Examensarbete

RESTful Cloud Server

av

Linus Karlsson

LIU-IDA/LITH-EX-G—13/018--SE

2013-06-10
RESTful Cloud Server

Kandidatuppsats, 16 poäng, skriven av
Linus Karlsson
2013-06-10

ISRN LIU-IDA/LITH-EX-G—13/018--SE

RESTful Cloud Server
ABSTRACT
This report is about the development of a cloud service server that will be used to store user information. The cloud's purpose is to store information about the user and the devices he or she uses. The cloud has web services that enable users to retrieve and store the data on the server. It is also responsible for user administration and security of the data. A web client is also developed in parallel with the cloud. The web client uses the cloud's web services to show and modify data for the user.

INTRODUCTION
The RESTful Cloud Server is a web service server. Different clients will be able to connect to the RESTful Cloud Server, like an Android application that is used by a user that will upload the data to the RESTful Cloud Server. The user should be able to upload any information from a device to the application and then from the application to the RESTful Cloud Server.

By doing this the user will get information about the usage of the devices. It could also be used by the device manufacturers to retrieve information in a simpler way.

On the web client for the RESTful Cloud Server the users will be able to view their information about the usage of the devices, like heart rate, if that is uploaded by the user.

The administrators will also be able to view the user’s usage of the devices, information and be able to do user administration.

This project will focus on implementing a cloud service for saving and viewing data that the users upload using a client. The project will additionally secure the web services from unwanted actions. The web client will be implemented in parallel to the cloud server. The report will also clarify required changes to the development process when implementing for a service that could be used to store data from a medical device, the standards it brings and what is important to take into consideration during development of the product.

The question formulation is the following:
How can a REST-based cloud service server be implemented to be used in correlation with medical devices?

The report will start with theory where the necessary concepts and terminology for the rest of the report will be explained. After the theory chapter comes the method which contains the tools and work processes of the project. This will be followed by the results that have been implemented during the project. These results will be discussed and determined in the chapters called discussion and conclusion.

THEORY
This part will explain why we choose the different libraries and applications we have used in the creation of the RESTful Cloud Server (RCS). There will also be some clarification about the definition of a cloud and how we chose the licenses for the project. It will be explained in this order:

- Cloud
- Licenses
- Web client
- Web Service Server
- Development tools and processes

Cloud
For a cloud server it is important to be scaleability. This will help on peak times, since then the cloud can be upscaled and then when the peak time is over it can be downscaled again. For this to be effective, the cloud needs to have a good structure of its layers, each layer should have a well defined purpose. In this way the cloud could also use other services to get information. [1][2]

Licenses
When deciding what libraries we should use in the making of the RCS it was important to look on the license for using the library. When looking at the license it should specify that you are permitted to redistribute the source modified or not as you like. This will insure that even if the developer’s stops developing on their work, you can still continue updating it yourself. Most of the libraries we have chosen to work with are under two types of licenses; they are well known and are commonly used by open source projects.

MIT License
MIT License [3] allows us to use the software free of charge and we may modify it as we like and still distribute it with the client.

Apache License
Apache License [4] allows us to use the software free of charge and we may modify it as we like but if we modify any code in the sources that has the Apache License we must add a notice that we have modified the source.
Web client
When making the web client we will use HTML, JavaScript and CSS. We will use different libraries to help us with this:

jQuery
jQuery [5] is a JavaScript library, it helps with writing JavaScript’s for Ajax, which this project will use a lot since we are getting the data that is shown on the web client from the RCS. It also makes other web programming related implementations easier, and Knockout also uses jQuery.
jQuery is using the MIT License.

Knockout
Knockout [6] is a JavaScript library which focuses on having dependency tracking and bindings to the User Interface (UI), so that if a variable is updated it will update the UI as well. This will be used in RCS’s webpage so the presentation of data will be simpler and more interactive.
Knockout is using the MIT License.

