Jupyter Boxes

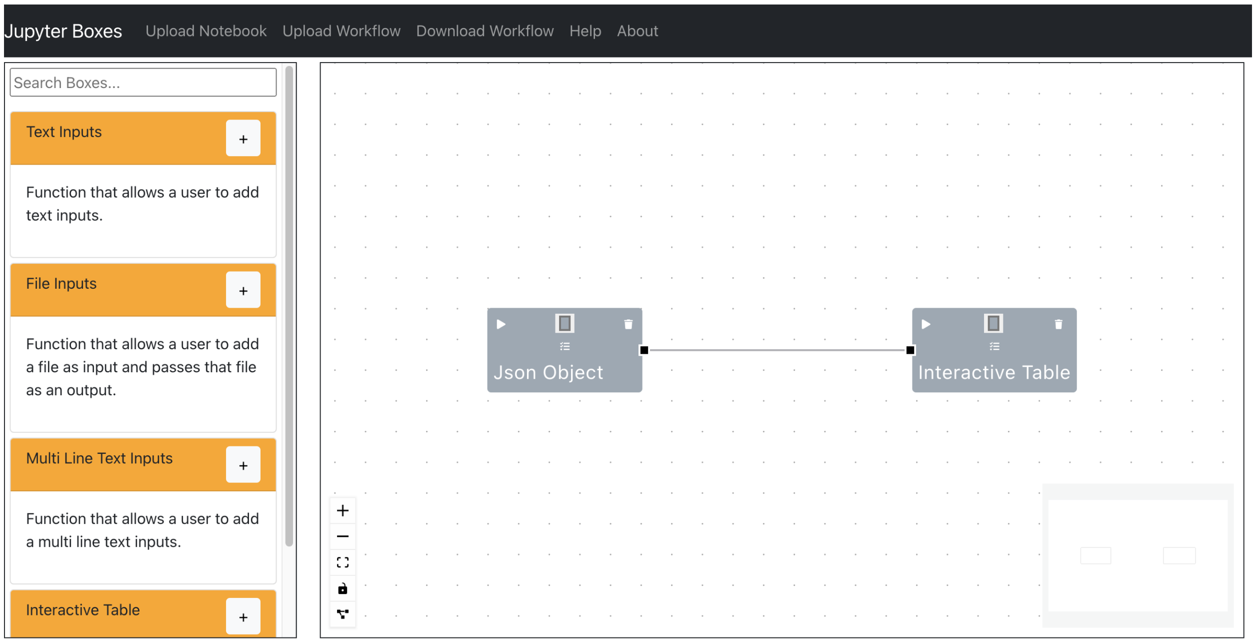
Release 0.2.2

David Fernandez

Apr 15, 2023

GETTING STARTED

Jupyter Boxes is a framework that allows a user to create a drag and drop workflow utilizing Jupyter notebook cells. The framework is build on top of React Flow. A user will add comments to the beginning of a cell that will tell the framework how to convert it into a usable block. The framwork is composed a back-end/front-end (port 9494). The backend deployed contains the code for the frontend and the parsing code for the backend. File watcher looks at the notebook directory to put new notebooks to be processed automatically. Once processed, the notebook is not looked at again (unless db reset). The processed information get stored into a file called db.json. This file contians all the blocks to display.


Jupyter Boxes
Upload Notebook
Upload Workflow
Download Workflow
Help
About
Search Boxes...
Text Inputs
Function that allows a user to add text inputs.
File Inputs
Function that allows a user to add a file as input and passes that file as an output.
Multi Line Text Inputs
Function that allows a user to add a multi line text inputs.
Interactive Table
Json Object
Interactive Table

CHAPTER

ONE

INSTALLATION

To install Jupyter boxes run the following command

```
npm install jupyter_boxes -g
```

CHAPTER TWO

JBOX COMMANDS

Once installed, the library will contain a set of commands that can be run from the terminal. These commands will help on building the environment before you cam actually use the framework.

2.1 jbox_build

usage: jbox_build [–url_path <path>] [–no_frontend]

This command will build the python virtualenv for the backend to work. Will install all required libraries into a directory containing the virtual environment called *jbox_venv*. It also installs and build the front end. Url path is used when building the frontend

2.2 jbox_reset_db

usage: jbox_reset_db [–rerun_notebooks true] [–celery true]

This command will reset the backend "database" containing the notebooks that have been parsed. This will take any notebooks in the parsed directory and return them to be parsed again. It also restarts the db.json "database" file.

