--[api resource]---

mashapes: /workspaces/mashape\_spider & chromedriver

from wenwen, urls crawling works, but extracting functions from a url does not work

for me:

three parts:

urls /mashape\_spider/usrs

get all outputs /mashape\_spider/mashape\_spiderMultiple (only check output, and save URL/)

get one /mashape\_spider/mashape\_spiderSingle (all)

get all

to-do: all mashpe apis -> api signature

to-do: wadl, wsdl -> api signature

to-do: how about the CloudRISE

to-do2: widgets/

**go through GIS example**

**idea:**

**API signature: (flat apis)**

**management: import:mashape/wsdl/wadl/manual, U/D**

**tag mine/update/delete**

**widgets (auto & upload)**

**how to auto?**

**MDL/WDL?**

**Run?**

**Operator (what)?**

**tag tree (learning, new/update/delete)**

**mashup**

**wiring**

**widgets (all APIs), use tag to guild, but link manually**

**operators (all) – dummy, split, merge**

**configue (show/unshow, rerun/timeout)**

**view / “run”**

---[note of open specification of FiWare]

Web application mashups integrate heterogeneous data, application logic, and UI components (widgets/gadgets) sourced from the Web to create new coherent and value-adding composite applications.

Issue:

Web application mashups can be manually developed using conventional web programming technologies, but this fails to take full advantage of the approach. Application mashup tools and platforms such as the one being specified by the FI\_WARE's *Application Mashup GE* are aimed at development paradigms that do not require programming skills and, hence, target end users (being them business staff, customers or citizens). They also help to leverage innovation through experimentation and rapid prototyping by enabling their users (a) to **discover** the best suited mashable components (widgets, operators and off-the-shelf mashuplets) for their devised mashup from a vast, ever-growing distributed catalogue, and (b) to **visually** mash them up to compose the application.

For SAMSaaS: a) to manage and identify APIs

b) to discoverbest suited APIs

c) to visual and run simulation mashup as application

key features:

mashup definition language (**MDL**) & widget definition language (**WDL**)

xml/rdf template schemas – no use

usdl extensions – no use

**zipped** file format (**wgt**) that allows mashable components to be conveniently stored and distributed.

**Wiring**: a mechanism empowering end users to easily connect widgets in a mashup to create a fully-fledged event-driven dashboard/cockpit with RIA functionality.

**Piping**: a mechanism empowering end users to easily connect widgets to back-end services or data sources through an extendable set of operators, including filters, aggregators, adapters, etc.

**Visual** rendering of widgtees in the application amashup UI

MAC (widget, operator and mashup) **life-cycle** management support.

Application mashup **execution** **engine** model:  capable of deploying and running a mashup from a MDL file. It provides support for managing mashup state persistence, and for managing the wiring and piping mechanisms

**catalogue** of mashable compnnets: the catalogue empowers end users to store and share their newly created application mashups with other colleagues by communicating with the Store GE

target usage: The platform enables composition either from the front-end perspective --**application** **mash-up**s- or the **back-end perspective** --composite services. Specifically, the Application Mashup GE **targets** composition from the **front-end perspective** and is expected to leverage the creation and the execution of value-added applications not only by application providers, but also by intermediaries and end users acting as composers, a.k.a. prosumers. **Prosumers** are consumer-side end users who cannot find an application that fits their needs and therefore modify/create an application mashup in an ad-hoc manner for their own consumption. As the capabilities and skills of the target users being considered are expected to be very diverse, all kinds of usability issues, conceptual simplification, recommendation and guidance, etc. are taken into consideration.

-basic concepts-

The Application Mashup GE describes a Web platform that helps users to easily and visually create and run their own Web application mashups. Its functionality can be divided into a **client-side** part running on the user web browser and a **server-side** part running on a web server.

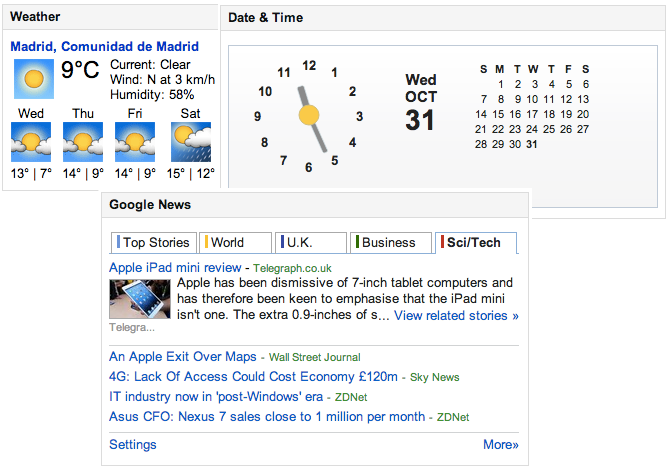
The Application Mashup GE is based on a composition model already published by the authors of this specification in a journal publication: ( which has been specifically designed to empower end users with few or no programming skills to create and share their own web composite applications in a fully visual fashion:)

[↑](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.Apps.ApplicationMashup" \l "cite_ref-0) Lizcano, D., Alonso, F., Soriano, J., and López, G. (2011) [A new end user composition model to empower knowledge workers to develop rich Internet applications](http://dl.acm.org/citation.cfm?id=2230838). In Journal of Web Engineering, Vol.10 No. 3, pp. 197-233, Rinton Press Inc. Sept. 2011

widgest:

***Widgets*** are the key elements of the composition model that the Application Mashup GE must support. Together with connectors and mashups (mashups are considered as building blocks for other application mashups), they make up the complete set of Mashable Application Components (MAC – widget – operator – mashup). A **widget** is a lightweight Web application that runs on the user's web browser, Widgets are usually developed using current Web technologies (HTML(5), CSS, Javascript, ...) and they are bound to heterogeneous data coming from the Web (e.g. **Web APIs – sixuan: it is generated from a web api)**. They can be regarded as the service front-end, because they offer to users a graphical user interface (GUI), so that they can easily get a visual representation of the service data and functionality to which the widget is bound.

The figure below shows an example of what iGoogle's widgets look like:



wiring:

These *early* widgets *per se* are isolated applications that do not interact with each other. However, the Application Mashup GE aims at providing a mechanism to visually compose a fully-fledged web application from different widgets that can now interact with each other via events and data sharing. This mechanism is what the composition model calls **wiring**. The idea behind wiring is easy: widgets expose (data/event) inputs and (data/event) outputs, so that an output from one widget can be linked to other widgets' inputs following a composition technique based on pre- and post-condition mechanisms. This way, the Application Mashup GE manages the data/event flow between widgets.

The mechanism allows for the use of event-driven *programming* features, e.g. a widget can send an event through one of its outputs on an event trigger. The figure below shows an example of the wiring metaphor:



**An abstract representation of how the Application Mashup GE could support the Wiring mechanism**

piping:

Widgets supported by the Application Mashup GE must be able to access their data from services in at least the following two ways: *programmatically (in widgets)* or by means of *operators* and through a visual technique called *piping* that establishes how these operators can be combined to form a *pipe*.

Programmatically: invocation of services from the widget's code: through WidgetAPI

operators: piping: for users with few programming skills: an *operator* does not offer a GUI but, like widgets, it has an abstract representation with both inputs and outputs and can thus be wired to widgets allowing the data flow between them. Operators are usually bound to some kind of data source (SOAP service, REST API, etc.). In other words, operators are configured out-of-the-box to get access to a backend service, but they can also be made to subscribe to and get events from a publish/subscribe system. They can also act as filters, aggregators, mediators, etc. when used in the *piping* technique to build a *pipe*.

Sum up

To sum up, an implementation of the Application Mashup GE must support the process of visually creating a composite web application by composing different widgets using the wiring mechanism, which interconnects those widgets, along with the piping mechanism, that makes use of operators to get access to new data, perform an operation on that data, and finally pass it to the widgets through their inputs.

The figure below shows and example of what a web application mashup looks like. It is made up of a number of widgets, which interoperate with each other by exchanging data and events following the connections defined by the user. These widgets can be easily repositioned and/or resized to reflect the user needs and/or preferences.

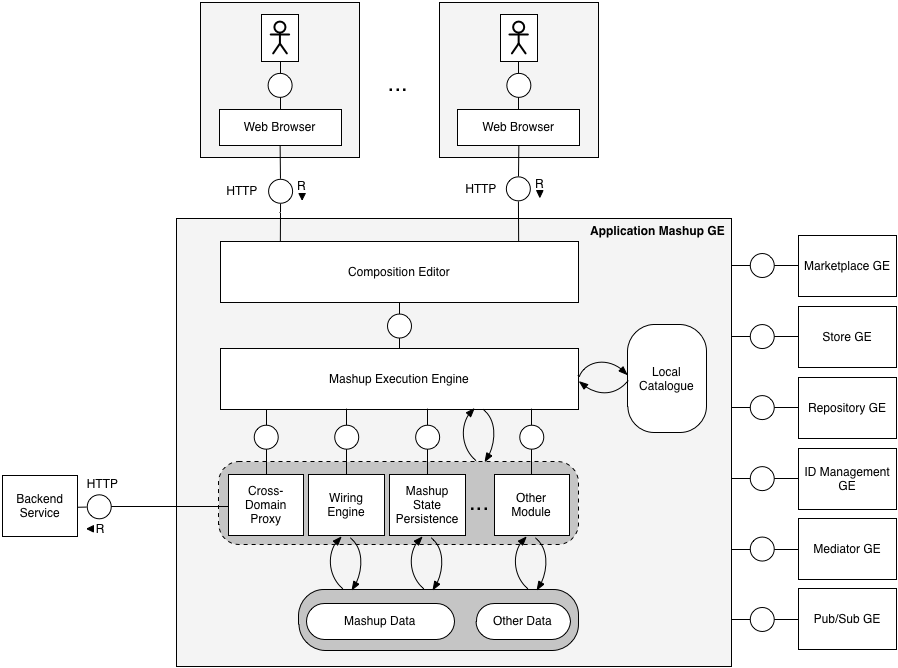
---archiecture--

This section describes the Application Mashup GE architecture. The diagrams use FMC (Fundamental Modelling Concepts) notation to facilitate the communication not only between technical experts but also between them and business or domain experts. The Application Mashup GE provides the functionality necessary for developing and executing mashups. As the figure below shows, the core of the Application Mashup has three main components: the **Composition Editor (application layer)**, the **Mashup Execution Engine (mashup wiring & execution layer)**, and the **Local Catalogue (api component layer)**.

To sixuan: api comonent, tag ontology, mashup wiring, mashup execution, application

so the mashup wiring of (widget / operator / tag semantic), API widget view and execution is the objectives.