Twitter Bootstrap
For the front-end framework it was decided to use Twitter Bootstrap [7]. This framework will make it easier for us to get a working front-end fast with working scaling and other good UI functions.
We will use the Twitter Bootstrap CSS and functions as much as possible to make it easier for a designer to come in later and make changes to the UI. The goal of this report is to make the RCS, so it will focus on the underlying functions and not the cosmetics of the webpage.
Twitter Bootstrap is using the Apache License.

Web Service Server
This will be written in C# using Microsoft Visual Studios 2012 [8] as our development environment.

ServiceStack
ServiceStack [9] is a REST Web Service [10] framework that will be used in the back-end of the RCS to make the web services that are needed. It has a lot of built in functions that will help in the implementation of the RCS, like authentication, authorization and validation, just to name a few.
ServiceStack is using its own license [11], it is similar to the MIT license but with some additional conditions, we are still able to modify it as we like. But when we are distributing it we are not allowed to use the name ServiceStack nor its contributors to endorse or promote the product.

ServiceStack OrmLite
The OrmLite [12] library is used to make object-relational mapping (ORM) [13] in a lightweight way. This will be used in the RCS to easily make database models that can be used in both SQLite [14] and SQL server [15].
Since the RCS is going to support both of these types of databases this will save some time comparing to doing it manually. It will also offer us a better view over how the tables will look like and other customizations that could be useful.
ServiceStack OrmLite is using the same license as ServiceStack.

Log4net
Log4net [16] is a library for .NET projects that helps with logging for the RCS. With Log4net different severity logs can be made:

Debug
• Info
• Warn
• Error
• Fatal
When the RCS is later up and running we can specify that the logging should only be made if Warn or higher. With this the administrator can later keep track of the RCS and be alerted if there are any problems with the RCS at run time.
Log4net is using the Apache License.

Development tools and processes

Mercurial
Mercurial [17] is a distributed source control management tool. We use it for the project, so each programmer can have a local copy of the project and then when a task is done, you merge your code with the rest of the project. Mercurial will show if there is any difference between the project that is located on the server and the one being pushed so that it can be corrected.
It also has revision control. So if anything goes wrong you can always go back to an earlier state and start again from there.

FogBugz
For task handling we will use FogBugz [18] for the whole project, so the other project parts will also be seen in this application.
In FogBugz we can specify tasks and subcases to the larger tasks. Each task will be linked to the projects wiki so it later can be traced for documentation and validation. The tasks and sub-cases will also be linked to the Mercurial commit messages, and with the help of release notes it will create traceability throughout the whole project.
FogBugz also gives us a good overlook over all the tasks in the project. So you can see the progress on the other parts
of the project.

**Kanban**
Kanban [19] is a method for software development, and will be used in the process in making the RCS. The Kanban board will be divided into six columns: Proposed, Approved, In Progress, Ready for testing, ready to ship, Shipped. Each task will start out in Proposed and finally end up in Shipped.

**Jenkins**
To help with the testing of the RCS we will use Jenkins [20]. Jenkins is an application that can run the tests we have made for the project and also sets up a test server like it would be done for the final product.

Each time that a task is done, and it gets pushed to the server, Jenkins will run the tests of the RCS and build it to a test server. By doing this we will see if the code is working as it should after the changes of the task. And hopefully this will mean that bugs will be seen sooner than later.

**IEC 62304**
IEC 62304 [21] is a standard for developing and designing medical devices. It requires that the company that is developing the software for the medical device has traceability throughout the development.

This means that everything should be traceable throughout the project, so commits made should be linked to a case and a case should be linked to a part of the documentation.

This standard is supported in both United States as well as in Europe Union.

**METHOD**
In this section we will talk about how the project was implemented and what programs were used. It will be explained in this order:

- Implementation
- Work Iterations
- Start

**Implementation**
When implementing, we used Visual Studio 2012 since the server is implemented in the programming language C# and Visual Studio also offers help when implementing for the web client, mainly syntax correction of the JavaScript’s.

