Master in Informatics Engineering
Internship
Final Report

PulseViews: A Solution for Pulse Dashboarding

Luís Filipe Alves Cardoso
lfac@student.dei.uc.pt

Supervisors:
Eng. Olga Filipova
Prof. Tiago Baptista

20 of September of 2012

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Abstract

Today most Business Intelligence solutions are still far from being real-time. FeedZai is a technological spin-off from the University of Coimbra which develops FeedZai Pulse. Pulse processes large amounts of data in real-time to help business users make better decisions. The objective of this work was to enhance the capabilities of Pulse, more specifically to improve the quality and ease of implementation of the dashboards that the users will interact with. This was accomplished by creating Pulse Views. Pulse Views is an easy to use web graphical interface to create Pulse Applications and web dashboards empowering business users. Pulse Views has been deployed as part of Pulse, version 12.1.0 in July 2012.

Keywords

"Dashboard" "FeedZai Pulse" "HTML5" "JavaScript" "BAM"

Nomenclature

BAM Business activity monitoring

BI Business Intelligence

CSS Cascading Style Sheets

DOM Document Object Model

HTML HyperText Markup Language

IT Information technology

KPI Key performance indicator

OS Operating system

UI User interface

SQL Structured Query Language

SVG Scalable Vector Graphics

WDO Widget Definition Object

Acknowledgments

I would like to thank my family, especially my parents, for giving me the support to get here.

I would like to thank my mentors, for all their advice, counsel and motivational speeches.

I would like to thank FeedZai, for giving me the opportunities that I was looking for.

I would like to thank my friends, for all the help and good moments.

You are all great!

Table of Contents

List of Tables

List of Figures

Chapter 1 - Introduction

1.1 Purpose

This document describes the design and implementation of Pulse Views. Pulse Views is a dashboarding solution developed for the product FeedZai Pulse as a part of the thesis of the Master in Informatics Engineering of the University of Coimbra.

This project took place during the academic year of 2011/2012 and was hosted by FeedZai at Instituto Pedro Nunes.

1.2 Vision

One of the toughest problems companies face today is "how to make decisions that matter". Moreover, those decisions have to be made quickly and accurately – in real-time. Unfortunately, more and more, a huge amount of data is flooding through the organization's data processing systems and companies are unable to access the information they so vitally need. And, while current database systems do allow storing large amounts of data, being able to act in real time over that data is an enormous challenge. FeedZai is a technological spin-off of the University of Coimbra. FeedZai develops a product --FeedZai Pulse-- that closes that gap. It allows the processing of very large amounts of data, before the data is even stored, producing real-time, actionable, business intelligence.

Pulse has, besides its data processing and integration components, a JavaScript toolkit: jPulse. jPulse allows developers to add real-time dashboarding widgets to web dashboards. For creating a dashboard a developer needs to code a HTML page with snippets that call this Pulse library that enables to retrieve and show updates in real time using several components (e.g., series charts, bullets and others).

See Figure 1 for an example of a web page containing several jPulse widgets.

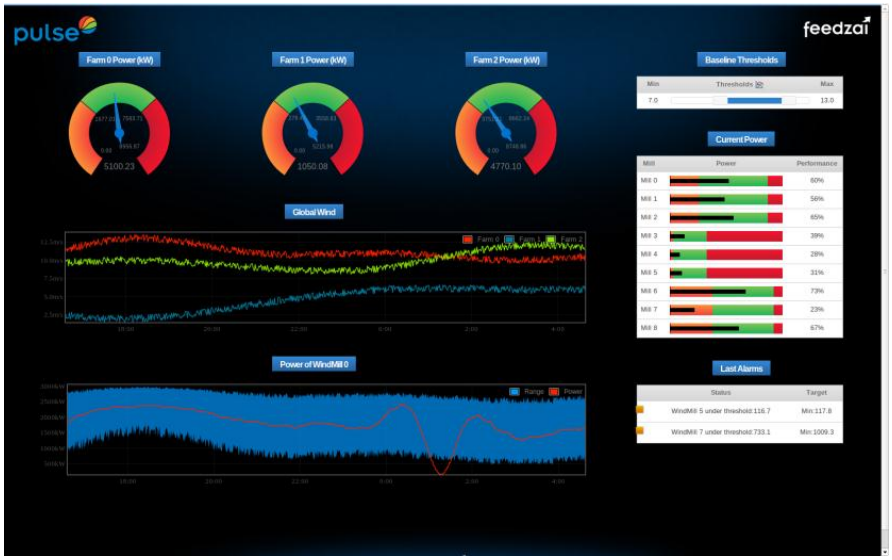


Figure 1. A dashboard that utilizes jPulse.

A major downside of the jPulse toolkit is that developers must use it and integrate its components by writing code. This can be a tedious, error prone and a costly process; even more if you want to deploy to various platforms (e.g., web, iOS, Android). To improve productivity a more visual and easy to use approach is needed. Pulse Views aims to do just

that: it allows creating, updating and viewing dashboards as well as controlling almost any aspect of Pulse Server. All of this is made available via a web browser as soon as the user finishes installing FeedZai Pulse so that they can create and share dashboards right away.

1.3 Contributions

In particular, the following contributions were made:

- Pulse Views dashboard builder, which allows creating and editing dashboards.
- Pulse Views dashboard viewer, which allows viewing dashboards.
- Pulse Views embeddable dashboards, which allows embedding Pulse Views dashboards in external HTML pages.
- Pulse Views Widget API, which renders and allows authoring of widgets.
- Several Pulse Views Widgets.
- Implemented support for iOS in jPulse components.

Overall, the results of the internship were very positive. The objectives set forward were accomplished and the contributions of the project were integrated into Pulse 12.1.0 which is now being used by several FeedZai clients.

1.4 Document Organization

This document is organized as follows:

- Chapter 2, "State of the Art", explains the concepts behind dashboards, compares four existing dashboard building solutions and describes FeedZai Pulse.
- Chapter 3, "Views: a Dashboarding Solution for Pulse", gives an overview of Pulse Views, its requirements and architecture.
- Chapter 4, "Implementation", takes a detailed look at the main components of the system.
- Chapter 5, "Results", discuss the achieved results and tests.
- Chapter 6, "Work Plan and Methodology", looks at the initial plan for the internship, explains how that plan evolved and makes an evaluation of how well that plan was executed and why.
- Chapter 7, "Conclusion and Future Work", summarizes the lessons learned and explains where to go next.

1.5 Annexes

- **Annex A** State of the Art
- **Annex B** flot and jqPlot Performance Tests
- **Annex C** Requirements
- **Annex D** Planning
- **Annex E** Implementing a Simple Dashboard
- **Annex F** Early Versions of the Map Widget

Chapter 2 - State of the Art

2.1 Introduction

Stephen Few, one of the lead visual information designers worldwide, defines a dashboard as:

"(…) a visual display of the most important information needed to achieve one or more objectives; consolidated and arranged on a single screen so the information can be monitored at a glance." [1, pp. 34]

The concept of dashboard comes from the *Executive Information Systems* (EISs), first developed in the 1980s [1, pp. 6]. Their value is their potential to help with the problem of information overload and deliver insight in an especially powerful way. They do this by using high-resolution graphics, emphasizing performance management and metrics and by using knowledge of the visual perception field [1, pp. 37].

According to Few, dashboards can have three different roles:

- Strategic – This is the primary use of dashboards. In this role a dashboard "provides the quick overview that decision makers need to monitor the health and opportunities of the business."
- Analytical – As the purpose of these dashboards is to perform some kind of analysis. They provide "rich comparisons, more extensive history, and subtler performance evaluators."
- Operational – These dashboards are used to monitor operations, and thus they allow the operator to "maintain awareness of activities and events that are constantly changing and might require attention and response at a moment's notice."

An example of a dashboard can be seen on Figure 2.

Figure 2. Example of a dashboard. Source: http://www.microstrategy.com/mobile/graphics/ipad/cfodashboard-4.png

Currently Pulse dashboards have mostly an operational role although they have some analytical features. To give an example, SIBS uses Pulse to monitor transactions and other metrics in real time: if a bank system goes down the operations are immediately alerted and proper action can be taken. Other example is ServeBase (a payment processor) that uses Pulse to do analytics, namely to compare metrics form several customers (see Figure 3).

Figure 3. Example of an analytical dashboard built with Pulse.

2.2 Generic Structure of a Dashboard

The basic element of a dashboard is the widget. Widgets are self-contained UI elements that accomplish a specific function (e.g., a pie chart). Typically in dashboards the function of a widget is to display information.

Besides widgets, dashboards can also contain navigation elements (e.g., link) and visual (e.g., image) elements. Figure 4 shows an example, where you can see a dashboard and several widgets and elements.

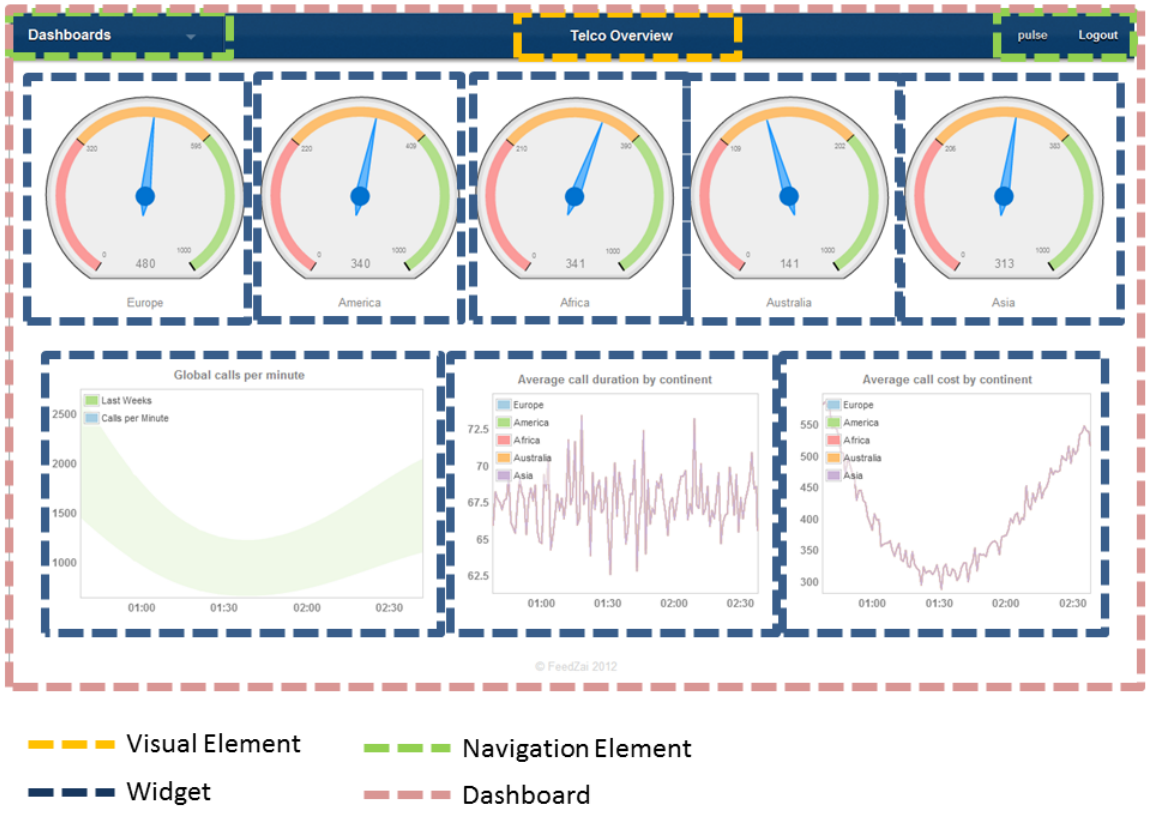


Figure 4. Dashboard elements.

The position and size of the elements on the dashboard is determined by the layout manager. Typically a layout is composed of containers to where the user can assign widgets. For instance to give an example, Java has several layout managers, like BoxLayout (see Figure 5) which puts elements in a single row or column.

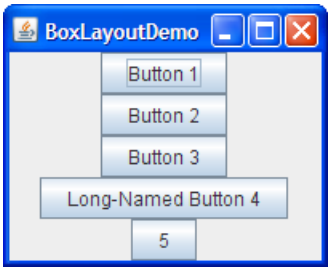


Figure 5. Java BoxLayout layout manager.

The function of a dashboard builder is simply to allow the user to choose the desired layout, the desired element (widgets or otherwise) and their configuration and the container of the layout where each element should be rendered.

2.3 Existing Dashboard Building Tools

2.3.1 Introduction

Real-time dashboarding, with automatic update latencies inferior to one second, is a relatively new area. Since there aren't a lot of solutions in this regard, most of the current state of the art is focused in solutions that present data from relatively static sources (like

traditional databases or data warehouses). In this realm there are solutions from various vendors and in this work we choose to analyze the following ones:

1. MicroStrategy Mobile Suite [2] - strategic dashboards
2. QlikView 11 [3] - analytical dashboards
3. Tableau Desktop [4] - analytical dashboards
4. Splunk [5] - strategic/analytical dashboards

MicroStategy Mobile Suite is a traditional BI tool that is oriented to reports and strategic dashboards. It works on top of data warehouses and data marts.

QlikView is an analytics oriented tool that works on top of a custom in-memory database that allows for data to be filtered by several dimensions very fast.

Tableau Desktop is an analytics oriented tool similar to QlickView but much more focused on following good information design (i.e., how to display data) and usability practices (which are one of their main selling points).

Splunk is a tool for a very specific vertical: machine data (e.g., logs) analysis. Although not currently a direct concurrent of Pulse, it shares several key aspects that make it interesting to analyze.

The choice of the solutions was based on their relevance in the market and their similarity with Pulse in terms of dashboard objectives. The detailed state of the art can be found in the chapter 2.1 of the Annex A.

2.3.2 Comparison

All products share some common aspects, namely:

1. Support for multiple platforms (e.g. web, iOS), which is important because it allows users to view dashboards in the device that is most convenient to them.
2. Focus on avoiding scrolling on the dashboards, which is important because it helps the user to keep focused and compare data.
3. Focus on visual ways to navigate through data, which is important because it makes the discovery process more intuitive.
4. High variety of widgets, which gives users more options to display the data. This is useful only to the extent that it gives users the right tools for the job and not just different ways of showing data the wrong way.
5. Ability to be used by non-IT personnel (to some degree):
    a. Emphasis on being easy to develop (e.g., providing undo/redo). This is important because it allows users to build dashboards without the help of IT experts (which slows the process down and increases its cost).
    b. Ability to share and export content on the dashboards. This is important because it allows users to communicate and store the information present on the dashboard.

On the other hand, there are some very distinct aspects:

1. QlikView and Tableu use an associative model for data exploration in which users can easily filter data by several dimensions and immediately see that data in charts, tables, etc. Splunk uses a search query based model in which users explore data by

interactively writing queries. These two approaches are, arguably, more intuitive than the traditional BI model which is used by MicroStrategy.

2. The MicroStrategy suite has more charts (see Table 6 in Annex A). QlickView and Tableu have fewer but more adequate charts. Especially Tableu that goes so far as to suggest the best charts for different types of data. This is important because if we help the user to build better dashboards (in this case by helping in picking the right widget) the information in them will be more efficiently transmitted.

QlikView and Splunk have a more open architecture (they use standards like HTML and JavaScript). This makes them less dependent on third party vendors and increases the number of compatible platforms.

In terms of dashboard building the highlights can be seen in Table 1.

	MicroStrateby	QlickView	Tableau	Splunk
Layout Approach	12 layouts or absolute positioning.	Absolute positioning or gird.	Based on horizontal and vertical flows.	Row based.
Layout Complexity	Average	High	Average	Low
Layout management/preview separation	No	No	No	Yes
Number of widgets	High	High	Low	Low
Widget Configuration	Complex	Complex	Simple	Simple

Table 1. Dashboard building tools overview.

In conclusion Splunk has the simpler and more easy to use dashboard editor (although it is not as powerful as the others).

2.3.3 Lessons Learned

Given that one of the main objectives was to have an easy to use tool we can conclude that key features to have are:

- Distinct modes for the layout editor and the preview. This allows for the user to either focus on the layout or in the widgets by simplifying the two use cases.
- Simple and straightforward widget configuration: number of options reduced to a minimum and the use of smart defaults. This reduces the time it takes configure a widget.
- Web interface as the way to control the entire application. This approach is more modern, less OS dependent and less costly to deploy and maintain.
- Fewer but better widgets. Given that most users do not have much formation in Information Dashboard Design it's better to give them less and better options in terms of ways to display data.
- Simple layout manager (possibly row based). This reduces the time it takes for a user to build a dashboard.

2.4 FeedZai Pulse

2.4.1 Introduction

FeedZai Pulse is a product being developed by FeedZai in the area of real-time data processing. In this context "real-time" means the ability to produce actionable information updated to the second.

In particular, Pulse aims to solve the following problem: How to provide decision makers with Key Performance Indicators (KPIs) about their operations, updated to the second, being possible to instantaneously compare those KPIs against historical baselines, dissect the information as it arrives, generate alarms and actions when unexpected events occur. Figure 6 illustrates the KPI concept and how it can be compared to the past and future.

Furthermore, Pulse is optimized for being able to do this for huge volumes of data coming from operational systems.

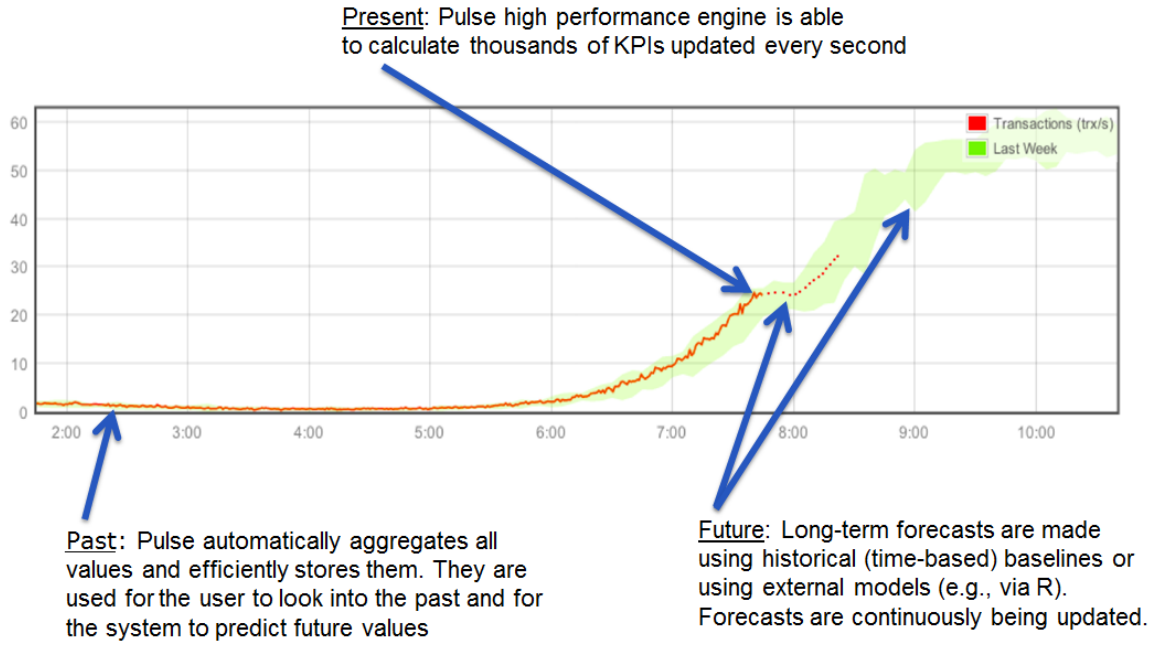


Figure 6. How Pulse handles the past, present and future. Source: internal FeedZai presentation.

2.4.2 Architecture

In terms of architecture, operational systems feed a Pulse server with data. Pulse provides a number of input connectors which allow information to arrive in real-time (e.g., Sockets/HTTP-JSON/WebServices, etc.). Information is processed in the Pulse Application Server which, in the end, can either be feed to other systems or immediately made available using a web interface called HTTP Output Adapter [6] which is used by jPulse [7], a JavaScript library that is the basis of the real time dashboarding components [8]. This architecture is illustrated by Figure 7.

At the core of the FeedZai Pulse engine resides a high-performance Complex Event Processing (CEP) engine. The engine allows continuous queries to be defined which produce results as new data arrives at the system. A continuous query typically works over a time window. For example, a possible query is: "calculate the average power consumption of the electric over a time window of the last 10 minutes". The system maintains in memory the events corresponding to the time window needed to calculate the average. As new events

arrive them are added to the window, and as they become obsolete they are removed from the window. Note that the result of the query can be computed extremely efficiently. As the query is continuously producing an "average", the system only needs to maintain a "total" value and a counter of events being used. These two values are updated with the new data coming to the window (increasing the total and the count) or as old data expires (decreasing the total and the count). Contrary to what happens in a database, there's no need to revisit all the records (events) to produce a new result. Furthermore, the events that are not being used in the time window, for time windows that are quite large, can be temporally sent do disk in bulk.