The wirecloud is for the application client part & some widgets



**The Application Mashup GE Architecture**

The **Composition Editor** component is the web-based tool with which end users interact via a web browser in order to create their **own mashup applications.** This component must, at least, offer end users a kind of ***workspace***where they can spatially place or arrange widgets, plus an extra **view of the wiring mechanism** to set the interconnection between the arranged widgets. Because this component is a visual editor, this document does not set the visual appearance that this tool must have. It is up to the GE's implementation developers to create their **own look** & **feel** for this tool.

The **Mashup Execution Engine** component is probably the **most important** part of the GE. It coordinates **widget execution** and controls the **data flow** between widgets. It can access the **Local Catalogue to deploy and execute stored widgets**. The functionality of this component can be connected to and extended by a number of plug-in modules as shown in the Application Mashup GE Architecture figure. Module functionality is exposed to the **widgets** by means of the **WidgetAPI** (see [Open API Specifications](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Application_Mashup_Open_RESTful_API_Specification_(PRELIMINARY))). Some of these plug-in modules must always be there:

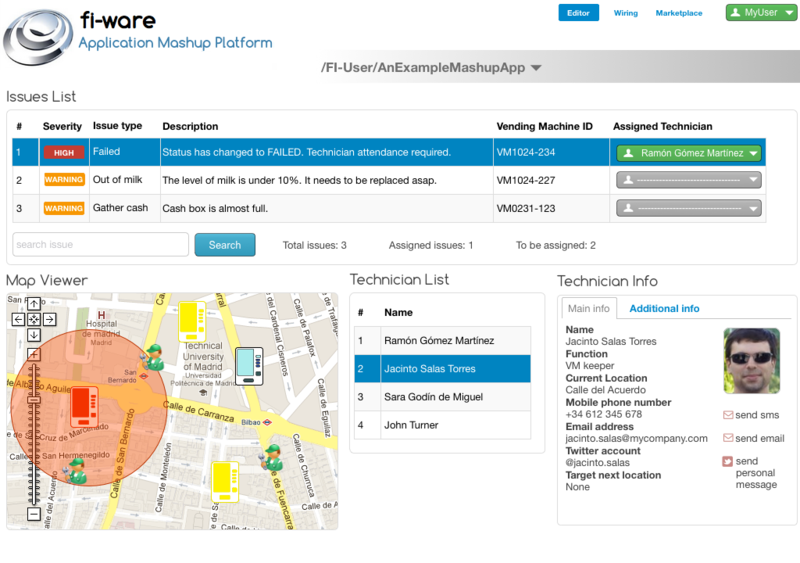
* The **Cross-Domain Proxy** module: this component will provide widgets with a proxy to overcome the Javascript cross-domain problem.
* The **Wiring Engine** Module: **this component manages the wiring mechanism**.
* The **Mashup State Persistence** Module: this module is in charge of guaranteeing the persistence of the MACs under execution. This includes not only to store the **widgets** and **operators** involved in the mashup and their state, but also their **position** in the editor view, their interconnections (wiring and piping) and so on.

It must be possible to enhance widget functionality by adding new modules to the Mashup Execution Engine. For example, a Publish/Subscribe module could be added to provide widgets with the ability to receive and publish data in a pub/sub fashion using the new added module.

All plug-in modules must be able to make use of internal storage (i.e. a database) for their specific persistence needs.

The third main component is the **Local Catalogue**. This component is where MACs, either purchased from the FI-WARE Store GE or installed (uploaded) by the end-user, are stored, configured, and set ready for deployment and execution. This component should be a kind of showcase for the logged user of the Application Mashup GE.

The following sections describe the languages needed to support widgets, operators and mashups, including the USDL extensions for all business-related GEs to process any MAC as an offering, and the specific file formats used to store and distribute widgets. By implementing these artifacts, a concrete implementation of the Application Mashup GE will support an internal representation of MACs (widgets and mashups), which is necessary to interact with them.

[](https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/File:MashupEditor.png)

**An implementation of the Application Mashup GE showing how the mashup developers can arrange widgets to create their own Web mashup**

-- Mashup and Widget Template

WDL – template description as a XML Schema Definition

Template:

catalog.**ResoruceDescription** //name, author, description, image, wikiuri,

Platform.preferences //user preference – no need

Platform.StateProperties //persitant widget state, store the sate of the widget while it is executing, has name, type, label

**platform**.**Wiring** //list of input/output endpoints

outputEndpoint – type, name, label, description, friendcode

inputEndpoint– type, name, label, action\_lable, description, friendcode

Widgets may send data (events) through an output endpoint. To do so, they must declare the endpoint using the *OutputEndpoint* element. These elements have the following attributes:

* **name:** output endpoint name.
* **type:** output endpoint data type: only "text" (string) datatype does make sense in here.
* **label:** text to be displayed in the user interface.
* **description:** text that describes the output.
* **friendcode:** keyword used as an output endpoint tag: it will help the platform to make suggestions in the wiring process.

On the other hand, widgets can receive asynchronous data through the input endpoints. These endpoints are meant to be used by the widget for receiving data (events) coming from other widgets. The required *InputEndpoint* elements requires the following attributes:

* **name:** input endpoint name.
* **type:** input endpoint data type: only "text" (string) datatype does make sense in here.
* **label:** text to be displayed in the user interface.
* **action\_label:** short text that describes what is going to happen if an event is sent to this input endpoint. Widgets could use this text in buttons, selection boxes, etc... allowing end users to select what to do (and the widget will send an event to the associated target endpoint)
* **description:** text that describes the input.
* **friendcode:** keyword used as an input endpoint tag: it will help the platform to make suggestions in the wiring process.

platform.**link** //source code of the widget, XHTML, href/contecttype/cacheable

platform.rendering // how to show, width, height

MDL – template description as a XML Schema Definition

Template :

catalog.resourceDescription //name, author, description, image, wikiuri,

// widget reources (position xyz, rendering width, height,layout, minimized)

platform.wiring //list of input/output endpoints of the widgets contained in the mashup

//outputendpint

//inputendpoint

//operator

//connection source- type(widget/operator) id, endpoint; target)

--**WGT zipped file format--**

The Mashup Application GE relies on PKWare's Zip specification as the archive format for the self-packaged version of the Mashable Application Components. The packaging format acts as a container for files used by a MAC whereas the only initial requirement is to have a configuration document declaring metadata and configuration parameters for the MAC. This configuration file must use one of the metadata description languages supported by the Mashup Application GE (WDL and MDL in either of its flavours: **XML** or RDF). This configuration file must be present at the root of the Zip container and the name must be **config.xml** or config.rdf. Any relative path/URL included in the configuration document will use the root of the zip file as the base path/URL.

The Mashup Application GE should prohibit relative paths for accessing files outside the container. This is especially important as the Mashup Application GE may extract these files to the file system.

## Life-cycle of a Mashable Application Component (mashup, widget and operators)

Note that Web mashups are aimed at leveraging the "long tail" of the Internet of Services by exploiting rapid development, the "Do-It-Yourself" (DIY) metaphor, and the ability to share mashable components. They typically serve a specific situational (i.e. immediate, short-lived, customized, specific) need, often with a high reuse potential. This need for sharing means that the Application Mashup GE should be fully compliant with the FI-WARE's Marketplace, Store and Repository Generic Enablers. The fact that a MAC can be offered in a Store before being used in the Application Mashup GE results in the definition of the following MAC life-cycle:



**Lifecycle of a MAC (mashup, widget and operator)**

Mashable Application Components (i.e. mashups, widgets and operators) must pass through the following states:

* **Published**
* **Bought/installed**
* **Deployed**

The init state for a MAC means that the MAC is neither published in the store, bought, nor installed in the user local repository. A MAC is **published** when it is made available to Store customers. Users that are interested in using a **published** MAC, can buy the MAC, thus transferring it to a *bought* state. Once *bought*, the MAC is automatically **installed** in the local catalogue of the Application Mashup GE. An alternative is to *upload* a MAC that the users have developed and which they do not have to buy. This is why the state is named **bought/installed**. Once the users have*uploaded* the MAC to the local catalogue, they can proceed to *publish* the MAC. Once the MAC is installed, it can be **deployed** in the user workspace. *Bought* and *installed* MACs must be**deployed** in the user workspace before they can be *configured*.

---Basic Design Principles

* **API Technology Independence**

The API abstracts from the specific implementation technology. Implementations using more than one type of platform and framework should be possible.

* **Web Browsers should not limit the functionalities of the Application Mashup GE**

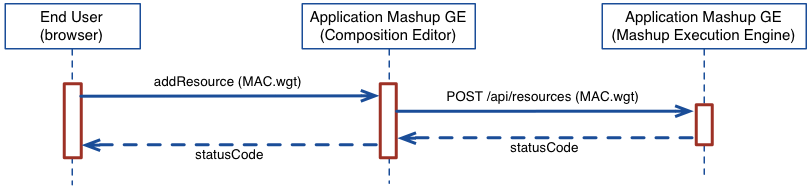
HTML5, CSS and JavaScript must be used to fully exploit the brand new Web applications capabilities.

* **User-matched interaction abstraction level**

Editors could cater for different user expertise (from technical experts skilled in the composition language to domain experts without technical expertise or even simple end users with no programming or technical skills) and roles (from composed service creators, to resellers and finally prosumers) by hiding complexity behind different types of building blocks, trading off flexibility for simplicity.

#### **Upload a Mashable Application Component to the Local Catalogue**

MAC developers must be able to upload their own developed resources to the local catalogue of the Application Mashup GE. End users must also be able to upload the MACs they already have stored in their local hard disk to the local catalogue. The implementation of the GE must enable users to select their new \*.wgt packaged MAC and upload it to the Local Catalogue.

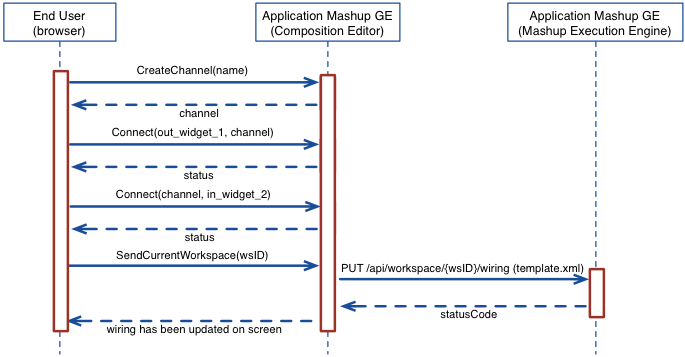


#### **Interconnect widgets/operators using wiring**

Wiring functionality enables users to connect one or more widgets/operators to one or more other widgets/operators by means of a channel. The Composition Editor will help users to connect one or more of the possible outputs of a widget/operator with the input of another widget/operator. This way, data flows between MACs allowing the mashup application to act as an information and process dashboard.

