

# REPORT ON DELIVERABLE 5.3.2 Updated Knowledge Discovery Workbench

PROJECT NUMBER: 619186  
START DATE OF PROJECT: 01/03/2014  
DURATION: 42 months



DAIAD is a research project funded by European Commission's 7th Framework Programme.

The information in this document reflects the author's views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided "as is" without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/ her sole risk and liability.

<b>Dissemination Level</b>	Public
<b>Due Date of Deliverable</b>	Month 40, 29/06/2017
<b>Actual Submission Date</b>	29/06/2017
<b>Work Package</b>	WP5 Big Water Data Analysis
<b>Task</b>	Task 5.4 Knowledge Discovery Workbench
<b>Type</b>	Prototype
<b>Approval Status</b>	Submitted for approval
<b>Version</b>	1.0
<b>Number of Pages</b>	48
<b>Filename</b>	D5.3.2_Updated_Knowledge_Discovery_Workbench.pdf

## Abstract

This report presents an overview of the Prototype Deliverable D5.3.2 "Updated Knowledge Discovery Workbench". The Workbench is a collection of web based UI elements enabling users to analyze, discover, and experiment with water consumption data. The Workbench UI elements invoke our Analytics and Forecasting engine in a seamless manner, hiding away the inherent complexity of Big Data management and analysis, and enabling users to focus on exploration, analysis and knowledge extraction.

## History

version	date	reason	revised by
0.1	28/03/2017	First draft	Spiros Athanasiou
0.4	12/05/2017	Revisions in all sections	Yannis Kouvaras, Michalis Alexakis, Nikos Georgomanolis, Stelios Manousopoulos, Nikos Karagiannakis
0.5	28/05/2017	Revisions in all sections	Giorgos Giannopoulos
0.8	12/06/2017	Revisions in all sections	Yannis Kouvaras, Michalis Alexakis
0.9	20/06/2017	Minor edits, updated screenshots	Nikos Georgomanolis, Stelios Manousopoulos, Nikos Karagiannakis
1.0	29/06/2017	Final version	Spiros Athanasiou

## Author list

organization	name	contact information
ATHENA RC	Spiros Athanasiou	<a href="mailto:spathan@imis.athena-innovation.gr">spathan@imis.athena-innovation.gr</a>
ATHENA RC	Giorgos Giannopoulos	<a href="mailto:giann@imis.athena-innovation.gr">giann@imis.athena-innovation.gr</a>
ATHENA RC	Yannis Kouvaras	<a href="mailto:jkouvar_@imis.athena-innovation.gr">jkouvar_@imis.athena-innovation.gr</a>
ATHENA RC	Michalis Alexakis	<a href="mailto:alexakis@imis.athena-innovation.gr">alexakis@imis.athena-innovation.gr</a>
ATHENA RC	Stelios Manousopoulos	<a href="mailto:smanousopoulos@imis.athena-innovation.gr">smanousopoulos@imis.athena-innovation.gr</a>
ATHENA RC	Nikos Georgomanolis	<a href="mailto:ngeorgomanolis@imis.athena-innovation.gr">ngeorgomanolis@imis.athena-innovation.gr</a>
ATHENA RC	Nikos Karagiannakis	<a href="mailto:nkaragiannakis@imis.athena-innovation.gr">nkaragiannakis@imis.athena-innovation.gr</a>

# Executive Summary

This report presents an overview of the Prototype Deliverable D5.3.2 “Updated Knowledge Discovery Workbench”. The Workbench is a collection of web-based UI elements enabling expert users to analyze, discover, and experiment with water consumption data. In the context of the entire DAIAD system, the Workbench provides the essential building blocks for the DAIAD@commons and DAIAD@utility software. In this manner, it invokes our Analytics and Forecasting engine (D5.2.2) in a seamless manner, hiding away the inherent complexity of Big Data management and analysis, enabling users to focus on exploration, analysis and knowledge extraction.

The remainder of this document is structured as follows.

In Section 1 we present the architecture of the Knowledge Discovery Workbench, elaborating on the intended functionality in the context of the entire DAIAD system. Further, we present the core libraries and frameworks used during development, as well as dependencies with DAIAD APIs.

In Section 2 we present the core UI elements of the Knowledge Discovery Workbench in detail, presenting for each one its high-level operation and implementation, its various properties, an invocation example, and indicative examples of its application within DAIAD. The full documentation and source code for each UI element are available in our public repository.

# Abbreviations and Acronyms

API	Application Programming Interface
AJAX	Asynchronous JavaScript and XML
AOP	Aspect Oriented Programming
APK	Android application package
BT	Bluetooth
CI	Continuous Integration
CORS	Cross-Origin Resource Sharing
CSRF	Cross-Site Request Forgery
CSS	Cascading Style Sheets
DOM	Document Object Model
DTO	Data Transfer Object
JSON	JavaScript Object Notation
MR	MapReduce
MVC	Model View Controller
MVP	Minimum Viable Product
OGC	Open Geospatial Consortium
ORM	Object Relational Mapper
RERO	Release Early, Release Often
REST	Representational State Transfer
RF	Radio Frequency
RPC	Remote Procedure Call
SPA	Single Page Application
SWM	Smart Water Meter
UI	User Interface

# Table of Contents

1. Implementation .....	7
1.1. Overview.....	7
1.1.1. Application Patterns and Design.....	8
1.2. Libraries and Frameworks .....	8
1.2.1. React.....	8
1.2.2. Redux.....	9
1.2.3. React-Router-Redux.....	10
1.2.4. React-Bootstrap .....	10
1.2.5. ECharts.....	10
1.3. Dependencies .....	11
1.3.1. Data API.....	11
1.3.2. Configuration API.....	12
2. UI Elements .....	14
2.1. LineChart.....	14
2.2. Chart.....	20
2.3. ChartContainer .....	22
2.4. Leaflet Wrapper .....	23
2.5. Table.....	29
2.6. Toolbar .....	34
2.7. ReportPanel.....	36
2.8. UnitReport .....	38
2.9. Widgets.....	40
2.10. Wizard .....	42

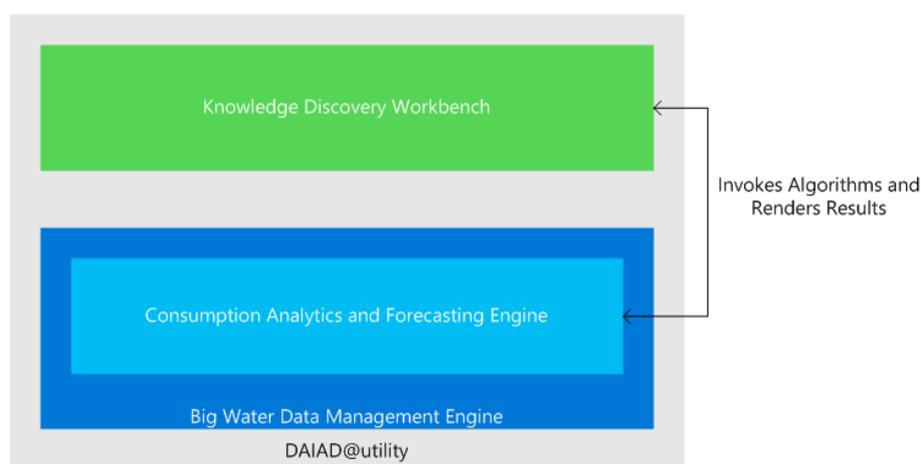
# 1. Implementation

## 1.1. Overview

The Knowledge Discovery Workbench is a collection of web-based UI elements enabling expert users to analyze, discover, and experiment with water consumption data. In the content of the entire DAIAD system, the Workbench provides the essential building blocks for the DAIAD@commons and DAIAD@utility software. In this manner, it invokes our Analytics and Forecasting engine (D5.2.2) in a seamless manner, hiding away the inherent complexity of Big Data management and analysis, enabling users to focus on exploration, analysis and knowledge extraction.

Specifically, all software components and elements developed in the context of WP5 (see Figure 1) have the following responsibilities:

- D5.1.2 Updated Big Water Data Management Engine. It handles the entire data lifecycle, providing scalable data management and querying services. An overview of the Data Engine is provided in the report for Prototype Deliverable D5.1.2.
- D5.2.2 Updated Consumption Analytics and Forecasting Engine. It provides the full suite of analytics and forecasting services, executed within the Big Water Data Management Engine. Essentially, it provides the ‘business logic’ of both applications, executing the appropriate analysis algorithms and returning the results to the UI elements. An overview of the Engine is provided in the Report for Prototype Deliverable D5.2.2.
- D5.3.2 Updated Knowledge Discovery Workbench. It provides all UI elements for presenting and invoking the analysis results of D5.2.2. In this manner, we hide the complexity of the underlying analytics and data engine, providing a coherent and simple to use interface for users.



*Figure 1: Software components and elements developed in the context of WP5*

The Knowledge Discovery Workbench applies external libraries and frameworks detailed in the following subsections. During the presentation that follows, we frequently make references to these libraries and frameworks, since they affect the implementation details of each component. For a thorough presentation of the complete DAIAD system architecture the reader is invited to consult the Report on Prototype Deliverable D1.4 ‘Final DAIAD System’.

### 1.1.1. Application Patterns and Design

In DAIAD we apply the Model View Controller pattern (MVC) and the Single Page Application (SPA) web application design. This pattern and design are used extensively and they strongly influence the structure of the source code. A short explanation for each follows; a broader coverage of these topics is outside the scope of this document:

- **Model View Controller (MVC).** The goal of the Model View Controller pattern is to separate code responsibilities into three parts. The Model, which represents application domain data and logic, the View, which is responsible for the data presentation and the Controller, who receives user interactions and updates the Model appropriately. This separation increases code testability and also improves a developer team’s productivity. Nowadays, there are many variants of the MVC pattern and each MVC framework may implement the pattern in different ways. For the DAIAD implementation we are using the Spring Framework and its corresponding MVC module.
- **Single Page Applications (SPAs)** offer increased UI usability that is in par with desktop applications. In contrast to traditional web applications, a SPA application is initialized by loading only a single web page. After initialization, any additional resources such as data or JavaScript code files are loaded dynamically on demand using Asynchronous JavaScript and XML (AJAX) requests. Moreover, client-side code is usually implemented using the MVC pattern or some variant of it.

## 1.2. Libraries and Frameworks

### 1.2.1. React

React<sup>1</sup> is a JavaScript framework for building interactive User Interfaces. React can be thought as the View in the MVC pattern that allows users to build reusable UI components and promotes composition of existing ones. Each component maintains its internal state which controls the rendering process. Whenever state changes, only the parts of the Document Object Model (DOM) that are affected are updated. This is achieved by using a virtual representation of the DOM that efficiently detects changes to the actual DOM. The latter feature makes React interoperability with other UI libraries more challenging. Recommended templating in React is performed with the help of JSX<sup>2</sup>, an XML-like syntax with a smooth learning curve for HTML-familiar developers.

