TaskLabs - Pyodide Tools Documentation

Table of Contents

- Overview
- Folder Structure
- Good Practices
- Javascript States
- JavaScript Functions
- JavaScript Forms & Input
- Python and JavaScript Bridging
- Pyodide Python Envrionment & Functions

Overview

Every tool has a unique way of being built, however, the fundamentals of the data flow for every tool is the same.

Most tools will be made using a combination of JavaScript and Python code. We'll be using **Pyodide**, which will help set up a virtual environment to run Python code within the browser. A bridge is essentially created to exchange data between the two environments.

A user uploads a set of data through a form, and this **data is stored in states**. Input validation is performed on this data. **Files will be read as binary strings** and stored in a list.

All of the inputted data within JavaScript will be declared as **global variables in the Python** environment. The next step is to run a Python script that performs all the tool related tasks that are required and will return the downloadable data and error messages back to the JavaScript enviornment.

A downloadable BLOB URL will be created with the downloadable data, and error messages will be displayed eventually.

Folder Structure

All the JavaScript code will be written in a `page.tsx` file in the `src/app/tools/<tool_name>` folder. Whereas the Python code will be placed in a string within a `<tool_name>.js` file inside the `src/python/<tool_category>` folder.

Good Practices

- It's adviced that all your code is written in a `.py` file first, and is then put into a string. It is suggested to make changes first only in the `.py` file, and then test the working of that `.py` file before putting it into the string.
- It's recommended to create functions for readability purposes and to reuse code in several places instead of re-writing them.
- Try to keep most code in Python within `try-except` blocks in order to avoid abprut program terminations due to errors.
- Consider several edge cases that might arise and try to accordingly tackle them.

Javascript States

1. **Pyodide State** - Within the `usePyodide` function, we pass in a list of all the Python libraries (as strings) we want to install for the functioning of our Python script.
   Example:

```
const [pyodide, isPyodideLoaded] = usePyodide(["pandas", "openpyxl"])
```

2. **Input Data State** - Within the `data` state, we will store `download_blob`, `download_file_name`, and `not_converted_files` along with several *other input data variables*.
   Example:

```
const [data, setData] = useState<{

        // mandatory states or pyodide result states
        download_blob: string;
        download_file_name: string;
        not_converted_files: string[];

        // states of input fields
        files: FileList | null;
        all_in_one: boolean;

}>({
// initialization
        download_blob: "",
        download_file_name: "",
        not_converted_files: [],

        files: null,
        all_in_one: false,

});
```

JavaScript Functions

1. `setEmptyResult` - This function takes in no argument. It's primary purpose is to reset the states of `download_blob`, `download_file_name`, and `not_converted_files` (also known as *pyodide result states*) by making them empty. This function doesn't interfere/change the values of any other states present in the `data` state.

```
const setEmptyResult = () => {
        setData((prev) => {
                return {
                        ...prev,
                        download_blob: "",
                        download_file_name: "",
                        not_converted_files: [],
                };
        });
};
```

2. `downloadFile` - This function takes 3 arguments: `content` which is the downloadable data, `filename` which is the name of the file we want the user to download, and `mime_type` which stores the MIME type of the file (true identity of the type of the file). This function then creates a BLOB object with the content and a data URL for the same. Furthermore, the states of the `download_blob` and `download_file_name` are updated so that they can be picked by the download button that appears post a successful result. **It is to be noted that this JavaScript funtion will be declared as a Python global function and will be called from Python.**

```
const downloadFile = (content: any, filename: string, mime_type: any) => {

        const blob = new Blob([content.toJs()], { type: mime_type });
        const url = URL.createObjectURL(blob);
        setData((prev) => {
                return { ...prev, download_blob: url, download_file_name:
filename };
        });
};
```