Besides, piping deals with how operators can bind to a specific backend service to gain access to data provided by the service. Then, users can wire the operator outputs either to other operators in order to perform some kind of data filtering/adaptation processes or to other widgets to consume the data.

The figure below shows how users set up channels between widgets/operators for wiring/piping.



The Widget API (the subject of this entry) is a JavaScript API, while the ApplicationMashupAPI is a RESTful one.

1

<https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Widget_Open_API_Specification>

Widget api: This document provides a full specification of how to make widgets interoperate with the Mashup Execution Engine. The Widget API is a JavaScript API that allows deployed widgets in a Mashup Execution Engine to gain access to its functionalities. It does not make sense to expose it as a RESTful API since it needs to be consumed by a widget in its own local execution environment. Amongst other functionalities, this API allows the widgets to gain access to remote resources. For example, in order to gain access to a remote REST API or to resolve cross-domain problems, a widget needs to use a proxy through the Widget API.

2

<https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Application_Mashup_Open_RESTful_API_Specification>

The Application Mashup API is a RESTful, resource-oriented API accessed via HTTP that uses various representations for information interchange. This API provides the functionality to create and modify workspaces and the functionality to manage the resources available for building these workspaces.



----Tutorial of developing widget--

# Tutorial. How to develop widgets

In this tutorial we are going to implement a Weather Widget as an excuse to learn how to use the most commonly used features of Wirecloud, from a point of view of Widgets. Our intention is to create a Widget capable of making AJAX request to a external service and of communicating with other widgets in a mashup. This guide does not cover the development of the widget's user interface, which doesn't have to do with Wirecloud and it's based on standard HTML, JavaScript and CSS code.

First of all, you can download a "little" initial code from this [link](https://conwet.fi.upm.es/docs/download/attachments/1278018/Example1Skel.zip?version=3&modificationDate=1373275046000).

Then, you will need to create a new API key for the Weather Underground API using this [link](http://www.wunderground.com/weather/api/d/login.html).

Once you have the key, it's time to learn how to make requests to the service.

## Making request to Weather Underground

## HTTP GET

Our widget is going to provide the weather forecast for a given location, this location is going to be defined by a coordinate.

Weather Underground provide a rest API for this for this purpose (documented [here](http://www.wunderground.com/weather/api/d/docs)), but we cannot access this API using normal AJAX request (using XMLHttpRequest) due browsers applying the same origin policy to javascript code. Fortunately, Wirecloud provides the MashupPlatform.http.makeRequest method for dealing with this problem. A possible way to access to this API is by using the following code:

sixuan: here is define function to invoke the web api, based on the input

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | **var** getForecastByCoord = **function** getForecastByCoord(coord, onSuccess, onError) {  **var** url;  **if** ((**typeof** onSuccess !== 'function') || (**typeof** onError !== 'function')) {  **throw** **new** TypeError();      }      url = 'http://api.wunderground.com/api/' + API\_KEY +'/conditions/forecast/q/';      url += coord.lat + ',' + coord.lon;      url += '.json';      MashupPlatform.http.makeRequest(url, {          method: 'GET',          onSuccess: **function** (transport) {  **var** response;              response = JSON.parse(transport.responseText);  **if** (response.response.error) {                  onError();              } **else** {                  onSuccess(response);              }          },          onError: **function** (transport) {              onError();          }      });  }; |

The getForecastByCoord function makes the appropriated request to Weather Underground and passes the result to theonSuccess callback.

In the next section we'll learn how to prepare a widget to receive events from other widgets.

## Adding an input endpoint

Input endpoints must be declared into the widget template before it can be used by the javascript code of the widget. To do so, open template.xml and add an InputEndpoint element into the Platform.Wiring section. The final result should look like:

[?](https://conwet.fi.upm.es/wirecloud/tutorial/2)

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...      <**Platform.Wiring**>          <**InputEndpoint** name="coord" type="text" label="Show forecast by coord" description="Shows the weather forecast for a given location (a latitude longitude coordinate)." friendcode="location"/>      </**Platform.Wiring**>  ... |

* The name attribute will be use to reference to the input endpoint when using the javascript API.
* The label attribute will be used mainly in the Wiring Editor and will be the official name by which end users will know the input endpoint. Also, this attribute can be translated whereas the name attribute not.
* The description attribute is used to provided end user with a description of what is going to happen if an event arrives the input endpoint. This description is very important for the wiring process as the user needs this information for taking decisions on howto wire widgets.
* The friendcode is used by the Wiring Editor to provide basic wiring recommendations. In this case, we are declaring that we accept data produced by output endpoints with a friendcode of "location". The format of this data is a string with the longitude and the latitude separated by a comma.

Once declared the input endpoint in the widget template, you can register a callback for this endpoint making use of theMashupPlatform.wiring.registerCallback method. In addition to registering the input endpoint, we need to process event data before using it and to notify the user that the forecast data for the given location is being requested. This can be accomplished by using the following code:

sixuan: here is the listener to receive the event from input and call the function defined before

[?](https://conwet.fi.upm.es/wirecloud/tutorial/2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **var** searchByCoordListener = **function** searchByCoordListener(event\_data) {  **var** tmp, coord;      tmp = event\_data.split(',');      coord = {          lat: tmp[1],          lon: tmp[0]      };        startLoadingAnimation();      getForecastByCoord(coord, **function** (results) {          updateWeatherForecast(results);          stopLoadingAnimation();      }, **function** () {          clearWeatherForecast();          stopLoadingAnimation();      });  };    MashupPlatform.wiring.registerCallback('search\_coord', searchByCoordListener); |

## Adding an output endpoint

As we did with the input endpoint, we need to declare the new output endpoint in the weather widget's template.

[?](https://conwet.fi.upm.es/wirecloud/tutorial/3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...      <**Platform.Wiring**>          <**InputEndpoint** name="coord" type="text" label="Show forecast by coord" description="Shows the weather forecast for a given location (a latitude longitude coordinate)." friendcode="location"/>          <**OutputEndpoint** name="location\_coord" type="text" label="Forecast location" description="This event is launched when the user clicks on the location name of current forecast." friendcode="location"/>      </**Platform.Wiring**>   ... |

Description of the attributes:

* The name attribute will be use to reference to the output endpoint when using the javascript API.
* The label attribute will be used mainly in the Wiring Editor and will be the official name by which end users will know the output endpoint. Also, this attribute can be translated whereas the name attribute not.
* The description attribute is used to provide the end user with a description of when events are sent using this output endpoint. This description is very important for the wiring process, as the user needs this information for taking decisions on how to wire widgets.
* The friendcode is used by the Wiring Editor to provide basic wiring recommendations. In this case, we are declaring that we send data aligned with the friendcode "location".

After adding the output endpoint to the widget description, we can send data through it using the MashupPlatform.wiring.pushEvent method. The following code adds an event listener to the location title that sends the location of the current forecast:

sixuan: here is send event to output with the message, can do it after the input handling fucntion above

[?](https://conwet.fi.upm.es/wirecloud/tutorial/3)

|  |  |
| --- | --- |
| 1  2  3  4  5 | document.getElementById('title').onclick = **function** (event) {      MashupPlatform.wiring.pushEvent('location\_coord',      forecast\_data.current\_observation.display\_location.longitude + ',' +      forecast\_data.current\_observation.display\_location.latitude);  }; |

----XML HTTP request synchronous vs. asynchronous

XMLHttpRequest supports both synchronous and asynchronous communications. In general, however, asynchronous requests should be preferred to synchronous requests for performance reasons.

In short, synchronous requests block the execution of code which creates "freezing" on the screen and an unresponsive user experience.

----**-----[cloud wire]-**--

Open Source CloudWire – Setup

Data: 2015, 1, 10

Very important:

Python 2.7 (NOT 2.6)

Dajango: 1.6 (NOT 1.7)

Only one pip 1.6 (or 1.5)

Source:

See the latest version of <https://github.com/Wirecloud/wirecloud>,

Or the downloaded “wirecloud-originalsource-2015-Jan-10.zip”

Follow the following steps

when “wirecloud-admin”, if it says something missing, then “pip install XXX”

download IDE for Dajango project

tar -C komodo -xvf Komodo-Edit-8.5.4-14424-linux-x86\_64.tar.gz

The following is from the website:

Installation & Administration Guide

# Requirements

In order to get Wirecloud up and running, the following software is needed:

* A Database Manager (MySQL, PostgreSQL, SQLite3...)
* Python 2.7. Python 2.6/3 and other versions are currently not supported.
* Django 1.6
* South 0.7.3+
* lxml
* BeautifulSoup
* python-requests
* django-compressor 1.2
* rdflib 3.2.0+
* pytz

All these dependencies are available for Linux, Mac OS and Windows, so Wirecloud should work on any of these operating systems. However, it is better to use Debian Wheezy, CentOS 6.3, Ubuntu 11.10+ or Mac OS X as these operating systems are actively tested. **(my version: Ubuntu12.14 LTS)**

Most of software above can be easily installed using pip (the database manager, python and pip itself can be installed using the package management tools provided by your operating system or using the available installers):

[?](https://conwet.fi.upm.es/wirecloud/requirements)

|  |  |
| --- | --- |
| 1  2 | $ sudo pip install "**Django==1.6**" "south<2.0" BeautifulSoup lxml  $ sudo pip install "django\_compressor>=1.2" "rdflib>=3.2.0" requests pytz |

**NOTES**:

* See [http://lxml.de/installation.html#installation](http://lxml.de/installation.html" \l "installation) if in trouble installing lxml. For example, in Debian and Ubuntu you probably have to install the libpython-dev, libxml2-dev and libxslt1-devpackages:

[?](https://conwet.fi.upm.es/wirecloud/requirements)

|  |  |
| --- | --- |
| 0 | $ sudo apt-get install python-dev libxml2-dev libxslt1-dev |

* Wirecloud can make use of the Marketplace, Store and Repository GEs. If you want to exploit this support, you can choose between installing these GEs or using any of the instances publicly available, for example, on the FI-WARE testbed (see the "Instances" tab of the corresponding entries at [http://catalogue.fi-ware.eu](http://catalogue.fi-ware.eu/)).
* You can install pip for Windows from <https://sites.google.com/site/pydatalog/python/pip-for-windows>.

# Installing Wirecloud

Wirecloud can be installed in two ways: using pip or getting the source code.

## Installing Wirecloud using pip

The easiest way to install Wirecloud is making use of [pip](http://www.pip-installer.org/en/latest/), the tool for installing and managing Python packages.

