Scuola di Ingegneria di Firenze
Dipartimento di Ingegneria dell'Informazione

Dashboard management system

Technical documentation

Summary

Pictures index

Introduction

The Dashboard Management System (also called “Dashboard Manager”) is a Web application whose main target is to allow users to create and manage fully customized dashboards to show or produce data.

Datasources

The system can extract data from external databases (relational, object oriented, RDF), REST APIs, IOT brokers, IOT NodeRed applications, WebSocket servers.

GUI

The graphical user interface is responsive and has got a fully configurable main menu.

Dashboard and widget creation

New dashboards and widgets can be created via a step-by-step wizard which drives the user in the choice of main aspects of new items being instantiated.

Dashboard ownership and access rights

The creator of a dashboard can decide its visibility (private or public) and, in the private case, add/remove access rights to other users of his choice.

Dashboard layout

The user can design his dashboards easily using available controls on the following domains.

Sizes

Dashboard width can be choosen by the user (while height is automatic), widgets’ dimensions are decided by the user too with a real time size adjuster keyboard .

Widgets’ position

Widgets can be positioned with a drag&drop system on a grid.

Appearance

Colors, text contents (titles, subtitles, complementary informations, data labels…) can be edited in place with context menus and real time modifications preview, dashboard and widgets’ headers can be shown or hidden.

Responsiveness

Each dashboard can be set to have fixed width or to be responsive. In the responsive case dashboard width will fit viewport width and widgets will scale their sizes to fit in their turn without changing their positions on the grid.

Dashboards list

Dashboards are shown in a responsive list of rectangular cards, each card shows a preview image of the dashboard and its management buttons.

Used technologies

Client side

Markup

All the front-end markup is written in HTML5 (with no intensive use of semantic tags) and styled with CSS3 (with no use of preprocessors like SASS or LESS).

Business logic

The client side business logic is built with JQuery core v1.10.1.

GUI responsivess, general box model

Bootstrap v3.3.7 grid layout has been used extensively both in the general GUI and in the dashboards. Some GUI components like buttons, inputs, modals and navbars have been used too.

Drag&drop system

The drag&drop grid system of the dashboard editor is based on Gridster plugin.
Attention: the current version available on GitHub here and documented here (from contributor dsmorse) is different in some behaviours from the original one (available here) used in the Dashboard Manager and developed from Ducksboard, but documentation of the original one seems to be unavailable, dsmorse documentation covers well original version too, anyway.

Charts

Classical charts like bars, cartesian plane, gauges etc… are built with Highcharts v5.0.6

Maps

Maps are built with OpenStreetmap using Leaflet library v1.3.1

Wizard tables

The tables showing datasources on add dashboard/widget wizard have been realized with DataTables component

Cards lists of items (dashboards, external services, micro applications)

List of items composed by cards have been built with a custom extension (in particular with an ex-novo rendering function and customized filters behavior) of Dynatable component.

Color pickers

Color pickers have been built with Bootstrap Colorpicker (on GitHub too)

Icons

The main icon set of the project is FontAwesome

Font

The font of the application is Montserrat.

Browser compatibility

The reference browser for the Dashboard Manager is Google Chrome. The application works well on Mozilla Firefox too (with some minor issues) and Microsoft Edge (with a known issue on wizard table), while it hasn't been tested on Microsoft Internet Explorer.

Server side

The server of the application is Apache v2.4.23, server side business logic is written in PHP v5.6.25. LDAP interfacing is made using core PHP LDAP module, while integration with Keycloack SSO is implemented with PHP OpenID Connect Basic Client .

The WebSocket server for real time data flows between IOT apps and widgets is a customized extension of PHP-Websockets server by contributor ghedipunk.

