Manual for Developer

Daniel Melvin 28.07.2023

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

Denis Hnidenko 14.08.2023 (updated)

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

GitHub link to project: [qc\_website](https://github.com/DenisSKK/qc_website-main) (email me if you want access to edit this repository, or fork it by yourself)

Obrázok, na ktorom je text, snímka obrazovky, softvér, počítačová ikona

Automaticky generovaný popis

Table of Contents

[1. Webpage Structures: 3](#_Toc142984818)

[2. HTML Explanation, Improvement, and Problems 3](#_Toc142984819)

[2.1. Login.html and register.html 3](#_Toc142984820)

[2.1.1. Improvement: 4](#_Toc142984821)

[2.2. Index.html: 4](#_Toc142984822)

[2.2.1. The logic for update status: 4](#_Toc142984823)

[2.2.2. Plot on the Dashboard or index.html: 5](#_Toc142984824)

[2.2.3. Start Experiment: 5](#_Toc142984825)

[2.3. Logging for all pages: 9](#_Toc142984826)

[2.4. Laser.html, Caylar.html, RFSoC.html, Mercury.html, sx199, and sr830: 10](#_Toc142984827)

[2.4.1. Construct\_object.py(construct object, construct toptica, construct mercury, construct caylar and construct the rfsoc): 10](#_Toc142984828)

[2.4.2. Forms.py: 10](#_Toc142984829)

[2.5. Manual.html: 10](#_Toc142984830)

[2.6. Plotresults.html: 10](#_Toc142984831)

[2.7. Update button: 11](#_Toc142984832)

[3. PYTHON CONTROLLER 11](#_Toc142984833)

[3.1. Toptica Laser: 11](#_Toc142984834)

[3.2. Mercury: 11](#_Toc142984835)

[3.3. Flow of ophyd: 11](#_Toc142984836)

[3.4. RFSoC: 12](#_Toc142984837)

[3.5. Caylar: 12](#_Toc142984838)

[3.6. SX199 controlling up to two CS580: 12](#_Toc142984839)

[3.7. SR830: 13](#_Toc142984840)

[4. Current Problem or issue: 13](#_Toc142984841)

[5. Advice for future developer(s): 15](#_Toc142984842)

[Possible implementation of experiment page 15](#_Toc142984843)

[How to add new device with instrbuilder 16](#_Toc142984844)

[USB over IP 17](#_Toc142984845)

[For group of developers 17](#_Toc142984846)

[6. Future to do list: 17](#_Toc142984847)

# 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 are 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 page 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, Mercury.html, sx199, and sr830:

All 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, rfsoc\_page\_view, sx\_page\_view, and sr\_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, construct 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 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 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:**  
Every time 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.

## SX199 controlling up to two CS580:

To control CS580 or two CS580 you need to connect to the SX199 optical interface controller. CS580 is connected to one of 4 ports. When you want to control CS580 connected to Port 1, you need to link SX199 to Port 1 and then every command sent to SX199 is forwarded to Port 1 (Until SX receives termination character. Default is “!”.)

My implementation is for Ports 1 and 2. You can control first, second, or both.

SX199 is connected via network. To change type of connection, edit of update\_configs.py is needed.

Implemented similar way as MercuryITC.

Future improvements:

Keep alive connection could be improved.

There is a delay of 0.1s after linking to port, before executing CS commands. It is unknown to me why it doesn’t work without that.

## SR830:

SR830 is connected via AnywhereUSB by DIGI. Because of that it acts as a device connected via USB. That depends only on the adapter. This specific instrument is connected via GPIB to USB adapter.

To change type of connection, edit of update\_configs.py is needed.

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

# Config.yaml

Config file for editing instruments address/IP address is located at qc\_website/staticfiles.

This file is supposed to be edited before starting a website to take effect.

# 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)
* Website is made with Django. It contains multiple pages (Dashboard and one for every implemented instrument). This approach makes things difficult. After loading server for the first time, instruments are connected one by one in a sequence (Only on Dashboard). In this process you cannot reload/move pages (it will mess up connecting and start of the logging instruments). Back-end doesn't work independently to the front-end. This leaves room for mistakes.
* These connections are stored in global variable, so the logging could continue if you switch to instrument's page. Then, every time you move to another page, different JavaScript is used to continue logging every 'updateFrequency' (This is the only way to keep callback for logging function alive).
* Django has functions in views.py that connect to the instruments, keep connection alive, call getters and setters for instruments. This could work without a problem.
* Maybe would be better to make one static page where the dashboard is, which would be interactive. This would eliminate repeated startup of JavaScript (which keeps logging alive).

