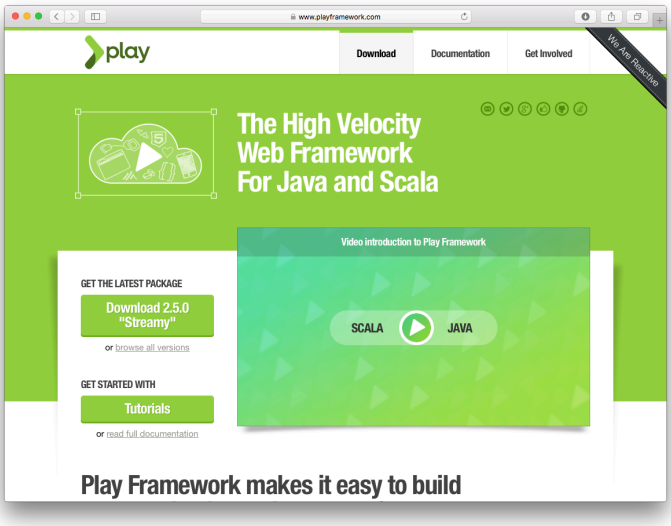
Water Usage Visualization Tutorial

1. Installation

Go to www.playframework and download the current version.



For this tutorial we will use Play 2.5.1 “Streamy”. Make sure that you downloaded the offline distribution of Play.

Check that you have java 8+ installed in your machine. You can follow the installation instructions required for this version of play at:
https://www.playframework.com/documentation/2.5.x/Installing

Optionally, you can install an IDE such as **ecplise** if you prefer not to use the provided editor of Play/Activator.
To use eclipse, add the following line to project/plugins.bst

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" % "4.0.0")
```

After building your app, you should be able to use eclipse and import the project into your workspace.

2. Start play and build a first seed java application

2.1 If you love command lines

To create your new play java application you simply need to type the following command: (note that WaternomicsApp is the name of the application)

```
[activator folder location]/bin/activator new WaternomicsApp play-java
```

To start the application, we use the following command:

```
cd WaternomicsApp
```

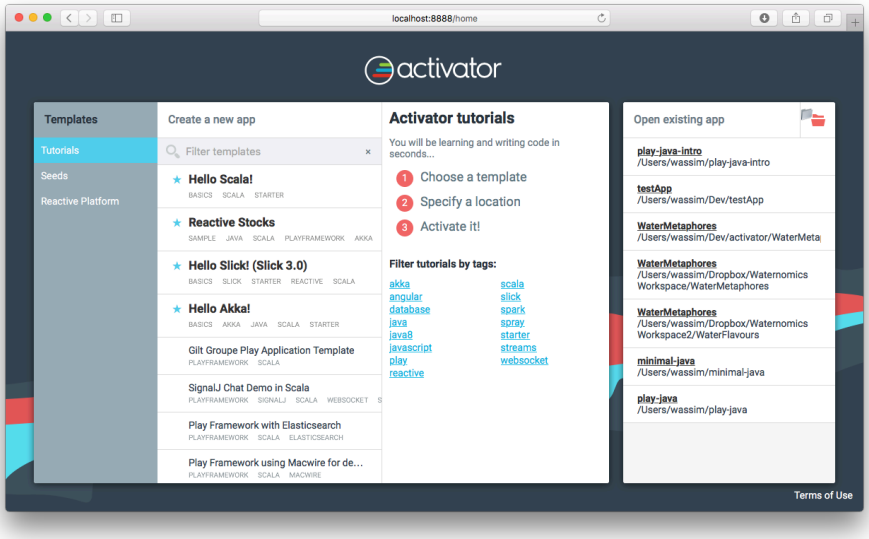
```
[activator folder location]/bin/activator ui
```

2.2 If you prefer UIs

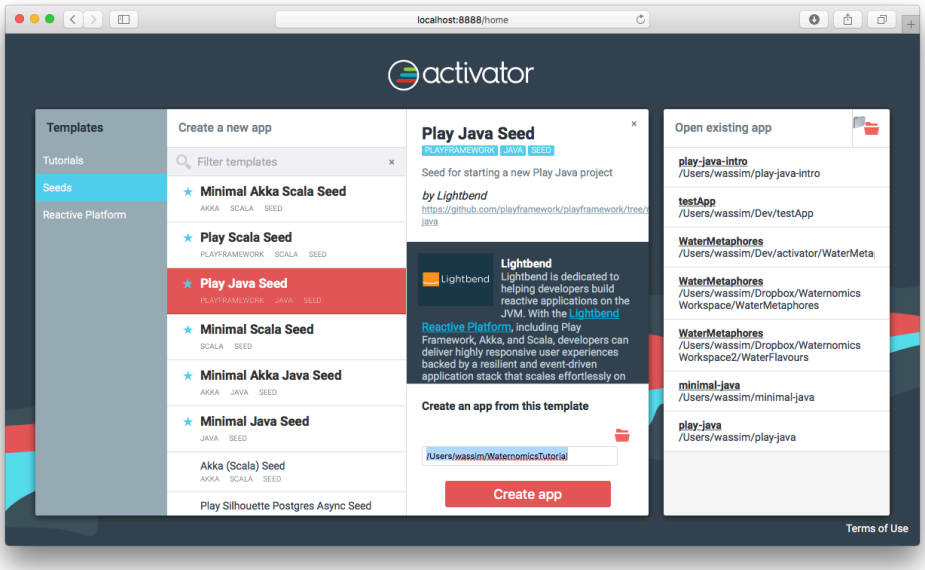
To create your application using activator, type the following command:

