MetaCall Jupyter Kernel

Release 0.1

Harsh Bardhan Mishra

Aug 24, 2021

CONTENTS:

MetaCall Jupyter Kernel is an open-source wrapper kernel that implements cross-language function calls through the MetaCall Core and the Polyglot REPL. MetaCall Core is an open-source library that brings the polyglot programming experience to Developers. With MetaCall, developers can embed different programming languages through an easy-to-use high-level API.

The Kernel exposes the MetaCall Core API which can be loaded and launched through a Jupyter Notebook interface. With this Notebook, the users can try out writing, mixing and embedding code in different programming languages.

When a Jupyter notebook of type `metacall_kernel` is opened, the Kernel starts a new Polyglot REPL subprocess. In fact, the `metacall_kernel` makes the kernel behave like a wrapper over the Polyglot REPL subprocess where standard input is imparted and the standard output is fetched and displayed on the client interface.

To put it in a nutshell, `metacall_kernel` essentially comprises three files which has to be installed:

- **Kernel Spec**: kernel.json
- **Wrapper Kernel**: kernel.py
- **REPL subprocess**: repl.js

Cell input in the Jupyter notebook is taken by kernel.py and sent via repl.js to Polyglot REPL running through the subprocess. Output from Polyglot REPL is collected by repl.js and given back to kernel.py where it is analysed and transformed into an output format that the Jupyter notebook understands, and thus eventually displayed.

CHAPTER

ONE

INSTALLATION

When you have the MetaCall's Jupyter Kernel repository cloned and set up, you are ready to install the software and tools you will use to modify the kernel and push your changes.

1.1 What you require?

- A bash shell environment (Linux and OS X include a bash shell environment out of the box, but if you are on Windows you can use Cygwin)
- Python 3.x
- Git
- NodeJS
- A web browser (Firefox, Chrome, or Safari)
- Docker

1.2 Install the required software dependencies on a Linux system

It is recommended to use a Virtual Environment to manage your dependenices and the application build. We will first start with setting up the Local Project Environment:

```
virtualenv env
source env/bin/activate
```

Next we can download all the dependenices and setup the Kernel:

```
curl -sL https://raw.githubusercontent.com/metacall/install/master/install.sh | sh
python3 -m pip install --upgrade pip
pip3 install -r requirements.txt
python3 setup.py install
python3 -m metacall_jupyter.install
metacall npm install
```

Start your Jupyter Notebook by pushing the following command:

```
python3 -m metacall_jupyter.launcher
```

You can pick `metacall_jupyter` from the drop-down options and start working with the Jupyter Notebook interface.

1.3 Building the Kernel

With the initial setup complete, you are ready to make changes to the kernel. From the `metacall_kernel` directory, once you have made your changes, run through your changes:

```
python3 -m metacall_jupyter.install
python3 -m metacall_jupyter.launcher
```

1.4 Docker setup

Build the image:

```
docker build -t metacall/jupyter .
```

Run the image:

```
docker run --rm --network=host -it metacall/jupyter
```

1.5 Run the Tests

To run the tests, push the following command:

```
pytest test-kernel.py
```

The script will run all the tests. To generate a coverage report, we are using the pytest-cov plugin, which can be invoked by pushing the following command:

```
pytest --cov=metacall_jupyter test-kernel.py
```

1.6 Set up the documentation

To setup the Sphinx documentation on your local machine, enter into the `docs` directory and install all the local dependences:

```
cd docs
pip3 install -r requirements.txt
```

You can now build your documentation's static html assets with sphinx using `make`:

```
make html
```

After making the changes, you will be able to rebuild your documentation's html:

```
make clean && make html
```

1.7 Code Formatting

We use PyLint and Flake8 for code linting and Black for code formatting. Flake8 is used in our Continuous Integration pipeline on GitHub, and hence we would like to see zero Flake8 issues before code merge. To verify the issues raised by Flake8, just run:

```
flake8
```

To run Black against the source directory or a particular file you have edited, run:

```
black <SOURCE_DIRECTORY_OR_FILE>
```

CHAPTER

TWO

GETTING STARTED

2.1 Starting a notebook

To start a notebook, type in a terminal:

```
jupyter notebook
```

Your default webbrowser should open with a list of the files in the current directory. Choose `New->metacall_kernel` to open a new notebook of type `metacall_kernel`.