# Advice for future developer(s):

• Study already implemented code components (Ophyd drivers for controlling instruments are very useful and reusable to make new instruments for the controller)

• First, all the instruments should be controllable via network, only after that it would be good to proceed with controlling multiple instruments at once (as a part of the experiment control)

• Then think carefully about how to proceed with experiment part of the controller

• After that, make front-end part work with the back end. It would be best, if they worked more independently, then they work now. Now, Django server acts as front-end with back-end as well. (Back-end should work on interactions between devices. Front-end should just ask back-end for information and send commands on what to do. Right now, it's kind of messy)

## Possible implementation of experiment page

* One page design
* Connect all devices and collect all getters and setters available from each connected device.
* Use collected getters and setters and make modular experiment sequencer. Probably something that can add as many getters to read devices, and many setters with which it should iterate.
* Kresby rukou
  main.py
  Initialize connection between back-end and devices (may be implemented as adding them to some devices.py???).
  Call log() from logging.py every interval (interval should be settable from front-end)
  Also waiting for GET and POST requests from Front-End is necessary (calling appropriate functions when receiving GET and POST).
  Call create_experiment() function from experiments.py when received information about it from the Front-End (as mentioned, can be implemented as some xml/yaml/json file with all components and instructions for it)
  Kresby rukou
  logging.py
  Should read every READ function from every connected device.
  Write all read values into logging folder.
  Asynch/Multithreading for logging all connected devices into logging file (Already implemented function update_live_plot() in qc_website-main/apps/home/views.py could work just fine.)
  Kresby rukou
  experiment.py
  Should read every READ function from every connected device (from devices.py).
  Write all read values into logging folder.
  Asynch/Multithreading for logging all connected devices into logging file (Already implemented function update_live_plot() in qc_website-main/apps/home/views.py could work just fine.)
  !!! If we want to use Bluesky !!!
  This experiment.py should be only used to wrap Bluesky already existing functions. This way logging.py would not be really needed as Bluesky is in role for logging data and reading/writing devices.
  This approach might not be straight forward. Depending on what kind of experiments are needed, custom combination of Bluesky functions might need to be developed (or some factory function that will create sequence of Bluesky functions).
  Kresby rukou
  devices.py
  Can be done from config file where all addresses are supposed to be.
  Connect to each device from config file.
  Save them to a list of "connected" and "not_connected".
  Add functions to reconnect all/not connected devices so the user can try again.
  Get device object from list by name.
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  Kresby rukou
  device.py
  Some abstract class that will have information about device (name, address, available getters and setters).
  Basically wrapper function for ophyd wrappers (ophyd wrappers are probably needed because of the instruments like SX199 that communicates with up to 4 devices like CS580. Without this ophyd wrapper it would be hard to do that. You need to communicate with SX199, link to port 1-4 and then communicate with CS580, after done with that CS580 it is wise to unlink if other CS580 will be in use later).
  Kresby rukou
  Kresby rukou
  ophyd_device
  This should be group of files like:
  SX199_ophyd.py
  SR830_ophyd.py
  ITC_ophyd.py
  etc…
  Best to wrap functions that are going to be used and also make a list of them that device.py can use to know what functions are available.
  Kresby rukou
  Component and modular design is very much preferred.
* Kresby rukou
  FRONT-END
  Communicates with back-end.
  Asking for reconnecting to not available devices.
  Sends requests for data.
  Sends commands about experiment:
  New experiment
  Begin experiment
  Pause experiment
  Abort experiment
  Best to use something that won't reload (maybe React???)
  If there will be used Django again, make it all on one page, reloading pages is not the way to go from what I see.
  Kresby rukou
  BACK-END
  When started, attempt to connect to every device that is in config file with valid address to connect to.
  Acquiring getters and setters from connected devices.
  Logging all connected devices.
  Waiting for front-end's GET and POST requests.
  Running experiments (while also listening to GET and POST requests from front-end.
  Possible to make some form with checkboxes or dropdown menus, that will create xml/yaml/json file from that with info about experiment (what getters, what setters with values included

## How to add new device with instrbuilder

Instrbuilder is one way, to make an Ophyd object to control the device. If the device has some Ophyd driver library, you should use that. If it has a python library to control the instrument, you can use that instead of Instrbuilder, and make an Ophyd wrapper for that (ITC\_ophyd.py does just that [coded by Melvin Daniel, extended functions by Denis Hnidenko]).

