JupyterLab Documentation

Release 2.0.0a1

Project Jupyter

Nov 09, 2019

GETTING STARTED

1 Overview 3
2 Installation 5
3 Starting JupyterLab 7
4 Reporting an issue 9
5 Frequently Asked Questions (FAQ) 11
6 JupyterLab Changelog 13
7 The JupyterLab Interface 35
8 JupyterLab URLs 41
9 Working with Files 45
10 Text Editor 49
11 Notebooks 51
12 Code Consoles 53
13 Terminals 55
14 Managing Kernels and Terminals 57
15 Command Palette 59
16 Documents and Kernels 61
17 File and Output Formats 63
18 Extensions 69
19 JupyterLab on JupyterHub 79
20 Exporting Notebooks 81
21 General Codebase Orientation 83
22 Extension Developer Guide 87

23 Common Extension Points 99

24 Contributing to JupyterLab 107

25 Documents 111

26 Notebook 115

27 Design Patterns 121

28 CSS Patterns 125

29 React 131

30 Examples 133

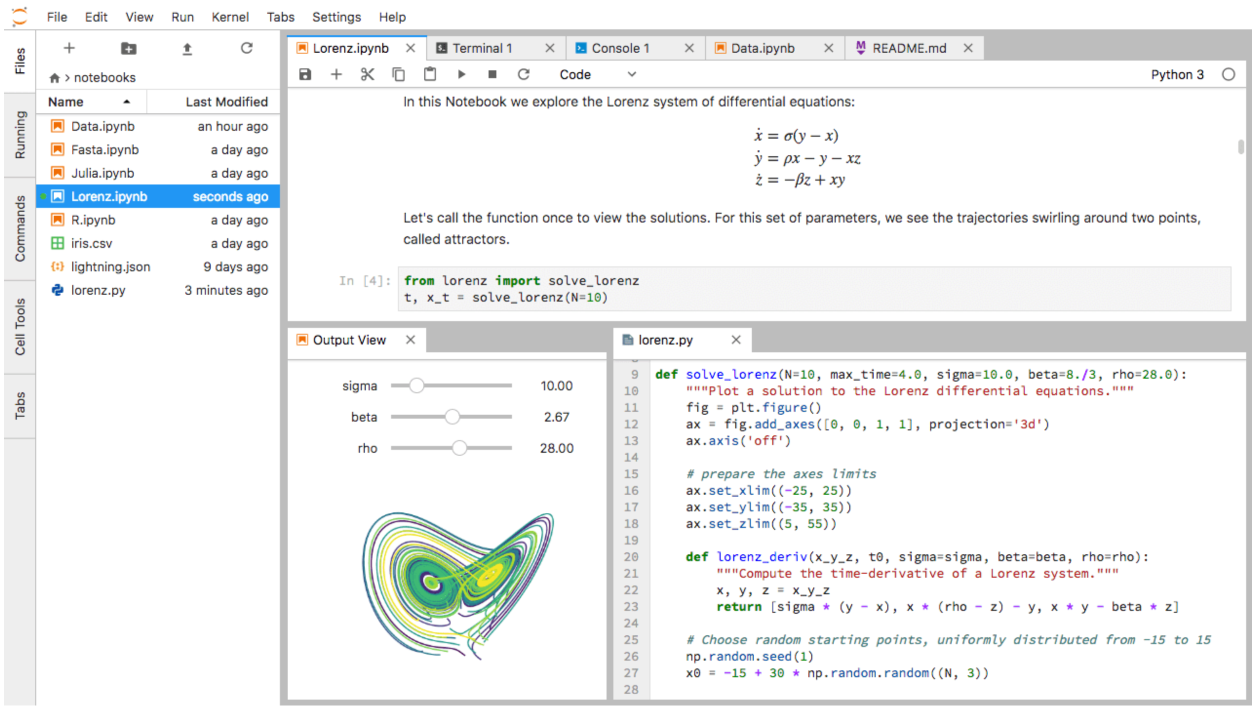
31 User Interface Helpers 137

32 Terminology 141

33 Let’s Make an Astronomy Picture of the Day JupyterLab Extension 143

34 Indices and Tables 159

JupyterLab is the next-generation web-based user interface for Project Jupyter. Try it on Binder. JupyterLab follows the Jupyter Community Guides.



Name	Last Modified
Data.ipynb	an hour ago
Fasta.ipynb	a day ago
Julia.ipynb	a day ago
Lorenz.ipynb	seconds ago
R.ipynb	a day ago
iris.csv	a day ago
lightning.json	9 days ago
lorenz.py	3 minutes ago

In this Notebook we explore the Lorenz system of differential equations:

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= \rho x - y - xz \\ \dot{z} &= -\beta z + xy \end{aligned}$$

Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors.

```
In [4]: from lorenz import solve_lorenz
        t, x_t = solve_lorenz(N=10)
```

sigma	10.00
beta	2.67
rho	28.00

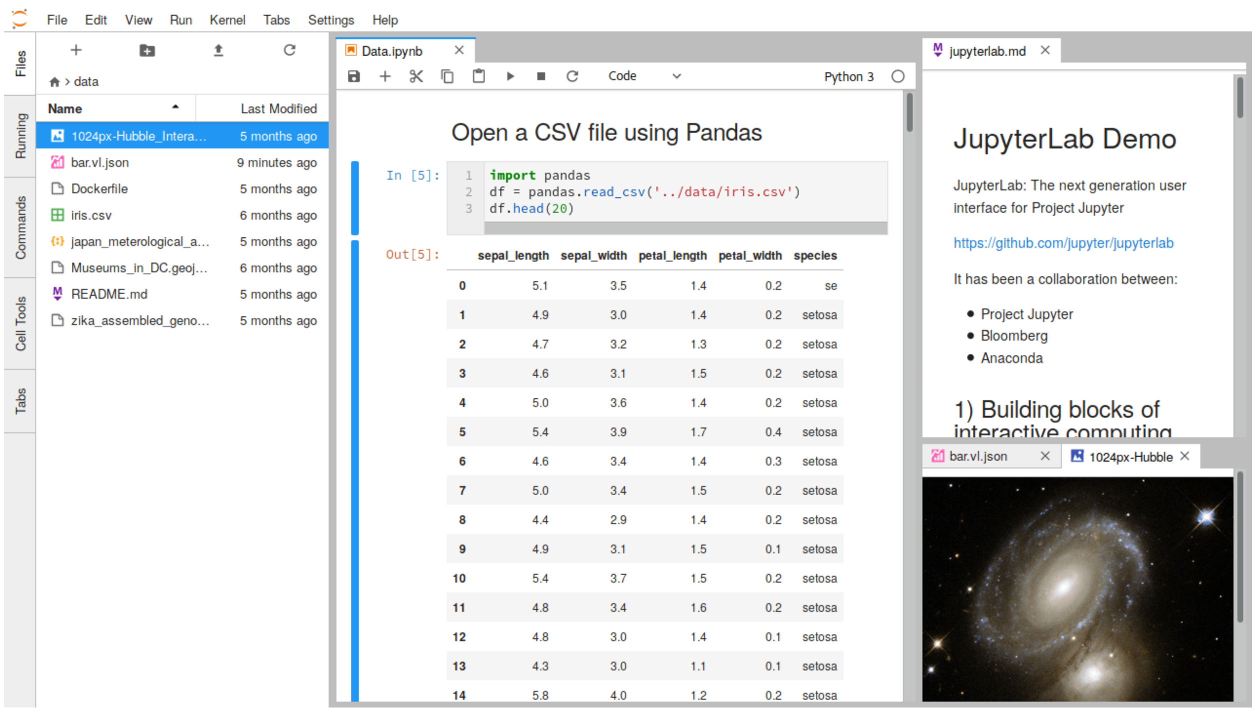
```
def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
    """Plot a solution to the Lorenz differential equations."""
    fig = plt.figure()
    ax = fig.add_axes([0, 0, 1, 1], projection='3d')
    ax.axis('off')

    # prepare the axes limits
    ax.set_xlim((-25, 25))
    ax.set_ylim((-35, 35))
    ax.set_zlim((5, 55))

    def lorenz_deriv(x_y_z, t0, sigma=sigma, beta=beta, rho=rho):
        """Compute the time-derivative of a Lorenz system."""
        x, y, z = x_y_z
        return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]

    # Choose random starting points, uniformly distributed from -15 to 15
    np.random.seed(1)
    x0 = -15 + 30 * np.random.random((N, 3))

```

CHAPTER

ONE

OVERVIEW

JupyterLab is a next-generation web-based user interface for Project Jupyter.



Name	Last Modified
1024px-Hubble_Intera…	5 months ago
bar.vl.json	9 minutes ago
Dockerfile	5 months ago
iris.csv	6 months ago
japan_meterological_a…	5 months ago
Museums_in_DC.geoj…	6 months ago
README.md	5 months ago
zika_assembled_geno…	5 months ago

Open a CSV file using Pandas

In [5]:
```
import pandas
df = pandas.read_csv('../data/iris.csv')
df.head(20)
```

Out[5]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	se
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.3	3.0	1.1	0.1	setosa
14	5.8	4.0	1.2	0.2	setosa

JupyterLab Demo

JupyterLab: The next generation user interface for Project Jupyter

https://github.com/jupyter/jupyterlab

It has been a collaboration between:

- Project Jupyter
- Bloomberg
- Anaconda

1) Building blocks of interactive computing

JupyterLab enables you to work with documents and activities such as *Jupyter notebooks*, text editors, terminals, and custom components in a flexible, integrated, and extensible manner. You can *arrange* multiple documents and activities side by side in the work area using tabs and splitters. Documents and activities integrate with each other, enabling new workflows for interactive computing, for example:

- *Code Consoles* provide transient scratchpads for running code interactively, with full support for rich output. A code console can be linked to a notebook kernel as a computation log from the notebook, for example.
- *Kernel-backed documents* enable code in any text file (Markdown, Python, R, LaTeX, etc.) to be run interactively in any Jupyter kernel.
- Notebook cell outputs can be *mirrored into their own tab*, side by side with the notebook, enabling simple dashboards with interactive controls backed by a kernel.
- Multiple views of documents with different editors or viewers enable live editing of documents reflected in other viewers. For example, it is easy to have live preview of *Markdown*, *Delimiter-separated Values*, or *Vega/Vega-Lite* documents.

JupyterLab also offers a unified model for viewing and handling data formats. JupyterLab understands many file formats (images, CSV, JSON, Markdown, PDF, Vega, Vega-Lite, etc.) and can also display rich kernel output in these formats. See *File and Output Formats* for more information.

To navigate the user interface, JupyterLab offers *customizable keyboard shortcuts* and the ability to use *key maps* from vim, emacs, and Sublime Text in the text editor.

JupyterLab *extensions* can customize or enhance any part of JupyterLab, including new themes, file editors, and custom components.

JupyterLab is served from the same server and uses the same notebook document format as the classic Jupyter Notebook.

1.1 JupyterLab Releases

Since JupyterLab 0.32 (February 2018), the releases of JupyterLab are suitable for general daily use by both Jupyter novices and users experienced with the Classic Notebook interface. As of the 1.0 release (June 2019), it is additionally ready for extension writers who wish to further customize the JupyterLab experience for others. Please review the *JupyterLab Changelog* for detailed descriptions of each release.

The extension developer API is evolving, and we also are currently iterating on UI/UX improvements. We appreciate feedback on our GitHub issues page as we evolve towards a stable extension development API.

JupyterLab will eventually replace the classic Jupyter Notebook. Throughout this transition, the same notebook document format will be supported by both the classic Notebook and JupyterLab.

CHAPTER

TWO

INSTALLATION

JupyterLab can be installed using `conda`, `pip`, `pipenv` or `docker`.

2.1 conda

If you use `conda`, you can install it with:

```
conda install -c conda-forge jupyterlab
```

2.2 pip

If you use `pip`, you can install it with:

```
pip install jupyterlab
```

If installing using `pip install --user`, you must add the user-level `bin` directory to your `PATH` environment variable in order to launch `jupyter lab`.

2.3 pipenv

If you use `pipenv`, you can install it as:

```
pipenv install jupyterlab
 pipenv shell
```

or from a git checkout:

```
pipenv install git+git://github.com/jupyterlab/jupyterlab.git#egg=jupyterlab
 pipenv shell
```

When using `pipenv`, in order to launch `jupyter lab`, you must activate the project's virtualenv. For example, in the directory where `pipenv`'s `Pipfile` and `Pipfile.lock` live (i.e., where you ran the above commands):

```
pipenv shell
 jupyter lab
```

2.4 Docker

If you have Docker installed, you can install and use JupyterLab by selecting one of the many ready-to-run Docker images maintained by the Jupyter Team. Follow the instructions in the Quick Start Guide to deploy the chosen Docker image. NOTE: Ensure your docker command includes the *-e JUPYTER_ENABLE_LAB=yes* flag to ensure JupyterLab is enabled in your container.

2.5 Installing with Previous Versions of Notebook

If you are using a version of Jupyter Notebook earlier than 5.3, then you must also run the following command to enable the JupyterLab server extension:

```
jupyter serverextension enable --py jupyterlab --sys-prefix
```

2.6 Prerequisites

JupyterLab requires the Jupyter Notebook version 4.3 or later. To check the version of the `notebook` package that you have installed:

```
jupyter notebook --version
```

2.7 Supported browsers

The latest versions of the following browsers are currently known to work:

- Firefox
- Chrome
- Safari

Earlier browser versions may also work, but come with no guarantees.

JupyterLab uses CSS Variables for styling, which is one reason for the minimum versions listed above. IE 11+ or Edge 14 do not support CSS Variables, and are not directly supported at this time. A tool like postcss can be used to convert the CSS files in the `jupyterlab/build` directory manually if desired.

CHAPTER

THREE

STARTING JUPYTERLAB

Start JupyterLab using:

```
jupyter lab
```

JupyterLab will open automatically in your browser.

You may access JupyterLab by entering the notebook server's *URL* into the browser. JupyterLab sessions always reside in a *workspace*. The default workspace is the main `/lab` URL:

```
http(s)://<server:port>/<lab-location>/lab
```

Because JupyterLab is a server extension of the classic Jupyter Notebook server, you can launch JupyterLab by calling `jupyter notebook` and visiting the `/lab` URL.

Like the classic notebook, JupyterLab provides a way for users to copy URLs that *open a specific notebook or file*. Additionally, JupyterLab URLs are an advanced part of the user interface that allows for managing *workspaces*. To learn more about URLs in Jupyterlab, visit *JupyterLab URLs*.

To open the classic Notebook from JupyterLab, select "Launch Classic Notebook" from the JupyterLab Help menu, or you can change the URL from `/lab` to `/tree`.

JupyterLab has the same security model as the classic Jupyter Notebook; for more information see the security section of the classic Notebook's documentation.

CHAPTER

FOUR

REPORTING AN ISSUE

Thank you for providing feedback about JupyterLab.

4.1 Diagnosing an Issue

If you find a problem in JupyterLab, please follow the steps below to diagnose and report the issue. Following these steps helps you diagnose if the problem is likely from JupyterLab or from a different project.

1. Try to reproduce the issue in a new environment with the latest official JupyterLab installed and no extra packages.

   If you are using conda:

   1. create a new environment:

      ```
      conda create -n jlab-test --override-channels --strict-channel-priority -c␣
      ↪conda-forge -c anaconda jupyterlab
      ```

   2. Activate the environment:

      ```
      conda activate jlab-test
      ```

   3. Start JupyterLab:

      ```
      jupyter lab
      ```

   - I cannot reproduce this issue in a clean environment: The problem is probably not in JupyterLab itself. Go to step 2.
   - I can reproduce this issue in a clean environment: Go to step 3.

2. Perhaps the issue is in one of the JupyterLab extensions you had installed. Install any JupyterLab extensions you had one at a time, checking for the issue after each one.

   - I can reproduce the issue after installing a particular extension: That extension may be causing the problem. File an issue with that extension's issue tracker. Be sure to mention what you have done here to narrow the problem down.
   - I cannot reproduce the issue after installing all my extensions: Good news! Likely all you have to do is update your JupyterLab and extensions. If that fixes the issue, great! If it doesn't fix the issue, you may have a more complicated issue. Go directly to *Creating an issue*.

3. Try to reproduce the issue in the classic Jupyter Notebook. Launch the classic notebook from the JupyterLab help menu to ensure you are getting exactly the same notebook server that JupyterLab is using.

   - I can reproduce the issue with the classic Jupyter Notebook: The problem is probably not from JupyterLab. It may be in the Jupyter Notebook server, your kernel, etc. Use your best judgement to file an issue with the appropriate project.
   - I cannot reproduce the issue in classic Jupyter Notebook: Go to step 4.

4. Try to reproduce the issue in your browser incognito or private browsing mode. Running in private browser mode ensures your browser state is clean.
   - I cannot reproduce the issue in private browsing mode: Perhaps resetting your cookies or other browser state would help.
   - I can reproduce the issue in private browsing mode: Go to *Creating an issue*.

You might also check your system for:

- Security software that might be preventing access to files or network interfaces
- Network equipment, routers, or proxies that might be preventing communication between the browser and the server
- Browser extensions that might be changing the JupyterLab code or application page

4.2 Creating an issue

Before creating an issue, search in the issue tracker for relevant issues. If you find an issue describing your problem, comment there with the following information instead of creating a new issue. If you find a relevant resolved issue (closed and locked for discussion), create a new issue and reference the resolved issue.

To create an issue, collect the following contextual information:

- relevant package versions, including:
  - `jupyterlab` and `notebook` versions
  - browser versions affected (please try to reproduce in Chrome and Firefox at least)
  - operating system and version
- relevant server and JavaScript error messages
- screenshots or short screencasts illustrating the issue

Create a new issue. Include the contextual information from above. Describe how you followed the diagnosis steps above to conclude this was a JupyterLab issue.

Communication in JupyterLab follows the Jupyter Community Guides.

CHAPTER

FIVE

FREQUENTLY ASKED QUESTIONS (FAQ)

Below are some frequently asked questions. Click on a question to be directed to relevant information in our documentation or our GitHub repo.

5.1 General

- *What is JupyterLab?*
- *Is JupyterLab ready to use?*
- *What will happen to the classic Jupyter Notebook?*
- Where is the official online documentation for JupyterLab?
- *How can you report a bug or issue?*

5.2 Development

- How can you contribute?
- *How can you extend or customize JupyterLab?*
- In the classic Notebook, I could use custom Javascript outputed by a cell to programatically control the Notebook. Can I do the same thing in JupyterLab?

  JupyterLab was built to support a wide variety of extensibility, including dynamic behavior based on notebook outputs. To access this extensibility, you should write a custom JupyterLab extension. If you would like trigger some behavior in response to the user executing some code in a notebook, you can output a custom mimetype (*Mime Renderer Extensions*). We currently don't allow access to the JupyterLab API from the Javsacript renderer, because this would tie the kernel and the notebook output to JupyterLab and make it hard for other frontends to support it. If you have comments or suggestions on changes here, please comment on this issue.

5.3 Tips and Tricks

- How do I start JupyterLab with a clean workspace every time?

Add '*c.NotebookApp.default_url = '/lab?reset'* to your *jupyter_notebook_config.py*. See [How to create a jupyter_notebook_config.py](https://jupyter-notebook.readthedocs.io/en/stable/config.html) for more information.

CHAPTER

SIX

JUPYTERLAB CHANGELOG

6.1 v1.2.0

6.1.1 October 29, 2019

Here are some highlights for this release. See the JupyterLab 1.2.0 milestone on GitHub for the full list of pull requests and issues closed.

6.1.2 User-facing changes

- Select cells from the current cell to the top of the notebook with `Shift Home`, to the bottom of the notebook with `Shift End` (#7336, #6783)
- Add a log console extension to display unhandled messages and other activity (#7318, #7319, #7379, #7399, #7406, #7421)
- Allow the npm `max-old-space` option to be specified outside of JupyterLab (#7317)
- Only display node structure in a JSON tree view for arrays and empty objects (#7261)
- Make much smaller distribution packages by not building JavaScript source maps for releases. (#7150)
- Add support for pasting cell attachments and dragging attachments from the file browser (#5913, #5744)
- Add a new `registry` configuration parameter to override the default yarn repository when building (#7363, #7109)

6.1.3 For developers

- Update the Markdown renderer (`marked`) to 0.7.0 (#7328)
- Remove datagrid as a singleton, allowing extensions to use newer versions (#7312)
- Add metadata to the kernelspec information (#7234)
- Allow different mimetypes for the clipboard data (#7233)
- Add inline svg icon support to toolbar buttons (#7232)
- Add PageConfig functions to query if a plugin is deferred or disabled (#7216)
- Allow for renderers for nbformat.ierror to be created (#7203, #7193)
- Refactor `fileeditor-extension` for modularization (#6904)
- Add execution timing to cells (#6864, #3320)

6.1.4 Bugfixes

- Fix the `file-browser-path` query parameter (#7313)
- Skip custom click behavior on links when the download attribute is set (#7323)
- Fix opening multiple browser tabs in Safari (#7322)
- Fix context menus on SVG icons (#7263)
- Fix overwriting of target attribute of anchors rendered by `IPython.display` (#7231)
- Fix multi-cursor backspacing (#7205, #7401, #7413)
- Fix mult-cursor cell splitting (#7207, #7417, #7419)

6.2 v1.1.0

6.2.1 August 28, 2019

Here are some highlights of what is in this release. See the JupyterLab 1.1.0 milestone on GitHub for the full list of pull requests and issues closed.

6.2.2 User-facing changes

- `jupyter lab build` now has a `--minimize=False` option to build without minimization to conserve memory and time (#6907)
- Fix workspace reset functionality (#7106, #7105)
- Restore behavior of the "raises-exception" cell tag (#7020, #7015)
- Add settings to override theme font sizes (#6926)
- Accept query parameter to optionally change file browser location (#6875)
- Pressing escape in the console should switch out of edit mode (#6822)
- Fix file browser downloads in Google Chrome (#6686)
- Make it possible to override the default widgets to view a file (#6813, #4048)
- Support installing multiple versions of the same extension (#6857)
- Support JupyterHub server name for JupyterHub 1.0 (#6931)
- Add docs to help users diagnose issues before creating them (#6971)
- The JupyterLab conda-forge package is now a *noarch* package. If you are using JupyterLab with *notebook* version 5.2 or earlier, you may need to manually enable the JupyterLab server extension. See the issue for more details (#7042)

6.2.3 For developers

- Expose install_kernel for tests so that outside projects can better use the testing framework (#7089)
- Fix `comm_info_request` content to conform to the Jupyter message specification in a backwards-compatible way (#6949, #6947)
- Add yarn package resolution to build to constrain core package versions to patch semver ranges (#6938)

- Make handling comm messages optional in a kernel connection. (#6929)
- Expose icon svg to theme css (#6034, #7027)
- Expose convenience functions for open dialogs (#6366, #6365)
- Add debug messages to possible kernel messages (#6704)
- Add server side coreconfig object (#6991)

6.2.4 Bug fixes

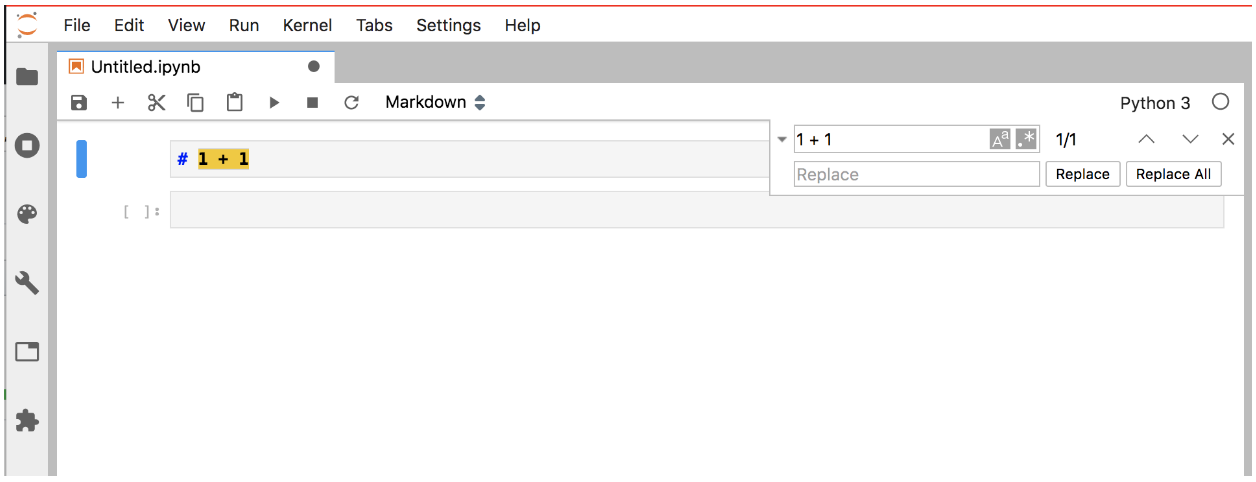
- Handle errors that occur during kernel selection (#7094)
- Fix escaping issues for page config and other template variables (#7016, #7024, #7061, #7058, #6858)
- Require jinja2 2.10+ to fix escaping issues (#7055, #7053)
- Increase the search debounce from 100ms to 500ms to increase incremental search responsiveness in large documents (#7034)
- Fix vega downloads and download urls in general (#7022, #7017, #7098, #7047)
- Do not complain in the build about duplicate or optional packages (#7013)
- Fix contextual help layout for R help (#6933, #6935)

6.3 v1.0.0

6.3.1 June 28, 2019

See the JupyterLab 1.0.0 milestone on GitHub for the full list of pull requests and issues closed in 1.0.0, and other 1.0.x milestones for bugs fixed in patch releases.

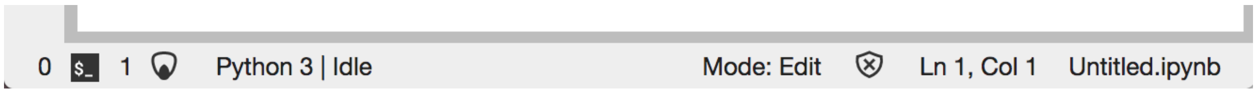
6.3.2 Find and Replace



```
# 1 + 1
```

We have added first class support for find and replace across JupyterLab. It is currently supported in notebooks and text files and is extensible for other widgets who wish to support it. (#6350, #6322, #6301, #6282, #6256, #6241, #6237, #6159, #6081, #6155, #6094, #6024, #5937, #5795, #1074)

6.3.3 Status Bar



We have integrated the JupyterLab Status Bar package package into the core distribution. Extensions can add their own status to it as well (#5577, #5525 #5990, #5982, #5514, #5508, #5352).

6.3.4 JupyterHub Integration

- We now include the JupyterHub extension in core JupyterLab, so you no longer need to install `@jupyterlab/hub-extension`. (#6451, #6428)
- JupyterLab now has a File > Logout menu entry when running with JupyterHub (#6087, #5966)

6.3.5 Printing

We now have a printing system that allows extensions to customize how documents and activities are printed. (#5850, #1314)

6.3.6 Other User Facing Changes

- The launcher displays longer kernel names and supports keyboard navigation (#6587)
- Notebook outputs without any valid MimeType renderers will not be displayed, instead of displaying an error (#6559, #6216)
- Add tooltip to file browser root breadcrumb icon showing the server root, if it is available (#6552)
- Downloading a file will no longer open a new browser window (#6546)
- Rename the help "Inspector" to "Contextual Help" and move it to the "Help" menu (#6493, #6488, #6678, #6671)
- Update many of the icons to make them more consistent (#6672, #6618, #6664, #6621)
- Update the settings UI to remove the table view (#6654, #6622, #6653, #6623, #6646, #6642)
- Replace FAQ Extension with link to JupyterLab documentation (#6628, #6608, #6625, #6610)
- Change the default keyboard shortcut for closing a tab to be `Alt+w` instead of `Cmd/Ctrl+w` to avoid conflicts with operating systems. (#6486, #6357)
- Show help text in Inspector window to describe you should select a function (#6476)
- Fixes SVG rendering (#6469, #6295)
- Add support for dropping a tab in the tab bar area. (#6454, #5406)
- Switch some default shortcuts to use `Accel` instead of `Ctrl` so they are more natural for Mac users (#6447, #5023)
- Add ability to tell between hover and selected command palette items (#6407, #279)
- Hide the "Last Modified" column when the file browser is narrow (#6406, #6093)
- Support copy/paste in terminal and Mac OS using `Ctrl+C` and `Ctrl+V` (#6391, #6385, #1146)
- Support scrolling in running kernels panel (#6383, #6371)

- Adds ability to "Merge Selected Cells" in the context menu in the notebook (#6375, #6318)
- Turn On Accessibility In Xterm.js to make it more compatible for screen readers (#6359)
- When selecting cells using the keyboard shortcuts, we now skip collapsed cells (#6356, #3233)
- Supporting opening `.geojson` files in JSON viewer (#6349)
- Performance fixes for text-based progress bars (#6304, #4202)
- Add support for rendering Vega 5 and Vega Lite 3 while keeping the existing Vega 4 and Vega Lite 2 renderers (#6294, #6133, #6128, #6689, #6685, #6684, #6675, #6591, #6572)
- Drag and drop console cells into a notebook or text editor (#5585, #4847)
- Drag and drop notebook cells into a console or text editor (#5571, #3732)
- The extension manager search now sorts extensions by the score assigned to them by NPM instead of alphabetically (#5649)
- Notify the user when a kernel is automatically restarted, for example, if crashes from an out of memory error (#6246, #4273)
- Expose the extension manager in a command and menu item (#6200)
- Add command to render all Markdown cells (#6029, #6017)
- Supports using shift to select text in output area (#6015, #4800)
- Output areas that opened in new views are restored properly now on reload (#5981, #5976)
- Add support for managing notebook metadata under a new "Advanced Tools" section in the cell tools area. The cell and notebook metadata now always reflect the current state of the notebook (#5968, #5200)
- Inherit terminal theme from core theme (#5964)
- Adds a built-in HTML viewer so that you can view HTML files (#5962, #5855, #2369)
- New workspaces are now automatically generated when you create a new window with the same workspace name. (#5950, #5854, #5830, #5214)
- We now add a hint to the context menu to describe how you can access the native browser menu (#5940, #4023)
- The tabs on the left panel have changed to make them more understandable (#5920, #5269)
- Start a new terminal when the page is refreshed and the old terminal has died (#5917)
- Add a command to open the main menus, which can be assigned to a keyboard shortcut to open and navigate menus without a mouse (#5910, #3074)
- The contextual help now updates based on changes in the cursor from the mouse instead of just from the keyboard (#5906, #5899)
- The launcher now updates when the kernels change on the server (#5904, #5676)
- Retain cell auto scroll behavior even when a cell output is cleared (#5817, #4028)
- If you link to a relative path that is not a file in a markdown cell, this will now be preserved instead of changing it to a file URL (#5814)
- Adds the ability to link to a certain row in a CSV file and have the viewer open to that row (#5727, #5720)
- We have improved the performance of switching to a large notebook (#5700, #4292, #2639)
- The vdom extension now supports event handling, so that you can have kernel code run in response to user interaction with the UI (#5670)

- Adds the ability to run "Run All Code" and "Restart Kernel and Run All Code" in code and markdown files (#5641, #5579)
- We now remember what line ending a text file has when loading it, so that files with `CRLF` line endings will properly be saved with the same endings (#5622, #4464, #3901, #3706)
- Fixes rendering of SVG elements in HTML MimeType output (#5610, #5610, #5589)
- Allow copying files by holding down `Ctrl` when dragging them in the file browser (#5584, #3235)
- Switch the hover modified time in the file browser to use the local format (#5567)
- We have added a default keyboard shortcut of `Ctrl Shift Q` for closing and cleaning up a file (#5534, #4390)
- Adds the ability to find and go to a certain line in the CSV viewer (#5523)
- Add the ability to create new text and markdown files from the launcher and command palette (#5512, #5511)
- A "New Folder" option has been added to the file browser context menu (#5447)
- The ANSI colors are now the same as those in the classic notebook (#5336, #3773)
- Send complete statements instead of current lines when stepping through code in a cell (#6515, #6063)
- Description list styles (`dl`, `dt`, `dd`) are improved to be consistent with the nteract project (#5682, #2399)

6.3.7 Settings

- The settings system has been rewritten (#5470, #5298) and now uses json5 as the syntax, which supports comments and other features for better human readability (#6343, #6199).
- The keyboard shortcut system has been rewritten and now displays a list of system commands in the settings comments (#5812, #5562).

