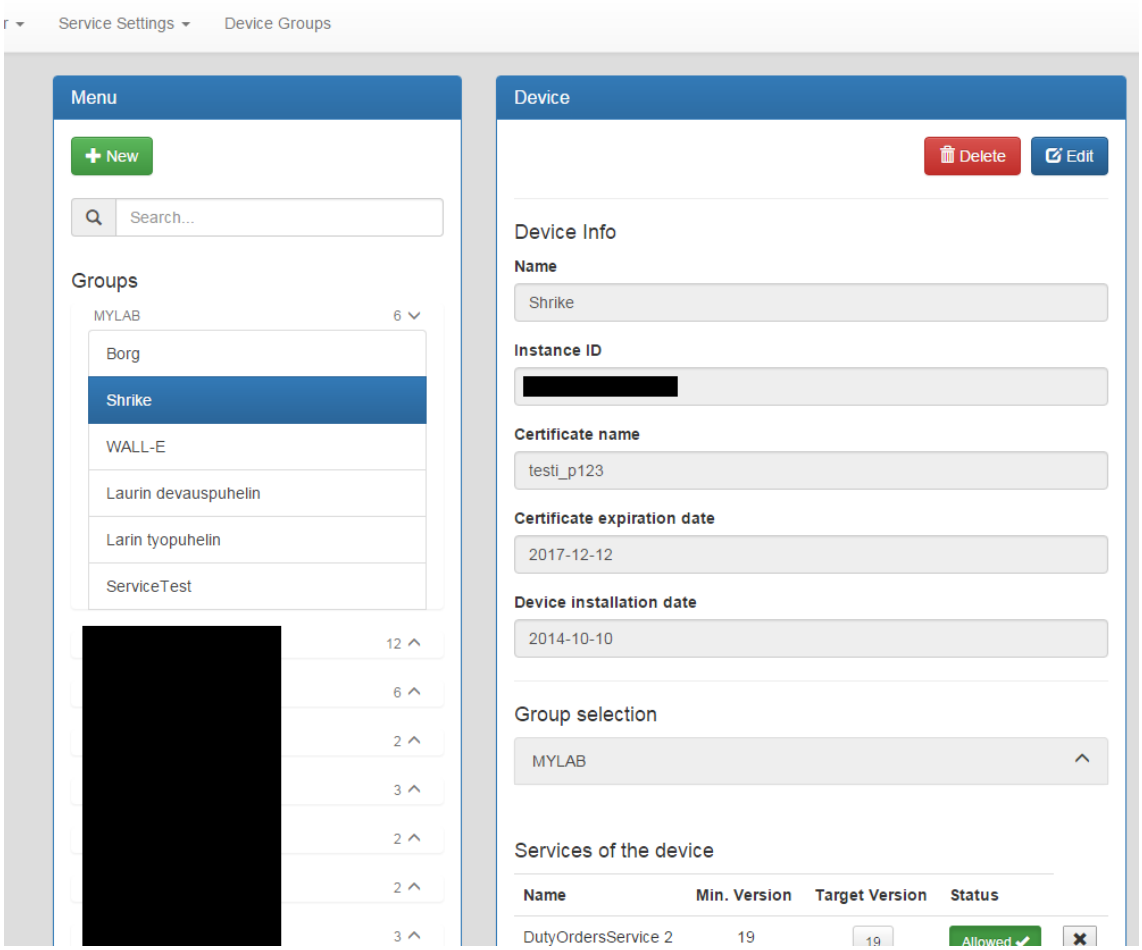
R. Elmasri. S. Navathe. 2004. Fundamentals of Database Systems. Pearson Education, Inc. Read 22.08.2015

Simple Object Access Protocol (SOAP) 1.1. 2000. Read 22.07.2015
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