You can also start a nootbook directly from the commandline using our launcher script:

```
python3 -m metacall_jupyter.launcher
```

2.2 Closing a notebook

To close the notebook, choose:

```
File -> Close and Halt
```

from the menu.

Note that simply closing the browser tab does not close the notebook or the running MetaCall Polyglot REPL subprocess(es) behind it. You can reopen the tab by clicking on the name of your notebook (next to the then green icon).

It is also possible to kill the MetaCall REPL subprocess(es) running behind the notebook by clicking on the Shutdown button in the running notebook section of your Jupyter session.

You can optionally use the programmatic way to shutdown by pushing it in the cell and executing it:

```
$shutdown
```

It will gracefully kill the running subprocess and you can safely exit from the notebook.

2.3 Running a notebook

MetaCall Jupyter kernel supports a few commands that allows you to interact with the MetaCall Polyglot REPL to load and execute code in different languages. The other commands and magics allow you to load foreign functions on the language, interact with the shell and inspect the meta-object protocol.

You can check-out all the available functionalities using the `$help` command on the cell and execute it. You will get the following output:

```
1. ! : Run a Shell Command on the MetaCall Jupyter Kernel
2. $shutdown : Shutdown the MetaCall Jupyter Kernel
3. $inspect : Inspects the MetaCall to check all loaded functions
4. $loadfile: Loads a file onto the MetaCall which can be evaluated
5. $newfile: Creates a new file and appends the code mentioned below
6. %repl <tag>: Switch from different REPL (available tags: node, py)
7. >lang: Execute scripts using the MetaCall exec by saving them in a temporary file␣
↪(available languages: python, javascript)
8. $loadcell <tag>: Loads a function onto the MetaCall to be evaluated
9. $help: Check all the commands and tags you can use while accessing the MetaCall Kernel
10. $available: Checks all the available REPLs on the Kernel
```

You can load a REPL, by just passing `%repl <tag>` where you can replace `<tag>` with the languages available. You can check-out the available languages through the `$available` command.

2.4 Exporting the notebook

You can export a reproducible copy of the Notebook by choosing:

```
File -> Download as
```

You can download the Notebook in a Notebook (`.ipynb`), PDF, HTML, Markdown and more formats easily. Make sure to save and checkpoint to ensure a reproducible copy on your own local machine:

```
File -> Save and Checkpoint
```

CHAPTER

THREE

USAGE

Start the notebook server from the command line:

```
jupyter notebook
```

You should see the notebook open in your browser.

3.1 Create a new notebook

Chose the new `metacall_kernel` kernel by

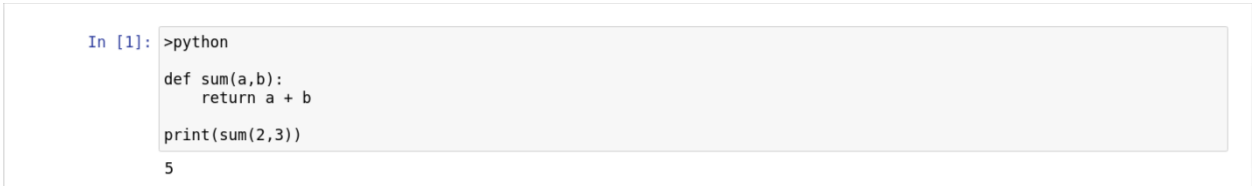
```
New -> metacall_kernel
```

The *console* window will show you kernel messages, especially it will tell you when the notebook was saved (periodically).

3.2 Send input

Send Key: `Shift-Return`

You may enter single line code followed by a Shift-Return, i.e. hold down the Shift key, then press the Return key.