There are new settings for many following items, including:

- Adds an option to shut down terminals and notebook kernels when they are closed (#6285, #6275)
- Scrolling past the end of a notebooks and text editor document (#5542, #5271, #5652, #4429)
- Text editor code folding, rulers, and active line highlighting (#5761, #4083, #5750, #4179, #5529, #5528)
- Markdown viewer options (#5901, #3940)
- Terminal scrollback and other settings (#5609, #3985)
- The autosave interval (#5645, #5619)
- The file browser showing the current active file (#5698, #4258)
- Custom scrollbar styling for dark themes (#6026, #4867)

6.3.8 Command Line Changes

- Installing extensions will be quieter and adds a `--debug` to extension installing (#6567, #6499, #5986)
- We now support running JupyterLab when its application directory is a symlink (#6240, #6166)
- Add `--all` flag to `labextension uninstall` to remove all extensions (#6058, #6006)
- Adds the ability to override the base URLs from the config (#5518, #5503)
- Updates to workspaces CLI command (#6473, #5977, #6276, #6234, #6210, #5975, #5695, #5694)

6.3.9 Extension Development Changes

- We have rewritten how extensions provide keyboard shortcuts and interact with the settings system. If you previously defined keyboard shortcuts or used the settings mechanism, you will need to update your extension (#5470, #5298)
- We have renamed the plugin type from `JupyterLabPlugin` to `JupyterFrontEndPlugin`. The application arg is also renamed from `JupyterLab` to `JupyterFrontEnd` and some its functionality has been moved to a separate `ILabShell` plugin (#5845, #5919)
- The lab shell `addToMainArea`, `addToLeftArea`, `addToTopArea`, `addToRightArea`, and `addToBottomArea` functions have been replaced with a single `add()` function that takes the area as an argument. Replace `addToMainArea(widget, options)` with `add(widget, 'main', options)`, etc. (#5845)
- Rename `pageUrl` to `appUrl` in the server connection (#6509, #6508, #6585, #6584)
- `MainAreaWidget` instances now forward update requests to their `content` (#6586, #6571)
- The theme data attributes are renamed and moved to the document body element. If you are relying on these attributes in CSS to conditionally style based on the theme, you should update their names. For example `data-theme-light` is now `data-jp-theme-light`. (#6566, #6554)
- Extensions which require CSS should no longer import their CSS files into their Javascript files. Instead, they should specify a root CSS file in the `style` attribute in their `package.json`, and JupyterLab will automatically import that CSS file. (#6533, #6530, #6395, #6390)
- `Dialog.prompt` has been replaced by a number of type-specific dialogs such as `InputDialog.getString`, `InputDialog.getBoolean`, etc. (#6522, #6378, #6327, #6326)
- When a `RenderMime` widget is re-rendered, the default behavior is to remove any existing content in the DOM. This can be overridden if needed. (#6513, #6505, #6497)
- We have updated our internal TypeScript version to 3.5.1 and our compile target to `ES2017`. Extensions may need to upgrade their TypeScript version and target as well. (#6440, #6224)
- We have updated the typings for some of the Kernel messages so that they better match the spec. (#6433)
- A `connectionFailure` signal has been added to some of the manager classes, which can be used to detect when a connection to the server is lost (#6399, #6176, #3324)
- Add rate limiting and polling utilities to `coreutils` to use for throttling and debouncing of API requests (#6345, #6346, #6401, #6305, #6157, #6192, #6186, #6141, #3929, #6141, #3929, #6186, #6192, #6401 ,'#6305 <https://github.com/jupyterlab/jupyterlab/pull/6305>'__, #6157)
- Require session when instantiating terminal widget (#6339, #5061)
- Provides a signal to see what items are opened in a directory listing (#6270, #6269)
- Ads the ability to add widget above the main work area to a top header area (#5936)
- Renames `contextMenuFirst` to `contextMenuHitTest` in the `JupyterFrontEnd` (#5932)
- Removes the `initialCommand` arg from the terminal creation command. (#5916)
- Adds `--jp-code-cursor-width0`, `--jp-code-cursor-width1`, and `--jp-code-cursor-width2` variables to the themes to support changing the cursor width if you change the font size (#5898)
- Adds the ability to insert a new item to the toolbar before or after another item (#5896, #5894)
- Adds the ability for extensions to register new CodeMirror modes (#5829)

- We have removed the `JUPYTERLAB_xxx_LOADER` Webpack loaders, instead you should use the loader directly in the URL as Webpack supports it (#5709, #4406)
- Adds the ability to handle fragments for document widgets (#5630, #5599)
- We have added a `@jupyterlab/ui-components` package that contains reusable React components to be used internally and in extensions. Feel free to use this to create extension UIs with consistent styles (#5538)
- The `showErrorMessage` function now lets you customize the buttons it uses (#5513)
- We now provide helpers for using React components within JupyterLab. If you were previously using `ReactElementWidget` you should switch to using `ReactWidget`. (#5479, #5766, #6595, #6595)
- The share link command has been moved to its own extension so that it can be overridden (#5460, #5388)
- Creating a new services session now requires passing a kernel model instead of a kernel instance (#6503, #6142)
- We upgraded the Webpack raw file loader. The new version of the raw loader exports ES2015 modules, so this may require changes in extensions that import files using the raw loader. For example, if you did `require('myfile.md')` to get the content of *myfile.md* as a string, you now should import it using ES2015 *import* syntax, or use *require('myfile.md').default*.
- Widget factories now can support custom cloning behavior from an optional source widget (#6060, #6044)
- We have renamed the type `InstanceTracker` to `WidgetTracker` (#6569).
- In order to add widgets to the main area (e.g. as in the old XKCD extension tutorial), the correct syntax is now `app.shell.add(widget)` or `app.shell.add(widget, 'main')`, see here.

6.4 v0.35.0

6.4.1 October 3, 2018

See the JupyterLab 0.35.0 milestone on GitHub for the full list of pull requests and issues closed.

6.4.2 Features

- A notebook cell can now be readonly, reflecting its `enabled` metadata. (#5401, #1312)
- Add "Go To Line" in the Edit menu for text editors. (#5377)
- Sidebar panels can now be switched between left and right sidebars. Right-click on a sidebar tab to move it to the other sidebar. (#5347, #5054, #3707)
- Make the sidebar a bit narrower, and make the minimum width adjustable from a theme. (#5245)
- Populate the File, Export Notebook As… submenu from the server nbconvert capabilities. (#5217)
- Server contents managers can now tell JupyterLab to open files as notebooks. For example, several custom contents managers save and open notebooks as Markdown files. (#5247, #4924)
- Add a command-line interface for managing workspaces. (#5166)
- Allow safe inline CSS styles in Markdown. (#5012, #1812)
- Add Quit to File menu when appropriate. (#5226, #5252, #5246, #5280)
- Rework extension manager user experience. (#5147, #5042)

6.4.3 Dark theme

- Show a dark splash screen when using a dark theme. (#5339, #5338, #5403)
- Fix code completion menu for a dark theme. (#5364, #5349)
- Style CSV viewer for a dark theme. (#5304, #3456)
- Make Matplotlib figures legible in a dark theme. (#5232)
- Fix notebook cell dropdown legibility in a dark theme. (#5168)

6.4.4 Bug fixes

- Various save options in the file menu and toolbar are now disabled when a file is not writable. (#5376, #5391)
- Kernel selector dialog no longer cuts off kernel names. (#5260, #5181)
- Fix focus issues with the toolbar. (#5344, #5324, #2995, #5328)
- Fix toolbar button enabled/disabled status. (#5278)
- Table alignment is now respected in Markdown. (#5301, #3180)
- Fix syntax highlighting for Markdown lists. (#5297, #2741)
- Use the current filebrowser instead of the default one for various commands. (#5390)
- Fix escaping in link handling to conform to Markdown syntax. This means that spaces in link references now need to be encoded as `%20`. (#5383, #5340, #5153)

6.4.5 Build system

- Use Typescript 3.1. (#5360)
- Use Lerna 3.2.1. (#5262)
- Node >=6.11.5 is now required. (#5227)
- Pin vega-embed version to 3.18.2. (#5342)
- Use Jest for services tests. (#5251, #5282)
- Make it easier for third party extensions to use the JupyterLab test app and testing utilities. (#5415)
- Fix `jupyter lab clean` on Windows. (#5400, #5397)
- Fix `jupyter lab build` on NFS. (#5237, #5233)
- Build wheels for Python 3 only. (#5287)
- Migrate to using `jupyterlab_server` instead of `jupyterlab_launcher` and fix the app example. (#5316)
- Move Mathjax 2 typesetter to a library package. (#5259, #5257)

6.4.6 For Developers

- Default toolbar buttons can be overridden, and mime renderers can now specify toolbar buttons. (#5398, #5370, #5363)

- A JupyterLab application instance can now be given a document registry, service manager, and command linker. (#5291)

6.5 v0.34.0

6.5.1 August 18, 2018

See the JupyterLab 0.34.0 milestone on GitHub for the full list of pull requests and issues closed.

6.5.2 Key Features

- Notebooks, consoles, and text files now have access to completions for local tokens.
- Python 3.5+ is now required to use JupyterLab. Python 2 kernels can still be run within JupyterLab.
- Added the pipe (`|`) character as a CSV delimiter option.
- Added "Open From Path…"" to top level `File` menu.
- Added "Copy Download Link" to context menu for files.

6.5.3 Changes for Developers

- Notebooks, consoles, and text files now have access to completions for local tokens. If a text file has a running kernel associated with its path (as happens with an attached console), it also gets completions and tooltips from that kernel. (#5049)
- The `FileBrowser` widget has a new constructor option `refreshInterval`, allowing the creator to customize how often the widget polls the storage backend. This can be useful to prevent rate-limiting in certain contexts. (#5048)
- The application shell now gets a pair of CSS data attributes indicating the current theme, and whether it is light or dark. Extension authors can write CSS rules targeting these to have their extension UI elements respond to the application theme. For instance, to write a rule targeting whether the theme is overall light or dark, you can use

  ```
  [data-theme-light="true"] your-ui-class {
    background-color: white;
  }
  [data-theme-light="false"] your-ui-class {
    background-color: black;
  }
  ```

  The theme name can also be targeted by writing CSS rules for `data-theme-name`. (#5078)
- The `IThemeManager` interface now exposes a signal for `themeChanged`, allowing extension authors to react to changes in the theme. Theme extensions must also provide a new boolean property `isLight`, declaring whether they are broadly light colored. This data allows third-party extensions to react better to the active application theme. (#5078)
- Added a patch to update the `uploads` for each `FileBrowserModel` instantly whenever a file upload errors. Previously, the upload that erred was only being removed from uploads upon an update. This would allow the status bar component and other extensions that use the `FileBrowserModel` to be more precise. (#5077)
- Cell IDs are now passed in the shell message as part of the cell metadata when a cell is executed. This helps in developing reactive kernels. (#5033)

- The IDs of all deleted cells since the last run cell are now passed as part of the cell metadata on execution. The IDs of deleted cells since the last run cell are stored as `deletedCells` in `NotebookModel`. This helps in developing reactive kernels. (#5037)
- The `ToolbarButton` in `apputils` has been refactored with an API change and now uses a React component `ToolbarButtonComponent` to render its children. It is now a `div` with a single `button` child, which in turn as two `span` elements for an icon and text label. Extensions that were using the `className` options should rename it as `iconClassName`. The `className` options still exists, but it used as the CSS class on the `button` element itself. The API changes were done to accommodate styling changes to the button. (#5117)
- The `Toolbar.createFromCommand` function has been replaced by a dedicated `ToolbarButton` subclass called `CommandToolbarButton`, that wraps a similarly named React component. (#5117)
- The design and styling of the right and left sidebars tabs has been improved to address #5054. We are now using icons to render tabs for the extensions we ship with JupyterLab and extension authors are encouraged to do the same (text labels still work). Icon based tabs can be used by removing `widget.caption` and adding `widget.iconClass = '<youriconclass> jp-SideBar-tabIcon';`. (#5117)
- The style of buttons in JupyterLab has been updated to a borderless design. (#5117)
- A new series of helper CSS classes for stying SVG-based icons at different sizes has been added: `jp-Icon`, `jp-Icon-16`, `jp-Icon-18`, `jp-Icon-20`.
- The rank of the default sidebar widget has been updated. The main change is giving the extension manager a rank of `1000` so that it appears at the end of the default items.
- Python 3.5+ is now required to use JupyterLab. Python 2 kernels can still be run within JupyterLab. (#5119)
- JupyterLab now uses `yarn 1.9.4` (aliased as `jlpm`), which now allows uses to use Node 10+. (#5121)
- Clean up handling of `baseUrl` and `wsURL` for `PageConfig` and `ServerConnection`. (#5111)

6.5.4 Other Changes

- Added the pipe (`|`) character as a CSV delimiter option. (#5112)
- Added `Open From Path...` to top level `File` menu. (#5108)
- Added a `saveState` signal to the document context object. (#5096)
- Added "Copy Download Link" to context menu for files. (#5089)
- Extensions marked as `deprecated` are no longer shown in the extension manager. (#5058)
- Remove `In` and `Out` text from cell prompts. Shrunk the prompt width from 90px to 64px. In the light theme, set the prompt colors of executed console cells to active prompt colors and reduced their opacity to 0.5. In the dark theme, set the prompt colors of executed console cells to active prompt colors and set their opacity to 1. (#5097 and #5130)

6.5.5 Bug Fixes

- Fixed a bug in the rendering of the "New Notebook" item of the command palette. (#5079)
- We only create the extension manager widget if it is enabled. This prevents unnecessary network requests to `npmjs.com`. (#5075)
- The `running` panel now shows the running sessions at startup. (#5118)
- Double clicking a file in the file browser always opens it rather than sometimes selecting it for a rename. (#5101)

6.6 v0.33.0

6.6.1 July 26, 2018

See the JupyterLab 0.33.0 milestone on GitHub for the full list of pull requests and issues closed.

6.6.2 Key Features:

- *No longer in beta*
- *Workspaces*
- *Menu items*
- *Keyboard shortcuts*
- *Command palette items*
- *Settings*
- *Larger file uploads*
- *Extension management and installation*
- *Interface changes*
- *Renderers*
- *Changes for developers*
- *Other fixes*

6.6.3 No longer in beta

In JupyterLab 0.33, we removed the "Beta" label to better signal that JupyterLab is ready for users to use on a daily basis. The extension developer API is still being stabilized. See the release blog post for details. (#4898, #4920)

6.6.4 Workspaces

We added new workspace support, which enables you to have multiple saved layouts, including in different browser windows. See the *workspace documentation* for more details. (#4502, #4708, #4088, #4041 #3673, #4780)

6.6.5 Menu items

- "Activate Previously Used Tab" added to the Tab menu (`Ctrl/Cmd Shift '`) to toggle between the previously active tabs in the main area. (#4296)
- "Reload From Disk" added to the File menu to reload an open file from the state saved on disk. (#4615)
- "Save Notebook with View State" added to the File menu to persist the notebook collapsed and scrolled cell state. We now read the `collapsed`, `scrolled`, `jupyter.source_hidden` and `jupyter.outputs_hidden` notebook cell metadata when opening. `collapsed` and `jupyter.outputs_hidden` are redundant and the initial collapsed state is the union of both of them. When the state is persisted, if an output is collapsed, both will be written with the value `true`, and if it is not, both will not be written. (#3981)
- "Increase/Decrease Font Size" added to the text editor settings menu. (#4811)

- "Show in File Browser" added to a document tab's context menu. (#4500)
- "Open in New Browser Tab" added to the file browser context menu. (#4315)
- "Copy Path" added to file browser context menu to copy the document's path to the clipboard. (#4582)
- "Show Left Area" has been renamed to "Show Left Sidebar" for consistency (same for right sidebar). (#3818)

6.6.6 Keyboard shortcuts

- "Save As…" given the keyboard shortcut `Ctrl/Cmd Shift S`. (#4560)
- "Run All Cells" given the keyboard shortcut `Ctrl/Cmd Shift Enter`. (#4558)
- "notebook:change-to-cell-heading-X" keyboard shortcuts (and commands) renamed to "notebook:change-cell-to-heading-X" for X=1…6. This fixes the notebook command-mode keyboard shortcuts for changing headings. (#4430)
- The console execute shortcut can now be set to either `Enter` or `Shift Enter` as a Console setting. (#4054)

6.6.7 Command palette items

- "Notebook" added to the command palette to open a new notebook. (#4812)
- "Run Selected Text or Current Line in Console" added to the command palette to run the selected text or current line from a notebook in a console. A default keyboard shortcut for this command is not yet provided, but can be added by users with the `notebook:run-in-console` command. To add a keyboard shortcut `Ctrl G` for this command, use the "Settings" | "Advanced Settings Editor" menu item to open the "Keyboard Shortcuts" advanced settings, and add the following JSON in the shortcut JSON object in the User Overrides pane (adjust the actual keyboard shortcut if you wish). (#3453, #4206, #4330)

```
{
  "command": "notebook:run-in-console",
  "keys": ["Ctrl G"],
  "selector": ".jp-Notebook.jp-mod-editMode"
}
```

- The command palette now renders labels, toggled state, and keyboard shortcuts in a more consistent and correct way. (#4533, #4510)

6.6.8 Settings

- "fontFamily", "fontSize", and "lineHeight" settings added to the text editor advanced settings. (#4673)
- Solarized dark and light text editor themes from CodeMirror. (#4445)

6.6.9 Larger file uploads

- Support for larger file uploads (>15MB) when using Jupyter notebook server version >= 5.1. (#4224)

6.6.10 Extension management and installation

- New extension manager for installing JupyterLab extensions from npm within the JupyterLab UI. You can enable this from the Advanced Settings interface. (#4682, #4925)

- Please note that to install extensions in JupyterLab, you must use NodeJS version 9 or earlier (i.e., not NodeJS version 10). We will upgrade yarn, with NodeJS version 10 support, when a bug in yarn is fixed. (#4804)

6.6.11 Interface changes

- Wider tabs in the main working area to show longer filenames. (#4801)
- Initial kernel selection for a notebook or console can no longer be canceled: the user must select a kernel. (#4596)
- Consoles now do not display output from other clients by default. A new "Show All Kernel Activity" console context menu item has been added to show all activity from a kernel in the console. (#4503)
- The favicon now shows the busy status of the kernels in JupyterLab. (#4361, #3957, #4966)

6.6.12 Renderers

- JupyterLab now ships with a Vega4 renderer by default (upgraded from Vega3). (#4806)
- The HTML sanitizer now allows some extra tags in rendered HTML, including `kbd`, `sup`, and `sub`. (#4618)
- JupyterLab now recognizes the `.tsv` file extension as tab-separated files. (#4684)
- Javascript execution in notebook cells has been re-enabled. (#4515)

6.6.13 Changes for developers

- A new signal for observing application dirty status state changes. (#4840)
- A new signal for observing notebook cell execution. (#4740, #4744)
- A new `anyMessage` signal for observing any message a kernel sends or receives. (#4437)
- A generic way for different widgets to register a "Save with extras" command that appears in the File menu under save. (#3981)
- A new API for removing groups from a JupyterLab menu. `addGroup` now returns an `IDisposable` which can be used to remove the group. `removeGroup` has been removed. (#4890)
- The `Launcher` now uses commands from the application `CommandRegistry` to launch new activities. Extension authors that add items to the launcher will need to update them to use commands. (#4757)
- There is now a top-level `addToBottomArea` function in the application, allowing extension authors to add bottom panel items like status bars. (#4752)
- Rendermime extensions can now indicate that they are the default rendered widget factory for a file-type. For instance, the default widget for a markdown file is a text editor, but the default rendered widget is the markdown viewer. (#4692)
- Add new workspace REST endpoints to `jupyterlab_server` and make them available in `@jupyterlab/services`. (#4841)
- Documents created with a mimerenderer extension can now be accessed using an `IInstanceTracker` which tracks them. Include the token `IMimeDocumentTracker` in your plugin to access this. The `IInstanceTracker` interface has also gained convenience functions `find` and `filter` to simplify iterating over instances. (#4762)
- RenderMime render errors are now displayed to the user. (#4465)
- `getNotebookVersion` is added to the `PageConfig` object. (#4224)

- The session `kernelChanged` signal now contains both the old kernel and the new kernel to make it easy to unregister things from the old kernel. (#4834)
- The `connectTo` functions for connecting to kernels and sessions are now synchronous (returning a connection immediately rather than a promise). The DefaultSession `clone` and `update` methods are also synchronous now. (#4725)
- Kernel message processing is now asynchronous, which guarantees the order of processing even if a handler is asynchronous. If a kernel message handler returns a promise, kernel message processing is paused until the promise resolves. The kernel's `anyMessage` signal is emitted synchronously when a message is received before asynchronous message handling, and the `iopubMessage` and `unhandledMessage` signals are emitted during asynchronous message handling. These changes mean that the comm `onMsg` and `onClose` handlers and the kernel future `onReply`, `onIOPub`, and `onStdin` handlers, as well as the comm target and message hook handlers, may be asynchronous and return promises. (#4697)
- Kernel comm targets and message hooks now are unregistered with `removeCommTarget` and `removeMessageHook`, instead of using disposables. The corresponding `registerCommTarget` and `registerMessageHook` functions now return nothing. (#4697)
- The kernel `connectToComm` function is synchronous, and now returns the comm rather than a promise to the comm. (#4697)
- The `KernelFutureHandler` class `expectShell` constructor argument is renamed to `expectReply`. (#4697)
- The kernel future `done` returned promise now resolves to undefined if there is no reply message. (#4697)
- The `IDisplayDataMsg` is updated to have the optional `transient` key, and a new `IUpdateDisplayDataMsg` type was added for update display messages. (#4697)
- The `uuid` function from `@jupyterlab/coreutils` is removed. Instead import `UUID` from `@phosphor/coreutils` and use `UUID.uuid4()`. (#4604)
- Main area widgets like the launcher and console inherit from a common `MainAreaWidget` class which provides a content area (`.content`) and a toolbar (`.toolbar`), consistent focus handling and activation behavior, and a spinner displayed until the given `reveal` promise is resolved. Document widgets, like the notebook and text editor and other documents opened from the document manager, implement the `IDocumentWidget` interface (instead of `DocumentRegistry.IReadyWidget`), which builds on `MainAreaWidget` and adds a `.context` attribute for the document context and makes dirty handling consistent. Extension authors may consider inheriting from the `MainAreaWidget` or `DocumentWidget` class for consistency. Several effects from these changes are noted below. (#3499, #4453)
  - The notebook panel `.notebook` attribute is renamed to `.content`.
  - The text editor is now the `.content` of a `DocumentWidget`, so the top-level editor widget has a toolbar and the editor itself is `widget.content.editor` rather than just `widget.editor`.
  - Mime documents use a `MimeContent` widget embedded inside of a `DocumentWidget` now.
  - Main area widgets and document widgets now have a `revealed` promise which resolves when the widget has been revealed (i.e., the spinner has been removed). This should be used instead of the `ready` promise.

Changes in the JupyterLab code infrastructure include:

- The JupyterLab TypeScript codebase is now compiled to ES2015 (ES6) using TypeScript 2.9. We also turned on the TypeScript `esModuleInterop` flag to enable more natural imports from non-es2015 JavaScript modules. With the update to ES2015 output, code generated from async/await syntax became much more manageable, so we have started to use async/await liberally throughout the codebase, especially in tests. Because we use Typedoc for API documentation, we still use syntax compatible with TypeScript 2.7 where Typedoc is used. Extension authors may have some minor compatibility updates to make. If you are writing an extension in

TypeScript, we recommend updating to TypeScript 2.9 and targeting ES2015 output as well. (#4462, #4675, #4714, #4797)

- The JupyterLab codebase is now formatted using Prettier. By default the development environment installs a pre-commit hook that formats your staged changes. (#4090)
- Updated build infrastructure using webpack 4 and better typing. (#4702, #4698)
- Upgraded yarn to version 1.6. Please note that you must use NodeJS version 9 or earlier with JupyterLab (i.e., not NodeJS version 10). We will upgrade yarn, with NodeJS version 10 support, when a bug in yarn is fixed. (#4804)
- Various process utilities were moved to `jupyterlab_server`. (#4696)

6.6.14 Other fixes

- Fixed a rendering bug with the Launcher in single-document mode. (#4805)
- Fixed a bug where the native context menu could not be triggered in a notebook cell in Chrome. (#4720)
- Fixed a bug where the cursor would not show up in the dark theme. (#4699)
- Fixed a bug preventing relative links from working correctly in alternate `IDrives`. (#4613)
- Fixed a bug breaking the image viewer upon saving the image. (#4602)
- Fixed the font size for code blocks in notebook Markdown headers. (#4617)
- Prevented a memory leak when repeatedly rendering a Vega chart. (#4904)
- Support dropped terminal connection re-connecting. (#4763, #4802)
- Use `require.ensure` in `vega4-extension` to lazily load `vega-embed` and its dependencies on first render. (#4706)
- Relative links to documents that include anchor tags will now correctly scroll the document to the right place. (#4692)
- Fix default settings JSON in setting editor. (#4591, #4595)
- Fix setting editor pane layout's stretch factor. (#2971, #4772)
- Programmatically set settings are now output with nicer formatting. (#4870)
- Fixed a bug in displaying one-line CSV files. (#4795, #4796)
- Fixed a bug where JSON arrays in rich outputs were collapsed into strings. (#4480)

6.7 Beta 2 (v0.32.0)

6.7.1 Apr 16, 2018

This is the second in the JupyterLab Beta series of releases. It contains many enhancements, bugfixes, and refinements, including:

- Better handling of a corrupted or invalid state database. (#3619, #3622, #3687, #4114).
- Fixing file dirty status indicator. (#3652).
- New option for whether to autosave documents. (#3734).
- More commands in the notebook context menu. (#3770, #3909)

- Defensively checking for completion metadata from kernels. (#3888)
- New "Shutdown all" button in the Running panel. (#3764)
- Performance improvements wherein non-focused documents poll the server less. (#3931)
- Changing the keyboard shortcut for singled-document-mode to something less easy to trigger. (#3889)
- Performance improvements for rendering text streams, especially around progress bars. (#4045).
- Canceling a "Restart Kernel" now functions correctly. (#3703).
- Defer loading file contents until after the application has been restored. (#4087).
- Ability to rotate, flip, and invert images in the image viewer. (#4000)
- Major performance improvements for large CSV viewing. (#3997).
- Always show the context menu in the file browser, even for an empty directory. (#4264).
- Handle asynchronous comm messages in the services library more correctly (Note: this means `@jupyterlab/services` is now at version 2.0!) ([#4115](https://github.com/jupyterlab/jupyterlab/issues/4115)).
- Display the kernel banner in the console when a kernel is restarted to mark the restart ([#3663](https://github.com/jupyterlab/jupyterlab/issues/3663)).
- Many tweaks to the UI, as well as better error handling.

6.8 Beta 1 (v0.31.0)

6.8.1 Jan 11, 2018

- Add a `/tree` handler and `Copy Shareable Link` to file listing right click menu: https://github.com/jupyterlab/jupyterlab/pull/3396
- Experimental support for saved workspaces: #3490, #3586
- Added types information to the completer: #3508
- More improvements to the top level menus: https://github.com/jupyterlab/jupyterlab/pull/3344
- Editor settings for notebook cells: https://github.com/jupyterlab/jupyterlab/pull/3441
- Simplification of theme extensions: https://github.com/jupyterlab/jupyterlab/pull/3423
- New CSS variable naming scheme: https://github.com/jupyterlab/jupyterlab/pull/3403
- Improvements to cell selection and dragging: https://github.com/jupyterlab/jupyterlab/pull/3414
- Style and typography improvements: https://github.com/jupyterlab/jupyterlab/pull/3468 https://github.com/jupyterlab/jupyterlab/pull/3457 https://github.com/jupyterlab/jupyterlab/pull/3445 https://github.com/jupyterlab/jupyterlab/pull/3431 https://github.com/jupyterlab/jupyterlab/pull/3428 https://github.com/jupyterlab/jupyterlab/pull/3408 https://github.com/jupyterlab/jupyterlab/pull/3418

6.9 v0.30.0

6.9.1 Dec 05, 2017

- Semantic menus: https://github.com/jupyterlab/jupyterlab/pull/3182

- Settings editor now allows comments and provides setting validation: https://github.com/jupyterlab/jupyterlab/pull/3167
- Switch to Yarn as the package manager: https://github.com/jupyterlab/jupyterlab/pull/3182
- Support for carriage return in outputs: #2761
- Upgrade to TypeScript 2.6: https://github.com/jupyterlab/jupyterlab/pull/3288
- Cleanup of the build, packaging, and extension systems. `jupyter labextension install` is now the recommended way to install a local directory. Local directories are considered linked to the application. cf https://github.com/jupyterlab/jupyterlab/pull/3182
- `--core-mode` and `--dev-mode` are now semantically different. `--core-mode` is a version of JupyterLab using released JavaScript packages and is what we ship in the Python package. `--dev-mode` is for unreleased JavaScript and shows the red banner at the top of the page. https://github.com/jupyterlab/jupyterlab/pull/3270

6.10 v0.29.2

6.10.1 Nov 17, 2017

Bug fix for file browser right click handling. https://github.com/jupyterlab/jupyterlab/issues/3019

6.11 v0.29.0

6.11.1 Nov 09, 2017

- Create new view of cell in cell context menu. #3159
- New Renderers for VDOM and JSON mime types and files. #3157
- Switch to React for our VDOM implementation. Affects the `VDomRenderer` class. #3133
- Standalone Cell Example. #3155

6.12 v0.28.0

6.12.1 Oct 16, 2017

This release generally focuses on developer and extension author enhancements and general bug fixes.

- Plugin id and schema file conventions change. https://github.com/jupyterlab/jupyterlab/pull/2936.
- Theme authoring conventions change. #3061
- Enhancements to enabling and disabling of extensions. #3078
- Mime extensions API change (`name` -> `id` and new naming convention). #3078
- Added a `jupyter lab --watch` mode for extension authors. #3077
- New comprehensive extension authoring tutorial. #2921
- Added the ability to use an alternate LaTeX renderer. #2974
- Numerous bug fixes and style enhancements.

6.13 v0.27.0

6.13.1 Aug 23, 2017

- Added support for dynamic theme loading. https://github.com/jupyterlab/jupyterlab/pull/2759
- Added an application splash screen. https://github.com/jupyterlab/jupyterlab/pull/2899
- Enhancements to the settings editor. https://github.com/jupyterlab/jupyterlab/pull/2784
- Added a PDF viewer. #2867
- Numerous bug fixes and style improvements.

6.14 v0.26.0

6.14.1 Jul 21, 2017

- Implemented server side handling of users settings: https://github.com/jupyterlab/jupyterlab/pull/2585
- Revamped the handling of file types in the application - affects document and mime renderers: https://github.com/jupyterlab/jupyterlab/pull/2701
- Updated dialog API - uses virtual DOM instead of raw DOM nodes and better use of the widget lifecycle: https://github.com/jupyterlab/jupyterlab/pull/2661

6.15 v0.25.0

6.15.1 Jul 07, 2017

- Added a new extension type for mime renderers, with the `vega2-extension` as a built-in example. Also overhauled the rendermime interfaces. https://github.com/jupyterlab/jupyterlab/pull/2488 https://github.com/jupyterlab/jupyterlab/pull/2555 https://github.com/jupyterlab/jupyterlab/pull/2595
- Finished JSON-schema based settings system, using client-side storage for now. https://github.com/jupyterlab/jupyterlab/pull/2411
- Overhauled the launcher design. https://github.com/jupyterlab/jupyterlab/pull/2506 https://github.com/jupyterlab/jupyterlab/pull/2580
- Numerous bug fixes and style updates.

6.16 v0.24.0

6.16.1 Jun 16, 2017

- Overhaul of the launcher. #2380
- Initial implementation of client-side settings system. #2157
- Updatable outputs. #2439
- Use new Phosphor Datagrid for CSV viewer. #2433

- Added ability to enable/disable extensions without rebuilding. #2409
- Added language and tab settings for the file viewer. #2406
- Improvements to real time collaboration experience. #2387 #2333
- Compatibility checking for extensions. #2410
- Numerous bug fixes and style improvements.