---

<sup>1</sup> <https://facebook.github.io/react/>

<sup>2</sup> <https://facebook.github.io/jsx/>

## 1.2.2. Redux

Redux<sup>3</sup> is a predictable state container for JavaScript applications that is very popular for handling the increased application logic complexity that arises in Single Page Applications. In such applications, the state management becomes increasingly harder since various user interactions –which quite often involve asynchronous requests - result in state changes. Redux attempts to manage state in a predictable way by imposing specific restrictions on how and when state updates can occur. Redux makes a perfect match to React by deferring component state management to Redux. It was based on the principle ideas of Flux<sup>4</sup> for making the flow of an application unidirectional. The main difference Redux introduced is the core idea of keeping the state in a single store (following a Single source of Truth principle), instead of multiple stores. The single application state maps at any moment to its view representation via React UI components. User actions such as clicks may dispatch actions that change the state in a predefined way with the help of reducers that dictate how a specific action modifies the application state. In this manner, a one-way flow is achieved, making the application easy to reason about, debug and scale. The abstract application flow is shown in Figure 2.

- Store: In redux the store holds the entire application state, which is the representation of the application at any given time. It is an object containing any number of valid JS data types, such as numbers, strings, booleans, arrays, or other objects. A key concept is that the state object cannot be mutated directly, but only by emitting actions.
- Actions: Actions can be dispatched by user interactions or other actions and cause the state to change. There are two types of actions, simple and complex actions or thunks that execute with the help of a special thunk middleware<sup>5</sup>. Simple actions are plain objects containing the unique action type and any data that needs to be passed to the store. Thunks are functions that get access to the state and can perform asynchronous operations (such as fetching data from the API) and/or orchestrate multiple simple action dispatches.

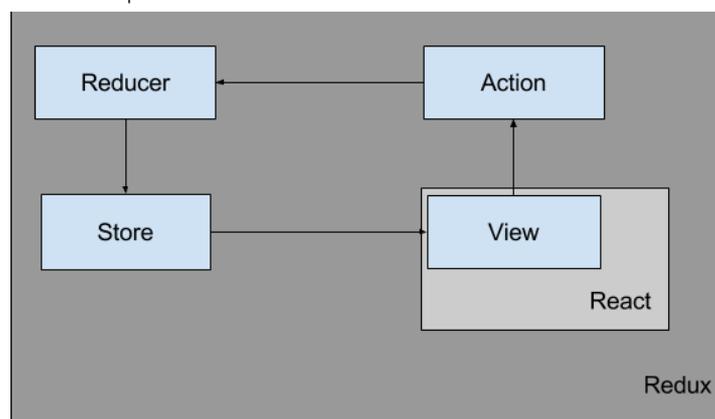


Figure 2: Redux Application flow

- Reducers: Reducers are pure functions that determine how an action modifies the state. Multiple reducers can be combined, each responsible for mutating a specific part of the state. Reducers do not

<sup>3</sup> <http://redux.js.org/>

<sup>4</sup> <https://facebook.github.io/flux/docs/overview.html>

<sup>5</sup> <https://github.com/gaearon/redux-thunk>

mutate the state, but instead return a new state object, which allows easy recognition of any changes so that the view can be updated.

- **Views:** Redux combines very well with React as its view layer with the help of react-redux library. In react-redux terminology, React components are divided into two types: smart components or containers and pure or presentational components. Containers are aware of redux and map parts of the state and action callback functions to react component properties, and are responsible for causing the components to re-render any time a mapped property has changed. On the other hand, presentational components are just pure functions of their input properties, completely ignorant of redux, allowing successful separation of logic and templating.

### 1.2.3. React-Router-Redux

React Router Redux is a JavaScript library that allows an application implemented using React and Redux to keep the application state in sync with routing information. This feature is achieved by automatically storing additional data about the current URL inside the state. This information is then propagated to React which can in turn suitably change the component tree rendering process. If there is no need for syncing routing information and application state, a simpler implementation can be obtained by using the React Router<sup>6</sup> library. The latter provides support for keeping only the UI in sync with the URL.

### 1.2.4. React-Bootstrap

React-Bootstrap<sup>7</sup> is a library of reusable UI components for the React framework. It offers the look-and-feel of the Twitter Bootstrap<sup>8</sup> library using the React syntax, but has no dependencies on any 3<sup>rd</sup> party libraries like jQuery. React-Bootstrap offers a comprehensive list of UI components such as buttons, menus, form input controls, modal dialog, to name a few. All components can be used as provided or customized using CSS.

### 1.2.5. ECharts

ECharts<sup>9</sup> is a rich and versatile JavaScript charting library for building interactive charts, based on a standalone and lightweight rendering framework (ZRender). It manages (using an opaque handle) a given DOM node as a subtree, and directly draws to a canvas element lying inside this subtree. It supports a great variety of chart types including line, column, scatter, pie, radar, candlestick, chord, gauge, funnel, map and heatmap charts. It also supports data visualizations, not usually regarded as charts, including a tree map, a tree graph and a Venn diagram. Moreover, individual charts can be composed to create more complex data representations.

ECharts is highly optimized for handling hundreds of thousands (can easily cope with 200K) of data points, making it a seamless solution for big data analysis and visualization. Further, Echarts is *theme-able*, i.e. a great subset of appearance-related chart options can be supplied by referencing an external theme JSON object. This allows an entire application to share (or override) a uniform look-and-feel for charts by keeping appearance-related options in a single place.

---

<sup>6</sup> <https://github.com/reactjs/react-router>

<sup>7</sup> <https://react-bootstrap.github.io/>

<sup>8</sup> <http://getbootstrap.com/>

<sup>9</sup> <https://ecomfe.github.io/echarts/index-en.html>

## 1.3. Dependencies

### 1.3.1. Data API

The Data Application Programming Interface (API) supports querying data persisted by the Big Water Data Management Engine developed in WP5 and presented in deliverable D5.1.2. It is exposed as a Hypertext Transfer Protocol (HTTP) Remote Procedure Call (RPC) API that exchanges JSON encoded messages and has two endpoints, namely, the Action API and HTTP API endpoints. The former is a stateful API that is consumed by the DAIAD web applications. The latter is a Cross-Origin Resource Sharing (CORS) enabled stateless API that can be used by 3<sup>rd</sup> party applications.

The API exposes data from three data sources, namely, smart water meter data, amphiro b1 data and forecasting data for smart water meters. The query syntax is common for all data sources. Moreover, smart water meter and amphiro b1 data can be queried simultaneously. However, a separate request must be executed for forecasting data.

The API accepts a set of configuration options and filtering criteria as parameters and returns one or more data series consisting of data points which in turn have one or more aggregate metrics like sum, min, max, count or average values. More specifically, the input parameters are:

- **Timezone:** Converts the results to the specified timezone. If a timezone is not set, the timezone defined in the authenticated user's profile is used. If the user has not set her timezone, the timezone is set to UTC.
- **Time:** Queries data for a specific time interval. An absolute time interval or a relative one (sliding window) can be defined. Optionally, the time granularity i.e. hour, day, week, month or year, can be declared that further partitions the time interval in multiple intervals. The Data API returns results for every of these time intervals.
- **Population:** Specifies one or more groups of users to query. For every user group, a new data series of aggregated data is returned. A query may request data for all the users of a utility, the users of a cluster, the users of an existing group, a set of specific users or just a single user.
  - Clusters are expanded to segments before executing the query. A segment is equivalent to a group of users. Thus, declaring a cluster is equivalent to declaring all of its groups.
  - Optionally, the users of a group may be ranked based on a metric.
- **Spatial:** A query may optionally declare one or more spatial constraints and filters. A spatial constraint aggregates data only for users whose location satisfies the spatial constraint e.g. it is inside a specific area. On the contrary, a spatial filter is similar to the population parameter and creates a group of users based on their location; hence a new data series is returned for every spatial filter.
- **Metric:** The metrics returned by the query. Data API supports min, max, sum, count and average aggregate operations. Not all data sources support all metrics. When forecasting data is requested, this parameter is ignored and sum is the only value returned.
- **Source:** Declares the data source to use. When forecasting data is requested, this parameter is ignored.

Detailed documentation on the Data API syntax and request examples, for both the Action and the HTTP API endpoints, can be found at:

- <https://app.dev.daiad.eu/docs/api/index.html>.

The Data API is implemented as part of the DAIAD Services presented in the Report for Prototype Deliverable D1.4 'Final DAIAD system'. Figure 3 illustrates the Data API implementation in more detail.

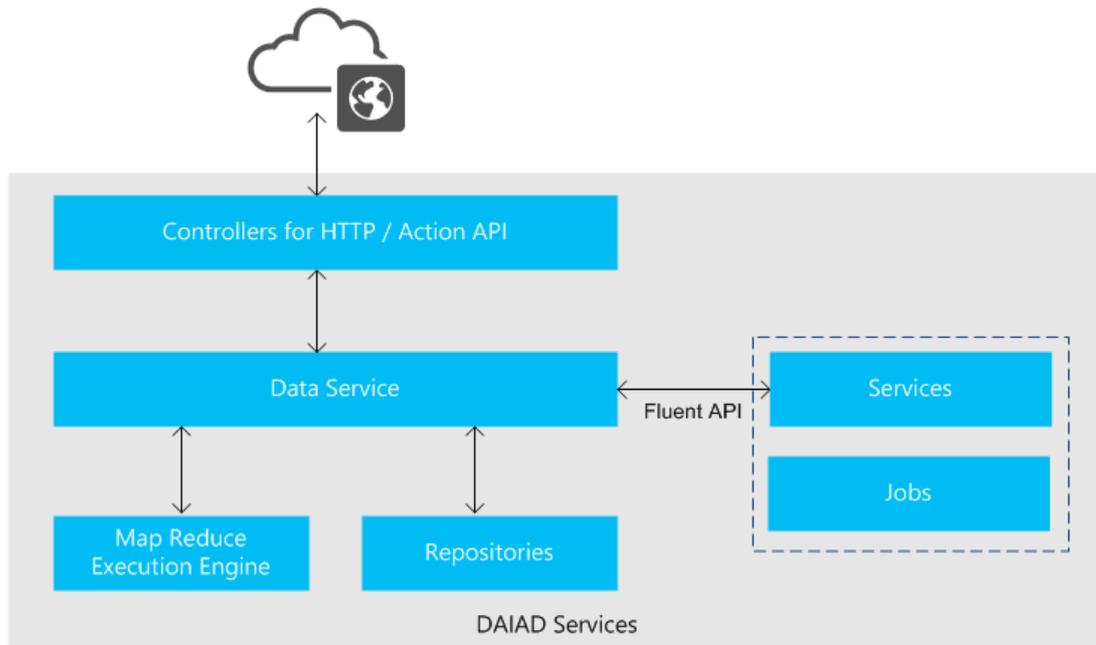


Figure 3: Data API implementation

Query requests are received by the DAIAD Services controller components and forwarded to the Data Service. The Data Service orchestrates data query execution. It accesses data from several repositories such as user profile information and smart water meter registration data and expands the query before execution. Query expansion refers to the process that selects all individual users and their corresponding devices for all groups to query. In addition, any spatial constraints are applied at this stage. The expanded query is submitted to the Map Reduce execution engine for computing the query results.

