Development of a Zenodo Jupyter Lab Extension

Michael Ryan Zengel
CERN Summer Student Programme

Abstract
The Virtual Research Environment aims to improve the technical experience of physicists by aggregating infrastructure and software into a single, cloud-based platform [1]. A significant portion of this Jupyter-based analysis facility is the ability to upload and download data from public repositories such as Zenodo. This project successfully saw the development of a Jupyter Extension that provides a visual interface to interacting with the Zenodo REST API.

Keywords
CERN report; open-source; VRE; cloud-based; Zenodo

Supervisors

Enrique Garcia Garcia and Giovanni Guerrieri (IT-ENG-GOV)

Introduction and Motivation

The Virtual Research Environment (VRE) aims to improve the technical experience of physicists by aggregating infrastructure and software into a single, cloud-based platform, following Open Source and FAIR best practices [1]. This involves the development of a Jupyter Hub based service with infrastructure built to develop a fully-scoped analysis facility. The development of this end-to-end scientific analysis workflow is built off of 4 main pillars of the life cycle of these scientific studies: data retrieval, data analysis, publishing of data, and the ability to easily re-analyze data. This workflow framework is shown in Figure 1.

This project focuses on the publication and preservation of results within the VRE. This goal is achieved via a connection to public repositories, such as ArXiv, Github, Zenodo, etc [2, 3, 4]. For the scope of this project, the implementation of a Zenodo feature in the VRE has been the focus. More specifically, the VRE is hosted in a Jupyter Hub service, and thus, the implementation of Zenodo into this environment has been via the development of a Jupyter Lab extension.

Besides as a feature in the development of the VRE, this extension also serves several other important applications. The development of a visual interface with the Zenodo REST API Command Line Interface (CLI) increases the speed and ease with which it can be used. Furthermore, with the implementation into a Docker image and a Jupyter framework, the need for local storage use in downloading or uploading various data to Zenodo is removed significantly, creating the ability to have a fully cloud-based interface.

Documentation

This section is meant to serve as a general overview of the structure and capability of the extension. The full documentation can be found attached to the Github Repository as a Github Pages documentation [5, 6].