6.17 v0.23.0

6.17.1 Jun 02, 2017

- Chat box feature. https://github.com/jupyterlab/jupyterlab/pull/2118
- Collaborative cursors. https://github.com/jupyterlab/jupyterlab/pull/2139
- Added concept of Drive to ContentsManager. https://github.com/jupyterlab/jupyterlab/pull/2248
- Refactored to enable switching the theme. https://github.com/jupyterlab/jupyterlab/pull/2283
- Clean up the APIs around kernel execution. https://github.com/jupyterlab/jupyterlab/pull/2266
- Various bug fixes and style improvements.

6.18 v0.22.0

6.18.1 May 18, 2017

- Export To… for notebooks. https://github.com/jupyterlab/jupyterlab/pull/2200
- Change kernel by clicking on the kernel name in the notebook. https://github.com/jupyterlab/jupyterlab/pull/2195
- Improved handling of running code in text editors. https://github.com/jupyterlab/jupyterlab/pull/2191
- Can select file in file browser by typing: https://github.com/jupyterlab/jupyterlab/pull/2190
- Ability to open a console for a notebook. https://github.com/jupyterlab/jupyterlab/pull/2189
- Upgrade to Phosphor 1.2 with Command Palette fuzzy matching improvements. #1182
- Rename of widgets that had `Widget` in the name and associated package names. https://github.com/jupyterlab/jupyterlab/pull/2177
- New `jupyter labhub` command to launch JupyterLab on JupyterHub: https://github.com/jupyterlab/jupyterlab/pull/2222
- Removed the `utils` from `@jupyterlab/services` in favor of `PageConfig` and `ServerConnection`. https://github.com/jupyterlab/jupyterlab/pull/2173 https://github.com/jupyterlab/jupyterlab/pull/2185
- Cleanup, bug fixes, and style updates.

6.19 v0.20.0

6.19.1 Apr 21, 2017

Release Notes:

- Overhaul of extension handling, see updated docs for users and developers. https://github.com/jupyterlab/jupyterlab/pull/2023
- Added single document mode and a `Tabs` sidebar. https://github.com/jupyterlab/jupyterlab/pull/2037
- More work toward real time collaboration - implemented a model database interface that can be in-memory by real time backends. https://github.com/jupyterlab/jupyterlab/pull/2039

Numerous bug fixes and improvements.

6.20 v0.19.0

6.20.1 Apr 04, 2017

Mainly backend-focused release with compatibility with Phosphor 1.0 and a big refactor of session handling (the ClientSession class) that provides a simpler object for classes like notebooks, consoles, inspectors, etc. to use to communicate with the API. Also includes improvements to the development workflow of JupyterLab itself after the big split.

https://github.com/jupyterlab/jupyterlab/pull/1984 https://github.com/jupyterlab/jupyterlab/pull/1927

6.21 v0.18.0

6.21.1 Mar 21, 2017

- Split the repository into multiple packages that are managed using the lerna build tool. https://github.com/jupyterlab/jupyterlab/issues/1773
- Added restoration of main area layout on refresh. https://github.com/jupyterlab/jupyterlab/pull/1880
- Numerous bug fixes and style updates.

6.22 v0.17.0

6.22.1 Mar 01, 2017

- Upgrade to new `@phosphor` packages - brings a new Command Palette interaction that should be more intuitive, and restores the ability to drag to dock panel edges https://github.com/jupyterlab/jupyterlab/pull/1762.
- Refactor of `RenderMime` and associated renders to use live models. See https://github.com/jupyterlab/jupyterlab/pull/1709 and https://github.com/jupyterlab/jupyterlab/issues/1763.
- Improvements and bug fixes for the completer widget: https://github.com/jupyterlab/jupyterlab/pull/1778
- Upgrade CodeMirror to 5.23: https://github.com/jupyterlab/jupyterlab/pull/1764
- Numerous style updates and bug fixes.

6.23 v0.16.0

6.23.1 Feb 09, 2017

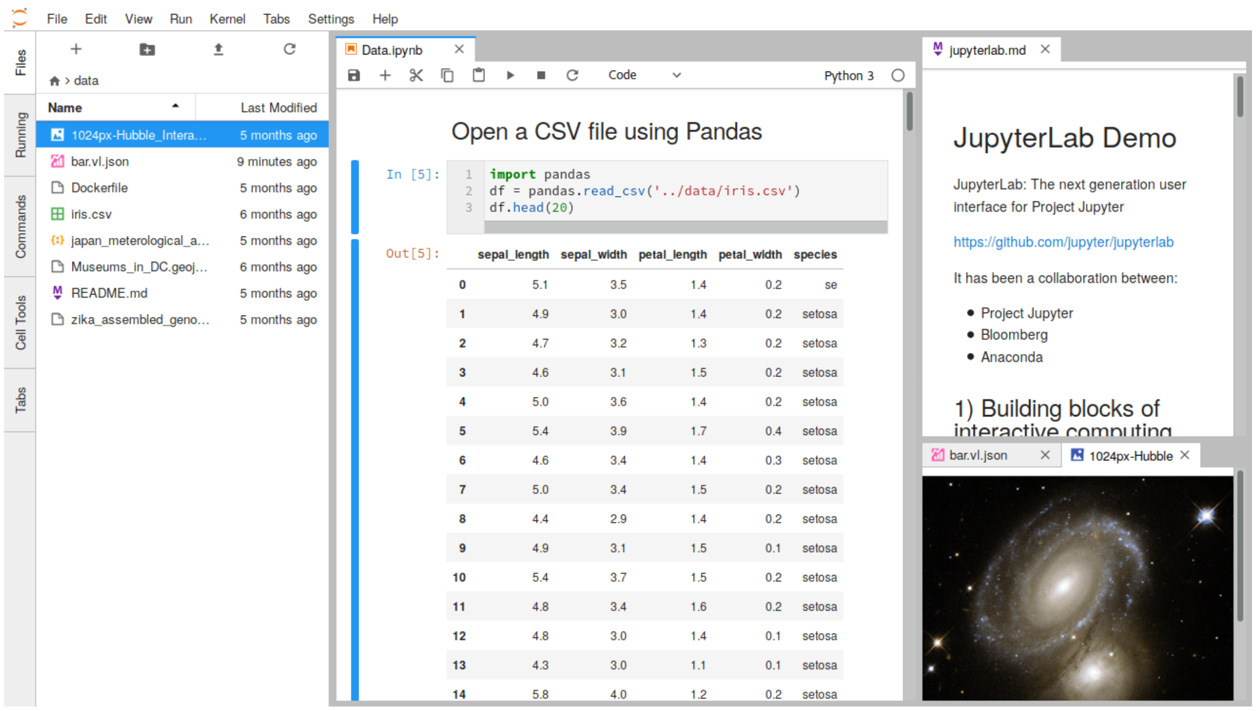
- Adds a Cell Tools sidebar that allows you to edit notebook cell metadata. #1586.
- Adds keyboard shortcuts to switch between tabs (Cmd/Ctrl LeftArrow and Cmd/Ctrl RightArrow). #1647
- Upgrades to xterm.js 2.3. #1664
- Fixes a bug in application config, but lab extensions will need to be re-enabled. #1607
- Numerous other bug fixes and style improvements.

CHAPTER SEVEN

THE JUPYTERLAB INTERFACE

JupyterLab provides flexible building blocks for interactive, exploratory computing. While JupyterLab has many features found in traditional integrated development environments (IDEs), it remains focused on interactive, exploratory computing.

The JupyterLab interface consists of a *main work area* containing tabs of documents and activities, a collapsible *left sidebar*, and a *menu bar*. The left sidebar contains a *file browser*, the *list of running kernels and terminals*, the *command palette*, the *notebook cell tools inspector*, and the *tabs list*.



Name	Last Modified
1024px-Hubble_Intera…	5 months ago
bar.vl.json	9 minutes ago
Dockerfile	5 months ago
iris.csv	6 months ago
japan_meterological_a…	5 months ago
Museums_in_DC.geoj…	6 months ago
README.md	5 months ago
zika_assembled_geno…	5 months ago

Open a CSV file using Pandas

In [5]:
```
import pandas
df = pandas.read_csv('../data/iris.csv')
df.head(20)
```

Out[5]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	se
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa
6	4.6	3.4	1.4	0.3	setosa
7	5.0	3.4	1.5	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
9	4.9	3.1	1.5	0.1	setosa
10	5.4	3.7	1.5	0.2	setosa
11	4.8	3.4	1.6	0.2	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.3	3.0	1.1	0.1	setosa
14	5.8	4.0	1.2	0.2	setosa

JupyterLab Demo

JupyterLab: The next generation user interface for Project Jupyter

https://github.com/jupyter/jupyterlab

It has been a collaboration between:

- Project Jupyter
- Bloomberg
- Anaconda

1) Building blocks of interactive computing

JupyterLab sessions always reside in a *workspace*. Workspaces contain the state of JupyterLab: the files that are currently open, the layout of the application areas and tabs, etc. Workspaces can be saved on the server with *named workspace URLs*. To learn more about URLs in Jupyterlab, visit *JupyterLab URLs*.

7.1 Menu Bar

The menu bar at the top of JupyterLab has top-level menus that expose actions available in JupyterLab with their keyboard shortcuts. The default menus are:

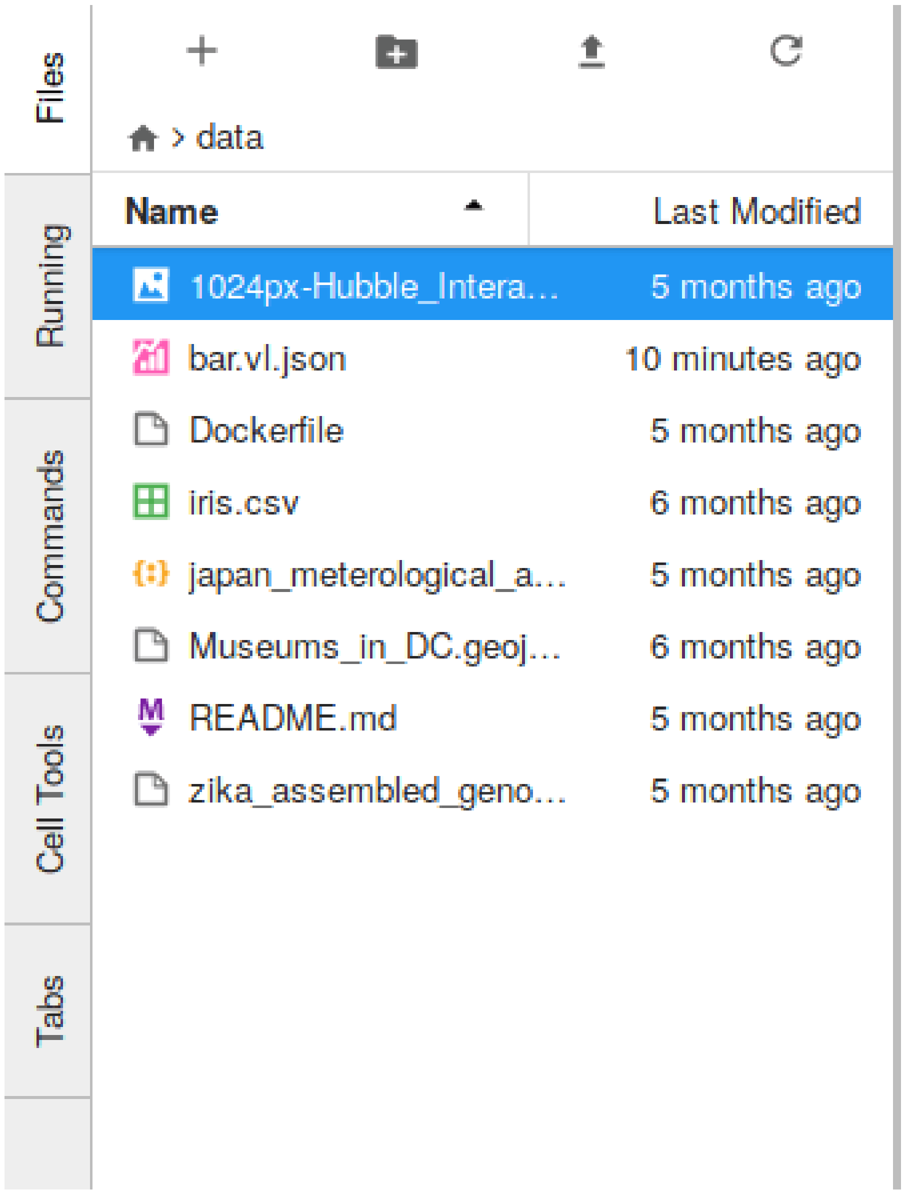
- **File**: actions related to files and directories

- **Edit**: actions related to editing documents and other activities
- **View**: actions that alter the appearance of JupyterLab
- **Run**: actions for running code in different activities such as notebooks and code consoles
- **Kernel**: actions for managing kernels, which are separate processes for running code
- **Tabs**: a list of the open documents and activities in the dock panel
- **Settings**: common settings and an advanced settings editor
- **Help**: a list of JupyterLab and kernel help links

JupyterLab extensions can also create new top-level menus in the menu bar.

7.2 Left Sidebar

The left sidebar contains a number of commonly-used tabs, such as a file browser, a list of running kernels and terminals, the command palette, and a list of tabs in the main work area:



Name	Last Modified
1024px-Hubble_Intera...	5 months ago
bar.vl.json	10 minutes ago
Dockerfile	5 months ago
iris.csv	6 months ago
japan_meterological_a...	5 months ago
Museums_in_DC.geoj...	6 months ago
README.md	5 months ago
zika_assembled_geno...	5 months ago

The left sidebar can be collapsed or expanded by selecting "Show Left Sidebar" in the View menu or by clicking on the active sidebar tab:

JupyterLab extensions can add additional panels to the left sidebar.

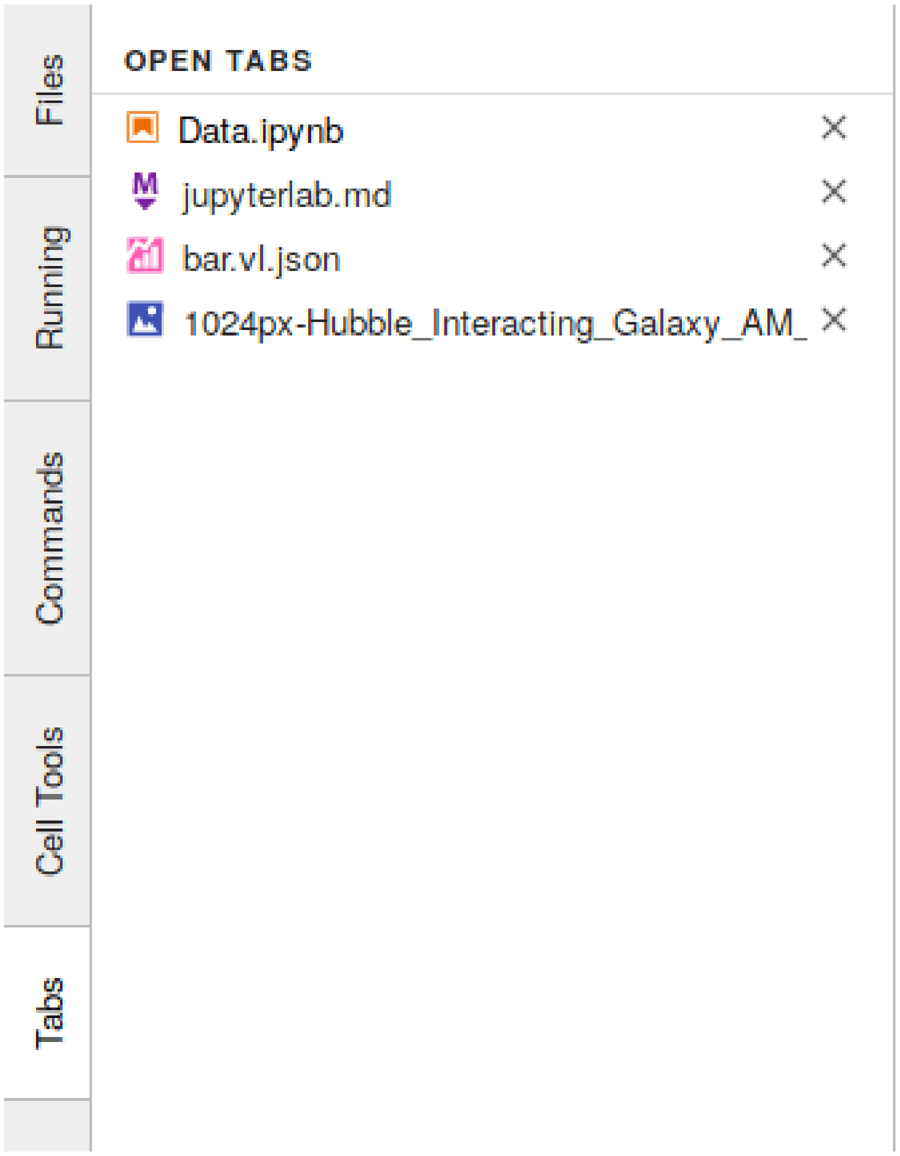
7.3 Main Work Area

The main work area in JupyterLab enables you to arrange documents (notebooks, text files, etc.) and other activities (terminals, code consoles, etc.) into panels of tabs that can be resized or subdivided. Drag a tab to the center of a tab panel to move the tab to the panel. Subdivide a tab panel by dragging a tab to the left, right, top, or bottom of the panel:

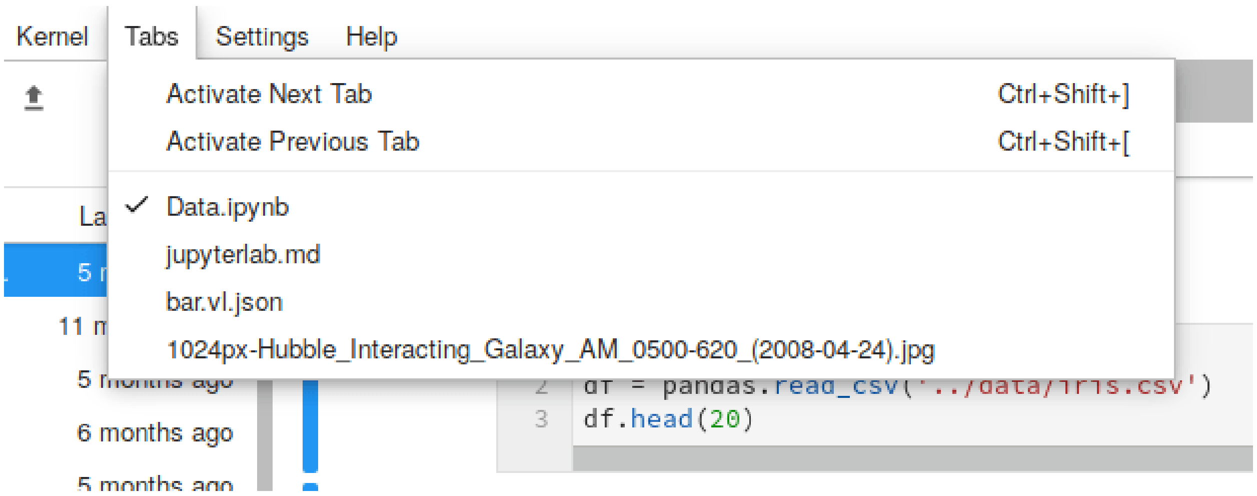
The work area has a single current activity. The tab for the current activity is marked with a colored top border (blue by default).

7.4 Tabs and Single-Document Mode

The Tabs panel in the left sidebar lists the open documents or activities in the main work area:



The same information is also available in the Tabs menu:


Kernel
Tabs
Settings
Help
Activate Next Tab Ctrl+Shift+]
Activate Previous Tab Ctrl+Shift+[
✓ Data.ipynb
jupyterlab.md
bar.vl.json
1024px-Hubble_Interacting_Galaxy_AM_0500-620_(2008-04-24).jpg
6 months ago
5 months ago

```
df = pandas.read_csv('../data/iris.csv')
df.head(20)
```

It is often useful to focus on a single document or activity without closing other tabs in the main work area. Single-document mode enable this, while making it simple to return to your multi-activity layout in the main work area. Toggle single-document mode using the View menu:

When you leave single-document mode, the original layout of the main area is restored.

7.5 Context Menus

Many parts of JupyterLab, such as notebooks, text files, code consoles, and tabs, have context menus that can be accessed by right-clicking on the element:

The browser's native context menu can be accessed by holding down `Shift` and right-clicking:

7.6 Keyboard Shortcuts

As in the classic Notebook, you can navigate the user interface through keyboard shortcuts. You can find and customize the current list of keyboard shortcuts by selecting the Advanced Settings Editor item in the Settings menu, then selecting Keyboard Shortcuts in the Settings tab.

You can also customize the *text editor* to use vim, emacs, or Sublime Text keyboard maps by using the Text Editor Key Map submenu in the Settings menu:

CHAPTER

EIGHT

JUPYTERLAB URLS

Like the classic notebook, JupyterLab provides a way for users to copy URLs that open a specific notebook or file. Additionally, JupyterLab URLs are an advanced part of the user interface that allows for managing workspaces. These two functions – file paths and workspaces – can be *combined in URLs that open a specific file in a specific workspace*.

8.1 File Navigation with `/tree`

JupyterLab's file navigation URLs adopts the nomenclature of the classic notebook; these URLs are `/tree` URLs:

```
http(s)://<server:port>/<lab-location>/lab/tree/path/to/notebook.ipynb
```

Entering this URL will open the notebook in JupyterLab in *single-document mode*.

By default, the file browser will navigate to the directory containing the requested file. This behavior can be changed with the optional `file-browser-path` query parameter:

```
http(s)://<server:port>/<lab-location>/lab/tree/path/to/notebook.ipynb?file-browser-
↪path=/
```

Entering the above URL will show the workspace root directory instead of the `/path/to/` directory in the file browser.

8.2 Managing Workspaces (UI)

JupyterLab sessions always reside in a workspace. Workspaces contain the state of JupyterLab: the files that are currently open, the layout of the application areas and tabs, etc. When the page is refreshed, the workspace is restored.

The default workspace does not have a name and resides at the primary `/lab` URL:

```
http(s)://<server:port>/<lab-location>/lab
```

All other workspaces have a name that is part of the URL:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/foo
```

Workspaces save their state on the server and can be shared between multiple users (or browsers) as long as they have access to the same server.

A workspace should only be open in a single browser tab at a time. If JupyterLab detects that a workspace is being opened multiple times simultaneously, it will prompt for a new workspace name. Opening a document in two different browser tabs simultaneously is also not supported.

8.3 Cloning Workspaces

You can copy the contents of a workspace into another workspace with the `clone` url parameter.

To copy the contents of the workspace `foo` into the workspace `bar`:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/bar?clone=foo
```

To copy the contents of the default workspace into the workspace `foo`:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/foo?clone
```

To copy the contents of the workspace `foo` into the default workspace:

```
http(s)://<server:port>/<lab-location>/lab?clone=foo
```

8.4 Resetting a Workspace

Use the `reset` url parameter to clear a workspace of its contents.

To reset the contents of the workspace `foo`:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/foo?reset
```

To reset the contents of the default workspace:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/lab?reset
```

8.5 Combining URL Functions

These URL functions can be used separately, as above, or in combination.

To reset the workspace `foo` and load a specific notebook afterward:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/foo/tree/path/to/notebook.ipynb?
↪reset
```

To clone the contents of the workspace `bar` into the workspace `foo` and load a notebook afterward:

```
http(s)://<server:port>/<lab-location>/lab/workspaces/foo/tree/path/to/notebook.ipynb?
↪clone=bar
```

To reset the contents of the default workspace and load a notebook:

```
http(s)://<server:port>/<lab-location>/lab/tree/path/to/notebook.ipynb?reset
```

8.6 Managing Workspaces (CLI)

JupyterLab provides a command-line interface for workspace `import` and `export`:

```
$ # Exports the default JupyterLab workspace
$ jupyter lab workspaces export
{"data": {}, "metadata": {"id": "/lab"}}
$
$ # Exports the workspaces named `foo`
$ jupyter lab workspaces export foo
{"data": {}, "metadata": {"id": "/lab/workspaces/foo"}}
$
$ # Exports the workspace named `foo` into a file called `file_name.json`
$ jupyter lab workspaces export foo > file_name.json
$
$ # Imports the workspace file `file_name.json`.
$ jupyter lab workspaces import file_name.json
Saved workspace: <workspaces-directory>/labworkspacesfoo-54d5.jupyterlab-workspace
```

The `export` functionality is as friendly as possible: if a workspace does not exist, it will still generate an empty workspace for export.

The `import` functionality validates the structure of the workspace file and validates the `id` field in the workspace `metadata` to make sure its URL is compatible with either the `workspaces_url` configuration or the `page_url` configuration to verify that it is a correctly named workspace or it is the default workspace.

CHAPTER

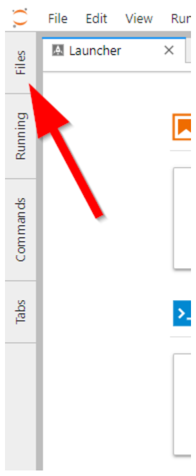
NINE

WORKING WITH FILES

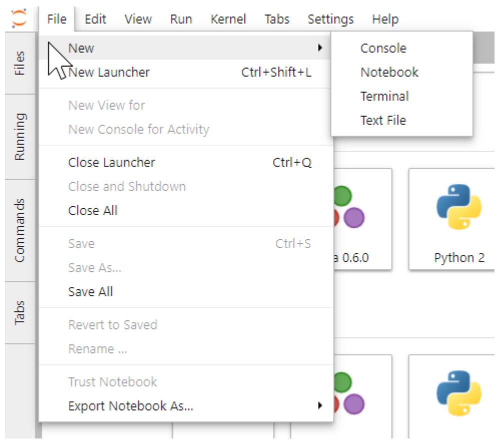
9.1 Opening Files

The file browser and File menu enable you to work with files and directories on your system. This includes opening, creating, deleting, renaming, downloading, copying, and sharing files and directories.

The file browser is in the left sidebar Files tab:



Many actions on files can also be carried out in the File menu:



To open any file, double-click on its name in the file browser:

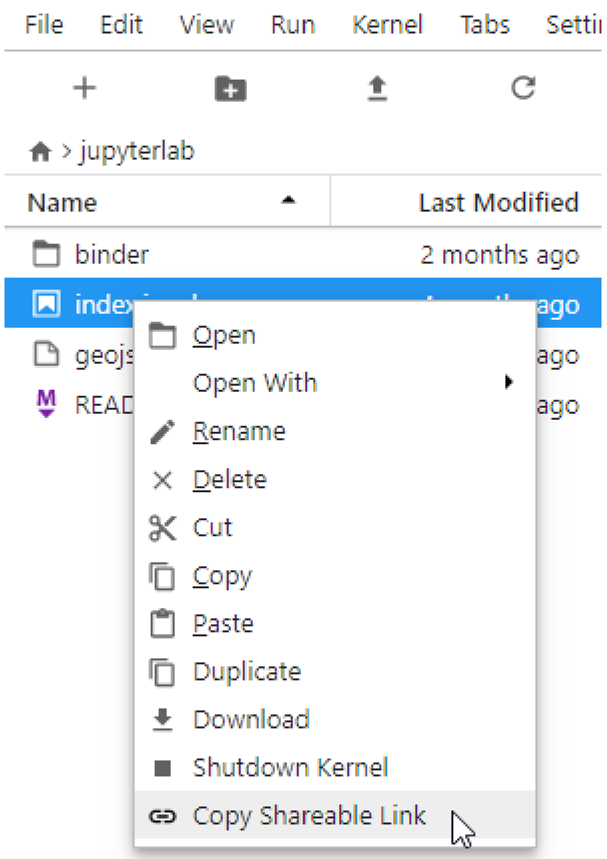
You can also drag a file into the main work area to create a new tab:

Many files types have *multiple viewers/editors*. For example, you can open a Markdown file in a *text editor* or as rendered HTML. A JupyterLab extension can also add new viewers/editors for files. To open a file in a nondefault viewer/editor, right-click on its name in the file browser and use the "Open With…" submenu to select the viewer/editor:

A single file can be open simultaneously in multiple viewer/editors and they will remain in sync:

The file system can be navigated by double-clicking on folders in the listing or clicking on the folders at the top of the directory listing:

Right-click on a file or directory and select "Copy Shareable Link" to copy a URL that can be used to open JupyterLab with that file or directory open.

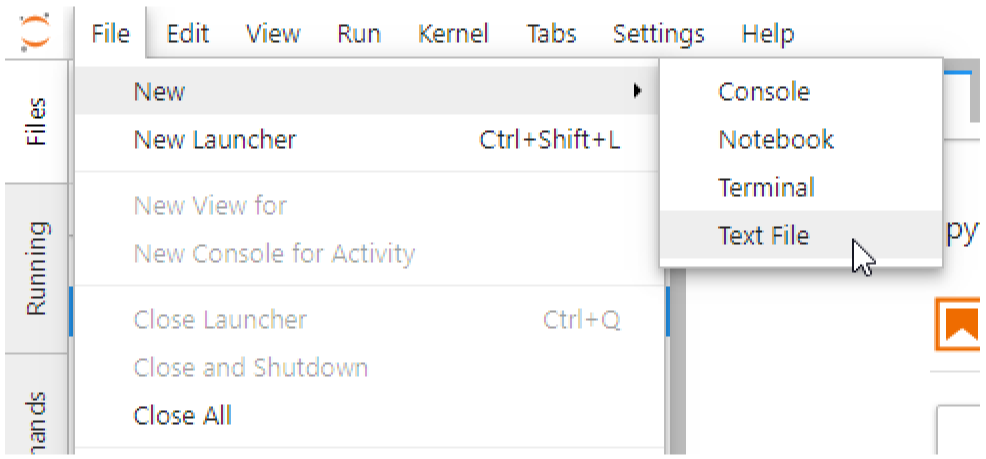


Right-click on a file or directory and select "Copy Path" to copy the filesystem relative path. This can be used for passing arguments to open files in functions called in various kernels.

