

How to Create your Own OpenSignals Plugin

Quick-Guide

QG 02042019

GENERAL DESCRIPTION

Science creates a boundless world where each obstacle is turned into a solution and unexpected ideas give rise to new and amazing discoveries.

However, Scientists/Researchers can only open our world when they are supported by powerful and interactive tools.

OpenSignals intends to be one of these interactive resources, supporting researcher on the simpler tasks, such as acquire, store and view the collected physiological signals, but also on the more difficult ones related with the signal processing and analysis.

The “Open” tag on **OpenSignals** name does not refer exclusively to the capability of the software to open and explore the acquired physiological signals, it also reflects the idea that user contribution is achievable.

In fact, **OpenSignals** is prepared to run custom processing applications created by our community.

However, like in any activity some sequential steps should be followed, being the current quick-guide a reasonable resource for starting the exploration and creation of your own **OpenSignals** add-on.

An extensive list of purchasable add-ons is available at <https://www.biosignalsplux.com/en/software/add-ons>, but you may want to go a little further for answering your research needs...

So, let's start your travel and we strongly hope that you enjoy!

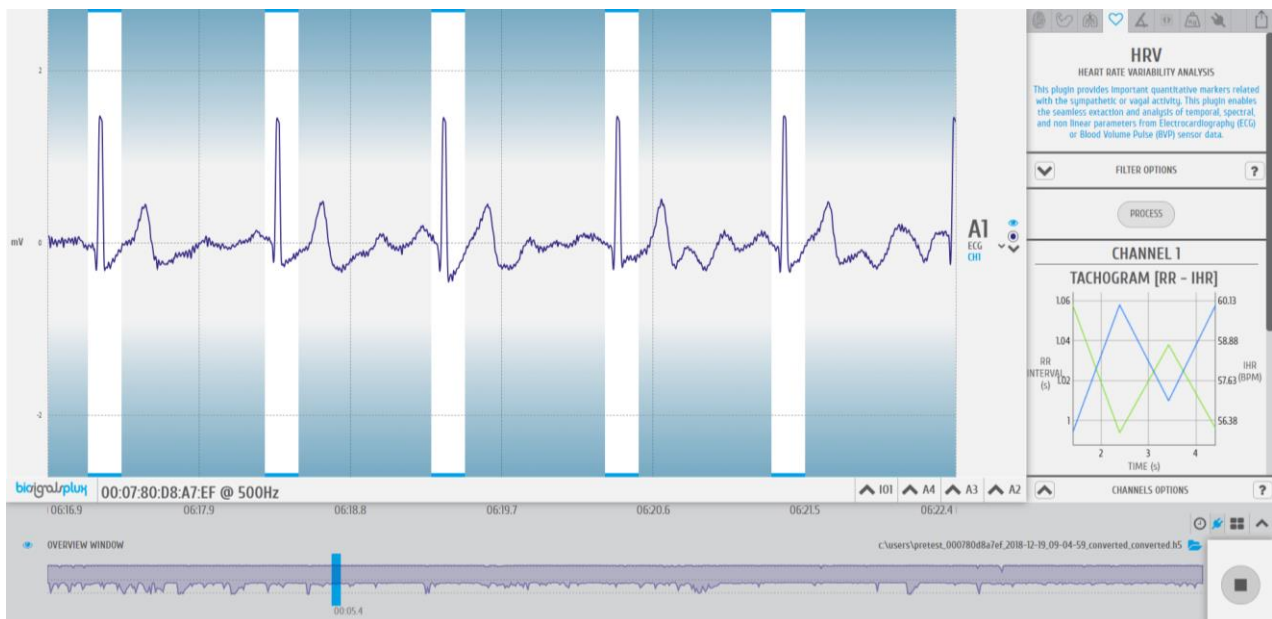


Fig. 1. OpenSignals Heart Rate Variability (HRV) Plugin



opensignals

PLUX – Wireless Biosignals, S.A.
Av. 5 de Outubro, n. 70 – 2.
1050-059 Lisbon, Portugal
plux@plux.info
<http://biosignalsplux.com/>

REV A

© 2019 PLUX

This information is provided "as is," and we make no express or implied warranties whatsoever with respect to functionality, operability, use, fitness for a particular purpose, or infringement of rights. We expressly disclaim any liability whatsoever for any direct, indirect, consequential, incidental or special damages, including, without limitation, lost revenues, lost profits, losses resulting from business interruption or loss of data, regardless of the form of action or legal theory under which the liability may be asserted, even if advised of the possibility of such damages.

PREPARE FILES

An **OpenSignals** plugin requires a set of files to work properly, in this section we will show you how to prepare them to your specific needs.

The required files have the following extensions:

- **.py**
*File dedicated to the signal processing tasks. Signal filtering, segmentation and feature extraction happens here, with a great support of the numerous and diversified **Python** packages.*
[Access an interactive Python tutorial at: <https://www.w3schools.com/python/>]
- **.html**
Inside this file the graphical user interface will be constructed with the insertion of buttons, static text areas, editable text inputs, plots, images...
[Access an interactive HTML tutorial at: <https://www.w3schools.com/html/>]
- **.js**
*With the **JavaScript** file (mainly based on **JQuery**), conferring some dynamism to the static **HTML** interface is possible, namely through events. Interaction with **HTML** interface will trigger a specific event defined inside the **JavaScript** file. When this event was triggered a sequence of instructions will be executed accordingly to the respective **JavaScript** definition of the triggered event.*
As an example, if you click in a button you will want to trigger a specific sequence of actions, for achieving this purpose it will only be necessary to program the “click” event of the button under analysis.
[Access an interactive JavaScript tutorial at: <https://www.w3schools.com/js/>]
- **.css**
*Here you can explore your imagination and creativity, defining the styles to be applied to elements or classes contained inside the **HTML** interface. For example, if you want to change the background colour of your text area or the font size this is the most appropriate file to be an artist.*
[Access an interactive CSS tutorial at: <https://www.w3schools.com/css/>]
- **lang.<plugin_name>.js**
This file is not mandatory, but it helps to keep our solution organised and ensures a quicker and more practical way to globally change a group of strings simultaneously.
*Essentially this file is a dictionary where each key can be invoked on our **JavaScript** file in order to get access to the respective value/string.*
Using an illustrative example, imagine that you use a specific string repeatedly in your plugin, if in the future you want to change the string you would need to change all occurrences individually.
With the present structure, you only need to change once on the dictionary.
- **.xml**
*As stated at **W3Schools** “XML was designed to store and transport data” and in our project the .xml file will store some parameters such as your plugin name, version, icon...*

1. Search for our “template” files

These files are contained inside **OpenSignals** folder. The predefined location on **Microsoft Windows** will be in **C:\Plux\OpenSignals (r)evolution\resources\app** while on **macOS** it should be available at **\Applications\OpenSignals (r)evolution.app\Contents\Resources\app** and finally in **Linux** the predefined folder is **/usr/lib/OpenSignals/resources/app**

On the following steps we will focus our description on **Microsoft Windows** approach, but for other operating systems the fundamental logic remains intact.

Location of each file:

template.py at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\template.py

template.xml at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\template.xml

template.html at C:\Plux\OpenSignals (r)evolution\resources\app\static\template.html

template.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\template.js

template.css at C:\Plux\OpenSignals (r)evolution\resources\app\static\css\template.css

lang.template.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\lang\en\lang.template.js

2. Make a copy of each one of the previous files

Probably you are asking yourself if it is possible to directly use the previous “template” files to create your plugin.