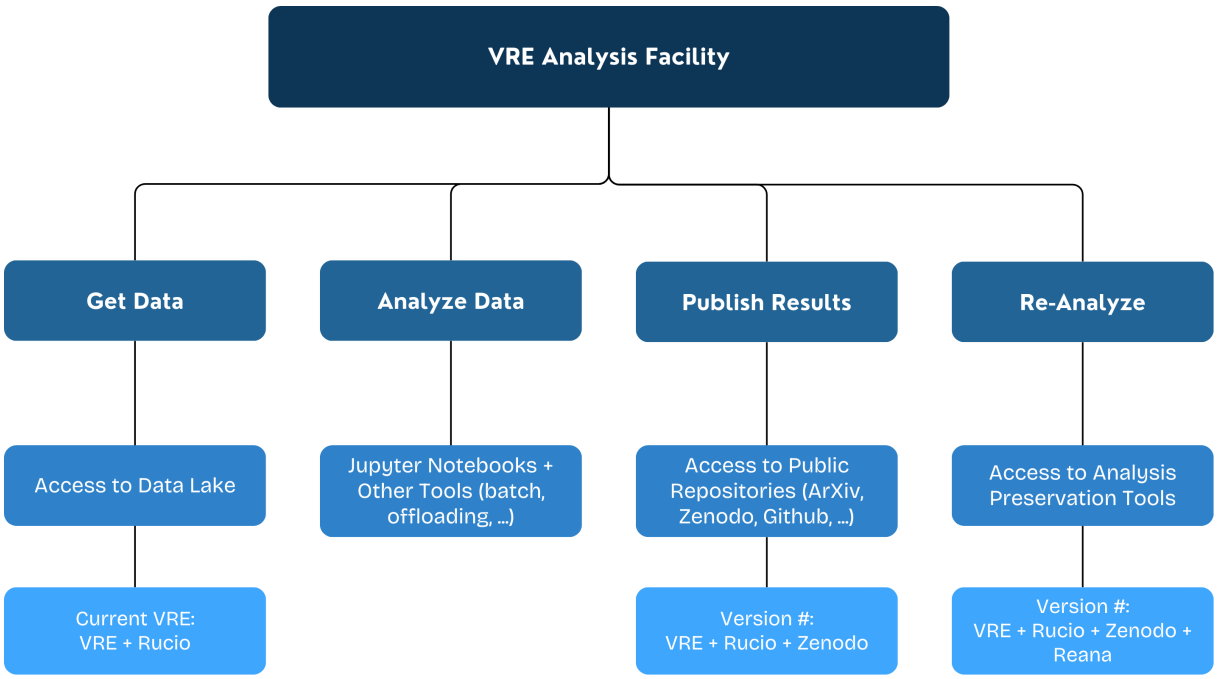

Figure 1: Diagram of VRE Workflow

General Framework

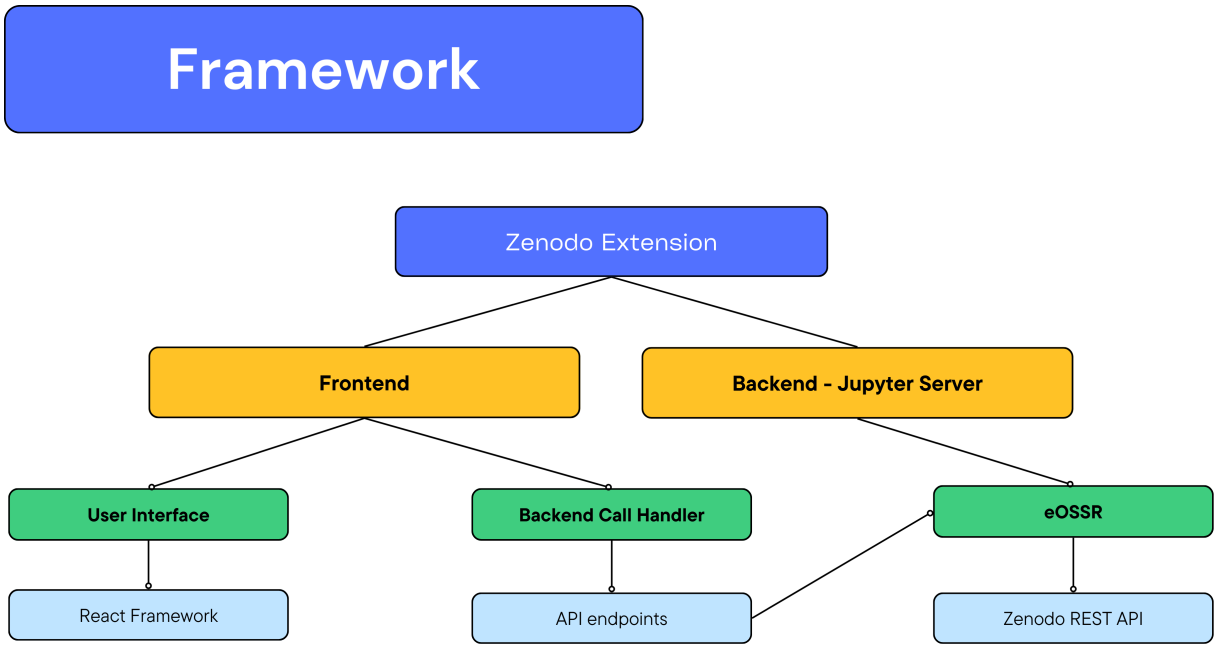

Figure 2: Diagram of Zenodo Jupyter Lab Extension

The general design of this extension is shown in Figure 2. The extension is organized in a frontend and a backend. The frontend component serves as an interface to the user, interpreting the users commands and communicating to the API endpoints defined in the backend. These two sections are discussed below.

Frontend

The frontend portion of the extension is referred to as a Jupyter Lab extension, as it extends the usage of the application Jupyter Lab. The compiling of the of the frontend JupyterLab extension is handled entirely within `pyproject.toml` in the root directory, which was originally generated from the Jupyter Lab Extenstion `copier` template [7]. This file defines the extension and registers it with the Jupyter service once the extension is installed. The rest of the frontend components were also originally generated with `copier` template, but have been modified heavily, including converting the majority of the extension into React components, rather than bare Typescript. The change to React was made due to React's component-based architecture and efficient state management, that lends itself to dynamic UI's that are seamlessly responsive to user's inputs. This extension is built with NodeJS >= 20 with node dependencies contained within `yarn.lock`; these node dependencies represent every imported package needed for the extenion to successfully build and render.