9.2 Creating Files and Activities

Create new files or activities by clicking the + button at the top of the file browser. This will open a new Launcher tab in the main work area, which enables you to pick an activity and kernel:

You can also create new documents or activities using the File menu:



The current working directory of a new activity or document will be the directory listed in the file browser (except for a terminal, which always starts in the root directory of the file browser):

A new file is created with a default name. Rename a file by right-clicking on its name in the file browser and selecting “Rename” from the context menu:

9.3 Uploading and Downloading

Files can be uploaded to the current directory of the file browser by dragging and dropping files onto the file browser, or by clicking the “Upload Files” button at the top of the file browser:

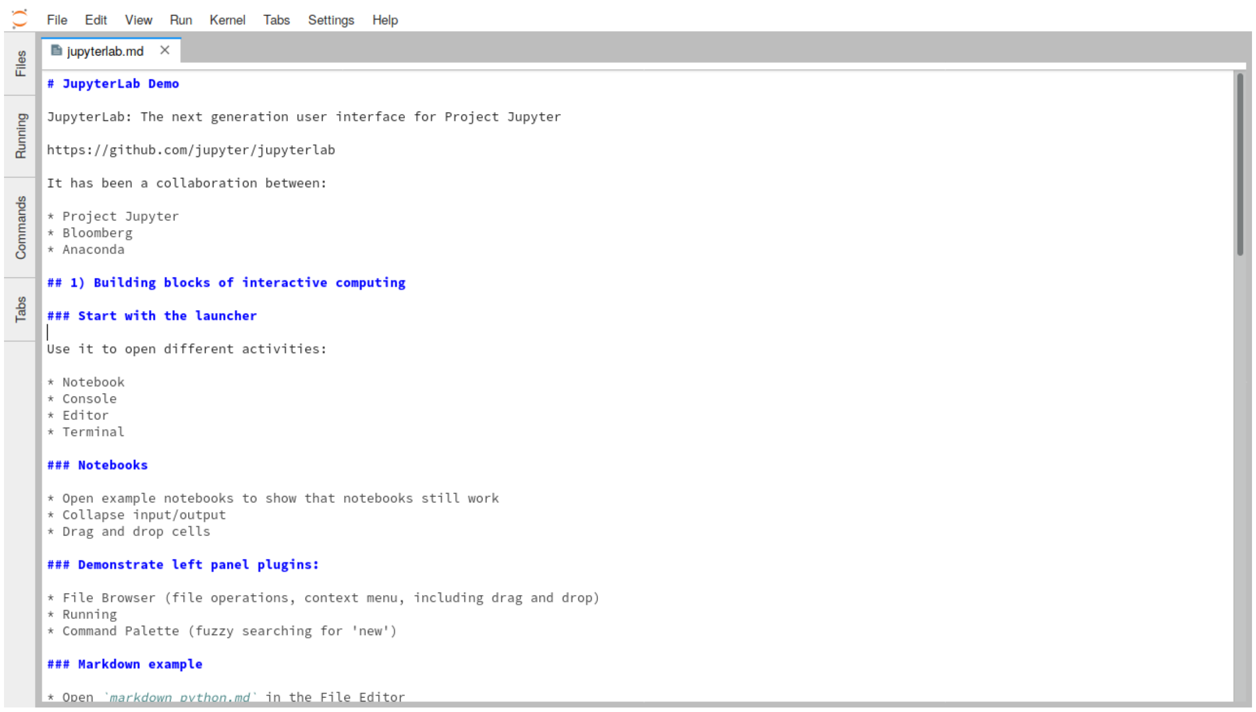
Any file in JupyterLab can be downloaded by right-clicking its name in the file browser and selecting “Download” from the context menu:

CHAPTER

TEN

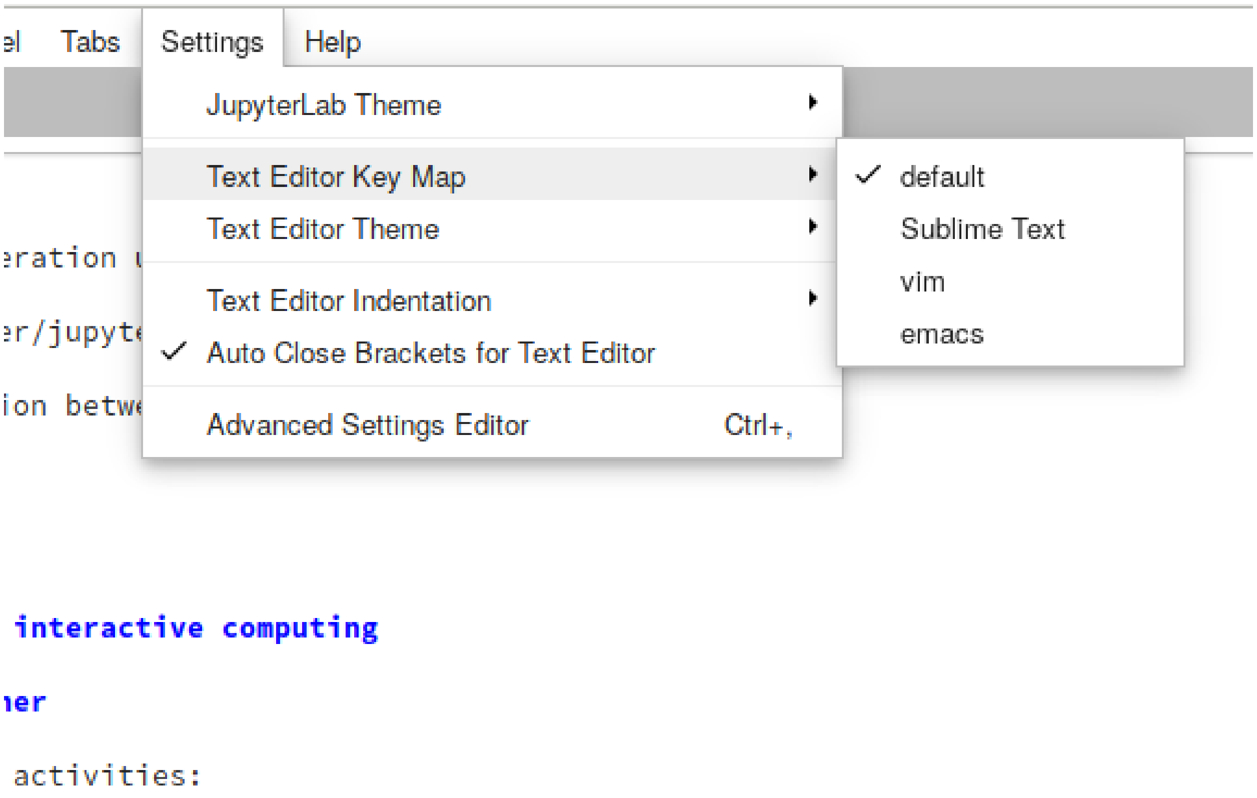
TEXT EDITOR

The text editor in JupyterLab enables you to edit text files in JupyterLab:



```
# JupyterLab Demo

JupyterLab: The next generation user interface for Project Jupyter

https://github.com/jupyter/jupyterlab

It has been a collaboration between:

* Project Jupyter
* Bloomberg
* Anaconda

## 1) Building blocks of interactive computing

### Start with the launcher

Use it to open different activities:

* Notebook
* Console
* Editor
* Terminal

### Notebooks

* Open example notebooks to show that notebooks still work
* Collapse input/output
* Drag and drop cells

### Demonstrate left panel plugins:

* File Browser (file operations, context menu, including drag and drop)
* Running
* Command Palette (fuzzy searching for 'new')

### Markdown example

* Open `markdown_python.md` in the File Editor
```

The text editor includes syntax highlighting, configurable indentation (tabs or spaces), *key maps* and basic theming. These settings can be found in the Settings menu:



To edit an existing text file, double-click on its name in the file browser or drag it into the main work area:

To create a new text file in the current directory of the file browser, click the + button at the top of the file browser to create a new Launcher tab, and click the “Text Editor” card in the Launcher:

You can also create a new text file with the File menu:

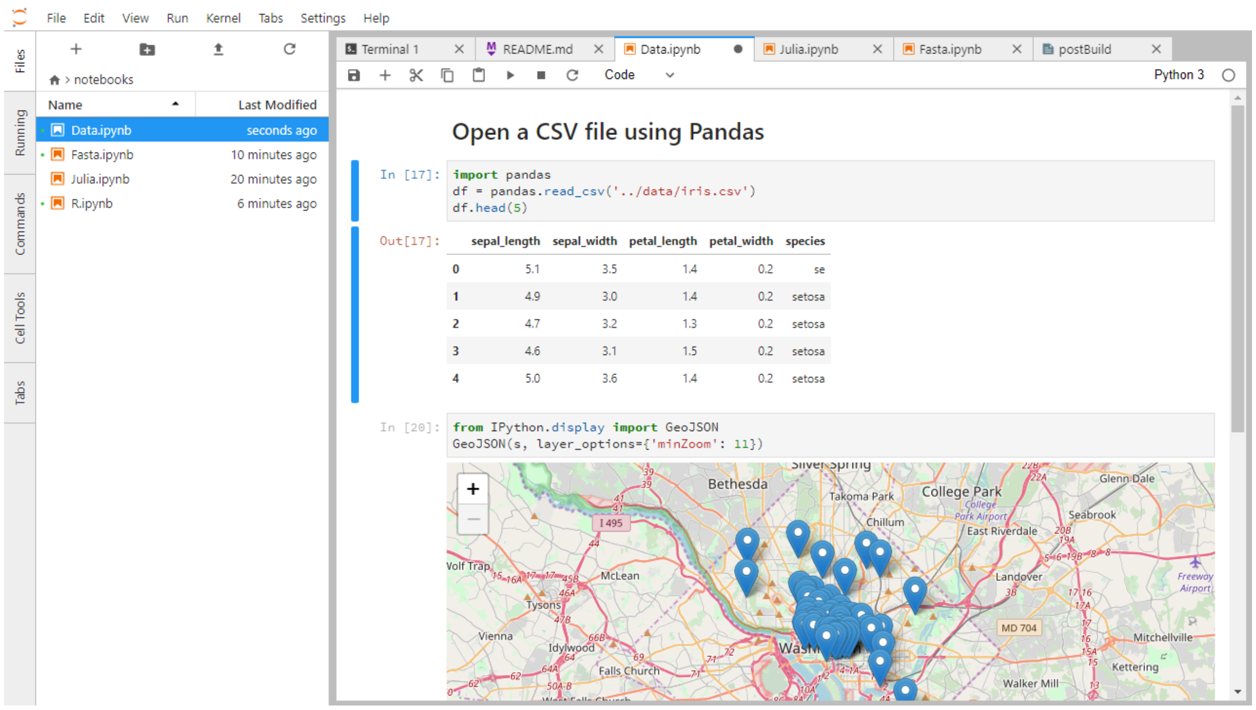
A new file is created with a default name. Rename a file by right-clicking on its name in the file browser and selecting “Rename” from the context menu:

CHAPTER

ELEVEN

NOTEBOOKS

Jupyter notebooks are documents that combine live runnable code with narrative text (Markdown), equations (LaTeX), images, interactive visualizations and other rich output:



Name	Last Modified
Data.ipynb	seconds ago
Fasta.ipynb	10 minutes ago
Julia.ipynb	20 minutes ago
R.ipynb	6 minutes ago

Open a CSV file using Pandas

```
In [17]: import pandas
         df = pandas.read_csv('../data/iris.csv')
         df.head(5)
```

Out[17]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	se
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
In [20]: from IPython.display import GeoJSON
         GeoJSON(s, layer_options={'minZoom': 11})
```

Jupyter notebooks (.ipynb files) are fully supported in JupyterLab. The notebook document format used in JupyterLab is the same as in the classic Jupyter Notebook. Your existing notebooks should open correctly in JupyterLab. If they don't, please open an issue on our GitHub issues page.

Create a notebook by clicking the + button in the file browser and then selecting a kernel in the new Launcher tab:

A new file is created with a default name. Rename a file by right-clicking on its name in the file browser and selecting "Rename" from the context menu:

The user interface for notebooks in JupyterLab closely follows that of the classic Jupyter Notebook. The keyboard shortcuts of the classic Notebook continue to work (with command and edit mode). However, a number of new things are possible with notebooks in JupyterLab.

Drag and drop cells to rearrange your notebook:

Drag cells between notebooks to quickly copy content:

Create multiple synchronized views of a single notebook:

Collapse and expand code and output using the View menu or the blue collapser button on left of each cell:

Enable scrolling for long outputs by right-clicking on a cell and selecting "Enable Scrolling for Outputs":

Create a new synchronized view of a cell's output:

Tab completion (activated with the `Tab` key) can now include additional information about the types of the matched items:

Note: IPython 6.3.1 has temporarily disabled type annotations. To re-enable them, add `c.Completer.use_jedi = True` to an ipython_config.py file.

The tooltip (activated with `Shift Tab`) contains additional information about objects:

You can connect a *code console* to a notebook kernel to have a log of computations done in the kernel, in the order in which they were done. The attached code console also provides a place to interactively inspect kernel state without changing the notebook. Right-click on a notebook and select "New Console for Notebook":

CHAPTER

TWELVE

CODE CONSOLES

Code consoles enable you to run code interactively in a kernel. The cells of a code console show the order in which code was executed in the kernel, as opposed to the explicit ordering of cells in a notebook document. Code consoles also display rich output, just like notebook cells.

Create a new code console by clicking the + button in the *file browser* and selecting the kernel:

Run code using `Shift Enter`. Use the up and down arrows to browse the history of previously-run code:

Tab completion (`Tab`) and tooltips (`Shift Tab`) work as in the notebook:

Clear the cells of the code console without restarting the kernel by right-clicking on the code console and selecting "Clear Console Cells":

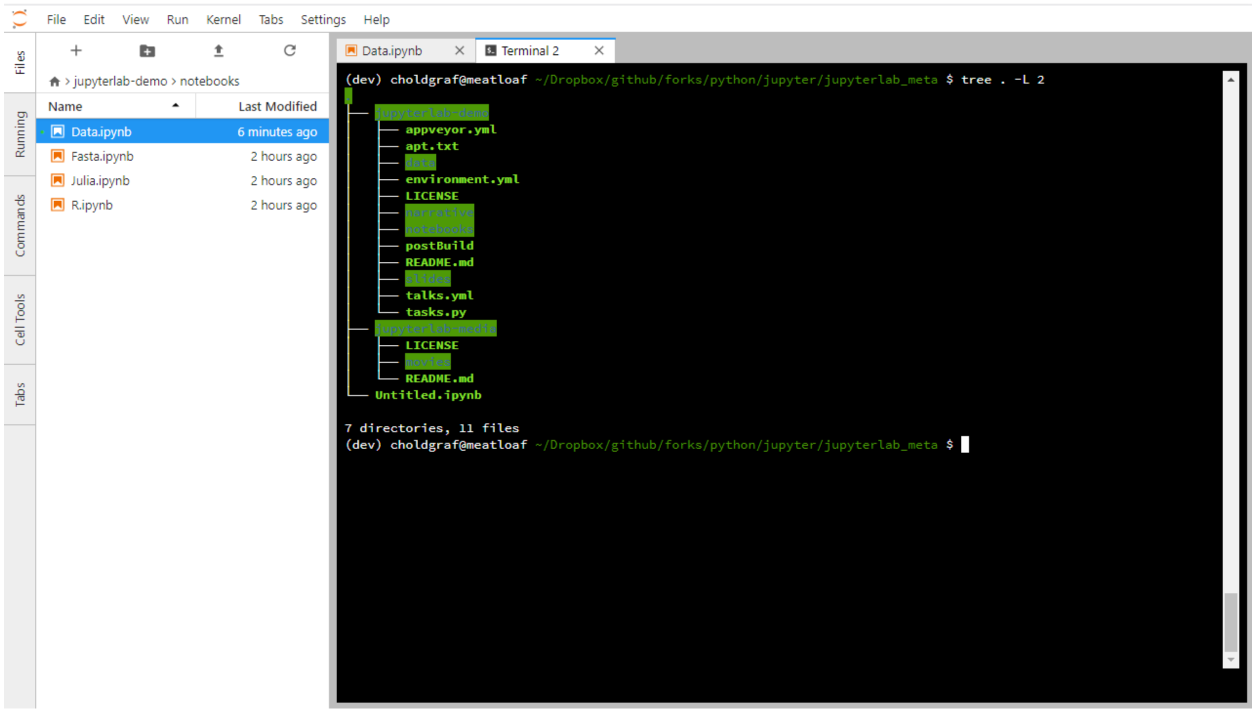
Creating a code console from the *file menu* lets you select an existing kernel for the code console. The code console then acts as a log of computations in that kernel, and a place you can interactively inspect and run code in the kernel:

CHAPTER

THIRTEEN

TERMINALS

JupyterLab terminals provide full support for system shells (bash, tsch, etc.) on Mac/Linux and PowerShell on Windows. You can run anything in your system shell with a terminal, including programs such as vim or emacs. The terminals run on the system where the Jupyter server is running, with the privileges of your user. Thus, if JupyterLab is installed on your local machine, the JupyterLab terminals will run there.



```
(dev) choldgraf@meatloaf ~/Dropbox/github/forks/python/jupyter/jupyterlab_meta $ tree . -L 2
.
├── jupyterlab-demo
│   ├── appveyor.yml
│   ├── apt.txt
│   ├── data
│   ├── environment.yml
│   ├── LICENSE
│   ├── narrative
│   ├── notebooks
│   ├── postBuild
│   ├── README.md
│   ├── slides
│   ├── talks.yml
│   └── tasks.py
├── jupyterlab-media
│   ├── LICENSE
│   ├── movies
│   └── README.md
└── Untitled.ipynb

7 directories, 11 files
(dev) choldgraf@meatloaf ~/Dropbox/github/forks/python/jupyter/jupyterlab_meta $
```

To open a new terminal, click the + button in the file browser and select the terminal in the new Launcher tab:

Closing a terminal tab will leave it running on the server, but you can re-open it using the Running tab in the left sidebar:

13.1 Copy/Paste

For macOS users, `Cmd+C` and `Cmd+V` work as usual.

For Windows users using `PowerShell`, `Ctrl+Insert` and `Shift+Insert` work as usual.

To use the native browser Copy/Paste menu, hold `Shift` and right click to bring up the context menu (note: this may not work in all browsers).

For non-macOS users, JupyterLab will interpret `Ctrl+C` as a copy if there is text selected. In addition, `Ctrl+V` will be interpreted as a paste command unless the `pasteWithCtrlV` setting is disabled. One may want to disable `pasteWithCtrlV` if the shortcut is needed for something else such as the vi editor.

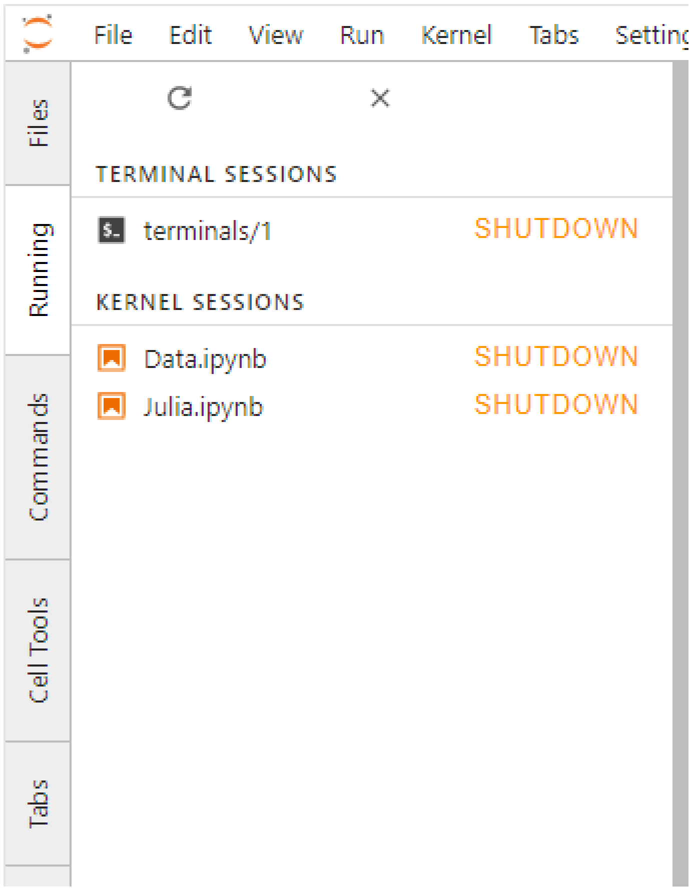
For anyone using a *nix shell, the default `Ctrl+Shift+C` conflicts with the default shortcut for toggling the command palette (`apputils:activate-command-palette`). If desired, that shortcut can be changed by editing the keyboard shortcuts in setttings. Using `Ctrl+Shift+V` for paste works as usual.

CHAPTER

FOURTEEN

MANAGING KERNELS AND TERMINALS

The Running panel in the left sidebar shows a list of all the kernels and terminals currently running across all notebooks, code consoles, and directories:



As with the classic Jupyter Notebook, when you close a notebook document, code console, or terminal, the underlying kernel or terminal running on the server continues to run. This enables you to perform long-running actions and return later. The Running panel enables you to re-open or focus the document linked to a given kernel or terminal:

Kernels or terminals can be shut down from the Running panel:

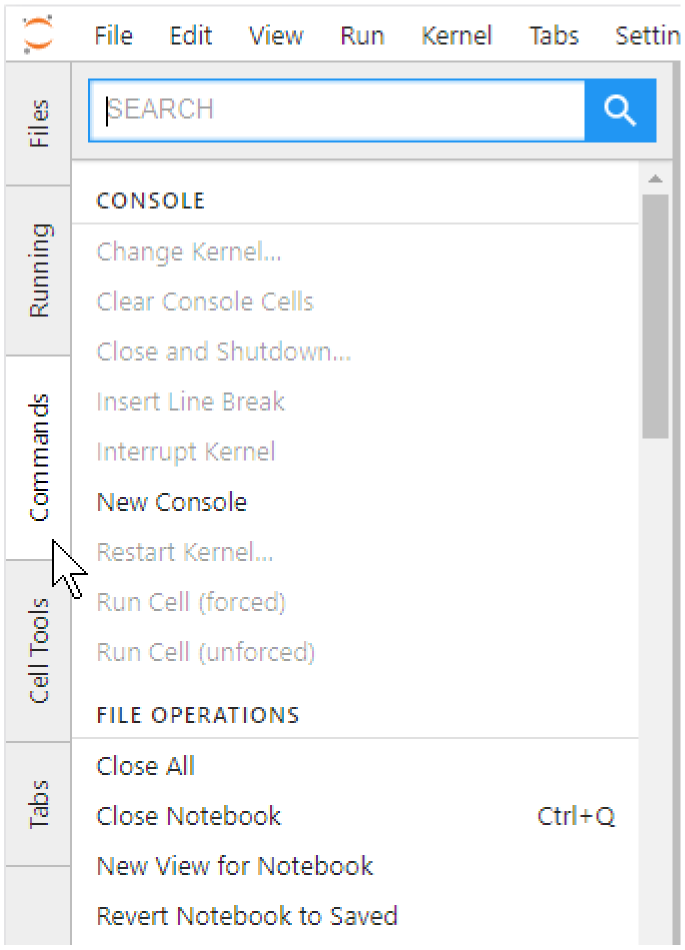
You can shut down all kernels and terminals by clicking the `X` button:

CHAPTER

FIFTEEN

COMMAND PALETTE

All user actions in JupyterLab are processed through a centralized command system. These commands are shared and used throughout JupyterLab (menu bar, context menus, keyboard shortcuts, etc.). The command palette in the left sidebar provides a keyboard-driven way to search for and run JupyterLab commands:



The command palette can be accessed using the keyboard shortcut `Command/Ctrl Shift C`:

CHAPTER

SIXTEEN

DOCUMENTS AND KERNELS

In the Jupyter architecture, kernels are separate processes started by the server that run your code in different programming languages and environments. JupyterLab enables you to connect any open text file to a *code console and kernel*. This means you can easily run code from the text file in the kernel interactively.

Right-click on a document and select “Create Console for Editor”:

Once the code console is open, send a single line of code or select a block of code and send it to the code console by hitting `Shift Enter`:

In a Markdown document, `Shift Enter` will automatically detect if the cursor is within a code block, and run the entire block if there is no selection:

Any text file (Markdown, Python, R, LaTeX, C++, etc.) in a text file editor can be connected to a code console and kernel in this manner.

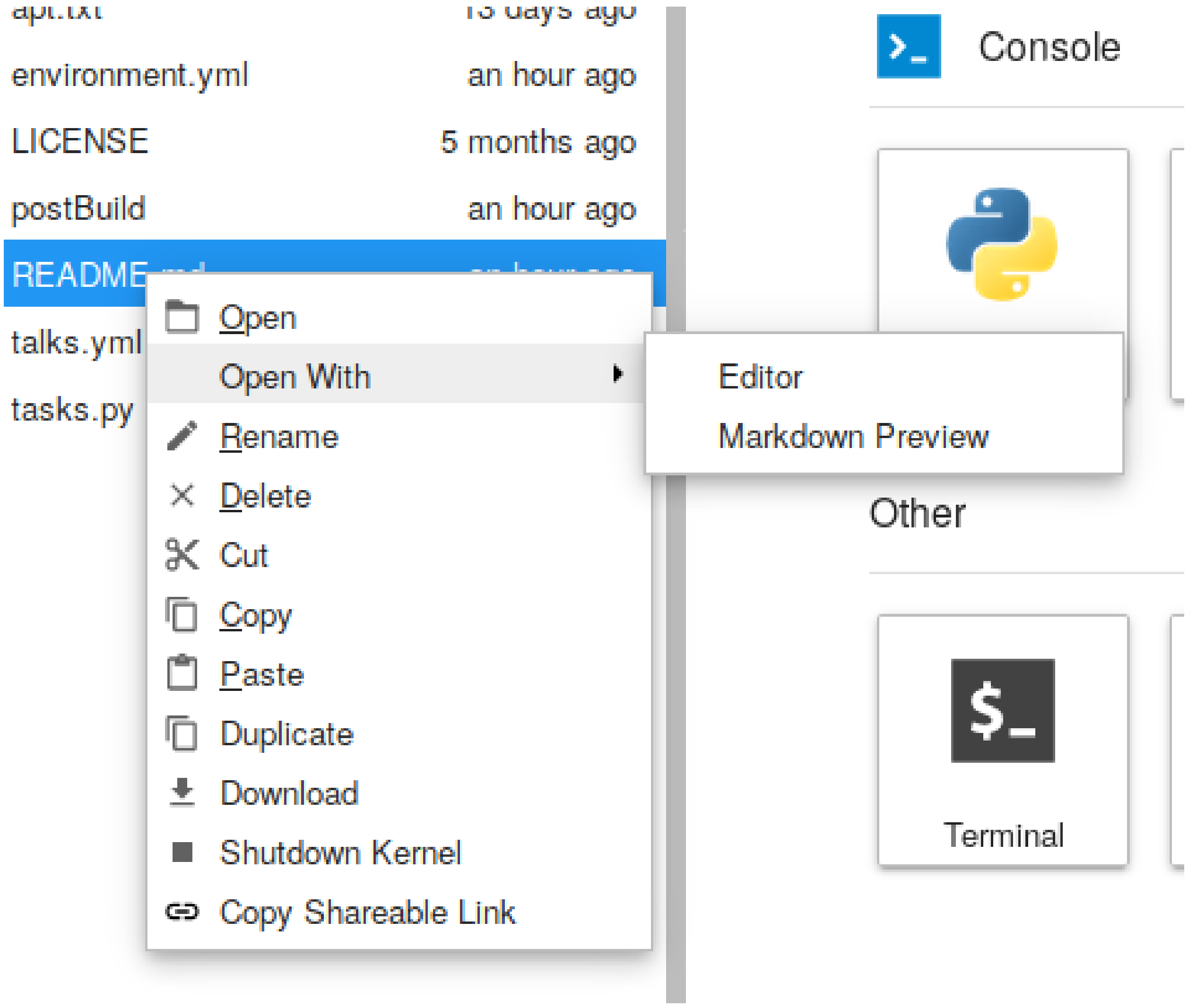
CHAPTER

SEVENTEEN

FILE AND OUTPUT FORMATS

JupyterLab provides a unified architecture for viewing and editing data in a wide variety of formats. This model applies whether the data is in a file or is provided by a kernel as rich cell output in a notebook or code console.

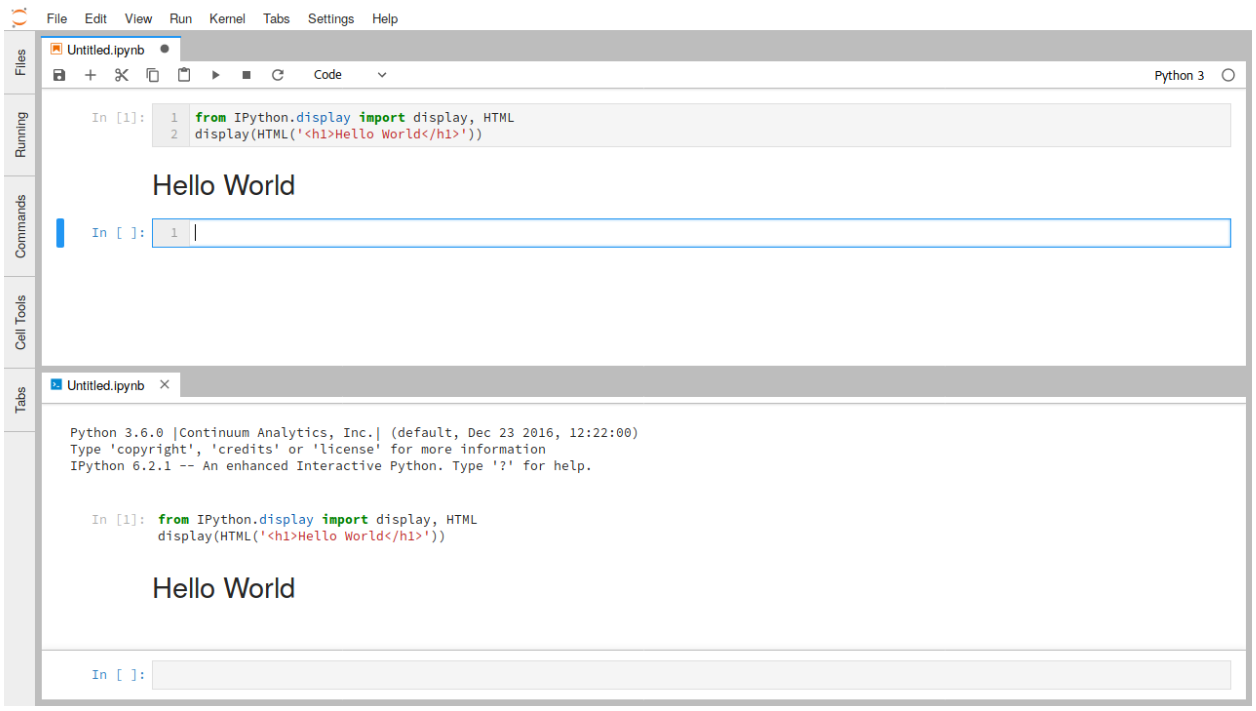
For files, the data format is detected by the extension of the file (or the whole filename if there is no extension). A single file extension may have multiple editors or viewers registered. For example, a Markdown file (`.md`) can be edited in the file editor or rendered and displayed as HTML. You can open different editors and viewers for a file by right-clicking on the filename in the file browser and using the "Open With" submenu:



To use these different data formats as output in a notebook or code console, you can use the relevant display API for the kernel you are using. For example, the IPython kernel provides a variety of convenience classes for displaying rich output:

```
from IPython.display import display, HTML
display(HTML('<h1>Hello World</h1>'))
```

Running this code will display the HTML in the output of a notebook or code console cell:



```
In [1]: from IPython.display import display, HTML
        display(HTML('<h1>Hello World</h1>'))
```

Hello World

```
In [ ]:
```

```
Python 3.6.0 |Continuum Analytics, Inc.| (default, Dec 23 2016, 12:22:00)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from IPython.display import display, HTML
        display(HTML('<h1>Hello World</h1>'))
```

Hello World

```
In [ ]:
```

The IPython display function can also construct a raw rich output message from a dictionary of keys (MIME types) and values (MIME data):