Unfortunately, the “template” filename is included inside the OpenSignals (r)evolution ignore list (in order to avoid that this sample appears for users not interested in developing add-ons).

If all “template” files are renamed, you can use them, however in PLUX we know that your creativity cannot be restricted to a single plugin, and because of that, it is advisable to preserve **template** files to be ready when you will need to create another plugin in the future.

The “copies” of the files should be placed on the same directory where the source file is located.

Returning to our practical example, we will have the following files after the current step (we named our plugin as **myplugin**):

Location of cloned files:

myplugin.py at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\myplugin.py

myplugin.xml at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\myplugin.xml

myplugin.html at C:\Plux\OpenSignals (r)evolution\resources\app\static\myplugin.html

myplugin.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\myplugin.js

myplugin.css at C:\Plux\OpenSignals (r)evolution\resources\app\static\css\myplugin.css

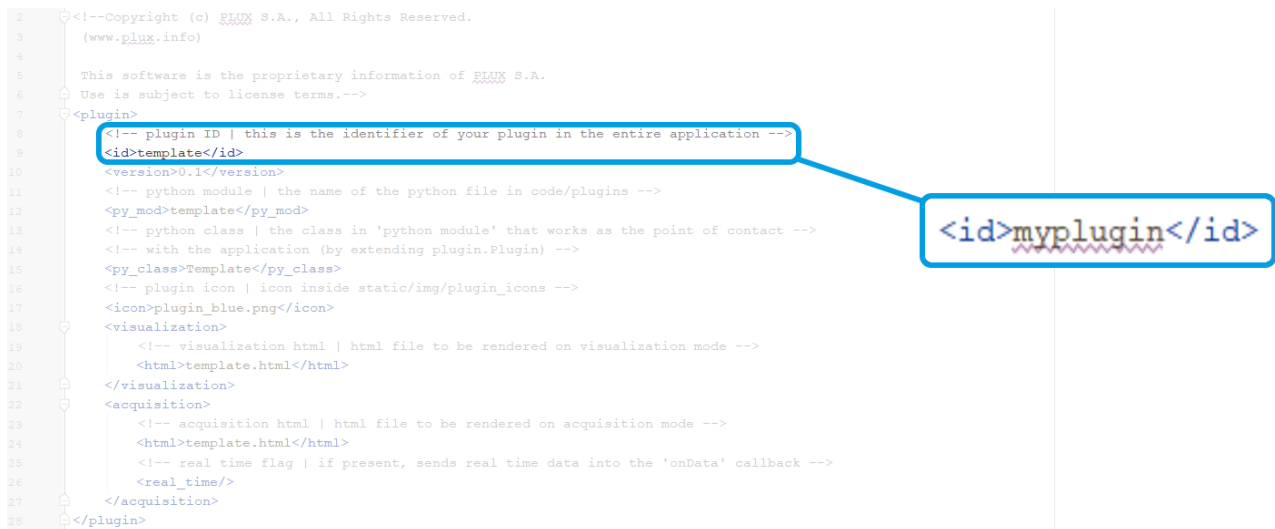
lang.myplugin.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\lang\en\lang.myplugin.js

You should open them, preparing the beginning of this great journey. For this task you can use an *Integrated Development Environment* (IDE) of your choice.

On the current quick-guide we will use **JetBrains PyCharm Community Edition**¹

3. Edit “myplugin.xml” file

3.1. Define a Plugin ID | “...this is the identifier of your plugin in the entire application...”



```
2 <!--Copyright (c) PLUX S.A., All Rights Reserved.
3 (www.plux.info)
4
5 This software is the proprietary information of PLUX S.A.
6 Use is subject to license terms.-->
7 <plugin>
8 <!-- plugin ID | this is the identifier of your plugin in the entire application -->
9 <id>template</id>
10 <version>0.1</version>
11 <!-- python module | the name of the python file in code/plugins -->
12 <py_mod>template</py_mod>
13 <!-- python class | the class in 'python module' that works as the point of contact -->
14 <!-- with the application (by extending plugin.Plugin) -->
15 <py_class>Template</py_class>
16 <!-- plugin icon | icon inside static/img/plugin_icons -->
17 <icon>plugin_blue.png</icon>
18 <visualization>
19 <!-- visualization html | html file to be rendered on visualization mode -->
20 <html>template.html</html>
21 </visualization>
22 <acquisition>
23 <!-- acquisition html | html file to be rendered on acquisition mode -->
24 <html>template.html</html>
25 <!-- real time flag | if present, sends real time data into the 'onData' callback -->
26 <real_time/>
27 </acquisition>
28 </plugin>
```

3.2. Define Plugin version number

```
<!-- plugin ID | this is the identifier of your plugin in the entire application -->
```

3.3 <id>myplugin</id>

```
<version>0.1</version>
```

¹ <https://www.jetbrains.com/pycharm/download/>

How to Create your Own OpenSignals Plugin

Quick-Guide

QG 02042019

As you know, all our Plugin files were renamed to **myplugin** differing only on the extension name.

So, our edition task will be easier, and our memory is also happy because we only need to remember a single name!

```
11 <!-- python module | the name of the python file in code/plugins --> <!-- python module | the name of the python file in code/plugins -->
12 <py_mod>template</py_mod> <py_mod>myplugin</py_mod>
```

3.4. Set `<py_class>` tag value | “...the class in 'python module' that works as the point of contact...”

This field is particularly relevant and needs to be in accordance with the class name inside the **.py** file of our plugin.

First, we should give a name to our class that fulfil the coding conventions, so, we will use our famous **myplugin** name but with uppercase letters “MyPlugin”.

```
13 <!-- python class | the class in 'python module' that works as the point of contact --> <!-- python class | the class in 'python module' that works as the point of contact -->
14 <!-- with the application (by extending plugin.Plugin) --> <!-- with the application (by extending plugin.Plugin) -->
15 <py_class>Template</py_class> <py_class>MyPlugin</py_class>
```

After the previous definition we should rename the class inside our **.py** file:

```
10 -----
11 .. module:: template
12
13 .. moduleauthor:: pgoncalves <pgoncalves@plux.info>
14
15 import ...
16
17
18 class Template(plugin.Plugin):
19     # class that extends plugin.Plugin and becomes the application point of contact.
20     def __init__(self):
21         super(Template, self).__init__()
22         pass
23
24     # Examples of a process functions that will be called from JavaScript
25     def please_avg_these_two_values(self, val1, val2):
26         return (val1 + val2) / 2
27
28     def please_rms_these_two_values(self, val1, val2):
29         return ((val1 ** 2 + val2 ** 2) / 2) ** 0.5
30
31     def send_to_JavaScript(self, arg1, arg2, arg3):
32         # sends to the JavaScript function takeThis:
33         # function takeThis(arg1, arg2, arg3) {}
34         self.javascript.takeThis(arg1, arg2, arg3)
35
36         # sends to the JavaScript function takeThat in Object template:
37         # var template = { takeThat: function(arg1, arg2, arg3) {} }
38         self.javascript.template.takeThat(arg1, arg2, arg3)
```