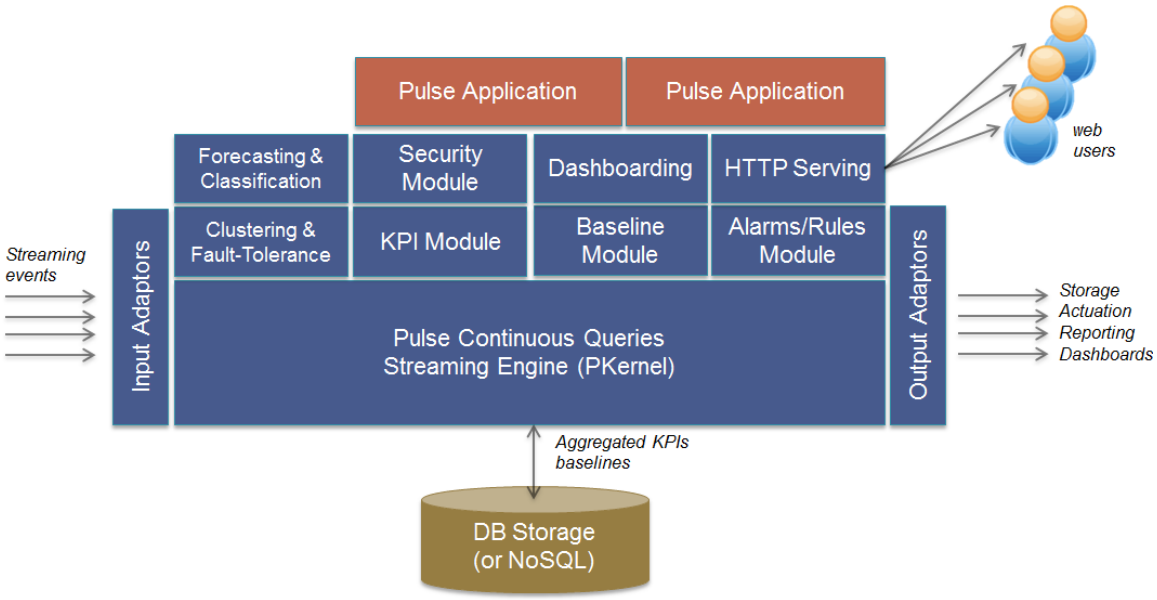


Figure 7. Architecture of Pulse.

2.4.3 Concepts

- A **stream** represents a source of real-time data. It's like a connection port between the world and Pulse.
- A **hierarchy** represents how you want to navigate the data for a number of related KPIs
- A **KPI** (Key Performance Indicator) represents a business metric that needs to be tracked in real-time. KPIs are always associated with a certain hierarchy.
- A **baseline** represents values that we expect a KPI to have at a certain moment.
- **Alerts** are notifications sent to the user whenever a KPI reaches some limits. Examples of limits are baseline boundaries and fixed values.

Chapter 3 - Views: a Dashboarding Solution for Pulse

3.1 Introduction

The main objective of this project was to build a dashboarding solution for Pulse. As Pulse already supported building dashboards, albeit manually, one of the first steps taken before defining what Pulse Views would be was to create a simple yet complete dashboard using Pulse.

The chosen application was a monitoring system for EDP and the resulting dashboard can be seen in Figure 8. It features a drillable map, three KPIs and is iOS optimized. A complete report can be found in Annex E.

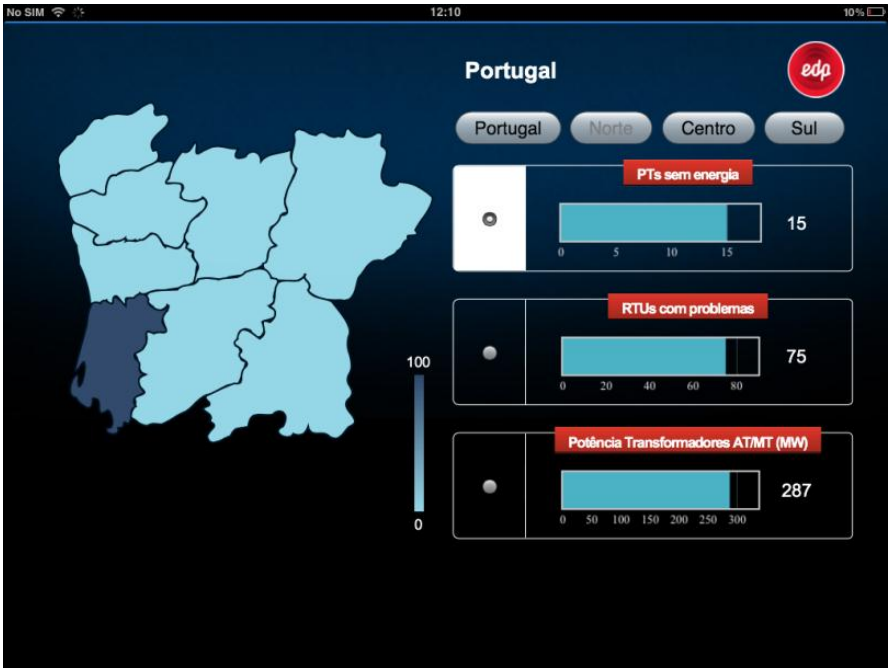


Figure 8. Simple dashboard built manually.

During this experience it was clear that the existing abstractions were too low level and even for programmers the learning curve was steep. With jPulse it was not feasible to expect non-IT personnel to build dashboards with Pulse and this was a major limitation. Pulse Views aimed to solve these problems by providing an easy to use and intuitive UI that can be used by non-IT personnel.

3.2 Requirements

On a very high level the requirements can be described by Figure 9.

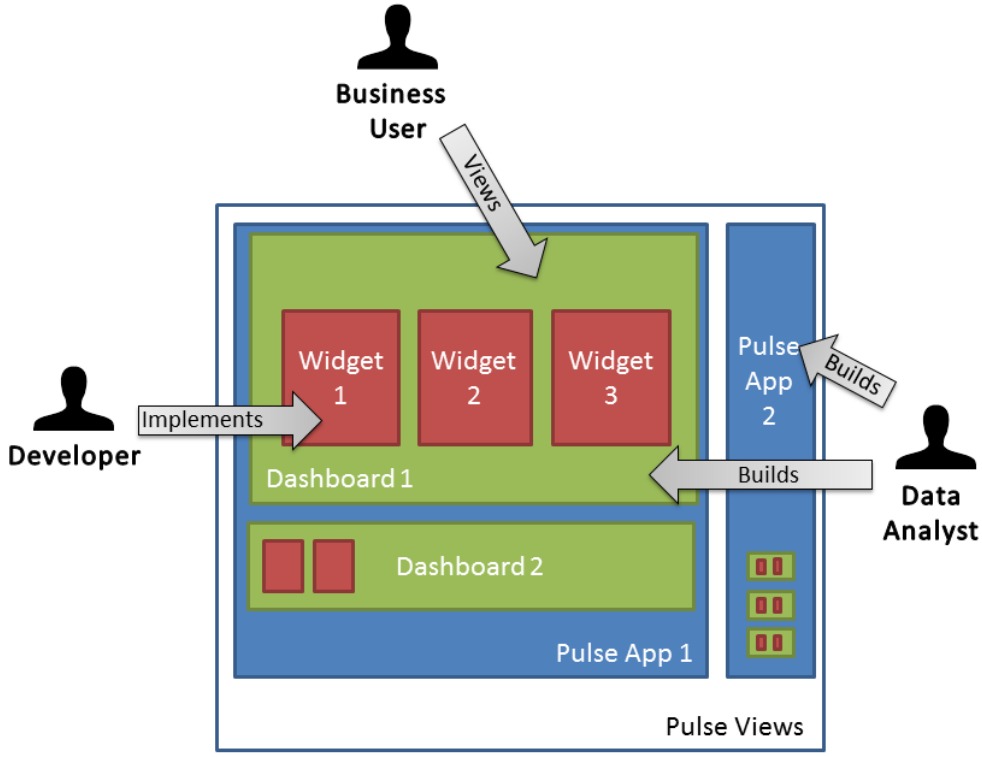


Figure 9. Overview of Pulse Views from a use case perspective.

A detailed analysis of the requirements can be found on Annex C. Even so overall, there are four main groups of requirements:

1. Global requirements
2. Building dashboards
3. Viewing dashboards
4. Widget authoring

And three roles:

- **Data Analyst** – The role of the data analyst is to explore the data and build the dashboards that better conveys its.
- **Business User** – The role of the business user is to use the information presented in the dashboards to make informed decisions.
- **Developer** – The roles of the developer are to author new widgets and to integrate the embedded dashboards into custom web pages.

The summary of global requirements follows:

- Pulse Views must be packaged inside the Pulse Server as a web application
- The user can only access Pulse Views with a valid username and password
- Pulse Views implements the concept of a Pulse App, which is a self-contained and independent entity created by a user. Inside a Pulse App there can be several dashboards that can only access the resources of that Pulse App.
- The possible data sources for widgets are:
    - KPIs: represents a business metric that needs to be tracked in real-time

  - Alerts: notifications sent to the user whenever a KPI reaches some limits
  - Custom Data Sources: connections to external databases that fetch records of a given table

The viewing dashboards requirements are pretty straightforward. The user must be able to:

- Open and view a dashboard if he has access to it
- Be able to switch to other dashboards of the same Pulse App

The building dashboards requirements include the ability to:

- Create, preview, edit and delete dashboards if the user has access to the dashboard
- A dashboard must have the following attributes:
  - A human readable name
  - A machine name (for use in URLs for example)
  - A description of its layout
  - A list of the widgets in the dashboard and their place in the layout
  - Security definitions
  - Custom CSS to apply to the dashboard
- Define an initial layout for the dashboard and change it later. The layout manager must allow the user to define containers with various row and column spans and to remove containers.
- Assign widgets to a layout container
- Configure the widget settings including:
  - Widget related settings
  - Data source related settings
- Generate a HTML snippet to embed dashboards in other web pages hosted in the same instance of the Pulse server.

In order to allow the authoring of widgets by both users and other FeedZai employees it must be possible to:

- Define a rendering function that is responsible to render the widget and that is agnostic to the way and the place where the rendering is done
- Define a set of options which will be configured by the user
- Define the necessary metadata to ensure that the widget is rendered properly
- Allow for event based communication between the widgets
- Define all aspects of a widget in a single file
- Add new widgets to the library in a Pulse production instance

3.3 Approach

3.3.1 Overview

This section provides an overview of the architecture by looking at the top level elements that constitute it. Figure 10 shows the architecture at a conceptual level.

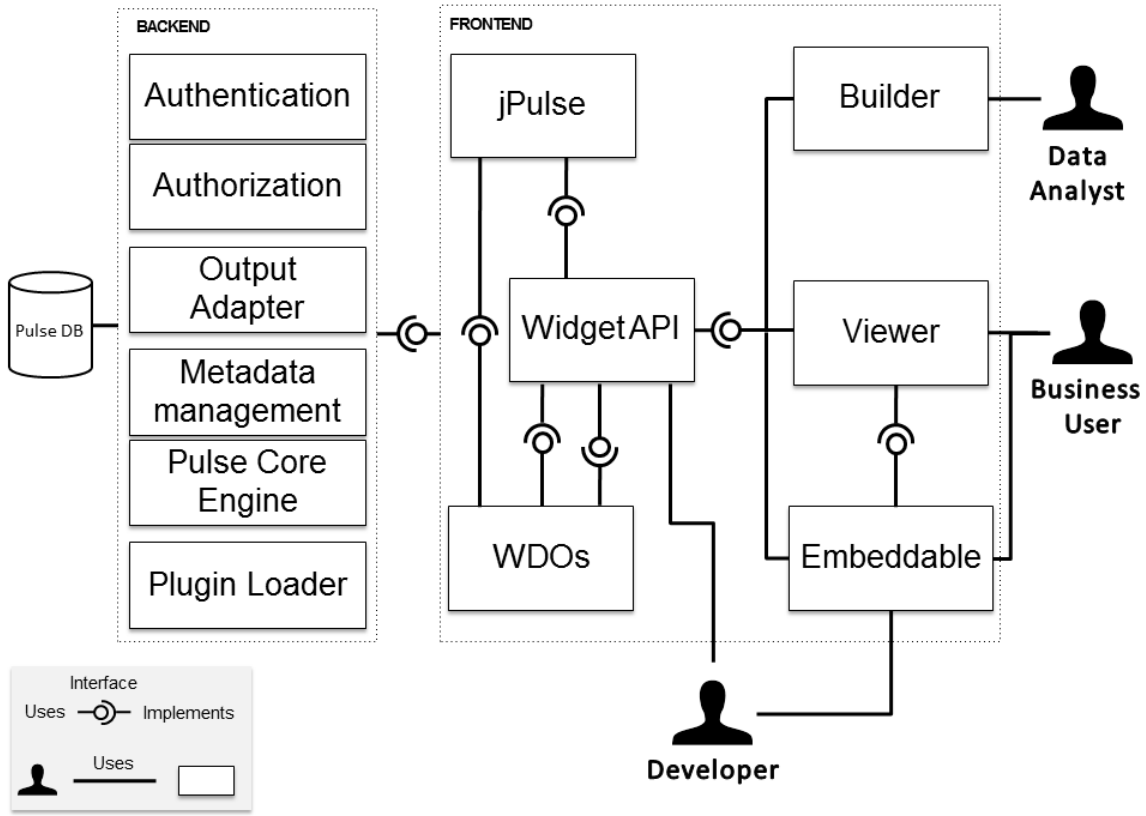


Figure 10. Pulse Views architecture on a conceptual level.

- **Pulse DB** – Storage abstraction used by the Pulse modules. It provides a common abstraction atop of several database engines. This abstraction implements a limited subset of SQL functionality.
- **Authentication** – Authentication of Pulse users.
- **Authorization** – Manages the access to various resources including Pulse Apps and dashboards.
- **Output Adapter** – Provides an API to access the output of the event processing engine.
- **Metadata management** – Provides an API to access metadata regarding Pulse Apps and their components.
- **Pulse Core Engine** – Modules responsible for the event processing, KPIs, baselines, rules and other internal modules.
- **Plugin Loader** – Responsible for discovering external widgets (i.e., widget implemented by Pulse developers) and making them available to be used by the Widget API.

- **Viewer** – Web application which sole responsibility is showing the dashboards to the user.
- **jPulse** – Library to facilitate the interaction with the Output Adapter.
- **Builder** – Web application that allows users to manage all aspects of the Pulse Apps. This includes managing and authoring dashboards and dashboard layouts, configuring widget options, accessing the Viewer and the embeddable script.
- **Embeddable** – Script that allows for rendering Pulse Views dashboards in external web pages. This includes loading all the necessary dependencies and initializing the Viewer.
- **Widget API** – Collection of modules that allow for loading, initializing and rendering widgets.
- **WDOs** – (Widget Definition Objects) Collection of modules that implement widgets.

3.3.2 Architecture

3.3.2.1 Introduction

The architecture of Pulse Views will be described in two perspectives: static and dynamic.

The static perspective shows the code modules (and their dependencies) of the system and the dynamic perspective shows the flow of data while the system is running. Each module corresponds to a physical source code file.

Typically a physical perspective is also included, but in this case that as deemed as unnecessary because it would be composed of only a web server and a web browser.

3.3.2.2 Static Perspective

This section will present a static perspective view of the architecture components (see Figure 10) implemented in this project. These components are the Viewer, Builder (only in part implemented in this project), Embeddable and Widget API. Regarding the WDOs only some of them were implemented in this project and those are presented in section *5.2.2 Widgets*.

On a source code perspective there is a component not present in Figure 10 that contains the code common to the builder and the viewer. This component is called *Commons*.

The diagram of the static perspective of the Widget API can be seen in Figure 11.

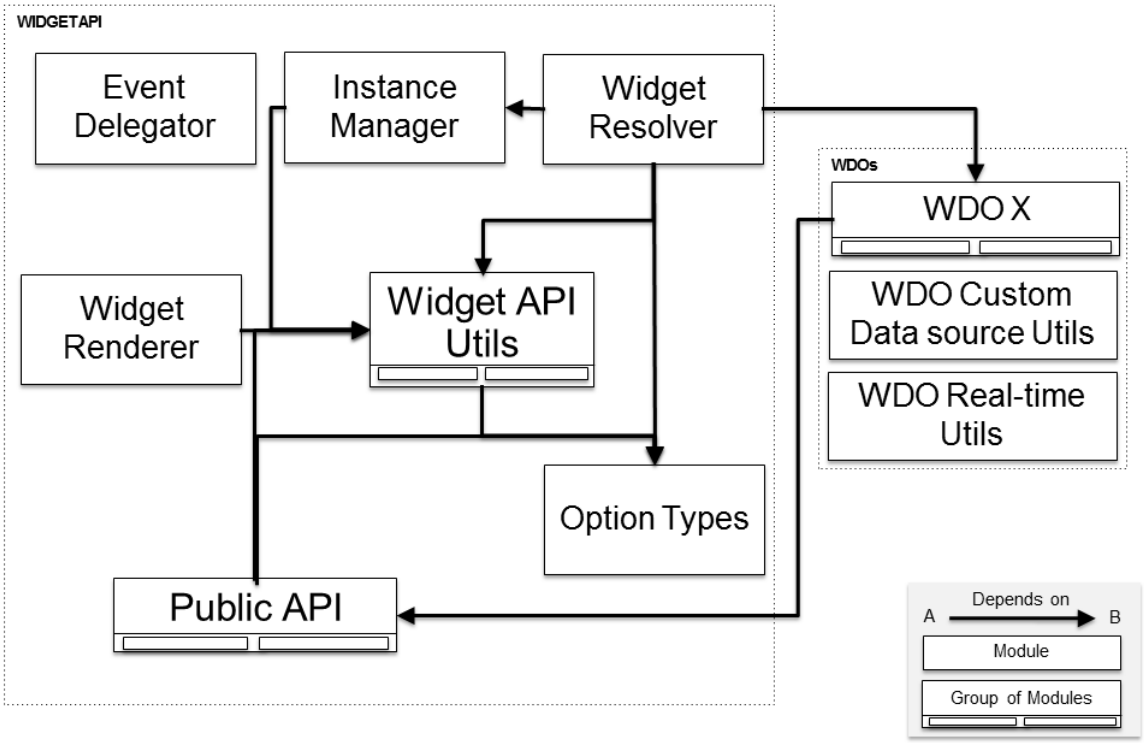


Figure 11. Static perspective of the Widget API.

A description of the modules follows:

- **Event Delegator** – Handles the event delegation between widgets.
- **Instance Manager** – Keeps track of all the widget instances in a dashboard. Used for reference when configuring the widget events.
- **Widget Renderer** – Responsible for rendering a given widget with a given configuration.
- **Widget API Utils** – Series of modules containing utilities used by the other modules.
- **Widget Resolver** – Initializes all the widgets and provides an interface for accessing WDOs.
- **Option Types** – Enumeration of the possible types of options (e.g., String, Integer).
- **Public API** – Modules accessible by the widget authors. Includes a proxy to the Option Types enumeration and a proxy to some of the most used Widget API Utils methods.
- **WDO X** – Represents all widget definitions. Each one defines how a widget renders, its user configurable options and some metadata. Each WDO can be composed of several modules.
- **WDO Real-time Utils** – Methods that facilitate the interaction with jPulse by providing some higher level abstractions.
- **WDO Custom Data Source Utils** – Methods that facilitate the fetching of records from custom data sources (i.e., external databases).

The diagram of the static perspective of the Viewer can be seen in Figure 12.