The JupyterLab extension is comprised of many React components imported into one main widget. These are all housed within the `src/` directory within the project. This widget, named `ZenodoWidget` extends the `@lumino/widgets/Widget` class. It should be noted that this component is wholly defined without React, and it is only within the sub-components that React begins to be used. Within the main component, state variables are defined which describe the status of which page is showing: login, search, or upload. An example of the user interface is shown in Figure 3. These are passed through to the `SideBarPanel` component within a triggerable `render` function. Within the main `index.tsx` file, an activation function is also defined that declares several functions responsible for toggling these state variables and declares the logic for rendering the widget if it is disposed and generates the tracker. One last feature of this file is it defined the `JupyterFrontEndPlugin` which expresses which Jupyter modules are required to render the widget and attaches the activation function to the plugin.

The rest of the frontend widget content is housed within the `SideBarPanel` component. Within this, there are three primary components: `login`, `search`, and `upload`. `Search` is the default component rendered upon initial loading of the extension. The documentation of the logic driving the functionalities behind these components is addressed in the following sections.

Login

This component houses a permanent field for the user to enter their Zenodo access token, as well as a checkbox field indicating whether or not the user wishes to log in to the main software or the sandbox software [4, 8]. Once login is triggered, functions, defined in `API/API_functions.tsx`, make the following requests to API endpoints defined in the backend portion of this application: a POST request to `zenodo-jupyter/env` to set the `ZENODO_API_KEY` and `ZENODO_SANDBOX` environmental variables, available only to the user within the Jupyter session in which they were defined, and a POST request to the `zenodo-jupyterlab/zenodo-api` API endpoint with the specified action of "check-connection". This last API call verifies the given Zenodo access token via the status code of a query of that user's deposits. The success or failure of both saving the variables to the environment and checking the validity of the access token are displayed below the login field.

Search

This component has a permanent search field, with checkboxes specifying whether the user wishes to search through Zenodo records or communities (which are collections of records). When the user searches, the `search_field` is passed to the `zenodo-jupyterlab/search-records` or