> *rerun_notebooks* - will move notebooks to rerun folder
>
> *celery* - will perform the reset on the celery/RabbitMQ directory.

2.3 jbox_start

usage: jbox_start [–backend_port <9494>] [–cron_log true] [–celery]

This command will start the whole framework. This includes the backend, frontend, and cron schedule. There are two optional parameters, one for setting the backend-port and the second to allow logs to be created for the cron job. This file could be big so take into consideration. To run on the background add the *&* at the end.

> *backend_port* - Port to have the JBox running out of. Defaults to 9494
>
> *cron_log* - Notebook parsers can print alot of infomation, by setting cron_log it defaults to store into a log file.
>
> *celery* - will run the app on the celery/RabbitMQ directory.

```
jbox_start --cron_log true --backend_port port_number --celery
```

2.4 jbox_start_back_end

usage: jbox_start_back_end [–backend_port <9494>] [–url_path </>]

Starts the backend with a specified port. you can set a url path if you dont want it to be on the generic /. Meant for testing. If frontend is build, can be used the deployment.

```
jbox_start_back_end --backend_port port_number --url_path /dns/
```

2.5 jbox_start_back_end_celery

usage: jbox_start_back_end_celery [–backend_port <9494>]

Starts the celery backend wihtout dameon. Meant for testing.

```
jbox_start_back_end_celery --backend_port port_number
```

2.6 jbox_paths

usage: jbox_paths

Command to display paths available to configure JBox.

CHAPTER THREE

HOW TO START

Once you have installed JBox as global, you will have access to all the above commands on the terminal. Lets start by building the python and node environment.

```
jbox_build
```

Once the command finishes, the system will have all the required python installs and node. It will also build the frontend into the flask backend. Now there are two backends JBox can use, a RabbitMQ/Celery and a Python/Thread version. Below we can see how to run both.

3.1 Celery/RabbitMQ

In order to use the Celery/RabbitMQ version, you need to have RabbitMQ installed on your system (follow RabbitMQ website for installation. To run automatically the backend with RabbitMQ you would need to create a user and a virtualhost for JBox. Below are the commands by us.

```
sudo rabbitmqctl add_user jbox jbox
sudo rabbitmqctl add_vhost jbox_vhost
sudo rabbitmqctl set_user_tags jbox administrator
sudo rabbitmqctl set_permissions -p jbox_vhost jbox ".*" ".*" ".*"
```

Once the commands are ran you can start the app.

```
jbox_start --celery true
```

Now you can go in localhost to port 9494 and you should have access to JBox.

http://localhost:9494/

Once you put a notebook on the notebooks directory and it gets parsed, you can access your db.json file from the browser by going go the *jbox_rest* endpoint.

http://localhost:9494/jbox_rest

3.2 Thread backend

This one is more simple. Once the build command as finished you can run the *jbox_start* and you can go on from there.

```
jbox_start
```

Now you can go in localhost to port 9494 and you should have access to JBox.

> http://localhost:9494/

Once you put a notebook on the notebooks directory and it gets parsed, you can access your db.json file from the browser by going go the *jbox_rest* endpoint.

> http://localhost:9494/jbox_rest

CHAPTER FOUR

CONVERT A NOTEBOOK FOR JBOX

To convert a notebook to be used in JBox you need to add comments into the cells you want to convert. There are 3 functions that are used, one for converting, the second one for installing needed packages for the notebook, and the third one for imports for the library. Every comment used will start the following way (remmeber the space between the # and jbox):

```
# jbox <commands>
```

CHAPTER

FIVE

INSTALLING PACKAGES

In order to have the packages the notebooks need you can run the following line at the top of a cell. This is normally done at the first cell on the beginning of the notebook.

```
# jbox installations <library_names>
```

You can have multiple libary names separated by spaces.

```
# jbox installations pandas numpy ipydatatable
```

CHAPTER

SIX

IMPORTING CODE AND LIBRARIES

When looking to import a whole cell and not convert it into an end point you can use the *import* command. Import can be use on any cell, for example a cell that contains all the imports for the code or a cell with multiple functions.

```
# jbox imports
import pandas as pd

def testing():
    print("hello world)
```

CHAPTER

SEVEN

CELL CONVERSION

In order to convert a cell in a notebook into block in JBox there are multiple options. We will go over all the options availble and how to use them. Below you have all the options available.