There are a lot of features in Jupyter, e.g. running all cells or a print preview (rendered HTML) which allows you to save the complete page (including images, outputs and so on) by your browser.

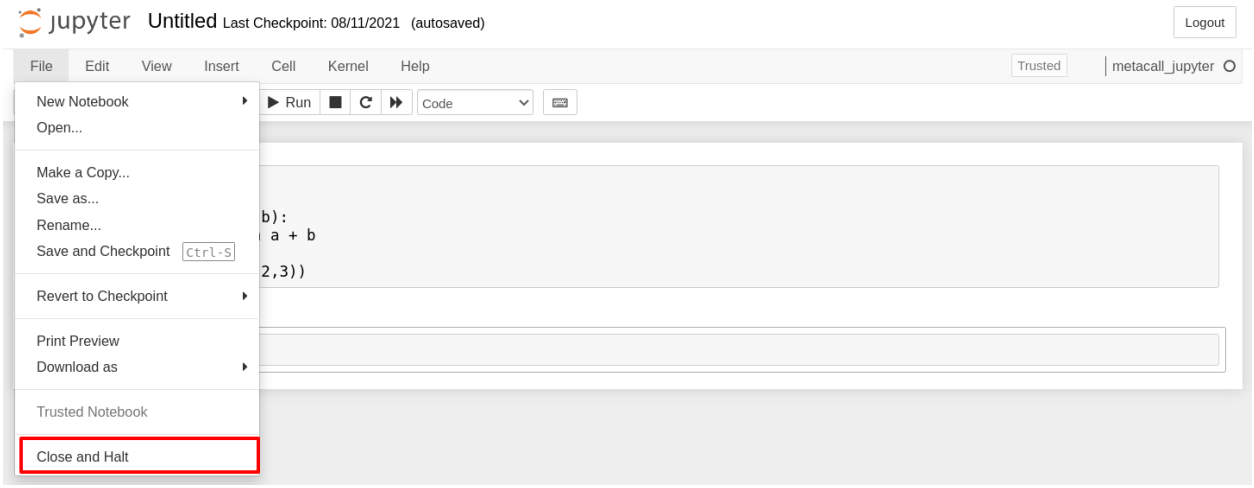
3.3 Close and Halt

The notebook may be terminated by

```
$shutdown
```

or from the menu (preferred)

```
File -> Close and Halt
```



3.4 Shell access

prefix: `!`

To run a shell command (e.g. `ls`), put a `!` in front of the command. Note that `!` must be the first character in the cell, otherwise it will not be interpreted as shell prefix character.

You can push in the following to test it out:

```
!curl -I --http2 https://www.ubuntu.com/
```

```
In [2]: !curl -I --http2 https://www.ubuntu.com/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
  0   175    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
HTTP/1.1 301 Moved Permanently
server: nginx/1.14.0 (Ubuntu)
date: Thu, 19 Aug 2021 17:11:19 GMT
content-type: text/html
content-length: 175
location: https://ubuntu.com/
link: <https://assets.ubuntu.com>; rel=preconnect; crossorigin, <https://assets.ubuntu.com>; rel=preconnect, <http
s://res.cloudinary.com>; rel=preconnect
x-cache-status: MISS from content-cache-gs2/1
```

3.5 Loading REPL

prefix: `%repl <tag>`

Under the hood, the MetaCall Jupyter Kernel interacts with the Polyglot REPL, a Read-Evaluate-Print-Loop (REPL) that allows us to interact with REPLs of multiple languages using MetaCall.

You can load a REPL, by just passing `%repl <tag>` where you can replace `<tag>` with the languages available. You can check-out the available languages through the `$available` command.

You can load the REPL via passing in `%repl <tag>` where tag can be replaced with either Python or Node. You can add support for more languages (like Ruby, Java and more) through the Polyglot REPL.

