



Developing Extensions for JupyterLab

Meet the instructors



Piyush Jain

AWS

Piyush is a software engineer working on JupyterLab



Alex Bozarth

IBM

Alex is a software engineer working on Elyra and JupyterLab



Martha Cryan

IBM

Martha is a software engineer working on Elyra and JupyterLab

Exploring Extensions

- What are extensions?
- Examples
 - DrawIO
 - Latex
 - Git
 - Spellchecker
 - Themes
- Installing prebuilt vs source



L^AT_EX



DrawIO

Anatomy of an Extension

- Extensions, plugins and widgets
- Code Walkthrough...
 - We will be using an example created from <https://github.com/jupyterlab/extension-cookiecutter-ts>
 - The example is loosely based on the in-depth example found in the jupyterlab documentation: https://jupyterlab.readthedocs.io/en/stable/extension/extension_tutorial.html

Code Exercise

1. Open `examples/tutorial_extension` dir in your IDE
2. Try adding a toolbar button that refreshes the image using the following hints:
 - The toolbar can be accessed from `MainAreaWidget.toolbar`
 - A `ToolBarButton` class can be found in `@jupyterlab/apputils`
3. An example answer can be found on the next slide if you get stuck

Code Exercise Example Answer

```
const button = new ToolbarButton({
  icon: refreshIcon,
  onClick: () => widget.load_image()
});
main.toolbar.addItem('refresh', button);
```

Debugging JupyterLab Extensions

- When is debugging useful
- Setting up for debugging
- Launching JupyterLab for debugging
- Setting breakpoints
- Other ways to debug

When is debugging useful

- To find errors in code
- Investigating unexpected results
- Understanding the code path
- Learning internals of other extensions

How to know something has gone wrong

- UI elements are missing
- Errors appear in the server log
- Errors appear in the browser console

Debugging in Visual Studio Code

- Instructions are in [DEBUGGING.md](#)
- Install the cookiecutter package
`pip install cookiecutter`
- Use the debug-config-cookiecutter
`cookiecutter ../debug-config-cookiecutter`
- Install the debug dependencies
`jlpm install`
- Build the extensions
`jlpm build`

Setting breakpoints

The screenshot shows the JupyterLab interface with a Python code editor. The code is for a Jupyter server handler. A red dot on line 13 indicates a breakpoint is set at the `self.finish` call. The breadcrumb navigation shows the current location: `tutorial_extension > handlers.py > RouteHandler > get`.

```
tutorial_extension > handlers.py > RouteHandler > get
1  import json
2
3  from jupyter_server.base.handlers import APIHandler
4  from jupyter_server.utils import url_path_join
5  import tornado
6
7  class RouteHandler(APIHandler):
8      # The following decorator should be present on all verb methods (head, get, post,
9      # patch, put, delete, options) to ensure only authorized user can request the
10     # Jupyter server
11     @tornado.web.authenticated
12     def get(self):
13         self.finish(json.dumps({
14             "data": "This is /tutorial-extension/get_example endpoint!"
15         }))
16
17
18 def setup_handlers(web_app):
19     host_pattern = ".*$"
20
21     base_url = web_app.settings["base_url"]
22     route_pattern = url_path_join(base_url, "tutorial-extension", "get_example")
23     handlers = [(route_pattern, RouteHandler)]
24     web_app.add_handlers(host_pattern, handlers)
25
```

The sidebar on the left contains several panels:

- VARIABLES**: Currently empty.
- WATCH**: Currently empty.
- CALL STACK**: Currently empty.

At the top of the interface, there are tabs for `handlers.py`, `TS handler.ts`, `TS index.ts`, and `DEBUGGING.md U`. The `handlers.py` tab is active.

Launching JupyterLab for debugging

The screenshot displays the JupyterLab interface with a Python file open for debugging. The code defines a `RouteHandler` class that inherits from `APIHandler`. The `get` method is currently selected, and a red dot indicates a breakpoint. The terminal shows a series of HTTP requests and a successful build.

```
handlers.py | tutorial_extension | handlers.py | RouteHandler
1 import json
2
3 from jupyter_server.base.handlers import APIHandler
4 from jupyter_server.utils import url_path_join
5 import tornado
6
7 class RouteHandler(APIHandler):
8     # The following decorator should be present on all verb methods (head, get, post,
9     # patch, put, delete, options) to ensure only authorized user can request the
10    # Jupyter server
11    @tornado.web.authenticated
12    def get(self):
13        self.finish(json.dumps({
14            "data": "This is /tutorial-extension/get_example endpoint!"
15        })))
```

WATCH

CALL STACK

- > jlab backend RUNNING
- jlab frontend: localh... RUNNING

BREAKPOINTS

- Caught Exceptions
- Uncaught Exceptions
- handlers.py tutorial_extensi... 13

TERMINAL

```
[W 2022-07-08 09:54:18.084 ServerApp] 404 GET /static/lab/3496.ecb0e7fcc54191234ae6.js.map?v=ec
[W 2022-07-08 09:54:18.101 ServerApp] 404 GET /static/lab/4429.c4f083ef6b6e29345fd4.js.map?v=c4
[W 2022-07-08 09:54:18.115 ServerApp] 404 GET /static/lab/4429.c4f083ef6b6e29345fd4.js.map?v=c4
[W 2022-07-08 09:54:18.139 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
[W 2022-07-08 09:54:18.151 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
[W 2022-07-08 09:54:18.192 ServerApp] 404 GET /static/lab/714.d38baae8faccca175d4b.js.map?v=d38
[W 2022-07-08 09:54:18.203 ServerApp] 404 GET /static/lab/714.d38baae8faccca175d4b.js.map?v=d38
[I 2022-07-08 09:54:24.703 LabApp] Build is up to date
```