Process of adding a new instrument:

• Installing Instrbuilder (follow the documentation's installation guide, don't forget it requires forked Ophyd package that is different version than latest one).

• Make a new CSV file for commands that instrument has (you can look at already made examples by Instrbuilder, or SX199 that doubles as CS580 made by me, or SR830).

• Create instrument class in instrbuilder\instruments.py (some instruments are as simple as creating class with \_\_init\_\_() function. But I have encountered a problem with SX199 termination characters. You can inspect the code it is quite straight forward).

• Add the instrument into the .instrbuilder\config.yaml file. The file is used for defining instrument name, python class in instruments.py, path to folder with commands.csv, and address (address will probably be "pyvisa: TCPIP::instrumentIP::instrumentPort::SOCKET". But USB, GPIB, Serial connection is also supported with instrbuilder).

○ Important! I have moved the .instrbuilder folder to Django website working directory inside staticfiles folder. Default is in ~/.instrbuilder/config.yaml. This would be a problem when controller website is installed and ran on different computers. More about how I changed Instrbuilder code to move .instrbuilder folder can be found on my GitHub commit here.

• Then test if it works.

○ One way is to follow Tutorial part in documentation where they use function open\_by\_name()

○ Other way (which also involves open\_by\_name() ) is to follow Interfacing an Instrbuilder Instrument to Bluesky part of documentation. There they also create an Ophyd object with generate\_ophyd\_obj() function from Ophyd library.

Then, I followed the previous work from Melvin Daniel and made a wrapper for that in file SX199\_ophyd.py. There you can find most of functions that are used in Django interface. Some are very simple, some are more complex because of the way SX must link to port before controlling current source CS580 and unlinking after it is done.

## USB over IP

For future server to host an instrument control website USB over IP will be a big part of the final solution. Unfortunately, the 3rd party software is needed. Software AnywhereUSB manager by DIGI can be installed as a stand-alone version or service. For server I think service is the way to go. This is not confirmed and needs to be looked at more. For more information about this look for section “Step 2: Determine how to run AnywhereUSB Manager: Service or stand-alone” in [DIGI AnywhereUSB manual](https://www.digi.com/resources/documentation/digidocs/PDFs/90002383.pdf).

Installation of AnywhereUSB manager:

* Install software from the [website](https://hub.digi.com/support/products/infrastructure-management/digi-anywhereusb-8-plus/?path=/support/asset-collection/anywhere-usb-plus-ez-os-specific-drivers/)
* Make a note of Client ID during installation (it can be anything)
* After installation on Manager panel right-click on found hub and “Open Web UI”. There you need to log in with credentials (name: admin, password: SLS\_user1). After that you need to add Client PC with his Client ID. Search for “Step 7: Verify initial connection” in manual for detailed step by step instructions.

## Tips for group of developers

If there will be team of at least 2 developers in the future, use GitHub.

Plan together and ahead. Ask for some experienced programmer for help (my independent work didn’t really work out well).

# Future to do list:

* Experiment files saved through the QC Webpage
* Add test case.
* Remake website to be a one-page design to get rid of reloading. (Can be also remade with something different than Django)
* Connect only to chosen devices to avoid conflict between multiple experiments/users (made with an config file/website choice options)
* Experiment sequencing and logging using Bluesky commands (plans, count, scan). Back-end wraps the user input into Bluesky commands.
* Auto launch AnywhereUSB, OPUS, etc.
* RS232 adapter to USB (get that to work)

# Useful link/documentations

[Bluesky doc](https://blueskyproject.io/bluesky/index.html)

[Instrbuilder doc](https://lucask07.github.io/instrbuilder/build/html/)

[Ophyd doc](https://nsls-ii.github.io/ophyd/index.html)

[Argon Dashboard Django website template](https://demos.creative-tim.com/argon-dashboard-django/docs/getting-started/getting-started-django.html)

[Daniel Melvin qc\_website GitHub (first version of this website)](https://github.com/Melvinwan/qc_website)

[Denis Hnidenko qc\_website GitHub (this version)](https://github.com/DenisSKK/qc_website-main)