```
from IPython.display import display
display({'text/html': '<h1>Hello World</h1>', 'text/plain': 'Hello World'}, raw=True)
```

Other Jupyter kernels offer similar APIs.

The rest of this section highlights some of the common data formats that JupyterLab supports by default. JupyterLab extensions can also add support for other file formats.

17.1 Markdown

- File extension: `.md`
- MIME type: `text/markdown`

Markdown is a simple and popular markup language used for text cells in the Jupyter Notebook.

Markdown documents can be edited as text files or rendered inline:

The Markdown syntax supported in this mode is the same syntax used in the Jupyter Notebook (for example, LaTeX equations work). As seen in the animation, edits to the Markdown source are immediately reflected in the rendered version.

17.2 Images

- File extensions: `.bmp`, `.gif`, `.jpeg`, `.jpg`, `.png`, `.svg`
- MIME types: `image/bmp`, `image/gif`, `image/jpeg`, `image/png`, `image/svg+xml`

JupyterLab supports image data in cell output and as files in the above formats. In the image file viewer, you can use keyboard shortcuts such as `+` and `-` to zoom the image, `[` and `]` to rotate the image, and `H` and `V` to flip the image horizontally and vertically. Use `I` to invert the colors, and use `0` to reset the image.

To edit an SVG image as a text file, right-click on the SVG filename in the file browser and select the "Editor" item in the "Open With" submenu:

17.3 Delimiter-separated Values

- File extension: `.csv`
- MIME type: None

Files with rows of delimiter-separated values, such as CSV files, are a common format for tabular data. The default viewer for these files in JupyterLab is a high-performance data grid viewer which can display comma-separated, tab-separated, and semicolon-separated values:

While tab-separated value files can be read by the grid viewer, it currently does not automatically recognize `.tsv` files. To view, you must change the extension to `.csv` and set the delimiter to tabs.

To edit a CSV file as a text file, right-click on the file in the file browser and select the "Editor" item in the "Open With" submenu:

JupyterLab's grid viewer can open large files, up to the maximum string size of the particular browser. Below is a table that shows the sizes of the largest test files we successfully opened in each browser we support:

Browser	Max Size
Firefox	1.04GB
Chrome	730MB
Safari	1.8GB

The actual maximum size of files that can be successfully loaded will vary depending on the browser version and file content.

17.4 JSON

- File extension: `.json`
- MIME type: `application/json`

JavaScript Object Notation (JSON) files are common in data science. JupyterLab supports displaying JSON data in cell output or viewing a JSON file using a searchable tree view:

To edit the JSON as a text file, right-click on the filename in the file browser and select the "Editor" item in the "Open With" submenu:

17.5 HTML

- File extension: `.html`
- MIME type: `text/html`

JupyterLab supports rendering HTML in cell output and editing HTML files as text in the file editor.

17.6 LaTeX

- File extension: `.tex`
- MIME type: `text/latex`

JupyterLab supports rendering LaTeX equations in cell output and editing LaTeX files as text in the file editor.

17.7 PDF

- File extension: `.pdf`
- MIME type: `application/pdf`

PDF is a common standard file format for documents. To view a PDF file in JupyterLab, double-click on the file in the file browser:

17.8 Vega/Vega-Lite

Vega:

- File extensions: `.vg`, `.vg.json`
- MIME type: `application/vnd.vega.v2+json`

Vega-Lite:

- File extensions: `.vl`, `.vl.json`
- MIME type: `application/vnd.vegalite.v1+json`

Vega and Vega-Lite are declarative visualization grammars that enable visualizations to be encoded as JSON data. For more information, see the documentation of Vega or Vega-Lite. JupyterLab supports rendering Vega 2.x and Vega-Lite 1.x data in files and cell output.

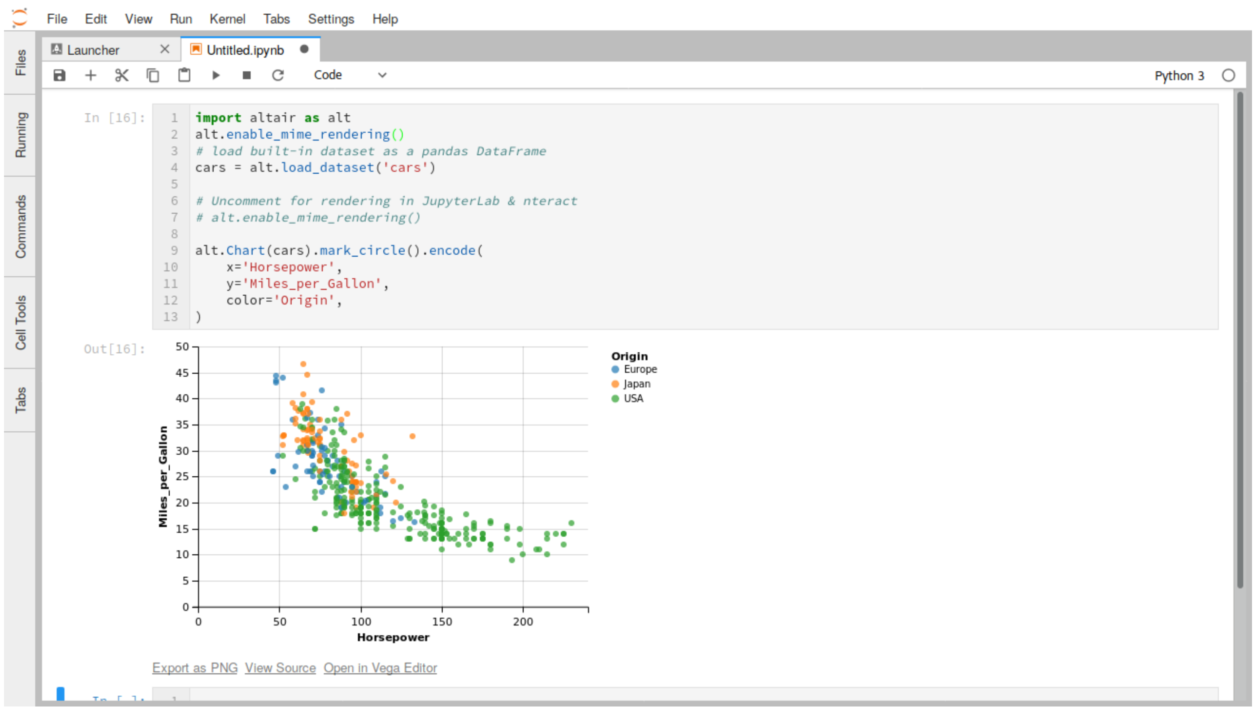
Vega-Lite 1.x files, with a `.vl` or `.vl.json` file extension, can be opened by double-clicking the file in the file browser:

The files can also be opened in the JSON viewer or file editor through the "Open With…" submenu in the file browser content menu:

As with other files in JupyterLab, multiple views of a single file remain synchronized, enabling you to interactively edit and render Vega/Vega-Lite visualizations:

The same workflow also works for Vega 2.x files, with a `.vg` or `.vg.json` file extension.

Output support for Vega/Vega-Lite in a notebook or code console is provided through third-party libraries such as Altair (Python), the vegalite R package, or Vegas (Scala/Spark).



```
In [16]:
import altair as alt
alt.enable_mime_rendering()
# load built-in dataset as a pandas DataFrame
cars = alt.load_dataset('cars')

# Uncomment for rendering in JupyterLab & nteract
# alt.enable_mime_rendering()

alt.Chart(cars).mark_circle().encode(
    x='Horsepower',
    y='Miles_per_Gallon',
    color='Origin',
)
Out[16]:
```

A JupyterLab extension that supports Vega 3.x and Vega-Lite 2.x can be found here.

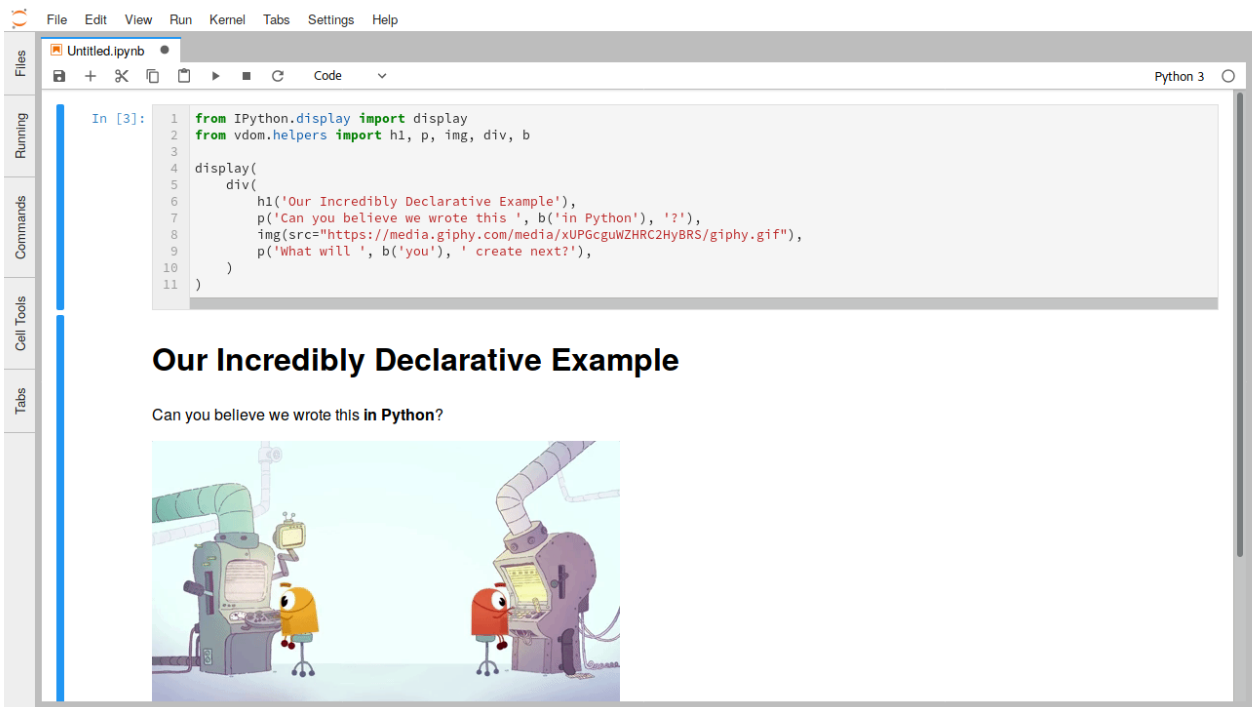
17.9 Virtual DOM

- File extensions: `.vdom`, `.json`
- MIME type: `application/vdom.v1+json`

Virtual DOM libraries such as react.js have greatly improved the experience of rendering interactive content in HTML. The nteract project, which collaborates closely with Project Jupyter, has created a declarative JSON format for virtual DOM data. JupyterLab can render this data using react.js. This works for both VDOM files with the `.vdom` extension, or within notebook output.

Here is an example of a `.vdom` file being edited and rendered interactively:

The nteract/vdom library provides a Python API for creating VDOM output that is rendered in nteract and JupyterLab:



```
from IPython.display import display
from vdom.helpers import h1, p, img, div, b

display(
    div(
        h1('Our Incredibly Declarative Example'),
        p('Can you believe we wrote this ', b('in Python'), '?'),
        img(src="https://media.giphy.com/media/xUPGcguWZHRC2HyBRS/giphy.gif"),
        p('What will ', b('you'), ' create next?'),
    )
)
```

Our Incredibly Declarative Example

Can you believe we wrote this **in Python**?

CHAPTER

EIGHTEEN

EXTENSIONS

Fundamentally, JupyterLab is designed as an extensible environment. JupyterLab extensions can customize or enhance any part of JupyterLab. They can provide new themes, file viewers and editors, or renderers for rich outputs in notebooks. Extensions can add items to the menu or command palette, keyboard shortcuts, or settings in the settings system. Extensions can provide an API for other extensions to use and can depend on other extensions. In fact, the whole of JupyterLab itself is simply a collection of extensions that are no more powerful or privileged than any custom extension.

JupyterLab extensions are npm packages (the standard package format in Javascript development). You can search for the keyword jupyterlab-extension on the npm registry to find extensions. For information about developing extensions, see the *developer documentation*.

Note: If you are a JupyterLab extension developer, please note that the extension developer API is not stable and will evolve in the near future.

In order to install JupyterLab extensions, you need to have Node.js installed.

If you use `conda`, you can get it with:

```
conda install -c conda-forge nodejs
```

If you use Homebrew on Mac OS X:

```
brew install node
```

You can also download Node.js from the Node.js website and install it directly.

18.1 Using the Extension Manager

To manage your extensions, you can use the extension manager. By default, the manager is disabled. You can enable it by searching **Extension Manager** in the command palette.

You can also enable it with the following steps:

- Go into advanced settings editor.
- Open the Extension Manager section.
- Add the entry “enabled”: true.
- Save the settings.
- If prompted whether you are sure, read the warning, and click “Enable” if you are still sure.

Once enabled, you should see a new tab appear in the *left sidebar*.

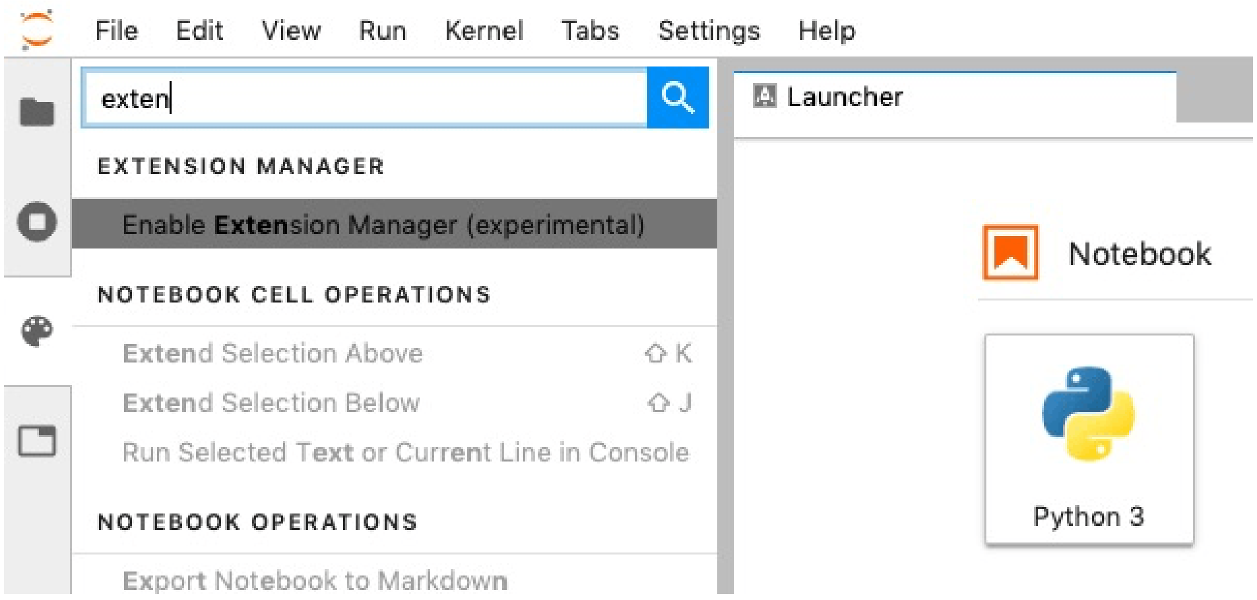

File
Edit
View
Run
Kernel
Tabs
Settings
Help
exten
EXTENSION MANAGER
Enable Extension Manager (experimental)
NOTEBOOK CELL OPERATIONS
Extend Selection Above ⇧ K
Extend Selection Below ⇧ J
Run Selected Text or Current Line in Console
NOTEBOOK OPERATIONS
Export Notebook to Markdown
Launcher
Notebook
Python 3

Fig. 1: **Figure:** Enable extension manager by searching in the command palette

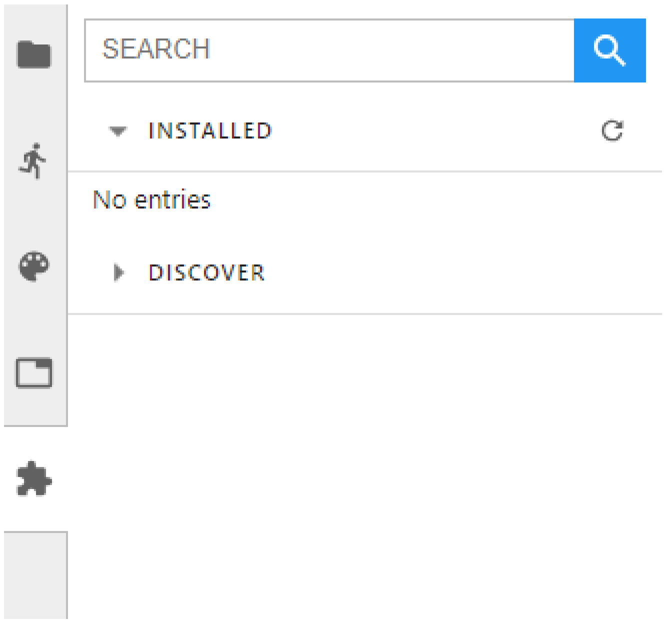
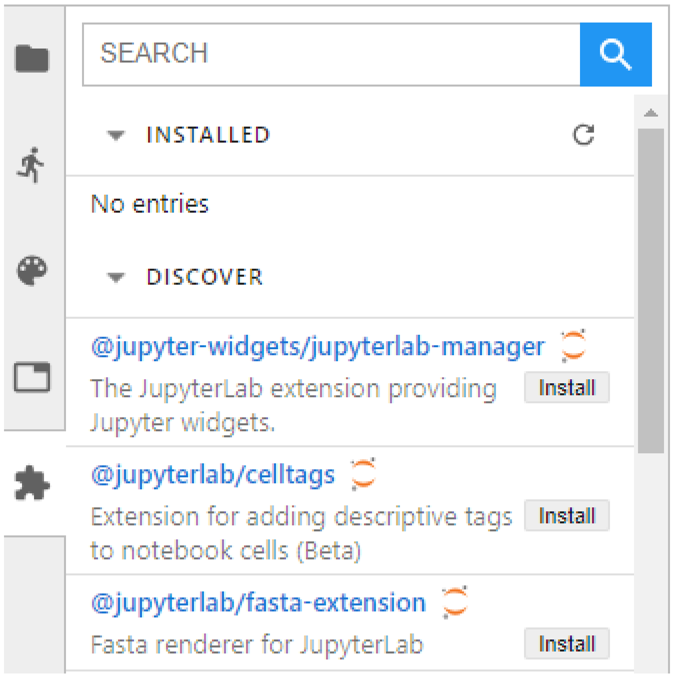


Fig. 2: **Figure:** The default view has three components: a search bar, an "Installed" section, and a "Discover" section.

18.1.1 Finding Extensions

You can use the extension manager to find extensions for JupyterLab. To discovery freely among the currently available extensions, expand the "Discovery" section. This triggers a search for all JupyterLab extensions on the NPM registry, and the results are listed according to the registry's sort order. An exception to this sort order is that extensions released by the Jupyter organization are always placed first. These extensions are distinguished by a small Jupyter icon next to their name.



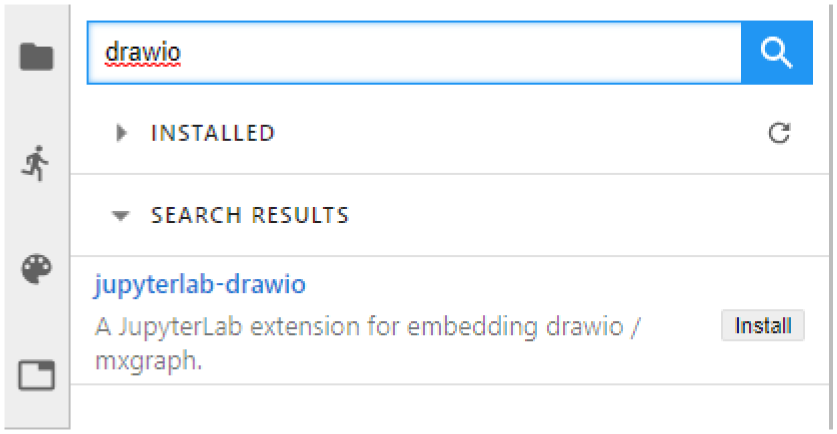
No entries

@jupyter-widgets/jupyterlab-manager
The JupyterLab extension providing Jupyter widgets.

@jupyterlab/celltags
Extension for adding descriptive tags to notebook cells (Beta)

@jupyterlab/fasta-extension
Fasta renderer for JupyterLab

Alternatively, you can limit your discovery by using the search bar. This performs a free-text search of JupyterLab extensions on the NPM registry.



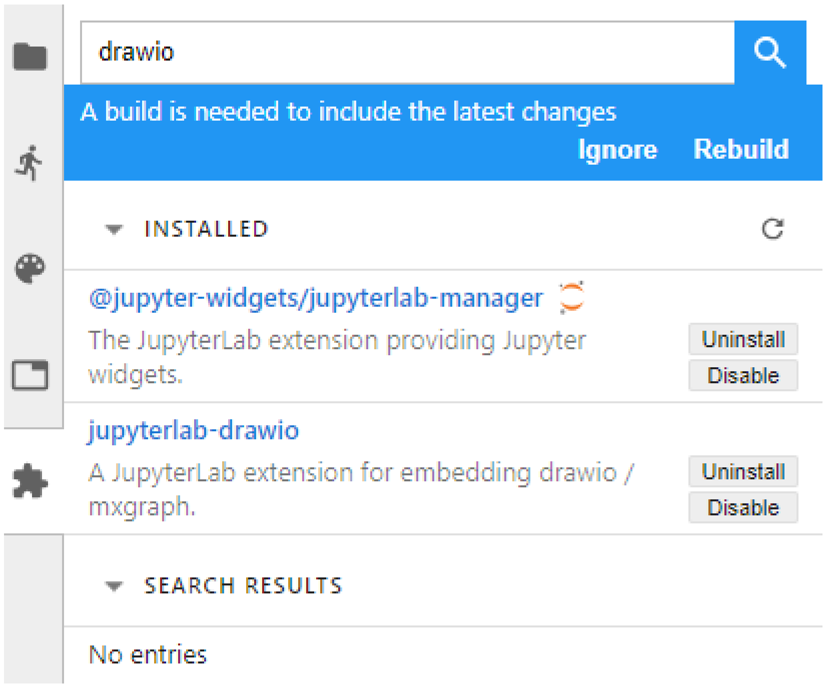
jupyterlab-drawio
A JupyterLab extension for embedding drawio / mxgraph.

18.1.2 Installing an Extension

Once you have found an extension that you think is interesting, install it by clicking the "Install" button of the extension list entry.

Danger: Installing an extension allows it to execute arbitrary code on the server, kernel, and in the client's browser. You should therefore avoid installing extensions you do not trust, and watch out for any extensions trying to masquerade as a trusted extension.

A short while after starting the install of an extension, a drop-down should appear under the search bar indicating that the extension has been downloaded, but that a rebuild is needed to complete the installation.


drawio

A build is needed to include the latest changes

@jupyter-widgets/jupyterlab-manager

The JupyterLab extension providing Jupyter widgets.

jupyterlab-drawio

A JupyterLab extension for embedding drawio / mxgraph.

No entries

If you want to install/uninstall other extensions as well, you can ignore the rebuild notice until you have made all the changes you want. Once satisfied, click the 'Rebuild' button to start a rebuild in the background. Once the rebuild completes, a dialog will pop up, indicating that a reload of the page is needed in order to load the latest build into the browser.

If you ignore the rebuild notice by mistake, simply refresh your browser window to trigger a new rebuild check.

18.1.3 Managing Installed Extensions

When there are some installed extensions, they will be shown in the "Installed" section. These can then be uninstalled or disabled. Disabling an extension will prevent it from being activated, but without rebuilding the application.

18.1.4 Companion packages

During installation of an extension, JupyterLab will inspect the package metadata for any *instructions on companion packages*. Companion packages can be:

- Notebook server extensions (or any other packages that need to be installed on the Notebook server).
- Kernel packages. An example of companion packages for the kernel are Jupyter Widget packages, like the ipywidgets Python package for the @jupyter-widgets/jupyterlab-manager package.

If JupyterLab finds instructions for companion packages, it will present a dialog to notify you about these. These are informational only, and it will be up to you to take these into account or not.

18.2 Using the Terminal

Another way of managing your extensions is from the terminal on the server, using the `jupyter labextension` entry point. In general, a simple help text is available by typing `jupyter labextension --help`.

18.2.1 Installing Extensions

You can install new extensions into the application using the command:

```
jupyter labextension install my-extension
```

where `my-extension` is the name of a valid JupyterLab extension npm package on npm. Use the `my-extension@version` syntax to install a specific version of an extension, for example:

```
jupyter labextension install my-extension@1.2.3
```

You can also install an extension that is not uploaded to npm, i.e., `my-extension` can be a local directory containing the extension, a gzipped tarball, or a URL to a gzipped tarball.

We encourage extension authors to add the `jupyterlab-extension` GitHub topic to any repository with a JupyterLab extension to facilitate discovery. You can see a list of extensions by searching GitHub for the jupyterlab-extension topic.

You can list the currently installed extensions by running the command:

```
jupyter labextension list
```

Uninstall an extension by running the command:

```
jupyter labextension uninstall my-extension
```

where `my-extension` is the name of the extension, as printed in the extension list. You can also uninstall core extensions using this command (you can always re-install core extensions later).

Installing and uninstalling extensions can take some time, as they are downloaded, bundled with the core extensions, and the whole application is rebuilt. You can install/uninstall more than one extension in the same command by listing their names after the `install` command.

If you are installing/uninstalling several extensions in several stages, you may want to defer rebuilding the application by including the flag `--no-build` in the install/uninstall step. Once you are ready to rebuild, you can run the command:

```
jupyter lab build
```

Note If using Windows, you may encounter a *FileNotFoundError* due to the default PATH length on Windows. Node modules are stored in a nested file structure, so the path can get quite long. If you have administrative access and are on Windows 8 or 10, you can update the registry setting using these instructions: https://stackoverflow.com/a/37528731.

18.2.2 Disabling Extensions

You can disable specific JupyterLab extensions (including core extensions) without rebuilding the application by running the command:

```
jupyter labextension disable my-extension
```

This will prevent the extension from loading in the browser, but does not require a rebuild.

You can re-enable an extension using the command:

```
jupyter labextension enable my-extension
```

18.3 Advanced Usage

Any information that JupyterLab persists is stored in its application directory, including settings and built assets. This is separate from where the Python package is installed (like in `site_packages`) so that the install directory is immutable.

The application directory can be overridden using `--app-dir` in any of the JupyterLab commands, or by setting the `JUPYTERLAB_DIR` environment variable. If not specified, it will default to `<sys-prefix>/share/jupyter/lab`, where `<sys-prefix>` is the site-specific directory prefix of the current Python environment. You can query the current application path by running `jupyter lab path`. Note that the application directory is expected to contain the JupyterLab static assets (e.g. *static/index.html*). If JupyterLab is launched and the static assets are not present, it will display an error in the console and in the browser.

18.3.1 JupyterLab Build Process

To rebuild the app directory, run `jupyter lab build`. By default, the `jupyter labextension install` command builds the application, so you typically do not need to call `build` directly.

Building consists of:

- Populating the `staging/` directory using template files
- Handling any locally installed packages
- Ensuring all installed assets are available
- Bundling the assets
- Copying the bundled assets to the `static` directory

Note that building will always use the latest JavaScript packages that meet the dependency requirements of JupyterLab itself and any installed extensions. If you wish to run JupyterLab with the set of pinned requirements that was shipped with the Python package, you can launch as `jupyter lab --core-mode`.

Note

The build process uses a specific `yarn` version with a default working combination of npm packages stored in a `yarn.lock` file shipped with JupyterLab. Those package source urls point to the default yarn registry. But if you defined your own yarn registry in yarn configuration, the default yarn registry will be replaced by your custom registry.

If then you switch back to the default yarn registry, you will need to clean your `staging` folder before building:

```
jupyter lab clean
jupyter lab build
```

18.3.2 JupyterLab Application Directory

The JupyterLab application directory contains the subdirectories `extensions`, `schemas`, `settings`, `staging`, `static`, and `themes`. The default application directory mirrors the location where JupyterLab was installed. For example, in a conda environment, it is in `<conda_root>/envs/<env_name>/share/jupyter/lab`. The directory can be overridden by setting a `JUPYTERLAB_DIR` environment variable.

It is not recommended to install JupyterLab in a root location (on Unix-like systems). Instead, use a conda environment or `pip install --user jupyterlab` so that the application directory ends up in a writable location.

Note: this folder location and semantics do *not* follow the standard Jupyter config semantics because we need to build a single unified application, and the default config location for Jupyter is at the user level (user's home directory). By explicitly using a directory alongside the currently installed JupyterLab, we can ensure better isolation between conda or other virtual environments.

extensions

The `extensions` directory has the packed tarballs for each of the installed extensions for the app. If the application directory is not the same as the `sys-prefix` directory, the extensions installed in the `sys-prefix` directory will be used in the app directory. If an extension is installed in the app directory that exists in the `sys-prefix` directory, it will shadow the `sys-prefix` version. Uninstalling an extension will first uninstall the shadowed extension, and then attempt to uninstall the `sys-prefix` version if called again. If the `sys-prefix` version cannot be uninstalled, its plugins can still be ignored using `ignoredPackages` metadata in `settings`.

schemas

The `schemas` directory contains JSON Schemas that describe the settings used by individual extensions. Users may edit these settings using the JupyterLab Settings Editor.

settings

The `settings` directory may contain `page_config.json`, `overrides.json`, and/or `build_config.json` files, depending on which configurations are set on your system.

page_config.json

The `page_config.json` data is used to provide configuration data to the application environment.

The following configurations may be present in this file:

1. `terminalsAvailable` identifies whether a terminal (i.e. `bash/tsch` on Mac/Linux OR `PowerShell` on Windows) is available to be launched via the Launcher. (This configuration was predominantly required for Windows prior to PowerShell access being enabled in Jupyter Lab.) The value for this field is a Boolean: `true` or `false`.
2. `disabledExtensions` controls which extensions should not load at all.
3. `deferredExtensions` controls which extensions should not load until they are required by something, irrespective of whether they set `autoStart` to `true`.

The value for the `disabledExtensions` and `deferredExtensions` fields are an array of strings. The following sequence of checks are performed against the patterns in `disabledExtensions` and `deferredExtensions`.

- If an identical string match occurs between a config value and a package name (e.g., `"@jupyterlab/apputils-extension"`), then the entire package is disabled (or deferred).
- If the string value is compiled as a regular expression and tests positive against a package name (e.g., `"disabledExtensions": ["@jupyterlab/apputils*$"]`), then the entire package is disabled (or deferred).

- If an identical string match occurs between a config value and an individual plugin ID within a package (e.g., `"disabledExtensions": ["@jupyterlab/apputils-extension:settings"]`), then that specific plugin is disabled (or deferred).
- If the string value is compiled as a regular expression and tests positive against an individual plugin ID within a package (e.g., `"disabledExtensions": ["^@jupyterlab/apputils-extension:set.*$"]`), then that specific plugin is disabled (or deferred).

An example of a `page_config.json` file is:

```
{
    "disabledExtensions": [
        "@jupyterlab/toc"
    ],
    "terminalsAvailable": false
}
```

overrides.json

You can override default values of the extension settings by defining new default values in an `overrides.json` file. So for example, if you would like to set the dark theme by default instead of the light one, an `overrides.json` file containing the following lines needs to be added in the application settings directory (by default this is the `share/jupyter/lab/settings` folder).