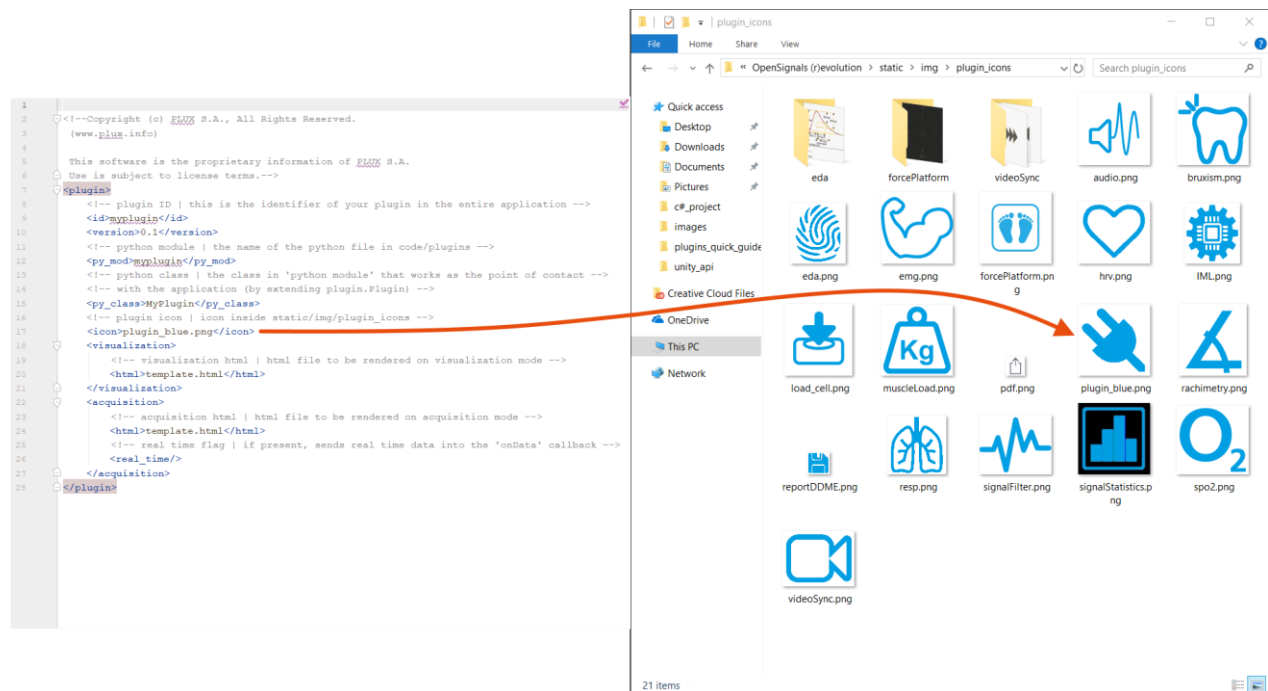


3.5. Choose an icon for identifying your plugin | “...icon inside static/img/plugin_icons...”

Create a new icon for your plugin or use one of the available images inside **C:\Plux\OpenSignals (r)evolution\resources\app\static\img\plugin_icons**.

Note that if you want to create a new icon it should be stored inside the previous directory.

On **<icon>** tag (defined inside our **.xml** file) it should be specified only the image filename, for our case, we will keep the predefined “plugin_blue.png”:



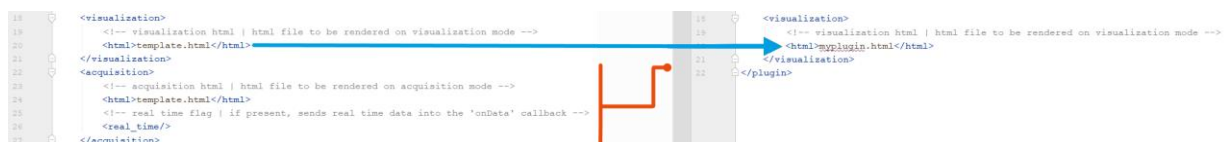
3.6. Definition of the Plugin type

The following two tags: **<visualization>** and **<acquisition>** don't need to be included at the same time. You should remove one of them accordingly to your plugin type (or keep both if your plugin is compatible with offline and real-time processing).

If you are start creating an **offline** plugin you can remove the **<acquisition>** tag and the content inside it.

Otherwise, if you are dedicated on creating a **real-time** plugin, you should remove the **<visualization>** tag and the respective content.

For now, let's focus on **offline** plugins, so, we will remove the **<acquisition>** tag and we need to rename the **<html>** sub-tag value from “template.html” to “myplugin.html”, referencing in which **HTML** file our graphical user interface is located.



4. Replace all “template” references to “myplugin” and “Template” references to “MyPlugin”

You will need to navigate through all files mentioned on step 2 (list replicated below) and use the **“Find and Replace”** option of your IDE, for our specific case of **PyCharm**: Edit > Find > Replace, or shortcut **Ctrl + R** (Windows Users) and **cmd + Ctrl + R** (for Mac users).

A special care is needed on selecting/working with some **Search Options** such as “Match Case” option, because in our plugin “myplugin” will be a distinct reference when compared with “MyPlugin”. Additionally, you should uncheck “Words” option, replacing “template” occurrences on merged words using “_” separator.

A Reminder about the location of cloned files:

myplugin.py at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\myplugin.py

myplugin.xml at C:\Plux\OpenSignals (r)evolution\resources\app\code\plugins\myplugin.xml

myplugin.html at C:\Plux\OpenSignals (r)evolution\resources\app\static\myplugin.html

myplugin.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\myplugin.js

myplugin.css at C:\Plux\OpenSignals (r)evolution\resources\app\static\css\myplugin.css

lang.myplugin.js at C:\Plux\OpenSignals (r)evolution\resources\app\static\lang\en\lang.myplugin.js

5. On our main .html file (myplugin.html) specify .js and .css file

This task was implicitly achieved with step 4, but we think that is relevant to present it in an explicit way, in order to be more intuitive how all files are connected.

In fact, this explicit connection between our **.html** file, containing the graphical user interface, and the **.css** file, where the styles were defined, is really easy.

You simply need to rename the **.css** file on the *href* attribute of **<link>** tag of the **.html** file:



Repeat the same logic on **<script>** tag. Here we should specify our **.js** filename in order to ensure the execution of dynamic events inside the **.html** graphical user interface:



How to Create your Own OpenSignals Plugin

Quick-Guide

QG 02042019

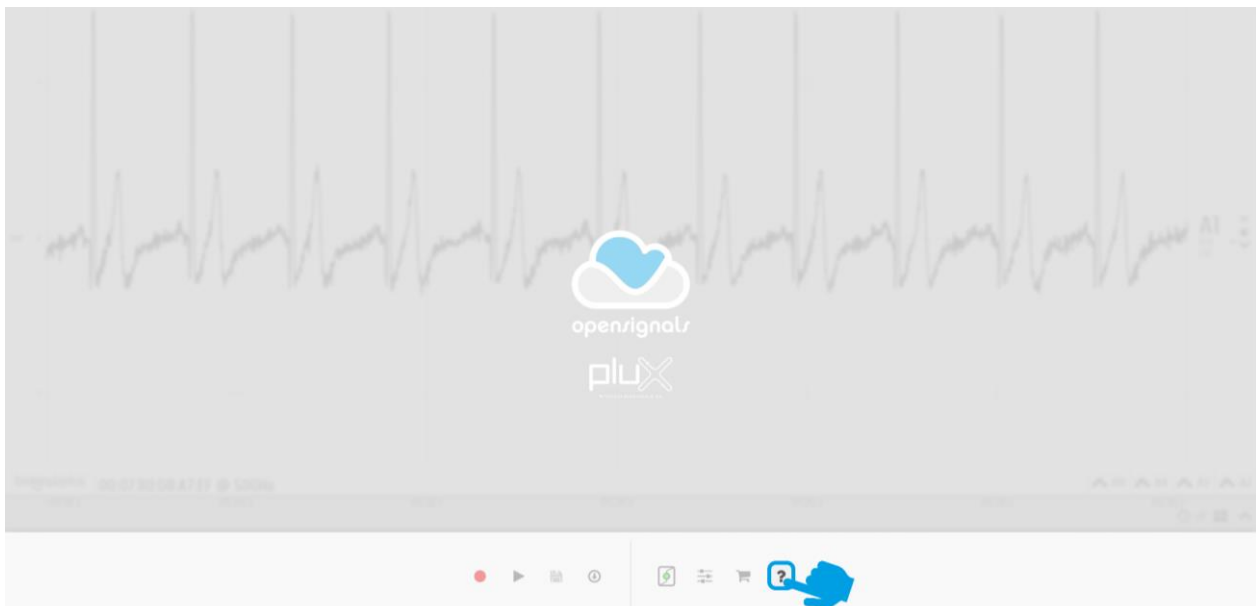
6. Activate “MyPlugin” on OpenSignals

6.1. Execute OpenSignals

Using **OpenSignals** the communication with PLUX signal acquisition devices is possible and now you can make **OpenSignals** even more powerful, taking into consideration the new plugins that you will create and eventually share with the web community.