```
[activator folder location]/bin/activator ui
```

Activator facilitates the first steps of creating your app. Activator ui will run as web application and should show the following screen on you browser running on http://localhost:8888/home

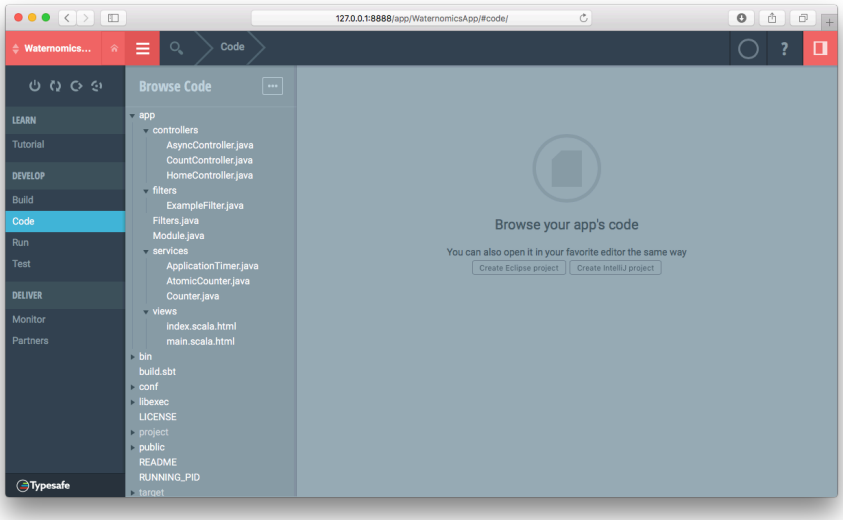


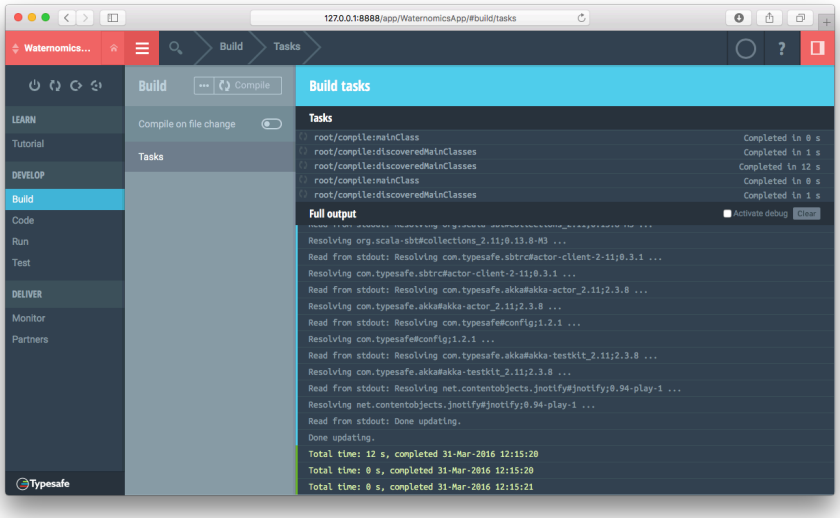
We will use a predefined template to speed up the development of our application. For this we will select a seed template for **Seeds > Play Java Seed** . Specify your application destination (This also will be your application name) directory and click on **Create app.**

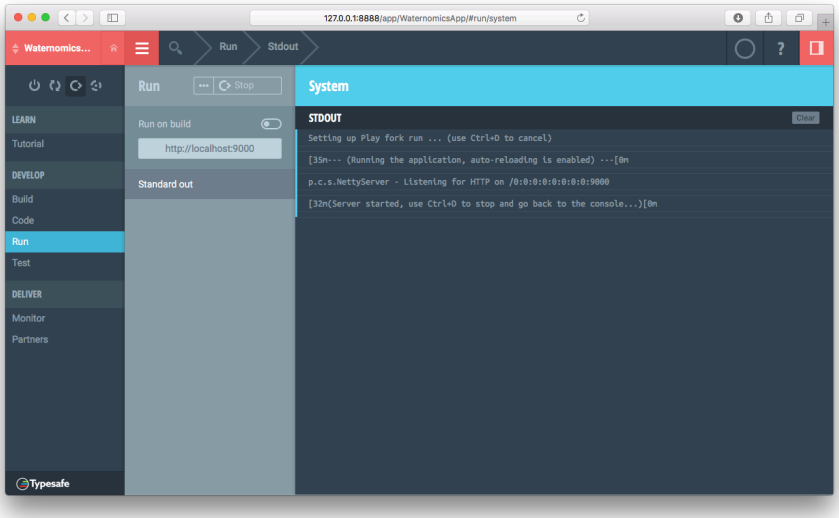


2.3 After creating your first application

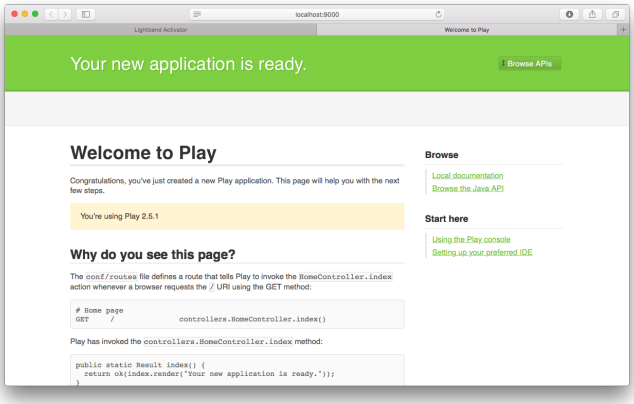
Activator allows you to view/edit the application code, compile the application, running etc.







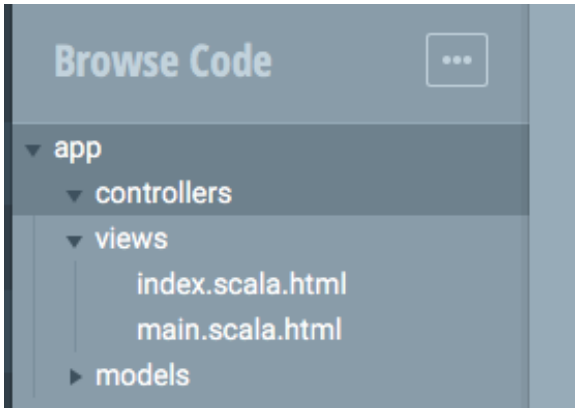
After running the application, you can visit the address: http://localhost:9000. 9000 is the default port where the application is running. You can change this port as well as other configuration parameters of your app by following the instructions given by the documentation of play framework.



This tutorial does not cover the part of configuration of the app. In the rest of the tutorial we will change the code of this application to make our own.

3. Hello World App

a. Remove all the files and directories from the app folder and keep only the following folders and files:



Play uses the MVC (Model View Controller) programming models where (very briefly) the following folders are needed:

- models: contains all the classes that describe your data
- controllers: contains all the classes defining all the operations that you want to do on this data (models)
- views: contains all the UIs for visualizing your data

b. Change the content of main.scala.html by copying the following code:

```
@(title: String)(content: Html)
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title>@title</title>
    <link rel="stylesheet" id='style-css' type='text/css' media='all'
href="@routes.Assets.versioned("stylesheets/style.css")"/>
    <link href='https://fonts.googleapis.com/css?family=Montserrat:400,700'
rel='stylesheet' type='text/css'>
    <link rel="shortcut icon"
href="@routes.Assets.versioned("images/drop.ico")" />
      <link href="@routes.Assets.versioned("stylesheets/c3.css")"
rel="stylesheet" type="text/css"/>
      <!-- Load d3.js and c3.js -->
      <script src="@routes.Assets.versioned("javascripts/d3.min.js")"
charset="utf-8"></script>
      <script
src="@routes.Assets.versioned("javascripts/c3.min.js")"></script>
</head>
<body class="overview-page">
<header>
       <h1>@title</h1>
</header>
    @content
    <footer>
    </footer>
</body>
</html>
```

This code is the template of our application views. It is written in scala. The main things to spot inthis code are the following:

- @(title: String)(content: Html) indicates that this scala code needs two parameters: **title** and **content**. These parameters are used in the following html code as **@title** and **@content**.
- "@routes.Assets.versioned("…") indicates the path to the files required such as the style.css and drop.ico. These files will be placed in the public folder of your application: e.g., [App lication folder path]/public/images (note that these files are provided in the tutorial accompanying archive).
- The rest is an html code that you should be able to understand.

c. Change the content of index.scala.html by copying the following code:

```
@main("Waternomics Tutorial") {
    <div class="container" id="main-container">
       <div class="section-title">
                  <h2>Title: Hello World!</h2>
        </div>
        <div id=".section-content">
          <p align="center">
                     Content: Hello World!
              </p>
       </div>
       </div>
    </div> <!-- /container -->
}
```

This code is also written in scala. The main highlights here are:

- The @main(parameter1){parameter 2} is a calla to main.scala.html that has been previously created.
- Parameter 1 corresponds to the @title defined in main.scala.html (note that it is of type String)
- Parameter 2 corresponds to the @content defined in main.schala.html (note that it is of type HTML)
- The rest is an html code that you should be able to understand.

d. Create the class Application.java in controllers package by copying the following code:

```
package controllers;

import play.mvc.*;

import views.html.*;

public class Application extends Controller {

  public Result index() {
        //Render the index page
    return ok(index.render());
  }
}
```

This class extends **play.mvc.Controller**, a predefined class from Play (note the **import play.mvc.*;**).
Application.java has only one method **index()** that calls the index.scala.html via **return ok(index.render());.**

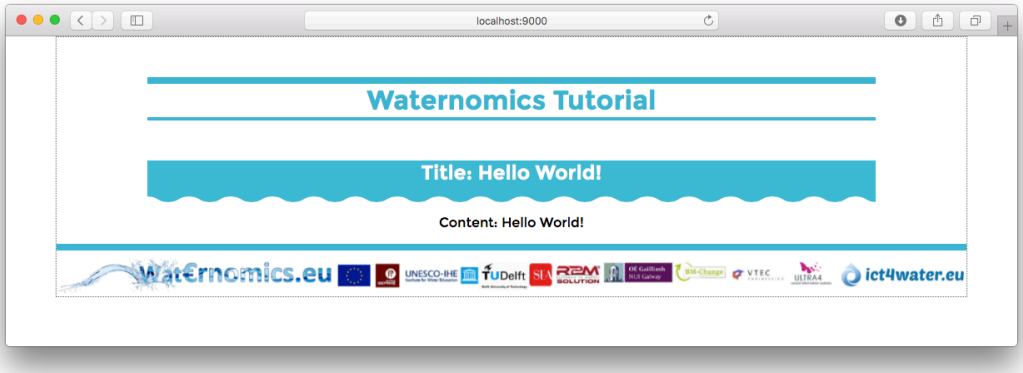
e. Change the routes to point to your new application

In Play, routes indicate what do you want to execute if a user visits a particular page. In our case, we need to indicate that each time the user visits http://localhost:9000, we call the method **controllers.Application.index()**.

This can be done via the file **conf/routes**. Change the content of this file by using the following listing:

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

# An example controller visiting the index page
GET     /                       controllers.Application.index

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file           controllers.Assets.versioned(path="/public", file: Asset)
```

f. All done!
You can now visit http://localhost:9000 and the following page should be displayed!



4. Make your app query DRUID

This part of the tutorial, we will create a class that queries DRUID to get the last 30 days of water consumption of a particular sensor. The results of the daily readings will be presented in JSON format as follows:

```
{
"Last30Days": [
    {
        "result":{
```

```
                    "consumption":47.000000298023224
                  },
                  "timestamp":"2016-03-01T00:00:00.000Z"
      },
…
]
}
```

As an example, we will use **M1n** as sensor ID to query the data for the last 30 days. A Druid Query will look like this:

```
{
"queryType": "timeseries",
 "dataSource": "test_neb4",
"intervals": [ "2016-03-05/2016-04-04" ],
"granularity": "day",
"filter": {"type": "selector","dimension": "dSensor", "value": "M1n"},
"aggregations": [    {"type": "doubleSum", "fieldName": "mValue", "name":
"consumption"} ]
}
```

We can define a method that creates this type of query as follows:

```
private String defineQuery(String sensorID){
              // This method constructs the DRUID query and returns it in the form of
a string
              // This method takes as input the sensorID. The interval is calculated
using the method getLast30DaysInterval()
              // IT can be further cutomized based on the user requirements and types
of queries that he wants to have.
                String query = "{"
                                    +"  \"queryType\": \"timeseries\","
                                    +"  \"dataSource\": \"test_neb4\","
                                    +"  \"intervals\": [
\""+getLast30DaysInterval()+"\" ],"
                                    +"  \"granularity\": \"day\","
                                    +"  \"filter\": {\"type\":
\"selector\",\"dimension\": \"dSensor\", \"value\": \""+sensorID+"\"},"
                                    +"  \"aggregations\": ["
                                    +"     {\"type\": \"doubleSum\", \"fieldName\":
\"mValue\", \"name\": \"consumption\"}"
                                    +"  ]"
                                    +"}";
              return query;
       }
```

