**Documentation:**

**By: Kajal Biju**

**Email: kajalkbiju25@gamil.com**

**Github (Link to the project):** [**kajalbiju/PSI\_bluesky\_backend**](https://github.com/kajalbiju/PSI_bluesky_backend)

**Bluesky and Ophyd**

**Aim: Maintain a website to control and run experiments with multiple instruments present in the Lab.**

### **Step-by-Step Guide to install and run the project:**

**1. Clone the GitHub Repository**

**First, clone your GitHub repository to your local machine.**

**Copy code**

**git clone https://github.com/kajalbiju/PSI\_bluesky\_backend.git**

**cd PSI\_bluesky\_backend**

#### **2. Set Up a Python Virtual Environment**

**It's a good practice to use a virtual environment to manage dependencies.**

**sh**

**Copy code**

**# Install virtualenv if you haven't already**

**pip install virtualenv**

**# Create a virtual environment**

**virtualenv venv**

**# Activate the virtual environment On Windows**

**venv\Scripts\activate**

**# On macOS/Linux**

**source venv/bin/activate**

#### **3. Install Required Python Packages**

**Next, install the required packages listed in your requirements.txt file. Ensure you have all necessary packages including pyvisa, ophyd, and bluesky.**

**Copy code**

**pip install -r requirements.txt**

**If you don't have a requirements.txt file, you can create one with the following content:**

**pyvisa**

**ophyd**

**bluesky**

**databroker**

**pandas**

**Then run:**

**Copy code**

**pip install -r requirements.txt**

#### **4. Install Visual Studio Code (VS Code)**

**Follow the instructions for your operating system to install VS Code.**

* **Windows:** [**Install VS Code on Windows**](https://code.visualstudio.com/docs/setup/windows)
* **macOS:** [**Install VS Code on macOS**](https://code.visualstudio.com/docs/setup/mac)
* **Linux:** [**Install VS Code on Linux**](https://code.visualstudio.com/docs/setup/linux)

#### **5. Open the Project in VS Code**

1. **Open VS Code.**
2. **Open the PSI\_bluesky\_backend directory.**

#### **6. Configure Python Interpreter**

**Make sure VS Code is using the correct Python interpreter.**

1. **Open the command palette (Ctrl+Shift+P or Cmd+Shift+P).**
2. **Type Python: Select Interpreter.**
3. **Choose the interpreter located in your virtual environment (it should be something like venv\Scripts\python.exe on Windows or venv/bin/python on macOS/Linux).**

#### **7. Run the Example Scripts**

**Now you can run your scripts. Open any of the main Python files (e.g., main.py, main2.py) and run them using the built-in terminal or the VS Code Run/Debug options.**

1. **Open main.py or any other script you want to run.**
2. **Press F5 to start debugging, or right-click on the script and select Run Python File in Terminal.**

**Section 1: Explanation of all the files and code**

**1. SR830.py**

**This module provides a class for interfacing with the SR830 lock-in amplifier using the PyVISA library.**

#### **SR830 Class**

* **Attributes:**
  + **address: The VISA address of the instrument.**
  + **gpib\_address: The GPIB address of the instrument.**
  + **timeout: The timeout for VISA communications.**
  + **rm: ResourceManager instance.**
  + **instrument: The instrument instance.**
* **Methods:**
  + **\_\_init\_\_(self, address, gpib\_address, timeout=10000): Initialises the SR830 class.**
  + **connect(self): Establishes a connection to the instrument.**
  + **disconnect(self): Closes the connection to the instrument.**
  + **set\_gpib\_address(self): Sets the GPIB address of the instrument.**
  + **query(self, command): Sends a command to the instrument and returns the response.**
  + **get\_idn(self): Queries the instrument identification.**
  + **get\_param(self, param): Queries a parameter from the instrument.**
  + **close(self): Closes the connection to the instrument.**

### **2. SR830\_ophyd.py**

**This module integrates the SR830 instrument with the Ophyd framework for control and data acquisition.**

#### **SR830Device Class**

* **Attributes:**
  + **idn: Signal for instrument identification.**
  + **freq: Signal for frequency.**
* **Methods:**
  + **\_\_init\_\_(self, address, gpib\_address, name="sr830", \*\*kwargs): Initializes the SR830Device class.**
  + **connect(self): Connects to the SR830 instrument.**
  + **disconnect(self): Disconnects from the SR830 instrument.**
  + **read\_idn(self): Reads the instrument identification.**
  + **read\_frequency(self): Reads the frequency from the instrument.**
  + **close(self): Closes the connection to the instrument.**

