**Manual for Developer**

**Daniel Melvin 28.07.2023**

**Email:** [**danielmelvin02@gmail.com**](mailto:danielmelvin02@gmail.com)

A screenshot of a computer

Description automatically generated

Table of Contents

[1. Webpage Structures: 3](#_Toc141445717)

[2. HTML Explanation, Improvement, and Problems 3](#_Toc141445718)

[2.1. Login.html and register.html 3](#_Toc141445719)

[2.1.1. Improvement: 3](#_Toc141445720)

[2.2. Index.html: 4](#_Toc141445721)

[2.2.1. The logic for update status: 4](#_Toc141445722)

[2.2.2. Plot on the Dashboard or index.html: 5](#_Toc141445723)

[2.2.3. Start Experiment: 5](#_Toc141445724)

[2.3. Logging for all pages: 7](#_Toc141445725)

[2.4. Laser.html, Caylar.html, RFSoC.html and Mercury.html: 7](#_Toc141445726)

[2.4.1. Construct\_object.py(construct object, construct toptica, construc mercury, construct caylar and construct the rfsoc): 8](#_Toc141445727)

[2.4.2. Forms.py: 8](#_Toc141445728)

[2.5. Manual.html: 8](#_Toc141445729)

[2.6. Plotresults.html: 8](#_Toc141445730)

[2.7. Update button: 8](#_Toc141445731)

[3. PYTHON CONTROLLER 9](#_Toc141445732)

[3.1. Toptica Laser: 9](#_Toc141445733)

[3.2. Mercury: 9](#_Toc141445734)

[3.3. Flow of ophyd: 9](#_Toc141445735)

[3.4. RFSoC: 9](#_Toc141445736)

[3.5. Caylar: 10](#_Toc141445737)

[4. Current Problem or issue: 10](#_Toc141445738)

[5. Future to do list 10](#_Toc141445739)

# Webpage Structures:

This webpage uses Django Argon as the template.

<https://demos.creative-tim.com/argon-dashboard-django/docs/getting-started/overview.html>

The structure has the default Django Argon structure.

All the controllers and the code for logic and backend can be found in the /staticfiles folder, for the HTML page is on the /apps/template, and usually most HTML is in the /apps/template/home folder, and last, the views and other Django python code for controlling the HTML and the commands on the /apps/home.

A diagram of a software flow

Description automatically generated

For direct control using bluesky and ophyd, please directly using the ophyd code at staticfiles and implement it with bluesky interfaces code in jupyter notebook.

# HTML Explanation, Improvement, and Problems

## Login.html and register.html

This HTMl controls the login page and register. For now, it is still a default login and registers. Users can make new accounts.

### Improvement:

* Register only can be made by admins.
* Change/forget password

## Index.html:

The index.html is the main page or home page; this page is controlled using the index, status, statusCaylar, statusLaser, statusMercury, statusRFSOC, update\_live\_plot and start\_experiment

### The logic for update status:

// Add event listener for the update button

  const updateButton = document.getElementById("update\_button");

  updateButton.addEventListener("click", function () {

    updateStatusColor(statusLaserElement);

    updateStatusColor(statusCaylarElement);

    updateStatusColor(statusMercuryElement);

    updateStatusColor(statusRFSoCElement);

    isDashboardUpdating = false;

    let scriptExecuted = sessionStorage.getItem("script\_executed");

    sessionStorage.setItem("script\_executed", false);

    updateStatusLaser(); // Update the status immediately when the page loads

  });

  // Start the script once the HTML has finished loading

  document.addEventListener("DOMContentLoaded", function () {

    // Check if the script has already been executed

    let scriptExecuted = sessionStorage.getItem("script\_executed");

    let folder\_name = sessionStorage.getItem("folder\_name");

    let experiment\_name = sessionStorage.getItem("experiment\_name");

    let description = sessionStorage.getItem("description");

    let selected\_devices = sessionStorage.getItem("selected\_devices");

    if (!scriptExecuted || scriptExecuted == "false") {

      updateStatusColor(statusLaserElement);

      updateStatusColor(statusCaylarElement);

      updateStatusColor(statusMercuryElement);

      updateStatusColor(statusRFSoCElement);

      updateStatusLaser(); // Update the status immediately when the page loads

    } else {

      updateStatusColor(statusLaserElement);

      updateStatusColor(statusCaylarElement);

      updateStatusColor(statusMercuryElement);

      updateStatusColor(statusRFSoCElement);

      updateStatus();

    }

  });

If the page is the first time rendered, the page will ask the status for each device one by one, it will try to make a new connection with the device as soon as the status page is called. Then, if the page is already reloaded once before, the page will use the previous connection.

There is also an “update status” button. This button will refresh all the connections by building new connections. So if the page is encountered, the error update status button will solve most of the issues

### Plot on the Dashboard or index.html:

So there are three plots here for the QC project, first toptica laser, mercury itc and the caylar magnet. All of these are controlled from the update\_live\_plot from the views.py.  
  
*Remarks:*

The plot will receive the datapoint from the CSV logging file once after the page of the dashboard is rendered, but after it, only the new datapoint will be pass into the plot.