Let's load a Python REPL and execute some code.

```
In [3]: %repl py
        REPL 'py' has been selected.

In [4]: print("Hello World")
        Hello World

In [5]: a = 2
        b = 3
        print(a+b)
        5
```

Let's load a Node REPL and execute some code.

```
In [6]: %repl node
        REPL 'node' has been selected.

In [7]: function sum(a,b) {
            return a + b;
        }
        undefined

In [8]: sum(3,4);
        7
```

3.6 Executing scripts

prefix: `>lang`

You can execute standalone scripts in Python and JavaScript (Node) using the MetaCall exec through the MetaCall Jupyter Kernel. It provides a handy way for you to execute scripts, check output and validate your polyglot applications, without needing to exploit cross-language function calls.

You can execute standalone scripts through `>lang` where the available languages are `python`, `javascript` or any other language that is supported by the MetaCall CLI.

In [1]:
```
>python

def rabin_karp(pattern, text):
    p_len = len(pattern)
    p_hash = hash(pattern)

    for i in range(0, len(text) - (p_len - 1)):
        text_hash = hash(text[i:i + p_len])
        if text_hash == p_hash and \
                text[i:i + p_len] == pattern:
            return True
    return False

pattern = "ABABX"
text = "ABABZABABYABABX"
print(rabin_karp(pattern, text))
```

```
True
```

In [2]:
```
>javascript

function nthPermutation(n, digits) {
    if (digits.length === 1) {
        return digits[0] + '';
    }
    var fact = factorial(digits.length - 1);
    var i = Math.floor(n/fact);
    var first = digits.splice(i, 1);
    return first + nthPermutation(n - i*fact, digits);
};

function factorial(n){
    var fact = 1;
    while (n) {
        fact *= n;
        n--;
    }
    return fact;
};

console.log(nthPermutation(1000000 - 1, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) );
```

```
2783915460
```

3.7 Newfile magics

prefix: `$newfile <filename>`

You can create new files that are saved on the local directory through the MetaCall Kernel over the Jupyter Notebook interface. You can make use of the functionality to save files on your disk, while creating polyglot applications and implement load from files.

To make use of this functionality, you can make use of `$newfile <filename>` to create the file. If the file doesn't exist, it will create a file and append the code mentioned below to the same. If the file does exist, the code would be appended below the existing code in the file.

In [1]:
```
$newfile app.py

# Demo from: https://github.com/metacall/url-shortener-example

import requests
from urllib.parse import urlencode
import sys


def make_tiny(url):
    """Function to shorten a URL using the TinyURL API"""
    request_url = ('http://tinyurl.com/api-create.php?' +
                   urlencode({'url': url}))
    result = requests.get(request_url)
    return result.text
```

```
File app.py is saved.
```

In [2]:
```
$newfile main.js

const { make_tiny } = require('./app.py');
console.log(make_tiny("https://github.com/metacall/core"));
```

```
File main.js is saved.
```

In [3]:
```
%repl node
```

```
REPL 'node' has been selected.
```

In [4]:
```
const { make_tiny } = require('./app.py');
```

```
undefined
```

In [5]:
```
make_tiny("https://github.com/metacall/jupyter-kernel")
```

```
https://tinyurl.com/yjn96y26
```

3.8 Load and inspect

prefix: `$loadfile <tag> <filename>` **prefix**: `$inspect`

Through the MetaCall Kernel, you can save code in form of a file over the disk using the `$newfile` magic. Through the `$loadfile <tag> <filename>` functionality, you can load these external files in form of functions over the MetaCall's meta-object protocol.