You only have to download the desired release of Wirecloud from the [FI-WARE PPP Public Files area](https://forge.fi-ware.eu/frs/?group_id=7), and once downloaded, install it using the following command (assuming you downloaded APPS-Application-Mashup-Wirecloud-2.3.0.tar.gz):

[?](https://conwet.fi.upm.es/wirecloud/installing-wirecloud)

|  |  |
| --- | --- |
| 1 | $ pip install APPS-Application-Mashup-Wirecloud-2.3.0.tar.gz |

You can always install the latest version of Wirecloud from PyPI using the following command:

[?](https://conwet.fi.upm.es/wirecloud/installing-wirecloud)

|  |  |
| --- | --- |
| 1 | $ pip install wirecloud |

## Installing Wirecloud from sources

The source code of Wirecloud is available at [GitHub](https://github.com/Wirecloud/wirecloud).

To get the latest development version of the code, you can choose between two options:

* Go to the Wirecloud repository on GitHub and click on the ZIP button to download the repository as a zip file, or just click on this [link](https://github.com/Wirecloud/wirecloud/zipball/develop). Unzip it.
* Or use a [GIT](http://git-scm.com/) client to get the latest development version via Git:

[?](https://conwet.fi.upm.es/wirecloud/installing-wirecloud)

|  |  |
| --- | --- |
| 1 | # git clone git://github.com/Wirecloud/wirecloud.git |

 Once downloaded the source code, you can install wirecloud using the setup.py script (this step requires root privileges):

[?](https://conwet.fi.upm.es/wirecloud/installing-wirecloud)

|  |  |
| --- | --- |
| 1  2 | $ cd <path/to/source/code>/src  $ python setup.py install |

# Creating a new instance

Once Wirecloud is installed, you will have access to the wirecloud-admin script. This script is, among other things, used for deploying new instances of Wirecloud. To do so, you have to use the startproject command. For example, you can create a "wirecloud\_instance" instance of Wirecloud running the following commands:

[?](https://conwet.fi.upm.es/wirecloud/new-instance)

|  |  |
| --- | --- |
| 1  2 | $ cd samsaas  $ wirecloud-admin startproject wirecloud\_instance |

There is also an option for doing a quick deployment using SQLite3 with a default admin user (password: admin). This method is very useful for creating a Wirecloud instance for testing:

[?](https://conwet.fi.upm.es/wirecloud/new-instance)

|  |  |
| --- | --- |
| 1  2 | $ cd samsaas  $ wirecloud-admin startproject --quick-start wirecloud\_instance |

If everything goes ok, you should be able to pass to the Running Wirecloud section.

# Database installation and configuration

To set up the database engine, it is necessary to modify the DATABASE configuration setting in the instancesettings.py file (e.g. /opt/wirecloud\_instance/wirecloud\_instance/settings.py). You can use any of the database engines supported by Django.

The following examples show you how to configure SQLite and PostgreSQL databases.

## SQLite

Setting up a SQLite database can be just accomplished within seconds by using the following parameters into thesettings.py file:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | DATABASES = {        'default': {               'ENGINE': 'django.db.backends.sqlite3',               'NAME': '<dbfile>',               'USER': '',               'PASSWORD': '',               'HOST': '',               'PORT': '',       }   }</dbfile> |

where <dbfile> is the full path to the database file.

Remember to have the python-pysqlite2 module installed:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ sudo apt-get install python-pysqlite2 |

Finally, please take into account that SQLite database is not recommended for production purposes. It is only useful for evaluation purposes.

## PostgreSQL

For production purposes, PostgreSQL database is a much better choice. To do so, the following parameters must be set insettings.py:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | DATABASES = {        'default': {               'ENGINE': 'django.db.backends.postgresql\_psycopg2',               'NAME': '<dbname>',               'USER': '<dbuser>',               'PASSWORD': '<dbpassword>',               'HOST': '',               'PORT': '',       }   }</dbpassword></dbuser></dbname> |

where <dbname> represents the name of the database, and <dbuser> is the name of the user with privileges on the database.

First install the object-relational database system:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ sudo apt-get install postgresql |

And then the python interface to the PostgreSQL database python-psycopg:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ sudo apt-get install python-psycopg2 |

Afterwards you have to create the project Database. We assume that your user has super administrator permissions in PostgreSQL. This usually means that you have to login as the postgres user (i.e. $ sudo su postgres).

Both the PostgreSQL database and its user can be created with the following commands:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2 | $ createuser <dbuser> [-P]  $ createdb --owner=<dbuser> <dbname></dbname></dbuser></dbuser> |

If you want to create a password protected user you must use the -P option.

If you want to create a database called 'wirecloud' and a user called 'wc\_user' with privileges on this database, you should write the following:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2 | $ createuser wc\_user [-P]  $ createdb --owner=wc\_user wirecloud |

Finally, it is also needed to allow local connections to the database, i.e. from the computer you are installing Wirecloud. To do so, add the following rules to the beginning of the /etc/postgresql/X.X/main/pg\_hba.conf file. In other words, the following two rules MUST be the first two rules of the file:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2  3 | # TYPE  DATABASE           USER            CIDR-ADDRESS          METHOD  local   wirecloud          wc\_user                               trust  local   test\_wirecloud     wc\_user                               trust # only necessary for testing Wirecloud |

Reload pg\_hba.conf in PostgreSQL server with the following command:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ sudo service postgresql reload |

And finally, restart PostgreSQL and check if your user has access using this command:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ psql wirecloud -U wc\_user |

## Database population

Before running Wirecloud, it is necessary to populate the database. This can be achieved by using this command:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | # python manage.py syncdb |

This command creates some tables and asks you if you want to create a Django superuser. This user is required to login into Wirecloud and to be able to perform administrative tasks; please respond yes. An example of the command output, where user/password are admin/admin, is the following:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ...    You just installed Django's auth system, which means you don't have any superusers defined.  Would you like to create one now? (yes/no): yes  Username (leave blank to use 'wirecloud'): admin  E-mail address: admin\_at\_c.com  Password: \*\*\*\*\* (admin)  Password (again): \*\*\*\*\* (admin) |

Finally, whenever the Wirecloud code is updated, the database must be migrated (and this is one of those times):

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | # python manage.py migrate |

Note: It is strongly recommended to perform a full database backup before starting to migrate wirecloud to a new version.

## Final steps

The settings.py file allows you to set several options in Wirecloud. If DEBUG is false you will need to collect Wirecloud static files using the following command and answering 'yes' when asked:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ python manage.py collectstatic |

If you use the runserver command (not recommended for production) you will have to call it with the --insecureswitch in order to make it serve the static files when not debugging.

In addition, you should serve the static files with a fast performance http server like [Nginx](http://nginx.org/) or [Apache](http://httpd.apache.org/). Django has documentation for this [topic](https://docs.djangoproject.com/en/dev/howto/deployment/).

Finally, you can compress css and javascript code files for better performance using the following command:

[?](https://conwet.fi.upm.es/wirecloud/database)

|  |  |
| --- | --- |
| 1 | $ python manage.py compress |

Note: Don't forget to rerun the collectstatic and compress commands each time Wirecloud's code is updated. This includes each time an add-on is added or removed.

# Adding other great features

## Integration with Django "sites" framework

Wirecloud uses the hostname provided by the http request when building internal URLs. This behavior is usually good for normal use.

However, when the "sites framework" is installed, Wirecloud make use of it to obtain the domain to use when building internal urls. This is quite useful when the hostname doesn't match the public name of the Wirecloud server.

## Installing the Wirecloud Pub Sub add-on

The development of the Pub Sub add-on is carried out at [github.com]. You can always find the latest information about how to install and use it on the main page of the repository.

Newer versions of the Pub Sub add-on can be installed directly using pip:

[?](https://conwet.fi.upm.es/wirecloud/adding-features)

|  |  |
| --- | --- |
| 1  2  3  4  5 | INSTALLED\_APPS = (      ...      'wirecloud\_pubsub',     ...  ) |

As last step, add a DEFAULT\_SILBOPS\_BROKER setting with the URL of the broker to use:

[?](https://conwet.fi.upm.es/wirecloud/adding-features)

|  |  |
| --- | --- |
| 1 | DEFAULT\_SILBOPS\_BROKER = 'http://pubsub.server.com:8080/silbops/CometAPI' |

Don't forget to run the collectstatic and compress commands on your Wirecloud installation:

[?](https://conwet.fi.upm.es/wirecloud/adding-features)

|  |  |
| --- | --- |
| 1  2 | $ ./manage.py collectstatic  $ ./manage.py compress |

## NGSI proxy

Wirecloud comes with a javascript library that allows widgets and operators to connect to NGSI-9/10 servers. This support comes for free when installing Wirecloud, except for the subscribe operations. These operations requires what is called NGSI proxy. This proxy is a facade that receives NGSI notifications and passes them to the final targets: Widgets and Operators. NGSI proxy doesn't need to be installed in the same machine as Wirecloud and can be shared with other Wirecloud instances.

You can install a NGSI proxy following those steps:

[?](https://conwet.fi.upm.es/wirecloud/adding-features)

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ apt-get install nodejs npm  $ ln -s /usr/bin/nodejs /usr/bin/node  $ git clone git://github.com/conwetlab/ngsijs.git  $ cd ngsijs/ngsi-proxy  $ npm install |

After this, you can run the NGSI proxy issuing the following command:

[?](https://conwet.fi.upm.es/wirecloud/adding-features)

|  |  |
| --- | --- |
| 1 | $ npm run start |

# Running Wirecloud

We recommend running Wirecloud based on an Apache Web Server. However, it is also possible to run it using the Django internal web server, just for testing purposes.

## Running Wirecloud using the Django internal web server

Be aware this way of running Wirecloud should be used just for evaluation/testing purposes. Do not use it in a production environment.

To start Wirecloud, type the following command:

[?](https://conwet.fi.upm.es/wirecloud/running)

|  |  |
| --- | --- |
| 1 | $ python manage.py runserver 0.0.0.0:8080 --insecure |

Then, go to [http://computer\_name\_or\_IP\_address:8080/](https://conwet.fi.upm.es/wirecloud/running) where computer\_name\_or\_IP\_address is the name or IP address of the computer on which Wirecloud is installed, and use the username and password you provided when populating the database to sign in on the platform.