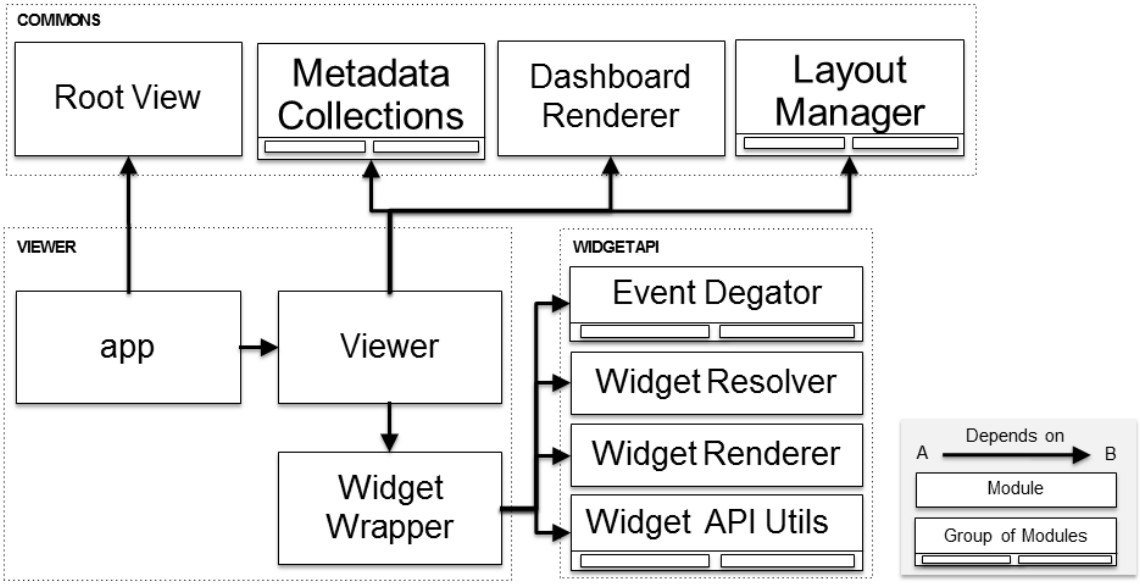


Figure 12. Static perspective of the Viewer. Note: only the modules of the Widget API with which the Viewer interfaces are shown.

A description of the modules follows:

- **Root View** – Implements the log in, error handling, DOM initialization both in the Viewer and in the Builder.
- **Layout Manager** – Build the DOM for a layout defined in a configuration object.
- **Metadata Collections** – Series of collections that map to the metadata API.
- **Dashboard Renderer** – Dashboard rendering related utilities.
- **app** – Starting point of the application.
- **Viewer** – Main module of the Viewer component. Coordinates all the other modules.
- **Widget Wrapper** – Responsible for rendering widgets.
- **Event Delegator** – Handles the event delegation between widgets.
- **Widget Resolver** – Initializes all the widgets and provides an interface for accessing WDOs.
- **Widget Renderer** – Responsible for rendering a given widget with a giver configuration.
- **Widget API Utils** – Series of modules containing utilities used by the other modules.

The diagram of the static perspective of the Embeddable can be seen in Figure 13.

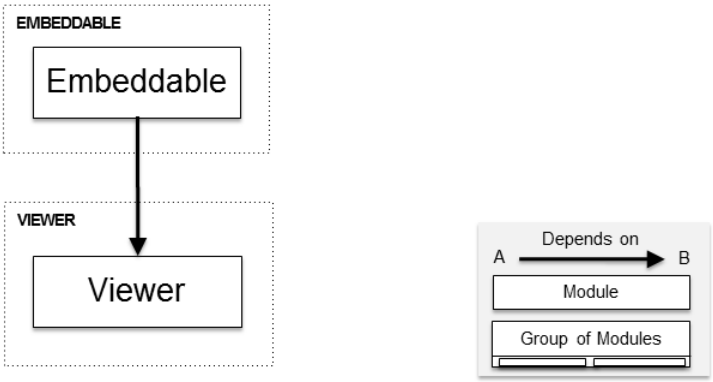


Figure 13. Static perspective of the Embeddable.

A description of the modules follows:

- **Embeddable** – Loads (if they haven't been loaded before) all the necessary dependencies (static libraries like jQuery) and the Viewer itself. Initializes the Viewer in the container and with the Pulse App and dashboard specified by the developer.
- **Viewer** – Main module of the Viewer component.

The diagram of the static perspective of the Builder can be seen in Figure 14.

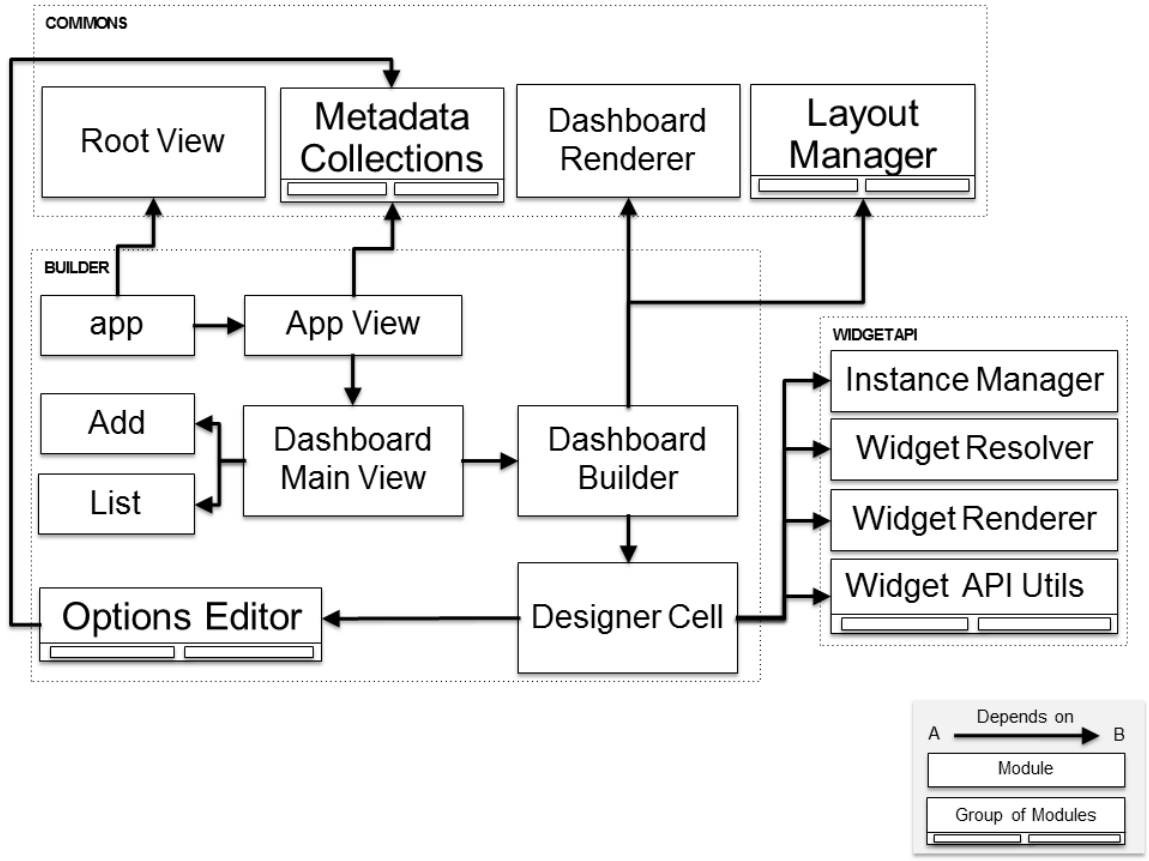


Figure 14. Static perspective of the Builder. Note: only the modules of the Widget API with which the Builder interfaces are shown.

A description of the modules follows:

- **Root View** – Implements the log in, error handling, DOM initialization both in the Viewer and in the Builder.
- **Layout Manager** – Builds the DOM for a layout defined in a configuration object.

- **Metadata Collections** – Series of collections that map to the metadata API.
- **Dashboard Renderer** – Dashboard rendering related utilities
- **app** – Starting point of the application.
- **App View** – Responsible for building the DOM around the dashboard section of Pulse Views.
- **Dashboard Main View** – Responsible for directing the user to the correct screen (wither Add, List or Dashboard Builder) according to the URL.
- **Add** – View used to create a dashboard or edit its metadata.
- **List** – View used to render a list of all existing dashboards. Also renders controls that allow, for each dashboard to, open it on the Viewer, clone it, open the Add screen and remove it.
- **Dashboard Builder** – Orchestrates the modules that allow configuring widgets and changing the layout of a dashboard.
- **Designer Cell** – Renders the preview of a widget.
- **Options Editor** – Renders the options of a widget.
- **Instance Manager** – Keeps track of all the widget instances in a dashboard. Used for reference when configuring the widget events.
- **Widget Resolver** – Initializes all the widgets and provides an interface for accessing WDOs.
- **Widget Renderer** – Responsible for rendering a given widget with a giver configuration.
- **Widget API Utils** – Series of modules containing utilities used by the other modules.

3.3.2.3 Dynamic Perspective

This section will present a dynamic perspective of the Builder and Viewer components. The objective of this perspective is to illustrate how data moves between the application modules. The frontend modules correspond to the ones presented in the static perspective.

The dynamic diagrams are composed of three layers:

- **Persistent Storage** – Responsible for storing data in a persistent way. Typically this will be a relational database.
- **Data Access** – Manages and facilitates the access to the application state. Acts as the model of the MVC pattern.
- **Presentation** – Renders and maintains the (non-persistent) state of UI. In terms of the MVC pattern the presentation layer acts as both the view and the controller.

The diagram of the dynamic perspective of the Widget API can be seen in Figure 15.

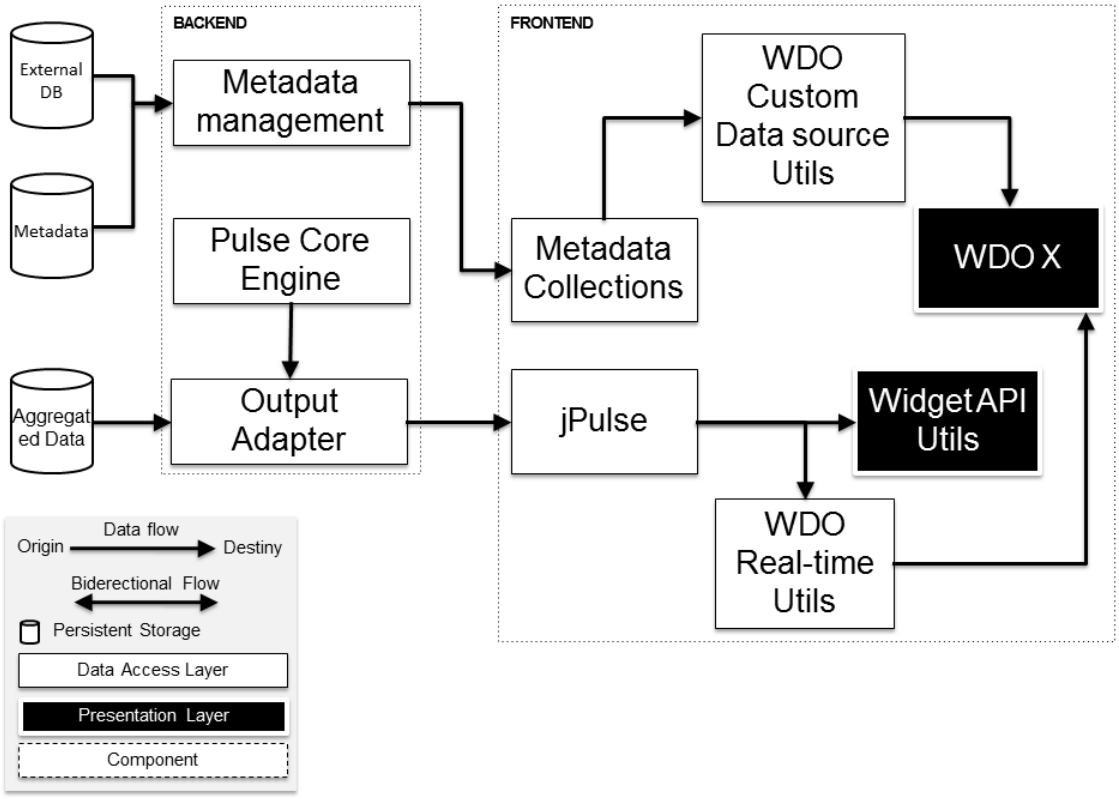


Figure 15. Dynamic perspective of the Widget API.

In the Widget API the data flow is focused in displaying data to the user, be it real-time events or records form a custom data source. In the case of real-time, historic data is retrieved from the Pulse DB (that stores both metadata and aggregated data) by the Output Adapter module, it is them passed to the client side jPulse library who passes it to the widgets that then show the data to the user. The WDO Real-time Utils acts simply as a proxy who provides an easier to user interface than jPulse. The real-time data follows the same path, except that it comes from the Pulse Core Engine instead of the Pulse DB.

In the case of custom data sources, the widget requests the records to the WDO Custom Data Source Utils module that uses the necessary collections from Metadata Collections to fetch the data from the Metadata Management module which is responsible to fetch the records from the external database.

The connection to the *Widget API Utils* is due to the necessity of doing some run time checks to evaluate if a widget has conditions to render.

The diagram of the dynamic perspective of the Viewer can be seen in Figure 16.

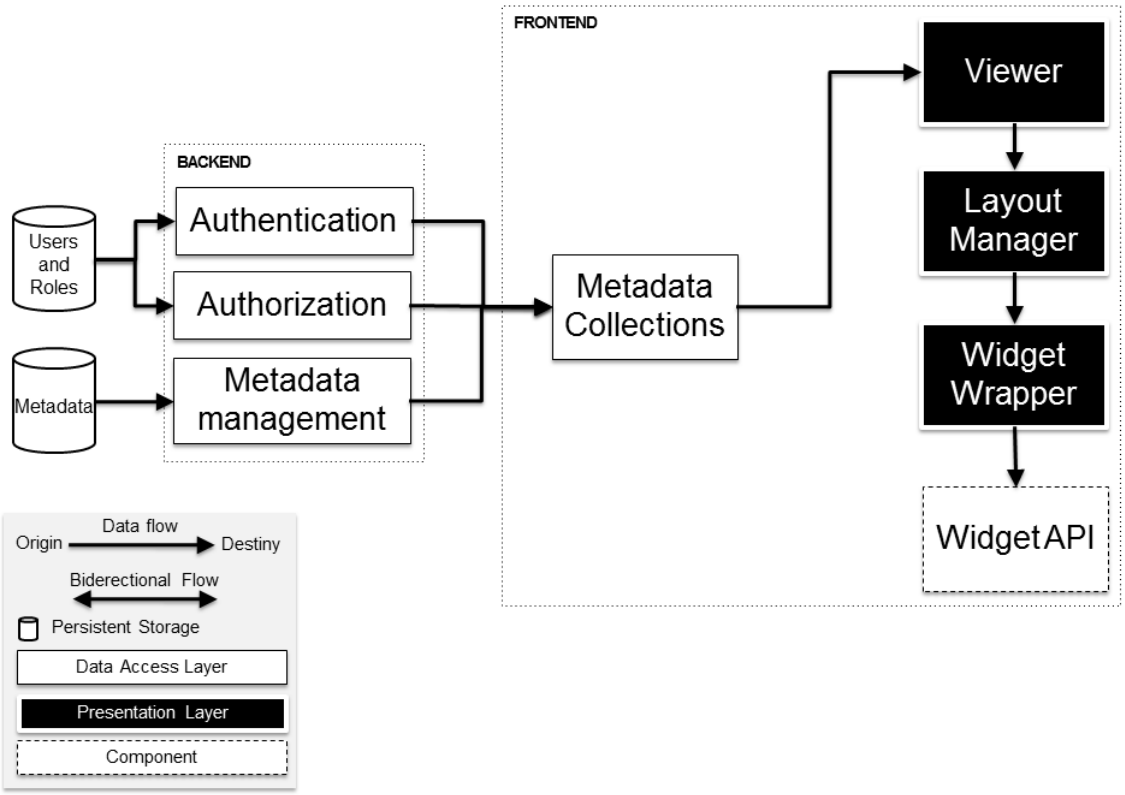


Figure 16. Dynamic perspective of the Viewer.

The Viewer module takes the target Pulse App and dashboard (passed in the URL, for example) and first asks the Metadata Collections for the data relative to the Pulse App and dashboard. This data is then passed to the layout manager (who builds the layout) which passes the individual widget configurations to the Widget Wrapper module. Finally the Widget Wrapper uses the Widget API to render the widget.

The Viewer also uses the Metadata Collections as the interface for authenticating the user.

The authorization checks are done in the server when a dashboard's data is requested. If the user is not authorized the Viewer takes care of communicating that to the user.

The dynamic perspective of the Embeddable does not really add anything new (it would be Figure 16 without authentication and authorization) so it is omitted.

The diagram of the dynamic perspective of the Builder can be seen in Figure 17.

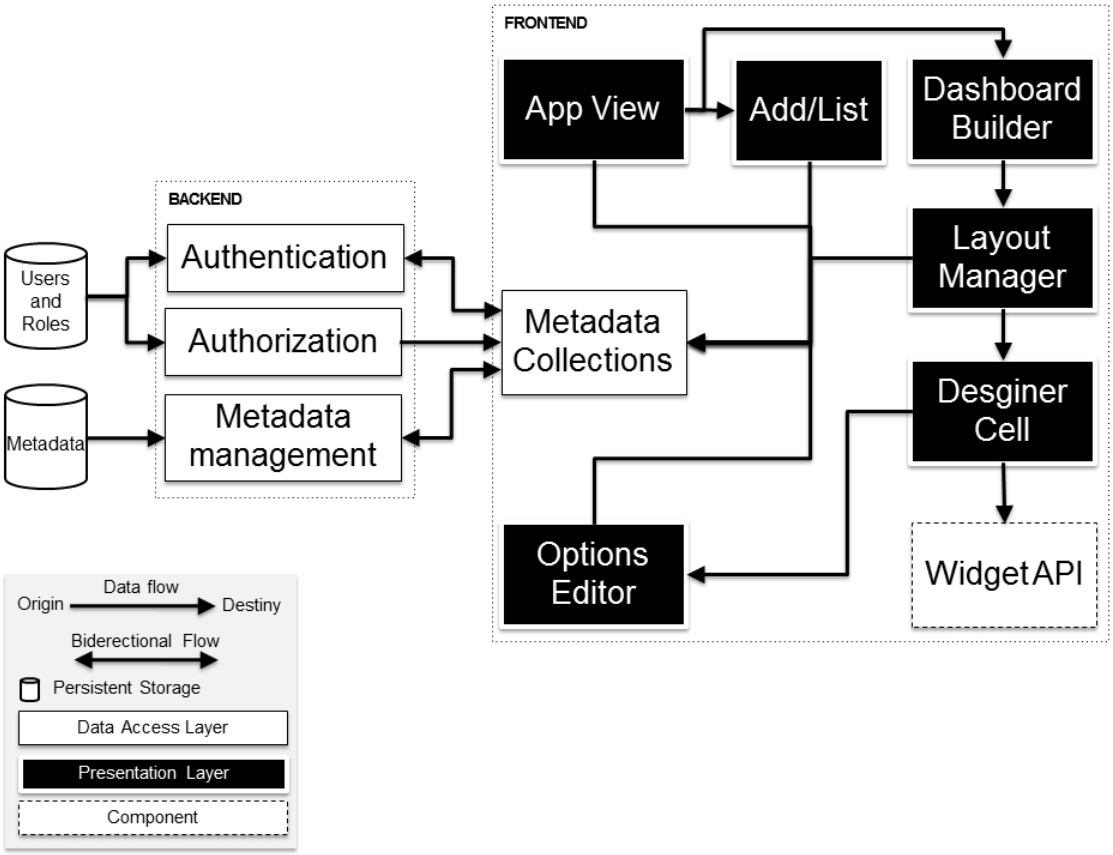


Figure 17. Dynamic perspective of the Builder.

The App View module takes a target Pulse App (passed in by URL for example) and uses the Metadata Collections to fetch the respective data. It also handles authentication and authorization. The List module lists all the dashboards in the current Pulse App using the data fetched by the App View and the Add module allows for adding new dashboards and editing the metadata of existing ones.

The data fetched by the App View is also used by the Dashboard Builder to render the preview of a specific dashboard. During the rendering of the preview the dashboard configuration is passed to the Layout Manager who passes the individual widget configurations to the Designer Cell. The Designer Cell module uses the individual configurations to render the preview of the widget (by using the Widget API) and to pass to the Options Editor module that renders the dialog that allows the widget configuration to be changed by the user. When the user wants to save the configuration this is done via the Metadata Collections.

Also, changes made to the layout are persisted via the Metadata Collections.

Chapter 4 – Implementation

4.1 Introduction

Figure 18 through Figure 23 shows some screens that resulted of the implementation of Pulse Views.