This method uses *getLast30DaysInterval()* for creating the interval part:

```
public static String getLast30DaysInterval() {
              // This method constructs a date interval that covers the last
30 days;
              // The result is a string in the form YYYY/MM/DD-YYY/MM/DD
(startDate-endDate)
              String result = ""; // String that contains the result

              Calendar cl = Calendar.getInstance(); // cl contains an
instance of the system calendar
```

```
            int endDay= cl.get(cl.DAY_OF_MONTH); // endDay contains the
current day of the month -- today
            int endMonth = cl.get(cl.MONTH)+1; // endMonth contains the
current month -- this month
            int endYear = cl.get(cl.YEAR); // endYear contains the current
year -- this year

              cl.add(cl.DAY_OF_MONTH, -30); // Change the date of cl by
moving 30 days back

              int startDay = cl.get(cl.DAY_OF_MONTH); // startDay
contains the day of the month 30 days ago
              int startMonth= cl.get(cl.MONTH)+1;// startMonth contains
the month 30 days ago
              int startYear = cl.get(cl.YEAR);// startYear contains the
year 30 days ago

              //construct the result in the form of
startYear/startMonth/startDay-endYear/endMonth/endDay --> YYYY/MM/DD-
YYYY/MM/DD
              // not that we can use String.format("%02d", n) for making n
hold on 2 digits. e.g, 2 will appear as 02
            result = startYear + "-" + String.format("%02d", startMonth)+"-
"+String.format("%02d", startDay)+"/"+endYear + "-" + String.format("%02d",
endMonth) +"-"+String.format("%02d", endDay);

            return result;
      }
```

After creating the query we need to execute it and get the results:

```
      private void executeQuery(String jsonQuery) throws Exception{
            //This method takes as input a DRUID query and updates the list of
readings (allReadings class variable)

            clear(); // rests the values of allReadings and totalConsumption

            //Querying DRUID using the Analytics service offered by the Waternomics
platform is done by sending the query at part the url.
            //In order to make sure that the analytics service correctly receives
the query, we need to ensure that UTF-8 encoding is applied on the query string.
            // URLEncoder.encode(jsonQuery, "UTF-8") guarantees a proper encoding
of the query string.
            String safeQuery = URLEncoder.encode(jsonQuery, "UTF-8");

            //Construct the URL to the analytics service.
            // We need:
            //         - the address of the service API:
http://linkeddataspace.waternomics.eu:8011/restapi/getrawreadings?
            //         - the userID and APIKey available upon request and approval
of the site manager and Waternomics platform administrators
            String safeUrl=
"http://linkeddataspace.waternomics.eu:8011/restapi/getrawreadings?"
                        +"userID=medcollege"
                        +"&APIKey=0b9a154c-e533-4d4f-850b-76f133c674d2"
                        +"&jsonQuery="+safeQuery;// put in your url

            //The following lines are regular http get requests sent to the
analytics service.
            // The response is parsed and captured in a StringBuilder
```

```
"queryResponse"
                StringBuilder queryResponse = new StringBuilder();
                URL url = new URL(safeUrl);
                HttpURLConnection conn = (HttpURLConnection) url.openConnection();
                conn.setRequestMethod("GET");
                BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
                String line;
                while ((line = rd.readLine()) != null) {
                        queryResponse.append(line);
                }
                rd.close();

                // Now we need to parse the results for further analytics.
                // As we know that the result is in json format, we can use the gson
library to facilitate its parsing.
                // gson is the variable that contains the actual response.
                // Gson library requires a parameter that describes the structure of
the data captured by the class JSONReading
                Gson gson = new Gson();
                rds = gson.fromJson(queryResponse.toString(), new
TypeToken<List<JSONReading>>(){}.getType()) ;

                double total = 0; // a variable that will contain the total
consumption of a 1 day

                Iterator<JSONReading> it = rds.iterator();// Iterator used to parse the
json response

                while (it.hasNext()) {
                        // this loop permits to parse the list JSONReading
                        // for earch reading we construct its instance of the calss
Reading
                        // we also update the totalConsumption that contains the
overall consumption of the last 30 days
                        JSONReading jr = (JSONReading) it.next();
                        total = jr.result.consumption;
                        String timestamp = jr.timestamp;
                        int year =
Integer.parseInt(timestamp.substring(0,timestamp.indexOf("-")));
                        timestamp = timestamp.substring(timestamp.indexOf("-
")+1,timestamp.length());

                        int month =
Integer.parseInt(timestamp.substring(0,timestamp.indexOf("-")));
                        timestamp = timestamp.substring(timestamp.indexOf("-
")+1,timestamp.length());
                        int day =
Integer.parseInt(timestamp.substring(0,timestamp.indexOf("T")));

                        Calendar cal = Calendar.getInstance();
                        cal.set(year, month-1, day, 0, 0, 0);

                        String date = String.format("%02d",year) +"-" +
String.format("%02d", month)+ "-" + String.format("%02d", day);

                        // Construct the instance of the reading
                        Reading reading = new Reading(cal.getTimeInMillis(), total,
date);
                        //update the totalConsumption
                        totalConsumption += total;
                        //add the reading to the list of Readings
                        allReadings.add(reading);
                        //Sort the list of readings using their timestamp.
                        // This is required for later use.
```

```
                    Collections.sort(allReadings, new Comparator<Reading>() {

                    public int compare(Reading r1, Reading r2) {
                        if (r2.getReadingTimeStamp() > r1.getReadingTimeStamp()){
                           return -1;
                        }
                        else{
                           return 1;
                        }
                    }
                });
                    }
        }
```