```
{
  "@jupyterlab/apputils-extension:themes": {
    "theme": "JupyterLab Dark"
  }
}
```

build_config.json

The `build_config.json` file is used to track the local directories that have been installed using `jupyter labextension install <directory>`, as well as core extensions that have been explicitly uninstalled. An example of a `build_config.json` file is:

```
{
    "uninstalled_core_extensions": [
        "@jupyterlab/markdownwidget-extension"
    ],
    "local_extensions": {
        "@jupyterlab/python-tests": "/path/to/my/extension"
    }
}
```

staging and static

The `static` directory contains the assets that will be loaded by the JupyterLab application. The `staging` directory is used to create the build and then populate the `static` directory.

Running `jupyter lab` will attempt to run the `static` assets in the application directory if they exist. You can run `jupyter lab --core-mode` to load the core JupyterLab application (i.e., the application without any extensions) instead.

themes

The `themes` directory contains assets (such as CSS and icons) for JupyterLab theme extensions.

18.3.3 JupyterLab User Settings Directory

The user settings directory contains the user-level settings for Jupyter extensions. By default, the location is `~/.jupyter/lab/user-settings/`, where `~` is the user's home directory. This folder is not in the JupyterLab application directory, because these settings are typically shared across Python environments. The location can be modified using the `JUPYTERLAB_SETTINGS_DIR` environment variable. Files are automatically created in this folder as modifications are made to settings from the JupyterLab UI. They can also be manually created. The files follow the pattern of `<package_name>/<extension_name>.jupyterlab-settings`. They are JSON files with optional comments. These values take precedence over the default values given by the extensions, but can be overridden by an `overrides.json` file in the application's settings directory.

18.4 JupyterLab Workspaces Directory

JupyterLab sessions always reside in a workspace. Workspaces contain the state of JupyterLab: the files that are currently open, the layout of the application areas and tabs, etc. When the page is refreshed, the workspace is restored. By default, the location is `~/.jupyter/lab/workspacess/`, where `~` is the user's home directory. This folder is not in the JupyterLab application directory, because these files are typically shared across Python environments. The location can be modified using the `JUPYTERLAB_WORKSPACES_DIR` environment variable. These files can be imported and exported to create default "profiles", using the *workspace command line tool*.

CHAPTER

NINETEEN

JUPYTERLAB ON JUPYTERHUB

JupyterLab works out of the box with JupyterHub, and can even run side by side with the classic Notebook.

When JupyterLab is deployed with JupyterHub it will show additional menu items in the File menu that allow the user to log out or go to the JupyterHub control panel.

19.1 Use JupyterLab by Default

If you install JupyterLab on a system running JupyterHub, it will immediately be available at the `/lab` URL, but users will still be directed to the classic Notebook (`/tree`) by default. To change the user's default user interface to JupyterLab, set the following configuration option in your `jupyterhub_config.py` file:

```
c.Spawner.default_url = '/lab'
```

In this configuration, users can still access the classic Notebook at `/tree`, by either typing that URL into the browser, or by using the "Launch Classic Notebook" item in JupyterLab's Help menu.

19.2 Example Configuration

For a fully-configured example of using JupyterLab with JupyterHub, see the jupyterhub-deploy-teaching repository.

CHAPTER

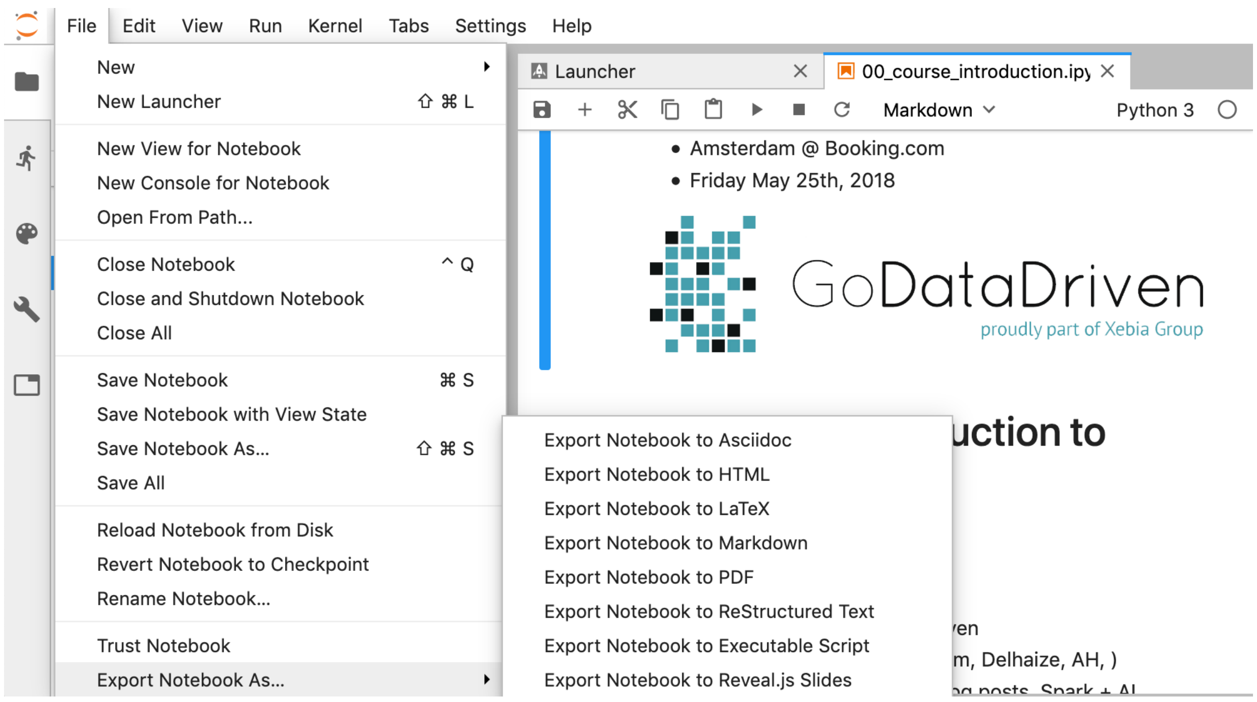
TWENTY

EXPORTING NOTEBOOKS

JupyterLab allows you to export your jupyter notebook files (`.ipynb`) into other file formats such as:

- Asciidoc `.asciidoc`
- HTML `.html`
- Latex `.tex`
- Markdown `.md`
- PDF `.pdf`
- ReStructured Text `.rst`
- Executable Script `.py`
- Reveal.js Slides `.html`

To access these options, while a notebook is open, browse the File menu:

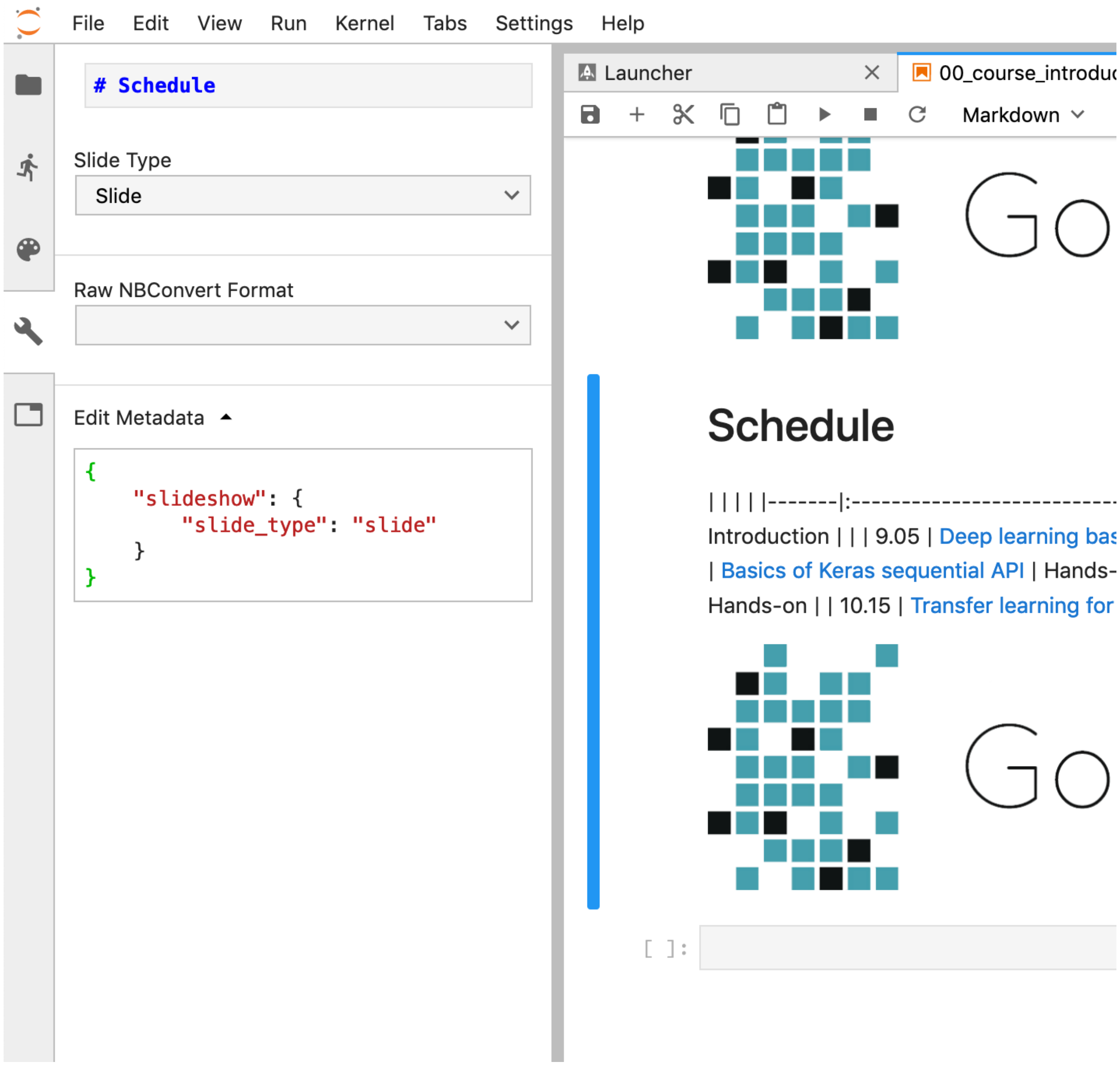


Note: The exporting options depend on your nbconvert configuration. For more information visit the official nbconvert documentation.

20.1 Reveal.js Slides

In order to export your notebooks as Reveal.js slides, follow these steps:

1. Open a notebook by double clicking it in the *file browser*.
2. Select Cell tools in the *left sidebar*.
3. Select the slide type (Slide, Subslide, Fragment, Skip, Notes).



```
# Schedule
```

Slide Type: Slide

Raw NBConvert Format

Edit Metadata

```
{
    "slideshow": {
        "slide_type": "slide"
    }
}
```

Schedule

| | | | |-------|:--------------------------
Introduction | | | 9.05 | Deep learning bas
| Basics of Keras sequential API | Hands-
Hands-on | | 10.15 | Transfer learning for

[ ]:

4. Activate another cell.
5. Repeat 3 and 4 until you selected the slide type for all of your cells.

After completing these steps, browse the file menu and export as described in the *exporting notebooks* section. A `.html` file that you will be prompted to download.

If you don't know how to navigate and interact with a Reveal.js presentation, visit the project's website.

CHAPTER

TWENTYONE

GENERAL CODEBASE ORIENTATION

The `jupyterlab` repository is a monorepo: it contains code for many packages that are versioned and published independently.

In particular, there are many TypeScript packages and a single Python package. The Python package contains server-side code, and also distributes the bundled-and-compiled TypeScript code.

See the Contributing Guidelines for detailed developer installation instructions.

21.1 Directories

The repository contains a number of top-level directories, the contents of which are described here.

21.1.1 Python package: `jupyterlab/`

This, along with the `setup.py`, comprises the Python code for the project. This includes the notebook server extension, JupyterLab's command line interface, entrypoints, and Python tests.

It also contains the final built JavaScript assets which are distributed with the Python package.

21.1.2 NPM packages: `packages/`

This contains the many TypeScript sub-packages which are independently versioned and published to `npmjs.org`. These are compiled to JavaScript and bundled with the Python package.

The bulk of JupyterLab's codebase resides in these packages. A common pattern for the various components in JupyterLab is to have one package that implements the component, and a second package postfixed with `-extension` that integrates that component with the rest of the application. Inspection of the contents of this directory shows many such packages.

You can build these packages by running `jlpm build:packages`.

21.1.3 Binder setup: `binder/`

This contains an environment specification for `repo2docker` which allows the repository to be tested on mybinder.org. This specification is developer focused. For a more user-focused binder see the JupyterLab demo

21.1.4 Build utilities: `builtutils/`

An `npm` package that contains several utility scripts for managing the JupyterLab build process.

You can build this package by running `jlpm build:utils`.

21.1.5 Design: `design/`

A directory containing a series of design documents motivating various choices made in the course of building JupyterLab.

21.1.6 Development-Mode: `dev_mode/`

An application directory containing built JavaScript assets which are used when developing the TypeScript sources. If you are running JupyterLab in `dev-mode`, you are serving the application out of this directory.

21.1.7 Documentation: `docs/`

This directory contains the Sphinx project for this documentation. You can create an environment to build the documentation using `conda create -f environment.yml`, and you can build the documentation by running `make html`. The entry point to the built docs will then be in `docs/build/index.html`.

21.1.8 Examples: `examples/`

The `examples/` directory contains stand-alone examples of components, such as a simple notebook on a page, a console, terminal, and a file browser. The `app` example illustrates a simplified combination of several of the components used in JupyterLab.

21.1.9 Jupyter Server Configuration: `jupyter-config/`

This directory contains metadata distributed with the Python package that allows it to automatically enable the Jupyter server extension upon installation.

21.1.10 Utility Scripts: `scripts/`

This directory contains a series of utility scripts which are primarily used in continuous integration testing for JupyterLab.

21.1.11 Testing: `tests/`

Tests for the TypeScript packages in the `packages/` directory. These test directories are themselves small `npm` packages which pull in the TypeScript sources and exercise their APIs.

21.1.12 Test Utilities: `testutils/`

A small `npm` package which is aids in running the tests in `tests/`.

21.1.13 TypeDoc Theming: `typedoc-theme`

A small theme used to help render our TypeDoc documentation.

CHAPTER

TWENTYTWO

EXTENSION DEVELOPER GUIDE

JupyterLab can be extended in four ways via:

- **application plugins (top level):** Application plugins extend the functionality of JupyterLab itself.
- **mime renderer extensions (top level):** Mime Renderer extensions are a convenience for creating an extension that can render mime data and potentially render files of a given type.
- **theme extensions (top level):** Theme extensions allow you to customize the appearance of JupyterLab by adding your own fonts, CSS rules, and graphics to the application.
- **document widget extensions (lower level):** Document widget extensions extend the functionality of document widgets added to the application, and we cover them in *Documents*.

See *Let's Make an Astronomy Picture of the Day JupyterLab Extension* to learn how to make a simple JupyterLab extension.

A JupyterLab application is comprised of:

- A core Application object
- Plugins

22.1 Plugins

A plugin adds a core functionality to the application:

- A plugin can require other plugins for operation.
- A plugin is activated when it is needed by other plugins, or when explicitly activated.
- Plugins require and provide `Token` objects, which are used to provide a typed value to the plugin's `activate()` method.
- The module providing plugin(s) must meet the JupyterLab.IPluginModule interface, by exporting a plugin object or array of plugin objects as the default export.

  We provide two cookiecutters to create JupyterLab plugin extensions in CommonJS and TypeScript.

The default plugins in the JupyterLab application include:

- Terminal - Adds the ability to create command prompt terminals.
- Shortcuts - Sets the default set of shortcuts for the application.
- Images - Adds a widget factory for displaying image files.
- Help - Adds a side bar widget for displaying external documentation.
- File Browser - Creates the file browser and the document manager and the file browser to the side bar.

- Editor - Add a widget factory for displaying editable source files.
- Console - Adds the ability to launch Jupyter Console instances for interactive kernel console sessions.

Here is a dependency graph for the core JupyterLab components:

Danger: Installing an extension allows for arbitrary code execution on the server, kernel, and in the client's browser. You should therefore take steps to protect against malicious changes to your extension's code. This includes ensuring strong authentication for your npm account.

22.2 Application Object

A Jupyter front-end application object is given to each plugin in its `activate()` function. The application object has:

- `commands` - an extensible registry used to add and execute commands in the application.
- `commandLinker` - used to connect DOM nodes with the command registry so that clicking on them executes a command.
- `docRegistry` - an extensible registry containing the document types that the application is able to read and render.
- `restored` - a promise that is resolved when the application has finished loading.
- `serviceManager` - low-level manager for talking to the Jupyter REST API.
- `shell` - a generic Jupyter front-end shell instance, which holds the user interface for the application.

22.3 Jupyter Front-End Shell

The Jupyter front-end shell is used to add and interact with content in the application. The `IShell` interface provides an `add()` method for adding widgets to the application. In JupyterLab, the application shell consists of:

- A `top` area for things like top level menus and toolbars.
- `left` and `right` side bar areas for collapsible content.
- A `main` work area for user activity.
- A `bottom` area for things like status bars.
- A `header` area for custom elements.

22.4 Phosphor

The Phosphor library is used as the underlying architecture of JupyterLab and provides many of the low level primitives and widget structure used in the application. Phosphor provides a rich set of widgets for developing desktop-like applications in the browser, as well as patterns and objects for writing clean, well-abstracted code. The widgets in the application are primarily **Phosphor widgets**, and Phosphor concepts, like message passing and signals, are used throughout. **Phosphor messages** are a *many-to-one* interaction that enables information like resize events to flow through the widget hierarchy in the application. **Phosphor signals** are a *one-to-many* interaction that enable listeners to react to changes in an observed object.

22.5 Extension Authoring

An Extension is a valid npm package that meets the following criteria:

- Exports one or more JupyterLab plugins as the default export in its main file.
- Has a `jupyterlab` key in its `package.json` which has `"extension"` metadata. The value can be `true` to use the main module of the package, or a string path to a specific module (e.g. `"lib/foo"`). Example:

```
"jupyterlab": {
  "extension": true
}
```

- It is also recommended to include the keyword `jupyterlab-extension` in the `package.json`, to aid with discovery (e.g. by the extension manager). Example:

```
"keywords": [
  "jupyter",
  "jupyterlab",
  "jupyterlab-extension"
],
```

While authoring the extension, you can use the command:

```
npm install   # install npm package dependencies
npm run build  # optional build step if using TypeScript, babel, etc.
jupyter labextension install  # install the current directory as an extension
```

This causes the builder to re-install the source folder before building the application files. You can re-build at any time using `jupyter lab build` and it will reinstall these packages.

You can also link other local `npm` packages that you are working on simultaneously using `jupyter labextension link`; they will be re-installed but not considered as extensions. Local extensions and linked packages are included in `jupyter labextension list`.

When using local extensions and linked packages, you can run the command

```
jupyter lab --watch
```

This will cause the application to incrementally rebuild when one of the linked packages changes. Note that only compiled JavaScript files (and the CSS files) are watched by the WebPack process. This means that if your extension is in TypeScript you'll have to run a `jlpm run build` before the changes will be reflected in JupyterLab. To avoid this step you can also watch the TypeScript sources in your extension which is usually assigned to the `tsc -w` shortcut. If WebPack doesn't seem to detect the changes, this can be related to the number of available watches.

Note that the application is built against **released** versions of the core JupyterLab extensions. If your extension depends on JupyterLab packages, it should be compatible with the dependencies in the `jupyterlab/static/package.json` file. Note that building will always use the latest JavaScript packages that meet the dependency requirements of JupyterLab itself and any installed extensions. If you wish to test against a specific patch release of one of the core JupyterLab packages you can temporarily pin that requirement to a specific version in your own dependencies.

If you must install a extension into a development branch of JupyterLab, you have to graft it into the source tree of JupyterLab itself. This may be done using the command

```
jlpm run add:sibling <path-or-url>
```

in the JupyterLab root directory, where `<path-or-url>` refers either to an extension `npm` package on the local file system, or a URL to a git repository for an extension `npm` package. This operation may be subsequently reversed by running

```
jlpm run remove:package <extension-dir-name>
```

This will remove the package metadata from the source tree and delete all of the package files.

The package should export EMCAScript 6 compatible JavaScript. It can import CSS using the syntax `require('foo.css')`. The CSS files can also import CSS from other packages using the syntax `@import url('~foo/index.css')`, where `foo` is the name of the package.

The following file types are also supported (both in JavaScript and CSS): `json`, `html`, `jpg`, `png`, `gif`, `svg`, `js.map`, `woff2`, `ttf`, `eot`.

If your package uses any other file type it must be converted to one of the above types or include a loader in the import statement. If you include a loader, the loader must be importable at build time, so if it is not already installed by JupyterLab, you must add it as a dependency of your extension.

If your JavaScript is written in any other dialect than EMCAScript 6 (2015) it should be converted using an appropriate tool. You can use Webpack to pre-build your extension to use any of it's features not enabled in our build configuration. To build a compatible package set `output.libraryTarget` to `"commonjs2"` in your Webpack configuration. (see this example repo).

If you publish your extension on `npm.org`, users will be able to install it as simply `jupyter labextension install <foo>`, where `<foo>` is the name of the published `npm` package. You can alternatively provide a script that runs `jupyter labextension install` against a local folder path on the user's machine or a provided tarball. Any valid `npm install` specifier can be used in `jupyter labextension install` (e.g. `foo@latest`, `bar@3.0.0.0`, `path/to/folder`, and `path/to/tar.gz`).

22.5.1 Testing your extension

There are a number of helper functions in `testutils` in this repo (which is a public `npm` package called `@jupyterlab/testutils`) that can be used when writing tests for an extension. See `tests/test-application` for an example of the infrastructure needed to run tests. There is a `karma` config file that points to the parent directory's `karma` config, and a test runner, `run-test.py` that starts a Jupyter server.

If you are using jest to test your extension, you will need to transpile the jupyterlab packages to `commonjs` as they are using ES6 modules that `node` does not support.

To transpile jupyterlab packages, you need to install the following package:

```
jlpm add --dev jest@^24 @types/jest@^24 ts-jest@^24 @babel/core@^7 @babel/preset-env@^
↪7
```

Then in *jest.config.js*, you will specify to use babel for js files and ignore all node modules except the jupyterlab ones:

```
module.exports = {
  preset: 'ts-jest/presets/js-with-babel',
  moduleFileExtensions: ['ts', 'tsx', 'js', 'jsx', 'json', 'node'],
  transformIgnorePatterns: ['/node_modules/(?!(@jupyterlab/.*)/)'],
  globals: {
    'ts-jest': {
      tsConfig: 'tsconfig.json'
    }
  },
  ... // Other options useful for your extension
};
```

Finally, you will need to configure babel with a `babel.config.js` file containing:

```
module.exports = {
  presets: [
    [
      '@babel/preset-env',
      {
        targets: {
          node: 'current'
        }
      }
    ]
  ]
};
```

22.6 Mime Renderer Extensions

Mime Renderer extensions are a convenience for creating an extension that can render mime data and potentially render files of a given type. We provide a cookiecutter for mime renderer extensions in TypeScript here.

Mime renderer extensions are more declarative than standard extensions. The extension is treated the same from the command line perspective (`jupyter labextension install`), but it does not directly create JupyterLab plugins. Instead it exports an interface given in the rendermime-interfaces package.

The JupyterLab repo has an example mime renderer extension for pdf files. It provides a mime renderer for pdf data and registers itself as a document renderer for pdf file types.

The JupyterLab organization also has a mime renderer extension tutorial which adds mp4 video rendering to the application here.

The `rendermime-interfaces` package is intended to be the only JupyterLab package needed to create a mime renderer extension (using the interfaces in TypeScript or as a form of documentation if using plain JavaScript).

The only other difference from a standard extension is that has a `jupyterlab` key in its `package.json` with `"mimeExtension"` metadata. The value can be `true` to use the main module of the package, or a string path to a specific module (e.g. `"lib/foo"`).

The mime renderer can update its data by calling `.setData()` on the model it is given to render. This can be used for example to add a `png` representation of a dynamic figure, which will be picked up by a notebook model and added to the notebook document. When using `IDocumentWidgetFactoryOptions`, you can update the document model by calling `.setData()` with updated data for the rendered MIME type. The document can then be saved by the user in the usual manner.

22.7 Themes

A theme is a JupyterLab extension that uses a `ThemeManager` and can be loaded and unloaded dynamically. The package must include all static assets that are referenced by `url()` in its CSS files. Local URLs can be used to reference files relative to the location of the referring sibling CSS files. For example `url('images/foo.png')` or `url('../foo/bar.css')` can be used to refer local files in the theme. Absolute URLs (starting with a `/`) or external URLs (e.g. `https:`) can be used to refer to external assets. The path to the theme asset entry point is specified `package.json` under the `"jupyterlab"` key as `"themePath"`. See the JupyterLab Light Theme for an example. Ensure that the theme files are included in the `"files"` metadata in `package.json`. Note that if you want to use SCSS, SASS, or LESS files, you must compile them to CSS and point JupyterLab to the CSS files.

The theme extension is installed in the same way as a regular extension (see *extension authoring*).

It is also possible to create a new theme using the TypeScript theme cookiecutter.

22.8 Standard (General-Purpose) Extensions

JupyterLab's modular architecture is based around the idea that all extensions are on equal footing, and that they interact with each other through typed interfaces that are provided by `Token` objects. An extension can provide a `Token` to the application, which other extensions can then request for their own use.

22.8.1 Core Tokens

The core packages of JupyterLab provide a set of tokens, which are listed here, along with short descriptions of when you might want to use them in your extensions.

- `@jupyterlab/application:IConnectionLost`: A token for invoking the dialog shown when JupyterLab has lost its connection to the server. Use this if, for some reason, you want to bring up the "connection lost" dialog under new circumstances.
- `@jupyterlab/application:IInfo`: A token providing metadata about the current application, including currently disabled extensions and whether dev mode is enabled.
- `@jupyterlab/application:IPaths`: A token providing information about various URLs and server paths for the current application. Use this token if you want to assemble URLs to use the JupyterLab REST API.
- `@jupyterlab/application:ILabStatus`: An interface for interacting with the application busy/dirty status. Use this if you want to set the application "busy" favicon, or to set the application "dirty" status, which asks the user for confirmation before leaving.
- `@jupyterlab/application:ILabShell`: An interface to the JupyterLab shell. The top-level application object also has a reference to the shell, but it has a restricted interface in order to be agnostic to different spins on the application. Use this to get more detailed information about currently active widgets and layout state.
- `@jupyterlab/application:ILayoutRestorer`: An interface to the application layout restoration functionality. Use this to have your activities restored across page loads.
- `@jupyterlab/application:IMimeDocumentTracker`: A widget tracker for documents rendered using a mime renderer extension. Use this if you want to list and interact with documents rendered by such extensions.
- `@jupyterlab/application:IRouter`: The URL router used by the application. Use this to add custom URL-routing for your extension (e.g., to invoke a command if the user navigates to a sub-path).
- `@jupyterlab/apputils:ICommandPalette`: An interface to the application command palette in the left panel. Use this to add commands to the palette.
- `@jupyterlab/apputils:ISplashScreen`: An interface to the splash screen for the application. Use this if you want to show the splash screen for your own purposes.
- `@jupyterlab/apputils:IThemeManager`: An interface to the theme manager for the application. Most extensions will not need to use this, as they can register a *theme extension*.
- `@jupyterlab/apputils:IWindowResolver`: An interface to a window resolver for the application. JupyterLab workspaces are given a name, which are determined using the window resolver. Require this if you want to use the name of the current workspace.
- `@jupyterlab/codeeditor:IEditorServices`: An interface to the text editor provider for the application. Use this to create new text editors and host them in your UI elements.
- `@jupyterlab/completer:ICompletionManager`: An interface to the completion manager for the application. Use this to allow your extension to invoke a completer.

- `@jupyterlab/console:IConsoleTracker`: A widget tracker for code consoles. Use this if you want to be able to iterate over and interact with code consoles created by the application.
- `@jupyterlab/console:IContentFactory`: A factory object that creates new code consoles. Use this if you want to create and host code consoles in your own UI elements.
- `@jupyterlab/coreutils:ISettingRegistry`: An interface to the JupyterLab settings system. Use this if you want to store settings for your application. See *extension settings* for more information.
- `@jupyterlab/coreutils:IStateDB`: An interface to the JupyterLab state database. Use this if you want to store data that will persist across page loads. See *state database* for more information.
- `@jupyterlab/docmanager:IDocumentManager`: An interface to the manager for all documents used by the application. Use this if you want to open and close documents, create and delete files, and otherwise interact with the file system.
- `@jupyterlab/documentsearch:ISearchProviderRegistry`: An interface for a registry of search providers for the application. Extensions can register their UI elements with this registry to provide find/replace support.
- `@jupyterlab/filebrowser:IFileBrowserFactory`: A factory object that creates file browsers. Use this if you want to create your own file browser (e.g., for a custom storage backend), or to interact with other file browsers that have been created by extensions.
- `@jupyterlab/fileeditor:IEditorTracker`: A widget tracker for file editors. Use this if you want to be able to iterate over and interact with file editors created by the application.
- `@jupyterlab/htmlviewer:IHTMLViewerTracker`: A widget tracker for rendered HTML documents. Use this if you want to be able to iterate over and interact with HTML documents viewed by the application.
- `@jupyterlab/imageviewer:IImageTracker`: A widget tracker for images. Use this if you want to be able to iterate over and interact with images viewed by the application.
- `@jupyterlab/inspector:IInspector`: An interface for adding variable inspectors to widgets. Use this to add the ability to hook into the variable inspector to your extension.
- `@jupyterlab/launcher:ILauncher`: An interface to the application activity launcher. Use this to add your extension activities to the launcher panel.
- `@jupyterlab/mainmenu:IMainMenu`: An interface to the main menu bar for the application. Use this if you want to add your own menu items.
- `@jupyterlab/markdownviewer:IMarkdownViewerTracker`: A widget tracker for markdown document viewers. Use this if you want to iterate over and interact with rendered markdown documents.
- `@jupyterlab/notebook:INotebookTools`: An interface to the `Notebook Tools` panel in the application left area. Use this to add your own functionality to the panel.
- `@jupyterlab/notebook:IContentFactory`: A factory object that creates new notebooks. Use this if you want to create and host notebooks in your own UI elements.
- `@jupyterlab/notebook:INotebookTracker`: A widget tracker for notebooks. Use this if you want to be able to iterate over and interact with notebooks created by the application.
- `@jupyterlab/rendermime:IRenderMimeRegistry`: An interface to the rendermime registry for the application. Use this to create renderers for various mime-types in your extension. Most extensions will not need to use this, as they can register a *mime renderer extension*.
- `@jupyterlab/rendermime:ILatexTypesetter`: An interface to the LaTeX typesetter for the application. Use this if you want to typeset math in your extension.

- `@jupyterlab/settingeditor:ISettingEditorTracker`: A widget tracker for setting editors. Use this if you want to be able to iterate over and interact with setting editors created by the application.
- `@jupyterlab/statusbar:IStatusBar`: An interface to the status bar on the application. Use this if you want to add new status bar items.
- `@jupyterlab/terminal:ITerminalTracker`: A widget tracker for terminals. Use this if you want to be able to iterate over and interact with terminals created by the application.
- `@jupyterlab/tooltip:ITooltipManager`: An interface to the tooltip manager for the application. Use this to allow your extension to invoke a tooltip.
- `@jupyterlab/vdom:IVDOMTracker`: A widget tracker for virtual DOM (VDOM) documents. Use this to iterate over and interact with VDOM instances created by the application.

22.8.2 Standard Extension Example

For a concrete example of a standard extension, see *How to extend the Notebook plugin*. Notice that the mime renderer extensions use a limited, simplified interface to JupyterLab's extension system. Modifying the notebook plugin requires the full, general-purpose interface to the extension system.

22.8.3 Storing Extension Data

In addition to the file system that is accessed by using the `@jupyterlab/services` package, JupyterLab offers two ways for extensions to store data: a client-side state database that is built on top of `localStorage` and a plugin settings system that provides for default setting values and user overrides.

Extension Settings

An extension can specify user settings using a JSON Schema. The schema definition should be in a file that resides in the `schemaDir` directory that is specified in the `package.json` file of the extension. The actual file name should use is the part that follows the package name of extension. So for example, the JupyterLab `apputils-extension` package hosts several plugins:

- `'@jupyterlab/apputils-extension:menu'`
- `'@jupyterlab/apputils-extension:palette'`
- `'@jupyterlab/apputils-extension:settings'`
- `'@jupyterlab/apputils-extension:themes'`

And in the `package.json` for `@jupyterlab/apputils-extension`, the `schemaDir` field is a directory called `schema`. Since the `themes` plugin requires a JSON schema, its schema file location is: `schema/themes.json`. The plugin's name is used to automatically associate it with its settings file, so this naming convention is important. Ensure that the schema files are included in the `"files"` metadata in `package.json`.

See the fileeditor-extension for another example of an extension that uses settings.

Note: You can override default values of the extension settings by defining new default values in an `overrides.json` file in the application settings directory. So for example, if you would like to set the dark theme by default instead of the light one, an `overrides.json` file containing the following lines needs to be added in the application settings directory (by default this is the `share/jupyter/lab/settings` folder).