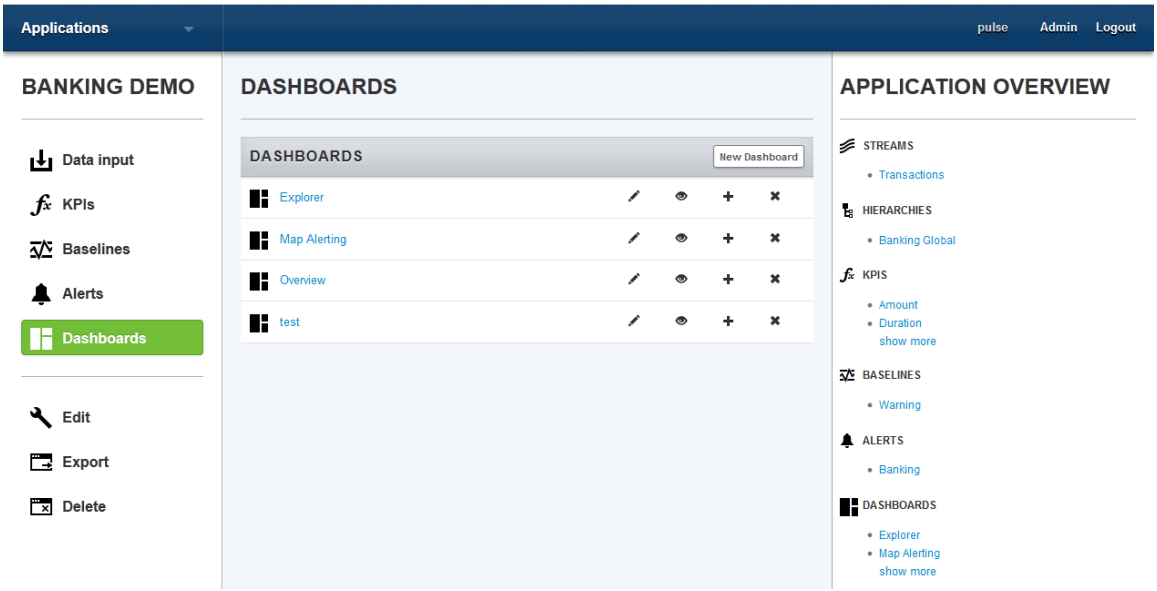


Figure 18. Dashboard list screen in the Builder.

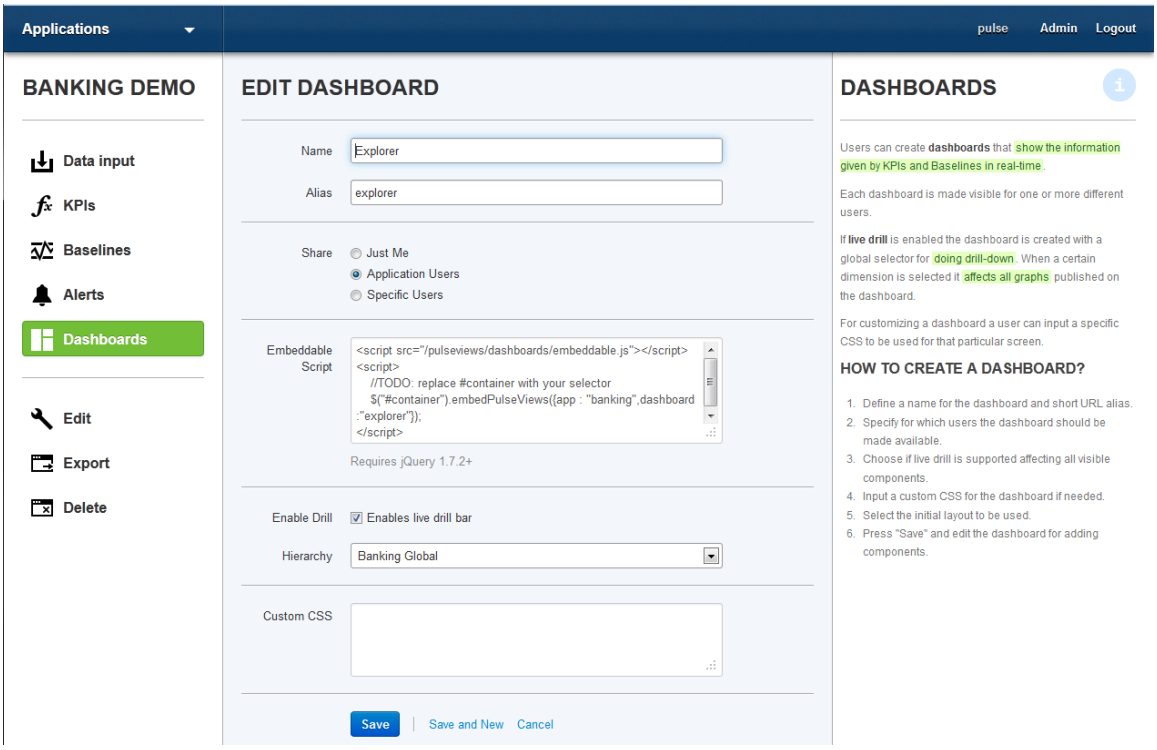


Figure 19. Dashboard edit metadata screen in the Builder.

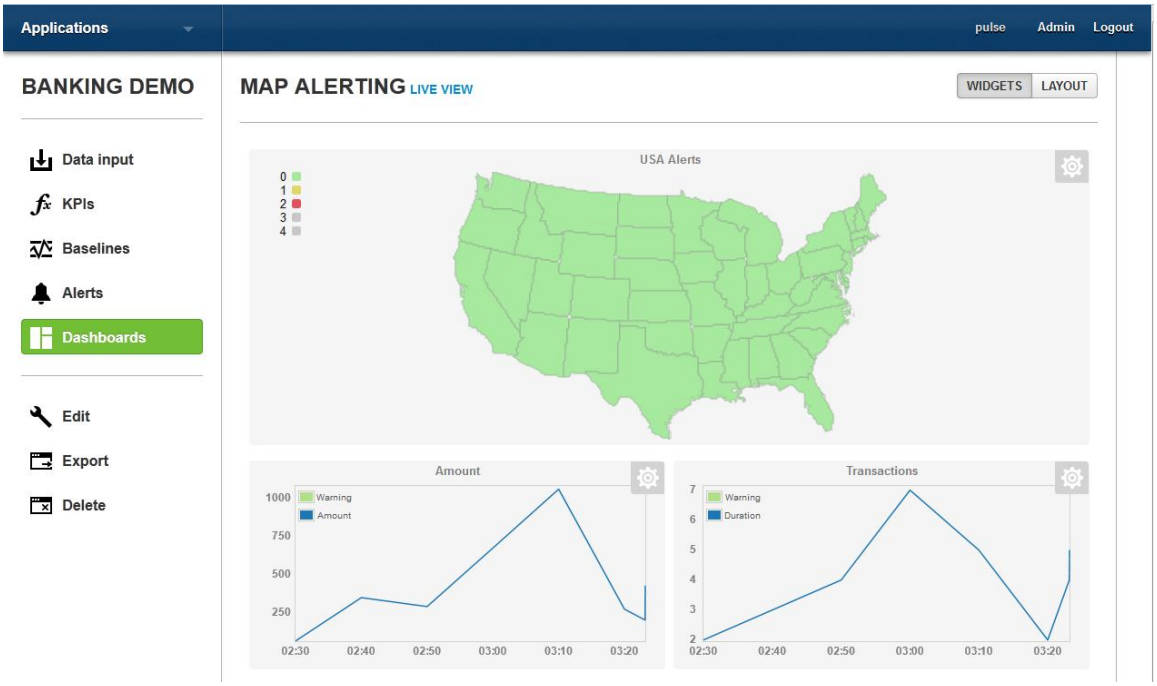


Figure 20. The dashboard preview screen in the Builder.

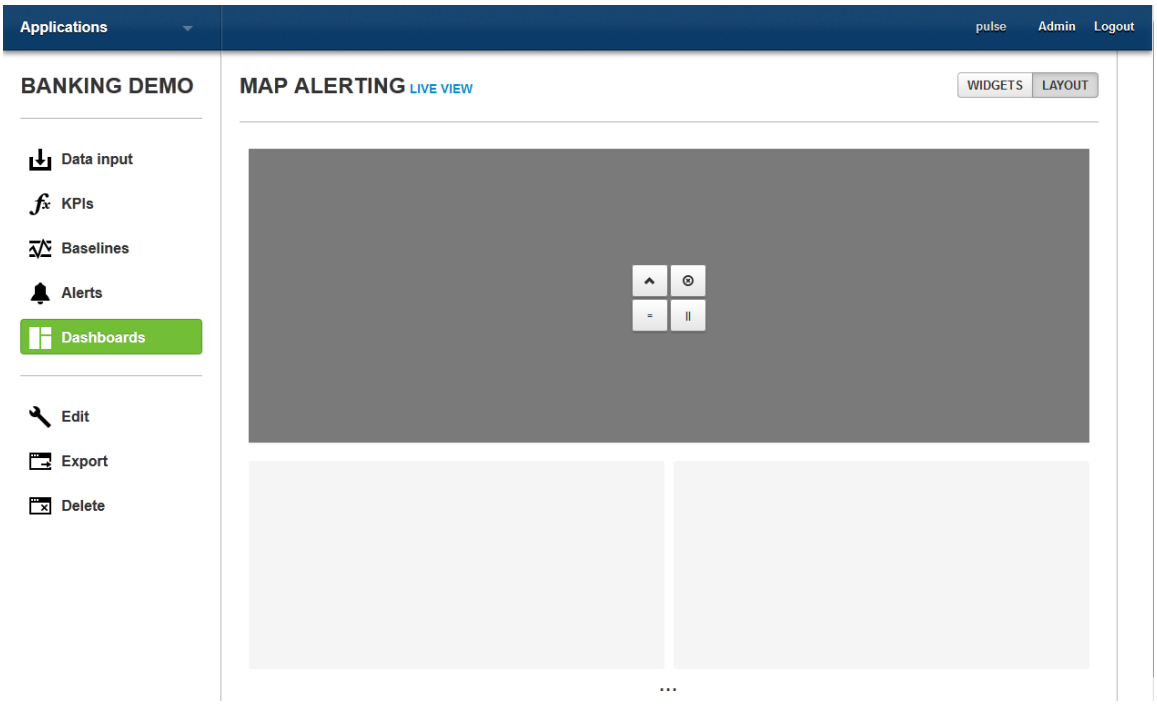


Figure 21. The dashboard layout authoring screen in the Builder.

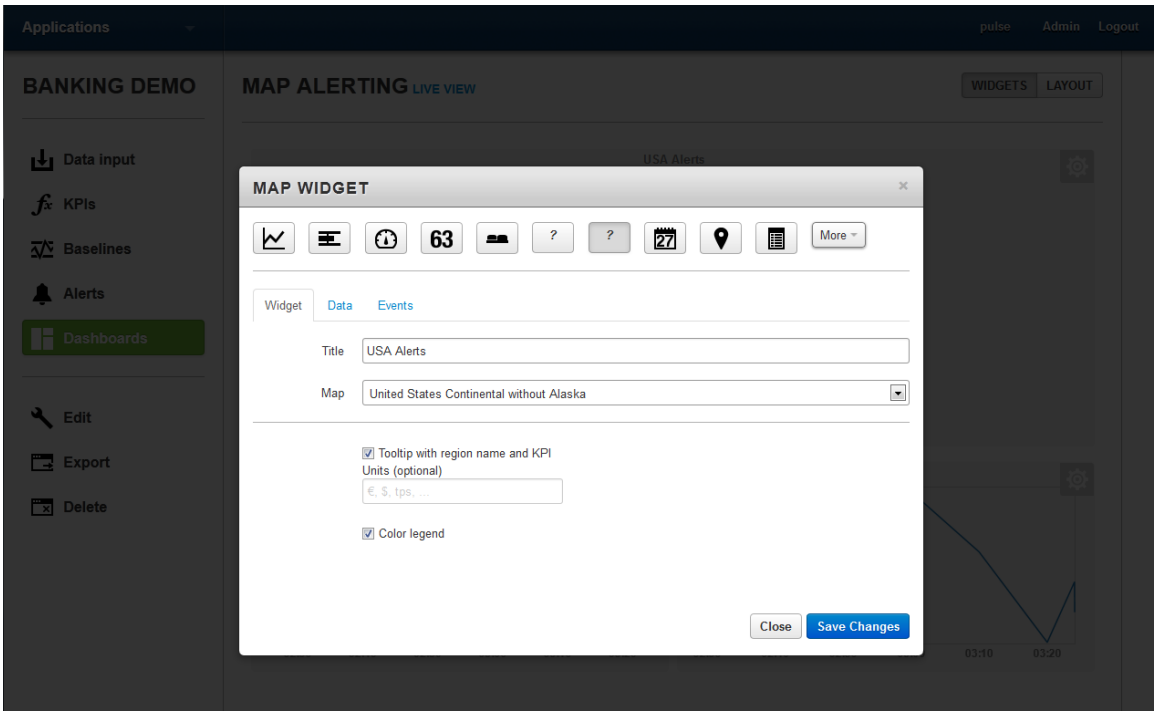


Figure 22. The widget options editor in the Builder.

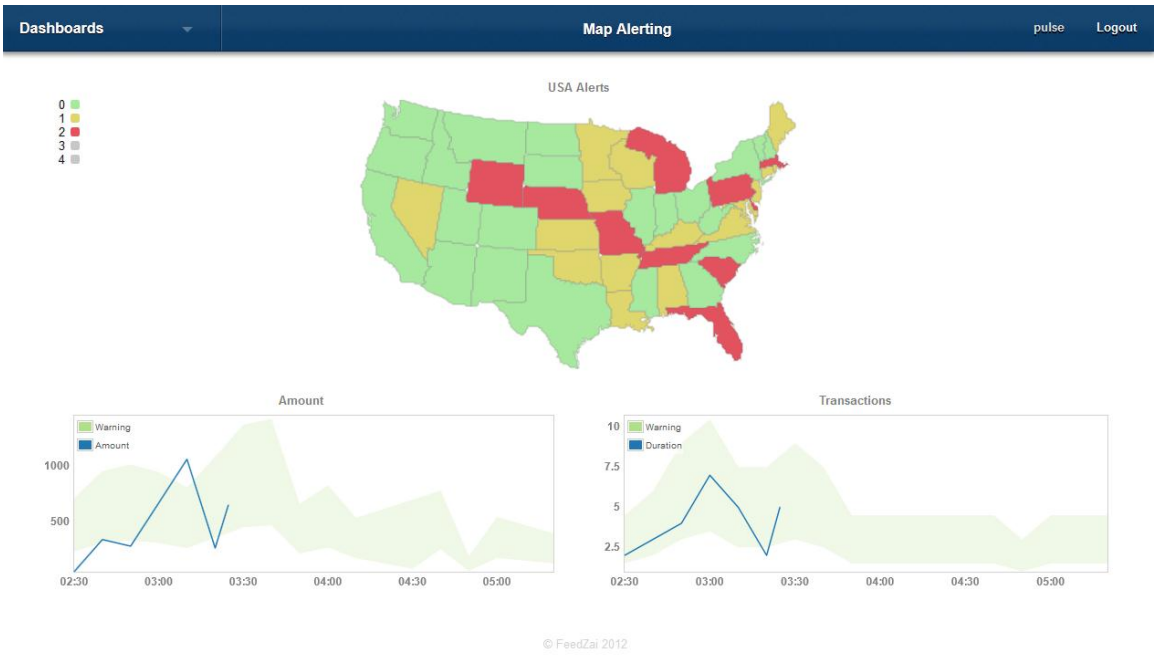


Figure 23. The Viewer.

4.2 Technologies and Libraries

In terms of technologies the choices were very simple. Given the necessity to integrate with jPulse, which is written in JavaScript, Pulse Views had to be written in JavaScript. In terms of communication with the server, given that we have a REST API the only option is HTTP.

There were several important decisions to make in terms of libraries and most of them are related with facilitating the development of a client side JavaScript application. Currently most of the JavaScript APIs provided by the browsers are quite hard to use and even

inconsistent between browsers. To mitigate this there are several classes of libraries that a developer can use:

- **AJAX and DOM manipulation** – gives the ability to perform AJAX requests and easily manipulate the DOM via a friendly and well documented API.
- **UI elements toolkit** – provides the developer with several ready-to-use and customizable UI elements.
- **HTML templating** – gives the ability to define templates to dynamically generate HTML client side.
- **Asynchronous Module Definition Loader** – allows organizing the code in modules (one by file). Each module can declare its dependencies (that are other modules).
- **MV* Framework** – MVC like frameworks that allow for a degree of separation of concerns.
- **CSS preprocessor** – allows CSS to be written using some abstractions typically not supported by browsers (e.g., variables).

The libraries chosen for each of the classes referred to can be found on Table 2.

Library Class	Name	Motivation
AJAX and DOM manipulation	jQuery [9]	Is the one used by jPulse and using other would make the integration much more difficult.
UI elements toolkit	Twitter Bootstrap [10]	Backed up and used by Twitter. Outstanding community (the biggest in Github). Comprehensive set of UI elements. Uses jQuery.
HTML templating	Google Closure Templates [11]	Developed by Google and used in heavy used services like Gmail. Good support for pre-compilation.
Asynchronous Module Definition Loader	RequireJS [12]	Agnostic to the AJAX and DOM library. Very good browser compatibility. No good alternative that meets the previous two criteria.
MV* Framework	Backbone [14]	Works with jQuery. Very stable (the last minor has 5 months) and used in big scale projects (e.g., LinkedIn Mobile App).
CSS preprocessor	None	The features of the available preprocessors didn't outweigh the need for compiling CSS and the debug difficulties that this encompasses.

Table 2. Main libraries used by Pulse Views.

4.3 A Quick Introduction to Backbone

Backbone is a MV* (similar to MVC but without the controller [13]) JavasScript framework that helps developers to structure their client side code. It provides "models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling, and connects it all to your existing API over a RESTful JSON interface." [14].

The most important components of Backbone are Models and Views. Models store the state of the application and Views render that state (while ideally being stateless). There are also Collections and Routers. Collections are lists of models with some convenience methods and Routers facilitate the implementation of navigation between application screens.

4.4 Metadata API

4.4.1 Introduction

The Metadata API is a REST [15] API exposed by the Pulse Server that allows all the aspects of a Pulse App to be accessed and (when possible) changed. This API is composed of a series of resources that are mapped on the client side to Backbone models and collections. The reference of the most relevant resources can be found on Table 3.

Resource URL	Description
/pulseviews/api/apps	Pulse App resource. Contains information like the Pulse Apps name, URL alias, id, security type, author, users associated with it and if it's enabled or not.
/pulseviews/api/apps/{Pulse App ID}/dashboards	Dashboard resource. Contains information like the name, URL alias, id, security type, users associated with it, author id, last author, creation timestamp, last modification timestamp and widget and dashboard configuration.
/pulseviews/api/apps/{Pulse App ID}/kpis	KPIs resource. Contains information like the name, id, window, expression, baselines associated with it and others.
/pulseviews/api/apps/{Pulse App ID}/baselines	Baselines resource. Contains information like the name, id, KPIs associated with it and others.
/pulseviews/api/apps/{Pulse App ID}/alerts	Alerts resource. Contains information like the name, id, expressions associated with it and others.
/pulseviews/api/apps/{Pulse App ID}/datasources	Custom data sources resource. Contains information like the name, id, connection, entity name, columns, filters (i.e., SQL WHERE conditions) and others.
/pulseviews/api/apps/{Pulse App ID}/hierarchies	Hierarchies resource. Contains information like the name, id, associated stream, hierarchy tree and others.

Table 3. Reference of the most relevant Metadata API resources.

4.4.2 Dashboard Configuration Data Model

The dashboard configuration (i.e., the layout and widgets configuration) are stored in the dashboard resource as a JSON [16] string. The reason for this configuration to be stored in an opaque string is that this way the schema of the configuration is defined by the client. This makes altering it much easier. Also it would be quite complex to model and manage the relational schema of the configuration because it is necessary to support custom options and those can have an arbitrary schema that, when using plugins, may only be known in run-time.

The configuration object has tree keys:

- **widgets** – List of individual widget configurations.
- **layout** – Layout tree, dashboard height and maximum node id.
- **options** – Dashboard rendering related options: custom CSS and drill widget options.

Each element of widget has the following keys:

- **type** – The id of the widget.
- **conf** – The current values of the widget options (the structure this object is defined in the WDO).
- **nodeId** – The id of the layout node where this widget is to be rendered.

Each node of the layout tree contains the following keys:

- **id** – The node id.
- **type** – Can be "container" (column) or "element" (row).
- **widthP** – width of the node in percentage of its parent width.
- **heightP** – height of the node in percentage of its parent height.
- **children** – list of nodes that are to be rendered inside the node.

4.4.3 Dashboard Permissions Model

A dashboard can be in one of three permission modes:

- **Just Me** – Just the current user can access the dashboard.
- **Application Users** – All of the Pulse App users can access the dashboard.
- **Specific Users** – Only me and the users specified by me can access the dashboard.

These permissions are implemented using the Pulse Authorization module: each application and each dashboard have a role associated with it.

4.5 Widget Definition Object

The Widget Definition Object (or WDO) is a JavaScript object with a well-defined structure that contains the blueprint for a widget. This object must be wrapped in a module. For an example please see [18].

The WDO has two keys:

- **render** – The function that is called when it's necessary to render an instance of the widget.