Persistence

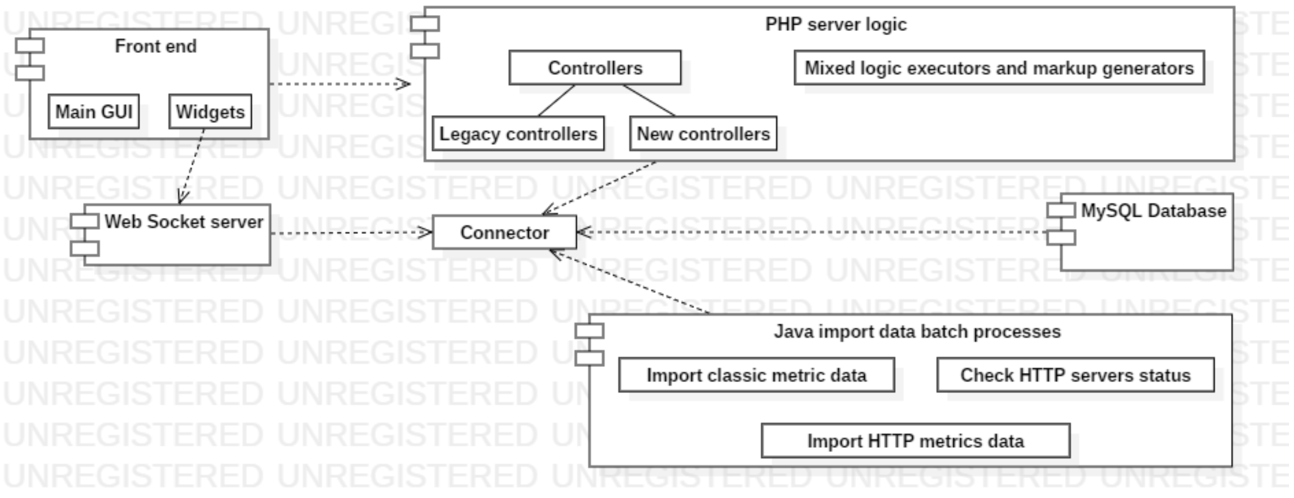
Gianni

Batch processes

Background processes for periodical data import from datasources have been written in Java SE v1.8.0

General architecture (figura da rifare con Modelio)

The system is distributed on different modules, as shown in the picture below.



1 - Component diagram

Front end

We can ideally divide the front end into two main parts: the main GUI of the application and the widgets. The main GUI is composed by many pages that communicate with the PHP controllers via HTTP calls or classical forms submissions.

On the other hand, widgets may exchange data with ad hoc PHP server controllers via HTTP calls (widgets polling the system for new data on regular intervals or actuator widgets sending data to the server) or with a Web Socket server for real time updates (mainly widgets communicating with NodeRed personal applications).

Web socket server

A web socket server (the current version is written in PHP and has a sequential logic, a Java multi-thread new version is under evaluation) receives real time data from NodeRed personal applications (these data sources are often called *personal metrics*) and forward such data to widgets subscribed to those metrics.

PHP server logic

The PHP server logic has a very composite nature. Some modules are mixed PHP/HTML markup generators (the widgets are written in PHP files too, further details are given in the proper section). Other modules (the oldest ones, called *Legacy controllers* in the picture) are only PHP logic, typically consumed by the client with form submissions and involving database and/or filesystem operations (add a new dashboard via the legacy form, add/edit/delete widgets with the legacy form, add/edit/delete classical metrics…). So called *new controllers* are small controllers performing single actions (typically towards database, filesystem, APIs) on new forms inputs via HTTP calls.

Java import data batch processes

There are 3 batch Java processes dedicated to data import from external sources to the application MySQL database.

Classical metrics data import process

This process reads the metrics list from the database and starts a new thread for each metric: this thread executes the corresponding query on the target database, saves new sample on Dashboard Manager database, sleeps for an amount of time (this interval can be set by GUI, so that you can decide update frequency for each metric) and then queries again.

HTTP servers status check process

This process reads the HTTP check status metrics list from the database and starts a new thread for each metric: this thread makes an HTTP call to a target server and searches for a certain token (a string) in the response payload. If the server answers and the token is found the server is considered in healthy status, otherwise not. Status (corresponding to HTTP response code of the testing call) is saved in application database.

HTTP metrics data import process

This process reads the HTTP metrics list from the database and starts a new thread for each metric: this thread makes an HTTP call to a target server which is expected to answer with a XML, JSON or plain text payload. Such payload is then parsed by a custom Javascript script (written by the user and executed in a Javascript in Java environment) to extract data of interest and save them in application database.

Source code tree

Here follows a description of how the source code is organized. Every item of the list below is a directory.