## Integrating Wirecloud with Apache

If you choose to deploy Wirecloud in Apache, the mod\_wsgi module must be installed (and so does Apache!). To do so, type the following command:

[?](https://conwet.fi.upm.es/wirecloud/running)

|  |  |
| --- | --- |
| 1 | $ sudo apt-get install apache2 libapache2-mod-wsgi |

Once you have installed Apache and mod\_wsgi, add a VirtualHost to the Apache's configuration files. For example, in Debian and Ubuntu, you can edit the /etc/apache2/sites-available/default configuration file:

[?](https://conwet.fi.upm.es/wirecloud/running)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <virtualhost \*:80="">          ...          ### Wirecloud ###          WSGIPassAuthorization On            WSGIDaemonProcess wirecloud python-path=<path\_to\_wirecloud>          WSGIScriptAlias / <path\_to\_wirecloud\_wsgi.py>          <location>                  WSGIProcessGroup wirecloud          </location>            Alias /static <path\_to\_wirecloud>/static          <location "="" static"="">                  SetHandler None                  <ifmodule mod\_expires.c="">                          ExpiresActive On                          ExpiresDefault "access plus 1 week"                  </ifmodule>                  <ifmodule mod\_headers.c="">                          Header append Cache-Control "public"                  </ifmodule>          </location>          <location "="" static="" cache"="">                  <ifmodule mod\_expires.c="">                          ExpiresDefault "access plus 3 years"                  </ifmodule>          </location>          ...  </path\_to\_wirecloud></path\_to\_wirecloud\_wsgi.py></path\_to\_wirecloud></virtualhost> |

Assuming that your wirecloud instance is available at /opt/wirecloud\_instance, you should have something similar to:

[?](https://conwet.fi.upm.es/wirecloud/running)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <virtualhost \*:80="">          ...          ### Wirecloud ###          WSGIPassAuthorization On            WSGIDaemonProcess wirecloud python-path=/opt/wirecloud\_instance          WSGIScriptAlias / /opt/wirecloud\_instance/wirecloud\_instance/wsgi.py          <location>                  WSGIProcessGroup wirecloud          </location>            Alias /static /opt/wirecloud\_instance/static          <location "="" static"="">                  SetHandler None                  <ifmodule mod\_expires.c="">                          ExpiresActive On                          ExpiresDefault "access plus 1 week"                  </ifmodule>                  <ifmodule mod\_headers.c="">                          Header append Cache-Control "public"                  </ifmodule>          </location>          <location "="" static="" cache"="">                  <ifmodule mod\_expires.c="">                          ExpiresDefault "access plus 3 years"                  </ifmodule>          </location>          ...  </virtualhost> |

Once you have the site enabled, restart Apache

[?](https://conwet.fi.upm.es/wirecloud/running)

|  |  |
| --- | --- |
| 1 | # apache2ctl graceful |

and go to [http://computer\_name\_or\_IP\_address/](https://conwet.fi.upm.es/wirecloud/running) to get into Wirecloud.

-----######################----

Developer's Guide

Don't you see a widget that fits your needs? Are you thinking of adapting one of the existing widgets to make it useful in your mashup? If you are looking for information about how to develop widgets or operators, these are your pages!

Before starting the creation of a widget, you must be aware of certain design principles of the widgets

* Widgets are supposed to be small, reusable and user centric web applications.
* Generic widgets are desirable, but ad-hoc solutions are allowed too if they are quick and cheap enough.
* Widgets should be adapted to real problems.
* Widgets are elements of the front-end layer (View). It's not desirable for widgets to perform back-end layer functions (controller or model) because they can be provided by the platform (persistent state).
* While developing widgets, any of the technologies accepted by web browsers (XHTML, JavaScript, SVG, Flash, applets...) can be used.

It is also important to mention that widgets are formed by three different components:

* The *Template*, which is a declarative description of the widget using the Widget Description Language. It represents its main entry point and contains, among other things, references to the rest of resources of a widget.
* The *Code*, composed of HTML, JavaScript and CSS files containing the definition and the behaviour of the widget. Here is where the JavaScript API provided by Wirecloud will help you.
* Some *Static resources*, such as images, documentation and so on.

This guide covers the following topics:

* The Widget Description Language
* The JavaScript API
* The Publish/Subscribe API
* The NGSI API
* How to develop the widgets of my own weather mashup (yes, it's a tutorial! :-) ): It teaches you on how to request data to some weather service, how to add input/output endpoints to widgets to wire them together, and finally, test the newly created application mashup.

# The Widget Description Language

First of all, widgets templates defined in XML should use the http://wirecloud.conwet.fi.upm.es/ns/template# namespace for the root element

## Description

Contains all the widget contextual information in an XML element called Catalog.ResourceDescription. This element is compulsory and only one element can be present in the XML document.

* Vendor: The distributor of the widget. It cannot contain the character "/".
* Name: Name of the widget. It cannot contain the character "/".
* Version: Current version of the widget. It must define starting sequences of numbers separated by dots. Moreover, zeros can only be used alone (e.g. 0.1 is valid but 03.2 is not).

These tree fields (vendor, name and version) uniquely identify the widget, therefore there can not be a repetition of such identifier in any collection of wirecloud resources (including widgets, mashups, operators, ...).

* DisplayName: Name used in the user interface for the widget. This field can be translated, therefore this field is not used to uniquely identify the widget.
* Author: Developer of the widget.
* Mail: E-mail address to get in touch with the developer(s).
* Description: A brief textual description of the widget.
* ImageURI: Absolute or template-relative URL of the widget image for the catalog.
* iPhoneImageURI: Image to be used in iPhone and other smartphones.
* WikiURI: Absolute or template-relative URL of the widget documentation.

## Integration Variables and Platform Elements

The variables that uses the widget to interact with the environment, associating concepts with aspects are defined in this block. Likewise, it also defines some other elements of the interface, such as the initial size of the widget. All of them are managed by the platform, which will ensure their persistence.

## Platform.Preferences

The user preferences, which may be changed through the platform interface. It's a mandatory element, consisting of one, several or even none preference sub-elements:

### Preference

Defines a user preference. It has the following required attributes:

* name: Name of the variable to reference it in the code.
* type: Data type of the variable: text (string), number, boolean, password and list.
* description: Descriptive text.
* label: Label which the variable will be shown in the user interface.
* default: Default value.

The Preference elements of type list specify the available choices using the Option element that defines an item of the selection list. It has the following attributes:

* name: Text to display in the selection list.
* value: The value used when the option is selected.

## Platform.StateProperties

This element contains some variables that reflects the persitant widget state. The state can be any information desired to be persisted. It's required element and contains a list of property definitions:

### Property

Defines a state variable. It has the following required attributes:

* name: Name of the variable.
* type: Data type of the variable. So far only the type text (string) is allowed.
* label: Label to be displayed in the user interface.

## Platform.Wiring

Defines the list of variables to communicate with other widgets. It may contain any number of these elements:

### OutputEndpoint

Widgets may send events through an output endpoint. But before they can use these output endpoints they must declare them using the OutputEndpoint element. OutputEndpoint elements require the following attributes:

* name: Name of the output endpoint.
* type: Data type of the output endpoint. So far only the type text (string) is allowed.
* label: Label to be displayed in the user interface.
* description: Descriptive text.
* friendcode: Keyword used to tag the output endpoint, so it can be easily suggested valid conection during the wiring process.

### InputEndpoint

Define an input endpoint which is going to used by the widget for receiving events from other widgets. Each input endpoint must have the following attributes:

* name: Name of the input endpoint.
* type: Data type of the input endpoint. So far only the type text (string) is allowed.
* label: Label to be displayed in the user interface.
* action\_label: Short text describing what is going to happen if an event is sent to this input endpoint. Other widgets will use this text in buttons, selection boxes, etc... allowing end users to select what to do (and the widget will send a event to the associated target endpoint)
* description: Descriptive text.
* friendcode: Keyword used to tag the input endpoint, so it can be easily suggested valid conection during the wiring process.

## Platform.Link

Widget source code related to the template. It's formed by an unique element:

### XHTML

This elements is used to link the template with the code of the widget:

* href: Absolute or template-relative URL of widget code.
* content-type: Content type of the linked resource. Suggested values are: text/html and application/xml+xhtml. Optional attribute, 'text/html' by default.
* cacheable: Whether this code can be cached by the platform. Possible values are "true" and "false". Optional attribute, "true" by default.
* use-platform-style: Use platform style to display HTML elements. Optional attribute, "false" by default.

## Platform.Rendering

Contains information about how to show the widget.

### width

Initial width of the widget in cells.

### height

Initial height of the widget in cells.

# The Widget JavaScript API

The Widget Javascript API allow Widgets to access the funcionalities offered by the Mashup Execution Engine like widget interconnection, state persistence, access to the cross-domain proxy, ...

## MashupPlatform.http

### buildProxyURL

Builds a URL suitable for working around the cross-domain problem. This usually is handled using the wirecloud proxy but it also can be handled using the access control request headers if the browser has support for them. If all the needed requirements are meet, this function will return a URL without using the proxy.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.http.buildProxyURL(url, options) |

* url is the target URL.
* options is an object with request options (see the request options section for more details).

### makeRequest

Sends a HTTP request.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.http.makeRequest(url, options) |

* url is the target URL of the request.
* options is an object with a list of request options (see the request options section for more details).

### request options

#### General options:

* asynchronous (Boolean; default true): Determines whether XMLHttpRequest is used asynchronously or not. Synchronous usage is strongly discouraged — it halts all script execution for the duration of the request and blocks the browser UI.
* contentType (String; default application/x-www-form-urlencoded): The Content-type header for your request. Change this header if you want to send data in another format (like XML).
* encoding (String; default UTF-8): The encoding for the contents of your request. It is best left as-is, but should weird encoding issues arise, you may have to tweak this.
* method (String; default POST): The HTTP method to use for the request. The other common possibilities are GET, PUTand DELETE.
* parameters (Object): The parameters for the request, which will be encoded into the URL for a get method, or into the request body for the other methods.
* postBody (String): Specific contents for the request body on a post method. If it is not provided, the contents of the parameters option will be used instead.
* requestHeaders (Object): A set of key-value pairs, with properties representing header names.
* forceProxy (Boolean; default false): Sends the request through the proxy regardless of the other options passed.
* context (Object; default null) is the value to be passed as the this parameter to the callbacks.

#### Callback options:

* onSuccess: Invoked when a request completes and its status code belongs in the 2xy family. This is skipped if a code-specific callback is defined (e.g., on200), and happens before onComplete.
* onFailure: Invoked when a request completes and its status code exists but is not in the 2xy family. This is skipped if a code-specific callback is defined (e.g. on403), and happens before onComplete.
* onXYZ (with XYZ representing any HTTP status code): Invoked just after the response is complete if the status code is the exact code used in the callback name. Prevents execution of onSuccess and onFailure. Happens beforeonComplete.
* onException: Triggered whenever an exception is raised while dispatching any of the other callbacks. Has a custom signature: the first argument is the response, and the second is the exception object.
* onComplete: Triggered at the very end of a request's life-cycle, after the request completes, status-specific callbacks are called, and possible automatic behaviors are processed. Guaranteed to run regardless of what happened during the request.

All these callbacks get the response object in the first argument. onException, in addition, receives the exception object as the second argument.

## MashupPlatform.wiring

### pushEvent

Sends an event through the wiring.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.wiring.pushEvent(outputName, data) |

* outputName is the name of the output endpoint as defined in the WDL.
* data is the content of the event.