- **info** – Metadata of the widget (e.g., name, icon, options). Most of the proprieties can be omitted and defaults (see [17]) will be used. The mandatory proprieties can be found in [18].

The render function receives two arguments:

1. **conf** – Contains all the initial state of the widget. The structure of this object is defined by the widget options.
2. **delegator** – Proxy to the event delegator. Implements the Widget Events API.

And returns an object that has a propriety named *destroy* which is a function that cleans all the state created by the render function.

For more details please refer to the Widget Definition Object documentation [19].

4.6 Layout Manager

4.6.1 Introduction

The layout manager is responsible for rendering the widgets in the correct position and with the correct size. The layout manager has two operation modes:

- **Ready Only** – used by the Viewer. In this mode the user cannot change the layout.
- **Read-Write** – used by the Builder. In this mode the dashboard can be rendered in either a preview mode (see Figure 24) or a layout edition mode where only the layout container and layout authoring controls are shown (see Figure 25).

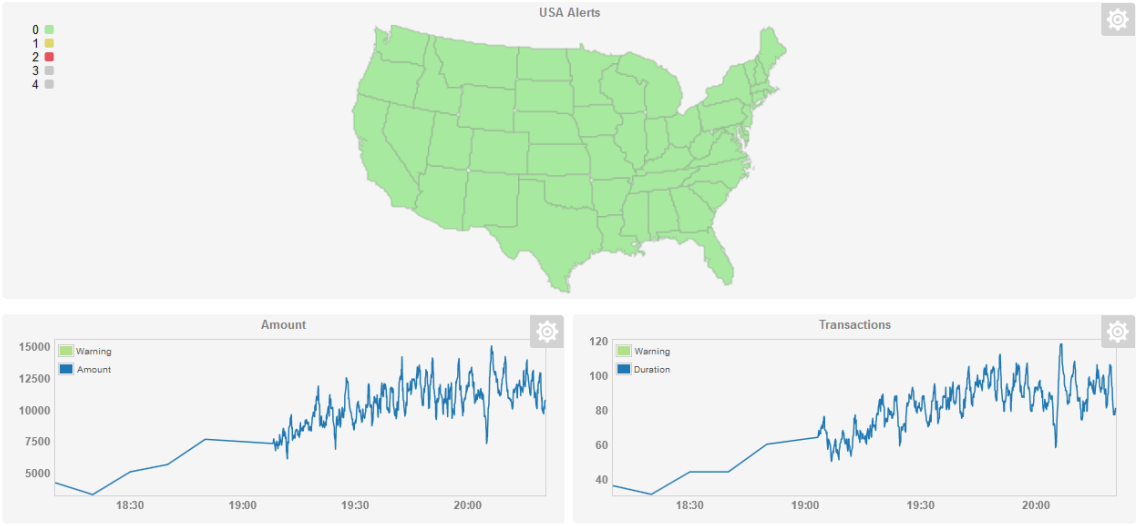
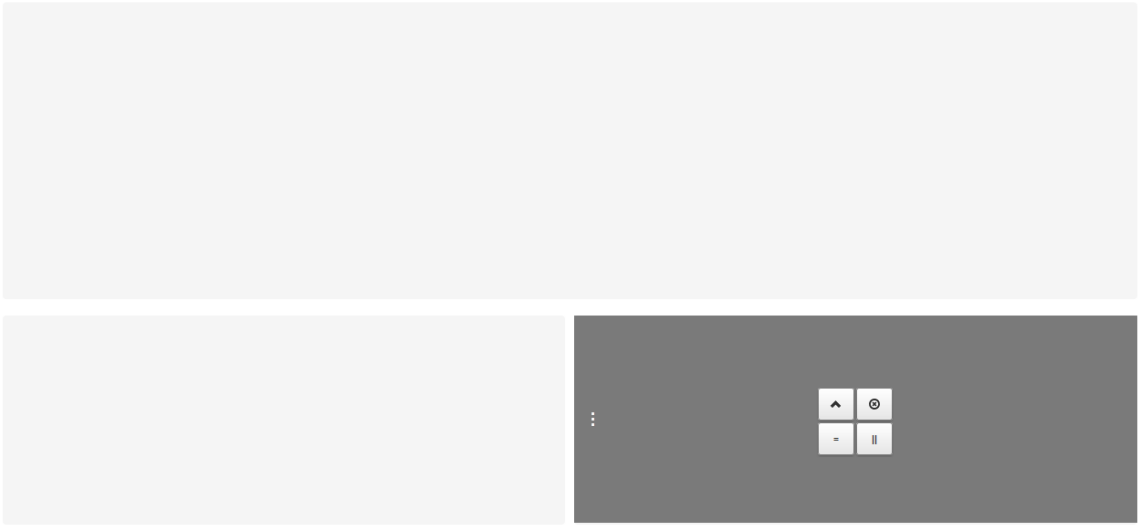


Figure 24. Dashboard in preview mode in the Builder.



...

Figure 25. Dashboard in layout authoring mode in the Builder. The container in the lower right is different because the mouse is over it.

4.6.2 Abstract Layout Model

The Pulse Views layout manager creates layouts by using two types of containers: R (row) and C (column). Both types of containers are similar in the sense that they define a rectangular region and that they can contain other containers inside then with the restriction that they all must be of the opposite type of their parent.

Sibling containers (containers that have the same parent) occupy all the available space in one dimension and use the other as the way to determine their position inside the parent: they are stacked according to their order in the parent's children list. Once stacked and given the margins the containers must occupy all the available space (in both dimensions) inside their parent.

For example, in Figure 26 the containers of type C occupy all the available height and are stacked in the order of their position in their parents' container list. It should be noted that the available width/height is not 100% because the space for the margins needs to be accounted for.

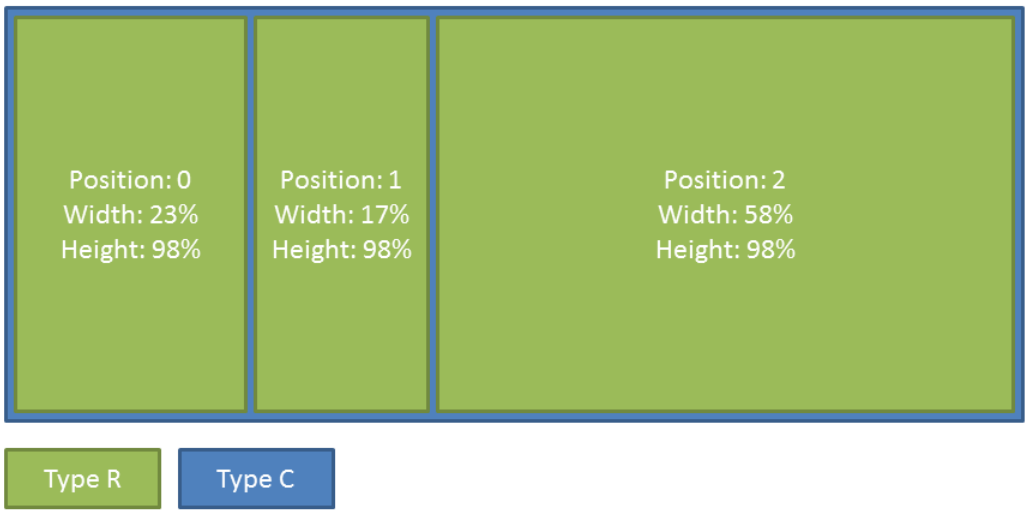


Figure 26. Example of the stack based rendering of containers.

The children of containers of type R (which are containers of type C) are stacked on a row (see Figure 26) and the children of contains of type C (which are containers of type R) are stacked on a column (see Figure 27).

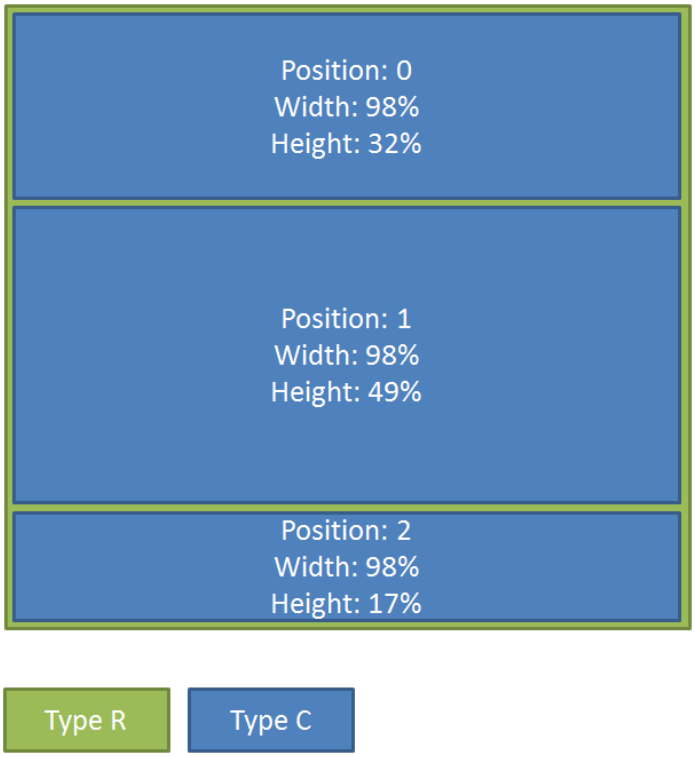


Figure 27. Example of stacking in a column.

The reasons why the layout manager was designed this way are:

- It resembles the way the default HTML layout manager works and thus it made it a little easier to implement (because some of the calculations could be delegated to the HTML renderer).
- The data structure is simple: a tree with multiple nodes; and easy to serialize into JSON.
- The percentage based container dimensions and some of the restrictions made it easy to allow the dashboard to be rendered at different widths.

The dimensions of a container are expressed in percentage and the children of a container can never expand beyond the limits of their parent.

On the root of the layout there must be one or more containers of the type R. Widgets can only be rendered inside containers of type C which enforces that the layout tree leafs are always of this type.

Three transformations can be applied to the containers: add neighbor, split and remove.

Adding a neighbor is simply adding other children to the targets parent. The space available is divided equally by all the children. Both containers of type C and R are valid targets for adding a neighbor.

Splitting involves adding two new containers to the target. The two new containers will split the available space evenly. Containers can be split horizontally or vertically depending on their type.

Both adding neighbors and split operations will be aborted if any of affected containers will have its dimensions below a certain threshold. This prevents containers becoming too small.

Removing a container (of any type) involves several steps (that are done so that the user does not have to do multiple removes):

1. The layout tree is traversed upward until a parent with more than one sibling is found.
2. The node found in 1 (and all its children) will be the one removed.
3. The space occupied in the stack by the node to be removed is passed to its closest sibling.
4. The node is removed.

If removing a given container will lead to the root container to be removed the operation will be aborted. This avoids the user to enter a state where he does not have any more containers available and cannot add more.

Because the margin of the containers must be constant in all layout levels when the dimensions of a container change it necessary to go through all of its children (and the children of the children and so forth) and update their dimensions.

4.6.3 Implementation

This is one of the most complex parts of the system and as such its code is divided in several modules which represent the several components of the layout manager. These modules can be found in Table 4.

Name	Description
layout.view	Core of the layout manager. Contains methods to create nodes and execute all the possible transformations.
layout.model	Manages the memory representation of the layout tree.
can.view	Contains several methods to test if a given transformation can be done without comprising the minimum size thresholds.
layoutBuilder.view	Renders a layout tree into the DOM.
toolbarManager.view	Handles the toolbars life cycle (i.e., when they are shown and removed and what is their target element).
elementToolbar.view	Contains the UI elements that allow adding neighbors.
containerToolbar.view	Contains the UI elements that allow splitting and removing containers. Also allows selecting the parent of the current container.

Table 4. Reference of the layout manager modules.

The toolbars are overlays rendered in the same region of the container that contain the layout transformation and navigation controls. They are shown when the mouse is over a container and when the user uses the go to parent button.

Both the add neighbor, split, remove and go to parent controls are rendered in the UI as buttons. The controls to resize containers are rendered as handlers that can be dragged with the mouse.

The draggable handlers were implemented on top of jQuery UI draggable [20]. This handler behavior was quite hard to implement because the situations where the mouse is moved beyond where the container can be resized to need to be accounted for and jQuery UI does not provide any support in this regard.

Also, for improved usability, when there is enough space the size and padding of the toolbar controls are increased.

4.7 Widget Options

4.7.1 Introduction

The widget options are what enables users to configure the initial state of widgets. This includes styling, choosing the data to display and others. For an example see Figure 28.

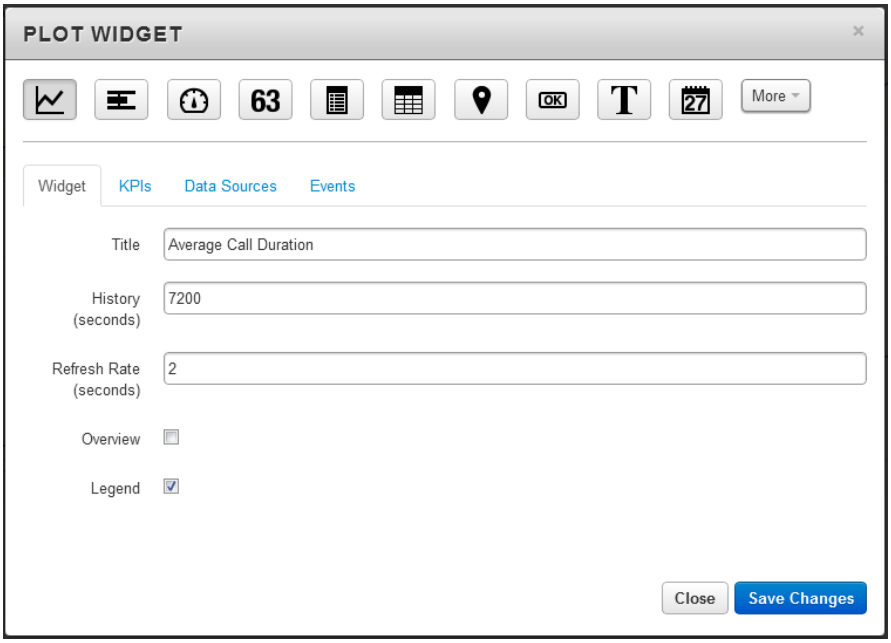


Figure 28. Options of the plot widget.

The options available when configuring a widget can be defined in two ways (which are described in the following sections):

- **Declaratively** – the widget author defines the options in an object literal and the application takes care of rendering and passing them to the widget render function. This approach is less flexible but reduces the development time greatly.
- **Custom** – the widget author has to implement the rendering of the options and the application takes care of passing the values to the widget render function. This approach is much more flexible but it is much harder to develop.

4.7.2 Defining Options Declaratively

The options are defined in an object assigned to the options *propriety* of the *info* propriety of the WDO.

The options object contains a *tabs* propriety which is a list of tabs that will be rendered in the options editor view. For example in Figure 28 there are four tabs: Widget, KPIs, Data Sources and Events. Each tab object contains an id, label (that will be shown to the user), list of options and some flags that indicate if the options should be repeated by KPI or custom data source.

Each option contains an id, label, type (the possible types can be found at [21]), default (that will be used when the widget needs to render before the user has configured it), validations (jQuery validate [22] validations can be used and/or a custom function can be defined), baseline options, possible values (when the type is to be rendered as select box), showme (a function that asserts whenever this options should appear or not) and others.

The complete reference of the tabs and options available proprieties can be found at [23].

Figure 29 shows an example of declarative options configuration. In the example there are two tabs: widget (with only one option) and kpis (with several options that will be repeated for each KPI).

```
{
    tabs : [
        {'id' : 'widget', label : 'Widget', byStream : false, options : [
            {id : 'title', label : 'Title', 'default' : '', type : types.string, validations : {required : false, maxlength: 64}}
        ]},
        {'id' : 'kpis', label : 'KPIs', byStream : true, options : [
            {id : 'label', label : 'Label', 'default' : '', type : types.string, validations : {required : false, maxlength: 25}},
            {id : 'units', label : 'Unit', 'default' : '', type : types.string, validations : {required : false, maxlength: 15}},
            {id : 'max', label : 'Max', 'default' : '', type : types.float, validations : {range: widgetUtils.MINMAX}},
            {id : 'sizes', label : 'Steps (%)', 'default' : [], type : types.array_int, showme : widgetUtils.has('noBaselines')},
            {id : 'markers', label : 'Markers', 'default' : [], type : types.array_float}
        ]}
    ]
}
```

Figure 29. Example of declarative options configuration.

When the options are passed to the widget render function they follow a structure based on the ids of tabs and options. If you have a tab with id *widget* and with the options *title* and *width* the options object passed to the render function will have a *widget* propriety which will be an object with the proprieties *title* and *width*. If the flag for repeating options by KPI or by custom data source is enabled the *widget* propriety will contains and array and each element will be and object with the proprieties *title* and *width*.

4.7.3 Rendering Declaratively Created Options

Each tab defined in the options object will be rendered visually as a tab and the tab header label will the label propriety of the tab object.

Inside a tab each option will be rendered in a line: on the left side of the line will be rendered the label of the option and on the right the option input element. An example can be seen in Figure 28.

When the KPI or custom data source flags is active the options are repeated one time for each selected KPI or custom data source and UI elements for adding/editing/removing KPIs and custom data sources are displayed. If the KPI flag is active and, if there are baseline options defined, they will also be repeated by each baseline in each KPI and if the distinct baseline positions flag is enabled the upper and lower baselines will also have separated options. An example of a tab with the KPI flag enabled and that includes some baselines can be seen in Figure 30. A tab with the custom data sources flag enabled is similar except it does not have baselines.

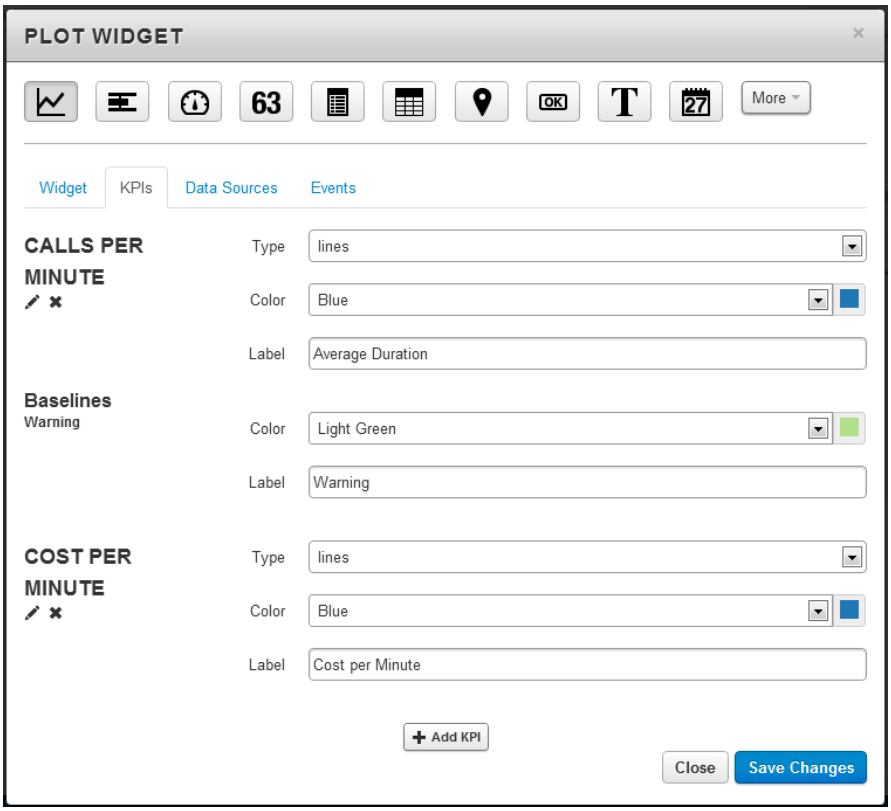


Figure 30. Example of an options tab with the KPI flag enabled.

When the user clicks on the Add KPI button (or the Add Data Source) the screen changes to one where he can configure the KPI to add. In the case of the custom data source the screen only contains a list where the user can choose one of the elements. In the case of the KPI the screen is more complex: it contains a dynamic tree that displays the hierarchy tree and that allows the user the select one node; when the user selects a node the input elements for choosing the KPI, baselines and the filters appear on the right side of the screen. For an example see Figure 31.