# jbox <block_name>

[-inputs <array_of_input_names>]

[-outputs <array_of_output_names>]

[-description <description_of_block>]

[-pydoc]

[-def]

7.1 Cell with no inputs and outputs

The most generic cell, with no inputs, outputs or a defined function.

```
# jbox hello_world -inputs [] -outputs [] -description 'Function to print Hello World!'
print("Hello World!")
```

7.2 Cell with inputs and outputs

In this case you will have a cell with variables been used on the code that can be used as inputs and also variables that will be used as an output.

```
# jbox input_world -inputs [a,b] -outputs [c] -description 'Function with 2 inputs and␣
↪one output.'

c = str(a)+" "+str(b)
```

In this example the inputs are *a* and *b* and the output will be the *c*.

7.3 Defined function block

In this case we are using a function that is already created and it already has a return. This one also has a pydoc used for the description of the block.

```
# jbox output_only_function -inputs [] -outputs [c,d] -pydoc -def

def output_only_function():
    '''
        Function to show off the using a defined function and a
        pydoc as the descriotion
    '''
    c = "Hello"
    d = "World"
    return [c,d]
```

CHAPTER

EIGHT

EXAMPLE NOTEBOOK

If you would like a notebook example to get started, JBox comes with one and can also be found on the following location: Gitlab.

CHAPTER

NINE

USING THE STANDALONE BACKEND

The JBox backend can be used without the need to use the created frontend. You can replace the build directory with your own source code. This will allow JBox to deploy the compile source code. You can also deploy JBox in a Docker container and access the backend from another Docker container. On this page we will only be covering the different endpoints that JBox provides and how to use them in order to create your requests to the backend.

CHAPTER

TEN

ENDPOINTS AVAILABLE IN JBOX

JBox contains a set of default endpoints (more can be added) that allows applications to interact with the JBox backend.

10.1 /jbox_rest (GET)

This endpoint contains all the available functions that JBox has parsed in the backend. It's a JSON object that contains a key and a value. The key is a hash for the function. This is done to have a unique key per parsed function. The value assocaited to the key is a JSON object that has the following values.

- **description**: Description provided to a function
- **extra**: Extra information that could be used inthe future.
- **file_name**: File name containing the parsed cells from the notebooks. (Hash of the file)
- **inputs**: A string containing the input variables available. These are in the format of [var1,var2,varn]
- **name**: Name of the function.
- **original_file**: Original file name.
- **outputs**: A string containing the output variables available. These are in the format of [var1,var2,varn]
- **type**: nb_functions is the default for any function parsed.

Below is an example of what a function looks like in the endpoint.

```
{
    49986303cfb12f77250f9242b7c37a02f3381838: {
        description: "Function to print Hello World!",
        extra: "",
        file_name: "c130f572ea4736b3b38236c802df972cfeb5cd90.py",
        inputs: "[]",
        name: "me45",
        original_file: "testing.ipynb",
        outputs: "[]",
        type: "nb_function"
    }
}
```

10.2 /jbox_rest/upload_notebook (POST)

Endpoint that allows you the user to upload a notebook to the notebook directory in order to parse it in the backend. The endpoint receives a file from a multipart/form-data content header. For example you can look at Gitlab Repo

10.2.1 Output Format

The output that gets returned from the backend looks like the following. You will see both a success and a failure Below.

Successful output

```
 {
    "children":[],
    "result":{
        "logs":["filename.ipynb saved in server."],
        "output":{}
    },
    "status":"SUCCESS",
    "task_id":"",
    "traceback":"filename.ipynb saved in server."
}
```

Failure output

```
 {
    "children":[],
    "result":{
        "logs":["File with non-allowable extension. Allowed externsions are ipynb"],
        "output":{}
    },
    "status":"FAILURE",
    "task_id":"",
    "traceback":"File with non-allowable extension. Allowed externsions are ipynb"
}
```

10.3 /jbox_rest/input_upload_file (POST)

Endpoint that allows you the user to upload a file to the file directory. The endpoint receives a file from a multipart/form-data content header. For example you can look at Gitlab Repo

10.3.1 Output Format

The output that gets returned from the backend looks like the following. You will see both a success and a failure Below.