### **3. instrument\_base.py**

**This module defines the base class for instruments and a basic signal class for use with the Ophyd framework.**

#### **InstrumentBase Abstract Class**

* **Attributes:**
  + **address: The address of the instrument.**
  + **instrument: The instrument instance.**
* **Methods:**
  + **connect(self): Abstract method to connect to the instrument.**
  + **disconnect(self): Abstract method to disconnect from the instrument.**
  + **query(self, command): Abstract method to send a query to the instrument.**
  + **write(self, command): Abstract method to send a command to the instrument.**

#### **BasicSignal Class**

* **Attributes:**
  + **get\_func: Function to get the signal value.**
  + **set\_func: Function to set the signal value.**
* **Methods:**
  + **get(self): Gets the current value of the signal.**
  + **put(self, value, \*, timestamp=None, force=False): Sets the value of the signal.**

### **4. main.py**

**This module demonstrates the use of Bluesky for running experiments and data acquisition.**

#### **Key Components**

* **RunEngine: Manages the execution of plans.**
* **PersistentDict: Stores metadata persistently.**
* **Temporary Catalog: Stores experiment data temporarily.**
* **print\_readings(name, doc): Callback function to print readings from events.**
* **create\_unique\_file\_prefix(prefix): Creates a unique file prefix based on the current timestamp.**
* **export(docs, directory, file\_prefix): Exports experiment data to a CSV file.**
* **display\_csv(csv\_file): Displays the contents of a CSV file.**

### **5. main2.py**

**Similar to main.py, this module also demonstrates the use of Bluesky for running experiments, specifically involving SR830 and SIM928 devices. Just another example for the developer.**

#### **Key Components**

* **print\_readings(name, doc): Callback function to print readings from events.**
* **create\_unique\_file\_prefix(prefix): Creates a unique file prefix based on the current timestamp.**
* **export(docs, directory, file\_prefix): Exports experiment data to a CSV file.**
* **display\_csv(csv\_file): Displays the contents of a CSV file.**

### **6. mercury\_itc.py**

**This module provides a class for interfacing with the Mercury ITC temperature controller.**

#### **MercuryITC Class**

* **Attributes:**
  + **device\_address: The address of the device within the Mercury ITC system.**
* **Methods:**
  + **connect(self): Connects to the Mercury ITC.**
  + **disconnect(self): Disconnects from the Mercury ITC.**
  + **query(self, command): Sends a command to the instrument and returns the response.**
  + **write(self, command): Sends a command to the instrument.**
  + **read\_temperature(self): Reads the temperature from the instrument.**
  + **set\_gas\_flow(self, temperature): Sets the gas flow based on the temperature.**
  + **set\_p\_value(self, p\_value): Sets the P value for the PID controller.**
  + **set\_i\_value(self, i\_value): Sets the I value for the PID controller.**
  + **set\_d\_value(self, d\_value): Sets the D value for the PID controller.**
  + **set\_heater\_setpoint(self, h\_setpoint): Sets the heater setpoint.**
  + **set\_flow\_setpoint(self, f\_setpoint): Sets the flow setpoint.**
  + **get\_gas\_flow(self): Gets the current gas flow.**
  + **get\_p\_value(self): Gets the current P value.**
  + **get\_i\_value(self): Gets the current I value.**
  + **get\_d\_value(self): Gets the current D value.**
  + **get\_heater\_setpoint(self): Gets the current heater setpoint.**

### **7. mercury\_itc\_ophyd.py**

**This module integrates the Mercury ITC temperature controller with the Ophyd framework.**

#### **MercuryITCController Class**

* **Attributes:**
  + **temperature: Signal for temperature.**
  + **d\_value: Signal for D value.**
* **Methods:**
  + **\_\_init\_\_(self, name, visa\_address, \*\*kwargs): Initializes the MercuryITCController class.**

### **8. sim928.py**

**This module provides a class for interfacing with the SIM928 voltage source.**

#### **SIM928 Class**

* **Methods:**
  + **connect(self): Connects to the SIM928.**
  + **disconnect(self): Disconnects from the SIM928.**
  + **query(self, command): Sends a command to the instrument and returns the response.**
  + **write(self, command): Sends a command to the instrument.**
  + **connect\_to\_slot(self, slot): Connects to a specific slot on the SIM928.**
  + **query\_idn(self): Queries the instrument identification.**
  + **set\_voltage(self, slot, voltage): Sets the voltage for a specific slot.**
  + **get\_voltage(self, slot): Gets the voltage for a specific slot.**