3. `validateFileSizes` - This function takes as argument a `FileList` . This function ensures that the number of files uploaded doesn't exceed a maximum uploadable limit, and verifies whether each file is below the maximum uploadable size limit per file. If either of the conditions is not met, then it returns with a status code of `0`. If both the conditions are met, then it returns with a status code of `1`.

```
const validateFileSizes = (fileList: FileList | null) => {

        if (fileList === null) {
                return 1;
        }
```

```
        if (fileList.length > max_number_of_files) {
                return 0;
        }
        for (let i = 0; i < fileList.length; i++) {
                if (fileList[i].size > max_mb_size * 1048576) {
                        return 0;
                }
        }
        return 1;
};
```

4. `convertFilesToBinaryString` - This function takes as argument a `FileList`. With the help of the `FileReader` class and `Promise`, it seequentally attempts to convert each file in the file list to a binary string. This function returns 2 arrays: `convertable`, having tuples of `(file_name,binary_string)` of all the files that could successfully be converted into one, and `not_convertable` which contains the names of the files that couldn't be converted to binary strings.

```
const convertFilesToBinaryString = (fileList: FileList | null) => {

  console.log("Entered convertToBinaryString Function");
  let convertable: any[] = [];
  let not_convertable: any[] = [];
  if (fileList === null) {
    return { convertable, not_convertable };
  }
  let promises: Promise<any>[] = [];
  for (let i = 0; i < fileList.length; i++) {
    console.log("Entered for loop: ", i);
    promises.push(
      new Promise((resolve, reject) => {
        var fr = new FileReader();
        fr.onload = () => {
          let fdata = fr.result;
          convertable.push([fileList[i].name, fdata]);
          resolve(fdata);
        };
        fr.onerror = () => {
          not_convertable.push(fileList[i].name);
          reject(`'${fileList[i].name}' cannot be read!`);
        };
        fr.readAsBinaryString(fileList[i]);
      })
    );
  }
  console.log("convertable: ", convertable);
  console.log("not_convertable: ", not_convertable);
  return Promise.all(promises).then(() => {
```

```
    return { convertable, not_convertable };
  });
};
```

5. `mainFunction` - This function is called when the form has been submitted by the user (ie when the `submit` button has been clicked). First if `pyodide` has been loaded is checked, then it is made sure that all the mandatory input fields have been entered. If either of these two conditions is not satisfied, then the function is exited with appropriate error messages. After this, files are validate and converted to binary strings upon successful validation. All the JavaScript variables needed in our Python environment will be set as Python global variables using `pyodide.globals.set(Python_variable_name,JS_variable_data)`. The Python script is then run using `pyodide.runPythonAsync(script)` and this will garner a set of Python variables that can be used in the JavaScript environment. Error messages can be accordingly shown then.

```
const mainFunction = () => {
  async function execute() {
    if (!pyodide) {
      toast.error("Internal Error! Please try again later :)", {
        duration: 5000,
      });
      return;
    }

    if (!data.files) { // for example, if files input is mandatory
      toast.error("Please select files to convert!", {
        duration: 5000,
      });
      return;
    }

    setLoading(true);
    try {
      if (!validateFileSizes(data.files)) {
        throw new Error(
          `Each file size should be less than ${max_mb_size} MB!`
        );
      }
      var { convertable, not_convertable } = await convertFilesToBinaryString(
        data.files
      );
    } catch (error) {
      return;
    }

    // setting the global variables essentially creating a bridge
    try {
```

```
        const destination_type = format?.toUpperCase();
        pyodide.globals.set("convertable", convertable);
        pyodide.globals.set("not_convertable", not_convertable);
        pyodide.globals.set("downloadFile", downloadFile);
        pyodide.globals.set("destination_type", destination_type);

        await pyodide.runPythonAsync(script);

                // fetching python global variables into the JS env
        const notConverted = pyodide.globals.get("not_converted").toJs();
        console.log(notConverted);

        setData((prev) => {
          return { ...prev, not_converted_files: notConverted };
        });
        setLoading(false);
        if (notConverted.length > 0) {
          toast.error(
            "Failed to convert the following files: \n" +
              notConverted.join("\n")
          );
        }
      } catch (error) {
        console.log(error);
      }
    }
    execute().then;
  };
```

The 5 functions mentioned above are to be used as they are in all the tools. Additionally, feel free to create more functions on a case by case basis for the tools if you feel that there's a need for it. For example: `handleChange`, etc.