### Start Experiment:

The start experiment will make a new configuration file containing all the configuration parameters chosen from the start experiment form.

Store all of the experiment information, such as name and description.

**Important!**

* If RFSoC is chosen, the pulse will be sent to the RFSoC for EOM and AOM.
* If the selected devices is off then it will not be run because it will check the device, if it is on or off before start experiment.

*Remarks:*

If you encounter a problem here, please look to the index.html and check id:experimentForms and the javascript below  
$("#experimentForms").on("submit", function (event) {

      event.preventDefault(); // Prevent the default form submission

      // Serialize the form data

      $("#messageLoadingContainer").html('<div class="alert alert-info" role="alert">Loading...</div>');

      // Make an AJAX POST request only if the form is valid

      if (this.checkValidity() && validateDeviceListForm()) {

        // Retrieve the CSRF token from the HTML page

        // Enable all input elements inside the container after click

        $(".container-fluid input").prop("disabled", true);

        $(".container-fluid textarea").prop("disabled", true);

        const csrfToken = $("input[name='csrfmiddlewaretoken']").val();

        // Include the CSRF token in the AJAX request headers

        $.ajaxSetup({

          headers: {

            "X-CSRFToken": csrfToken,

          },

        });

        // let formData = $(this).serialize();

        const selectedDevices = [];

        $("input[name='selected\_devices[]']:checked").each(function () {

          selectedDevices.push($(this).val());

        });

        sessionStorage.setItem("selected\_devices", selectedDevices);

        $.ajax({

          url: "{% url 'start\_experiment' %}",

          type: "POST",

          data: {

            file\_name: document.getElementById("id\_file\_Name").value,

            experiment\_name: document.getElementById("id\_experiment\_name").value,

            description: document.getElementById("id\_experiment\_description").value,

            selected\_devices: selectedDevices,

            startLogging: false,

          },

          success: function (response) {

            // Handle the response if needed

            isDashboardUpdating = false;

            $("#messageLoadingContainer .alert").remove();

            // Update the message dynamically from the response

            let messageContainer = $("#messageContainer");

            messageContainer.html('<div class="alert alert-success alert-dismissible fade show" role="alert">' + response.message + '<button type="button" class="close" data-dismiss="alert" aria-label="Close">' + '<span aria-hidden="true">&times;</span>' + "</button>" + "</div>");

            // Reset the form fields

            $("#experimentForms")[0].reset();

            // $("#experimentForms").addClass("hidden");

            isUpdating = true;

            isLogging = false;

            isDashboardUpdating = false;

            file\_name\_save = response.file\_name;

            sessionStorage.setItem("folder\_name", response.file\_name);

            // $("#buttonExperiment").removeClass("hidden");

            DashboardPlot(freqUpdate, false, selectedDevices);

          },

          error: function (xhr, status, error) {

            // Handle any error that occurs during the request

            $(".container-fluid input").prop("disabled", false);

            $(".container-fluid textarea").prop("disabled", false);

            // Update the message dynamically

            let messageContainer = $("#messageContainer");

            messageContainer.html('<div class="alert alert-danger alert-dismissible fade show" role="alert">' + xhr.responseJSON.message + '<button type="button" class="close" data-dismiss="alert" aria-label="Close">' + '<span aria-hidden="true">&times;</span>' + "</button>" + "</div>");

          },

          complete: function () {

            // Remove the loading indicator

            $("#messageLoadingContainer .alert").remove();

          },

        });

      } else {

        let messageContainer = $("#messageContainer");

        messageContainer.html('<div class="alert alert-danger alert-dismissible fade show" role="alert">' + "Please fill the name of experiment and the file path" + '<button type="button" class="close" data-dismiss="alert" aria-label="Close">' + '<span aria-hidden="true">&times;</span>' + "</button>" + "</div>");

      }

    });

  });

All of this experiment form is managed by the start\_experiment function at views.py.

## Logging for all pages:

The logging function should happen on every pages following this script which calling the update\_live\_plot function from views.py

function LoggingFile(interval) {

    const csrfToken = $("input[name='csrfmiddlewaretoken']").val();

    // Include the CSRF token in the AJAX request headers

    $.ajaxSetup({

      headers: {

        "X-CSRFToken": csrfToken,

      },

    });

    $.ajax({

      url: "{% url 'plot' %}", // URL mapped to the status view

      type: "POST",

      data: { changePage: false },

      dataType: "json",

      success: function (response) {},

      error: function (xhr, status, error) {

        console.log(xhr.responseText);

      },

      complete: function () {

        // Call the updatePlot function again after the specified interval

        setTimeout(function () {

          LoggingFile(interval);

        }, interval);

      },

    });

  }

  LoggingFile(1000);

Future Implementation:  
it will be great if the logging file is executed on the base html or in the parent html which the logging script only be placed at one of the html, not all html.

## Laser.html, Caylar.html, RFSoC.html and Mercury.html:

All of these pages are used to control each device's parameter. Every time the page is rendered, it will create a new connection to ensure the device is on before setting the configurations.  
  