In addition to the HTTP endpoints, the Data API also provides a fluent API for building queries at the server side. This feature is used by other services and jobs for querying water consumption data. Two distinctive examples are the Message Service<sup>10</sup> and the User Clustering Job<sup>11</sup> respectively. The former queries utility and individual user consumption data in order to generate alerts and recommendations. The latter clusters the users based on their total water consumption over a predefined time interval.

### 1.3.2. Configuration API

The client application, once initialized, will attempt to load configuration fragments from the server. Because the client is a single page application (SPA), this configuration step will only happen once and only when

<sup>10</sup> <https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/service/message/DefaultMessageService.java>

<sup>11</sup> <https://github.com/DAIAD/home-web/blob/master/src/main/java/eu/daiad/web/jobs/ConsumptionClusterJobBuilder.java>

needed (*i.e., in a lazy manner*). Loading the configuration is not different than setting something in global state, so it is actually performed using the known mechanism of Redux actions.

In order to fetch the configuration, the client sends requests to several Configuration API endpoints which are also part of the broader Action API. The basic endpoints contacted during a configuration action are capable of the following:

- Fetch all accessible utilities.
- Fetch all groups inside a utility.
- Fetch all segments of a cluster inside a utility.
- Fetch data for an individual consumer.
- Fetch spatial data about the administrative boundaries of a utility.

## 2. UI Elements

In this section, we present the core UI elements comprising the Knowledge Discovery Workbench. As analyzed in the previous section, they consist the essential building blocks for the interfaces of DAIAD@commons and DAIAD@utility applications.

### 2.1. LineChart

The LineChart UI element is a general-purpose UI control that represents a line (line/area) chart. It visualizes series of data points on a 2-dimensional Cartesian grid, while supporting a wide set of options and customizations. The source code is available at:

- <https://github.com/DAIAD/react-echarts/blob/master/src/js/components/line.js>

The LineChart UI element is a class that implements a React component and wraps an Echarts chart, thus making a subset of Echarts functionality available to React-based applications. It is designed to be a presentational-only component, so it assumes nothing about the origin of data, nor it attempts to shape them in any way. The capabilities of this element are:

- Draws data points and corresponding line/area segments on the grid. If requested so, it draws instead a best-fit curve for input points (based on spline interpolation).
- Draws axes, grid lines, and grid zones.
- Places axis ticks and generates customizable labels on those ticks.
- Provides customizable tooltips on data points or marker lines.
- Provides a customizable legend for input series under display. The legend allows a subset of selected series to be switched off in order to focus our interest on the rest.

The LineChart UI element has been implemented as follows:

- We have developed a class that wraps the opaque handle provided by Echarts library and acts as a React portal component. A React portal component excludes itself from React's change-render lifecycle and instead manages updates in a completely self-governed manner. In order to do so, it prevents normal re-rendering triggered by React, intercepts all "change" events and properly maps them to calls on the underlying Echarts handle.
- We have developed a properties interface that acts as a facade to a common subset of Echarts functionality. A basic duty of our class is to translate received properties to a consistent set of Echarts-specific options. On every "change" event, we re-compute and validate all options and then re-configure the underlying Echarts handle. Because a minimal set of properties is actually required by

our interface, we reside on multiple levels (theme-level, class-level) of fallback defaults in order to allow external themes or derived classes to provide charts with "pre-packaged" appearance.

- We have mapped the React component lifecycle events to object lifecycle events (*initialize*, *reset*, *cleanup*) of the underlying handle. This is required because React demands that all components follow certain lifecycle steps while mounting or unmounting to a target document.

The LineChart UI element receives properties which are categorized and described in the tables below:

- Properties for the portal element and the placement of the chart inside the parent document (Table 1)
- Properties for the appearance of the actual chart (Table 2).
- Properties for input series that feed the chart (Table 3).
- Properties for the status of plotted data (Table 4).

*Table 1: Properties for placement of the chart inside the parent document*

Name	Type	Description	Example
<b>width*</b>	Number, String	The width (pixels) of the container element.	250, 40%
<b>height*</b>	Number, String	The height (pixels) of the container element.	250, 40%

*Table 2: Properties related to chart's appearance*

Name	Type	Description	Example
<b>xAxis.data</b>	array	An array of distinct (aka category) values that x can take	['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
<b>xAxis.formatter</b>	(x)=>(String)	Formatter callback for x values	(x)=>(x.toString() + 'Km')
<b>xAxis.labelFilter</b>	(i, x) => (<Boolean>)	Decide if a label should be shown on the X axis.	(i)=>(i%2 == 0)
<b>xAxis.numTicks</b>	Number	A hint for the number of ticks on X axis	5
<b>xAxis.boundaryGap</b>	Boolean	Add a gap between min/max x value and axis boundaries	false

<b>xAxis.min</b>	Number	A maximum for displayed x values (meaningless if xAxis.data is supplied)	0
<b>xAxis.max</b>	Number	A minimum for displayed x values (meaningless if xAxis.data is supplied)	5.0
<b>yAxis.formatter</b>	(y)=>(String)	Formatter callback for y values	(y)=>(y.toString() + 'lt')
<b>yAxis.numTicks</b>	Number	A hint for the number of ticks on y axis	5
<b>yAxis.min</b>	Number	A maximum for displayed y values	-100
<b>yAxis.max</b>	Number	A minimum for displayed y values	+100
<b>grid.x</b>	String or Number	See <a href="#">ECharts - grid.x</a>	15%
<b>grid.y</b>	String or Number	See <a href="#">ECharts - grid.y</a>	10%
<b>grid.x2</b>	String or Number	See <a href="#">ECharts - grid.x2</a>	15%
<b>grid.y2</b>	String or Number	See <a href="#">ECharts - grid.y2</a>	10%
<b>color</b>	Array of String	A palette of preferred colors	['#C23531', '#2F4554']
<b>tooltip</b>	Boolean	Display tooltips for data points or marker points/lines	true
<b>smooth</b>	Boolean	Smoothen lines for all series (spline interpolation)	false
<b>lineWidth</b>	Number	The width (pixels) of all plotted lines	2
<b>legend</b>	Boolean or Array	Display legend. If an array is supplied, then it can control the order and layout of items (a nested array generates a legend that wraps over multiple lines)	true, [['A', 'B'], ['C', 'D']], ['A', 'C', 'D', 'B']

*Table 3: Properties for input series. The series property is an array that provides the actual data to be plotted. Each item inside series, carries the above (nested) properties.*

Name	Type	Description	Example
series.0.name*	String	The name of this dataset	Temperature - Athens
series.0.data*	Array	The actual array of data points. If xAxis.data is present (categorical data), then we expect an array of values mapping 1-1 to xaxis values. Else, we expect an arbitrary array of numerical (x,y) points.	[11.0, 11.5, 13, 14, 13, 15, 17]
series.0.color	String	The color for this line/area	'#C23531'
series.0.smooth	Boolean	Smoothen line for this series (perform spline interpolation)	false
series.0.fill	Number	Fill areas with the given opacity	null or 0.55
series.0.symbolSize	Number	Radius for symbols for (x,y) points	4
series.0.symbol	String	Choose a symbol for (x,y) points. One of: circle, rectangle, triangle, diamond, emptyCircle, emptyRectangle, emptyTriangle, emptyDiamond	emptyCircle
series.0.lineWidth	Number	The width (pixels) for this line	false
series.0.mark.points	Object	Describe marker points	[{type: "max", name: "Max Temperature"}]
series.0.mark.lines	Object	Describe marker lines	[{type: "min", name: "Min Temperature"}]

*Table 4: Properties for the status of plotted data*

Name	Type	Description	Example
loading	Object or Boolean	Provide a visual feedback on progress (spinner, progressbar)	{text: "Loading data...", progress: 0.7}

In the following, we provide a simple example of how the LineChart UI element can be invoked. We plot (fictional) temperatures for 3 cities over a period of 1 week. In this particular example, we supply our LineChart with categorical data, i.e. all X values are drawn from a set of predefined distinct values (days of a week).

```
<LineChart
  width='500px'
  height='300px'
  legend={[
    ['Athens', 'Thesalloniki'], ['Herakleion'],
  ]}
  xAxis={{
    data: ['Mo', 'Tu', 'We', 'Th', 'Fr', 'Sa', 'Su'],
  }}
  yAxis={{
    name: "Temperature",
    numTicks: 3,
    formatter: (y) => (y.toString() + " oC")
  }}
  series={[
    {
      name: 'Athens',
      smooth: true,
      fill: 0.4,
      data: [11.0, 11.5, 13, 14, 13, 15, 17],
      mark: {
        lines: [{type: "max", name: "Max Temperature"}],
      },
    },
    {
      name: 'Thesalloniki',
      data: [5.0, 8.5, 13.5, 14.7, 16, 19, 21.5],
    },
    {
      name: 'Herakleion',
      data: [15.0, 18.5, 19.5, 24.7, 26, 29, 31.5],
    },
  ]}
/>
```

The above example will produce the chart shown at Figure 4:

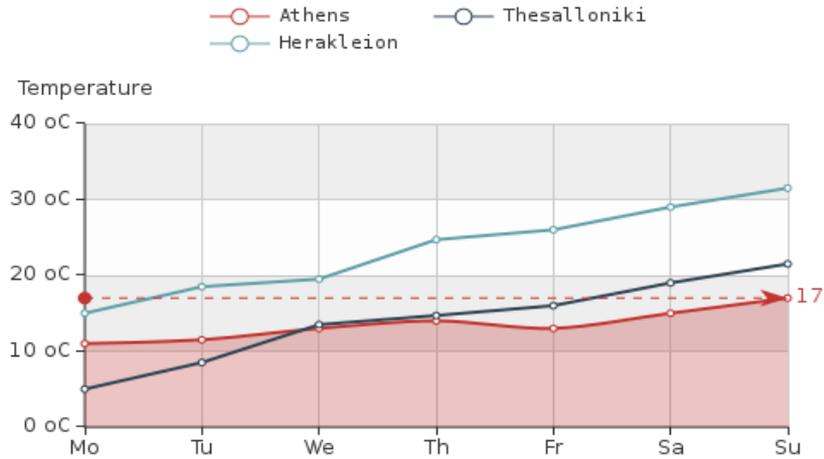


Figure 4: Result of example invocation

The following figures present some indicative uses of the LineChart UI element inside DAIAD's web applications.