JavaScript Forms & Input

All the input data of the input fields must be stored in their respective states. A **change in any input field** must update the state and must **trigger** the `setEmptyResult` function that clears out any previous download data such that old download data doesn't interfere with any new download data that might arise due to the result of the pyodide script after the user re-submits the form. When the form has been submitted, the `mainFunction` must be called which will in-turn call other functions.

Python and JavaScript Bridging

Any variable data type/function defined within the JavaScript environment can also be used as a Python global variable with the help of `pyodide.globals.set`. Similarly, any variable data type/function defined as a Python `global` variable may be accessed in JavaScript with the

help of `pyodide.globals.get`. Pyodide tries to create a proxy of the data type if cannot be directly converted from a JavaScript data type to a Python data type or vice versa. In such a case, `to_py()` can be used within Python to convert to a JS Proxy object to a Python object. Similarly, `toJS()` can be used within the JavaScript envitonment to convert a Python Proxy object to JavaScript object.

Pyodide Python Envrionment & Functions

The Python code (ie the tool) will be placed inside a string and will be exported through a `.js` file.
For example:

```
const script = `print("all of this code is being executed")`

export script;
```

This script will be executed using `pyodide.runPythonAsync(script)` within the `mainFunction`.

All the required libraries must first be imported. Then, the driver code will be written in the `main` function. After this function, several other **utility function definitions** and the **tool function definition** which will be called in the `main` function. The main function is called at the end by `main()`.

The following libraries must in most cases need to be imported at the beginning of your Python script:

- `BytesIO` from `io`
- `mimetypes`
- `os`
- `random`
- `shutil`

Ensure that only **pure Python packages** are used for all the tools, and **make sure that they are configurable in the pyodide environment**.

It's also **essential that all the logical code is to be written within `try-except` block(s)** so that the code doesn't abruptly stop in case of any exception. Error handling is extremely critical.

Functions that are defined within the Python code:

1. `file_uid_generator` - This function takes as argument 2 strings, one is a path of a file that is to be created, and the other is the file extension. It is essential that the file path doesn't already exist, and hence if this path already exists, a randomly generated 5-digit Unique ID (UID) is generated and returned. This can be attatched at the end of the file path thereby creating a new file path.

```
def file_uid_generator(path_name,destination_type):
        charset='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
        while True:
                fuid=''
                for i in range(5):
                        fuid+=random.choice(charset)
                if not os.path.exists(f"{path_name}_{fuid}.{destination_type}"):
                        return fuid
```

2. `empty_root_folder` - This function takes in no argument. The role of this function is to remove any existing folders/files in the home directory. Several times users without reloading the page might resubmit the form. In such a case, all the files from the previous pyodide execution might still be present in the environment. Therefore it is essential that this function is called before any code of the actual tool is executed so that all previous files/folders are erased.

```
def empty_root_folder():
        for item in os.listdir():
                if os.path.isfile(item):
                        os.remove(item)
                else:
                        shutil.rmtree(item)
```