6.2. Access the “Help” icon on the main page



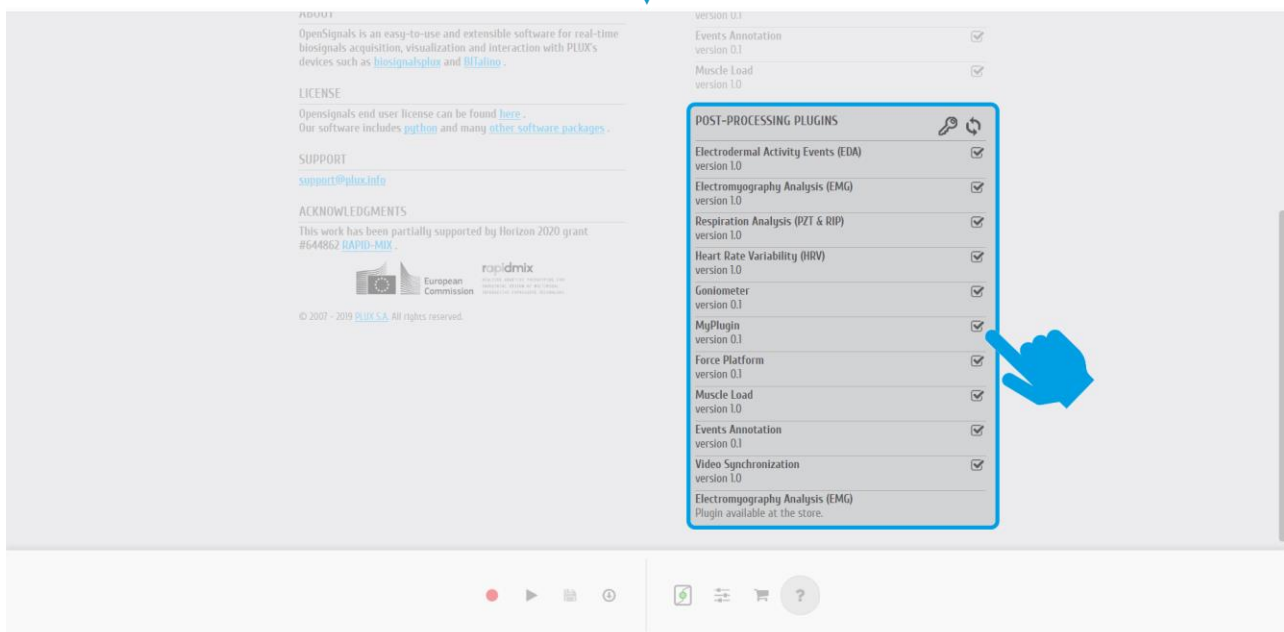
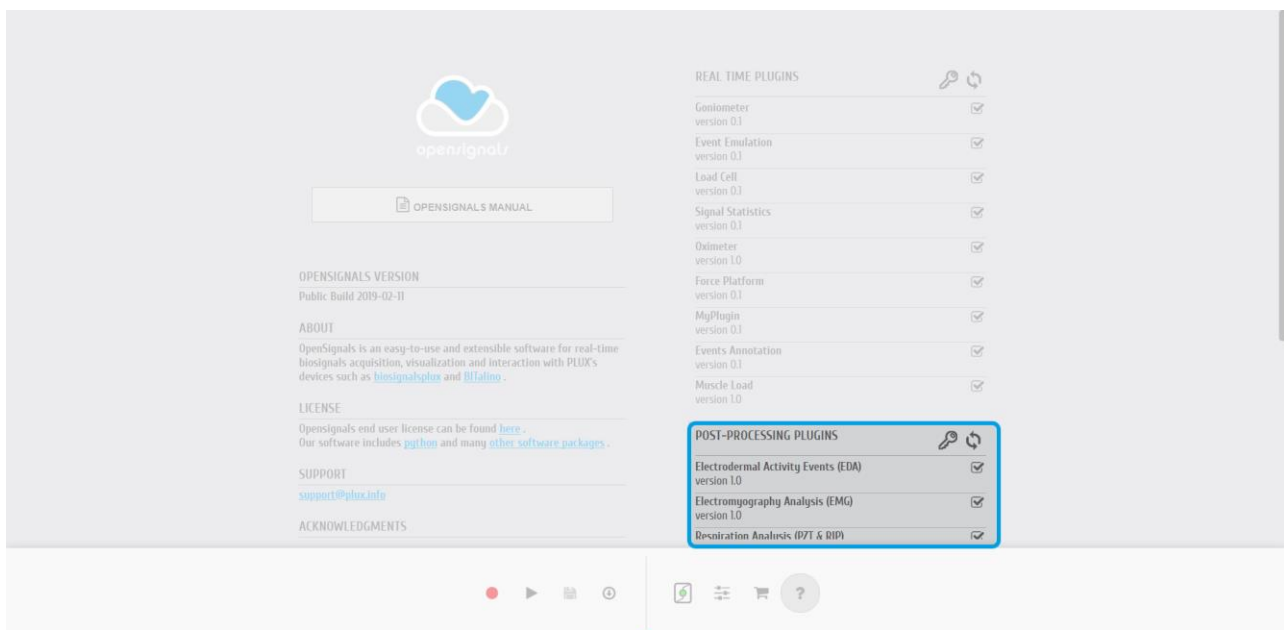
How to Create your Own OpenSignals Plugin

Quick-Guide

QG 02042019

6.3. Go to the “Post-Processing Plugins” section

Here it is “MyPlugin”, now you simply need to enable it checking the respective box (in case it isn't)!