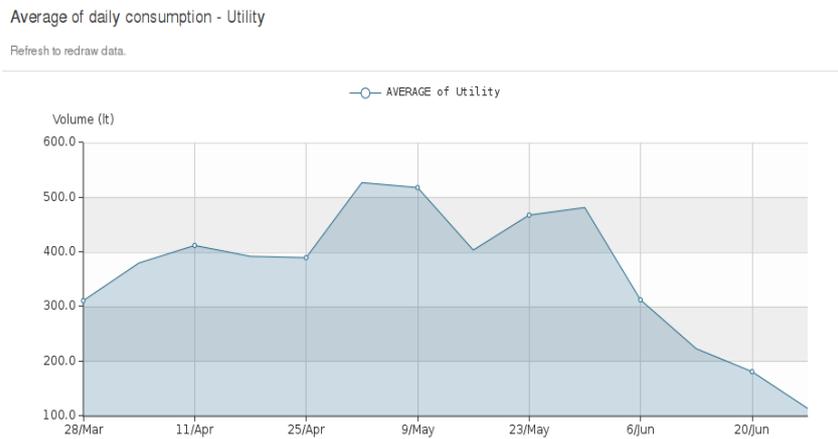


Figure 5: Average of daily consumption chart

Average of weekly consumption - Meter - Cluster: Household Members

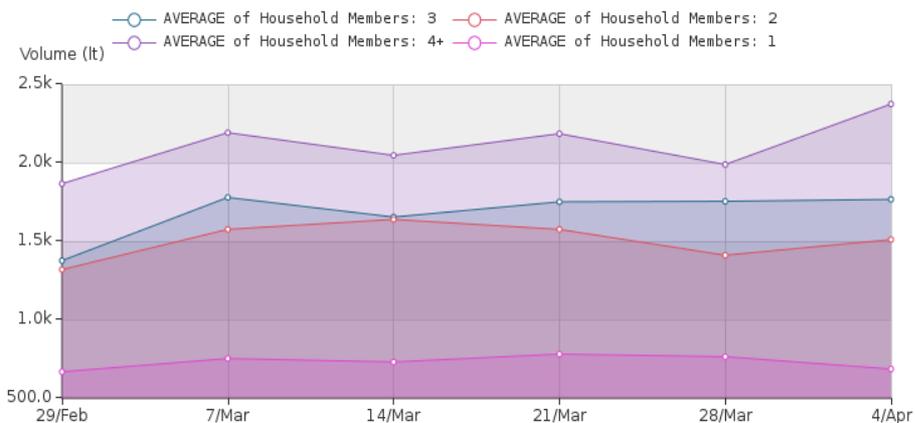


Figure 6: "Average of weekly consumption" for consumer groups of the "Household Members" cluster

## 2.2. Chart

The Chart UI element is a specialization of the LineChart element that focuses on the visualization of time-series data. The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/chart.js>

Specifically, it is a class that implements a presentational-only React component. It is built entirely on the LineChart component by making some assumptions on the nature of input data and adding data-shaping logic and defaults. More precisely:

- Narrows its input to time-series data, expected as series of (t, y) pairs. The time part is always expected as an Epoch timestamp, not necessarily at even steps.
- Assumes that represented time-varying entities are linked to metadata (e.g. unit of measurement) accessible via global client-side configuration.
- Is capable to perform data shaping in order to provide a requested level of detail. It does so by grouping data points into evenly-sized time buckets (of the target level) and then applying an aggregate function on them.
- Adds another layer of class-level defaults related to the time nature of the X axis.
- Expresses the time-axis in a meaningful manner in the case of overlapping mode, with possible different timespan inputs for each time-series.

The Chart UI element receives properties described in the table below. All properties marked as enumeration may take a value from a set of constants globally available as part of client-configuration.

*Table 5: Properties for Chart UI element*

Name	Type	Description	Example
<b>width</b>	String, Number	The width of LineChart element	400
<b>height</b>	String, Number	The height of LineChart element	400
<b>field</b>	String	The name of the physical entity under consideration (enumeration).	volume
<b>level</b>	String	The desired level of detail (enumeration).	Week, day
<b>reportName</b>	String	The name of the current report (enumeration).	avg
<b>series</b>	Array	Input series of data points.	[{  source: "volume",

			data: [ [t1, y1], [t2, y2] ] }]
<b>[overlapping]</b>	Boolean	Defines explicitly that the chart should handle the series as overlapping, with possibly different timespans.	false
<b>[overlap.label]</b>	String	The label of the time overlap.	Year
<b>[overlap.value]</b>	String	The value of the time overlap.	year
<b>[finished]</b>	Boolean, Number	Are series data considered as finished. If this is false, means that there is a ongoing request. If is a number represents the timestamp of last successful request.	1467716293
<b>[draw]</b>	Boolean	Allow parental control to redrawing. Used to explicitly prevent component updates.	true

In the following, we provide an example (expressed as JSX) of how the Chart UI element can be invoked:

```
<Chart
  draw={true}
  field="volume"
  level="week"
  reportName="avg"
  series={[
    source: "volume",
    data: [[t1, y1], [t2, y2], ... [tN, yN]]
  ]}
/>
```

The above example will produce the chart shown at Figure 7.

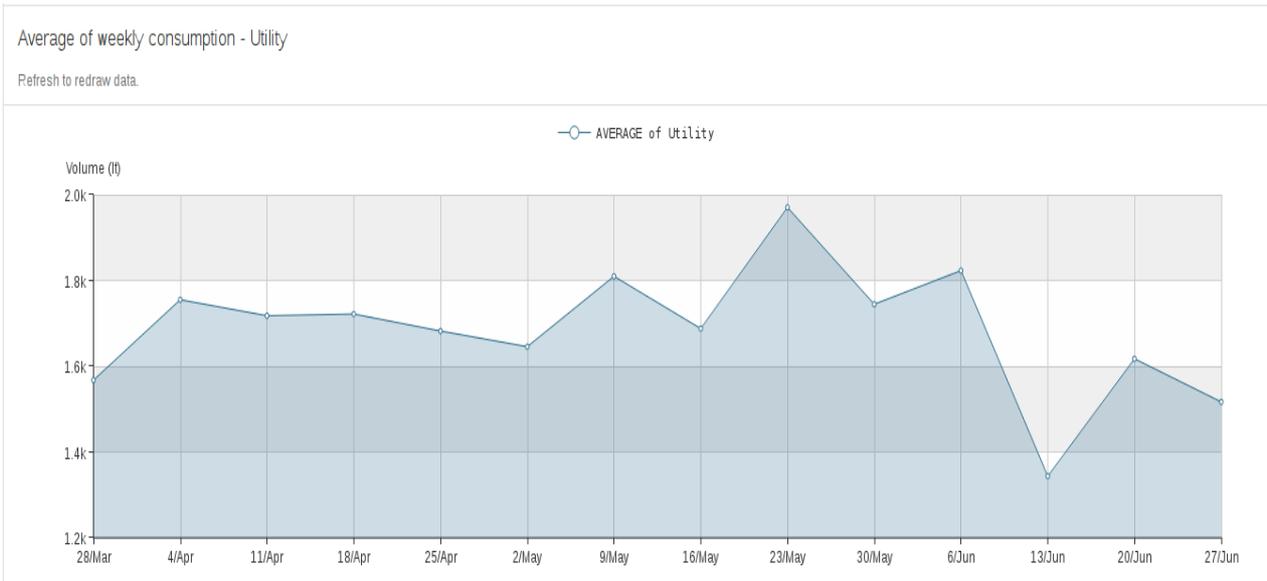


Figure 7: An example invocation of Chart UI element producing an "Average of Weekly Consumption" chart at DAIAD@utility application

## 2.3. ChartContainer

The ChartContainer UI element is a container of the Chart element capable to request and receive measurement data from DAIAD's Data API. The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/chart-container.js>

The ChartContainer UI element is a class that implements a container React component. It acts as a transparent proxy to an enclosed Chart component adding data-acquisition methods. Since we are using Redux to manage global client-side state, these methods just emit actions that Redux store will dispatch to the appropriate reducer. Briefly, these actions will result to the following:

- Request measurement data on the given Data API.
- Receive data from Data API and shape them according to way global state expects them. Handle possible failure in any phase of the request cycle.
- Feed the overall status of the request to the contained component.
- Select and feed (successfully) received data to the contained component.

The ChartContainer UI element has been implemented as a typical Redux container around Chart component. We have provided the two needed mapping functions to map global state and dispatch functions to component properties.

The ChartContainer UI element receives exactly the same properties as Chart element. As explained above, the only difference is that in ChartContainer some properties (series) are not passed explicitly from the owner component but instead are "silently" injected from Redux mappers.

## 2.4. Leaflet Wrapper

The new general-purpose Leaflet Wrapper UI element replaced the previously used LeafletMap component (see D5.3.1). The Leaflet Wrapper element is available in the form of a separate library, providing full control over the appearance of a map element using the Leaflet<sup>12</sup> JavaScript mapping library, and is fully integrated with the React library. It supports building custom maps by defining separate layers. The code for the Leaflet Wrapper UI element, as well as full documentation and examples are available at:

- <https://github.com/DAIAD/react-leaflet-wrapper>

The Leaflet Wrapper UI element supports several layer types including Tile, Marker, GeoJSON, Choropleth and Heat layers. Also, several controls including Layers Control, Info Control and Draw Control are supported and can be added to a map. The layers control lists the contained layers and allows the user to switch them off and on. The info control supports displaying layer-related or custom information to the user. Finally, the drawing tool is provided for creating, editing and deleting rectangular or polygon geometries, while only allowing a single drawn geometry to exist at any given time on the map. Finally, a map can contain an arbitrary combination of layers and controls.

The Leaflet Wrapper UI element has been implemented as follows:

- We have developed a reusable presentational map component in the form of a separate library, as a wrapper to the Leaflet JavaScript library. It is fully integrated with React and allows customizing maps using JSX syntax. It provides three separate component types, the Map, Layer and Control types whose specific properties are listed below.
- Polygon drawing is implemented using the Leaflet.draw<sup>13</sup> plugin. This plugin supports drawing points, lines and polygons. The current version of the component exposes only the functionality for drawing polygons.
- Heat maps are implemented using the Leaflet.heat<sup>14</sup> plugin. This is an experimental feature and is currently under evaluation. There are a few known issues related to layer ordering when a heat map layer is combined with GeoJSON layers.