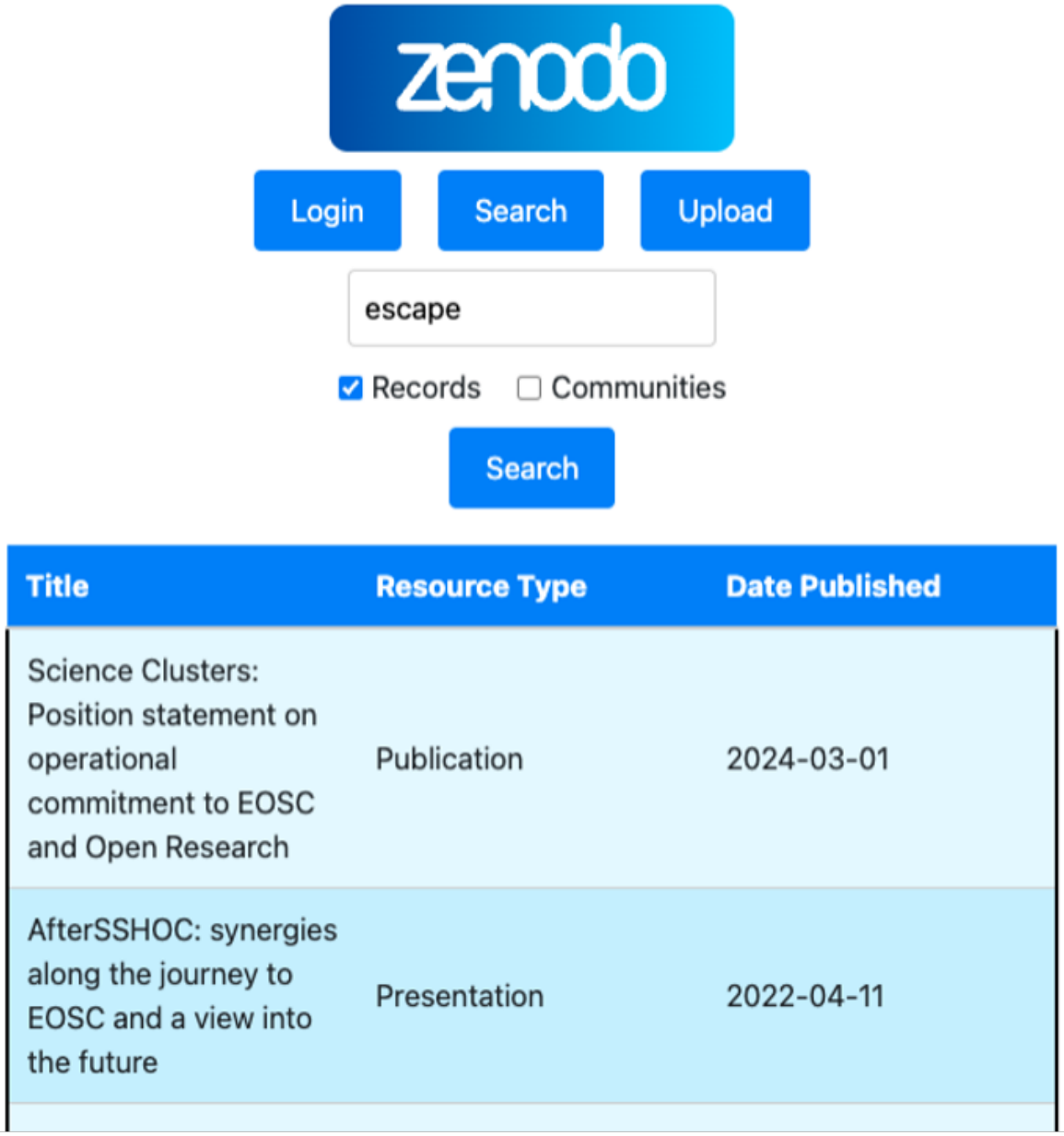

Figure 3: Example of the extension's user interface, showcasing the 3 tab options at the top: login, search, and upload. Also displays an example of the search feature.

`zenodo-jupyterlab/search-communities` API endpoints. These make API calls via `eOSSR`, a python package designed for interacting with Zenodo's REST API [9]. These search results are then mapped to a dynamic table and displayed in order of most recently modified on pages of 25, with varying levels of information for each: Title, Resource Type, and Date Published for records and Title and Date Published for communities. When the "Next Page" or "Last Page" buttons are clicked below the results, the page number is incremented or decremented, and passed back through the API call as an encoded URI component, leveraging the built-in `page` parameter in when querying records in `eOSSR`.

When specific returned results are clicked, there are more functionalities. If the user were to click on a record, a call is made to the `zenodo-jupyterlab/record-info` API endpoint, with the

specific record ID encoded. This returns more specific information on that record, such as the link, author list, and attached files with download links. This information is then dynamically displayed below the selected record, with the authors' affiliations appearing upon hovering.

When a user selects a specific community, the user is then able to search within this community. This is accomplished by making more calls to `zenodo-jupyterlab/search-records`, but with the community now encoded as well. This is then passed as a `kwarg` to the `eOSSR` function call.

Upload

This component serves as an visual interface for users to generate new records on Zenodo. An important feature to note here is that there is a readonly checkbox at the top of this component that indicates whether or not the user has logged in to the main Zenodo service or the sandbox version, as this will be the service to which the upload is made. Another important feature of this component is the `Filebrowser.tsx`. This file browser defaults to the `$HOME` directory within the Jupyter instance. This is significant in that when using this extension on a cloud based environment (i.e. the VRE), there is no need for local storage to make file uploads to Zenodo.

The component takes in the most basic required information for a deposit via a `FormData` object. Once the user presses next, they reach a confirmation page, which displays all of the entered information. If the user selects "Confirm", a call is made to the `zenodo-jupyterlab/zenodo-api` API endpoint, with the specified action of "upload" and the encoded form data. This portion of functionality is still a work in progress, but, as of the end of my appointment, once the API endpoint receives the command, it will generate a new deposit using the user's credentials, then add a title to this deposit using the form data and the record ID of the new deposit.