- *ScheduledTasks*: PHP scripts executed periodically (scheduled in the crontab of the server) to feed database tables relative to dashboards/widgets creation wizard
- *Api*: HTTP APIs (written in PHP) for external applications that interact with the dashboard manager, mainly for NodeRed personal applications (automatic dashboard and widget creation when creating a widget block on NodeRed flow, widget removal when deleting corresponding block from NodeRed flow…)
- *bootstrapTable:* third-part component for building HTML tables.
- *bootstrap3-editable*: third-part component for editable HTML items (not used).
- *bootstrapSelect:* third-part component for HTML multiselect menus.
- *bootstrapSlider:* third-part component for HTML slider input.
- *bootstrapToggleButton*: third-part component for HTML animated toggle input.
- *cam*: static images for mock city-webcams widgets.
- *conf*: application .ini configuration files.
- *confDefault*: backup default .ini configuration files.
- *controllers*: PHP server business logic controllers.
- *css*: CSS files for front end style.
- *datetimepicker:* third-part component for HTML datepicker.
- *dotdotdot:* third-part component for HTML/JS automatic text ellipsis (not used).

- *dynatable:* third-part component for HTML custom rendered tables.
- *fonts:* open-source fonts for the application GUI.
- *img:* icons and images used by the application.
- *js:* third-part and custom Javascript files.
- *leaflet-markercluster*: third-part plugin for Leaflet maps to aggregate close makers (not used).
- *leafletCore:* third-part library to generate and manage front end maps.
- *management:* various PHP modules, some generating GUI pages (dashboards list, login form, metrics management…), other exposing business logic to such pages (legacy dashboard and widget creation…)
- *moment:* third-part library to manage time data in Javascript
- *phpWebsockets:* third-part WebSocket server written in PHP.
- *phpmailer:* third-part PHP library to manage e-mail messages (not used anymore with introduction of Notificator).
- *sso:* thirs-part PHP library to manage single-sign-on authentication with Open Auth.
- *test:* custom test pages.
- *view:* PHP module that shows a dashboard.
- *widgets:* PHP modules with widgets source code.
- *wsServer:* our custom implementation of phpWebsockets.

Configuration files

Configuration files are in *conf* folder and are .ini files (to be managed in PHP). For backward compatibility reasons, such files are parsed by *config.php* module, which exposes their entries as PHP variables: *config.php* is typically included at the beginning of PHP business logic modules.

Moreover, such files are also parsed and exposed on GUI by *management/setup.php* page, so that an administrator can edit settings by the GUI itself.

The whole configuration system is organized on an *environment basis*: you can choose the environment (*dev*, *test*, *prod*) for your installation in *environment.ini*, then, for each entry in each .ini file, the system will read the setting corresponding to active environment.

conf.ini

Database connection settings (not used anymore?), Notificator API credentials.

database.ini

Database connection settings.

environment.ini

Active environment.

esb.ini

Enterprise Service Bus database settings.

general.ini

Host machine IP address, application Web URL, cache file age control for browser, session duration.

googleApis.ini

Google APIs connection settings (not used anymore?)

iotApplications.ini

URLs pointing APIs for personal IOT applications creations.

ldap.ini
Connection settings for LDAP central directory.

mailServer.ini
Connection settings for DISIT e-mail server (not used anymore?)

nodeEmittersApi.ini
Settings for HTTP data exchange between Dashboard Manager actuator widgets and NodeRed.

notificator.ini
Settings for HTTP data exchange between Dashboard Manager and Notificator.

orion.ini
Settings for HTTP data exchange between Dashboard Manager and Orion Context Broker.

ownership.ini
Settings for HTTP data exchange between Dashboard Manager and central ownership APIs.

protezioneCivile.ini
Settings for civil protection widget to get data from Florence Civil Protection (via HTTP).

serviceMapAndTV.ini
Settings for HTTP data request by the Dashboard Manager to the DiSiT ServiceMap APIs.

sso.ini
Settings for single sign on integration with Keycloack.

webSocketServer.ini
Settings for Web Socket PHP server.

Users

Origin
Dashboard Manager users can have different origin:

- *Local users:* these users are saved on database (*Users* table) and are typically used for development and testing only.
- *LDAP users:* in production environment users are typically stored in a LDAP directory service.

Profiles
There are 5 different user profiles, corresponding to different privileges in the application.