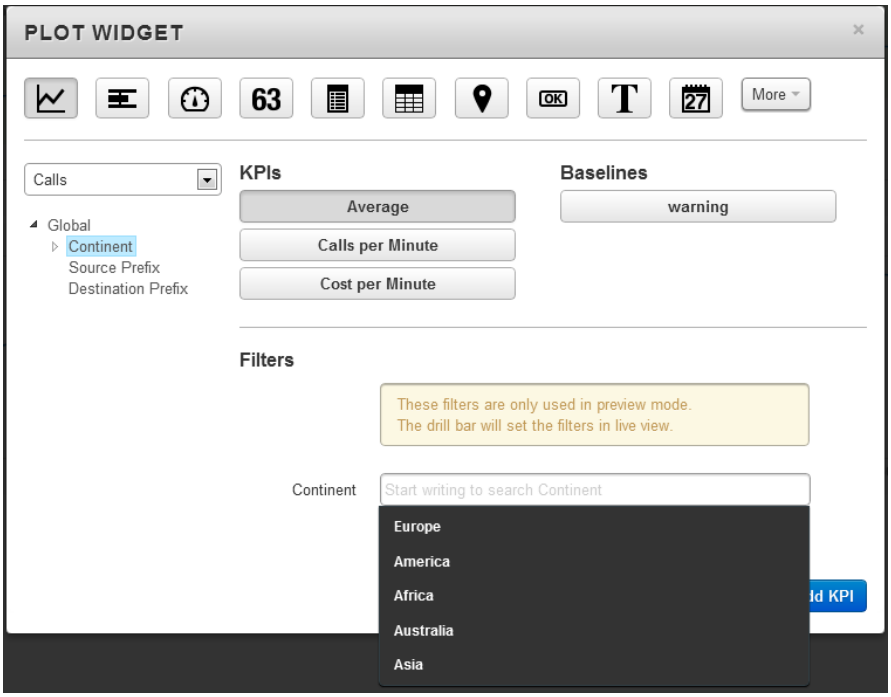


Figure 31. Example of the add KPI screen.

To fill in filters the user has the help of an autocomplete (that displays sever provided possible values) or can himself write the desired value in which case it will be validated if the type is correct (e.g., if the type is Integer a String will be invalid).

Internally the options dialog is rendered in several steps:

1. The modal is created and the backdrop and header are rendered.
2. The list of available widgets is rendered and if this is not the first the selected widget is marked as such.
3. An object containing the current options state is created using an iterator that visits each tab and option.
4. The object created in 3 is passed to the generic options template. This template is composed of several sub templates that render each type of tab and option (filling them with their current value).
5. A series of JavaScript initializations are made:
    a. Validations.
    b. Bindings to the buttons that add/edit/remove KPIs and custom data sources.
    c. Option inputs that require JavaScript (like date pickers)
    d. Truncation of labels where needed and tooltips.

4.7.4 Custom Option View

The custom options views approach allows the developer to completely customize the configuration of a widget. This is done via a Backbone view that is passed to the tabs propriety of the options object and that must implement the interface described in the Table 5. The value of the options is passed in as the model of the view.

Method name	Description	Parameters	Return value
render	Renders the view in its current element.	--	The view instance.
proceed	Method called when the users clicks the *Save Changes* button.	--	An object with the options or false if the current state of the view is not valid (e.g., if an option validation fails)

Table 5. Custom Options View interface.

When developing custom option views developers can use some generic views (see Table 6).

Name	Description
tabs.view	Renders several tabs in a similar way to the declarative approach. Allows each tab to be a custom view.
tab.view	Renders several options in a similar way to the declarative approach. Allows each option to be a custom view.
option.view	Renders an option in a similar way to the declarative approach. Accepts all the parameters applicable to options in tabs without flags with the exception of validations.

Table 6. Reference of the generic custom option views.

4.8 Widget Events

Widgets can communicate with other widgets by emitting and receiving events. This allows, for example, the user to show a chart by clicking on a button.

Who receives what events from whom is defined in the widget configuration in the events configuration tab. In Figure 32 we can see a tabs widget that can receive an event (called an *action*) to change the tab, a button widget that can trigger the event click (called an *event*) and the binding added by the user that connects the click event to the change tab action.

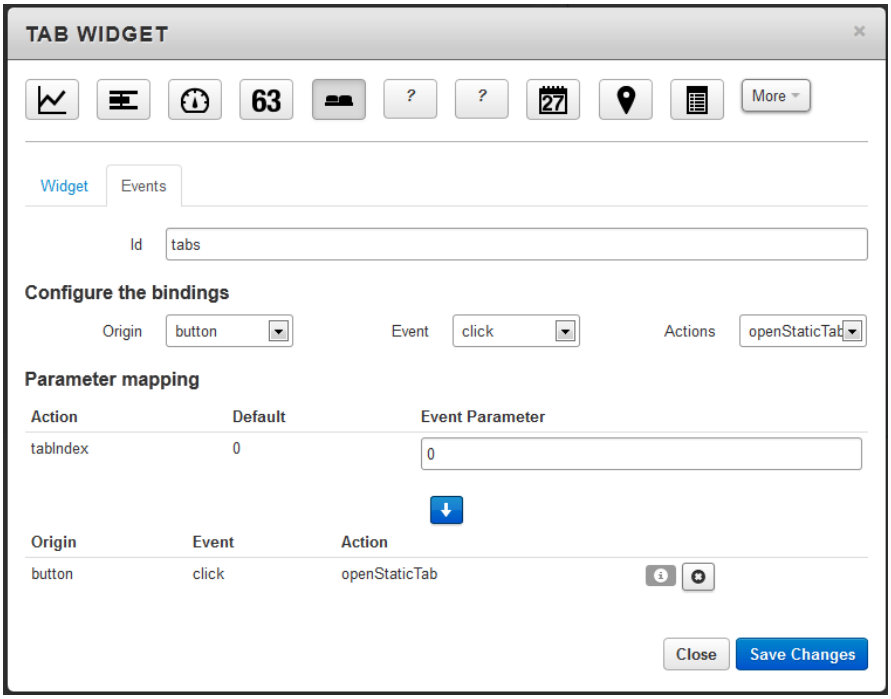


Figure 32. Events configuration tab.

The events also can be used as a way for the widget to communicate with the application. For example, there is an action that the widgets can declare (called *onDashboardLoad*) that is called when the dashboards loads. The binding for this type of actions is done by the application without the need of user intervention.

Both actions and events can have parameters that need to be explicitly declared by the programmer in the WDO. The user is responsible for choosing the mapping between event and action parameters. Optionally the programmer can declare that a parameter should have a static value chosen by the user (via a text input).

The binding and triggering of events is done inside the render function via three methods available in the delegator parameter:

- **on** – Bind to an action.
- **off** – Unbind an action.
- **trigger** – Trigger an event.

An important aspect of the events is that they are only active on the Viewer.

For the complete reference on the events API please refer to [24].

Internally the event delegator is composed of two components:

- **Core** – Wrapper around the Backbone publish/subscribe implementation [25]. There is an instance of the core for each dashboard rendered which allows having several dashboards rendered on the same page (which can happen when using embedded dashboards).
- **Delegaty** – Proxy passed to the render function to deny a widget instance the ability to send events that were not declared, to impersonate another widget or to send events to widgets other than the ones specified by the user. An instance of delegaty is created by each widget instance.

The usage of the publish/subscribe model made disabling of the events in the Builder trivial: it was only necessary to pass to the render a function a dummy object that implemented the API and which methods didn't do anything.

4.9 Embeddable Dashboards

Embeddable dashboards allow a developer to include a Pulse Views dashboard in an arbitrary HTML page. This is quite common in projects where custom dashboards, that include custom elements, need to be created.

There are, however, some restrictions:

- The page must have jQuery included.
- The page must be hosted inside the same Pulse server instance where the dashboard was created (this can be bypassed by using HTML iFrames).
- Authentication is required and it is of the responsibility of the developer.
- Some widgets may render differently when embedded due to absence of Pulse Views CSS styles.

Given the above preconditions the only thing necessary to embed a dashboard is to go the options of the dashboard to embed (see Figure 33) copy the embeddable script, paste it on the desired page and choose the HTML element where the dashboard should render.

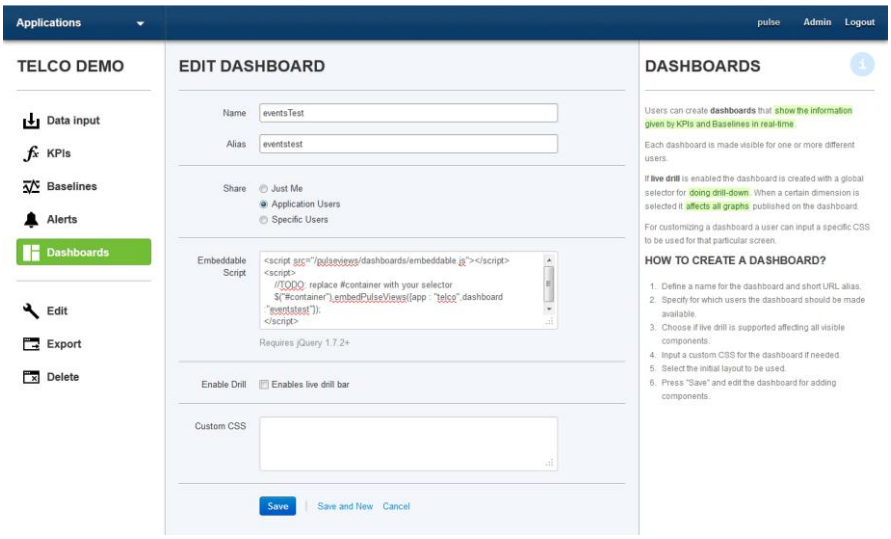


Figure 33. The options screen of a dashboard.

Due to the modular nature of the code (and after some minor changes to the Viewer) the embeddable is simply a wrapper that loads (if necessary) the necessary static dependencies (i.e., libraries like jQuery), does some initializations and then users RequireJS to load and initialize the Viewer with the desired dashboard. Rendering several dashboards in the same page is supported as well as the usage of plugins.

The developer interacts with the embeddable via a jQuery plugin.

4.10 Plugins

In Pulse Views a plugin is a widget authored by a Pulse developer and that, after being loaded, can be used in dashboards in the same way as any other widget. In order for a plugin to be loaded its source needs to be copied to a special folder inside the Pulse Server and the former should be restarted. After that the widget will be available to be used in Pulse Views. For more details please refer to [26].

The biggest downside of the current approach is that if the developer authoring a plugin loads a JavaScript that fails to initialize (due for example to a syntax error) it may cause Pulse Views to not load. This issue was not addressed because removing the problematic widget solves the issue and because only developers with access to the Pulse installation can load widgets.

There are also some public modules that developers can use when developing widgets:

- Option Types – Enumeration of the possible types of options.
- Widget Public API – Collection of utility methods useful when developing widgets.

The complete reference of the public modules can be found at [27].

Internally the loading of plugins is a very simple process:

1. When the Pulse Server loads a list of the files with the extension '.widget.js' in a special folder is compiled.
2. Using the list of widgets to load the code of a special module is dynamically generated so that it has as dependencies all the widgets to load. This module returns a list of references for the plugins.

3. The widget resolver (who loads all the base Pulse Views widgets) has as dependency the special module created at 2. On load it is only necessary to concatenate the list of base widgets with the list of plugins.

Although the architecture of the plugin system was defined in the scope of this project the implementation of the server side component was done by a third party because it needed to be implemented in the backend.

4.11 Drill Widget

The drill widget allows the user to explore the various levels of a hierarchy. It allows him to choose in which level he wants to be and which values he wants to apply to the filters. The widgets will then be re-rendered so that they show the data for the desired level. KPIs of different hierarchies will be unaffected.

A possible use case is: Pulse is calculating a KPI which is the average of the transactions across banks and cities. The drill widget allows the user to explore various banks and cities in a hierarchical way.

To activate the drill widget the user has to go to the dashboard options where it also needs to choose the target hierarchy.

As the drill widget communicates with the other widgets via the widget events it only works on the Viewer. On the Builder it is used to set the initial state of the drill widget.

An example of the usage of the drill widget can be seen in Figure 34.

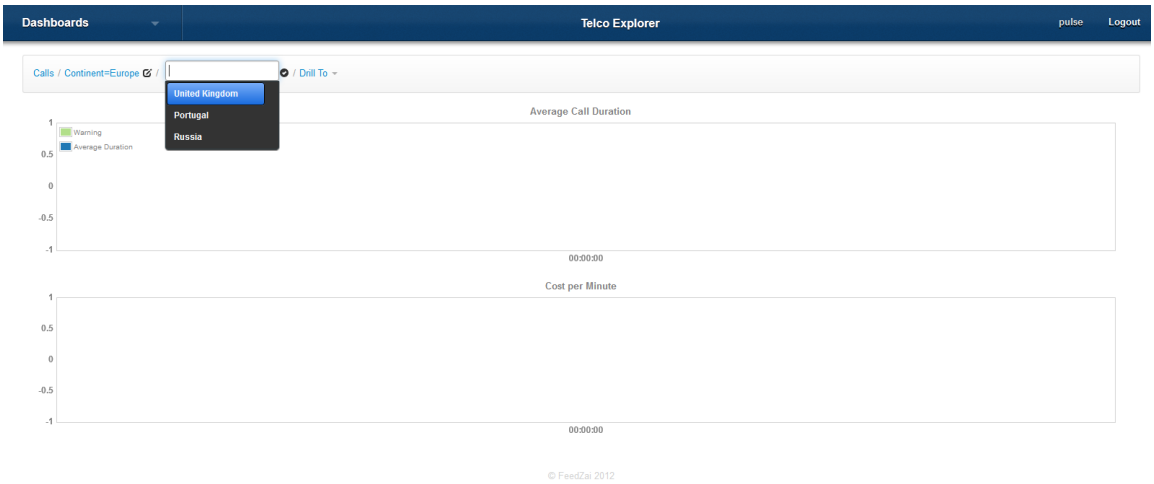


Figure 34. Example of the usage of the drill widget.

Contrary to what the name seems to imply the drill widget is not part of the Widget API. It does not even have a WDO. This approach has chosen because we wanted to render the widget in a predetermined place. Instead of a WDO the drill widget is a Backbone view that uses a Backbone model to store its state. This makes propagating state changes to the widgets (when in the Viewer) easy because is just a matter to listening to changes the model.

The drill widget is rendered as a breadcrumb that represents the current path in the hierarchy. The user can drill up by clicking in any of the levels, drill down by clicking in "Drill To" or change the value of any of the filters.

Changing the value of a filter is done with the help of an autocomplete (that displays sever provided possible values) which is the one provided by jQuery UI [28].

4.12 iOS Support

4.12.1 Introduction

The jPulse components were designed and implemented mostly with desktop browsers in mind and although they work in iOS devices the interaction is limited to the JavaScript events that are common to mobile and desktop browsers (e.g., click). The objective of this task was to enable the iOS users to have the same level of interaction that they have on desktop browsers.

4.12.2 Requirements

After an analysis of the existing components and their behavior on iOS devices a list of broken functionalities was compiled. The list also contained the actions to take and possible solutions. This list was then presented to the Product Owner who ordered the items by priority. The list can be seen on Table 7.

Description	Action	Solution	Priority
Plot			
Panning does not work	Fix	Catch drag & drop event and prevent default	Must
Zoom does not work	Fix	Catch pinch to zoom event and prevent default	Must
Double click does not work	Fix	Catch double tap event and prevent default	Must
Vertical marker does not work	Ignore	-	-
Selection on the overview does not work	Fix	Catch drag & drop event and prevent default	Must
Drill down table			
Selection input fields does not work very well (is very slow and events do not behave well)	Fix	Use native selection input fields	Should
Expand and collapse animation is slow	Fix	Do not use an animation	Should
The scroll bar does not appear	Fix	?	Should
A line on the table has a different color (probably related to the over event)	Ignore	-	-
Tree			
Drag and drop does not work	Fix	Catch drag & drop event and prevent default	Nice to have

The space between element labels in the tree is very small	Increase the padding/margin	Nice to have
The space between arrows is very small and they are very hard to click	Increase the padding/margin	Nice to have

Table 7. Requirements for adapting jPulse components to iOS

4.12.3 Results

All the solutions in Table 7 for the Plot and Drill Down Table widgets were implemented. The solutions for the Tree widgets were not implemented due to its low priority.

An image of the Plot component, running in an iOS device, before the adaptations can be seen in Figure 35 and after in Figure 36. The only visible difference is in the overview chart where the selection areas on the margin of the selected region are more pronounced and easy to target using the fingers. Besides that, all the other interaction mechanisms were implemented – the difference being that they now use the events of the Safari browser on the iOS platform. Also of note is that all of this is transparent for the developer using jPulse: the component knows when it is running on an iOS device and behaves accordingly.

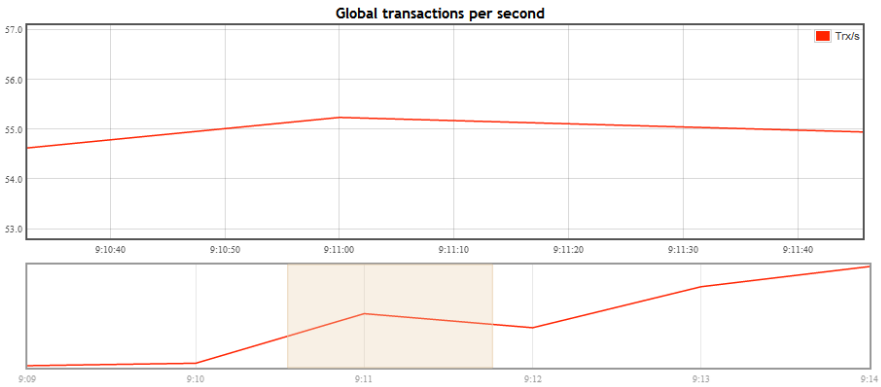


Figure 35. jPulse Plot chart before adaptation.

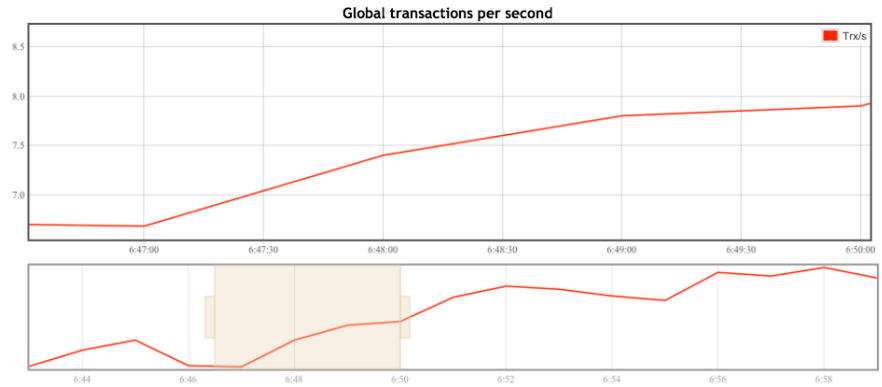


Figure 36. jPulse Plot chart after adaptation.

An image of the Drill Down Table components, running in a iOS device, before the adaptations can be seen in Figure 37 and after in Figure 38. The difference here is that the default HTML select element is now used as the usability is much higher that the custom element previously used. This was no simple task, as the whole component was not designed having in mind that a different selection mechanism might be needed. Some style changes were also applied to improve readability on devices with small screens.

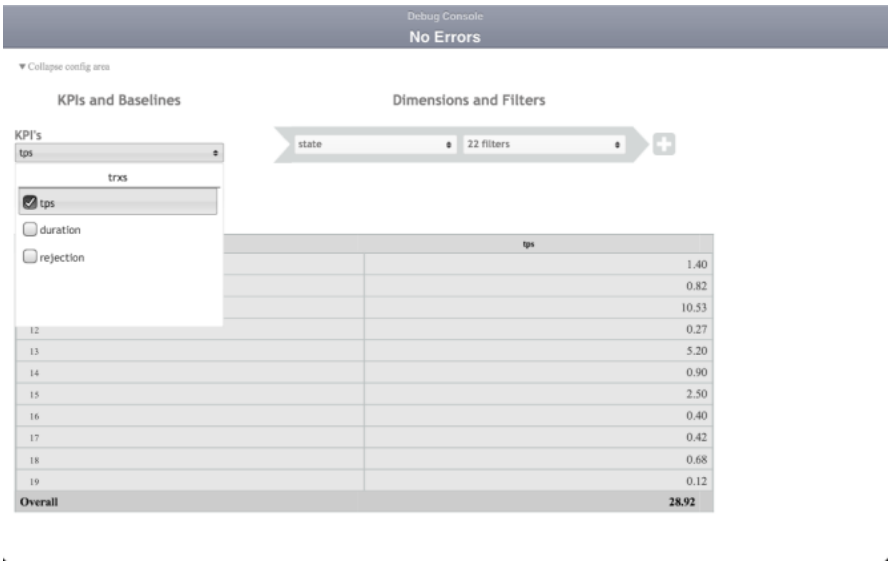


Figure 37. jPulse Drill Down Table before the adaptations.