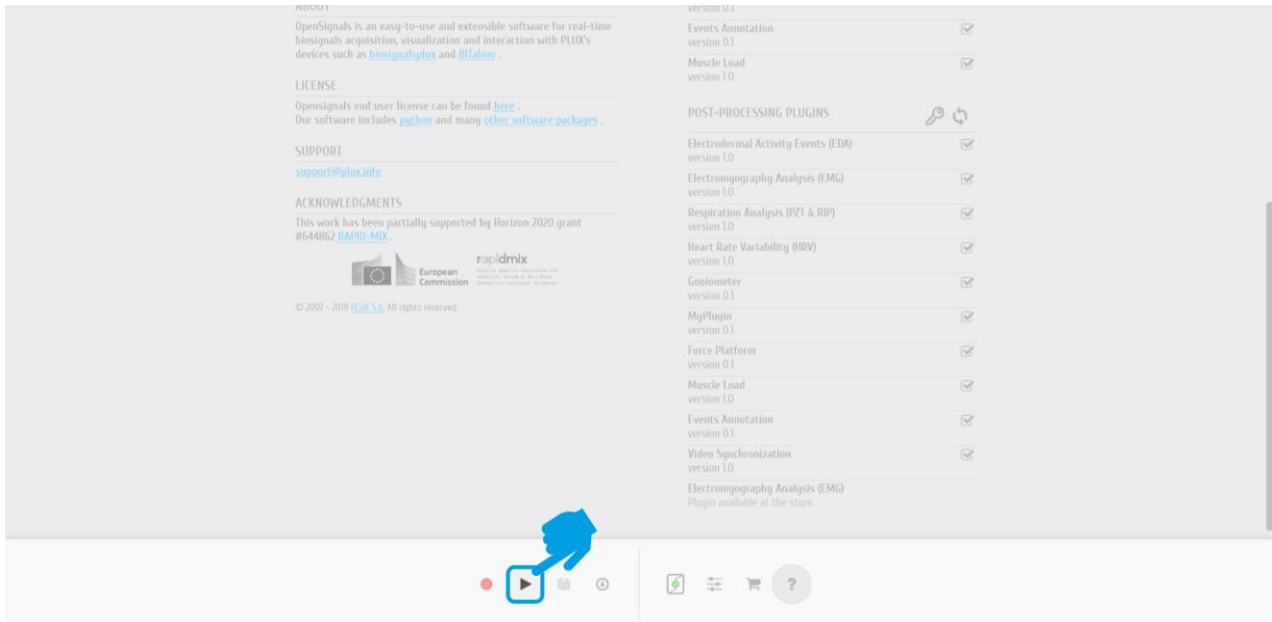
How to Create your Own OpenSignals Plugin

Quick-Guide

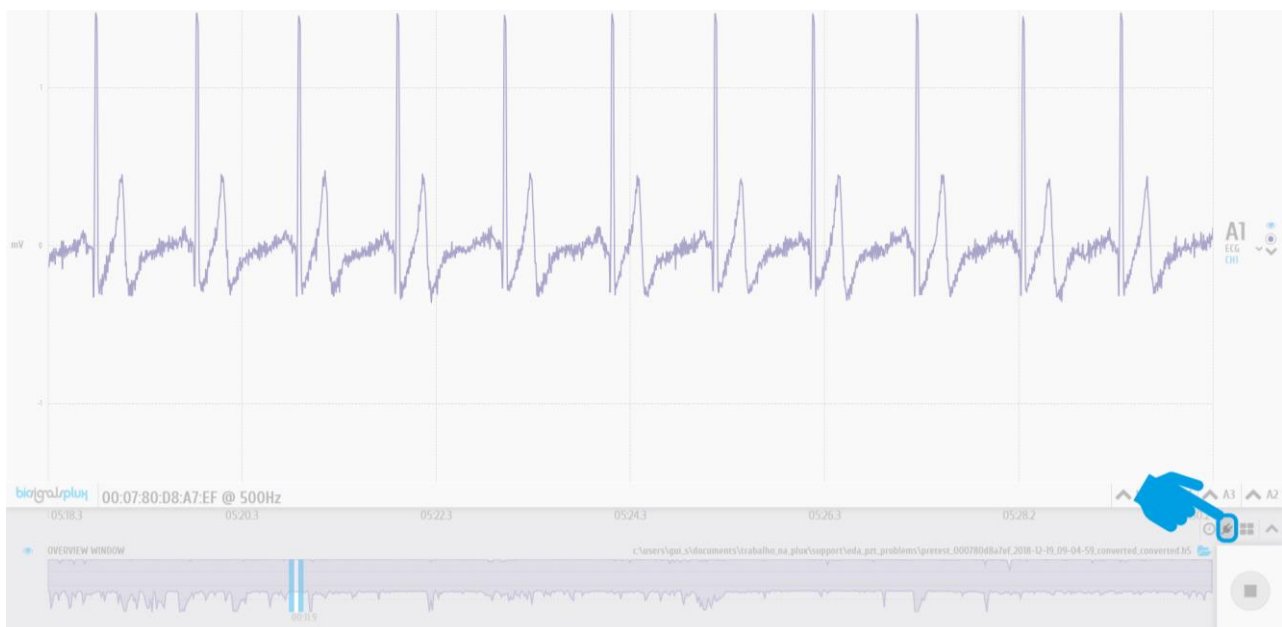
QG 02042019

6.4. Open “Visualisation” window

In this window you can visualise signals that you previously acquired. We also provide you some processing plugins... and now your plugin is also in this environment!



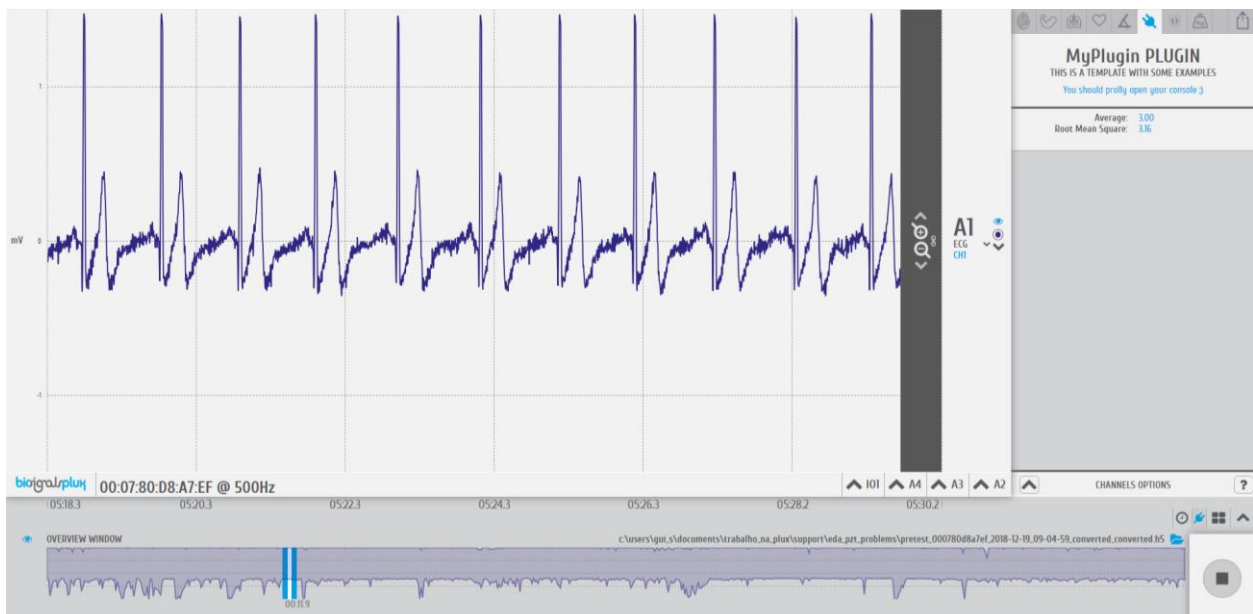
6.5. Show “Plugin Tab”, clicking on the bottom-right icon



6.6. Select “MyPlugin” on the new tab



Finally, we reached **MyPlugin** interface. For now, it only contains our **template**, but in the following steps we will go a little bit deeper... being creative with the inclusion of a few more elements!



On the following sections we will give you some hints to achieve one final purpose:

“Create a Plugin with three inputs, a button and a graphical zone”

The previous goal is a little bit generic but gives us some clues. Essentially, it will be created a Plugin where a time window is selected (using the two inputs for specifying the “begin” and “end” points). On the third input it will be specified the channel number to be analysed.