The configuration options for each of the three component types of the Leaflet Wrapper UI element are documented in the tables below. Layer and Control type options differ as they expose specific Leaflet layer or control properties. Below we provide the configuration options for some common layers, such as Tile and

---

<sup>12</sup> <http://leafletjs.com/>

<sup>13</sup> <https://github.com/Leaflet/Leaflet.draw>

<sup>14</sup> <https://github.com/Leaflet/Leaflet.heat>

GeoJSON, as well as Controls, such as Layers Control and Draw Control. Optional parameters have their names in square brackets:

*Table 6: Map configuration options*

Name	Type	Description	Example
<b>width</b>	Number, String	The width of the container	100, 100%
<b>height</b>	Number, String	The height of the container	200, 50%
<b>[center]</b>	Array	Map center coordinates expressed in WGS84 <sup>15</sup> Coordinates Reference System (CRS)	[38.36, 0.47]
<b>[zoom]</b>	Number	Map zoom level	13

*Table 7: Tile Layer configuration options*

Name	Type	Description	Example
<b>id</b>	String	An identification string	baselayer
<b>url</b>	String	The tile layer url	http://{s}.tile.osm.org/{z}/{x}/{y}.png
<b>attribution</b>	String	The attribution that is displayed usually on the bottom-right of the map	<a href= <a href="http://osm.org/copyright">http://osm.org/copyright</a> >OpenStreetMap</a> contributors

*Table 8: GeoJSON options*

Name	Type	Description	Example
<b>data</b>	Object	GeoJSON feature collection	{ "type": "Feature", "properties": { "popupContent": "Popup" }, "geometry": { "type": "Point", "coordinates": [-104, 39]

<sup>15</sup> <https://epsg.io/4326>

			}}
<b>style</b>	Function	A function for styling GeoJSON, lines and polygons	feature => ({ weight: 2, color: "#999", opacity: 1, fillColor: "#B0DE5C", fillOpacity: 0.8 })
<b>popupContent</b>	Function	Callback function for rendering the content of the feature popup. The function accepts the feature and layer as arguments and supports JSX syntax with events.	(feature, layer) => ( <div onClick={e => { alert('clicked')}} > <span>Clickable vector</span> </div> )
<b>filter</b>	Function	Callback function for filtering some of the layer features	(feature, layer) => feature.properties && feature.properties.visible
<b>circleMarkers</b>	Boolean	A Boolean to render GeoJSON point features as circle markers instead of the default marker icon	true
<b>markerStyle</b>	Object	An object to style markers if the circleMarkers option is set	{ weight: 2, color: "#999", opacity: 1, fillColor: "#B0DE5C", fillOpacity: 0.8 }
<b>markerOptions</b>	Object	An object to fully customize the appearance of GeoJSON points markers	{ icon: L.Icon({ iconUrl: 'dist/baseball- marker.png', iconSize: [32, 37], }) }

*Table 9: Layers Control options*

Name	Type	Description	Example
<b>position</b>	String	The position on the map where the control is rendered	topright

Table 10: Draw Control options

Name	Type	Description	Example
<b>controlled</b>	Boolean	A Boolean that sets whether the drawn layer is controlled, meaning it can be changed externally. In that case the geometry needs to be provided in the data property	true
<b>data</b>	Object	GeoJSON feature collection in case the layer is controlled	{ "type": "Feature", "properties": {  "popupContent": "Popup" }, "geometry": { "type": "Point", "coordinates": [-104, 39] } }
<b>style</b>	Object	An object with styling options for the drawn GeoJSON layer	{  weight: 2, color: "#999", opacity: 1, fillColor: "#B0DE5C", fillOpacity: 0.8 }
<b>onFeatureChange</b>	Function	A callback function that is invoked every time the drawn layer is changed	features => <a href="#">this.setState</a> ({ geometry: features }

In the following, we provide an example of how the Leaflet Wrapper UI element can be invoked. In this particular example, we create a map centered on the city of Denver with a series of GeoJSON layers.

```
<Map
  width='100%'
  height={400}
  center={[39.74739, -105]}
  zoom={13}
  >
  <TileLayer
url='https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token=XXX'
  attribution='Map data &copy; <a
href="http://openstreetmap.org">OpenStreetMap</a> contributors, <a
href="http://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, Imagery © <a
href="http://mapbox.com">Mapbox</a>'
```



```

    id='mapbox.light'
  />
  <GeoJSON
    data={[GEOJSON.bicycleRental, GEOJSON.campus]}
    style={feature => feature.properties && feature.properties.style}
    popupContent={(feature, layer) => (
      <span>
        <p>
          {`I started out as a GeoJSON ${feature.geometry.type}, but now I'm a
Leaflet vector!`}
        </p>
        <p>
          { feature.properties.popupContent }
        </p>
      </span>
    )}
    circleMarkers
    markerStyle={{
      radius: 8,
      fillColor: "#ff7800",
      color: "#000",
      weight: 1,
      opacity: 1,
      fillOpacity: 0.8
    }}
  />
  <GeoJSON
    data={GEOJSON.freeBus}
    popupContent={popupContent}
    filter={(feature, layer) =>
      feature.properties && feature.properties.underConstruction != null ?
!feature.properties.underConstruction : false
    }
  />
  <GeoJSON
    data={GEOJSON.coorsField}
    popupContent={popupContent}
    markerOptions={{
      icon: L.icon({
        iconUrl: 'dist/baseball-marker.png',
        iconSize: [32, 37],
        iconAnchor: [16, 37],
        popupAnchor: [0, -28]
      })
    }}
  />
</Map>

```



Figure 8: Example of Leaflet Wrapper component usage

In the following figures, we provide two representative examples of how the Leaflet Wrapper UI element is used in the context of the DAIAD@utility web application.

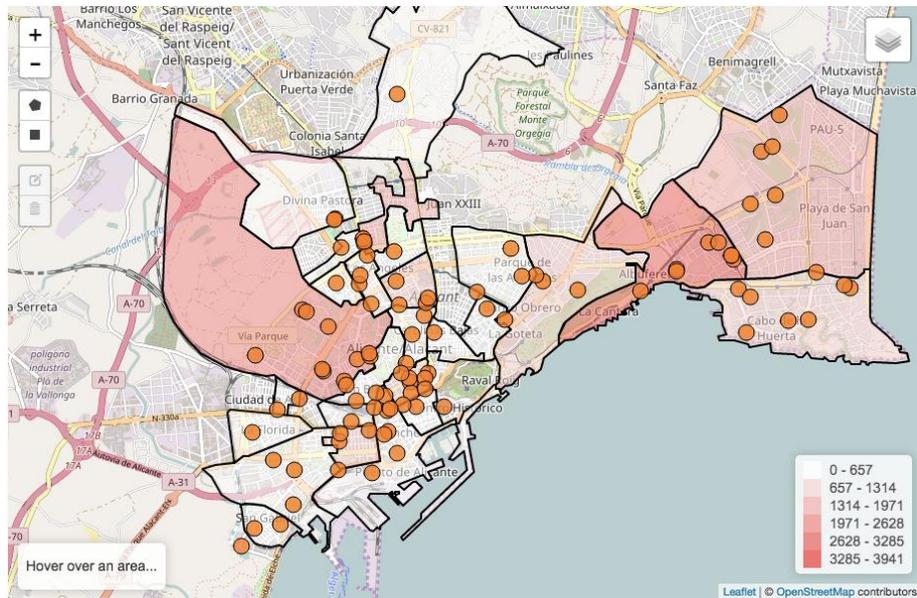


Figure 9: Choropleth of smart water meter consumption data combined with a GeoJSON layer of the meters' locations

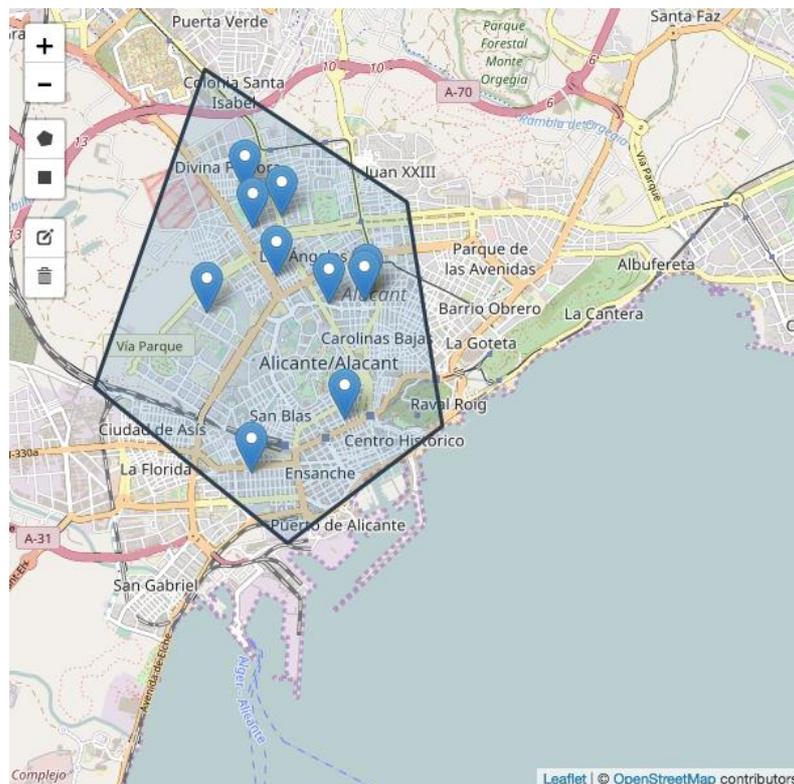


Figure 10: Combination of a GeoJSON layer of user meters' locations and a draw control with the corresponding drawn polygon

## 2.5. Table

The Table UI element is a component for displaying data in tabular format. The component can be used with minimal configuration and requires only the definition of table column names and row data. Row data is expressed as an array of JSON objects and is application specific. Column names correspond to row object properties. The component does not support nested properties for rows. The code for Table component is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/Table.js>

The Table UI element implements the following features:

- Data paging: The table element supports both client-side and server-side data paging. Server-side data paging is application specific and must be managed by the application using the Table element. The application is responsible for initializing the data.pager configuration options properly.
- Data formatting: The Table element uses the field.type configuration option for deciding which format to use for a cell value. If no type is defined, the value is treated as plain text. The component supports custom formatting for boolean, data and time values.
- Dynamic cell rendering: The configuration options for fields allow the dynamic computation of values at runtime using callback functions. Users can dynamically set the color of text, the icon of actions, the visibility of actions and the class of cell values using callbacks.
- Actions: Table UI element allows user interaction through the **action** field type. Actions are rendered as icons or images and can invoke a callback when clicked.