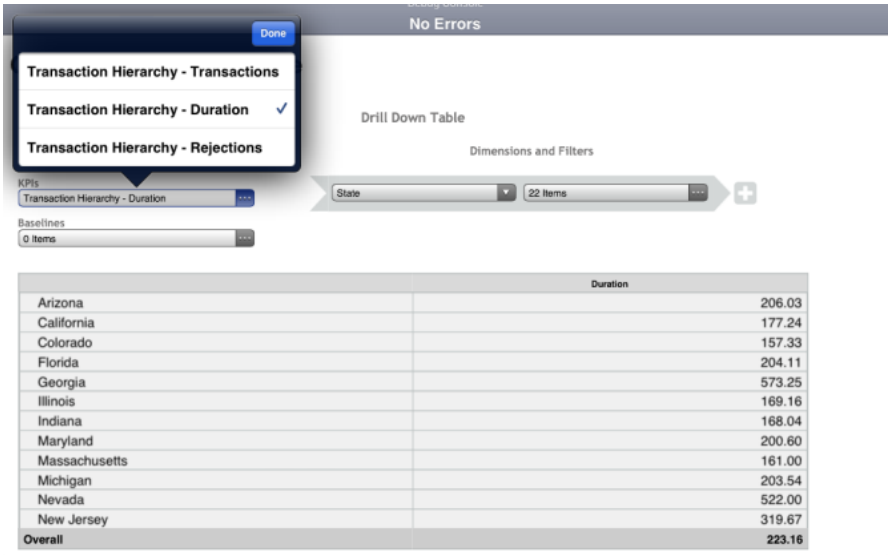


Figure 38. jPulse Drill Down Table after the adaptations.

The implementation of the Drill Down adaptation involved the re-implementation of all the eleven methods of the object responsible for drawing the select box. This task was even more difficult because due to the lack of isolation: the internal state of the object was directly accessed from the outside.

Chapter 5 - Results

5.1 Introduction

In this chapter the final results will be shown including examples of dashboard implemented with Pulse Views, widgets implemented in this projects and validation.

5.2 Examples

5.2.1 Dashboards

5.2.1.1 Introduction

In this section two dashboards created with Pulse Views will be presented. These dashboards represent uses cases that Pulse Views targets.

5.2.1.2 Banking Demo

The banking demo dashboard provides an overview over the operation off a (fictitious) USA bank. It allows, at a quick glance for users, to have a global vision over the transactions done with several card types, at the transactions by state, at the tendency of the transactions by second globally and at the various types of operations. An image of this dashboard can be seen in Figure 39.

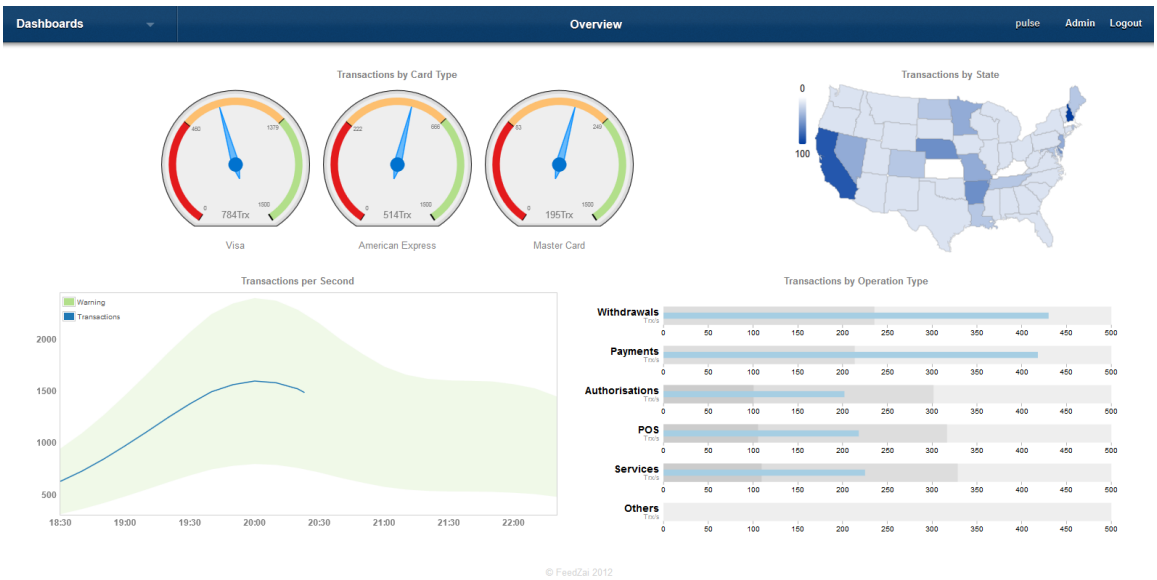


Figure 39. Banking demo dashboard image.

5.2.1.3 Telco Demo

The telco demo dashboard provides an overview over the operation off a (fictitious) international telecommunications company. It allows for users to see to amount of calls by continent, the evolution of the global number of calls, if there was any deviation from the norm and the average call duration and cost by continent. An image of this dashboard can be seen in Figure 40.

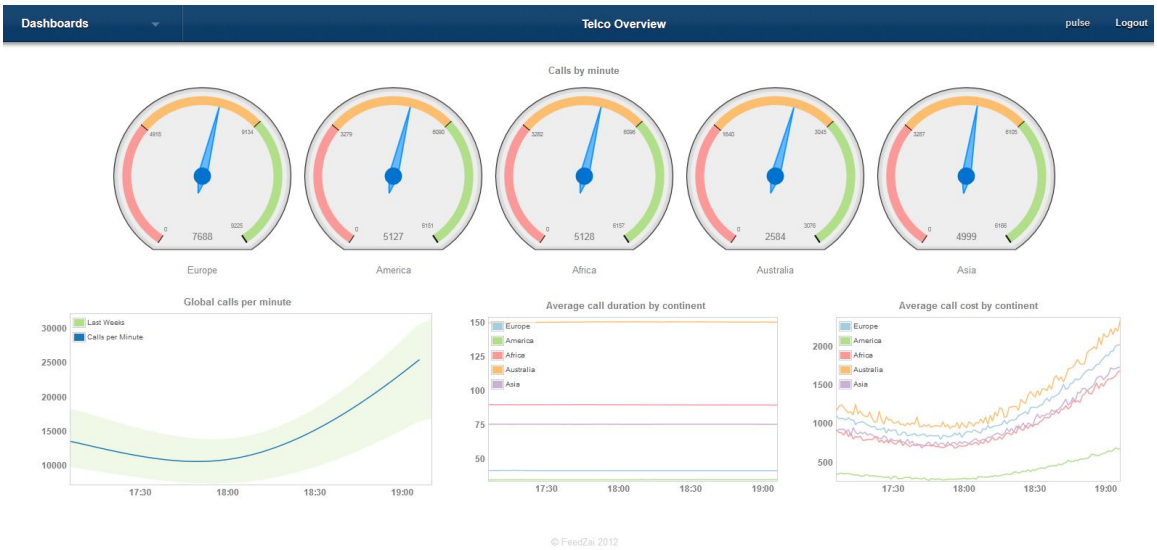


Figure 40. Telco demo dashboard image.

5.2.2 Widgets

5.2.2.1 Plot

The plot widget is used to represent the evolution of a set of numerical series trough time (see Figure 41 for an example). The plot widget can plot both real time data and custom data source data. The support for custom data source data was implemented outside of the scope of this project by a third party.

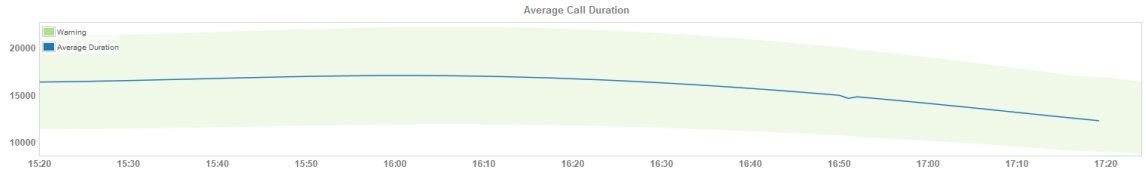


Figure 41. Example of an instance of the plot widget showing real time data.

Regarding the implementation for real time data, the user can plot several KPIs and baselines on the same chart. There are also several options available to configure including: title of the plot, amount of history shown, overview, legend, color of the KPIs and baselines, labels of the KPIs and baselines and type of representation for the KPI (e.g., lines, points).

The plot for real time data is a wrapper around the existing jPulse.plot component. Using the existing component allowed for huge savings in implementation time but there were difficulties, the major ones being: lack of support for axis labels and tweaking necessary to the settings to achieve the desired aspect (around 50 lines of configuration code).

5.2.2.2 Gauge

The gauge widget is used to represent a single value that corresponds to the current value of a given metric (see Figure 42 for an example). The gauge is only available for real time data as it only makes sense for this use case. This widget is similar in function to the bullet and text.

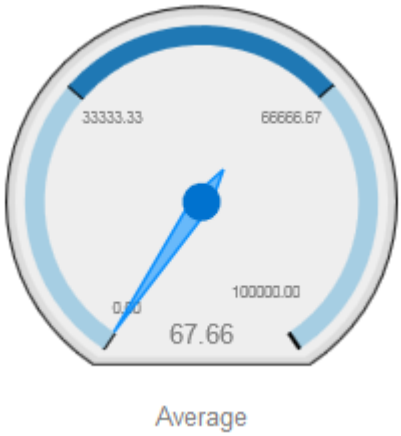


Figure 42. Example of the gauge widget.

The ranges can be statically defined by the user or can represent baselines.

Other options possible to configure are: the title of the widget, the label of the gauge, the global range of the gauge, the unit label, the number of decimals, the default color for ranges (will be used in zones outside of the defined ranges) and the color of each range.

Also the user can have multiple gauges in the same widget simply by adding more KPIs (see Figure 43).

Figure 43. Example of multiple gauges in the same widget.

The bullet implementation is a wrapper around the existing jPulse component. It should be noted that the multiples behavior is implemented by the wrapper and not jPulse.

5.2.2.3 Bullet

The bullet widget is used to represent a single value that corresponds to the current value of a given metric (see Figure 44 for an example). The bullet is only available for real time data as it only makes sense for this use case. This widget is similar in function to the gauge and text.

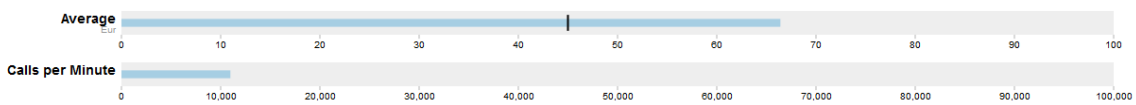


Figure 44. Example of the bullet widget with multiples.

As is the case with the gauge widget, the bullet supports rendering multiple bullets in a widget (one by KPI).

The options available for configuration are: widget title, KPI label, KPI unit, KPI max and KPI markers. Markers are vertical black lines rendered at fixed positions that represent an important threshold. An example of a marker can be seen in Figure 44 in the Average KPI.

The bullet implementation was an adaptation of the one found on [29]. The communication with pulse and the support for an arbitrary number of bullets and arbitrary boundaries was implemented by the student.

5.2.2.4 Text

The text widget is used to represent a single value that corresponds to the current value of a given metric (for an example see Figure 45, top right widget). The text widget is only available for real time data as it only makes sense for this use case. This widget is similar in function to the bullet and gauge, albeit it does not support multiples or baselines.

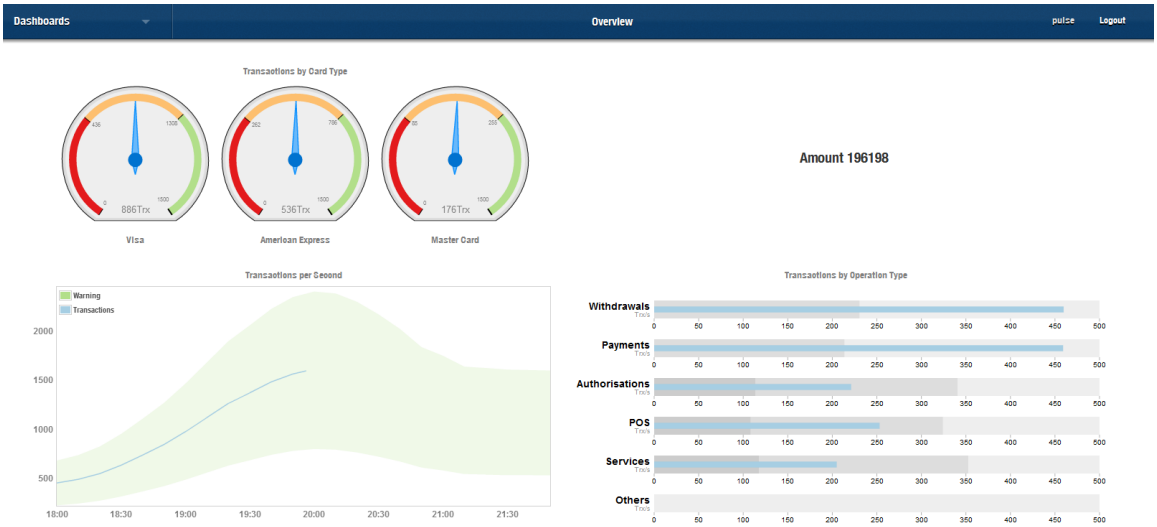


Figure 45. Example of the text widget.

The available configurations are: font size, label, unit label and number of decimals.

5.2.2.5 Tabs

The tabs widget allows the user to have tabs where otherwise he would have a widget. Each tab of the tabs widget can contain any widget (including the tab widget itself). For an example see Figure 46.

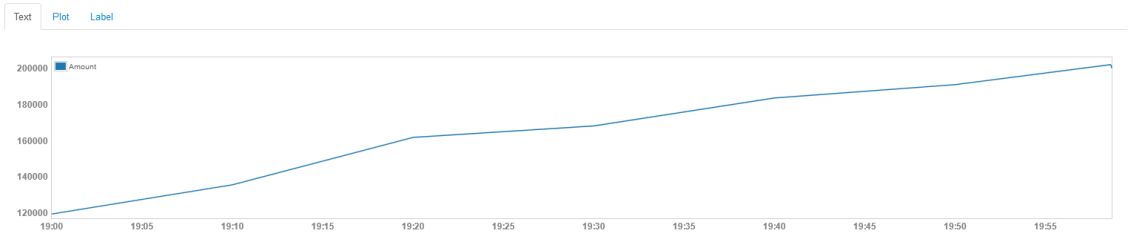


Figure 46. Example of the tabs widget.

The user can change the label of each tab header, what tab is open by default, the number and tabs and if the tab header should be shown or not. The tabs widget also implements actions that allow other widgets to change the displayed tab.

Internally the tabs widget is relatively simple, because it reuses the Widget API code to render the widgets inside each of its tabs.

5.2.2.6 Map

The objective of the map widget is to render geographic regions coloring each one according to a metric. An example can be seen in Figure 47.

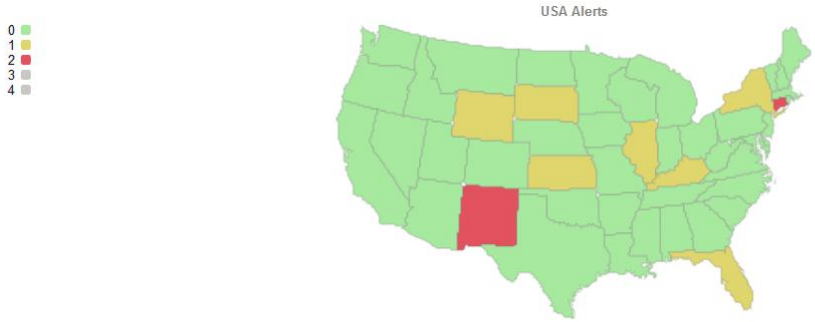


Figure 47. Example of the map widget.

A map widget was something that was on the roadmap from early on, and its development has seen several iterations (which are described in Annex F). The last iteration resulted in the widget currently in Pulse Views.

The map widget currently deployed in Pulse views is quite powerful: it colors regions based on KPIs (with and without baselines), custom data sources and alerts; it has maps with the main regions of all the world countries, the world and the continents. It allows the user to drill from world level to country level, has legends specific to the type of data being shown (see Figure 48) and tooltips. It also allows for configuring the widget title, the unit label of the tooltip and for custom mappings (to use when the source data region format is not the same used internally by the map widget).

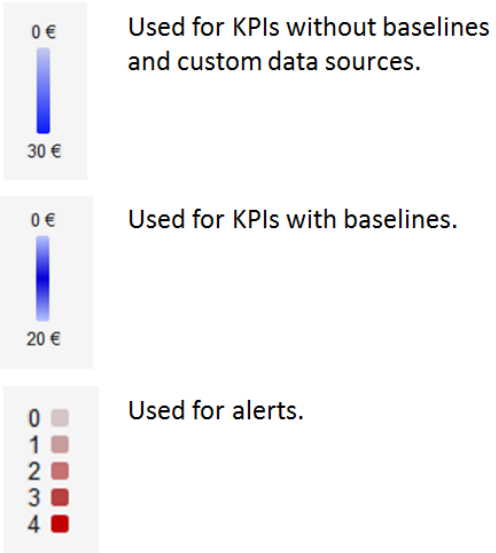


Figure 48. Different types of legends for the map widget.

The available options and the layout of the configuration view were designed in collaboration with key FeedZai stakeholders and an external user experience expert.

Internally the configuration view is implemented using the custom option view feature, which allowed for much more flexibility albeit at the cost of increased complexity: the

custom option view is implemented across 15 modules and has about 1060 lines of JavaScript code.

5.2 Tests

5.2.1 Introduction

The project was validated using unit, UI and compatibility tests.

5.2.2 Unit

The unit tests were implemented using BusterJS [30]. This framework was chosen because it allows for easily running tests in captured or headless browsers as well as in the server. This is good because it facilitates using BusterJS in continuous integration and it facilitates testing in multiple browsers. Although BusterJS has all those features the tests described in this section were run in the browser, as the overhead of configuring a more advanced setup didn't offer, yet, enough benefits.

The most heavily unit tested component of Pulse Views is the Widget API: all the modules that have methods are tested to some degree. The reason behind this is that unit testing UI components is quite complex even more if they are currently changing (which has the case).

In total there were implemented 169 tests which cover 11 modules. A more detailed look at the number of tests can be found in Table 8. The coverage is not reported because it's not supported yet in the framework used.

Module	Component	#tests	#methods	#total methods
widget.optionTypes.utils.js	Widget API	3	3	4
widget.render.js	Widget API	1	1	1
widget.resolver.js	Widget API	11	2	4
widget.utils.js	Widget API	69	19	33
optionsIterator.test.js	Widget API	1	1	1
layout.view.js	Builder/Viewer	8	2	41
widget.events.js	Widget API	27	6	7
widget.instanceManager.js	Widget API	9	9	10
functional.utils.js	Widget API	12	12	22
widget.events.delegaty.js	Widget API	29	12	13
layout.utils	Builder/Viewer	4	1	2

Table 8. Unit tests coverage.

5.2.3 UI

User interface tests were implemented by the quality assurance team using Selenium [31]. These tests are run on top of a live Pulse Views instance that is connected to a live Pulse Server and their cover high level aspects like navigation, interaction between different parts of the application and others.

Selenium tests mimic a human browser in the sense that they interact with the web application by the same means (at least as far as the web application is concerned). Assertions are done by inspection the DOM.

Two types of tests were implemented:

- **Functional** – they test whenever or not the functionality is working. There were implemented 189 functional tests.
- **Regression** – they test if bugs already resolved become bugs again. There were implemented 134 regression tests based on issues found and being tracked.

5.2.4 Compatibility

There were two main mechanisms in place to ensure that Pulse Views has compatible with the target browsers defined in the requirements (IE9, Chrome and Firefox 10):

1. During the development of a feature it would be manually tested by the developer in the three browsers. To a feature to be finished all the functionality that it encompasses must be present in all the target browsers.
2. At the end of a release cycle the quality assurance team would manually test the implemented features in the target browsers with the focus of finding bugs in less common situations.

5.3 Documentation

Both the Widget API and the dashboard builder usage were extensively documented. The documentation of the Widget API can be found at [32] and the user manual for the dashboard builder at [33].

Chapter 6 - Work Plan and Methodology

6.1 Introduction

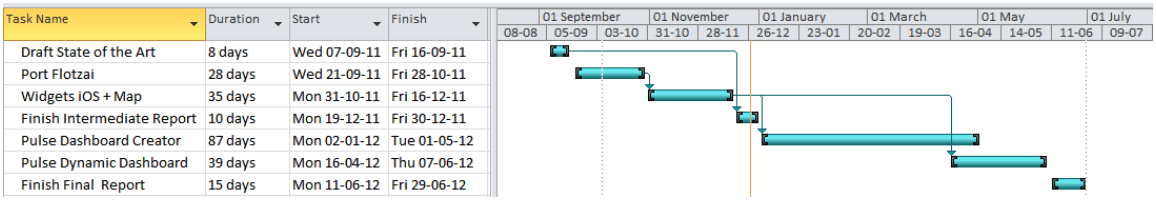
This chapter explains how (and why) the planning evolved during the project and the used development methodologies.