After pressing “Process” button the signal segment inside the selected time window will be plotted on a graphical area.

7. Insert user interface components on .html interface

Let's start inserting the three input boxes, where the beginning and end times, together with the channel under analysis, will be analysed. These elements can be easily inserted on our **.html** interface.

You just need to insert the following code (with small adjustments²) inside the `<div>` with class "myplugin_divider" (or "`<your_plugin_name>_divider`") keeping the code that is contained in our template (in spite of not being referred at this document it contains interesting functionalities that may be useful to you):

```
<div id="myplugin_LABEL" class="horizontal_half horizontal_half_second">
  <label for="myplugin_LABEL_input">LABEL (UNITS)</label>
  <input id="myplugin_LABEL_input" class="hover_edit" type="number"
    value="BEGIN|END" style="width: 90%;margin: 0px;" min="0">
</div>
```

This code must be inserted three times and adapted for each different input box, we need to replace "LABEL" by our desired identifier. For the three inputs the following labels were selected:

- **begin** | Identifying the input where the beginning of the processing window will be defined
- **end** | Identifying the input where the end of the processing window will be defined
- **channel** | Input for specification of the channel number under analysis

Additionally, the **UNITS** string should be replaced by the units of the quantity under analysis (for our application it will be seconds).

On the `<div>` tag, the `class` attribute is optional, however we will use "horizontal_half" and "horizontal_half_second" classes, which are global **OpenSignals** classes, ensuring that our `<div>` will use half of the available plugin interface width.

On `<label>` tag we can define the string that will be presented to the user, identifying the input being requested. The `for` attribute establishes a link with the respective `<input>`, so, the value of `for` needs to be the `id` of our `<input>` tag.

Finally, on the `<input>` tag some fields are especially relevant, namely `type` where we define that our input will be "numeric", `min`, used to define the minimum acceptable value and also the `style` attribute, ideal to establish some inline **css** styles.

For the current case we are simply changing the width and margin to be applied on the `<input>`.

The `value` field contains the predefined value that will be presented on the interface, we should replace **BEGIN|END** by the desired predefined value.

Now, we need to insert our "Process" button, using the following **.html** code (inside `<div>` with class "myplugin_divider"):

```
<div id="myplugin_LABEL_button_div" class="mL_divider mL_divider_small">
  <button id="myplugin_LABEL_button" class="plugin_button_process">
    LABEL
  </button>
</div>
```

Again, we should replace all **LABEL** entries on the previous code by the desired identifier of our button, for the current case our choice is "process".

OpenSignals internal "mL_divider" and "mL_divider_small" classes are used to centre the button both vertically and horizontally.

² The presented code is a template (which needs editing) for inserting one of the text boxes.

On the other hand, "plugin_button_process" class is a class that should be used for all "Processing" buttons on **OpenSignals** plugins, ensuring a coherence between all processing add-ons.

For accessing the complete definition of **OpenSignals** internal **css classes** you can explore the file "_OpenSignals.css" at "C:\Plux\OpenSignals (r)evolution\resources\app\static\css_OpenSignals.css"

The added code should have the following appearance:

Quick Note: The `<hr>` tag is used to insert a horizontal row on the graphical interface

```
<hr>
<div id="myplugin_begin" class="horizontal_half horizontal_half_second">
  <label for="myplugin_begin_input">Begin (s)</label>
  <input id="myplugin_begin_input" class="hover_edit" type="number"
    value="0" style="width: 90%;margin: 0px;" min="0">
</div>
<div id="myplugin_end" class="horizontal_half horizontal_half_second">
  <label for="myplugin_end_input">End (s)</label>
  <input id="myplugin_end_input" class="hover_edit" type="number"
    value="10" style="width: 90%;margin: 0px;" min="0">
</div>
<div id="myplugin_channel" class="horizontal_half horizontal_half_second">
  <label for="myplugin_channel_input">Channel Number</label>
  <input id="myplugin_channel_input" class="hover_edit" type="number"
    value="1" style="width: 90%;margin: 0px;" min="0">
</div>
<hr>
<div id="myplugin_process_button_div" class="mL_divider mL_divider_small">
  <button id="myplugin_process_button" class="plugin_button_process">
    PROCESS
  </button>
</div>
```

There is only one element left, i.e., our graphical zone where the signal of the channel/processing window under analysis will be plotted.

For now, we simply need to define the "graphical barebone" with a sequence of **<div>** tags with the following structure (for the graphical elements our **LABEL** is replaced by "window"):

```
<div id="myplugin_charts_LABEL">
  <div id="myplugin_charts_LABEL_border">
    <div id="myplugin_charts_LABEL_body"
      class="myplugin_chart_body myplugin_plugin">
    </div>
  </div>
</div>
```



At this stage, your code should be similar to the one presented on the following figure:

```
7 <!DOCTYPE HTML>
8 <html>
9 <head>
10 <title>MyPlugin Plugin</title>
11 <link rel="stylesheet" href="css/myplugin.css" type="text/css"/>
12 <script language="javascript" type="text/javascript" src="myplugin.js"></script>
13 </head>
14 <body>
15 <div class="plugin-master">
16 <div class="title">
17 <h1 id="myplugin_main_title"></h1>
18 <div id="myplugin_title" class="h1sub" align="center"></div>
19 <div id="myplugin_subtitle" class="h1subsub" align="center"></div>
20 </div>
21 <div class="myplugin_divider">
22 <div>
23 <div id="myplugin_avg_label" class="myplugin_label"></div>
24 <div id="myplugin_avg" class="myplugin_result"></div>
25 </div>
26 <div>
27 <div id="myplugin_rms_label" class="myplugin_label"></div>
28 <div id="myplugin_rms" class="myplugin_result"></div>
29 </div>
30 <hr>
31 <div id="myplugin_begin" class="horizontal_half horizontal_half_second">
32 <label for="myplugin_begin_input">Begin (s)</label>
33 <input id="myplugin_begin_input" class="hover_edit" type="number"
34 value="0" style="width: 90%;margin: 0px;" min="0">
35 </div>
36 <div id="myplugin_end" class="horizontal_half horizontal_half_second">
37 <label for="myplugin_end_input">End (s)</label>
38 <input id="myplugin_end_input" class="hover_edit" type="number"
39 value="10" style="width: 90%;margin: 0px;" min="0">
40 </div>
41 <div id="myplugin_channel">
42 <label for="myplugin_channel_input">Channel Number</label>
43 <input id="myplugin_channel_input" class="hover_edit" type="number"
44 value="1" style="width: 90%;margin: 0px;" min="0">
45 </div>
46 <hr>
47 <div id="myplugin_process_button_div" class="mL_divider mL_divider_small">
48 <button id="myplugin_process_button" class="plugin_button_process">
49 PROCESS
50 </button>
51 </div>
52 <hr>
53 <div id="myplugin_charts_window" class="remove">
54 <div id="myplugin_charts_window_border">
55 <div id="myplugin_charts_window_body" class="myplugin_chart_body myplugin_plugin">
56 </div>
57 </div>
58 </div>
59
60 </div>
61 </div>
62 </body>
63 </html>
```

We should define “myplugin_chart_body” and “myplugin_plugin” classes to ensure that our plot (based on **Dygraph** JavaScript library) will be correctly framed on our graphical user interface.