RootAdmin
The *RootAdmin* can take every possible action in the application:

- Full management of all the dashboards
- Full management of all IOT applications
- Full management of all Micro Applications
- Full management of all External Services
- Full management of Dashboard Manager settings

ToolAdmin

TBD – Che prerogative ha adesso?

AreaManager

The AreaManager has not administrative features, but manages pools of users belonging to him/her:

- Full management of own dashboards and dashboards of users belonging to own pools
- Full management of own IOT applications
- Full management of own Micro Applications
- Full management of own External Services

Manager

The Manager has not administrative features and can act only on entities of his/her own:

- Full management of own dashboards
- Full management of own IOT applications
- Full management of own Micro Applications
- Full management of own External Services

Observer

The Observer is a read-only user for restricted access dashboards:

- Not allowed to log in the application
- Allowed to log in single dashboards (view) if the author decides so.

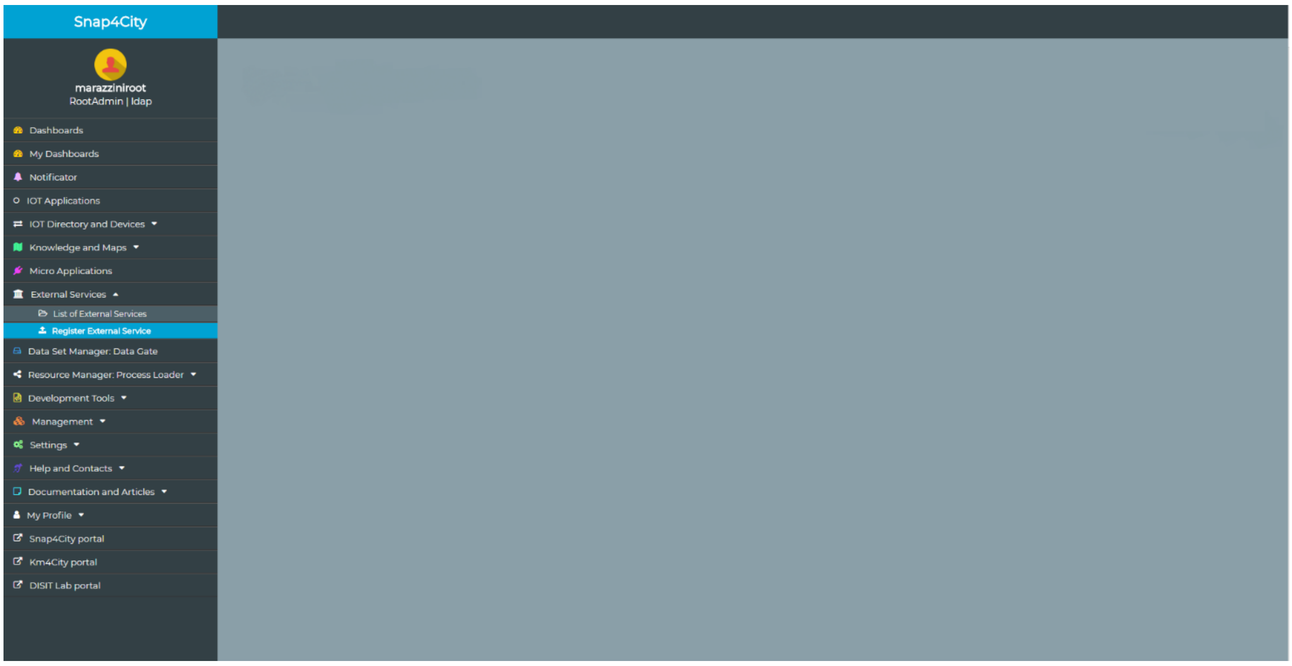
User data kept in session

After login, following user data are kept in session via PHP superglobal array *$_SESSION*:

- *$_SESSION['loggedUsername']*: username of currently logged user
- *$_SESSION['loggedRole']*: user profile of currently logged user (RootAdmin, ToolAdmin, AreaManager, Manager)
- *$_SESSION['loggedType']*: user origin of currently logged user (ldap, local)