Backend

The backend for this extension is built as a Jupyter server extension. The project entry points are specified with the `pyproject.toml` file in the root directory. These point to the `zenodo_jupyterlab.server` module, which contains the `extension.py` and `__init__.py` files which run the function that sets up the API handlers defined within other files in that directory. This section intends to outline the general interactions and sections of this backend service.

Interaction with `eOSSR`

`eOSSR` is a python package developed with the express functionality of interacting with the Zenodo REST API. It was developed as a part of the ESCAPE collaboration (those who are also responsible for the creation of the VRE). This package serves to simplify the extensive Zenodo REST API and make it more usable for specific cases by reducing functionality. In essence, it takes the burden of making API calls directly to Zenodo off of the user.

This package is how all interaction with Zenodo is done within the extension. More specifically, via the `search_records`, `search_communities`, and `get_record` functions, the search capabilities of the extension of achieved. As for the login and upload features, these are both handled by the `ZenodoAPI` object found within `eOSSR`. What this object does is save a connection configuration (i.e. access token, whether or not the connection is to the sandbox), and allows for multiple requests with the same configuration. Thus, when a user logs in, this object is generated and stored for use when uploading, to reduce both the amount of calls to the Zenodo API to avoid rate limits and the amount of complexity of making a new connection to the REST API.

API Handlers

The API endpoints defined within `zenodo-jupyterlab.server.handlers` act as listeners for calls from the frontend component. When accessed via their defined url, they are able to run python code on the actual instance of Jupyter where Jupyter Lab is being run. This removes the need for hosting a backend server in Flask for example. Further, this platform allows for the storage of environmental variables available only to the user on whichever shared Jupyter instance they are using. This platform also reduces the need for client side API calls to the Zenodo API, reducing the amount of network connections the user's local computer must make.

All of these API Handlers inherit from `jupyter_server.base.handlers.APIHandler`.

Packaging and Installation

Installation

There are two primary methods of installing and running the extension; they are as follows. An area of future development is the creation and deployment of a PyPi python package containing the extension for availability via `pip install zenodo-jupyterlab-extension`. This would also make it available within the Jupyter Lab Extension store.

Github Repository

The following steps require NodeJS >= 20. It is recommended to do these steps within a virtual environment of some sort, so that there are no invalid dependencies or package issues that affect other projects. *Note*: this consideration is handled when using Docker.

Clone the repository:
```
git clone https://vre-hub.github.io/zenodo-jupyterlab-extension/
```

Install `yarn`:
```
npm install -g corepack
corepack enable
```

Install Python dependecies:
```
python3 -m pip install -r requirements.txt
```

Install `yarn` dependencies:
```
jlpm
```

Install and Build the Extension:
```
python3 -m pip install "-e" .
```
Note: the `-e` tag enables editable mode, which is only recommended for developers.

Enable the server extension:
```
jupyter server extension enable zenodo_jupyterlab.server
```

If these steps are followed, the extension should be active once an instance of Jupyter Lab is run.

Docker

The package has also been packaged as a Docker image. This means that there is a downloaded environment which contains all of the required dependency software.

The most simple way to download and install the extension is via Docker:

`docker run -d -p 8888:8888 ghcr.io/vre-hub/zenodo-jupyterlab-extension:<version>`

All available versions can be found here [10]. *Note*: the latest version can be accessed via the <latest> tag, while the most recent development version can be accessed via the <dev> tag.

Once the image is running, the instance of Jupyter Lab with the extension installed and enabled should be available on localhost:8888.

Usage Guide

Searching

The user has the ability to search both records and communities, which are collections of records. The actual search query ability is derived from Elasticsearch, just as the Zenodo REST API is [11, 12].

The results are returned in pages of 25 results, with the ability to navigate the pages of results. This leverages the built-in `page` and `size` parameters present in the Zenodo REST API, which appear as `kwargs` in `eOSSR`, the python library on which all backend communication with Zenodo is based.

For searched records, the title, resource type, and date published are displayed in the table. For communities, only the title and date published are displayed. Both searches are returned and sorted by most recently updated.

Note: As each page is a new API call, if records are added that match the search parameters in between page calls, there is a possibility for repeating records, as they are pushed to later pages. Though, given the probable specificity of users' searches, this should not be an issue.