So, let's open “myplugin.css” file and add the following two classes and respective derived subclasses to it.

```
.myplugin_chart_body {
    width: 280px !important;
    height: 250px;
    box-sizing: border-box;
    position: relative;
}

.myplugin_plugin .dygraph-ylabel{
    font-size: 13px;
    -webkit-writing-mode: vertical-rl;
    width: 127px;
}

.myplugin_plugin .ylabel .dygraph-ylabel{
    margin-top: 10px;
}
```

```
.myplugin_plugin .dygraph-y2label {  
  -webkit-transform: scale(-1,-1);  
  font-size: 13px;  
  -webkit-writing-mode: vertical-rl;  
  margin-left: 40%;  
}  
.myplugin_plugin .y2label .dygraph-y2label {  
  margin-top: 10px;  
}
```

Note that the previous styles can be customised by you, the defined values ensured a good representation, but you can be creative!

The current appearance of our plugin will be similar to the one presented on the following figure:



As can be seen, we have an empty square on the plugin interface (below “PROCESS” button), which is reserved for dynamically plot our processing results using **JavaScript**.

The final user should not see this “empty” zone, so we need to hide it for now. It is very simple, just add “**remove**” to the class of the main div reserved to plot data, defined inside our **.html** file.

```
<div id="myplugin_charts_LABEL" class="remove">  
  <div id="myplugin_charts_LABEL_border">  
    <div id="myplugin_charts_LABEL_body"  
      class="myplugin_chart_body myplugin_plugin">  
    </div>  
  </div>  
</div>
```

Our graphical user interface is complete, but for now it is a static structure. To make it more fun we need dynamic behaviour which will be ensured by **JavaScript** component together with **Python**.

On the next topic it will be presented the basic procedure that should be followed to achieve the final purpose of the exercise, i.e., plotting of a specific time window from the acquired signal under analysis.

8. Program the “click” event of “PROCESS” button

The major changes on our `.js` file will be caused by the programming of “PROCESS” button “click” event.

JavaScript `myplugin_onReady()` function is invoked after **OpenSignals** successfully loads the `.html` file of our plugin, so, all our events should be inside this function.

For our add-on, JavaScript implementation is based on **jQuery**, you can use this library without restrictions, because it is included on **OpenSignals** project.

The reference to a “click” event respects the following structure:

```
$("#ELEMENT_ID").on("click", function(){  
    ... instructions to be executed after "click"...  
});
```

ELEMENT_ID needs to be replaced by the id of our “PROCESS” button, which is “myplugin_process_button”.

The internal operations/instructions to be executed should start by getting and storing the parameters specified on the `.html` interface and also the mac-address of the device under study.

Taking into consideration that we know the `.html` id of each interface element/component, we can easily do this task with the `val()` method:

```
$("#myplugin_process_button").on("click", function(){  
    var begin_time = $("#myplugin_begin_input").val();  
    var end_time = $("#myplugin_end_input").val();  
    var channel_number = $("#myplugin_channel_input").val();  
    var mac = $("input[name=  
        'plugin_general_device_selection']:checked").val();  
});
```

There is only one element remaining, i.e., the communication of this information to **Python** component. **Python** is a really useful programming language for processing tasks, taking into consideration its Open Source nature and the diversified set of packages available for the community.

For that, we can use one pre-implemented function on our **template** called `send_to_JavaScript`, making some small changes to meet the requirements of our application.

However, a more direct approach can be used: we only need to create a new function inside **MyPlugin** class declared inside the `.py` file of our project. This function can be called `process`, having the following instructions inside it.

```
def process(self, begin, end, channel, mac):  
    ... instructions to be executed ...
```

This function has four inputs, which are necessary for **Python** component in order to receive the values “begin_time”, “end_time”, “channel_number” and device “mac” address.

For ending our “click” event structure, we just need to call the previous `.py` function, passing the respective inputs from `.js` to `.py` file.

```
$("#myplugin_process_button").on("click", function(){  
    var begin_time = $("#myplugin_begin_input").val();  
    var end_time = $("#myplugin_end_input").val();  
    var channel_number = $("#myplugin_channel_input").val();  
    var mac = $("input[name=  
        'plugin_general_device_selection']:checked").val();  
  
    // Call "process" function  
    server.myplugin.process(parseFloat(begin_time),  
                            parseFloat(end_time),  
                            parseInt(channel_number), mac);  
});
```


The `parseFloat()` and `parseInt()` methods ensure that our interface values, interpreted as strings, will be converted to a numerical format.

As you can see, it is very simple to transmit data between files. Using the above green instruction an **OpenSignals** module called **server** is dealing with our communication request, so, essentially for communicating between **.js** and **.py** components it will only be necessary to respect the structure:

```
server.<plugin_name>.<py_function_name>(<arg_1>, <arg_2>...)
```

9. Definition of the processing instruction to be executed by “process” function

Now, let's focus our attention on the **process** function inside **.py** file!

With the given inputs, defined by the user in the interface, the first step to be followed is the data request, i.e., we need to get the signal samples on the channel defined by the user and inside the desired time window.

OpenSignals **fileReader** module contains lots of relevant functions, like for example **getSamplingFreq**, with which you can get the acquisition sampling rate.

Additionally, inside this module is defined **getRaw**, ideal for our goal of retrieving the acquired data from a specific channel and time window.

```
def process(self, begin, end, channel, mac):
    sampling_rate = self.fileReader.getSamplingFreq(mac)
```

On the previous instruction we get the acquisition sampling rate for the device identifiable by *mac* address selected on the user interface.

We will need this information for converting our time inputs into sample numbers, which are relevant to access data through indexation, as demonstrated below.

```
def process(self, begin, end, channel, mac):
    sampling_rate = self.fileReader.getSamplingFreq(mac)

    # Init and End sample.
    init_sample = int(begin * sampling_rate)
    end_sample = int(end * sampling_rate)
```

Finally, it will be possible to request the desired signal segment with **getRaw** method.

```
def process(self, begin, end, channel, mac):
    sampling_rate = self.fileReader.getSamplingFreq(mac)

    # Init and End sample.
    init_sample = int(begin * sampling_rate)
    end_sample = int(end * sampling_rate)

    # Get signal segment.
    data = self.fileReader.getRaw(channels=[channel],
                                  initSample=init_sample,
                                  finSample=end_sample, mac=mac)

    # Convert 'data' into a flatten list [[1], [2],...] -> [1, 2, ...]
    data = numpy.concatenate(data)
```

10. Generate a time-axis taking into consideration the selected number of samples

```
def process(self, begin, end, channel, mac):
    ...
    # Generate Time-Axis.
    time = numpy.linspace(begin, end, len(data))
```

11. Send the generated time-axis and segmented data to JavaScript

The remaining step in our journey lies on the graphical representation of the selected time window, however, data processed inside **.py** file needs to be sent to **.js** component to be presented on plugin interface.

A “receptor” function should be created on **.js** file, receiving two arguments, i.e., the **<div>** **id** where the graphical object will be generated (“myplugin_charts_LABEL_body”, or in our particular case **LABEL=“window”**) and a data array (containing a time-axis and the acquired data samples).

This “receptor” function should be placed outside **myplugin_onReady()** function where we previously defined our click event.

```
function createChart(element_id, data) {  
    return new Dygraph(document.getElementById(element_id),  
        createCSV(data, ['Time (s)', lang.myplugin.RAW]),  
    {  
        width: 290,  
        height: 250,  
        axisLabelFontSize: 12,  
        labelsDivWidth: 150,  
        labelsKMB: true,  
        series: {[lang.myplugin.RAW]: {axis: 'y'}},  
        xlabel: 'Time (s)',  
        ylabel: lang.myplugin.RAW  
    });  
};
```

Quick Note: The **labelKMB** parameter is applied in order to abbreviate axes values k→thousand, M→million...