For viewing the SQL Server database we used SQL Server Management Studio [22], this is also used for creating the SQL Server database.

**Work Iterations**
When a task is made, we sit down and discuss what this task is going to fulfill. We create a functional requirement for the task which will specify what is expected of the RCS after the task is done. When that is done we divide it up to smaller sub-cases. This is done so it will be easier to just look on them and to estimate the time it could take to make them. After this it is time to start working on the task, we move it to the column “In progress” in Kanban and then we start working on one of the sub-cases.

The first thing we do is to make the test cases for the new functions we are about to make for the task or if we are about to update an existing function we update the test cases that are already there. This helps us with the implementation that we start right after we are done with the test cases since we then have tests that will see if our new code is done in a way that they pass the test. When the task is done we look over the wiki to see if any changes that was made during the task effects anything on the wiki and updates it if needed.

We then push up the new RCS to the repository and make a commit message that contains which case it is regarding and any changes that need to be logged. After that the RCS will be built on Jenkins, if all the tests passes we mark the task as resolved and make a release note for the task. And the task will be moved to “Ready for testing” on the Kanban board. The task will then be reviewed and if something is missing, or something new has come up that is needed it will be moved back to “In Progress”, otherwise it will be moved to “Ready to Ship” on the Kanban board. In this way we can get a high quality product in the end and it also makes sure that we have thought over each part of the RCS but still have room for changes if needed.

![Figure 1: A diagram over the work iterations.](image)

**The Start**
When we started on the project there was already a skeleton
made of the RCS and the web client. So what we did first was to get the project to work, and then we had a look at the code that was already there.

After this we started to add some of our own code, so we decided to make the function to create, update, read and delete users on the server side. We didn’t do any documentation at this time since it was only for trying out how to do it. When the task was finished, that we had made for ourselves, we cloned the RCS to a new directory so we could do the same task, but this time in the way of the work iteration plan. And since we already had done the task before, it went faster than the first time and some of the code could be reused.

RESULT

In this section you will be able to read about the result of the Bionic Telemetry Cloud in the following order:

- Structure of the RCS
- Work iterations
- Web Services
- Security
- Test Cases
- Paging
- Database
- Web Client

Structure of the RCS

For the RCS it is important to have the back-end functionality divided up in classes and files. So each class is in the right file that does the same kind of work as the other classes in the same file. This is so that later if we want to change a part of the program we can do so by just switching the file. The files are also divided into different folders depending on the type of class they contain. The project structure has these parts; the relation can be seen in Figure 2:

UI: The user interface of the web client. The job for this part is to make the data that is coming from the RCS to be displayed in a nice way, so it is easy to understand. Here we use Twitter bootstrap to make the web clients interface.

JavaScript: We are using jQuery functions like AJAX to get, send or put calls to the RCS. We also use Knockout to display the results of the calls to the UI.

Web services: Receives REST calls and redirects the call to the service that is linked to the URL. We are here using ServiceStack to link URLs to services.

Services: Specifies what data it expects to receive in a call, it redirects the incoming variable to the business logic layer.

Business Logic: Takes care of the data manipulation and uses the database handler to get and set data. Here we find the extensions to the services for the RCS.

Database handler: Here all the SQL-queries are made; the functions take variables and use them to create the queries to get the data that is needed. It will make the SQL-queries to the OrmLite and will return the results to the Business Logic.

OrmLite: ServiceStack OrmLite handles the database. It creates and drops the tables on the database, runs the SQL-queries on the database and also provides with some basic function for the database that can be used to save and update data.

Database: Stores the data that comes into the RCS. The database can be either SQLite or SQL Server. The database is accessed through the OrmLite layer.

Figure 2: Diagram of the structure.

When making a new function it is then important to follow the structure and put it in the right folder, in the right file. If something is in the wrong file and folder it will later be harder to find and can easily get lost if a file or a folder of the RCS gets switched.