Notice that we are using Gson library for handling json data. This requires two additional tasks:

- Add the Gson library to the list of dependencies in the build.sbt file. This is done by adding "com.google.code.gson" % "gson" % "2.3.1" in libraryDependencies.

You list of dependencies will like like this:

```
libraryDependencies ++= Seq(
 javaJdbc,
 cache,
 javaWs,
  "com.google.code.gson" % "gson" % "2.3.1"
)
```

- Define the structure of the json data. We have to create the calss JSONReading as used in the code. This class simply indicates the json keys and types. See the class in the following:

```
package models;

public class JSONReading {
               public String timestamp;
               public Result result;

               public class Result {
                           public double consumption;
                    }
}
```

The clear method simply initializes some of the Class variables as follows:

```
        public static void clear() {
               allReadings.clear();
               rds.clear();
               totalConsumption = 0;
        }
```

Let's test the querying of DRUID and see the results as json. To do so we can create the following method

```
	public Result getReadingsInJSON(String sensorID) throws Exception{
		//This method allows to test if the readings are properly
queried and can be visualised in json format.
		// Results can be shown in your browser by visitng
localhost:9000/getReadingsInJSON/{sensorID}
		executeQuery(defineQuery(sensorID));
		return ok(Json.toJson(allReadings));
	}
```

Before testing the result, we need to make sure that Play knows how when to query this send and where to send it.
This is done via routes (we did already some changes to the routes in the beginning of our app).
Add the following line to your conf/routes file:

```
GET  /getReadingsInJSON/:sensorID
controllers.Readings.getReadingsInJSON(sensorID:String)
```

This means: if the user visits the link http://localhost:9000/getReadingsInJSON/M1n he gets the readings of the last 30 days for the sensor M1n.
Can you verify that you app is doing so?

5. Visualize results in a chart

First of all we have to define the template of the page that will show the results in a chart.
We need to create a new file called chart.scala.html in the views folder.
The following is the content of our page:

```
@(message: String, data: String, timeseries: Html)

@main("Water usage visualization") {
    <div class="container">
       <div class="section-title">
                    <h2>@message</h2>
        </div>
        <div class="section-content">
          <div id="chart"></div>
       <script>
            var chart = c3.generate({
    data: {
       x: 'x',
       xFormat : '%Y-%m-%d',
        columns: [
            ['x', @(timeseries) ],
            ['Daily Water Usage', @data ]
        ],
        type: 'bar'
    },
    bar: {
        width: {
            ratio: 0.5 // this makes bar width 50% of length between ticks
```

```
        }

    },
    axis: {
     x: {
       type: 'timeseries',
       tick: {
              format: '%Y-%m-%d',
              rotate: 75
       }
      },

      y: {
        label: {
          text: 'Water Volume in m3',
          position: 'outer-middle'
        }
      }
    }
});
              </script>
       </div>
    </div> <!-- /container -->
}
```

Note that we need to give few parameters for this scala code:

- message: the title that we want to display to the chart section
- data and timeseries: two variables that are required by C3 scipt to create the chart.