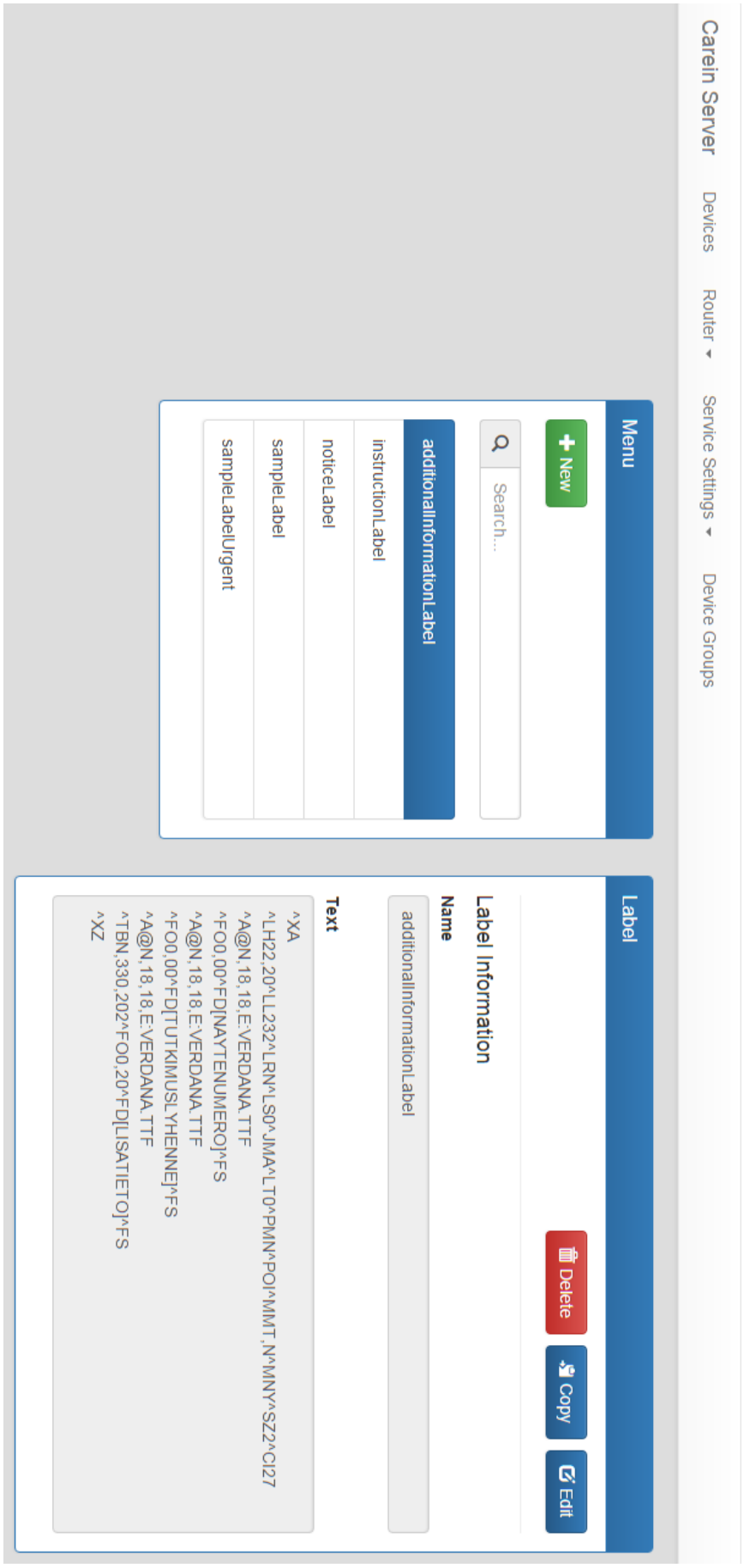
Tomcat JDBC Pool, 2015. Read 09.09.2014
https://tomcat.apache.org/tomcat-7.0-doc/jdbc-pool.html

APPENDICES

Appendix 1. Registry management desktop layout in English



Appendix 2. Registry management layout - label view



Appendix 3. Access Monitoring View