Graphical User Interface

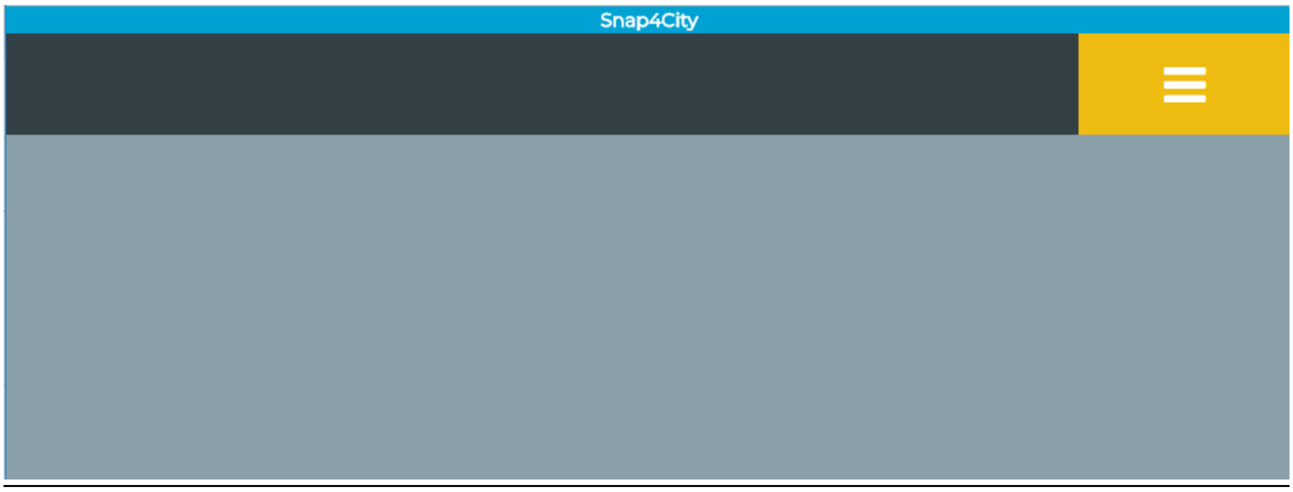
Main structure of GUI pages



2 - Main strutcture of GUI pages – Desktop version

In the desktop version, a GUI page is organized in this way:

- *Main menu*: the main menu is 2 columns wide (on the 12 available in the Bootstrap grid). Further details are given in the specific section.
- *Content section*: the content section is 10 columns wide (on the 12 available in the Bootstrap grid). In the upper dark header goes the page title, while in the grey rectangle goes the content of the page. Each page is arranged in custom manner.



3 - Main structure of GUI page - Mobile

In the mobile version, a GUI page is organized in this way:

- *Application title*: the light blue application title div is 12 columns wide (on the 12 available in the Bootstrap grid)
- *Page title and mobile menu button*: the page main title is 10 columns wide, wile the mobile menu toggle button is 2.

- *Content section*: the content section is 12 columns wide.

The break point between mobile and desktop version is set at *992px viewport width*, corresponding to Bootstrap *-md class*. This view-mode switch is obtained only with the use of such Bootstrap classes, with no further JS or CSS.

Main menu

The main menu is implemented in specific modules:

- *management/mainMenu.php*: module for desktop version
- *management/mobMainMenu.php*: module for mobile version

Desktop version

Items for the desktop main menu and submenus are stored in the database, in *MainMenu* and *MainMenuSubmenus* table.

Here follows a summary of menu population flow:

1. PHP module fetches records of *MainMenu* database table with a query
2. ResultSet is scanned: for every record a new HTML menu item is created by PHP module (with some custom aspects depending on the type of target page) if connected user has rights for that item.
    a. If current menu item has submenu, its submenu items are fetched from *MainMenuSubmenu* database table with a query
    b. This new resultset is scanned in its turn: for every record a new HTML submenu item is created by PHP module (with some custom aspects depending on the type of target page) if connected user has rights for that item.
    c. The submenu is hidden.