Work iterations

Since the work iteration system was very important for the project, it was followed as much as possible during the project. This resulted in an always up-to-date wiki over the project which proved useful when others in the company
wanted to get an overview over the project. Also the test cases helped a lot when developing, since when changes are made there is almost always something that happens that is not expected and with the help from them it was easier to catch these problems.

Web Services
When a REST call comes in to the RCS it will be taken care of by ServiceStack. ServiceStack redirects the call to the function that has been linked to the URL route of the call. For example:

```
Routes
    .Add<UsersGet>("/users/list");
```

The “UsersGet” specify what class the route should be linked to. So when the RCS /servicestack/users/list is called it will be directed to the class UsersGet. In the UsersGet class it is specified what variables the web service expects to receive in the call. The variables are then used to make the next function call that is to either a OrmLite function or a extention function to the web service. The extention contains all the data manipulation and calls to the database functions. And when the extention is done it returns the result to the service which in its turn returns it to the client.

![Figure 3: Diagram over the flow of web services.](image)

Security

**Authentication**
A safety issue that has to be dealt with was that nobody should be able to use the web services without being authenticated. If a user would be able to use web services without this it could be used in a way it was not meant to be used. Therefore we started with implementing authentication with the users e-mail and password. The client would use a service on the RCS and send with its e-mail and password. If the e-mail and password are correct the client will receive a cookie from ServiceStack that will be used with each web service request. ServiceStack also allows us to make custom changes to the session, so we can add more information of the user, like the Role the user has. With a use of annotation in the web services we can specify for ServiceStack which services should only be allowed to use when the user is authenticated.

When the user is authenticated at the RCS, the client calls a function on the web server. The login function on the server checks if the user is authenticated on the RCS and stores a HTTP Context session with the username, authentication status and the role of the user. This information will be used on the server when a user wants to open a web page that is restricted, then the server checks what role the user has and if the user is authenticated. If the user is not authenticated it will be redirected to the home page of the web client.

The web client also saves information about the user it receives from the authentication service. We are using the SessionStorage [23] for saving the information, with this we can save any information that we want on the client and use it when needed, so we don’t need to save any cookie on the client for information that we only want to use on the client side. This information is used to display the user’s e-mail at the header, this so that the user can see that it is logged in as for example the user “user@user.se”.

The flow of authentication, as seen in figure 4:

1. Web client makes a REST call with e-mail and password.
2. The RCS responds with either a cookie if the authentication was successful and continues with step 3. Otherwise RCS responds with an exception that the authentication failed.
3. The web client sends the e-mail and password to the web server.
4. The web server uses the e-mail and password to make an authentication check on the RCS.
5. RCS responds with the user Id and the role of the user.
6. The web server responds to the web client that the authentication was successful.
Password
The password for the user is saved on the server with hashing with the method RFC2898 [26]. C# had a built in function for this called "Rfc2898DeriveBytes", so before the password is stored on the server it is hashed with a salt. When a user tries to log in on the server it will take the password and use the same function and salt. Then it will use the hashed password and see if the password matches the stored password on the server. By doing this we won’t give away anyone’s password if the database is accessed by an outside source.

Test Cases
Test cases were made for the functions written on the RCS. To make the test that each part of the system worked we made tests that tested the functions directly and tests that used the REST calls to test the functions.

When the REST calls were made we had to create a user using the functions directly and then authenticate the user. This was because when we added authentication to use the services of the cloud we could no longer create the user using services without being logged in as a user.

Paging
For both the user list and the feed list we wanted to have paging, the reason for this was that when the RCS gets a lot of users or a user gets a lot of feeds it will be a lot of data to load at one time to an array each user wants to see the feeds or an administrator wants to see a list of the users in the system. So we decided to make server side paging for getting a list of data. When the service is called the client needs to specify the page size and what page number it wants, then the service uses this to take out just that page from the server. We made it so the client calls for one more object then the page size, and then on the client it will look if the list is full, if it is full the web client will know that there are more objects that can be fetched.