Once loaded, you can import these functions in a different language, thus implementing cross-language function calls. To check whether a function has been loaded, you can use `$inspect`

In [ ]:
```
$newfile main.py

def mock_function():
    return 4
```

In [1]:
```
$loadfile py main.py
```

```
The file has been successfully loaded
```

In [2]:
```
$inspect
```


```
{"node": [{"name": "repl.js", "scope": {"name": "global_namespace", "funcs": [], "classes": [], "objects": []}}], "py": [{"n
ame": "repl.py", "scope": {"name": "global_namespace", "funcs": [{"name": "py_repl_write", "signature": {"ret": {"type": {"n
ame": "", "id": 18}}, "args": [{"name": "str", "type": {"name": "", "id": 18}}]}, "async": false}, {"name": "py_repl_close",
"signature": {"ret": {"type": {"name": "", "id": 18}}, "args": []}, "async": false}], "classes": [{"name": "InteractiveConso
le"}, {"name": "StringIO"}, {"name": "Capturing"}], "objects": []}}, {"name": "main.py", "scope": {"name": "global_namespac
e", "funcs": [{"name": "mock_function", "signature": {"ret": {"type": {"name": "", "id": 18}}, "args": []}, "async": fals
e}], "classes": [], "objects": []}}]}
```

3.9 Loadcell

prefix: `$loadcell <tag>`

Through the MetaCall Kernel, you can save load functions through code cell directly using the `$loadcell <tag>` functionality. Through this, you don't have to specify the filename and the function is loaded directly by saving it on a temporary file.

Once loaded, you can import these functions in a different language, thus implementing cross-language function calls. To check whether a function has been loaded, you can use `$inspect`.

```
In [1]: $loadcell py

        def mock():
            return 4 + 8

        The code has been successfully loaded on Kernel

In [2]: $inspect

        {"node": [{"name": "repl.js", "scope": {"name": "global_namespace", "funcs": [], "classes": [], "objects": []}}], "py": [{"n
        ame": "tmps3m6rh6t.py", "scope": {"name": "global_namespace", "funcs": [{"name": "mock", "signature": {"ret": {"type": {"nam
        e": "", "id": 18}}, "args": []}, "async": false}], "classes": [], "objects": []}}, {"name": "repl.py", "scope": {"name": "gl
        obal_namespace", "funcs": [{"name": "py_repl_write", "signature": {"ret": {"type": {"name": "", "id": 18}}, "args": [{"nam
        e": "str", "type": {"name": "", "id": 18}}]}, "async": false}, {"name": "py_repl_close", "signature": {"ret": {"type": {"nam
        e": "", "id": 18}}, "args": []}, "async": false}], "classes": [{"name": "InteractiveConsole"}, {"name": "StringIO"}, {"nam
        e": "Capturing"}], "objects": []}}]}

In [3]: %repl node

        REPL 'node' has been selected.

In [4]: require('metacall').metacall('mock')

        12
```

You can find all the Notebook examples here.

CHAPTER

FOUR

SOURCE DOCUMENTATION

4.1 kernel.py

class kernel.**metacall_jupyter**(***kwargs*)

Defines the Jupyter Kernel declaration for MetaCall Core

available_repl()

Function to check for available REPLs on the MetaCall Kernel

Parameters **None** –

Returns List of REPLs available on the Kernel

Return type available_repl

banner = 'Wrapper Kernel for MetaCall Core Library leveraging IPython and Jupyter'

byte_to_string(*code*)

Function to convert the result of the execution to string

do_execute(*code*, *silent*, *store_history=True*, *user_expressions=None*, *allow_stdin=False*)

Executes the User Code

Parameters

- **code** – The code to be executed
- **silent** – Whether to display output
- **store_history** – Whether to record this code in history and increase the execution count
- **user_expressions** – Mapping of names to expressions to evaluate after the code has run
- **allow_stdin** – Whether the frontend can provide input on request

Returns The history of the session.

Return type history (str)

do_history(*hist_access_type*, *output*, *raw*, *session=None*, *start=None*, *stop=None*, *n=None*, *pattern=None*, *unique=False*)

Get the history of a session

Parameters

- **hist_access_type** (*str*) – 'tail' or 'range'
- **output** (*bool*) – If True, then the history is printed to stdout. Otherwise, it is returned as a string.
- **raw** (*bool*) – If True, then the history is not formatted in any way.