There are actual value reading sections below the configuration setting to make sure the current number, but it will only update once the page is reloaded.

This HTML will call their own page views python code from views.py, which laser\_page\_view, mercury\_page\_view, caylar\_page\_view and rfsoc\_page\_view.

The objects of all of the device to build the connection in the future will need construct\_object and all the forms will be manage at forms.py

### Construct\_object.py(construct object, construct toptica, construc mercury, construct caylar and construct the rfsoc):

So this code is used to build a new object for each device and needs to do (.try\_connect() function to build a new connection before reading and setting the value to the devices)

These methods are crucial in order to able to control all of the devices.

### Forms.py:

All of the forms are modelled using the Django form, not using the model therefore, every time you want to edit or update forms, please check the forms.py and find the corresponding form you will use.

## Manual.html:

This is used to put any information regarding the device reading, the webpage troubleshooting or any kind of information that will be used.

## Plotresults.html:

Still developing. But it will be used for plotting the DAQ photon detection reading; they want to make it continuously ready the .txt file, it should be possible to be done, but because of the time, I can't finish it. Currently, the user only can upload the text once and plot once.

## Update button:

Every controller page, usually has three different pages, such as, update ip, update config and update all.

1. Update IP – the web will send a request to change the XML file and update only IP and port or any device information or identification and try to reconnect with the new identification
2. Update config – the web will send a request to change the configuration XML file and if connected to the device, it will try to change the device parameter directly to the system
3. Update All – the web will send a request to change the XML file for all of the parameters inside the form, first it will connect with the new device and try to change all the device parameters.

*REMARKS!!!*  
I have put some documentation of the code itself for each method and function, please take a look at it directly from the code. If you find difficulties, you can ask ChatGPT or any AI model because the code logic is built systematically, so it should be easy to be understand.

# PYTHON CONTROLLER

Most of the Python controllers can be found in the \staticfiles folder

**Notes:**  
Everytime you want to make new devices set and get function, please go to the controller class first, then add a new signal class and last, at the main class, you put all the signal classes to the component for each get and set function. To be frank, you can implement all of these without ophyd, but as a requirement, we should implement everything in ophyd object. So, another department can understand how our controlling devices work.

## Toptica Laser:

The laser only needs to use toptica\_ophyd, which will make a new ophyd object for laser purposes only and the laser object for you to control the laser. Laser also has their own python package toptica\_lasersdk

## Mercury:

Mercury has their own driver mercuryitc driver. But we implement the driver to be compatible with the ophyd objects to control the devices

## Flow of ophyd:

User set and get function -> ophyd set and get methods -> Toptica\_lasersdk or mercuric driver setting and getting -> send the signal to the devices

## RFSoC:

RFSoC is quite complicated to be controlled because we need SSH to build connection with the devices. First, we need RFSoC\_controller.py, in order to build the RFSoC object we used object from the RFSoC controller.

The connection is controller using the SSH.py, also in the SSH.py transferring file and the run code can be found at SSH.py.

The pulse is controlled using the QICK package can be found online that runs inside the RFSoC to control the EOM and AOM. Thus, when we want to initiate and make pulse, we must run the RFSoC.py that can be found at /qick/qick\_demos/ssh\_control. All of the important files to run the code can be found at /qick/qick\_demos/ssh\_control.

Sequence.py is really important because it has a simple algorithm to break the TTL sequence with the correct sequence in a way AOM can understand it. So, if we have pins 0,1,2,3 time start sequence [[1,2,3,4],[2,3],[1,4],[2,4]] and length sequence [[0.5,0.5,0.5,0.5],[0.5,0.5],[0.5,0.5],[0.5,0.5]] then first it will find the pins that start the same start time, and then it will check how long it will be turned on. If, during the turning on, other pins need to be turned or off, then the current on time should abort and make new command to make the pins turned on again with a new one.

In order to build a configuration file or config.xml for RFSoC, all of the configuration information will be sent to the RFSoC folder in the form of the config\_file.py. This config\_file.py will be sent to the RFSoC through SSH, then executed, and it will generate the xilinx.xml for the configuration file.

Future improvements:

Now, in order to wait for the code is run successfully from the RFSoC I am using the time sleep. But we can implement the shell.recv\_ready() function will wait until we receive the message, but please note that it will only wait for one message from the response. Therefore, we need enough recv\_ready functions before executing the next line of the code. I once implemented the rech\_ready, but I got a problem because I did not know how many times or how long it needed to wait for the recv\_ready()

## Caylar:

Caylar is controlled using sockets directly to the devices. Thus it needs sockets.py and controller.py to make all the sockets connection can be compatible with the ophyd objects.

**Notes:**  
All of the needed manuals I put on the /staticfiles/manual.

# Current Problem or issue:

* Laser async problem, sometimes when the system gets multiple information from the devices, the previous information still loading, but the new command for get information has already executed the previous data comes into the new commands with different data types
* Continuous Result Plot DAQ
* Plot maybe can be improved if we get a lot of datapoints
* Connection to ITC doesn’t close when the server closes (ITC ethernet enable needs resetting)

# Future to do list

* Experiment files saved through the QC Webpage
* Add test case