Database
One of the requirements regarding the RCS was that it would support both SQLite and SQL Server. Therefore in the database class, were all the queries for the database is made, it had to check what kind of database the RCS was using.

An example of this is the function to get a paged list from a table. In SQLite the Limit function was used and since Limit does not exist in SQL Server another clause named "row_number()" had to be used.

SQLite:
"SELECT * FROM [{Table}] WHERE IsDeleted = 0 {other criteria} ORDER BY {orderBy} {order} LIMIT {start}, {count}"

SQL Server:
"SELECT * FROM (SELECT * ,row_number() OVER (ORDER BY {orderBy} {order}) as RowNumber FROM [{table}] WHERE IsDeleted = 0 {other criteria}) c WHERE RowNumber BETWEEN {start + 1} AND {start + count}"

This is made to be as dynamic as possible so that different kinds of paging can be made. That is why “Other criteria” can be used to insert other criteria, like a user id to just get one specific users feeds.

Structure
The structure of the database can be seen in Figure 5. The User table contains information about the user; it can be a user, administrator, etc. A user can have many devices and feeds.

The table Device can be any type of device; it has an owner that is a user. The feed is a collection of streams, with some basic information about what the collection of streams are.

A stream is a collection of many points that are of the same type, it could be the steps per minute or the heart rate during a workout. The stream also contains what unit the points are, how many points there are in the stream and some other useful information.

A point is a point in time with a value for that moment, so it could be the heart rate for one minute or the current step count.
Soft Delete
We implemented Soft delete for all the database tables in RCS. The reason is because we want to be able to still access data from users even though they have been deleted. Another scenario can be that someone accidentally deletes some data and then wants to restore it again. Each model in the database will be given a “IsDeleted” Boolean column, it will be set to false when a new entry is created. When the entry is going to get deleted it will be set to true.

This means that with every SQL-query that is made to get non-deleted data, it needs to make a check if this is set to false; otherwise it should not take it with in the result.

Web client
In the web client it is possible to login as a user or an administrator. The administrator can do things like: viewing all the users, creating new users, modifying existing once and delete users.

The user can view its own feeds and modify its own account information.

When the user logs in to the client, it sends the password and username to the RCS. The RCS will then check if this user exists and if the password matches the username. When that is done it will respond with user id of the user so it can be used to get information about the user on the client. When the client gets the response and all went well it will make a call to the server side, with the username and password, the server will then check if the user is authenticated on the server and if it is that it will save user information in the session on the server. This information will be used to see if the user is allowed to access some of the pages on the server side. This is so that nobody should be able to manipulate the JavaScript on the client side to make it look like it has access rights to the page.

Administration functions
As an administrator on the RCS you are able to:

- Create new users
- Modify user
- Delete user(s)

List users: On this page the administrator gets a list of a page of the users and uses the buttons in the menu to navigate to the other pages of users. This is divided using the paging function on the server and on the web client.

Create User: On this page the administrator gets to fill out a form that covers all the data that is needed to create a user. To check that the information is in the right format before sending it to the server we use an extension library to Knockout called Knockout Validation [24]. With this we can specify which fields are required and we can also use it with regular expressions to specify how the data should look like. It also makes it possible to write a message to the user if a field is not as it is supposed to be.

Modify User: This works almost like the Create User page but when the modify user page is open it will load the user ’s information into the form and it will be displayed for the page user.

Delete user(s): With the help of a button or by using the checkboxes on the User List page, the administrator can delete users from the RCS. The administrator can use the checkboxes to remove more than one user at a time. When the administrator presses the delete button, a popup will appear asking if the administrator is really sure that it should go through with the deletion.