We have now to prepare this data and send it to this page. The following two methods allow to parse the readings that we have already queried from DRUID and construct the data and timeseries strings required by C3 (syntax and use of C3 is out of scope of this tutorial, more details can be found in C3 documentation: http://c3js.org/gettingstarted.html).

```
      private String getChartReadings(){
             //In order to create a chart in C3 we need to prepare data that
goes on the x and y axis.
             //This method prepares the data for the y axis: values of the
readings.
             //The format of results should be in the form x1, x2, x3,...xn
             String result =""; // A string that contains the result
             for (Reading r : allReadings){
                    //to get only the integer part of the readings we can
use the .intValue of the class Double (notice the D vs. d)
                    result = result +
((Double)r.getReadingValue()).intValue() + ", ";
             }
             // before returning the result, we should remove the last two
characters (the last ", ")
             return result.substring(0,result.length()-2);
      }

      private String getChartTimeseries(){
             //In order to create a chart in C3 we need to prepare data that
```

```
goes on the x and y axis.
              //This method prepares the data for the x axis: timeseries
presented as dates.
              //The format of results should be in the form d1, d2, d3,...dn
              //Each date can be presented as 'YYYY-MM-DD'
              String result ="";
              for (Reading r : allReadings){
                     result = result +"'"+ r.getDate() + "', ";
              }
              return result.substring(0,result.length()-2);
       }
```

Now in order to get the chart created and displayed to use two main steps are required:

- define the method that creates the data.
- render the chart page.

This is done as follows:

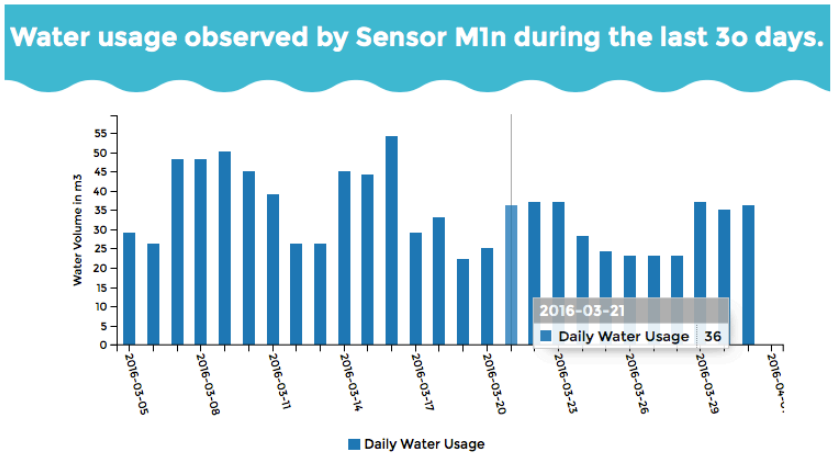
```
       public Result getChart(String sensorID) throws Exception{
              //Define and Execute the DRUID query for the requested sensor
              executeQuery(defineQuery(sensorID));

              //Create the data for the c3 chart
              String data = getChartReadings();
              //Create the timeseries for the c3 chart
              String time = getChartTimeseries();
              //Create the HTML content for the foorptin section of the
application

              //retunr the results to the scala page chart.scala.html
              return ok(chart.render(message,data,new Html(time)));
       }
```

Don't forget to add the following entry to your routes file:

```
GET  /:sensorID
controllers.Readings.getChart(sensorID:String)
```

Note the syntax of routes if we are using a parameter in the method!

After visiting this link: http://localhost:9000/M1n , you should be able to see this chart:



6. Add some flavours to your app

We will now use the Water Flavours application http://linkeddataspace.waternomics.eu:8012 in order to visualize water values with a new dimension different from volumes in m3.
One way to do this is to visualize the water volumes in terms of footprint of products, or domestics uses and metaphors.
In this tutorial we will use footprints. We will query the Water Flavours API to get one random water footprint.

We will use the class variable that contains the total consumption of the last 30 days and convert into a footprint value and visualize it a another section of the page chart.html.

To do so we need to create the following method to get a full description using the water footprint.

```
	private String getTotalHTMLDescription() throws Exception{
		//This method retunrs a string that contains that html content
required for the top part of the application.
		//It contains a statement regarding the water footprint of the
entire water consumed within the last 30 days.
		String result; //A string that contaisn thre result.

		//Use the Waternomics Water Flavours API to get the water
footprint.
		//To use this API we need:
		//		- Address of the API:
http://linkeddataspace.waternomics.eu:8012/restapi/getRandomFootprints
		//		- Number of returned footprint values ( in our
case it is 1)
		//		- The amount and unit of Water : in our case the
quantity is in totalConsumption and the unit is m3
		URL url = new URL
("http://linkeddataspace.waternomics.eu:8012/restapi/getRandomFootprints/1/"

	+((Double)totalConsumption).intValue()+"/m3");
		//The following few lines will request the footprint from the
```

```
Water Flavour API
                StringBuilder sb = new StringBuilder();
                HttpURLConnection conn = (HttpURLConnection)
url.openConnection();
                conn.setRequestMethod("GET");
                BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
                String line;
                while ((line = rd.readLine()) != null) {
                   sb.append(line);
                }
                rd.close();
                //The result from the Water Flavour App is in JSON.
                //We use Gson library to easily parse this result.
                try{
                Gson gson = new Gson();
                List <Footprint> ftJson = new ArrayList<Footprint>();
                //gson needs the format of the data that follows the
models.Footprint class
                 ftJson = gson.fromJson(sb.toString(), new
TypeToken<List<Footprint>>(){}.getType()) ;
                 //In this case, we know that the returned result contains
only 1 footprint.
                 //We simply get the first element of the returned list of
footprints.
                 Footprint ft = ftJson.iterator().next();
                 //Create the html content as a string in the variable result
                 result = ft.description
                                + " [<a target='_blank'
href='"+ft.reference+"'>Reference</a>"+"]"
                                +"<br>"
                                +"<img width ='200' src='"+ft.image+"'>"
                                + "<h1>" +((Double)ft.value).intValue() +"
"+ft.unit+"</h1>";
                }catch(Exception e){
                       result = "Unable to determine the Water Footprint for
this provided sensor!";
                 }
              return result;
      }
```