### registerCallback

Registers a callback for a given input endpoint. If the given endpoint already has registered a callback, it will be replaced by the new one.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.wiring.registerCallback(inputName, callback) |

* inputName is name of the input endpoint as defined in the WDL.
* callback is the callback function to use when an event reaches the given input endpoint.

## MashupPlatform.prefs

### get

Retrieves the value of a preference.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.prefs.get(key) |

* key is the preference to fetch.

### set

Sets the value of a preference.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.prefs.set(key, value) |

* key is the identifier of the preference.
* value is the new value to use for the preference.

### registerCallback

Registers a callback for listening to preference changes.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.prefs.registerCallback(callback) |

* callback is the callback function that will be called when the preferences of the widget changes.

## MashupPlatform.widget

### getVariable

Returns a widget variable by its name.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.widget.getVariable(name) |

* name is the name of the variable to retreive.

### drawAttention

Makes wirecloud notify that the widget needs user's attention.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.widget.drawAttention() |

### id

Returns the widget id.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.widget.id |

### log

Writes a message into the wirecloud's log console.

[?](https://conwet.fi.upm.es/wirecloud/widgetapi)

|  |  |
| --- | --- |
| 1 | MashupPlatform.widget.log(msg, level) |

* msg is the text of the message to log.
* level is an optional parameter with the level to uses for logging the message. (By default: info).

# The PUB/SUB API

Wirecloud comes with [SilboPS](https://svn.forge.morfeo-project.org/4caast/trunk/WP6/pubsub/), a publish/subscribe add-on ready to be used. Both widgets and operators declaring the use of the PubSub feature can take advantage of the PubSub functionalities through the MashupApplication.SilboPSobject.

Currently, the MashupApplication.SilboPS object only exports the PubEndPoint, SubEndPoint and Filterclasses defined by the original javascript bindings provided by SilboPS. Full documentation of SilboPS is available at<https://svn.forge.morfeo-project.org/4caast/trunk/WP6/pubsub/README.md>.

## Examples of usage

### Widget description using the XML flavor of the WDL

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | <?xml version="1.0" encoding="UTF-8"?>  <Template xmlns="http://wirecloud.conwet.fi.upm.es/ns/template#">        <Catalog.ResourceDescription>          <Vendor>CoNWeT</Vendor>          <Name>tourist-social-comments</Name>          <DisplayName>Tourist - Social comments</DisplayName>          <Version>0.27</Version>          <Author>UPM</Author>          <Mail>4caast\_at\_conwet.es</Mail>          <Description>Chat widget for commenting about tourist locations.                    Uses Pub/Sub as communication channel</Description>          <ImageURI>images/tourist-social.png</ImageURI>            <Requirements>              <Feature name="PubSub" />          </Requirements>        </Catalog.ResourceDescription>        <Platform.Wiring>          <InputEndpoint name="place" type="text" description="publish location"                            label="Messages Location" friendcode="connect\_location" />      </Platform.Wiring>        <Platform.Link>          <XHTML href="/wirecloud/ps.html"/>      </Platform.Link>        <Platform.Rendering width="9" height="25"/>    </Template> |

### Widget description using the RDF flavor of the WDL

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | @prefix dcterms: <http://purl.org/dc/terms/> .  @prefix foaf: <http://xmlns.com/foaf/0.1/> .  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .  @prefix usdl: <http://www.linked-usdl.org/ns/usdl-core#> .  @prefix vcard: <http://www.w3.org/2006/vcard/ns#> .  @prefix wire: <http://wirecloud.conwet.fi.upm.es/ns/widget#> .    <http://wirecloud.conwet.fi.upm.es/ns/widget#CoNWeT/tourist-social-comments/0.27> a wire:Widget;      dcterms:creator [ a foaf:Person;              foaf:name "UPM" ];      dcterms:description "Chat widget for commenting about tourist locations. It uses Pub/Sub";      dcterms:title "tourist-social-comments";      wire:displayName "Tourist - Social comments";      wire:hasImageUri <images/tourist-social.png>;      wire:hasPlatformRendering [ a wire:PlatformRendering;              wire:renderingHeight "25";              wire:renderingWidth "9" ];      wire:hasPlatformWiring [ a wire:PlatformWiring;              wire:hasInputEndpoint [ a wire:InputEndpoint;                      rdfs:label "Messages Location";                      dcterms:description "publish location";                      dcterms:title "place";                      wire:friendcode "connect\_location";                      wire:inputActionLabel "None";                      wire:type "text" ] ];      wire:hasRequirement [ a wire:Feature;              rdfs:label "PubSub" ];      usdl:hasProvider [ a <http://purl.org/goodrelations/v1#BusinessEntity>;              foaf:name "CoNWeT" ];      usdl:utilizedResource <ps.html>;      usdl:versionInfo "0.27";      vcard:addr [ a vcard:Work;              vcard:email "4caast\_at\_conwet.es" ] .    <ps.html> a usdl:Resource;      wire:codeCacheable "True" . |

### Publishing

1. Get a PubEndpoint from the SilboPS.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var myPubEndPoint = new MashupPlatform.SilboPS.PubEndPoint({      open: function(endpoint) {               // react to onopen.      },      close: function(endpoint) {               // react to onclose.      }  ); |

1. Create an Advertise object.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3 | var advertise = new MashupPlatform.SilboPS.Advertise()       .attribute("number", MashupPlatform.SilboPS.Type.LONG)       .attribute("other", MashupPlatform.SilboPS.Type.STRING); |

1. Advertise it.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1 | myPubEndPoint.advertise(advertise); |

1. Create a Notification object.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3 | var notification = new MashupPlatform.SilboPS.Notification()       .attribute("number", MashupPlatform.SilboPS.Type.LONG, 5)       .attribute("other", MashupPlatform.SilboPS.Type.STRING, "don't care"); |

1. Publish it.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1 | myPubEndPoint.publish(notification); |

1. Close the endpoint.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1 | myPubEndPoint.close(); |

### Subscribing

1. Get a PubEndpoint from the MashupPlatform.SilboPS.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var mySubEndPoint = new MashupPlatform.SilboPS.SubEndPoint({      open: function(endpoint) {               // react to onopen.      },      close: function(endpoint) {               // react to onclose.      },      ...      advertise: function(endpoint, advertise) {               // handle advertise      },      unadvertise: function(endpoint, unadvertise) {               // handle unadvertise      },      notify: function(endpoint, notification) {               // handle notification      }  }); |

1. Subscribe to a filter.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | var cxtFunc = new MashupPlatform.SilboPS.ContextFunction();  var stringType = MashupPlatform.SilboPS.Type.STRING;    var filter = new MashupPlatform.SilboPS.Filter()      .constrain("other", stringType).startsWith("don't")      .filter()    mySubEndPoint.subscribe(filter, cxtFunc); |

1. Handle notifications.

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // Use "notify" handler.  // ...      notify: function(endpoint, notification) {               // handle notification      }  //... |

1. Close the endpoint

[?](https://conwet.fi.upm.es/wirecloud/pubsubapi)

|  |  |
| --- | --- |
| 1 | mySubEndPoint.close(); |

# The NGSI API

Both Widgets and operators wishing to use the javascript bindings provided by Wirecloud for accessing the [FI-WARE NGSI Open RESTful API](https://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_NGSI_Open_RESTful_API_Specification_(PRELIMINARY)) in order to seamlessly interoperate with the [Samson Pub/Sub Context Broker](https://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FIWARE.OpenSpecification.Data.PubSub) must add the NGSI feature as a requirement into their description files (config.xml files).

The following is an example of a widget description using the XML flavor of the WDL:

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <?xml version="1.0" encoding="UTF-8"?>  <Template xmlns="http://wirecloud.conwet.fi.upm.es/ns/template#">      <Catalog.ResourceDescription>          <Vendor>CoNWeT</Vendor>          <Name>observation-reporter</Name>          <DisplayName>Observation Reporter</DisplayName>          <Author>aarranz</Author>          <Version>1.0</Version>          <Mail>aarranz\_at\_conwet.com</Mail>          <Description>Creates a new observation</Description>          <ImageURI>images/catalogue.png</ImageURI>          <iPhoneImageURI>images/smartphone.png</iPhoneImageURI>          <WikiURI>http://www.envirofi.eu/</WikiURI>            <Requirements>              <Feature name="NGSI"/>          </Requirements>        </Catalog.ResourceDescription>        <Platform.Link>          <XHTML href="/wirecloud/index.html" contenttype="text/html" cacheable="true" use-platform-style="true"/>      </Platform.Link>        <Platform.Rendering width="5" height="20"/>  </Template> |

The RDF flavor of the same widget description is:

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | <?xml version="1.0" encoding="utf-8"?>  <rdf:RDF    xmlns:foaf="http://xmlns.com/foaf/0.1/"    xmlns:wire="http://wirecloud.conwet.fi.upm.es/ns/widget#"    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"    xmlns:usdl="http://www.linked-usdl.org/ns/usdl-core#"    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"    xmlns:ns1="http://purl.org/goodrelations/v1#"    xmlns:dcterms="http://purl.org/dc/terms/"    xmlns:vcard="http://www.w3.org/2006/vcard/ns#"  >    <wire:Widget rdf:about="http://wirecloud.conwet.fi.upm.es/ns/widget#CoNWeT/observation-reporter/1.0">      <vcard:addr>        <vcard:Work rdf:nodeID="Nb17ce611aa2645e488515f86eb855e53">          <vcard:email>aarranz\_at\_conwet.com</vcard:email>        </vcard:Work>      </vcard:addr>      <usdl:utilizedResource>        <usdl:Resource rdf:about="index.html">          <wire:codeCacheable>True</wire:codeCacheable>        </usdl:Resource>      </usdl:utilizedResource>      <wire:hasPlatformWiring>        <wire:PlatformWiring rdf:nodeID="Neecb97db81ed40859b8c04e935a9a9cc"/>      </wire:hasPlatformWiring>      <wire:displayName>Observation Reporter</wire:displayName>      <wire:hasiPhoneImageUri rdf:resource="images/smartphone.png"/>      <usdl:versionInfo>1.0</usdl:versionInfo>      <usdl:hasProvider>        <ns1:BusinessEntity rdf:nodeID="N9a6bf56577c741ac806997a80281afff">          <foaf:name>CoNWeT</foaf:name>        </ns1:BusinessEntity>      </usdl:hasProvider>      <wire:hasImageUri rdf:resource="images/catalogue.png"/>      <wire:hasPlatformRendering>        <wire:PlatformRendering rdf:nodeID="N713e5ea11dce4750a592c754c748def7">          <wire:renderingHeight>20</wire:renderingHeight>          <wire:renderingWidth>5</wire:renderingWidth>        </wire:PlatformRendering>      </wire:hasPlatformRendering>      <wire:hasRequirement>        <wire:Feature rdf:nodeID="N3cb336bd9b6243ecbf345c80442498f9">          <rdfs:label>NGSI</rdfs:label>        </wire:Feature>      </wire:hasRequirement>      <dcterms:title>observation-reporter</dcterms:title>      <dcterms:description>Creates a new observation</dcterms:description>      <dcterms:creator>        <foaf:Person rdf:nodeID="Ndb72cb5a7f3844b29b72f304baaa14a7">          <foaf:name>aarranz</foaf:name>        </foaf:Person>      </dcterms:creator>    </wire:Widget>  </rdf:RDF> |

Once the NGSI feature is added to the widget/operator description file, widgets and operators will have access to the NGSI javascript object that conforms the core of the API. See the [Publish/Subscribe Context Broker - SAMSON Broker - User and Programmer Guide](https://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/Publish/Subscribe_Context_Broker_-_SAMSON_Broker_-_User_and_Programmer_Guide) for more details on the operations that can be invoked using the [RESTful API](https://forge.fi-ware.eu/plugins/mediawiki/wiki/fiware/index.php/FI-WARE_NGSI_Open_RESTful_API_Specification_(PRELIMINARY)). That guide can be used as a reference, due to each of the Pub/Sub Context Broker operations have an equivalent operation in the javascript bindings.