View feeds
When a user or administrator wants to view a feed, it will show some basic information about the feed. Some of the information is; when it was created and who owns the feed. When the feed is opened, the user will see a multigraph of some of the streams related to the feed. The user can choose to add or remove streams to this multigraph as the user wants. This is useful if the user wants to see correlations between two or more points from different streams. The user can also choose to open up separated streams; this will show a graph over just that stream.

To make the graphs we decided to use a library called Flot [25], this is used with jQuery to make it simpler to create graphs. For the multigraph we choose to use the built in function to have multiple Y-axis on the chart, and with the help of units from the stream we marked the Y-axis so it was clear which Y-axis is for which line.
**DISCUSSION**

Here we will discuss what we could have done differently, what options we had and why we chose as we did. The order of this section will be:

- Structure
- Security
- Test Cases
- Paging
- Database
- Web Client

**Structure**

We felt it was very important to follow the structure of the project, it helps with productivity since if you do it right you will know where to look for function and classes. And if any part of the RCS ever needs to be swapped it will be easy since no function should be in the wrong part of the structure. And they should never “jump” over a step on the structure since then if any part is swapped it might become a big failure in the code, since the new part might change how things are done completely.

If the project had been bigger it would have been wise to add more parts to the structure, to get each part to specialize on one thing only. But for the RCS we thought this was enough of parts.

**Security**

**Authentication**

Since the authentication is made in two steps by the client and it is checked by the server two times, it becomes more secure, since if we would just assume or save a session data on the client side to get to the pages on the web client someone could just change the data in the session and say that they are allowed and that the authentication has passed. But still if someone would have done this, they still could not have accessed the web services on the RCS since you need to be authenticated to use them.

A thing that should be implemented on the RCS is that when the e-mail and password are sent to the service, it should do so over HTTPS. This would make a “man-in-the-middle attack” [27] harder to accomplish since the call would be encrypted and the password would not be sent in clear text over HTTP as that would be an easier target for a man-in-the-middle attack.

**Password**

The password is hashed with RFC2898 and it is using the same salt for each user, another option to this would be to create a different salt for each user and then save it in the database and then use it again when a user tries to login. But if someone would access the database they would get the salt and the hashed password, as it is now they would need to get a hold of the code to get the salt.

Since we are only comparing the hashed passwords we never have to reverse the password into it first clear-text state.

**Test cases**

The test cases for the RCS could have been done in a more complex way, it might have been a good idea to have someone else to write some of the tests for the function so that more than one person thinks about what might be needed for the test of the function.

For the web client no tests were made for the JavaScript’s functions. This should have been done from the beginning, this could have saved some time since when you implement new function or change them you might break something you didn’t think of and then you have to do manual testing. Manual testing can take some time and the risk of missing to test something can be high so for this reason it can be very good to have a series of test that could be run each time a change was made or something was added.

**Paging**

The paging made it so we can load a small amount of data instead of the whole list of for example users. If we would have just loaded in the whole list of users and we would have larger user base, then it would take a lot of time and bandwidth for the user. This would also make it so the page would seem slow and not as responsive. So by specifying how many objects the list should contain and the page number, it was easy to calculate what index we should start on and stop on when we fetched the data from the database.

A thing that could have been done differently would have to have the RCS send back with the list if there was any object in the next page for the web client to get. But we thought it would be easier to have the function just return the list since you might want some kind of function that gets all the objects in the list. And to check on the web client if there is at least one more object on the next page would not slow it down or increase the bandwidth use by a high factor.
Database
Since the requirement for the RCS was that it had to support both SQLite and SQL Server it took a bit long time to make the SQL-queries since we had to make them for both of the different databases. The ServiceStack OrmLite supported different databases so that was no problem with creating the tables, but when it was time for testing you had to test both with the SQLite and the SQL Server, and the same on the web client since there might be some differences between these two.

So for the time invested into making the RCS able to use two different databases might have been worth it since you can install it on a server and use the kind of database that the server administrator likes the most.