Debugging Front End Extension

The screenshot displays the VS Code interface during a debugging session. The top bar shows the 'jlab debug' mode and several open files: handlers.py, TS handler.ts, TS widget.ts, TS index.ts, and README.md. The left sidebar contains icons for Explorer, Search, Run and Debug, and Extensions. The main editor area shows the file 'TS index.ts' with the following code:

```
40 console.log(data);
41 }
42 .catch(reason => {
43   console.error(
44     `The tutorial_extension server extension appears to be missing.\n${reason}`
45   );
46 });
47
48 app.commands.addCommand('tutorial:open', {
49   // code to run when this command is executed
50   execute: () => {
51     const widget = new TutorialWidget();
52     const main = new MainAreaWidget({ content: widget });
53     main.title.label = 'Tutorial Widget';
54     main.title.icon = jupyterIcon;
55     main.title.caption = widget.title.label;
56
57     app.shell.add(main, 'main');
58   },
59 }
```

The 'CALL STACK' pane on the left shows the following stack:

- ThreadPoolExecutor-... PAUSED
- activate src/index.ts 48:5

The 'TERMINAL' pane at the bottom shows the following log output:

```
[W 2022-07-08 10:03:19.862 ServerApp] 404 GET /static/lab/4429.c4f083ef6b6e29345fd4.js.map?v=c4
[.0.0.1] 5.70ms referer=None
[W 2022-07-08 10:03:19.878 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
[.0.0.1] 7.19ms referer=None
[W 2022-07-08 10:03:19.893 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
[.0.0.1] 6.21ms referer=None
[W 2022-07-08 10:03:20.066 ServerApp] 404 GET /static/lab/jlab_core.fc632a0f38747f007842.js.map
(127.0.0.1) 6.90ms referer=None
[W 2022-07-08 10:03:20.077 ServerApp] 404 GET /static/lab/jlab_core.fc632a0f38747f007842.js.map
(127.0.0.1) 6.37ms referer=None
```

Debugging Server Extension

The screenshot shows the JupyterLab IDE interface with the following components:

- File Explorer:** Shows the current file structure: `tutorial_extension > handlers.py > RouteHandler > get`.
- Code Editor:** Displays the `handlers.py` file with the following code:

```
1 import json
2
3 from jupyter_server.base.handlers import APIHandler
4 from jupyter_server.utils import url_path_join
5 import tornado
6
7 class RouteHandler(APIHandler):
8     # The following decorator should be present on all verb methods (head, get, post,
9     # patch, put, delete, options) to ensure only authorized user can request the
10    # Jupyter server
11    @tornado.web.authenticated
12    def get(self):
13        self.finish(json.dumps({
14            "data": "This is /tutorial-extension/get_example endpoint!"
15        })))
16
17
18 def setup_handlers(web_app):
19     host_pattern = ".*$"
```
- WATCH Panel:** Empty.
- CALL STACK Panel:** Shows the current execution path:
 - `get` in `handlers.py` at line 13:1
 - `wrapper` in `web.py` at line 3208:1
 - `_execute` in `web.py` at line 1711:1
 - `_run` in `events.py` at line 81:1
 - `_run_once` in `base_events.py`
 - `run_forever_base_event`
- BREAKPOINTS Panel:** Shows a breakpoint set at line 13 of `handlers.py` in `tutorial_extensi...`.
- TERMINAL Panel:** Displays the following log output:

```
.0.0.1) 7.39ms referer=None
[W 2022-07-08 10:02:08.469 ServerApp] 404 GET /static/lab/4429.c4f083ef6b6e29345fd4.js.map?v=c4
.0.0.1) 7.72ms referer=None
[W 2022-07-08 10:02:08.488 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
.0.0.1) 7.98ms referer=None
[W 2022-07-08 10:02:08.502 ServerApp] 404 GET /static/lab/7796.53c158c42e7f9697953b.js.map?v=53
.0.0.1) 7.64ms referer=None
[W 2022-07-08 10:02:08.663 ServerApp] 404 GET /static/lab/jlab_core.fc632a0f38747f007842.js.map
(127.0.0.1) 5.43ms referer=None
[W 2022-07-08 10:02:08.674 ServerApp] 404 GET /static/lab/jlab_core.fc632a0f38747f007842.js.map
(127.0.0.1) 5.38ms referer=None
```

Other ways to debug

- Front end extension
 - Use the browser directly to debug
- Server extensions
 - Python's command line debugger (pdb)

```
import pdb; pdb.set_trace()
```

- IPython pdb, a better alternative to pdb (pip install ipdb)

```
import ipdb; ipdb.set_trace()
```

Working on Your Own Extension

- Jupyter Server extension
- Theme extension
- Whatever you wanted to start on - or pick from [here](#)