Record Information

When a record is selected, more record information is displayed:

- Title of the record as a link to the Zenodo record entry
- Author list, with institutions available upon hovering
- List of files as download links (WIP, currently install to local computer, but in future updates will install to the `$HOME` directory in the Jupyter instance)

This interface is meant to mimic the record entries on the Zenodo web client.

Community Searching

Upon searching for a community, when clicked, the user will be able to search for specific records within that community, and a banner will display which community is being searched through above the results. *Note*: the community searching can be canceled by either pressing the "X" next to this community banner or searching for a new community.

Potential Future Features

Depending on the usefulness of the feature in terms of the scope of this extension, advanced search settings might be added in future updates. These would mimic those on the Zenodo web client: restrict file type, resource type, access level, etc.

ZenodoAPI interactions

This section involves all actions that interact with the `eossr.api.zenodo.ZenodoAPI` module, which creates a connection to Zenodo with an API access token for continued use.

Logging in

The user is able to either log in to the main Zenodo software or the Sandbox Zenodo software (for testing) using their Personl Access Token. This token is created in the Zenodo User Settings > Applications > Personal Access Tokens. When the user logs in, the validity of their token in the chosen software is determined via a query of that user's deposits. If the token is invalid, this is displayed clearly below the access token field.

The entered access token and choice of whether or not to work within the main or sandbox software are saved as environmental variables within the Jupyter Session: `ZENODO_API_KEY` and `SANDBOX_ZENODO`, respectively. These variables will be accessible across the Jupyter Session, however they will not be reflected in any open terminals; new terminals/notebooks must be opened to reflect the change.

Uploading A Record

The sandbox checkbox at the top of the Upload page is readonly and indicates whether of not the user logged in to the main software or the sandbox version. This is to inform the user of in which software their deposit will be made.

In order to continue, the user had to fill in the necessary information:

- Select at least one file to upload (the file browser initially shows the contents of the `$HOME` directory in the Jupyter Session)
- Title
- Select a Resource Type
- Include at least one Creator's name
- Optional Information:
  - DOI (otherwise generated automatically)
  - Multiple Creators and include affiliations

Once the "Next" button is selected, a confirmation page is shown with all of the entered information. When "Confirm" is pressed, a deposit is made and the appropriate metadata is assigned. *Note*: As is currently stands, only the title is able to be set via this mechanism, and the file upload has yet to be implemented.

Conclusion

The successful development of a Jupyter Lab extension for a visual interface with the Zenodo software has applications in any Jupyter-based environment. Most notably, this is compatible with the continuing development of the Virtual Research Environment by the ESCAPE collaboration. This extension thus contributes to the creation of a fully cloud-based, end-to-end analysis facility, in which researchers are able to retrieve data, analyze their data, publish their results, and create re-analysis workflows. Specifically, the integration of Zenodo into this environment removes the requirement for local storage when downloading and uploading files to Zenodo. Due to the open-source documentation and repository, the development and use of this extension is widely available.

Appendix

Structure of Code

Here is the basic folder structure of the major frontend and backend components of this extension. As is shown in the schema, the frontend components are contained within the `src/` directory, while the backend components are contained within the `zenodo_jupyterlab/server` directory.