Soft delete
This can be argued that it might take too much space after a while if there is a lot of data in the database that is marked as "IsDeleted".

A thing that could be implemented in a later stage is a function that could hard delete all data that is soft deleted after a specific amount of time. This would open up for more space and you would still have the safety that comes with having a soft delete. There could also be different policies depending on what type of data it is, like for a user data might be more worth saving for a longer period of time then saving the points in a feed.

Web client
UI
Since the main goal of this thesis were to implement the RCS it was not our main focus to work on the UI, but the foundation for a UI designer to make a new look is there since we used the classes from Twitter Bootstrap and did not add any new once ourselves. And because of this, a UI designer could choose to take a premade look that is made for Twitter Bootstrap and just swap it out for the new one.

So by using only the classes set by Twitter Bootstrap we have made it so time and money can be saved on the design if it ever will be changed. Also if Twitter Bootstrap is update it will follow the same pattern and nothing from our own design will be outdated.

Administration function
One administration function that is missing is that an administrator should be able to view the users feeds, this could be done with an web service that gets all the users in the RCS in a list, then the administrator could open up a user and see what feeds it has, and then in that way see the feeds of the user.

Another thing that could be useful to implement would be to be able as an administrator view a list of streams with one kind of device or another variable that the administrator would like to see. Since if someone is developing for one type of device that developer could get a list of information just regarding a specific device.

Paging
The paging on the web client works as it is now, but it might have been a nice feature to have a page number, so that the user of the list could navigate through it by using the page numbers. But for this to work it the RCS service for getting list would have to calculate and return how many pages there are in total.

This might be a thing to add in a later stage in the development of the RCS, but for our time period of work and the focus we choose this was not a priority.

Multigraph
For this function we could have made a normalization of the values of the different graphs so that they could use the same Y-axis. If we would have chosen this approach we would have to write out what scale each line has. And this could make it more complicated for the user, since the user would then have to calculate the line to the scale. Another thing that could have been done in this would be to have dots that automatically calculates the value if you hover over it with the mouse.

There was also the option to make the Y-axis in color code, so that the Y-axis was in the same color as the respective line in the graph. That would make the width of the Y-axis much smaller, but since the Flot library didn’t support this, it would take too long time to implement this, and the other option with units on the Y-axis is also good since you can easily see which one is for which line on the graph.

CONCLUSION
The RESTful Cloud Server can now store incoming data from web services. So when the client is ready, it can start using the RCS to store its data. The RCS also has users and administrators that have different authorizations, so that an administrator has more rights than a normal user.

The user can login to the web client and view his or hers feeds data that can be displayed in graphs and multigraphs. The multigraph is for viewing more than one graph at a time in a dynamic view, the user can select which streams in the feed it wants to see in it. The user can also modify the feeds of the user.

When the user logs in on the web client it gets authenticated using the e-mail and password. The e-mail and password are sent using REST call to the RCS. The password is hashed on the server using RFC2898 hashing, and is stored on the server with that hashing. When the authentication call is received, the password will be hashed with the same function and compared to the password in RCS’s database.

The services on the RCS can only be used if the client is authenticated and has the right authorization to use the service. Functions like paging are implemented to increase the speed and decrease the bandwidth use for the client.
REFERENCES
During the time of implementation of the RCS we used www.google.com to search for answers for problems that came up. When looking at the answers it was important for us to look for the answers with the most relevance, for example: When implementing function with the use of ServiceStack, when searching for a solution for a problem, the creator of ServiceStack had answered a lot of them before. Otherwise we read a lot of the information for the respective libraries documentation pages, since the information there is very relevant to the respective library.

Another way to determine if an answer was relevant was to look on how many have supported the answer and if there were more answers that contained the same type of solution.

The dates for webpages are the last time a check was made of the references.


25. Flot homepage, Ole Laursen, 29-05-2013, IOLA. http://www.flotcharts.org/


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Linus Karlsson]