There are some important parameters in the database tables *MainMenu* e *MainMenuSubmenus*:

- *linkUrl*: the *<a>* menu and submenu items have their *href* attribute populated with this value.
- *icon*: icon from FontAwesome icon set for this menu item.
- *text:* text inside the menu item.
- *privileges:* a JSON array containing user roles authorized to see this menu item.
- *userType:* a string containing the user type (*local*, *ldap*, *any*) authorized to see this menu item.
- *externalApp:* this flag (*yes* or *no*) is used by the menu logic to detect if a menu item opens an external application or a page of the Dashboard Manager.
- *openMode:* this setting (*submenu, samePage, newTab, iframe*) is used by the menu client logic in conjunction with *externalApp* to decide the target of the link to be opened in the browser:
    - *submenu:* for submenu opening/closing
    - *samePage:* page to be opened in the same tab of the current one.
    - *newTab:* page to be opened in a new tab.
    - *Iframe:* page to be opened in the *iframe app* module (see next paragraphs).

  The menu logic makes a switch on this flag and sets the *href* and *target* attributes consequently.

Mobile version

The logic is exactly the same of the desktop version, but implemented in *mobMainMenu.php* and on *MobMainMenu* and *MobMainMenuSubmenus* database tables. In this way the two menu versions are independent and can be populated to adapt to different devices.

Available types of page

Custom

Custom pages have specific markup and logic code:

- *management/account.php*
- *management/dashboards.php*
- *management/datasources.php*
- *management/externalServices.php*
- *management/externalServicesForm.php*
- *management/iotApplications.php*
- *management/metrics.php*
- *management/microApplications.php*
- *management/pools.php*
- *management/poolsManagement.php*
- *management/setup.php*
- *management/widgets.php*

Iframe app

The page *management/iframeApp.php* is a universal embedded web page displayer. It has an *iframe* node in its markup (node *<iframe id="iframeApp"></iframe>*) and its logic is designed to display a Web page of user's choice integrating it with the application GUI.

The flow of this feature is the following:

1. The user puts a record in the *MainMenu* database table (or in *MainMenuSubmenus*, *MobMainMenu, MobMainMenuSubmenus*, according to the desired position of the menu item) with *externalApp = 'yes'* and *openMode = 'iframe'*.
2. When the menu population algorithm executes in *mainMenu.php* or *mobMainMenu.php*, the new menu item (an hyperlink *<a>*) is created with *data-externalApp="yes"* and *data-openMode="iframe"* attributes.
3. Client side (but again in *mainMenu.php* or *mobMainMenu.php*) the click event on menu and submenu items has got a custom handler: if the clicked menu item has *data-externalApp="yes"* and *data-openMode="iframe"* attributes, browser location is moved to *"iframeApp.php?linkUrl=" + encodeURI(link_to_desired_page) + "&linkId=" + link_identifier_of_choice + "&pageTitle=" + page_title_of_choice*.
4. The *iframeApp.php* module loads in its iframe the URL received in *linkUrl* parameter and shows in its header the title received in the *pageTitle* parameter.

Pages displaying cards list of items

The following pages display a list of items organized in responsive cards:

- *management/dashboards.php*: list of dashboards with preview image and management buttons
- *management/iotApplications.php*: list of IOT applications with preview image and management buttons
- *management/microApplications.php*: list of Disit WebApp micro applications with preview image
- *management/externalServices.php*: list of Web pages or Web applications with preview image

This kind of page follows the following pattern:

1. The *Dynatable* component is imported in the head of the HTML document.

2. In the HTML markup a new table is created with thead populated with as many headers as the fields we want to show and each <*th*> with a *data-dynatable-column* attribute valorized with the corresponding in-memory data structure field holding data for the table (see next passages)
3. Data for the table are retrieved by the client from the server with an AJAX HTTP call.
4. A custom function to build list as cards is defined, its name is *myCardsWriter*: this function is executed by Dynatable on every data item to render it on the GUI in a custom way. It takes four parameters:
    a. *rowIndex:* index of the data row to be rendered.
    b. *record:* reference to the data row to be rendered.
    c. *columns:* array containing the field names of the row to be rendered.
    d. *cellWriter (not used):* optional custom function to render single data fields.

    This function returns a string containing the HTML markup of your choice that renders the data item. In our case we build a div containing item preview image, title and optional control buttons

5. If and when such data arrive, first of all the *dynatable:afterProcess* event is bound to the table: put in the handler function all the code you want to be executed after every table processing made by the component (creation, filtering…). Even if it's anti intuitive, it's important to put such binding *before* calling *Dynatable* constructor, otherwise it doesn't work.
6. *Dynatable constructor* is called, passing to *_rowWriter* our custom rendering function *myCardsWriter()* defined at point 4.