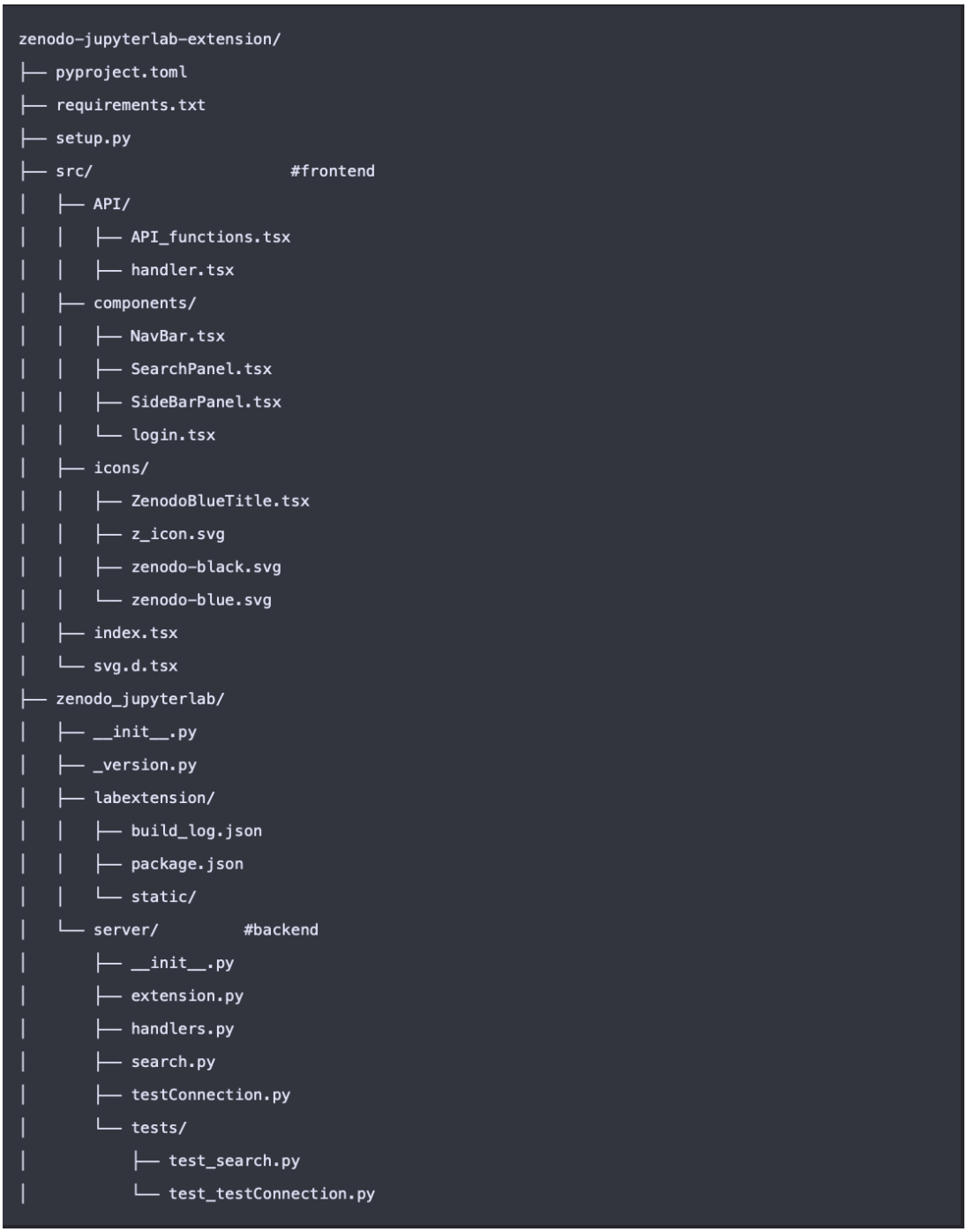

Figure .1: Diagram of Zenodo Jupyter Lab Extension

List of URLs

[1] *VRE Hub on GitHub*. URL: https://github.com/vre-hub.
[2] *ArXiv*. URL: https://arxiv.org.
[3] *GitHub*. URL: https://github.com.
[4] *Zenodo*. URL: https://zenodo.org.
[5] *Zenodo JupyterLab Extension - VRE Hub*. URL: https://github.com/vre-hub/zenodo-jupyterlab-extension/tree/main.
[6] *Zenodo JupyterLab Extension Documentation*. URL: https://vre-hub.github.io/zenodo-jupyterlab-extension/.
[7] *JupyterLab Extension Template*. URL: https://github.com/jupyterlab/extension-template.
[8] *Zenodo Sandbox*. URL: https://sandbox.zenodo.org.
[9] *eOSSR on PyPI*. URL: https://pypi.org/project/eossr/.
[10] *Zenodo JupyterLab Extension Docker Image*. URL: https://github.com/vre-hub/zenodo-jupyterlab-extension/pkgs/container/zenodo-jupyterlab-extension.
[11] *Query DSL - Elasticsearch*. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-query-string-query.html.
[12] *Zenodo REST API*. URL: https://developers.zenodo.org/#rest-api.