### **9. sim928\_ophyd.py**

**This module integrates the SIM928 voltage source with the Ophyd framework.**

#### **SIM928Controller Class**

* **Attributes:**
  + **voltage: Signal for voltage.**
* **Methods:**
  + **\_\_init\_\_(self, name, visa\_address, slot, \*\*kwargs): Initializes the SIM928Controller class.**
  + **\_set\_voltage(self, voltage): Sets the voltage using the SIM928 class.**
  + **\_get\_voltage(self): Gets the voltage using the SIM928 class.**

**Section 2: Current Problems with the code**

1. **The code is completely backend and therefore not as abstract. There is no proper experiment control, in the sense that plotting of parameters cannot be done just yet. I have implemented the code in main.py to do graphical plottings, but don't have enough time to actually implement that with the code. This could be included fairly easily using matplotlib or some other library.**
2. **There has to be another level of abstraction like, maybe start/stop the experiment, get/set multiple parameters at once, while the code runs. I believe this can be added once the frontend is developed.**

**Section 3: Integrating Frontend and Backend**

**This project is designed to create a web application that allows users to conduct multiple experiments, control various parameters simultaneously, and visualise the results. The backend is built using BlueSky, and the frontend can be built using React. This provides a guide for developers to understand the structure and workflow of the project, offering a structured approach while leaving room for customization based on the developer's preferences.**

#### **Possible Project Structure**

* **Backend: Manages API requests, connects to experimental devices, runs experiments using BlueSky, and stores data.**
  + **Technologies: Flask, BlueSky, SQLAlchemy (for database interactions), DataBroker.**
* **Frontend: Provides a user interface to interact with the backend.**
  + **Technologies: React, Axios, Chart.js/D3.js (for plotting data).**

### **Backend**

#### **Setting Up the Backend**

1. **Project Initialization**
   * **Create a virtual environment and install necessary packages.**
2. **Flask Application**
   * **Initialise the Flask application.**
   * **Set up database configurations using lib like SQLAlchemy.**
   * **Define your database models.**
   * **Create API endpoints to handle experiment requests and data retrieval.**
3. **BlueSky Integration**
   * **Set up BlueSky to run and manage experiments.**
   * **Ensure experiments' results are inserted into DataBroker for easy retrieval and analysis.**
4. **Database Setup**
   * **Use SQLAlchemy to define models for storing experiment data.**
   * **Configure the database to store and retrieve data efficiently.**

#### **API Design**

* **Experiment Control**
  + **Endpoints to start, stop, and monitor experiments.**
  + **Endpoints to adjust experiment parameters dynamically.**
* **Data Retrieval**
  + **Endpoints to fetch experiment results.**
  + **Endpoints to query historical data.**

### **Frontend**

#### **Setting Up the Frontend**

1. **Project Initialization**
   * **Set up a React project**
   * **Install necessary packages (e.g., Axios for API calls, Chart.js/D3.js for data visualisation).**
2. **Component Structure**
   * **Plan the component structure to ensure maintainability and scalability.**
   * **Consider components for:**
     + **Experiment control panel.**
     + **Parameter adjustment interface.**
     + **Data visualisation and plotting.**
3. **API Integration**
   * **Use Axios to connect the React frontend to the Flask backend.**
   * **Ensure efficient handling of data fetching and error management.**
4. **Data Visualization**
   * **Implement dynamic charts to visualize experiment data.**
   * **Ensure real-time updates and interactivity for better user experience.**

### **Integration with BlueSky**

1. **Setting Up BlueSky**
   * **Ensure BlueSky is correctly set up and integrated with the Flask backend.**
   * **Define workflows for running experiments and handling data.**
2. **Experiment Workflows**
   * **Create modular workflows that can be easily extended or modified.**
   * **Ensure the system can handle multiple experiments simultaneously.**
3. **Data Management**
   * **Use DataBroker to manage and retrieve experiment data efficiently.**
   * **Ensure data integrity and consistency across different parts of the application.**

### **Deployment and Maintenance**

1. **Deployment**
   * **Set up deployment pipelines for both frontend and backend.**
   * **Consider using cloud services like AWS, Azure, or Heroku.**
2. **Maintenance**
   * **Ensure proper logging and monitoring.**
   * **Set up automated tests and continuous integration/continuous deployment (CI/CD) pipelines.**