This method will be sent to the chart page via another parameter. We can also create a title for this new section and use it in the page.

To do so we need to change the chart.scala.html to the following (Changes are highlighted in red):

```
@(message: String, data: String, timeseries: Html, total: String,
totalContent: Html)

@main("Water usage visualization") {
    <div class="container">
        <div class="section-title">
                <h2>The total water Used in the last 30 days is @total
</h2>
```

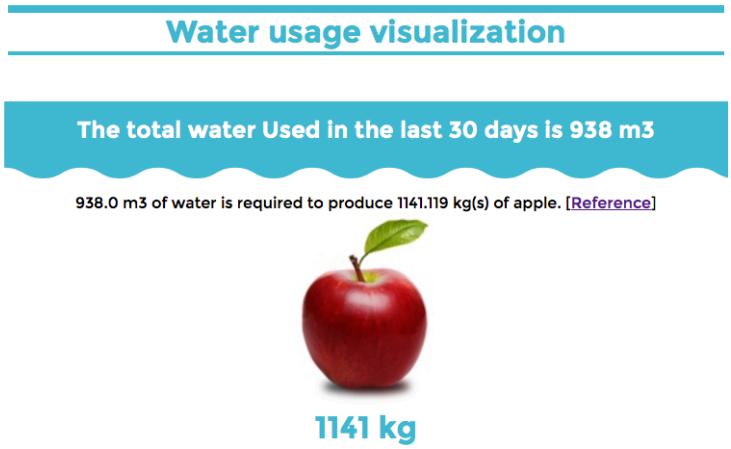
```
        </div>
        <div class="section-content">
        @totalContent
        </div>
      <div class="section-title">
                <h2>@message</h2>
       </div>
       <div class="section-content">
         <div id="chart"></div>
      <script>
            var chart = c3.generate({
    data: {
       x: 'x',
       xFormat : '%Y-%m-%d',
        columns: [
            ['x', @(timeseries) ],
            ['Daily Water Usage', @data ]
        ],
        type: 'bar'
    },
    bar: {
        width: {
            ratio: 0.5 // this makes bar width 50% of length between ticks
        }
   },
    axis: {
     x: {
       type: 'timeseries',
       tick: {
              format: '%Y-%m-%d',
              rotate: 75
       }
      },

      y: {
        label: {
          text: 'Water Volume in m3',
          position: 'outer-middle'
        }
      }
    }
});
             </script>
      </div>
    </div> <!-- /container -->
}
```

We also need to change the getChart method to the following:

```
      public Result getChart(String sensorID) throws Exception{
            //Define and Execute the DRUID query for the requested sensor
            executeQuery(defineQuery(sensorID));
            //Create the data for the c3 chart
            String data = getChartReadings();
            //Create the timeseries for the c3 chart
            String time = getChartTimeseries();
            //Create a title for the chart section
```

```
            String message = "Water usage observed by Sensor " +sensorID +"
during the last 30 days.";
            //Create the HTML content for the footprint section of the
application
            String total = getTotalHTMLDescription();
            //Prepare a string with the total Water consumption over the
last 30 days to be used in the footprint print title
            String totalCon = ((Double)totalConsumption).intValue() + "
m3";

            //retunr the results to the scala page chart.scala.html
            return ok(chart.render(message,data,new Html(time),totalCon,
new Html(total)));
        }
```

After visiting this link: http://localhost:9000/M1n , you should be able to see the footprint section as follows:



That's it! All done for today… More ideas and applications are welcome!

Do not hesitate to contact me or any other person of the Waternomics team http://waternomics.eu.

Job Title:
Researcher

Research Group:
Linked Data Semantic Web

Insight Affiliation: NUIG

Email:
wassim.derguech@insight-centre.org

Phone: +353 91 495113