As the internship planning was quite different in the two semesters there is a separated overview of the first and second semesters.

6.2 First Semester

6.2.1 High Level Planning

At the beginning of the first semester a high level plan was elaborated, having in mind the tasks that, at the time, comprised the product backlog. This plan assumed that all tasks would be performed sequentially and that the tasks or the nature of the tasks would not vary too much. Said plan can be seen in Figure 49.



Task Name	Duration	Start	Finish
Draft State of the Art	8 days	Wed 07-09-11	Fri 16-09-11
Port Flotzai	28 days	Wed 21-09-11	Fri 28-10-11
Widgets iOS + Map	35 days	Mon 31-10-11	Fri 16-12-11
Finish Intermediate Report	10 days	Mon 19-12-11	Fri 30-12-11
Pulse Dashboard Creator	87 days	Mon 02-01-12	Tue 01-05-12
Pulse Dynamic Dashboard	39 days	Mon 16-04-12	Thu 07-06-12
Finish Final Report	15 days	Mon 11-06-12	Fri 29-06-12

Figure 49. The original work plan.

What actually happened can be seen in Figure 50.

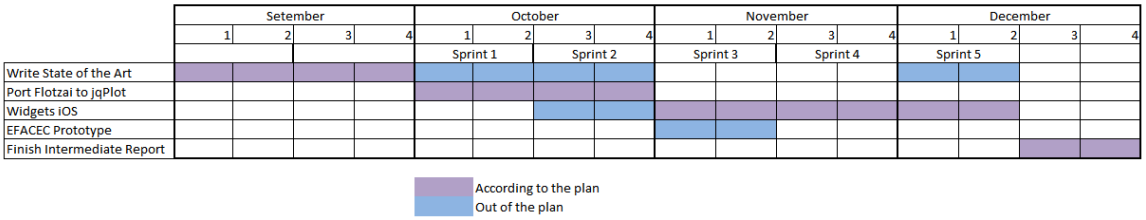


Figure 50. High level view of how the work progressed.

As you can see most of the time the initial plan was followed with three exceptions:

- The state of the art took a bit more than was expected – this was mainly because the start-up time was not considered in the initial estimative.
- The Widgets iOS task started earlier – this was because the Product Owner decided to drop the *Port Flotzai to jqPlot* task.
- A new task (EDP Dashboard Prototype) was created – more details on Annex E.

6.2.2 Operational Planning

During the first semester there were 5 sprints (with two week duration). The sprints calendar can be found on Table 9. The first sprint starts only at 06-10 due to a start-up period of almost a month. More details on the tasks can be found in Annex D.

Sprint #	Start	End	Tasks
1	06-10-2011	21-10-2011	Migration from flot to jqPlot.
2	26-10-2011	04-11-2011	flot to jqPlot benchmark. iOS adaptations.
3	07-11-2011	18-11-2011	EDP dashboard prototype.
4	23-11-2011	02-12-2011	iOS adaptations.
5	07-12-2011	16-12-2011	iOS adaptations. Intermediate Report.

Table 9. Sprints calendar for the first semester.

All artifacts related to planning can be found on Annex D.

6.3 Second Semester

6.3.1 High Level Planning

At the beginning of the second semester the Pulse Dashboard Creator task was started but it was soon realized that it made much more sense to merge the Pulse Dashboard Creator and Pulse Dynamic Dashboard into a web based unified product: Pulse Views. The development of Pulse Views was divided in two high level phases: first the requirements, technologies and high level architecture were defined and a proof of concept was implemented; secondly Pulse Views was implemented and the architecture components were defined as needed. The revised plan can be seen in Figure 51. The goals of the new high level plan were achieved.

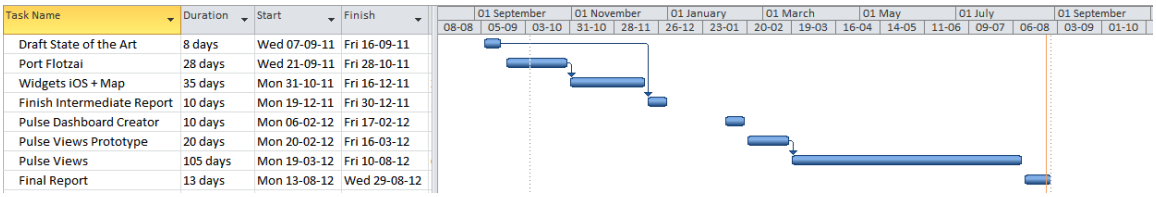


Figure 51. The work plan revised to include Pulse Views.

6.3.2 Operational Planning

During the second semester there were 10 sprints. The duration of the sprints changed to three weeks because the student was integrated in the Pulse development team and they do sprints of that length. The sprints calendar can be found on Table 10.

Sprint #	Start	End	Tasks
6	06-02-2012	17-02-2012	Pulse Dashboard Creator.
7	20-02-2012	02-03-2012	Pulse Views prototype.
8	05-03-2012	16-03-2012	Pulse Views prototype.
Pulse Dev 19	19-03-2012	06-04-2012	Dashboard builder.
Pulse Dev 20	09-04-2012	27-04-2012	Dashboard builder.
Pulse Dev 21	30-04-2012	11-05-2012	Layout manager, tab widget, tests.

Pulse Dev 22	14-05-2012	08-06-2012	Dashboard builder, embeddable, drill widget.
Pulse Dev 23	11-06-2012	03-07-2012	Bullet, gauge, plot and text widget.
Pulse Dev 24	09-07-2012	20-07-2012	Consolidation, documentation and plugins.
Pulse Dev 25	23-07-2012	10-08-2012	Map widget and custom options.

Table 10. Sprints calendar for the second semester.

All artifacts related to planning can be found on Annex D.

6.4 Methodology

6.4.1 Scrum

The work of this project follows the Scrum agile methodology which is "a framework structured to support complex product development. Scrum consists of Scrum Teams and their associated roles, events, artifacts, and rules." [34, pp.5].

In the Scrum framework there are three types of roles, five types of events and tree types of artifacts. Consult Table 11 and Figure 52 for more details.

Name	Description
Roles	
Product Owner	"(…) responsible for maximizing the value of the product and the work of the Development Team" [34, pp.5]
Development Team	"(…) consists of professionals who do the work of delivering a potentially releasable Increment of "Done" product at the end of each Sprint" [34, pp. 6]
Scrum Master	"(…) responsible for ensuring Scrum is understood and enacted" [34, pp. 6]
Events	
The Sprint	"(…) time-box of one month or less during which a "Done", useable, and potentially releasable product Increment is created" [34, pp. 8]
Sprint Planning Meeting	"The work to be performed in the Sprint is planned at the Sprint Planning Meeting" [34, pp. 9]
Daily Scrum	"(…) 15-minute time-boxed event for the Development Team to synchronize activities and create a plan for the next 24 hours." [34, pp. 10]
Sprint Review	"A Sprint Review is held at the end of the Sprint to inspect the Increment and adapt the Product Backlog if needed." [34, pp. 11]
Sprint Retrospective	"The Sprint Retrospective is an opportunity for the Scrum Team to inspect itself and create a plan for improvements to be enacted during the next Sprint" [34, pp. 11]
Artifacts	
Product Backlog	"(…) ordered list of everything that might be needed in the product and is the single source of requirements for any changes to be made to the product." [34, pp. 12]
Sprint Backlog	"(…) set of Product Backlog items selected for the Sprint plus a plan for delivering the product Increment and realizing the Sprint Goal." [34, pp. 14]
Increment	"(…) sum of all the Product Backlog items completed during a Sprint

and all previous Sprints." [34, pp. 14]

Table 11. The roles, events and artifacts of Scrum.

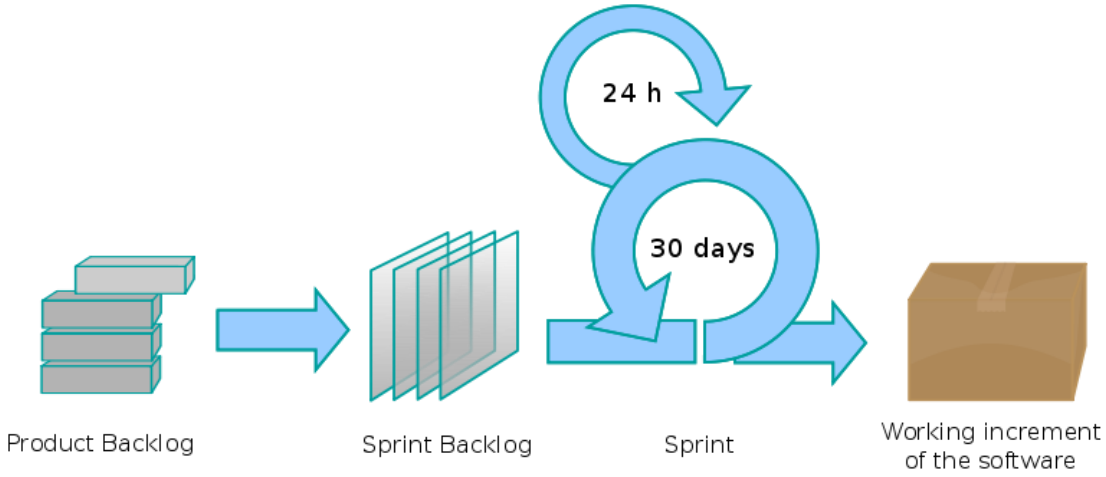


Figure 52. The Scrum Process [35].

At the core of Scrum are three base principles: transparency, inspection and adaptation [34, pp. 4]. These principles are enforced by a set of events and artifacts.

Scrum is used because it's what is used at Feedzai and because as most of the tasks are inside a very dynamic context, requirements may change and some tasks may need to be dropped. This rules out the classic Waterfall models and favors more of an agile approach.

6.4.2 Roles

- Product Owner – Paulo Marques (FeedZai's CTO).
- Scrum Master – Olga Filipova (the mentor at FeedZai).

6.4.3 Events

- The Sprint – The sprints had the duration of two and three weeks.

6.4.4 Artifacts

There are two additions relative to what is defined in Table 11:

- 15/5 Reports – These are weekly reports sent to Olga Filipova, Paulo Marques and Tiago Baptista. They shortly describe what was done during the week, what is planned for next week and any problems encountered. The reason because they are used is the increase of transparency and inspection.
- Burn-down charts – Typical burn-down charts in hours.

The Sprint Backlog and Burn-down charts artifacts are created and maintained by the JIRA platform [36] (which is the one used internally by FeedZai).

6.5 Version Control System

This project used Git [37] as a version control system. The branching model used is the one adopted at FeedZai. It is composed of five types of branches: master (latest stable version), develop (development main branch), features (for new features), quality (for quality assurance), hot fixing (for bug fixing, one branch for minor release). An image of the branching model can be seen on Figure 53.

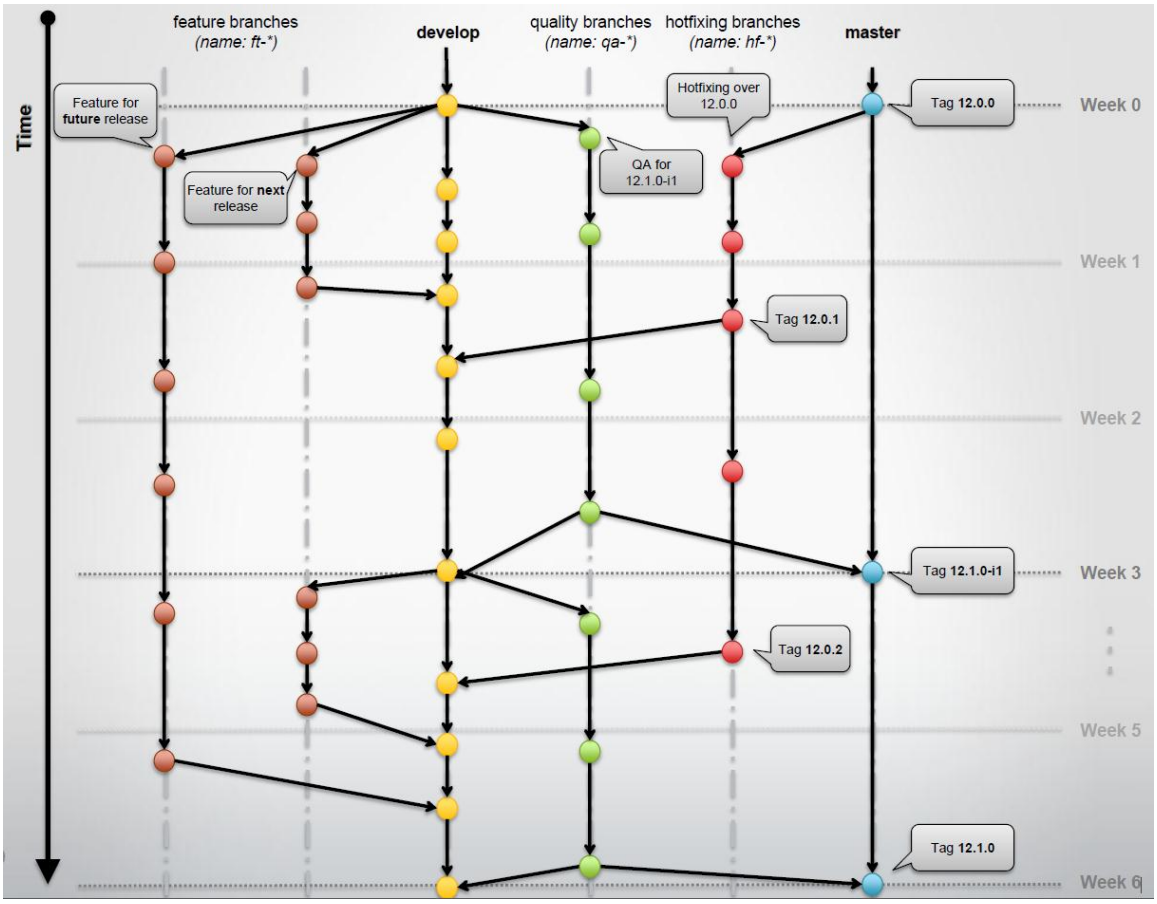


Figure 53. Git branching model. Source: FeedZai internal presentation.

Chapter 7 - Conclusions and Future Work

Looking back to all that was done the conclusion is that this project was a success: Pulse Views was implemented and it allows for users to create and view Pulse dashboards and to developers to create new widgets and use Pulse Views dashboards in custom pages. More than that, it was integrated in Pulse 12.1.0 and is now used by several clients.

This internship was an excellent opportunity to try the fun and hard work spirit that defines most of the successful high-tech startups; it dealt with very interesting subjects, namely data visualization, JavaScript and HTML5 and it allowed for the personal and professional growth that a company as FeedZai provides.

In terms of future work, there are several aspects of Pulse Views that can be improved:

- The performance of the layout manager for complex layouts is not ideal: resizing a dashboard with more than 40 containers is hardly fluid.
- The number of AJAX requests done to the metadata API could be decreased in some cases if a local cache was used.
- There is quite a lot of CSS and templates code that is no longer used by the application: finding and removing this code would improve the load times.
- Compared to other dashboard solutions in the market Pulse Views still lacks several widgets (like a heat map).
- It's possible that the layout edition could be simplified if the split concept was removed (the split would still be done, just implicitly). In this case the only creation transformation primitive would be adding a neighbor.
- Being able change the container of a widget by drag and drop.
- Undo/Redo in the Builder.

References

1. FEW, S. 2006. Information Dashboard Design, O'Reilly.
2. MICROSTRATEGY. MicroStrategy Mobile Suite. http://www.microstrategy.com/freemobilebi/. Accessed on December 19, 2011.
3. QLIKTECH. QlikView. http://www.qlikview.com/. Accessed on December 19, 2011.
4. TABLEAU SOFTWARE. Tableau Desktop. http://www.tableausoftware.com/products/desktop. Accessed on August 13, 2011.
5. SPLUNK. Splunk. http://www.splunk.com/. Accessed on August 13, 2011.
6. FEEDZAI. HTTP Output Adapter. https://docs.feedzai.com/display/pulse/HTTP+Output+Adapter. Accessed on August 14, 2012.
7. FEEDZAI. jPulse. https://docs.feedzai.com/display/pulse/jPulse. Accessed on August 14, 2012.
8. FEEDZAI. Real Time Dashboarding. https://docs.feedzai.com/display/pulse/Real+Time+Dashboarding. Accessed on August 14, 2012.
9. THE JQUERY FUNDATION. jQuery. http://jquery.com/. Accessed on August 15, 2012.
10. TWITTER. Twitter Bootstrap. http://twitter.github.com/bootstrap/. Accessed on August 15, 2012.
11. GOOGLE. Google Closure Templates. https://developers.google.com/closure/templates/. Accessed on August 15, 2012
12. BURKE, J. RequireJS. http://requirejs.org/. Accessed on August 15, 2012.
13. OSMANI, A. Understanding MVC And MVP (For JavaScript And Backbone Developers). http://addyosmani.com/blog/understanding-mvc-and-mvp-for-javascript-and-backbone-developers/. Accessed on August 28, 2012.
14. DOCUMENT CLOUD. Backbone. http://backbonejs.org/. Accessed on August 15, 2012.
15. FIELDING, T.F. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine.
16. CROCKFORD, D. JSON. http://www.json.org/. Accessed on August 16, 2012.
17. FEEDZAI. Default WDO Values. https://docs.feedzai.com/display/pulse/Default+WDO+Values. Accessed on August 15, 2012.
18. FEEDZAI. Basic Widget Example. https://docs.feedzai.com/display/pulse/Basic+Widget+Example. Accessed on August 17, 2012.
19. FEEDZAI. Widgets API Widget Definition Object. https://docs.feedzai.com/display/pulse/Widgets+API#WidgetsAPI-WidgetDefinitionObject. Accessed on August 20, 2012.
20. THE JQUERY FUNDATION. jQuery UI Draggable. http://jqueryui.com/demos/draggable/. Accessed on August 20, 2012.
21. FEEDZAI. Option Types. https://docs.feedzai.com/display/pulse/Option+Types Accessed on August 20, 2012.

22. ZAEFFERER, J. Validation. http://docs.jquery.com/Plugins/Validation. Accessed on August 21, 2012.
23. FEEDZAI. WDO Info Object Nested Proprieties. https://docs.feedzai.com/display/pulse/WDO+Info+Object+Nested+Proprieties#WDOInfoObjectNestedProprieties-Options. Accessed on August 20, 2012.
24. FEEDZAI. Widgets API Events API. https://docs.feedzai.com/display/pulse/Widgets+API#WidgetsAPI-EventsAPI
25. DOCUMENT CLOUD. Backbone Events. http://documentcloud.github.com/backbone/#Events. Accessed on August 21, 2012.
26. FEEDZAI. Widgets API Importing Custom Widgets. https://docs.feedzai.com/display/pulse/Widgets+API#WidgetsAPI-ImportingCustomWidgets. Accessed on August 21, 2012.
27. FEEDZAI. https://docs.feedzai.com/display/pulse/Widgets+API#WidgetsAPI-ImportantModules. Accessed on August 21, 2012.
28. THE JQUERY FUNDATION. jQuery UI Autocomplete. http://jqueryui.com/demos/autocomplete/. Accessed on August 23, 2012.
29. BOSTOCK, M. D3js Bullet. http://mbostock.github.com/d3/ex/bullet.html. Accessed on August 23, 2012.
30. JOHANSEN, C., AND LILLEAAS, A. BusterJS. http://busterjs.org/. Accessed on August 23, 2012.
31. SELENIUM. Selenium. http://seleniumhq.org/. Accessed on August 24, 2012.
32. FEEDZAI. Widgets API. https://docs.feedzai.com/display/pulse/Widgets+API. Accessed on August 25, 2012.
33. FEEDZAI. Pulse Views Dashboards. https://docs.feedzai.com/display/pulse/5.+Dashboards. Accessed on August 25, 2012.
34. SCHWABER, K., AND SUTHERLAN, J. 2011. The Scrum Guide. http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf
35. LAKEWORKS. Scrum Process. http://en.wikipedia.org/wiki/File:Scrum_process.svg. Accessed on December 22, 2011
36. ALTASSIAN. JIRA. http://www.atlassian.com/software/jira/overview. Accessed on December 22, 2011
37. SOFTWARE FREEDOM CONSERVANCY. Git. http://git-scm.com/. Accessed on August 27, 2012.