What follows exemplifies the use of this API for updating an entity.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | var connection = new NGSI.Connection('<url of the Samson Pub/Sub Context Broker instance>');  connection.updateAttributes([{      entity: {          id: 'iss8',          type: 'Issue'      },      attributes:[{          name: 'technician',          contextValue: 'tech1'      }]  }], {      onSuccess: function () {          // notify success      },      onFailure: function () {          // show error      }  }); |

## Data types used by the library

The Entity type is used to reference entities. This type is defined as an object composed of the following fields:

* id is a string with the id of the entity. Some times you will be able to use patterns in this field.
* isPattern is a boolean indicating whether the id field contains a pattern. This field is optional.
* type is the type of the entity. This field is optional.

The Attribute type is used to reference attributes. This type is defined as an object composed of the following fields:

* name is the name of the attribute.
* type is the type of the Attribute. This field is optional.

The Duration type is used to describe time intervals and defined as a string following the format defined at<http://books.xmlschemata.org/relaxng/ch19-77073.html>.

The Condition type is used to declare the condition that will trigger notifications. This type is defined as an object composed of the following fields:

* type is a string containing 'ONTIMEINTERVAL' or 'ONCHANGE'.
* values is an array of string. The meaning of this field depends on the value of the typefield:
  + 'ONTIMEINTERVAL': exactly one value SHALL be present representing the time interval between notifications (using the Duration type).
  + 'ONCHANGE': this element SHALL contain the name(s) of the Context Attributes to be monitored for changes.

The AttributeValue type is used to assign values to attributes. This type is defined as an object composed of the following fields:

* name is the name of the attribute.
* type is the type of the attribute. This field is optional.
* contextValue is the value to assign to the attribute.

The AttributeUpdate type is used to describe a context update. This type is defined as an object composed of the following fields:

* entity is the entity affected by the update. Type: Entity.
* attributes is the new values for the attributes of the entity. Type: AttributeValue.

The AttributeDeletion type is used to describe the deletion of attributes from an entity. This type is defined as an object composed of the following fields:

* entity is the entity affected by the update. Type:Entity.
* attributes is the new values for the attributes of the entity. Type:Attribute.

## NGSI.Connection

A new NGSI.Connection can be instantiated using the following constructor:

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | NGSI.Connection(url[, option]) |

* url is the url of the Samson Pub/Sub Context Broker instance.
* optionsis an object with extra options. This parameter may be null if no extra option is needed. Current supported options are:
  + ngsi\_proxy is the url of the NGSI proxy used for subscriptions.

All the methods of NGSI.Connection supports an options parameter. This parameter is used, among other things, to pass callbacks functions. This parameter is a JavaScript object containing pairs of key-value options. Moreover, all methods of NGSI.Connectionsupport at least the following callbacks:

* onSuccess is called when the request finishes successfully. The parameters passed to this callback depends on the invoked method.
* onFailure is called when the request finish with errors.
* onComplete is called when the request finish regardless of whether the request is successful or not.

### createRegistration

Register context information (entities and attributes) in the NGSI server.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | createRegistration(entities, attributes, duration, providingApplication, options) |

* entities is the list of Entities that are going to be registered.
* attributes is a list of the Attributes that are going to be assigned to the previous list of entities.
* duration is the Duration for this registration.
* providingApplication is the URI of the application to which this registration belongs to.

The onSuccess callback will receive an object with the following fields:

* registrationId is the final assigned id. This id can be used in the updateRegistration andcancelRegistration methods.
* duration is the final assigned duration for this registration.

### updateRegistration

Updates a particular registration.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | updateRegistration(regId, entities, attributes, duration, providingApplication[, options]) |

* regId is the id of the registration to update.
* entities is the list of Entities that its going to replace the previous established one.
* attributes is a list of the Attributes that are going to be assigned to the provided list of entities.
* duration is the new Duration for the registration identified by regId.
* providingApplication is the new value for the providingApplication property of the registration.

The onSuccess callback will receive an object with the following fields:

* registrationId is the id of the registration.
* duration is the final assigned duration for this registration.

### cancelRegistration

Cancels or deletes a particular registration.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | cancelRegistration(regId[, options]) |

* regId is the id of the registration to cancel.

### discoverAvailability

Discover context information registrations in the NGSI server.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | discoverAvailability(entities, attributeNames, options) |

* entities is the list of Entities that are going to be queried.
* attributeNames is the list of attribute names that are going to be queried. This parameter is optional and thusnull is a valid value.

### query

Query for context information.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | query(entities, attributeNames[, options]) |

* entities is the list of Entities to query.
* attributeNames is the list of attribute names to query.

### updateAttributes

Update context information.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | updateAttributes(update[, options]) |

* update a list of AttributeUpdates.

### addAttributes

Add attributes to entities.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | addAttributes(toAdd[, options]) |

* toAdd a list of AttributeUpdates.

### deleteAttributes

Delete attributes form entities.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | deleteAttributes(toDelete[, options]) |

* toDelete a list of AttributeDeletion.

### createSubscription

Subscribe to changes in the context information.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | createSubscription(entities, attributeNames, duration, throttling, conditions[, options]) |

* entities is the list of Entities to query in this subscription.
* attributeNames is the list of attribute names to query in this subscription.
* duration is the Duration of this subscription.
* throttling is the proposed minimum interval between notifications. This value must be provided using theDuration type. null is also valid.
* conditions is a list of Conditions that will trigger queries using the provided information and their subsequent notifications to the onNotify callback.

This method, supports a new type of callback: onNotify. This callback is required and can be either an URL or a function. In the later case, the NGSI Connection must be created using a NGSI proxy and will be called every time a notification comes from the NGSI server. The first parameter of a onNotify callback function will be an object with the response data.

### updateSubscription

Update context subscription.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | updateSubscription(subId, entities, attributeNames, duration, throttling, conditions[, options]) |

* subId is the id of the context subscription to cancel.
* entities is the list of Entities to query in this subscription.
* attributeNames is the list of attribute names to query in this subscription.
* duration is the Duration of this subscription.
* throttling is the proposed minimum interval between notifications. This value must be provided using theDuration type. null is also valid.
* conditions is a list of Conditions that will trigger queries using the provided information and their subsequent notifications to the onNotify callback.

### cancelSubscription

Cancels or deletes context subscription.

[?](https://conwet.fi.upm.es/wirecloud/ngsiapi)

|  |  |
| --- | --- |
| 1 | cancelSubscription (subId[, options]) |

* subId is the id of the context subscription to cancel.

# Tutorial. How to develop widgets

In this tutorial we are going to implement a Weather Widget as an excuse to learn how to use the most commonly used features of Wirecloud, from a point of view of Widgets. Our intention is to create a Widget capable of making AJAX request to a external service and of communicating with other widgets in a mashup. This guide does not cover the development of the widget's user interface, which doesn't have to do with Wirecloud and it's based on standard HTML, JavaScript and CSS code.

First of all, you can download a "little" initial code from this [link](https://conwet.fi.upm.es/docs/download/attachments/1278018/Example1Skel.zip?version=3&modificationDate=1373275046000).

Then, you will need to create a new API key for the Weather Underground API using this [link](http://www.wunderground.com/weather/api/d/login.html).

Once you have the key, it's time to learn how to make requests to the service.

## Making request to Weather Underground

Our widget is going to provide the weather forecast for a given location, this location is going to be defined by a coordinate.

Weather Underground provide a rest API for this for this purpose (documented [here](http://www.wunderground.com/weather/api/d/docs)), but we cannot access this API using normal AJAX request (using XMLHttpRequest) due browsers applying the same origin policy to javascript code. Fortunately, Wirecloud provides the MashupPlatform.http.makeRequest method for dealing with this problem. A possible way to access to this API is by using the following code:

[?](https://conwet.fi.upm.es/wirecloud/tutorial)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | var getForecastByCoord = function getForecastByCoord(coord, onSuccess, onError) {      var url;      if ((typeof onSuccess !== 'function') || (typeof onError !== 'function')) {          throw new TypeError();      }      url = 'http://api.wunderground.com/api/' + API\_KEY + '/conditions/forecast/q/';      url += coord.lat + ',' + coord.lon;      url += '.json';      MashupPlatform.http.makeRequest(url, {          method: 'GET',          onSuccess: function (transport) {              var response;              response = JSON.parse(transport.responseText);              if (response.response.error) {                  onError();              } else {                  onSuccess(response);              }          },          onError: function (transport) {              onError();          }      });  }; |

The getForecastByCoord function makes the appropriated request to Weather Underground and passes the result to theonSuccess callback.

In the next section we'll learn how to prepare a widget to receive events from other widgets.

# Tutorial. How to develop widgets (2)

## Adding an input endpoint

Input endpoints must be declared into the widget template before it can be used by the javascript code of the widget. To do so, open template.xml and add an InputEndpoint element into the Platform.Wiring section. The final result should look like:

[?](https://conwet.fi.upm.es/wirecloud/tutorial/2)

|  |  |
| --- | --- |
| 1  2  3  4  5 | ...      <Platform.Wiring>          <InputEndpoint name="coord" type="text" label="Show forecast by coord" description="Shows the weather forecast for a given location (a latitude longitude coordinate)." friendcode="location"/>      </Platform.Wiring>  ... |

* The name attribute will be use to reference to the input endpoint when using the javascript API.
* The label attribute will be used mainly in the Wiring Editor and will be the official name by which end users will know the input endpoint. Also, this attribute can be translated whereas the name attribute not.
* The description attribute is used to provided end user with a description of what is going to happen if an event arrives the input endpoint. This description is very important for the wiring process as the user needs this information for taking decisions on howto wire widgets.
* The friendcode is used by the Wiring Editor to provide basic wiring recommendations. In this case, we are declaring that we accept data produced by output endpoints with a friendcode of "location". The format of this data is a string with the longitude and the latitude separated by a comma.