The Table UI element is a composite element. The components used for building a Table are enumerated below. A user can build a table explicitly by using these components but this syntax is a lot more verbose since properties for every component must be set manually.

- Table: This is the top-level component that configures the table layout. It renders the Bootstrap pagination component if data.pager property is defined and renders the template.empty component if no data is found.
- Header: Renders the table header row and sets column visibility based on the data.field configuration options
- Body: Renders the table rows. Before rendering the Body component filters table rows depending on the pagination options. If pagination is done locally, the Body component selects the rows of the currently selected page index to display.
- Row: Renders the cells of a row and sets the column visibility based on the data.field configuration options.
- Cell: Renders the value for a single row object property. The rendering process is driven by the configuration options of the corresponding data.field object. Optionally, if callback functions are used, values of other row object properties effect the rendering process.

The configuration options for the Table UI element are documented in the tables below. Optional parameters have their names in square brackets:

*Table 11: Table configuration options*

Name	Type	Description	Example
fields	Array	The table schema	
data	Array	Table data rows array containing objects reflecting the provided schema option	
pager	Object	The pagination object	
[onPageIndexChange]	function	Callback function invoked when the current data page index changes. The new data page index is passed as an argument	
[template]	Object	Configuration options for the table default views	
[style]	Object	Table elements custom styles	

*Table 12: Field options*

Name	Type	Description	Example
name	String	Field name. Each row Object is expected to have a property with this name	
title	String	Column header	
[hidden]	Boolean	True if the column is visible; Otherwise False, Default value is True	
[type]	String	Field data type. Default value is undefined and the cell value is treated as text. Valid values are:  <b>action</b> : Renders a button using an icon <b>datetime</b> : Formats the value of the cell as a date time value <b>date</b> : Formats the value of the cell as a date value <b>time</b> : Formats the value of the cell as a time value <b>progress</b> : Renders a progress bar. The value is expected to be a number in the interval 0 to 100 inclusive. <b>boolean</b> : A read-only checkbox. The value is expected to be a boolean <b>alterable-boolean</b> : A writable checkbox. The value is expected to be a boolean	
[icon]	String, Function	Applicable only to fields of type <b>action</b> . The icon to display for the action.	clock-o



		If a string is specified, it is treated as a Font Awesome <sup>16</sup> class name. If a function is specified, the class name is the function call return value. The function accepts two arguments. The field configuration object and the row object the value belongs to.	
[image]	String	Applicable only to fields of type <b>action</b> . The image to display for the action. This property is overridden by the icon property.	
[color]	String, Function	If a string is specified, it is treated as a color value. If a function is specified, the color value is the function call return value. The function accepts two arguments. The field configuration object and the row object the value belongs to.	#9E9E9E
[handler]	Function	Applicable only to fields of type <b>action</b> . The callback function invoked when the action is executed. The function accepts two arguments. The field configuration object and the row object the value belongs to.	
[visible]	Booelan, Function	Applicable only to fields of type <b>action</b> . Shows or hides the contents of a cell dynamically at runtime. The visibility status is the function call return value. The function accepts two arguments. The field configuration object and the row object the value belongs to.	
[align]	String	Header and cell text alignment. Valid values are left, right, center and justify. Default value is left.	
[width]	Number	Column width in pixels	
[link]	String, Function	Renders a link instead of the property value. If a string is specified, it is treated as a link template. Link templates contain place holders which are column names in curly brackets e.g. the {address} placeholder is replaced with the value of the row object address property. If a function is specified, the template is created dynamically before generating the actual link. The function accepts the row object as an argument.	
[className]	Function	Returns a class name to apply to the cell contents. The function accepts the value of the cell as an argument.	

*Table 13: Pager options*

Name	Type	Description	Example
------	------	-------------	---------

<sup>16</sup> <http://fontawesome.io/>

[index]	Number	Current data page index. Default value is 0.	
[size]	Number	Number of rows per data page. Default value is 10.	
[count]	Number	Total number of rows. Default value is 0.	
[mode]	String	Enables data paging at the client or server side. Valid values are:  Table.PAGING_CLIENT_SIDE  Table.PAGING_SERVER_SIDE  Default value is Table.PAGING_CLIENT_SIDE.	
[onPageIndexChange]	function	Callback function invoked when the current data page index changes. The new data page index is passed as an argument	

Table 14: Style configuration options

Name	Type	Description	Example
[row]	Object	Style for cells of a row	{ color: '#9E9E9E' }

In the following, we provide an example of how the Table UI element can be invoked. In this example, we create a simple table with three columns and two rows. For the third column, a callback is defined for setting the cell value color dynamically at runtime.

```
const fields = [{
  name: 'index',
  title: 'Index'
}, {
  name: 'name',
  title: 'Product Name'
}, {
  name: 'value',
  title: 'Price',
  color: function (field, row) {
    if (row.value > 100) {
      return '#FF0000';
    }
    return '#00FF00';
  }
}];

const rows = [
  {index: 1, name: 'Product A', value: 120.0},
  {index: 2, name: 'Product B', value: 45.0}
];

const pager = {
  index: 0,
  size: 10,
  count: 2,
  mode: Table.PAGING_CLIENT_SIDE
};

<Table fields={fields}
```

```
data={ rows }
pager={ pager }
/>
```

Index	Product Name	Price
1	Product A	120
2	Product B	45

« < 1 > »

Figure 11: Result of Table code example

In the following figures, we provide two representative examples of how the Table UI element is used in the context of the DAIAD@utility web application.

Name	Started On	Completed On	Status Code	Exit Code
AMAEM-SFTP	06/07/2016, 12:00	06/07/2016, 12:00	COMPLETED	COMPLETED
AMAEM-SFTP	06/07/2016, 08:00	06/07/2016, 08:00	COMPLETED	COMPLETED
AMAEM-SFTP	06/07/2016, 04:00	06/07/2016, 04:00	COMPLETED	COMPLETED
DAILY-STATS-COLLECTION	06/07/2016, 04:00	06/07/2016, 04:00	COMPLETED	COMPLETED
AMAEM-SFTP	06/07/2016, 00:00	06/07/2016, 00:00	COMPLETED	COMPLETED
AMAEM-SFTP	05/07/2016, 20:00	05/07/2016, 20:00	COMPLETED	COMPLETED
AMAEM-SFTP	05/07/2016, 16:00	05/07/2016, 16:00	COMPLETED	COMPLETED
AMAEM-SFTP	05/07/2016, 12:00	05/07/2016, 12:00	COMPLETED	COMPLETED
AMAEM-SFTP	05/07/2016, 08:00	05/07/2016, 08:00	COMPLETED	COMPLETED
AMAEM-SFTP	05/07/2016, 04:00	05/07/2016, 04:00	COMPLETED	COMPLETED

Figure 12: Job execution history

Type	Name	# of members	Updated On
SEGMENT	Age: 18 - 24	2	16/06/2016, 01:02
SEGMENT	Age: 25 - 34	15	16/06/2016, 01:02
SEGMENT	Age: 35 - 44	27	16/06/2016, 01:02
SEGMENT	Age: 45 - 54	25	16/06/2016, 01:02
SEGMENT	Age: 55 - 64	10	16/06/2016, 01:02
SEGMENT	Age: 65 - 74	2	16/06/2016, 01:02
SEGMENT	Apartment Size: 31-60	0	16/06/2016, 01:02
SEGMENT	Apartment Size: 61-80	11	16/06/2016, 01:02
SEGMENT	Apartment Size: 81-110	48	16/06/2016, 01:02
SEGMENT	Apartment Size: > 111	22	16/06/2016, 01:02

« < 1 2 3 > »

Figure 13: List of cluster segments

## 2.6. Toolbar

The Toolbar UI element is a general-purpose control that builds a button toolbar as a group of button groups. A common set of options and customizations (icons, text, tooltips) is supported. The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/toolbars.js>

The Toolbar UI element is a class that implements a presentational-only React component. It receives (as a property) a plain JSON specification that describes each button in a button group, and according to that specification it builds a toolbar element. Finally, it also receives a callback property to be invoked with the particular key of a button that was just clicked.

The Toolbar UI element has been implemented as follows:

- We have developed a React component that receives a toolbar specification and renders a toolbar based on React-Bootstrap buttons and button groups. This specification should assign a unique key to each button and should describe the appearance (icons, text, CSS classes) and its status (enabled, activated).
- We attach to all (enabled) buttons an event handler that wraps and calls the user-supplied (property) callback function using group's key and button's key as the first 2 parameters.
- We use the FontAwesome icon set (<http://fontawesome.io/icons/>) to improve a button's appearance and/or give a visual clue of its function. If an icon is to be used, the specification must refer to it by its name in FontAwesome set.

The Toolbar UI element receives properties described in the tables below.

*Table 15: Properties for Toolbar UI element*

Name	Type	Description	Example
<b>groups*</b>	Array	The spec that describes the toolbar	[[<group-spec>], ...]
<b>groups.0.key*</b>	String	A unique key for a group of buttons	actions
<b>groups.0.buttons*</b>	Array	An array of button specs	[[<button-spec>], ...]
<b>onSelect*</b>	(groupkey, key) => ()	A callback to be invoked when a button is clicked.	(groupKey, key) => (alert(key))
<b>className</b>	String	An additional CSS class for the toolbar HTML element	toolbar-a

Table 16: Specification for a button inside Toolbar UI Element

Name	Type	Description	Example
key*	String	A unique key for a button	save
text	String	Text for a button	Save
tooltip.message	String	A message for a button tooltip	Save everything
tooltip.placement	String	A hint for the placement of the tooltip. One of: bottom, top, left, right	bottom
iconName	String	A name for a button icon. Must refer to FontAwesome icon set.	save
buttonProps	Object	A set of properties to be forwarded to Bootstrap button	{disabled: false}

In the following example (expressed as JSX) we create a toolbar that consists of 2 button groups.

```
var toolbarSpec = [
  {
    key: 'parameters',
    buttons: [
      {
        key: 'source',
        tooltip: {message: 'Select source of measurements', placement: 'bottom'},
        iconName: 'cube',
      },
      {
        key: 'report',
        tooltip: {message: 'Choose type of report', placement: 'bottom'},
        iconName: 'area-chart',
      },
      {
        key: 'timespan',
        tooltip: {message: 'Define a time range', placement: 'bottom'},
        iconName: 'calendar',
      },
      {
        key: 'population-group',
        tooltip: {message: 'Define a population target', placement: 'bottom'},
        iconName: 'users',
      },
    ],
  },
  {
    key: 'actions',
    buttons: [
      {
        key: 'export',
        tooltip: {message: 'Export to a CSV table', placement: 'bottom'},
        text: 'Export',
        iconName: 'table',
        buttonProps: {disabled: true},
      },
    ],
  }
]
```

```

        key: 'refresh',
        tooltip: {message: 'Re-generate report and redraw the chart', placement:
'bottom'},
        text: 'Refresh',
        iconName: 'refresh',
        buttonProps: {bsStyle: 'primary' },
      },
    ],
  },
];
<Toolbar
  groups={toolbarSpec}
  onSelect={(groupKey, key) => (console.log(key))}
/>

```

The above example will build a toolbar as the one shown in Figure 14.