3. `tool_function` - This function takes argument `folder_name` which is essentially the name of the folder in which all the data will be saved. If there are files to be modified/utilized by the tool, then `files` is also taken as an argument which is a list containing several tuples of `file_name,binary_string_of_file`. Additionally, any form of data that is required by the tool is passed using this function. For example, an image converter tool would not only require a file list for conversion, but would also maybe need the type of the image we want to convert the files to. Therefore this function can be defined as `def tool_function(folder_name,additional_parameter1, additional_parameter2,...)`. This function first calls the `empty_root_folder` function and then creates a new folder with the name as `folder_name`. If a file list is also an argument, then the binary string of that file is converted to a `BytesIO` object. Operations are performed on this `BytesIO` object after that. If sucessful, then the new file/data is saved in the folder `folder_name`. If unsucessful, then the file name is stored in a list of unsucessfully converted files. Once all the execution is done, either the path of a single file, or a zip folder containing all the processed files will be returned as the `downloadable_location` along with the list `not_converted` containing the names of all the files that couldn't be processed. Here's an example of the complete Python code for an `excel-to-csv` conversion tool.

```
import random
import os
import shutil
from io import BytesIO
```

```
import pandas
import mimetypes

print("All imports done, Pyodide is running")

# defining the main function which will contain the driver code
def main():

        # defining global variables so that they can be accessed within the JS
env
    global download_location,not_converted

    # calling the excel_to_csv function and passing a folder name and a list of
tuples containing file data to be processed. The function returns a location of
the final output file and a list of unsuccessful conversions.
    download_location,not_converted=excel_to_csv('excel-to-csv',convertable)

        # using the JS not_convertable array and converting it to a python list
using to_py() and performing list concatenation to have a final list of the names
of files that couldn't be processed. (some couldn't be processed by JS to convert
the file to a binary string, and the remaining couldn't be processed by the
python tool function)
    not_converted=not_convertable.to_py()+not_converted

        # if there is a file to be downloaded (successful outcome), then we will
call the downloadFile function defined within the JS environment
    if download_location:
        mime=mimetypes.guess_type(download_location)[0]
        with open(download_location,"rb") as f:
            file_content=f.read()
        downloadFile(file_content,download_location.split('/')[-1],mime)


# defining the function which generates a unique ID in case a path already exists
def file_uid_generator(path_name,destination_type):

    charset='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
    while True:
        fuid=''
        for i in range(5):
            fuid+=random.choice(charset)
        if not os.path.exists(f"{path_name}_{fuid}.{destination_type}"):
            return fuid

# defining a function which removes all folders/files from the current directory
def empty_root_folder():

    for item in os.listdir():
        if os.path.isfile(item):
```

```
            os.remove(item)
        else:
            shutil.rmtree(item)

# defining the main tool function
def excel_to_csv(folder_name,files):

    empty_root_folder() # it's essential to call this function
    os.makedirs(f'{folder_name}')
    not_converted=[]

    for file_name,bin_str in files:
        bytes_obj=BytesIO(bytes(bin_str,encoding="raw_unicode_escape"))

        # it's essential to keep the whole code for any new tool within this try
and except block!
        try:
            df=pandas.read_excel(bytes_obj,sheet_name=None)
            filename=file_name[::-1].split('.',maxsplit=1)[-1][::-1]
            for sheet_name,sheet_data in df.items():
                path_without_extension=f"{folder_name}/{filename}_{sheet_name}"
                fuid='' if not os.path.exists(path_without_extension+".csv") else
'_'+file_uid_generator(path_without_extension,"csv")
                sheet_data.to_csv(f'{path_without_extension+fuid}.csv',
index=False)

        except Exception as e:
            not_converted.append(f"{file_name}")
            print(e)
            continue

        # generating the download_location. If there's just a single file, then
that file path. However, if there are multiple files, then create a zip file of
that folder and delete the original folder and return the path of this new zip
file.
    files_in_dir=[f for f in os.listdir(f"{folder_name}") if f"{f}"[0].isalnum()]
    print(files_in_dir)
    total_files=len(files_in_dir)
    downloadable_location=None
    if total_files>1:
        shutil.make_archive(f"{folder_name}","zip",f"{folder_name}")
        shutil.rmtree(f"{folder_name}")
        downloadable_location=f"{folder_name}.zip"
    elif total_files==1:
        downloadable_location=f"{folder_name}/{files_in_dir[0]}"
    else:
        shutil.rmtree(f"{folder_name}")

    return [downloadable_location,not_converted]
```

```
main()
```

Feel free to add any other functions as well in order to make the code more reusable.

Ideally it is recommended that for any new tool that is being made, the same template shared above is used. **All changes may be done only within the `try-except` block, and that there is no requirement to change any other code in the above given template.**

Ensure that no global variables are accessed by any function apart from `main`. Pass any data only as arguments to all the utility and tool functions.

Once the Python script has been executed, we access the global variables created in Python within the JavaScript environment and use it accordingly for the tool's needs.

Happy Developing :)