The following snippet shows how to declare the input endpoint when using RDF (turtle):

[?](https://conwet.fi.upm.es/wirecloud/tutorial/2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | ...  wire:hasPlatformWiring [ a <http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformWiring>;          wire:hasInputEndpoint [ a <http://wirecloud.conwet.fi.upm.es/ns/widget#InputEndpoint>;                  rdfs:label "Forecast location";                  dcterms:description "This event is launched when the user clicks on the location name of current forecast.";                  dcterms:title "location\_coord";                  wire:friendcode "location";                  wire:type "text" ] ];  ... |

Once declared the input endpoint in the widget template, you can register a callback for this endpoint making use of theMashupPlatform.wiring.registerCallback method. In addition to registering the input endpoint, we need to process event data before using it and to notify the user that the forecast data for the given location is being requested. This can be accomplished by using the following code:

[?](https://conwet.fi.upm.es/wirecloud/tutorial/2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | var searchByCoordListener = function searchByCoordListener(event\_data) {      var tmp, coord;      tmp = event\_data.split(',');      coord = {          lat: tmp[1],          lon: tmp[0]      };        startLoadingAnimation();      getForecastByCoord(coord, function (results) {          updateWeatherForecast(results);          stopLoadingAnimation();      }, function () {          clearWeatherForecast();          stopLoadingAnimation();      });  };    MashupPlatform.wiring.registerCallback('search\_coord', searchByCoordListener); |

In the next section we'll learn how to send data by means of events to other widgets.

# Tutorial. How to develop widgets (3)

## Adding an output endpoint

As we did with the input endpoint, we need to declare the new output endpoint in the weather widget's template.

[?](https://conwet.fi.upm.es/wirecloud/tutorial/3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ...      <Platform.Wiring>          <InputEndpoint name="coord" type="text" label="Show forecast by coord" description="Shows the weather forecast for a given location (a latitude longitude coordinate)." friendcode="location"/>          <OutputEndpoint name="location\_coord" type="text" label="Forecast location" description="This event is launched when the user clicks on the location name of current forecast." friendcode="location"/>      </Platform.Wiring>   ... |

Description of the attributes:

* The name attribute will be use to reference to the output endpoint when using the javascript API.
* The label attribute will be used mainly in the Wiring Editor and will be the official name by which end users will know the output endpoint. Also, this attribute can be translated whereas the name attribute not.
* The description attribute is used to provide the end user with a description of when events are sent using this output endpoint. This description is very important for the wiring process, as the user needs this information for taking decisions on how to wire widgets.
* The friendcode is used by the Wiring Editor to provide basic wiring recommendations. In this case, we are declaring that we send data aligned with the friendcode "location".

This is how to declare the output endpoint when using RDF (turtle):

[?](https://conwet.fi.upm.es/wirecloud/tutorial/3)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | ...  wire:hasPlatformWiring [ a <http://wirecloud.conwet.fi.upm.es/ns/widget#PlatformWiring>;          wire:hasInputEndpoint [ a <http://wirecloud.conwet.fi.upm.es/ns/widget#OutputEndpoint>;                  rdfs:label "Show forecast by coord";                  dcterms:description "Shows the weather forecast for a given location (a latitude longitude coordinate).";                  dcterms:title "coord";                  wire:friendcode "location";                  wire:type "text" ] ];          wire:hasOutputEndpoint [ a <http://wirecloud.conwet.fi.upm.es/ns/widget#InputEndpoint>;                  rdfs:label "Forecast location";                  dcterms:description "This event is launched when the user clicks on the location name of current forecast.";                  dcterms:title "location\_coord";                  wire:friendcode "location";                  wire:type "text" ];  ... |

After adding the output endpoint to the widget description, we can send data through it using theMashupPlatform.wiring.pushEvent method. The following code adds an event listener to the location title that sends the location of the current forecast:

[?](https://conwet.fi.upm.es/wirecloud/tutorial/3)

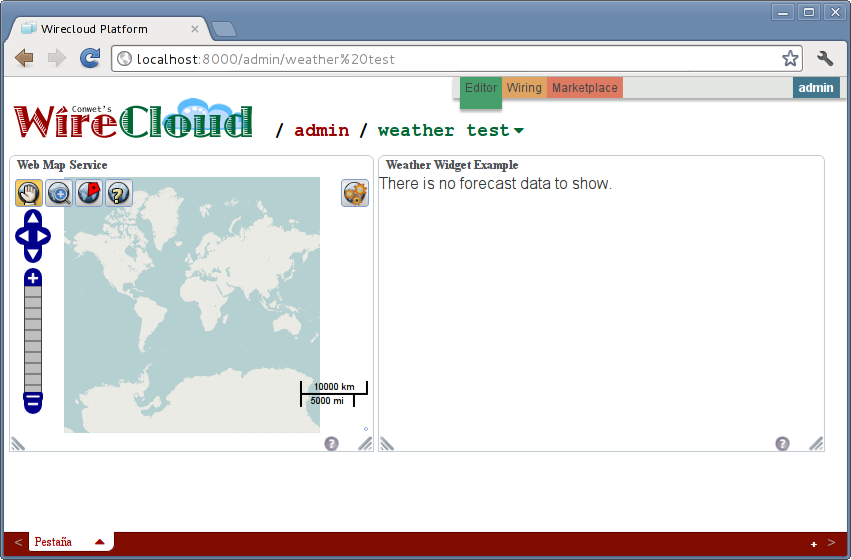
|  |  |
| --- | --- |
| 1  2  3  4  5 | document.getElementById('title').onclick = function (event) {      MashupPlatform.wiring.pushEvent('location\_coord',      forecast\_data.current\_observation.display\_location.longitude + ',' +      forecast\_data.current\_observation.display\_location.latitude);  }; |

In the next section we'll create and test our brand new weather mashup.

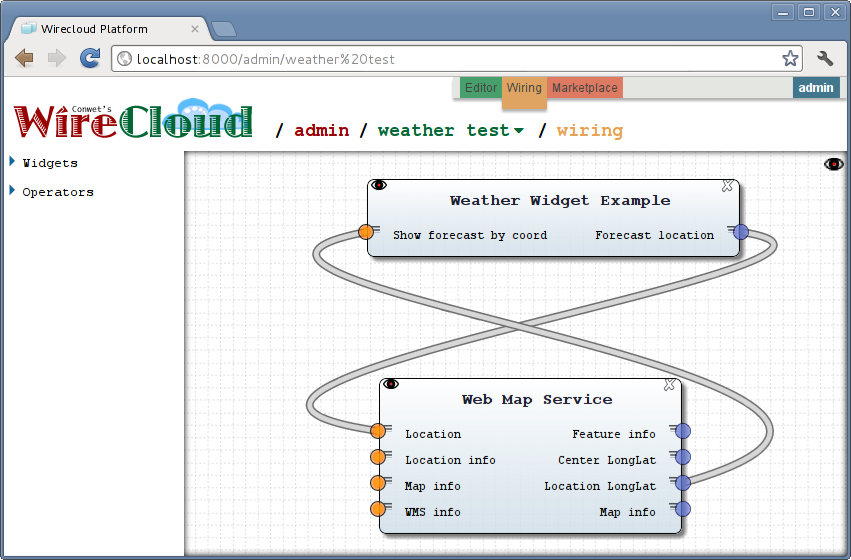
# Tutorial. How to develop widgets (4)

## Testing it

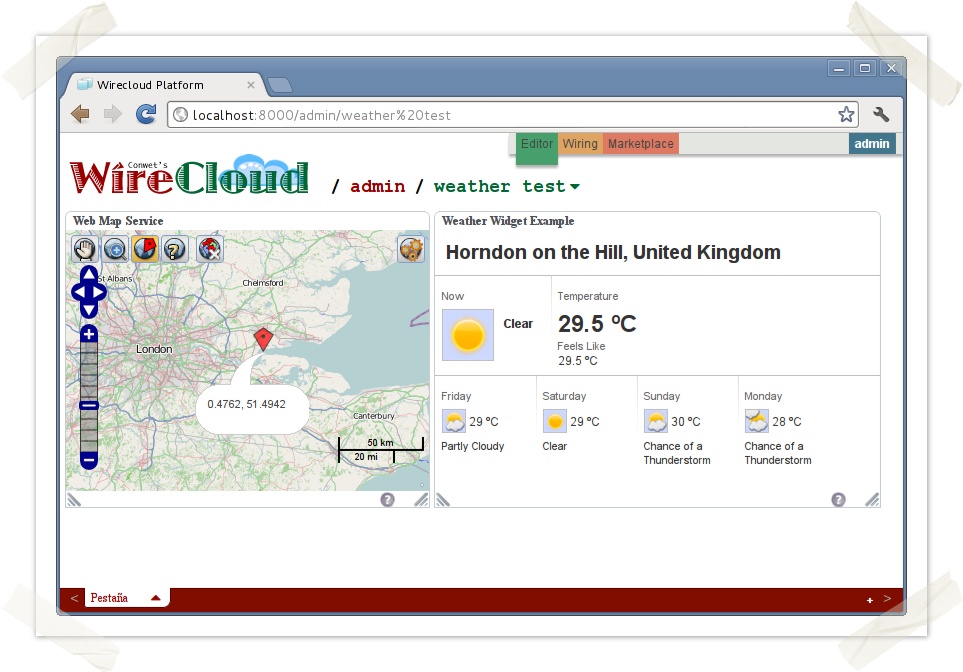
Now that we have implemented the weather widget we can make use of it to test it is working as expected. One of the Widget that should be compatible with our newly created weather widget is the "Web Map Service" Widget so a way to test our widgets is to create a new Workspace with both widgets. Here is a screenshot of a workspace with both widgets:



Now we have to wire them together:



Now it is time to test it!



You can download both [our implementation](https://conwet.fi.upm.es/docs/download/attachments/1278018/CoNWeT_weather-example_1.0.1.wgt?version=6&modificationDate=1356561927000) of the widget as the [mashup example](https://conwet.fi.upm.es/docs/download/attachments/1278018/CoNWeT_weather-mashup-example_1.0.wgt?version=2&modificationDate=1366963686000).

Finally, you should test that sending events from the widget is working. In our implementation of the widget this can be accomplished by clicking on the location title ("Horndon on the Hill, United kingdom" in the screenshot) so visit other places using the Web Map Service Widget and click on the location title to see if the Web Map Service goes back to the forecast location.