- **session** (*str*) – The name of the session.
- **start** (*int*) – The first execution count from the history to get.
- **stop** (*int*) – The last execution count from the history to get.
- **n** (*int*) – The number of executions to get.
- **pattern** (*str*) – The pattern to search the history with.
- **unique** (*bool*) – If True, then only unique history items are shown.

Returns The history of the session.

Return type history (str)

do_shutdown(*restart*)
Shuts down the Kernel

Parameters **restart** – Boolean value to determine the kernel is shutdown or restarted

Returns Boolean value to signal the kernel shutdown

Return type restart

error_message(*code*)
Highlights the error message in red color

get_range(*session*, *start*, *stop*, *raw*, *output*)
Gets the range of history items for a session

Parameters

- **session** – session name
- **start** – start line number
- **stop** – stop line number
- **raw** – True to get the raw input
- **output** – True to get the formatted output

Returns A list of tuples (source, input, output)

Return type result

get_tail(*n*, *raw*, *output*, *include_latest*)
Gets the last n lines of history, formatted nicely

Parameters

- **n** (*int*) – The number of lines to be fetched
- **raw** (*bool*) – Whether to include raw_history
- **output** (*bool*) – Whether to include output
- **include_latest** (*bool*) – Whether to include the latest

Returns A list of lines

Return type list

help_links
An instance of a Python list.

history = {}

history_db_ready = False

implementation = 'Jupyter Kernel for MetaCall Core'

implementation_version = '0.1'

language = 'MetaCall Core'

language_info = {'file_extension': '.txt', 'mimetype': 'text/plain', 'name': 'MetaCall Core'}

language_version = '0.4.12'

metacall_execute(*code*, *extension*)
Executes the Code passed by creating a temporary file using a MetaCall Subprocess

Parameters

- **code** – Code to executed by the MetaCall subprocess
- **extension** – The extension of the code to create a temporary file from

Returns The log output generated by the subprocess after a successful execution

Return type logger_output

metacall_repl(*code*)
Function to execute the user code and return the result through MetaCall subprocess.

Parameters **code** – The code to be executed

Returns The result of the execution

Return type result

newfile_magic(*code*)
Function to save a new file using the *$newfile* magic.

Parameters **code** (*str*) – The code to be executed

record_history(*session*, *count*, *code*, *data*)
Record the user code and the result of the user code to the history database.

results = {}

shell_execute(*code*, *shcmd*)
Executes the Shell Commands using a Subprocess.

Parameters

- **code** – Shell Command to executed by the subprocess
- **shcmd** – Configuration to call Shell Commands

Returns The log output generated by the subprocess after a successful execution

Return type logger_output

start_history()
Starts the MetaCall Kernel History Database

4.2 install.py

install.**install_my_kernel_spec**(*user=True*, *prefix=None*)
Installs the Kernel Specification

Parameters

- **user** – Checks the User installation
- **prefix** – Checks for the specific prefix

Returns None

install.**main**(*argv=None*)
Creates a function to pass Argument Parser

CHAPTER

FIVE

CONTRIBUTING

Bug fixes, performance improvements, code formatting. There are a lot ways in which you can contribute! The issues list of a project is a great place to find something that you can help us with.

To increase the chances of your contribution getting merged, please ensure that:

- You satisfy our code of conduct.
- Your code follows our coding guidelines.
- Your submission follows Vincent Driessen's Git Branching System.
- Your pull request:
  - Passes all checks and has no conflicts.
  - Has a well-written title and message that briefly explains your proposed changes.

We welcome all kinds of bug reports, user feedback and feature requests! To get started with contributing for the very first time on GitHub, we have a few steps outlined for you.

5.1 Create a GitHub account

Before you can contribute to MetaCall's Jupyter Kernel, you must sign up for a GitHub account.

5.2 Set up authentication

When you have your account set up, follow the instructions to generate and set up SSH keys on GitHub for proper authentication between your workstation and GitHub.