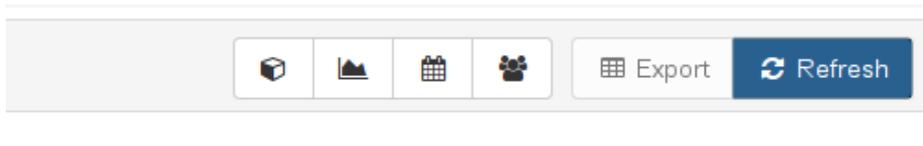


Figure 14: A toolbar built from a Toolbar UI element

## 2.7. ReportPanel

The ReportPanel UI element is a control that facilitates the generation of custom reports on measurement data. It collects common report-related parameters via user-friendly graphical interaction, validates and sanitizes input parameters, and finally generates an on-demand report. The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/pane.js>

The ReportPanel UI element is class that implements a React component. It carries both presentational and container logic and is composed of several other React components:

- A Toolbar component that navigates the user through several form fragments and provides a set of actions on the report itself (e.g. refresh, export).
- A Form component that collects and validates user input for various report parameters (e.g. time interval, source of measurements etc.)
- A ChartContainer component that holds the chart generated by the last user-supplied active set of parameters.
- An Info component that displays helpful messages on the overall status (e.g. last successful update).

The ReportPanel UI element has been implemented as follows:

- We have developed a class for a presentational React component that renders a React-Bootstrap panel. The panel header contains a button toolbar (a Toolbar UI element) built from a proper button

specification. The panel body contains several subsections arranged as React-Bootstrap ListGroup items:

- Form section: a form fragment that collects user input relevant to the current group of parameters (e.g. parameters related to the population target). All available report parameters are logically grouped into form fragments, and the user can switch to another group of parameters using the header toolbar.
  - Report title section: a brief description of the currently active report. Essentially, it translates the active set of parameters to a human-friendly textual description (e.g. "Average of weekly consumption").
  - Chart section: a chart generated (on-demand) by the active set of parameters. This section consists of a ChartContainer UI element on which we explicitly control when a re-rendering will take place (so that the canvas is redrawn).
  - Info section: an informational section summarizing the status of the report, implemented as a local component. The status of the current report (as reflected by the global state) is translated to a human-friendly message.
- We have wrapped the presentational part of ReportPanel into a Redux container component, which provides the following facilities:
    - Dispatch methods that emit actions to Redux store. These actions push user-supplied report parameters to the global state (setting them as active), and also trigger requests that fetch measurement data (on top of Data API) for the active report.
    - Properties reflected from global state. These properties include the active report, the active set of parameters for that given report, received measurement data and overall status of the report.

The ReportPanel UI element does not receive any properties directly from its owner component. Instead, all properties are injected from Redux mappers by reflecting a part of global state.

Due to the fact that no properties are explicitly passed to this component, an example invocation (expressed as JSX) would be as simple as:

```
<ReportPanel />
```

The result of the above invocation is shown at Figure 15.

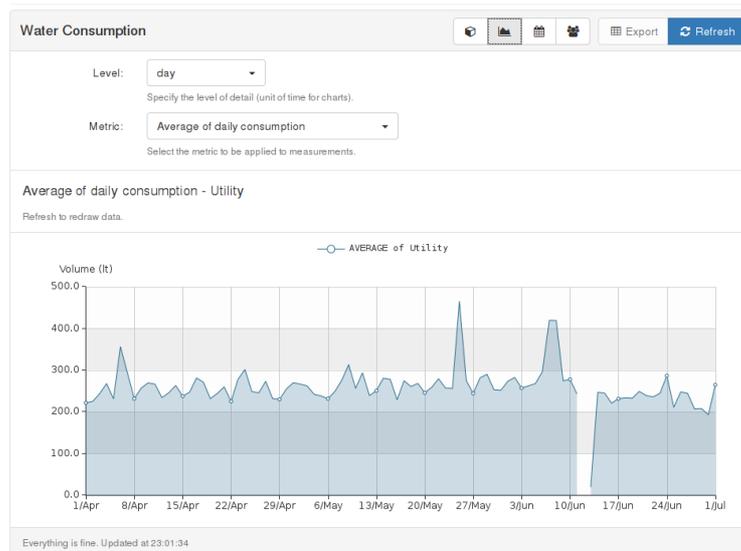


Figure 15: The ReportPanel UI element

## 2.8. UnitReport

The UnitReport UI element is a control that includes and enables several views on measurement data while staying in a level of detail and spanning over a period expressed as a multiple of some given time unit (e.g. week as a unit, 3 weeks as a period). The mainly examined period spans over exactly one time unit and starts at a given reference time (e.g. last week referring to May 12, 2016). The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/unit-reports.js>

The UnitReport UI element is a class that implements a React component. It carries both presentational and container logic, so it is actually a Redux container wrapping a presentational component. For a given level of detail and for a given period, several views share a lot of request and data-shaping logic. Taking this into account, the UnitReport UI element is the container that preprocesses report data, understands a set of views as drop-ins and renders a proper visual representation for each view. UnitReport will scan and will try to recognize its children from a set of predefined views (e.g. a summary view). Once a child is matched against a known view, the respective view is activated and is rendered from the parent component. We must note that this matching mechanism is not interface-based, simply a set of view classes are recognized as valid views. The source code that defines the available views can be found at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/views.js>

The UnitReport UI element has been implemented as follows:

- We have created a set of view classes as pseudo-components. These pseudo-components do not actually render anything, they only serve to enable a certain view inside UnitReport (once found as a child of it) and possibly hold view-specific parameters. For the time being, the following views are recognized:
  - summary: display totals of the main period (e.g. totals for the last day).
  - simple-chart: display a chart over the main period (e.g. last day).
  - comparison-chart: display a chart that compares main period with previous over an interval of one time unit (e.g. compare last day to last 3 days).
- We have developed a class for a React component that will play the role of the UnitReport UI element. Whenever this component receives new properties, it preprocesses report data and holds them as part of the internal state. This preprocessing includes:
  - Sorting of data points by ascending time.
  - Consolidation of data points at the current level of detail.
  - Creating a dense array of data points with even steps (original data points may be sparse).
  - Computing important aggregates (e.g. totals) for the main period.
- We have made the render method of UnitReport aware of the aforementioned views. The component scans all its children and if an instance of known view is encountered then an inner view-specific rendering method is invoked to generate the respective component's sub-tree.
- We have wrapped the presentational part of UnitReport into a Redux container that provides:
  - Dispatch methods that emit actions to Redux store. These actions initialize and refresh measurement data for a certain report.
  - Data series received for a certain report, as a property reflected from global state.

The UnitReport UI Element receives the properties described in the table below:

*Table 17: Properties of UnitReport UI element*

Name	Type	Description	Example
<b>source*</b>	String	The name of the data source. One of: meter, device	meter
<b>field*</b>	String	The name of the measured physical entity. It depends on the source,	volume
<b>uom*</b>	String	The unit of measurement for field	lt
<b>now*</b>	Integer	An Epoch timestamp for the reference time	1457481600000
<b>report.reportName</b>	String	The name of the report. Refers to a static set of names included in global configuration	avg

<b>report.startsAt</b>	String	The boundary at which a period starts. One of: hour, day, week, isoweek, month, quarter, year.	day
<b>report.level</b>	String	The level of detail. One of hour, day, week, isoweek, month, quarter, year.	hour
<b>title</b>	String	A title for this collection of views	Last Day

In the following example (expressed as JSX) we create a UnitReport around a timestamp of 1457481600000 (8 Mar 2016), for a time unit of a day and with three enabled views. Note that ReportByDay is just a subclass of UnitReport adding some class-level defaults for a day (e.g. date formatting).

```
var views = require('./views');
var reports = require('./unit-reports');
var ReportByDay = (reportProps) => (
  <reports.ReportByDay
    field="volume"
    uom="1t"
    report={{reportName: 'sum', startsAt: 'day', level: 'hour'}}
    source="meter"
    now={1457481600000}
  >
  <views.Summary />
  <views.SimpleChart />
  <views.ComparisonChart />
</reports.ReportByDay>
);
```

The above example will produce a report as shown in Figure 16.

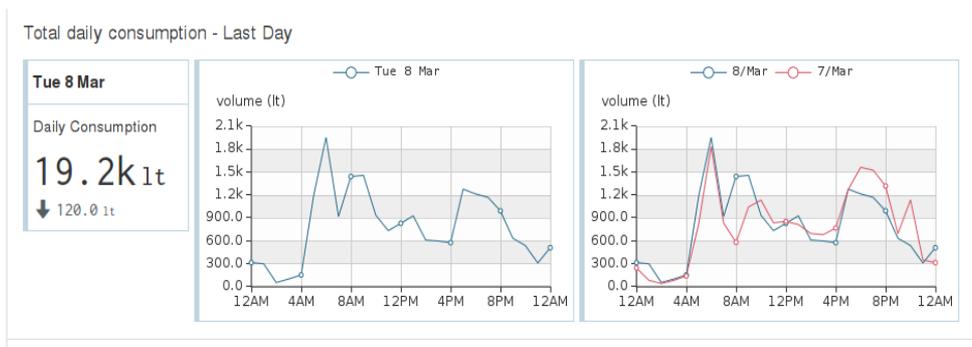


Figure 16: A UnitReport at a unit of a day, with the following views: a summary, a simple chart and a comparison chart

## 2.9. Widgets

The Widgets UI element is a simple control that displays values of measured quantities in a gauge-like fashion. It serves the purpose of displaying important values and highlighting the difference with respective previous values. The source code is available at:

- <https://github.com/DAIAD/home-web/blob/master/src/main/resources/public/assets/js/src/utility/components/reports-measurements/measurement-value.js>

The Widgets UI element is a class that implements a presentational-only React component. It receives as properties a current value and a previous value, and displays the current value, the difference with the previous value, and the increasing/decreasing tendency.