Successful output

```
{
    "children":[],
    "result":{
        "logs":["/path/on/server/filename.csv saved in server."],
        "output":{
            "dir":"/path/on/server/filename.csv"
        }
    },
    "status":"SUCCESS",
    "task_id":"",
    "traceback":"/path/on/server/filename.csv saved in server."
}
```

Failure output

```
{
    "children":[],
    "result":{
        "logs":["File with non-allowable extension. Allowed externsions are txt, jpg,␣
↪png, jpeg, pdf, ipynb, xlsx, csv, doc, xls"],
        "output":{}
    },
    "status":"FAILURE",
    "task_id":"",
    "traceback":"File with non-allowable extension. Allowed externsions are txt, jpg,␣
↪png, jpeg, pdf, ipynb, xlsx, csv, doc, xls"
}
```

10.4 /jbox_rest/execute_box (POST)

This endpoint is the one that executes the code blocks in the backend. It uses the hash ID for the function as the way to determine what to call and the inputs passed in as an array where each value in the array corresponds to each value in the inputs value of the function. Below is an example of the POST body. **Even if inputs is empty, it has to be populated.**

Example with no inputs

```
{
    "id":"49986303cfb12f77250f9242b7c37a02f3381838",
```

(continues on next page)

(continued from previous page)

```
    "inputs":[]
}
```

Example with inputs

```
{
    "id":"842220c372269eec93089c356afe50280d060a63",
    "inputs":["Hello","World!"]
}
```

10.4.1 Output Format

The output that gets returned from the backend looks like the following. You will see both a success and a failure Below.

Successful output

```
 {
    "children":[],
    "result":{
        "logs":[],
        "output":{"job_id":"2753929b-fa63-4ff1-ac90-f037deb8f3c9"}
    },
    "status":"SUCCESS",
    "task_id":"",
    "traceback":null
}
```

Failure output

```
{
    "children":[],
    "result":{
        "logs":["ID provided does not match any in the DB.json. Please try a new value.
↪"],
        "output":{}
    },
    "status":"FAILURE",
    "task_id":"",
    "traceback":"ID provided does not match any in the DB.json. Please try a new value."
}
```

10.5 /jbox_rest/task_status (POST)

This endpoint checks the status of the job running. It takes the **job_id** from the **execute_box** endpoint output. If using the celery backend you will get better feedback. This includes any print statement in the cells while its running can be seen in the logs. Also if there is an error on the backend code running it will provide info on the traceback value.

Example of request

```
{
    "job_id":"132345f1-2599-4217-9ba1-ea5c8d68c4f6"
}
```

10.5.1 Output Format

The output, if there is an output to be returned, can be found in the reult -> output. In there you will have a dictionary with the output per output variables that was available on the db.json.

```
{
    "children":[],
    "result":{
        "logs":[],
        "output":{
            "c":"hello world!"
        }
    },
    "status":"SUCCESS",
    "task_id":"d29f1eab-2602-41c3-bfbc-d3bb70a84d8f",
    "traceback":null
}
```

CHAPTER

ELEVEN

CONFIGURING JBOX BACKEND

JBox has several configuration options that can be applied to the backend. This includes, but not limited to, adding new endpoints, adding directory paths, celery configuration, etc. Below we have explanations on how to work with this features.

CHAPTER

TWELVE

ADDING NEW ENDPOINTS

In order to add new endpoints, when doing Celery or not, there is a file called **extra_endpoints.py**. This file location can be found using the bash command provided by JBox. (**jbox_paths**) This command will tell you the location for the extra endpoints file depending if doing celery or not. You can update this file in order to create new endpoint in the backend. Below is an example that can easily be added at the end of the file and that would add the endpoint to the backend.

```
@app.route(path+'jbox_rest/extra_endpoints', methods=['GET'])
def test():
    return 'Create more end points by adding them in this file.'
```

Just change the value after the **jbox_rest/**. Make sure it's unique and not in use already by JBox. You also have access to the global variables from JBox as they are imported at the beginning of the file.

```
from main import app, tasks, db, path
```

CHAPTER

THIRTEEN

ADDING CONFIGS TO FLASK

When adding a new endpoint you may want to have access to new directories created or other configurations. In order for Flask to have knowledge of the directories or configs you can add them to the file called **extra_configs.py**. This file contains two JSON objects, one for configs and one for paths. If doing paths add to **extra_file_paths**. This is important as JBox adds the required paths to reach the directory.

```
extra_file_paths = {
    "certs":"certificate"
}

extra_configs = {
    "SECRET_KEY":"123456678"
}
```

CHAPTER

FOURTEEN

CONFIGURING CELERY

To configure celery there is a file called **celeryconfig.py** and can also be found by the bash command provided by JBox. (**jbox_paths**) You can add all the configs required by celery. The configuration values avaialable to Celery can be found in the following link.