Confirm authentication is working correctly with the following command:

```
ssh -T git@github.com
```

5.3 Fork and clone the repository

You must fork and set up the MetaCall's Jupyter Kernel repository on your workstation so that you can create PRs and contribute. These steps must only be performed during initial setup.

1. Fork the https://github.com/metacall/jupyter-kernel repository into your GitHub account from the GitHub UI. You can do this by clicking on **Fork** in the upper right-hand corner.
2. In the terminal on your workstation, change into the directory where you want to clone the forked repository.
3. Clone the forked repository onto your workstation with the following command, replacing with your actual GitHub username: `git clone git@github.com:<user_name>/jupyter-kernel.git`
4. Change into the directory for the local repository you just cloned. `cd jupyter-kernel`
5. Add an upstream pointer back to the MetaCall's remote repository, in this case `jupyter-kernel`. `git remote add upstream git@github.com:metacall/jupyter-kernel.git` This ensures that you are tracking the remote repository to keep your local repository in sync with it.

5.4 Making the changes

Follow the install instructions to setup the project locally. After you are done making the changes, make sure to run Black and Flake8 for code linting and formatting respectively. We have a pre-configured Flake8 configuration and once you run `black` against the source directory or file, run Flake8 to verify that linting checks pass:

```
flake8
```

Optionally, we would also like the Continous Integration to pass successfully and would advise for the usage of `act` for running the workflows locally. `act` is a tool offered by Nektos which provides a handy way to run GitHub Actions locally using Docker.

`act` can be set up locally with Homebrew, Chocolatey or even a simple BASH script. To set it up using the BASH script, just push the following command on your terminal:

```
curl https://raw.githubusercontent.com/nektos/act/master/install.sh | sudo bash
```

Next step is to define the custom image that we can use to run our actions locally. `act` provides a micro, medium and larger Docker image for Ubuntu GitHub runner. `act` does not support Windows and macOS images yet.

While running `act` for the first time, we can define the image that we would like to utilize for our local CI runs. The configuration is saved inside the `~/.actrc` file.

In the cloned repository, while running `act` for the first time, it will find the `./.github/workflows` and all the workflows present. To checkout the jobs listed as part of the GitHub Actions CI, push the following command:

```
act -l
```

It will list all the jobs and you can pick up the particular jobs you wish to run. If you are looking to run a particular job, push in the following command:

```
act -j <JOB_NAME>
```

To run the job in dry run, push in the following command:

```
act -n
```

To run the job with verbose logging, push in the following command:

```
act -v
```

To reuse the containers in `act` to maintain state, push in the following command:

```
act -j <JOB_NAME> --bind --reuse
```

If the workflow is running successfully, you can now be confident about your changes and be ready to send a Pull Request for the same.

5.5 Sending a Pull Request

When your work is ready and complies with the project conventions, upload your changes to your fork, by making a clean commit. Make sure that the changes being proposed are from a branch and not the `master`.

```
git push -u origin Branch_Name
```

Go to your repository on your browser and click on **Compare** and pull requests. Add a title and description to your pull request that explains your contribution. Voila! Your Pull Request has been submitted and will be reviewed by the maintainers and merged.

CHAPTER

SIX

CHANGE LOG

6.1 0.1.0

Released on August 22, 2021.

6.1.1 Added

First version of the MetaCall Jupyter kernel that follows the wrapper kernel mechanism using IPython and the Jupyter client:

- Setup the wrapper kernel using IPython and the Jupyter client.
- Integrated the Polyglot REPL into the Jupyter client and added support for Python & NodeJS.
- Added user-experience features for the kernel such as `$loadfile`, `$loadcell`, `$inspect` to load foreign modules and implement cross-language function calls. More information here.
- Added support to run Shell commands from the kernel.
- Implemented the CI/CD pipelines for the project using GitHub Actions.
- Containerized the project using Docker.
- Implemented the tests for the project using PyTest.
- Developed the documentation using Sphinx, published on ReadTheDocs here.

PYTHON MODULE INDEX

INDEX