The Widgets UI element receives properties described in the table below:

*Table 18: Properties of Widgets UI element*

Name	Type	Description	Example
value*	Integer, Float	The current value	19200
prevValue*	Integer, Float	The previous value	19320
unit*	String	The unit of measurement for values	lt
title*	String	A title for the widget	Tue 8 Mar
subtitle*	String	A subtitle for the widget	Daily Consumption

In the following example, we create a Widgets UI element by supplying needed values and titles:

```
var {Widget: MeasurementValue} = require('./measurement-value');
<Widget
  value={19200}
  prevValue={19320}
  unit="lt"
  title="Tue 8 Mar"
  subtitle="Daily Consumption"
/>
```

The result of the above invocation is shown at Figure 17.



*Figure 17: A Widgets UI element showing measurement values*

## 2.10. Wizard

The Wizard UI element is a general-purpose component for step-by-step data gathering built for React. It is especially flexible, with each step fully customized with individual components. For simplicity, a wrapper component can be used to abstract the common wizard step displays (e.g., *title*, *description*) and functions (*next*, *previous*, *reset*). The wizard component supports validation for each step, as well as non-linear flows between steps. The source code for the Wizard component as well as full documentation and examples are available at:

- <https://github.com/DAIAD/react-wizard>

The Wizard UI element is implemented using the higher-order component pattern. The library exports a `createWizard` function instead of the Wizard component itself. If no arguments are provided, then the returned component does not use a wrapper. Alternatively, a component can be passed as an argument to the `createWizard`, which wraps each wizard item component. This saves a lot of boilerplate code when creating several steps that need to implement similar functionalities, such as next, previous, reset buttons and a uniform appearance. All wizard functionality is handled by the Wizard component and convenient functions and properties are provided to the user to implement the wizard items with full flexibility.

The configuration options for the Wizard UI element are documented in the tables below. Optional parameters have their names in square brackets:

*Table 19: Wizard configuration options*

Name	Type	Description	Example
<code>onComplete</code>	function	Callback function that executes when the wizard completes	<pre>(values) =&gt; { alert('wizard complete with ' + JSON.stringify(values)); }</pre>
<code>[validateLive]</code>	boolean	Option to call the validation callback function in each step on each render (e.g., while typing in a text input)	true
<code>children</code>	Array/Object	The wizard takes its individual steps as children	See Table 20
<code>[promiseOnNext]</code>	boolean	Option to return promise onNextClicked step function return	true

Table 20: Wizard children configuration options

Name	Type	Description	Example
<b>id</b>	String	The id of the wizard step. Note that id='complete' is reserved	id0
<b>[title]</b>	String	The title of the wizard step that gets passed down to the wrapper and child component. Useful when a component wrapper is used	Step 1
<b>[description]</b>	String	The description of the wizard step that gets passed down to the wrapper and child component. Useful when a component wrapper is used	The first step
<b>initialValue</b>	Any	The initial value that is passed down to the step component	0
<b>validate</b>	function	Callback function that handles validation and must throw string	<code>value =&gt; !value ? throw 'No value' : null</code>
<b>[next]</b>	function	Callback function that allows changing the default flow of the wizard based on the value of the current step (by default it leads to the next wizard child)	<code>value =&gt; value == 1 ? 'id1' : 'id2'</code>

In the following, we provide an example of how the Wizard UI element can be invoked. In this particular example, we create a wizard with two steps; the first prompts the user to enter her name and the second is a confirmation of the first step input.

```
function Step1 (props) {
  const { value, setValue, errors, onNextClicked } = props;
  return (
    <div>
      <h3>Hello</h3>
      <p>Just write your name</p>
      <input onChange={e => setValue(e.target.value)} placeholder="Input" value={value}
    />
    { errors ? <div><br /><span>Validation fail: {errors}</span></div> : <div /> }
    <button onClick={onNextClicked} style={{float: 'right'}}>Next</button>
  </div>
  );
}

function Step2 (props) {
  const { values, onPreviousClicked, onComplete } = props;
  return (
    <div>
      <h3>Confirmation</h3>
      <p>Are you sure you want to submit, {values.name ? values.name.values : ''}?</p>
      <button onClick={onPreviousClicked} style={{float: 'left'}}>Previous</button>
      <button onClick={onComplete} style={{float: 'right'}}>Submit</button>
    </div>
  );
}
```



```

    </div>
  );
}

function WizardExample (props) {
  return (
    <Wizard
      onComplete={values => { alert(`Wizard complete: \n ${JSON.stringify(values)}`); }}
    >
      <Step1
        id='name'
        initialValue=''
        validate={value => { if (!value) { throw 'No name provided'; } }}
      />
      <Step2
        id='confirm'
      />
    </Wizard>
  );
}

```

## Hello

Just write your name

Next

Figure 18: Step 1 of example wizard

## Confirmation

Are you sure you want to submit, Jane ?

Previous

Send

Figure 19: Step 2 of example wizard

In the following figures, we provide two representative examples of how the Wizard UI element is used in the context of the DAIAD@utility web application.

## Who

Select all population or narrow savings potential calculation to selected groups



⚠ Who not set

Previous

↺ Reset

Next

Figure 20: Savings potential wizard, population selection step

## Confirmation

This job will take a while to finish. Are you sure you want to continue?

Previous ↻ Reset Calculate

---

Who: All  
Where: All  
Data: Last year  
Name: Test

Figure 21: Savings potential wizard, confirmation step

## 2.11. Counter

The Counter UI element is a simple control that displays the value and variance of a measured quantity. It serves the purpose of displaying important values and highlighting the difference with respective previous values. The source code is available at:

- <https://github.com/DAIAD/web-client-utility/blob/master/components/Counter.js>

The Counter UI element configuration options are described in Table 21. Optional parameters have their names in square brackets.

Table 21: Counter UI element configuration options

Name	Type	Description	Example
text	String	Counter name or description	Users
value	Number	Counter value	104
[format]	String	Value format: 1. number (default) 2. currency 3. percent	number
[color]	String	Left border color	#5bc0de
variance	Number	Value variance	0
[minusColor]	String	Color for negative variance values	#c0392b
[plusColor]	String	Color for positive variance values	#27ae60
[link]	String	Link to an external resource	/users

Next, an example of using the Counter UI element is displayed.

```
var Counter = require('../Counter');  
  
<Counter  
  text= {'Users'}  
  value= {95}  
  variance= {5}  
  link='/users'  
>
```

The result of the above invocation is shown at Figure 22 .

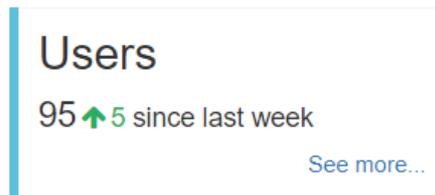


Figure 22: Counter UI element example

## 2.12. Timeline

The Timeline UI element is a simple control that displays an HTML5 input element of type “range”<sup>17</sup>, whose value is in sync with a text control. Every position of the scrollbar is associated with a value-label pair. Whenever the user updates the scrollbar position, the corresponding label is rendered to the text control and optionally a callback function is invoked. Users can either update the scrollbar position manually or enable automatic updates. In the latter case, the scrollbar position is updated at regular intervals. When the maximum position index is reached, the position is set to 0. The Timeline UI element is used for displaying the lineage of data series. The source code is available at:

- <https://github.com/DAIAD/web-client-utility/blob/master/components/Timeline.js>

The Timeline UI element configuration options are enumerated in Table 22. All optional parameters have their names in square brackets.

Table 22: Timeline configuration options

Name	Type	Description	Example
------	------	-------------	---------

<sup>17</sup> <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/range>

<b>[onChange]</b>	function	Callback function that executes when the timeline index is updated. The arguments of the callback are: 1. The current value 2. The current label 3. The current index	(value, label, index) => { ... }
<b>labels</b>	String Array	List of labels	['30/06/2017', '01/07/2017', '02/07/2017', ...]
<b>[values]</b>	Object Array	List of values	[1, 2, 3, ...]
<b>[defaultIndex]</b>	Number	Current timeline index	1
<b>[speed]</b>	Number	Time interval in milliseconds after which the timeline index is incremented	1000
<b>[animate]</b>	Boolean	Enables timeline animation	false

Next, an example of using the Timeline UI element is displayed.

```
var Timeline = require('.././Timeline');

<Timeline
  onChange= {value, label, index} => {console.log(value);}
  labels= [['30/06/2017', '01/07/2017', '02/07/2017', '03/07/2017']]
  values= {[1, 2, 3, 4]}
  defaultIndex= {1}
  speed= {1000}
  animate={false}
/>
```

The result of the above invocation is shown at Figure 23.



*Figure 23: Timeline UI element example*

In DAIAD@utility, the Timeline UI element is always used in combination with a chart or map UI element as shown in Figure 24. In this example, the Timeline control is used for browsing the evolution of water consumption over the last 30 days. Changing the TimeLine scrollbar position results in updating both the date and the map.

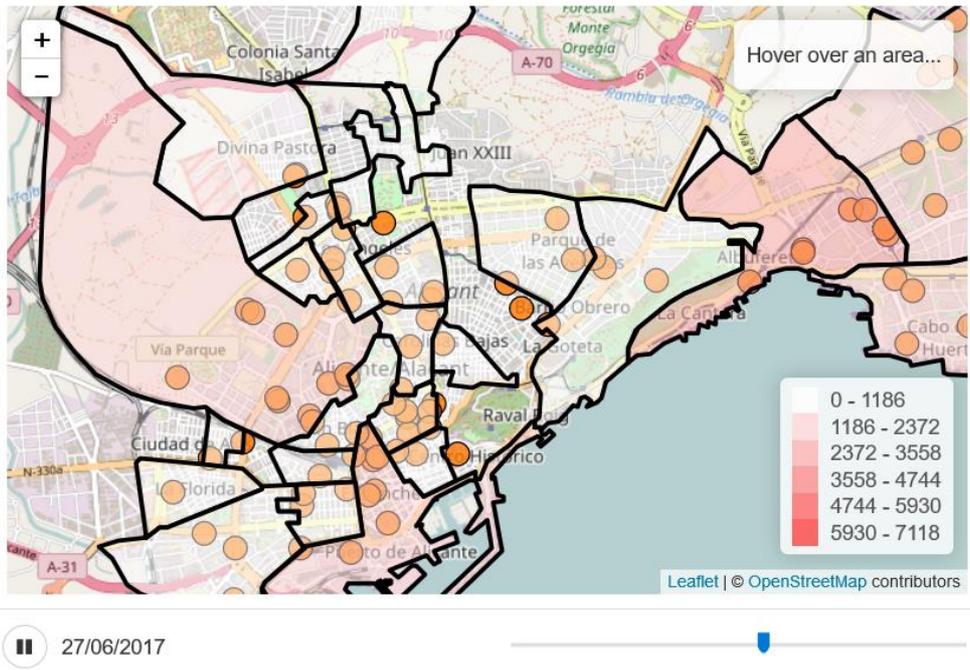


Figure 24: Timeline and Leaflet Wrapper combination