Essentially, the previous function returns a **Dygraph**. All the specified contents and customization options can easily be changed by consulting the respective documentation³.

Note: Here we used a reference **lang.myplugin.RAW**. With this instruction the value linked with “RAW” key of the dictionary (inside **lang.myplugin.js** file) is returned. The returned value is “RAW Data Samples”.

After adding the “RAW” key to **lang.myplugin.js** the content of the file will be the following:

```
1 lang.myplugin = {  
2   NAME: "MyPlugin",  
3   MAIN_TITLE: "MyPlugin PLUGIN",  
4   TITLE: "This is a template with some examples",  
5   SUBTITLE: "You should prolly open your console ;)",  
6   AVG: "Average: ",  
7   RMS: "Root Mean Square: ",  
8   RAW: "RAW Data Samples"  
9 };
```

In the future if we want to change this string, it will only be necessary to change it on our **lang.myplugin.js** file instead of the three references on the previous code segment.

As demonstrated on the previous example, it is necessary to deliver our data in a CSV format, in order to be used by **Dygraph** library.



³ <http://dygraphs.com/>

So, an auxiliary function responsible for this task should be considered (like **createChart** the following auxiliary function should also be inserted outside **myplugin_onReady()**):

```
function createCSV(data, labels) {
    var data_csv = ""
    for(var lbl = 0; lbl < labels.length; lbl++) {
        data_csv += labels[lbl] + ","
    }
    data_csv = data_csv.substring(0, data_csv.length - 1) + "\n"
    for(var ind = 0; ind < data[0].length; ind++) {
        for(var d = 0; d < data.length; d++) {
            data_csv += data[d][ind] + ","
        }
        data_csv = data_csv.substring(0, data_csv.length - 1) + "\n"
    }
    return data_csv;
};
```

There is a missing step, because, if you remember, at the end of **section 7** we hide our graphical zone.

So, it will be necessary to "unhide" it again. You can do this with a single line on **createChart** function this problem can be easily solved:

```
function createChart(element_id, data) {
    // New instruction to 'unhide' the graphical zone/div.
    $("#myplugin_charts_window").removeClass("remove");
    return new Dygraph(document.getElementById(element_id),
        createCSV(data, ['Time (s)', lang.myplugin.RAW]),
        {
            width: 290,
            height: 250,
            axisLabelFontSize: 12,
            labelsDivWidth: 150,
            labelsKMB: true,
            series: {[lang.myplugin.RAW]: {axis: 'y'}},
            xlabel: 'Time (s)',
            ylabel: lang.myplugin.RAW
        });
};
```

The final step will consist on calling this function on **process** function defined inside our **.py** file:

```
def process(self, begin, end, channel, mac):
    ...
    # Send graphical data to JavaScript.
    self.javascript.createChart("myplugin_charts_window_body",
        [list(time), list(data)])
```

In **red** we highlighted a small but very relevant detail, all data sent to JavaScript should be in a JSON⁴ serializable format, so we convert our data structures to **Python** lists.

After a long and amazing journey through **OpenSignals** add-ons creation we reached our destination!

There are more elements to explore and, like previously declared, your creativity will be a very important element to ensure infinity possibilities.

Despite this guide is not exhaustive, we think that it contains the main principles and tools to guide you.

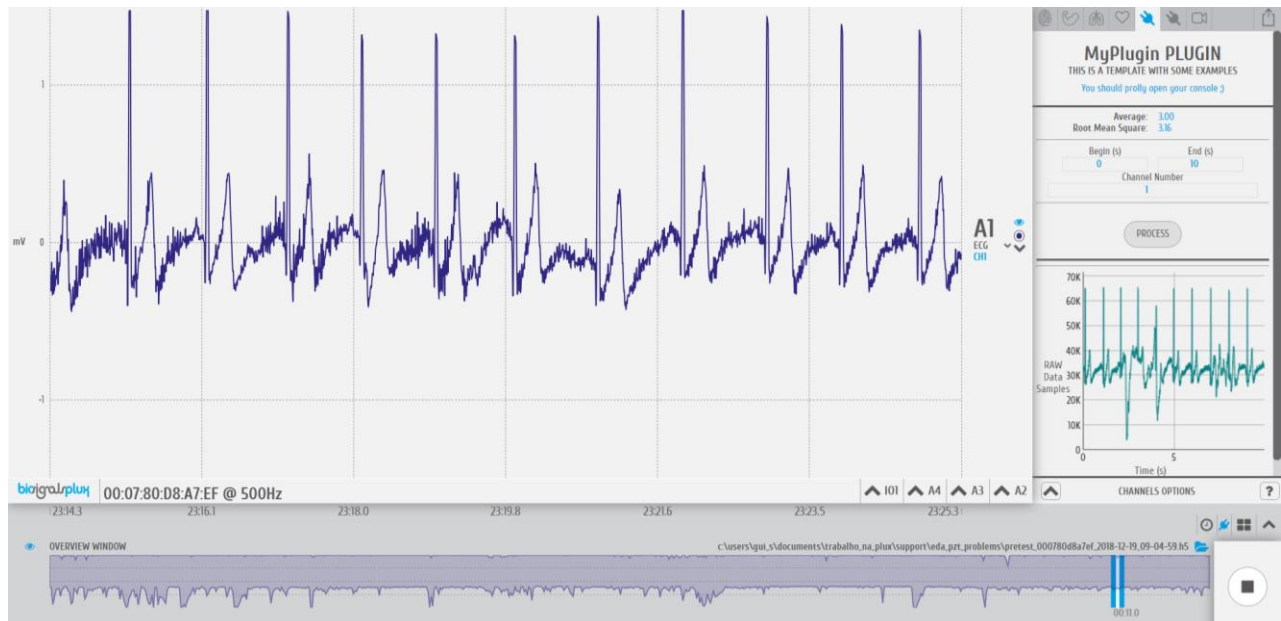
⁴ As defined by **W3Schools** "JSON is a syntax for storing and exchanging data... being also a text, written with JavaScript object notation"

How to Create your Own OpenSignals Plugin

Quick-Guide

QG 02042019

Let's take a look to the final result of our journey:



Extra

The previous implementation is mainly focused on offline processing, but the great majority of concepts can be extended to real-time processing.

First of all, if you want to create a **real-time** plugin, we should return to the .xml file and redo **step 3.6**.

For a real-time application you should keep the **<acquisition>** tag and remove **<visualization>** tag, however, if you want to create a multi-paradigm plugin (real-time and offline) you should keep the two tags.

After this step you will only need to write your code instructions inside **onData** function, a *callback* contained on our .py file.

The input arguments of **onData** are *mac* (containing the mac-address of the device under analysis), *nSeq* (a list with the number of sequence of the samples inside the received package of data), *digitalInputs* (digital values/flags), *data* (list with a package of raw data samples), *convertedData* (list with a package of converted data samples).

In terms of communication between Python and JavaScript, it is similar to the procedure described at the end of **section 8 (JavaScript→Python)** and **section 11 (Python→JavaScript)**.

Conclusion

In spite of the previous steps were not exhaustive they can be an excellent starting point of your journey on **OpenSignals**.

So, as a synthesis, for creating an **OpenSignals** plugin you need to design your interface on the .html file, program events, related with the interaction between the user and the graphical interface, on the .js file and finally expand your processing capabilities through **Python** packages applying new and interesting algorithms.

This guide is continuously being updated, so, if you have further questions to be solved, please do not hesitate to contact our support team: support@plux.info

We will be extremely happy to help and guide you through this amazing adventure!