```
{
  "@jupyterlab/apputils-extension:themes": {
    "theme": "JupyterLab Dark"
  }
}
```

State Database

The state database can be accessed by importing `IStateDB` from `@jupyterlab/coreutils` and adding it to the list of `requires` for a plugin:

```
const id = 'foo-extension:IFoo';

const IFoo = new Token<IFoo>(id);

interface IFoo {}

class Foo implements IFoo {}

const plugin: JupyterFrontEndPlugin<IFoo> = {
  id,
  requires: [IStateDB],
  provides: IFoo,
  activate: (app: JupyterFrontEnd, state: IStateDB): IFoo => {
    const foo = new Foo();
    const key = `${id}:some-attribute`;

    // Load the saved plugin state and apply it once the app
    // has finished restoring its former layout.
    Promise.all([state.fetch(key), app.restored])
      .then(([saved]) => { /* Update `foo` with `saved`. */ });

    // Fulfill the plugin contract by returning an `IFoo`.
    return foo;
  },
  autoStart: true
};
```

22.8.4 Context Menus

JupyterLab has an application-wide context menu available as `app.contextMenu`. See the Phosphor docs for the item creation options. If you wish to preempt the application context menu, you can use a 'contextmenu' event listener and call `event.stopPropagation` to prevent the application context menu handler from being called (it is listening in the bubble phase on the `document`). At this point you could show your own Phosphor contextMenu, or simply stop propagation and let the system context menu be shown. This would look something like the following in a `Widget` subclass:

```
// In `onAfterAttach()`
this.node.addEventListener('contextmenu', this);

// In `handleEvent()`
case 'contextmenu':
  event.stopPropagation();
```

22.8.5 Using React

We also provide support for using *React* in your JupyterLab extensions, as well as in the core codebase.

22.8.6 Companion Packages

If your extensions depends on the presence of one or more packages in the kernel, or on a notebook server extension, you can add metadata to indicate this to the extension manager by adding metadata to your package.json file. The full options available are:

```
"jupyterlab": {
  "discovery": {
    "kernel": [
      {
        "kernel_spec": {
          "language": "<regexp for matching kernel language>",
          "display_name": "<regexp for matching kernel display name>"   // optional
        },
        "base": {
          "name": "<the name of the kernel package>"
        },
        "overrides": {   // optional
          "<manager name, e.g. 'pip'>": {
            "name": "<name of kernel package on pip, if it differs from base name>"
          }
        },
        "managers": [   // list of package managers that have your kernel package
            "pip",
            "conda"
        ]
      }
    ],
    "server": {
      "base": {
        "name": "<the name of the server extension package>"
      },
      "overrides": {   // optional
        "<manager name, e.g. 'pip'>": {
          "name": "<name of server extension package on pip, if it differs from base␣
↪name>"
        }
      },
      "managers": [   // list of package managers that have your server extension␣
↪package
          "pip",
          "conda"
      ]
    }
  }
}
```

A typical setup for e.g. a jupyter-widget based package will then be:

```
"keywords": [
    "jupyterlab-extension",
    "jupyter",
```

(continues on next page)

(continued from previous page)

```
    "widgets",
    "jupyterlab"
],
"jupyterlab": {
  "extension": true,
  "discovery": {
    "kernel": [
      {
        "kernel_spec": {
          "language": "^python",
        },
        "base": {
          "name": "myipywidgetspackage"
        },
        "managers": [
            "pip",
            "conda"
        ]
      }
    ]
  }
}
```

Currently supported package managers are:

- `pip`
- `conda`

22.8.7 Shipping Packages

Most extensions are single JavaScript packages, and can be shipped on npmjs.org. This makes them discoverable by the JupyterLab extension manager, provided they have the `jupyterlab-extension` keyword in their `package.json`. If the package also contains a server extension (Python package), the author has two options. The server extension and the JupyterLab extension can be shipped in a single package, or they can be shipped separately.

The JupyterLab extension can be bundled in a package on PyPI and conda-forge so that it ends up in the user's application directory. Note that the user will still have to run `jupyter lab build` (or build when prompted in the UI) in order to use the extension. The general idea is to pack the Jupyterlab extension using `npm pack`, and then use the `data_files` logic in `setup.py` to ensure the file ends up in the `<jupyterlab_application>/share/jupyter/lab/extensions` directory.

Note that even if the JupyterLab extension is unusable without the server extension, as long as you use the companion package metadata it is still useful to publish it to npmjs.org so it is discoverable by the JupyterLab extension manager.

The server extension can be enabled on install by using `data_files`. an example of this approach is jupyterlab-matplotlib. The file used to enable the server extension is here. The logic to ship the JS tarball and server extension enabler is in setup.py. Note that the `setup.py` file has additional logic to automatically create the JS tarball as part of the release process, but this could also be done manually.

Technically, a package that contains only a JupyterLab extension could be created and published on `conda-forge`, but it would not be discoverable by the JupyterLab extension manager.

CHAPTER

TWENTYTHREE

COMMON EXTENSION POINTS

Most of the component parts of JupyterLab are designed to be extensible, and they provide public APIs that can be requested in extensions via tokens. A list of tokens that extension authors can request is documented in *Core Tokens*.

This is intended to be a guide for some of JupyterLab's most commonly-used extension points. However, it is not an exhaustive account of how to extend the application components, and more detailed descriptions of their public APIs may be found in the JupyterLab and Phosphor API documentation.

Table of contents

- *Commands*
- *Command Palette*
- *Main Menu*
- *Context Menu*
- *Keyboard Shortcuts*
- *Launcher*
- *Left/Right Areas*
- *Status Bar*
- *Widget Tracker*
- *Copy Shareable Link*

23.1 Commands

Perhaps the most common way to add functionality to JupyterLab is via commands. These are lightweight objects that include a function to execute combined with additional metadata, including how they are labeled and when they are to be enabled. The application has a single command registry, keyed by string command IDs, to which you can add your custom commands.

The commands added to the command registry can then be used to populate several of the JupyterLab user interface elements, including menus and the launcher.

Here is a sample block of code that adds a command to the application (given by `app`):

```
const commandID = 'my-command';
const toggled = false;
```

(continues on next page)

(continued from previous page)

```
app.commands.addCommand(commandID, {
  label: 'My Cool Command',
  isEnabled: true,
  isVisible: true,
  isToggled: () => toggled,
  iconClass: 'some-css-icon-class',
  execute: () => {
    console.log(`Executed ${commandID}`);
    toggled = !toggled;
});
```

This example adds a new command, which, when triggered, calls the `execute` function. `isEnabled` indicates whether the command is enabled, and determines whether renderings of it are greyed out. `isToggled` indicates whether to render a check mark next to the command. `isVisible` indicates whether to render the command at all. `iconClass` specifies a CSS class which can be used to display an icon next to renderings of the command.

Each of `isEnabled`, `isToggled`, and `isVisible` can be either a boolean value or a function that returns a boolean value, in case you want to do some logic in order to determine those conditions.

Likewise, each of `label` and `iconClass` can be either a string value or a function that returns a string value.

There are several more options which can be passed into the command registry when adding new commands. These are documented here.

After a command has been added to the application command registry you can add them to various places in the application user interface, where they will be rendered using the metadata you provided.

23.2 Command Palette

In order to add a command to the command palette, you need to request the `ICommandPalette` token in your extension. Here is an example showing how to add a command to the command palette (given by `palette`):

```
palette.addItem({
  command: commandID,
  category: 'my-category'
  args: {}
});
```

The command ID is the same ID that you used when registering the command. You must also provide a `category`, which determines the subheading of the command palette in which to render the command. It can be a preexisting category (e.g., `'notebook'`), or a new one of your own choosing.

The `args` are a JSON object that will be passed into your command's functions at render/execute time. You can use these to customize the behavior of your command depending on how it is invoked. For instance, you can pass in `args: { isPalette: true }`. Your command `label` function can then check the `args` it is provided for `isPalette`, and return a different label in that case. This can be useful to make a single command flexible enough to work in multiple contexts.

23.3 Main Menu

There are three main ways to extend JupyterLab's main menu.

1. You can add your own menu to the menu bar.

2. You can add new commands to the existing menus.
3. You can register your extension with one of the existing semantic menu items.

In all three cases, you should request the `IMainMenu` token for your extension.

23.3.1 Adding a New Menu

To add a new menu to the menu bar, you need to create a new Phosphor menu.

You can then add commands to the menu in a similar way to the command palette, and add that menu to the main menu bar:

```
const menu = new Menu({ commands: app.commands });
menu.addItem({
  command: commandID,
  args: {},
});

mainMenu.addMenu(menu, { rank: 40 });
```

As with the command palette, you can optionally pass in `args` to customize the rendering and execution behavior of the command in the menu context.

23.3.2 Adding a New Command to an Existing Menu

In many cases you will want to add your commands to the existing JupyterLab menus rather than creating a separate menu for your extension. Because the top-level JupyterLab menus are shared among many extensions, the API for adding items is slightly different. In this case, you provide a list of commands and a rank, and these commands will be displayed together in a separate group within an existing menu.

For instance, to add a command group with `firstCommandID` and `secondCommandID` to the File menu, you would do the following:

```
mainMenu.fileMenu.addGroup([
  {
    command: firstCommandID,
  },
  {
    command: secondCommandID,
  }
], 40 /* rank */);
```

23.3.3 Registering a Semantic Menu Item

There are some commands in the JupyterLab menu system that are considered common and important enough that they are treated differently.

For instance, we anticipate that many activities may want to provide a command to close themselves and perform some cleanup operation (like closing a console and shutting down its kernel). Rather than having a proliferation of similar menu items for this common operation of “closing-and-cleanup”, we provide a single command that can adapt itself to this use case, which we term a “semantic menu item”. For this example, it is the File Menu `closeAndCleaners` set.

Here is an example of using the `closeAndCleaners` semantic menu item:

```
mainMenu.fileMenu.closeAndCleaners.add({
  tracker,
  action: 'Shutdown',
  name: 'My Activity',
  closeAndCleanup: current => {
    current.close();
    return current.shutdown();
  }
});
```

In this example, `tracker` is a *Widget Tracker*, which allows the menu item to determine whether to delegate the menu command to your activity, `name` is a name given to your activity in the menu label, `action` is a verb given to the cleanup operation in the menu label, and `closeAndCleanup` is the actual function that performs the cleanup operation. So if the current application activity is held in the `tracker`, then the menu item will show `Shutdown My Activity`, and delegate to the `closeAndCleanup` function that was provided.

More examples for how to register semantic menu items are found throughout the JupyterLab code base. The available semantic menu items are:

- `IEditMenu.IUndoer`: an activity that knows how to undo and redo.
- `IEditMenu.IClearer`: an activity that knows how to clear its content.
- `IEditMenu.IGoToLiner`: an activity that knows how to jump to a given line.
- `IFileMenu.ICloseAndCleaner`: an activity that knows how to close and clean up after itself.
- `IFileMenu.IConsoleCreator`: an activity that knows how to create an attached code console for itself.
- `IHelpMenu.IKernelUser`: an activity that knows how to get a related kernel session.
- `IKernelMenu.IKernelUser`: an activity that can perform various kernel-related operations.
- `IRunMenu.ICodeRunner`: an activity that can run code from its content.
- `IViewMenu.IEditorViewer`: an activity that knows how to set various view-related options on a text editor that it owns.

23.4 Context Menu

The application context menu is shown when the user right-clicks, and is populated with menu items that are most relevant to the thing that the user clicked.

The context menu system determines which items to show based on CSS selectors. It propagates up the DOM tree and tests whether a given HTML element matches the CSS selector provided by a given command.

Here is an example showing how to add a command to the application context menu:

```
app.contextMenu.addItem({
  command: commandID,
  selector: '.jp-Notebook'
})
```

In this example, the command indicated by `commandID` is shown whenever the user right-clicks on a DOM element matching `.jp-Notebook` (that is to say, a notebook). The selector can be any valid CSS selector, and may target your own UI elements, or existing ones. A list of CSS selectors currently used by context menu commands is given in *Commonly used CSS selectors*.

23.5 Keyboard Shortcuts

There are two ways of adding keyboard shortcuts in JupyterLab. If you don't want the shortcuts to be user-configurable, you can add them directly to the application command registry:

```
app.commands.addKeyBinding({
  command: commandID,
  args: {},
  keys: ['Accel T'],
  selector: '.jp-Notebook'
});
```

In this example `my-command` command is mapped to `Accel T`, where `Accel` corresponds to `Cmd` on a Mac and `Ctrl` on Windows and Linux computers.

The behavior for keyboard shortcuts is very similar to that of the context menu: the shortcut handler propagates up the DOM tree from the focused element and tests each element against the registered selectors. If a match is found, then that command is executed with the provided `args`. Full documentation for the options for `addKeyBinding` can be found here.

JupyterLab also provides integration with its settings system for keyboard shortcuts. Your extension can provide a settings schema with a `jupyter.lab.shortcuts` key, declaring default keyboard shortcuts for a command:

```
{
  "jupyter.lab.shortcuts": [
    {
      "command": "my-command",
      "keys": ["Accel T"],
      "selector": ".jp-mod-searchable"
    }
  ]
}
```

Shortcuts added to the settings system will be editable by users.

23.6 Launcher

As with menus, keyboard shortcuts, and the command palette, new items can be added to the application launcher via commands. You can do this by requesting the `ILauncher` token in your extension:

```
launcher.add({
  command: commandID,
  category: 'Other',
  rank: 0
});
```

In addition to providing a command ID, you also provide a category in which to put your item, (e.g. 'Notebook', or 'Other'), as well as a rank to determine its position among other items.

23.7 Left/Right Areas

The left and right areas of JupyterLab are intended to host more persistent user interface elements than the main area. That being said, extension authors are free to add whatever components they like to these areas. The outermost-level of the object that you add is expected to be a Phosphor `Widget`, but that can host any content you like (such as React components).

As an example, the following code executes an application command to a terminal widget and then adds the terminal to the right area:

```
app.commands
  .execute('terminal:create-new')
  .then((terminal: WidgetModuleType.Terminal) => {
    app.shell.add(terminal, 'right');
  });
```

23.8 Status Bar

JupyterLab's status bar is intended to show small pieces of contextual information. Like the left and right areas, it only expects a Phosphor `Widget`, which might contain any kind of content. Since the status bar has limited space, you should endeavor to only add small widgets to it.

The following example shows how to place a status item that displays the current "busy" status for the application. This information is available from the `ILabStatus` token, which we reference by a variable named `labStatus`. We place the `statusWidget` in the middle of the status bar. When the `labStatus` busy state changes, we update the text content of the `statusWidget` to reflect that.

```
const statusWidget = new Widget();
labStatus.busySignal.connect(() => {
  statusWidget.node.textContent = labStatus.isBusy ? 'Busy' : 'Idle';
});
statusBar.registerStatusItem('lab-status', {
  align: 'middle',
  item: statusWidget
});
```

23.9 Widget Tracker

Often extensions will want to interact with documents and activities created by other extensions. For instance, an extension may want to inject some text into a notebook cell, or set a custom keymap, or close all documents of a certain type. Actions like these are typically done by widget trackers. Extensions keep track of instances of their activities in `WidgetTrackers`, which are then provided as tokens so that other extensions may request them.

For instance, if you want to interact with notebooks, you should request the `INotebookTracker` token. You can then use this tracker to iterate over, filter, and search all open notebooks. You can also use it to be notified via signals when notebooks are added and removed from the tracker.

Widget tracker tokens are provided for many activities in JupyterLab, including notebooks, consoles, text files, mime documents, and terminals. If you are adding your own activities to JupyterLab, you might consider providing a `WidgetTracker` token of your own, so that other extensions can make use of it.

23.10 Copy Shareable Link

The file browser provides a context menu item "Copy Shareable Link". The desired behavior will vary by deployment and the users it serves. The file browser supports overriding the behavior of this item.

```
import {
  IFileBrowserFactory
} from '@jupyterlab/filebrowser';

import {
  JupyterFrontEnd, JupyterFrontEndPlugin
} from '@jupyterlab/application';


const shareFile: JupyterFrontEndPlugin<void> = {
  activate: activateShareFile,
  id: commandID,
  requires: [IFileBrowserFactory],
  autoStart: true
};

function activateShareFile(
  app: JupyterFrontEnd,
  factory: IFileBrowserFactory
): void {
  const { commands } = app;
  const { tracker } = factory;

  commands.addCommand('filebrowser:share-main', {
    execute: () => {
      const widget = tracker.currentWidget;
      if (!widget) {
        return;
      }
      const path = encodeURI(widget.selectedItems().next().path);
      // Do something with path.
    },
    isVisible: () =>
      tracker.currentWidget &&
      toArray(tracker.currentWidget.selectedItems()).length === 1,
    iconClass: 'jp-MaterialIcon jp-LinkIcon',
    label: 'Copy Shareable Link'
  });
}
```

Note that before enabling this plugin in the usual way, you must *disable* the default plugin provided by the built-in file browser.

```
jupyter labextension disable @jupyterlab/filebrowser-extension:share-file
```

CHAPTER

TWENTYFOUR

CONTRIBUTING TO JUPYTERLAB

This page provides general information about contributing to the project.

Table of contents

- *Writing Documentation*
- *User Interface Naming Conventions*
- *Keyboard Shortcuts*
- *Screenshots and Animations*

24.1 Writing Documentation

This section provides information about writing documentation for JupyterLab. See our Contributor Guide for details on installation and testing.

24.1.1 Writing Style

- The documentation should be written in the second person, referring to the reader as “you” and not using the first person plural “we.” The author of the documentation is not sitting next to the user, so using “we” can lead to frustration when things don’t work as expected.
- Avoid words that trivialize using JupyterLab such as “simply” or “just.” Tasks that developers find simple or easy may not be for users.
- Write in the active tense, so “drag the notebook cells. . . ” rather than “notebook cells can be dragged. . . ”
- The beginning of each section should begin with a short (1-2 sentence) high-level description of the topic, feature or component.
- Use “enable” rather than “allow” to indicate what JupyterLab makes possible for users. Using “allow” connotes that we are giving them permission, whereas “enable” connotes empowerment.

24.2 User Interface Naming Conventions

24.2.1 Documents, Files, and Activities

Files are referrred to as either files or documents, depending on the context.

Documents are more human centered. If human viewing, interpretation, interaction is an important part of the experience, it is a document in that context. For example, notebooks and markdown files will often be referring to as documents unless referring to the file-ness aspect of it (e.g., the notebook filename).

Files are used in a less human-focused context. For example, we refer to files in relation to a file system or file name.

Activities can be either a document or another UI panel that is not file backed, such as terminals, consoles or the inspector. An open document or file is an activity in that it is represented by a panel that you can interact with.

24.2.2 Element Names

- The generic content area of a tabbed UI is a panel, but prefer to refer to the more specific name, such as "File browser." Tab bars have tabs which toggle panels.
- The menu bar contains menu items, which have their own submenus.
- The main work area can be referred to as the work area when the name is unambiguous.
- When describing elements in the UI, colloquial names are preferred (e.g., "File browser" instead of "Files panel").

The majority of names are written in lower case. These names include:

- tab
- panel
- menu bar
- sidebar
- file
- document
- activity
- tab bar
- main work area
- file browser
- command palette
- cell inspector
- code console

The following sections of the user interface should be in title case, directly quoting a word in the UI:

- File menu
- Files tab
- Running panel
- Tabs panel
- Single-Document Mode

The capitalized words match the label of the UI element the user is clicking on because there does not exist a good colloquial name for the tool, such as "file browser" or "command palette".

See *The JupyterLab Interface* for descriptions of elements in the UI.

24.3 Keyboard Shortcuts

Typeset keyboard shortcuts as follows:

- Monospace typeface, with spaces between individual keys: `Shift Enter`.
- For modifiers, use the platform independent word describing key: `Shift`.
- For the `Accel` key use the phrase: `Command/Ctrl`.
- Don't use platform specific icons for modifier keys, as they are difficult to display in a platform specific way on Sphinx/RTD.

24.4 Screenshots and Animations

Our documentation should contain screenshots and animations that illustrate and demonstrate the software. Here are some guidelines for preparing them:

- Make sure the screenshot does not contain copyrighted material (preferable), or the license is allowed in our documentation and clearly stated.
- If taking a png screenshot, use the Firefox or Chrome developer tools to do the following:
  - set the browser viewport to 1280x720 pixels
  - set the device pixel ratio to 1:1 (i.e., non-hidpi, non-retina)
  - screenshot the entire *viewport* using the browser developer tools. Screenshots should not include any browser elements such as the browser address bar, browser title bar, etc., and should not contain any desktop background.
- If creating a movie, adjust the settings as above (1280x720 viewport resolution, non-hidpi) and use a screen capture utility of your choice to capture just the browser viewport.
- For PNGs, reduce their size using `pngquant --speed 1 <filename>`. The resulting filename will have `-fs8` appended, so make sure to rename it and use the resulting file. Commit the optimized png file to the main repository. Each png file should be no more than a few hundred kilobytes.
- For movies, upload them to the IPython/Jupyter YouTube channel and add them to the jupyterlab-media repository. To embed a movie in the documentation, use the `www.youtube-nocookie.com` website, which can be found by clicking on the 'privacy-enhanced' embedding option in the Share dialog on YouTube. Add the following parameters the end of the URL `?rel=0&amp;showinfo=0`. This disables the video title and related video suggestions.
- Screenshots or animations should be preceded by a sentence describing the content, such as "To open a file, double-click on its name in the File Browser:".
- We have custom CSS that will add box shadows, and proper sizing of screenshots and embedded YouTube videos. See examples in the documentation for how to embed these assets.

To help us organize screenshots and animations, please name the files with a prefix that matches the names of the source file in which they are used:

```
sourcefile.rst
sourcefile_filebrowser.png
sourcefile_editmenu.png
```

This will help us to keep track of the images as documentation content evolves.

CHAPTER

TWENTYFIVE

DOCUMENTS

JupyterLab can be extended in several ways:

- **Extensions (top level)**: Application extensions extend the functionality of JupyterLab itself, and we cover them in the *Extension Developer Guide*.
- **Document widget extensions (lower level):** Document widget extensions extend the functionality of document widgets added to the application, and we cover them in this section.

For this section, the term 'document' refers to any visual thing that is backed by a file stored on disk (i.e. uses Contents API).

25.1 Overview of document architecture

A 'document' in JupyterLab is represented by a model instance implementing the IModel interface. The model interface is intentionally fairly small, and concentrates on representing the data in the document and signaling changes to that data. Each model has an associated context instance as well. The context for a model is the bridge between the internal data of the document, stored in the model, and the file metadata and operations possible on the file, such as save and revert. Since many objects will need both the context and the model, the context contains a reference to the model as its *.model* attribute.

A single file path can have multiple different models (and hence different contexts) representing the file. For example, a notebook can be opened with a notebook model and with a text model. Different models for the same file path do not directly communicate with each other.

Document widgets represent a view of a document model. There can be multiple document widgets associated with a single document model, and they naturally stay in sync with each other since they are views on the same underlying data model.

The Document Registry is where document types and factories are registered. Plugins can require a document registry instance and register their content types and providers.

The Document Manager uses the Document Registry to create models and widgets for documents. The Document Manager handles the lifecycle of documents for the application.

25.2 Document Registry

Document widget extensions in the JupyterLab application can register:

- file types
- model factories for specific file types
- widget factories for specific model factories

- widget extension factories

25.2.1 Widget Factories

Create a widget for a given file.

Example

- The notebook widget factory that creates NotebookPanel widgets.

25.2.2 Model Factories

Create a model for a given file.

Models are generally differentiated by the contents options used to fetch the model (e.g. text, base64, notebook).

25.2.3 Widget Extension Factories

Adds additional functionality to a widget type. An extension instance is created for each widget instance, enabling the extension to add functionality to each widget or observe the widget and/or its context.

Examples

- The ipywidgets extension that is created for NotebookPanel widgets.
- Adding a button to the toolbar of each NotebookPanel widget.

25.2.4 File Types

25.2.5 Document Models

Created by the model factories and passed to widget factories and widget extension factories. Models are the way in which we interact with the data of a document. For a simple text file, we typically only use the `to/fromString()` methods. A more complex document like a Notebook contains more points of interaction like the Notebook metadata.

25.2.6 Document Contexts

Created by the Document Manager and passed to widget factories and widget extensions. The context contains the model as one of its properties so that we can pass a single object around.

They are used to provide an abstracted interface to the session and Contents API from `@jupyterlab/services` for the given model. They can be shared between widgets.

The reason for a separate context and model is so that it is easy to create model factories and the heavy lifting of the context is left to the Document Manager. Contexts are not meant to be subclassed or re-implemented. Instead, the contexts are intended to be the glue between the document model and the wider application.

25.3 Document Manager

The *Document Manager* handles:

- document models

- document contexts

The *File Browser* uses the *Document Manager* to open documents and manage them.

CHAPTER

TWENTYSIX

NOTEBOOK

26.1 Background

A JupyterLab architecture walkthrough from June 16, 2016, provides an overview of the notebook architecture.

The most complicated plugin included in the **JupyterLab application** is the **Notebook plugin**.

The NotebookWidgetFactory constructs a new NotebookPanel from a model and populates the toolbar with default widgets.

26.2 Structure of the Notebook plugin

The Notebook plugin provides a model and widgets for dealing with notebook files.

26.2.1 Model

The NotebookModel contains an observable list of cells.

A cell model can be:

- a code cell
- a markdown cell
- raw cell

A code cell contains a list of **output models**. The list of cells and the list of outputs can be observed for changes.

Cell operations

The NotebookModel cell list supports single-step operations such as moving, adding, or deleting cells. Compound cell list operations, such as undo/redo, are also supported by the NotebookModel. Right now, undo/redo is only supported on cells and is not supported on notebook attributes, such as notebook metadata. Currently, undo/redo for individual cell input content is supported by the CodeMirror editor's undo feature. (Note: CodeMirror editor's undo does not cover cell metadata changes.)

Metadata

The notebook model and the cell model (i.e. notebook cells) support getting and setting metadata through an IObservableJSON object. You can use this to get and set notebook/cell metadata, as well as subscribe to changes to it.

26.2.2 Notebook widget

After the NotebookModel is created, the NotebookWidgetFactory constructs a new NotebookPanel from the model. The NotebookPanel widget is added to the DockPanel. The **NotebookPanel** contains:

- a Toolbar
- a Notebook widget.

The NotebookPanel also adds completion logic.

The **NotebookToolbar** maintains a list of widgets to add to the toolbar. The **Notebook widget** contains the rendering of the notebook and handles most of the interaction logic with the notebook itself (such as keeping track of interactions such as selected and active cells and also the current edit/command mode).

The NotebookModel cell list provides ways to do fine-grained changes to the cell list.

Higher level actions using NotebookActions

Higher-level actions are contained in the NotebookActions namespace, which has functions, when given a notebook widget, to run a cell and select the next cell, merge or split cells at the cursor, delete selected cells, etc.

Widget hierarchy

A Notebook widget contains a list of cell widgets, corresponding to the cell models in its cell list.

- Each cell widget contains an InputArea,
  - which contains n CodeEditorWrapper,
    - which contains a JavaScript CodeMirror instance.

A CodeCell also contains an OutputArea. An OutputArea is responsible for rendering the outputs in the OutputAreaModel list. An OutputArea uses a notebook-specific RenderMimeRegistry object to render `display_data` output messages.

Rendering output messages

A **Rendermime plugin** provides a pluggable system for rendering output messages. Default renderers are provided for markdown, html, images, text, etc. Extensions can register renderers to be used across the entire application by registering a handler and mimetype in the rendermime registry. When a notebook is created, it copies the global Rendermime singleton so that notebook-specific renderers can be added. The ipywidgets widget manager is an example of an extension that adds a notebook-specific renderer, since rendering a widget depends on notebook-specific widget state.

26.3 How to extend the Notebook plugin

We'll walk through two notebook extensions:

- adding a button to the toolbar
- adding an ipywidgets extension

26.3.1 Adding a button to the toolbar

Start from the cookie cutter extension template.

```
pip install cookiecutter
cookiecutter https://github.com/jupyterlab/extension-cookiecutter-ts
cd my-cookie-cutter-name
```

Install the dependencies. Note that extensions are built against the released npm packages, not the development versions.

```
npm install --save @jupyterlab/notebook @jupyterlab/application @jupyterlab/apputils␣
↪@jupyterlab/docregistry @phosphor/disposable
```

Copy the following to `src/index.ts`:

```
import {
  IDisposable, DisposableDelegate
} from '@phosphor/disposable';

import {
  JupyterFrontEnd, JupyterFrontEndPlugin
} from '@jupyterlab/application';

import {
  ToolbarButton
} from '@jupyterlab/apputils';

import {
  DocumentRegistry
} from '@jupyterlab/docregistry';

import {
  NotebookActions, NotebookPanel, INotebookModel
} from '@jupyterlab/notebook';


/**
 * The plugin registration information.
 */
const plugin: JupyterFrontEndPlugin<void> = {
  activate,
  id: 'my-extension-name:buttonPlugin',
  autoStart: true
};


/**
 * A notebook widget extension that adds a button to the toolbar.
 */
export
class ButtonExtension implements DocumentRegistry.IWidgetExtension<NotebookPanel,␣
↪INotebookModel> {
  /**
   * Create a new extension object.
   */
  createNew(panel: NotebookPanel, context: DocumentRegistry.IContext<INotebookModel>
↪): IDisposable {
```

(continues on next page)

(continued from previous page)

```
    let callback = () => {
      NotebookActions.runAll(panel.content, context.session);
    };
    let button = new ToolbarButton({
      className: 'myButton',
      iconClassName: 'fa fa-fast-forward',
      onClick: callback,
      tooltip: 'Run All'
    });

    panel.toolbar.insertItem(0, 'runAll', button);
    return new DisposableDelegate(() => {
      button.dispose();
    });
  }
}

/**
 * Activate the extension.
 */
function activate(app: JupyterFrontEnd) {
  app.docRegistry.addWidgetExtension('Notebook', new ButtonExtension());
};


/**
 * Export the plugin as default.
 */
export default plugin;
```

Run the following commands:

```
npm install
npm run build
jupyter labextension install .
jupyter lab
```

Open a notebook and observe the new "Run All" button.

26.3.2 The *ipywidgets* third party extension

This discussion will be a bit confusing since we've been using the term *widget* to refer to *phosphor widgets*. In the discussion below, *ipython widgets* will be referred to as *ipywidgets*. There is no intrinsic relation between *phosphor widgets* and *ipython widgets*.

The *ipywidgets* extension registers a factory for a notebook *widget* extension using the Document Registry. The `createNew()` function is called with a NotebookPanel and DocumentContext. The plugin then creates a ipywidget manager (which uses the context to interact the kernel and kernel's comm manager). The plugin then registers an ipywidget renderer with the notebook instance's rendermime (which is specific to that particular notebook).

When an ipywidget model is created in the kernel, a comm message is sent to the browser and handled by the ipywidget manager to create a browser-side ipywidget model. When the model is displayed in the kernel, a `display_data` output is sent to the browser with the ipywidget model id. The renderer registered in that notebook's rendermime is asked to render the output. The renderer asks the ipywidget manager instance to render the corresponding model, which returns a JavaScript promise. The renderer creates a container *phosphor widget* which it hands back synchronously to the OutputArea, and then fills the container with the rendered *ipywidget* when the promise resolves.

Note: The ipywidgets third party extension has not yet been released.

CHAPTER

TWENTYSEVEN

DESIGN PATTERNS

There are several design patterns that are repeated throughout the repository. This guide is meant to supplement the TypeScript Style Guide.

27.1 TypeScript

TypeScript is used in all of the source code. TypeScript is used because it provides features from the most recent EMCAScript 6 standards, while providing type safety. The TypeScript compiler eliminates an entire class of bugs, while making it much easier to refactor code.

27.2 Initialization Options

Objects will typically have an `IOptions` interface for initializing the widget. The use of this interface enables options to be later added while preserving backward compatibility.

27.3 ContentFactory Option

A common option for a widget is a `IContentFactory`, which is used to customize the child content in the widget. If not given, a `defaultRenderer` instance is used if no arguments are required. In this way, widgets can be customized without subclassing them, and widgets can support customization of their nested content.

27.4 Static Namespace

An object class will typically have an exported static namespace sharing the same name as the object. The namespace is used to declutter the class definition.

27.5 Private Module Namespace

The "Private" module namespace is used to group variables and functions that are not intended to be exported and may have otherwise existed as module-level variables and functions. The use of the namespace also makes it clear when a variable access is to an imported name or from the module itself. Finally, the namespace enables the entire section to be collapsed in an editor if desired.

27.6 Disposables

JavaScript does not support "destructors", so the `IDisposable` pattern is used to ensure resources are freed and can be claimed by the Garbage Collector when no longer needed. It should always be safe to `dispose()` of an object more than once. Typically the object that creates another object is responsible for calling the dispose method of that object unless explicitly stated otherwise.

To mirror the pattern of construction, `super.dispose()` should be called last in the `dispose()` method if there is a parent class. Make sure any signal connections are cleared in either the local or parent `dispose()` method. Use a sentinel value to guard against reentry, typically by checking if an internal value is null, and then immediately setting the value to null. A subclass should never override the `isDisposed` getter, because it short-circuits the parent class getter. The object should not be considered disposed until the base class `dispose()` method is called.

27.7 Messages

Messages are intended for many-to-one communication where outside objects influence another object. Messages can be conflated and processed as a single message. They can be posted and handled on the next animation frame.

27.8 Signals

Signals are intended for one-to-many communication where outside objects react to changes on another object. Signals are always emitted with the sender as the first argument, and contain a single second argument with the payload. Signals should generally not be used to trigger the "default" behavior for an action, but to enable others to trigger additional behavior. If a "default" behavior is intended to be provided by another object, then a callback should be provided by that object. Wherever possible as signal connection should be made with the pattern `.connect(this._onFoo, this)`. Providing the `this` context enables the connection to be properly cleared by `clearSignalData(this)`. Using a private method avoids allocating a closure for each connection.

27.9 Models

Some of the more advanced widgets have a model associated with them. The common pattern used is that the model is settable and must be set outside of the constructor. This means that any consumer of the widget must account for a model that may be `null`, and may change at any time. The widget should emit a `modelChanged` signal to enable consumers to handle a change in model. The reason to enable a model to swap is that the same widget could be used to display different model content while preserving the widget's location in the application. The reason the model cannot be provided in the constructor is the initialization required for a model may have to call methods that are subclassed. The subclassed methods would be called before the subclass constructor has finished evaluating, resulting in undefined state.

27.10 Getters vs. Methods

Prefer a method when the return value must be computed each time. Prefer a getter for simple attribute lookup. A getter should yield the same value every time.

27.11 Data Structures

For public API, we have three options: JavaScript `Array`, `IIterator`, and `ReadonlyArray` (an interface defined by TypeScript).

Prefer an `Array` for:

- A value that is meant to be mutable.

Prefer a `ReadonlyArray`

- A return value is the result of a newly allocated array, to avoid the extra allocation of an iterator.
- A signal payload - since it will be consumed by multiple listeners.
- The values may need to be accessed randomly.
- A public attribute that is inherently static.

Prefer an `IIterator` for:

- A return value where the value is based on an internal data structure but the value should not need to be accessed randomly.
- A set of return values that can be computed lazily.

27.12 DOM Events

If an object instance should respond to DOM events, create a `handleEvent` method for the class and register the object instance as the event handler. The `handleEvent` method should switch on the event type and could call private methods to carry out the actions. Often a widget class will add itself as an event listener to its own node in the `onAfterAttach` method with something like `this.node.addEventListener('mousedown', this)` and unregister itself in the `onBeforeDetach` method with `this.node.removeEventListener('mousedown', this)` Dispatching events from the `handleEvent` method makes it easier to trace, log, and debug event handling. For more information about the `handleEvent` method, see the EventListener API.

27.13 Promises

We use Promises for asynchronous function calls, and a shim for browsers that do not support them. When handling a resolved or rejected Promise, make sure to check for the current state (typically by checking an `.isDisposed` property) before proceeding.

27.14 Command Names

Commands used in the application command registry should be formatted as follows: `package-name:verb-noun`. They are typically grouped into a `CommandIDs` namespace in the extension that is not exported.

CHAPTER TWENTYEIGHT

CSS PATTERNS

This document describes the patterns we are using to organize and write CSS for JupyterLab. JupyterLab is developed using a set of npm packages that are located in `packages`. Each of these packages has its own style, but depend on CSS variables defined in a main theme package.

28.1 CSS checklist

- CSS classnames are defined inline in the code. We used to put them as all caps file-level `consts`, but we are moving away from that.
- CSS files for packages are located within the `style` subdirectory and imported into the plugin's `index.css`.
- The JupyterLab default CSS variables in the `theme-light-extension` and `theme-dark-extension` packages are used to style packages wherever possible. Individual packages should not npm-depend on these packages though, to enable the theme to be swapped out.
- Additional public/private CSS variables are defined by plugins sparingly and in accordance with the conventions described below.

28.2 CSS variables

We are using native CSS variables in JupyterLab. This is to enable dynamic theming of built-in and third party plugins. As of December 2017, CSS variables are supported in the latest stable versions of all popular browsers, except for IE. If a JupyterLab deployment needs to support these browsers, a server side CSS preprocessor such as Myth or cssnext may be used.

28.2.1 Naming of CSS variables

We use the following convention for naming CSS variables:

- Start all CSS variables with `--jp-`.
- Words in the variable name should be lowercase and separated with `-`.
- The next segment should refer to the component and subcomponent, such as `--jp-notebook-cell-`.
- The next segment should refer to any state modifiers such as `active`, `not-active` or `focused`: `--jp-notebook-cell-focused`.
- The final segment will typically be related to a CSS properties, such as `color`, `font-size` or `background`: `--jp-notebook-cell-focused-background`.

28.2.2 Public/private

Some CSS variables in JupyterLab are considered part of our public API. Others are considered private and should not be used by third party plugins or themes. The difference between public and private variables is simple:

- All private variables begin with `--jp-private-`
- All variables without the `private-` prefix are public.
- Public variables should be defined under the `:root` pseudo-selector. This ensures that public CSS variables can be inspected under the top-level `<html>` tag in the browser's dev tools.
- Where possible, private variables should be defined and scoped under an appropriate selector other than `:root`.

28.2.3 CSS variable usage

JupyterLab includes a default set of CSS variables in the file:

`packages/theme-light-extension/style/variables.css`

To ensure consistent design in JupyterLab, all built-in and third party extensions should use these variables in their styles if at all possible. Documentation about those variables can be found in the `variables.css` file itself.

Plugins are free to define additional public and private CSS variables in their own `index.css` file, but should do so sparingly.

Again, we consider the names of the public CSS variables in this package to be our public API for CSS.

28.3 File organization

We are organizing our CSS files in the following manner:

- Each package in the top-level `packages` directory should contain any CSS files in a `style` subdirectory that are needed to style itself.
- All local styles should be consolidated into a `style/base.css` file.
- The top level `index.css` file is templated by `buildutils` as part of the `integrity` script. It imports the CSS in dependency order, ending with the local `./base.css`. CSS from external libraries is determined by their `style` field in `package.json`. If additional files are desired or the external library does not have a `style` field, we use the `jupyterlab: { "extraStyles": { "fooLibrary": ["path/to/css"] } }` pattern in our `package.json` to declare them. For imports that should not be added to `index.css`, update ``SKIP_CSS` in `buildutils/src/ensure-repo.ts`.

28.4 CSS class names

28.4.1 CSS class naming conventions

We have a fairly formal method for naming our CSS classes.

First, CSS class names are associated with TypeScript classes that extend `phosphor.Widget`:

The `.node` of each such widget should have a CSS class that matches the name of the TypeScript class:

```
class MyWidget extends Widget {

  constructor() {
    super();
    this.addClass('jp-MyWidget');
  }

}
```

Second, subclasses should have a CSS class for both the parent and child:

```
class MyWidgetSubclass extends MyWidget {

  constructor() {
    super(); // Adds `jp-MyWidget`
    this.addClass('jp-MyWidgetSubclass');
  }

}
```

In both of these cases, CSS class names with caps-case are reserved for situations where there is a named TypeScript `Widget` subclass. These classes are a way of a TypeScript class providing a public API for styling.

Third, children nodes of a `Widget` should have a third segment in the CSS class name that gives a semantic naming of the component, such as:

- `jp-MyWidget-toolbar`
- `jp-MyWidget-button`
- `jp-MyWidget-contentButton`

In general, the parent `MyWidget` should add these classes to the children. This applies when the children are plain DOM nodes or `Widget` instances/subclasses themselves. Thus, the general naming of CSS classes is of the form `jp-WidgetName-semanticChild`. This enables the styling of these children in a manner that is independent of the children implementation or CSS classes they have themselves.

Fourth, some CSS classes are used to modify the state of a widget:

- `jp-mod-active`: applied to elements in the active state
- `jp-mod-hover`: applied to elements in the hover state
- `jp-mod-selected`: applied to elements while selected

Fifth, some CSS classes are used to distinguish different types of a widget:

- `jp-type-separator`: applied to menu items that are separators
- `jp-type-directory`: applied to elements in the file browser that are directories

28.4.2 Edge cases

Over time, we have found that there are some edge cases that these rules don't fully address. Here, we try to clarify those edge cases.

When should a parent add a class to children?

Above, we state that a parent (`MyWidget`), should add CSS classes to children that indicate the semantic function of the child. Thus, the `MyWidget` subclass of `Widget` should add `jp-MyWidget` to itself and `jp-MyWidget-toolbar` to a toolbar child.

What if the child itself is a `Widget` and already has a proper CSS class name itself, such as `jp-Toolbar`? Why not use selectors such as `.jp-MyWidget .jp-Toolbar` or `.jp-MyWidget > .jp-Toolbar`?

The reason is that these selectors are dependent on the implementation of the toolbar having the `jp-Toolbar` CSS class. When `MyWidget` adds the `jp-MyWidget-toolbar` class, it can style the child independent of its implementation. The other reason to add the `jp-MyWidget-toolbar` class is if the DOM structure is highly recursive, the usual descendant selectors may not be specific to target only the desired children.

When in doubt, there is little harm done in parents adding selectors to children.

28.4.3 Commonly used CSS selectors

We use CSS selectors to decide which context menu items to display and what command to invoke when a keyboard shortcut is used. The following common CSS selectors are intended to be used for adding context menu items and keyboard shortcuts.

CSS classes that target widgets and their children

- `jp-Activity`: applied to elements in the main work area
- `jp-Cell`: applied to cells
- `jp-CodeCell`: applied to code cells
- `jp-CodeConsole`: applied to consoles
- `jp-CodeConsole-content`: applied to content panels in consoles
- `jp-CodeConsole-promptCell`: applied to active prompt cells in consoles
- `jp-DirListing-content`: applied to contents of file browser directory listings
- `jp-DirListing-item`: applied to items in file browser directory listings
- `jp-FileEditor`: applied to file editors
- `jp-ImageViewer`: applied to image viewers
- `jp-InputArea-editor`: applied to cell input area editors
- `jp-Notebook`: applied to notebooks
- `jp-SettingEditor`: applied to setting editors
- `jp-SideBar`: applied to sidebars
- `jp-Terminal`: applied to terminals

CSS classes that describe the state of a widget

- `jp-mod-current`: applied to elements on the current document only
- `jp-mod-completer-enabled`: applied to ediors that can host a completer
- `jp-mod-commandMode`: applied to a notebook in command mode
- `jp-mod-editMode`: applied to a notebook in edit mode
- `jp-mod-has-primary-selection`: applied to editors that have a primary selection
- `jp-mod-in-leading-whitespace`: applied to editors that have a selection within the beginning whitespace of a line
- `jp-mod-tooltip`: applied to the body when a tooltip exists on the page

CSS selectors that target data attributes

- `[data-jp-code-runner]`: applied to widgets that can run code
- `[data-jp-interaction-mode="terminal"]`: applied when a code console is in terminal mode
- `[data-jp-interaction-mode="notebook"]`: applied when a code console is in notebook mode
- `[data-jp-isdir]`: applied to describe whether file browser items are directories
- `[data-jp-undoer]`: applied to widgets that can undo
- `[data-type]`: applied to describe the type of element, such as "document-title", "submenu", "inline"

CHAPTER

TWENTYNINE

REACT

Many JupyterLab APIs require Phosphor Widgets which have some additional features over native DOM elements, including:

- Resize events that propagate down the Widget hierarchy.
- Lifecycle events (`onBeforeDetach`, `onAfterAttach`, etc.).
- Both CSS-based and absolutely positioned layouts.

We support wrapping React components to turn them into Phosphor widgets using the `ReactWidget` class from `@jupyterlab/apputils`:

```
import * as React from 'react';

import { Widget } from '@phosphor/widgets';
import { ReactWidget } from '@jupyterlab/apputils';

function MyComponent() {
  return <div>My Widget</div>;
}

const myWidget: Widget = ReactWidget.create(<MyComponent />);
```

Here we use the `create` static method to transform a React element into a Phosphor widget. Whenever the widget is mounted, the React element will be rendered on the page.

If you need to handle other life cycle events on the Phosphor widget or add other methods to it, you can subbclass `ReactWidget` and override the `render` method to return a React element:

```
import * as React from 'react';

import { Widget } from '@phosphor/widgets';
import { ReactWidget } from '@jupyterlab/apputils';

function MyComponent() {
  return <div>My Widget</div>;
}
class MyWidget extends ReactWidget {
  render() {
    return <MyComponent />;
  }
}
const myWidget: Widget = new MyWidget();
```

We use Phosphor Signals to represent data that changes over time in JupyterLab. To have your React element change in response to a signal event, use the `UseSignal` component, which implements the "render props".

The running component and the `createSearchOverlay` function in the search overlay use both of these features and serve as a good reference for best practices.

We currently do not have a way of embedding Phosphor widgets inside of React components. If you find yourself trying to do this, we would recommend either converting the Phosphor widget to a React component or using a Phosphor widget for the outer layer component.

We follow the React documentation and "React & Redux in TypeScript - Static Typing Guide" for best practices on using React in TypeScript.

CHAPTER THIRTY

EXAMPLES

The `examples` directory in the JupyterLab repo contains:

- several stand-alone examples (`console`, `filebrowser`, `notebook`, `terminal`)
- a more complex example (`lab`).

Installation instructions for the examples are found in the project's README.

After installing the jupyter notebook server 4.2+, follow the steps for installing the development version of JupyterLab. To build the examples, enter from the `jupyterlab` repo root directory:

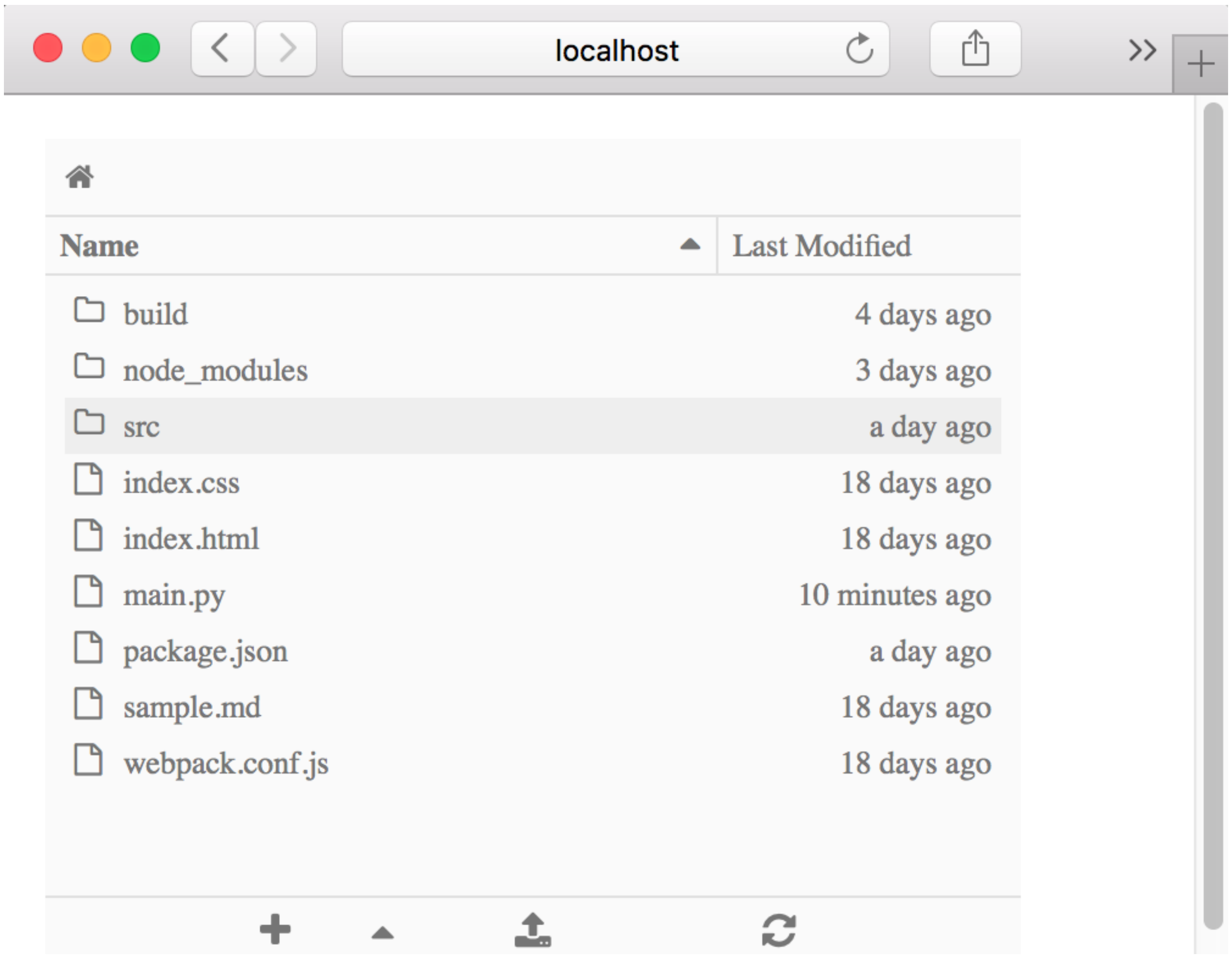
```
jlpm run build:examples
```

To run a particular example, navigate to the example's subdirectory in the `examples` directory and enter:

```
python main.py
```

30.1 Dissecting the 'filebrowser' example

The filebrowser example provides a stand-alone implementation of a filebrowser. Here's what the filebrowser's user interface looks like:

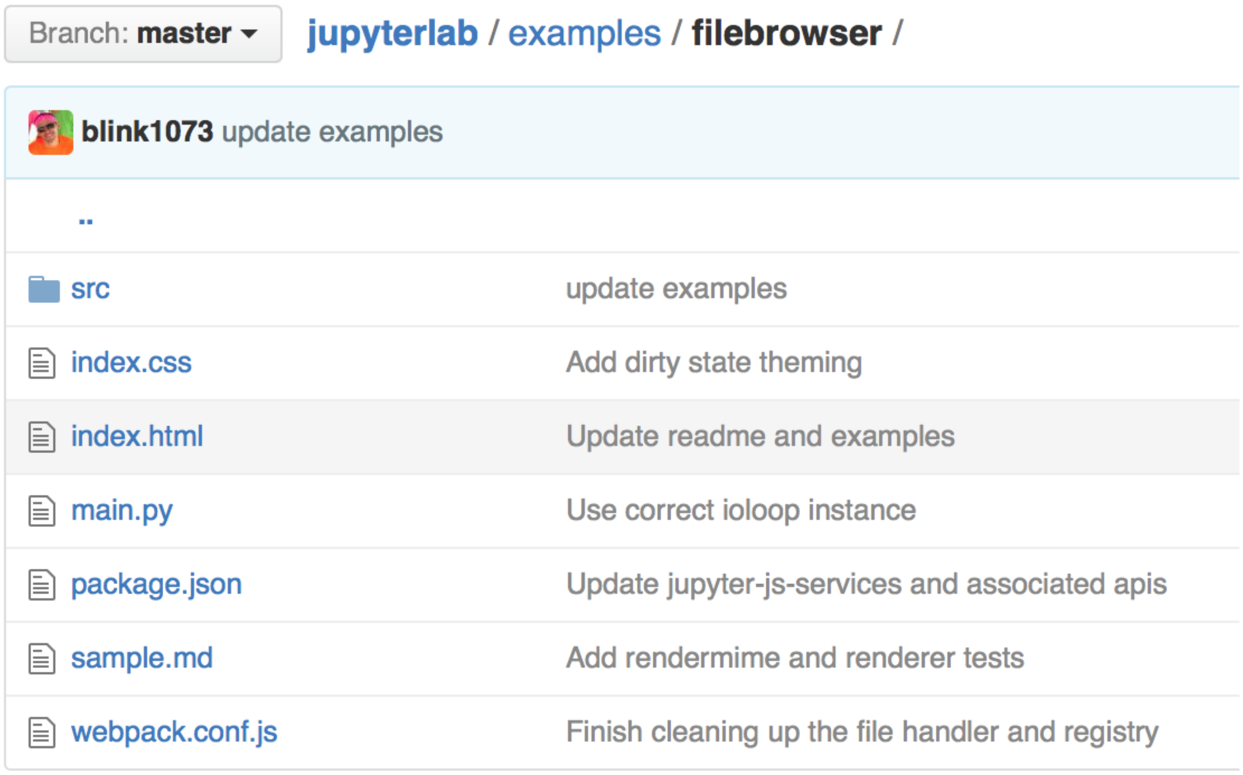


Name	Last Modified
build	4 days ago
node_modules	3 days ago
src	a day ago
index.css	18 days ago
index.html	18 days ago
main.py	10 minutes ago
package.json	a day ago
sample.md	18 days ago
webpack.conf.js	18 days ago

Let's take a closer look at the source code in `examples/filebrowser`.

30.1.1 Directory structure of 'filebrowser' example

The filebrowser in `examples/filebrowser` is comprised by a handful of files and the `src` directory:



Branch: **master** ▾ jupyterlab / examples / **filebrowser** /

blink1073 update examples

..

src	update examples
index.css	Add dirty state theming
index.html	Update readme and examples
main.py	Use correct ioloop instance
package.json	Update jupyter-js-services and associated apis
sample.md	Add rendermime and renderer tests
webpack.conf.js	Finish cleaning up the file handler and registry

The filebrowser example has two key source files:

- `src/index.ts`: the TypeScript file that defines the functionality
- `main.py`: the Python file that enables the example to be run

Reviewing the source code of each file will help you see the role that each file plays in the stand-alone filebrowser example.

CHAPTER

THIRTYONE

USER INTERFACE HELPERS

JupyterLab comes with helpers to show or request simple information from a user. Those speed up development and ensure a common look and feel.

31.1 Dialogs

31.1.1 Message Dialogs

Helper functions to show a message to the user are available in the `apputils` package. These dialogs return a `Promise` resolving when the user dismisses the dialog.

There is one helper:

- `showErrorMessage` : show an error message dialog.

31.1.2 Input Dialogs

Helper functions to request a single input from the user are available in the `apputils` package within the `InputDialog` namespace. There are four helpers:

- `getBoolean` : request a boolean through a checkbox.
- `getItem` : request a item from a list; the list may be editable.
- `getNumber` : request a number; if the user input is not a valid number, NaN is returned.
- `getText` : request a short text.

All dialogs are built on the standard `Dialog`. Therefore the helper functions each return a `Promise` resolving in a `Dialog.IResult` object.

```
// Request a boolean
InputDialog.getBoolean({ title: 'Check or not?' }).then(value => {
  console.log('boolean ' + value.value);
});

// Request a choice from a list
InputDialog.getItem({
  title: 'Pick a choice',
  items: ['1', '2']
}).then(value => {
  console.log('item ' + value.value);
});
```

(continues on next page)

(continued from previous page)

```
// Request a choice from a list or specify your own choice
InputDialog.getItem({
  title: 'Pick a choice or write your own',
  items: ['1', '2'],
  editable: true
}).then(value => {
  console.log('editable item ' + value.value);
});

// Request a number
InputDialog.getNumber({ title: 'How much?' }).then(value => {
  console.log('number ' + value.value);
});

// Request a text
InputDialog.getText({ title: 'Provide a text' }).then(value => {
  console.log('text ' + value.value);
});
```

31.1.3 File Dialogs

Two helper functions to ask a user to open a file or a directory are available in the `filebrowser` package under the namespace `FileDialog`.

Here is an example to request a file.

```
const dialog = FileDialog.getExistingDirectory({
  iconRegistry, // IIconRegistry
  manager, // IDocumentManager
  filter: model => model.type == 'notebook' // optional (model: Contents.IModel) =>␣
↪boolean
});

const result = await dialog;

if(result.button.accept){
  let files = result.value;
}
```

And for a folder.

```
const dialog = FileDialog.getExistingDirectory({
  iconRegistry, // IIconRegistry
  manager // IDocumentManager
});

const result = await dialog;

if(result.button.accept){
  let folders = result.value;
}
```

Note: The document manager and the icon registry can be obtained in a plugin by requesting

`IFileBrowserFactory` token. The `manager` will be accessed through `factory.defaultBrowser.model.manager` and the `iconRegistry` through `factory.defaultBrowser.model.iconRegistry`.

CHAPTER

THIRTYTWO

TERMINOLOGY

Learning to use a new technology and its architecture can be complicated by the jargon used to describe components. We provide this terminology guide to help smooth the learning the components.

32.1 Terms

- *Application* - The main application object that hold the application shell, command registry, and keymap registry. It is provided to all plugins in their activate method.
- *Extension* - an npm package containing one or more *Plugins* that can be used to extend JupyterLab's functionality.
- *Plugin* - An object that provides a service and or extends the application.
- *Phosphor* - The JavaScript library that provides the foundation of JupyterLab, enabling desktop-like applications in the browser.
- *Standalone example* - An example in the `examples/` directory that demonstrates the usage of one or more components of JupyterLab.
- TypeScript - A statically typed language that compiles to JavaScript.

CHAPTER

THIRTYTHREE

LET'S MAKE AN ASTRONOMY PICTURE OF THE DAY JUPYTERLAB EXTENSION

JupyterLab extensions add features to the user experience. This page describes how to create one type of extension, an *application plugin*, that:

- Adds a "Random Astronomy Picture" command to the *command palette* sidebar
- Fetches the image and metadata when activated
- Shows the image and metadata in a tab panel

By working through this tutorial, you'll learn:

- How to set up an extension development environment from scratch on a Linux or OSX machine. (You'll need to modify the commands slightly if you are on Windows.)
- How to start an extension project from jupyterlab/extension-cookiecutter-ts
- How to iteratively code, build, and load your extension in JupyterLab
- How to version control your work with git
- How to release your extension for others to enjoy

Sound like fun? Excellent. Here we go!

33.1 Set up a development environment

33.1.1 Install conda using miniconda

Start by installing miniconda, following Conda's installation documentation.

33.1.2 Install NodeJS, JupyterLab, etc. in a conda environment

Next create a conda environment that includes:

1. the latest release of JupyterLab
2. cookiecutter, the tool you'll use to bootstrap your extension project structure (this is a Python tool which we'll install using conda below).
3. NodeJS, the JavaScript runtime you'll use to compile the web assets (e.g., TypeScript, CSS) for your extension
4. git, a version control system you'll use to take snapshots of your work as you progress through this tutorial

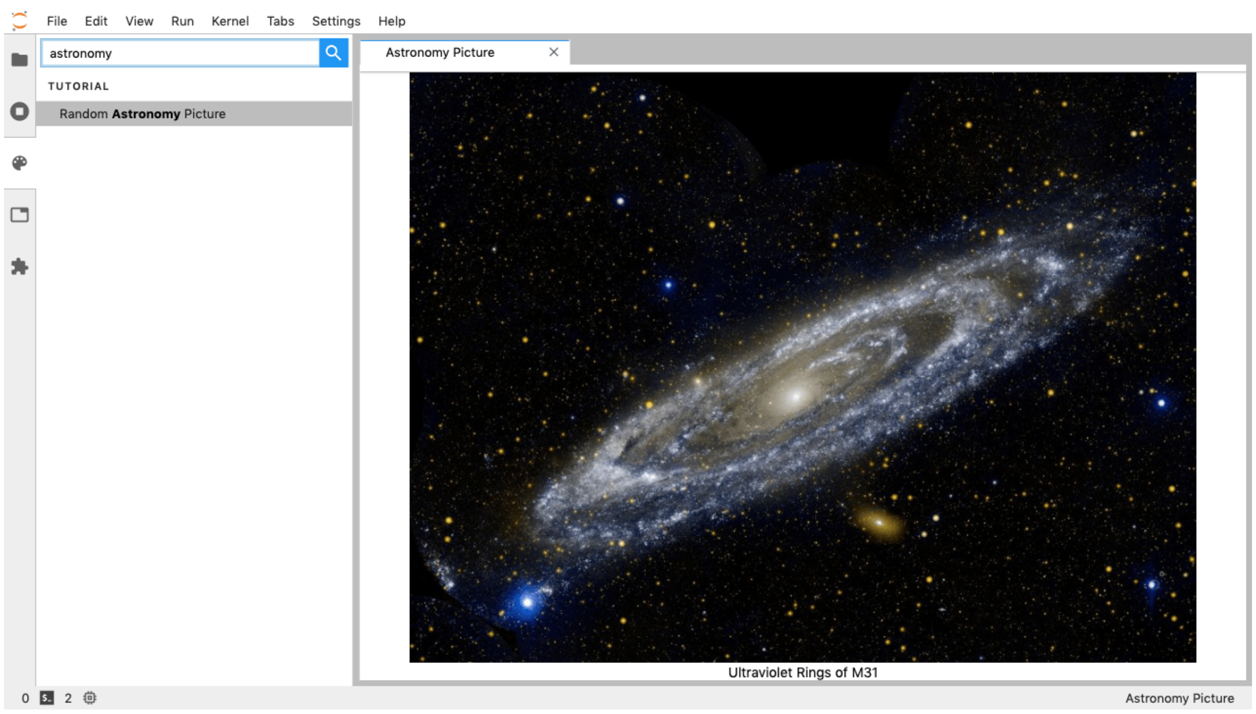


Fig. 1: The completed extension, showing the Astronomy Picture of the Day for 24 Jul 2015.

It's best practice to leave the root conda environment (i.e., the environment created by the miniconda installer) untouched and install your project-specific dependencies in a named conda environment. Run this command to create a new environment named `jupyterlab-ext`.

```
conda create -n jupyterlab-ext --override-channels --strict-channel-priority -c conda-
↪forge -c anaconda jupyterlab cookiecutter nodejs git
```

Now activate the new environment so that all further commands you run work out of that environment.

```
conda activate jupyterlab-ext
```

Note: You'll need to run the command above in each new terminal you open before you can work with the tools you installed in the `jupyterlab-ext` environment.

Note: if you have an older version of JupyterLab previously installed, you may need to update the version of JupyterLab manually.

```
conda install -c conda-forge jupyterlab=1
```

33.2 Create a repository

Create a new repository for your extension (see, for example, the GitHub instructions. This is an optional step, but highly recommended if you want to share your extension.

33.3 Create an extension project

33.3.1 Initialize the project from a cookiecutter

Next use cookiecutter to create a new project for your extension. This will create a new folder for your extension in your current directory.

```
cookiecutter https://github.com/jupyterlab/extension-cookiecutter-ts --checkout v1.0
```

When prompted, enter values like the following for all of the cookiecutter prompts (`apod` stands for Astronomy Picture of the Day, the NASA service we are using to fetch pictures).

```
author_name []: Your Name
extension_name [myextension]: jupyterlab_apod
project_short_description [A JupyterLab extension.]: Show a random NASA Astronomy␣
↪Picture of the Day in a JupyterLab panel
repository [https://github.com/my_name/jupyterlab_myextension]: https://github.com/my_
↪name/jupyterlab_apod
```

Note: if not using a repository, leave the repository field blank. You can come back and edit the repository field in the `package.json` file later.

Change to the directory the cookiecutter created and list the files.

```
cd jupyterlab_apod
ls
```

You should see a list like the following.

```
README.md     package.json  src           style         tsconfig.json
```

33.3.2 Build and install the extension for development

Your new extension project has enough code in it to see it working in your JupyterLab. Run the following commands to install the initial project dependencies and install it in the JupyterLab environment. We defer building since it will be built in the next step.

Note: This tutorial uses `jlpm` to install Javascript packages and run build commands, which is JupyterLab's bundled version of `yarn`. If you prefer, you can use another Javascript package manager like `npm` or `yarn` itself.

```
jlpm install
jupyter labextension install . --no-build
```

After the install completes, open a second terminal. Run these commands to activate the `jupyterlab-ext` environment and to start a JupyterLab instance in watch mode so that it will keep up with our changes as we make them.

```
conda activate jupyterlab-ext
jupyter lab --watch
```

33.3.3 See the initial extension in action

After building with your extension, JupyterLab should open in your default web browser.

In that browser window, open the JavaScript console by following the instructions for your browser:

- Accessing the DevTools in Google Chrome
- Opening the Web Console in Firefox

After you reload the page with the console open, you should see a message that says `JupyterLab extension jupyterlab_apod is activated!` in the console. If you do, congratulations, you're ready to start modifying the extension! If not, go back make sure you didn't miss a step, and reach out if you're stuck.

Note: Leave the terminal running the `jupyter lab --watch` command open.

33.3.4 Commit what you have to git

Run the following commands in your `jupyterlab_apod` folder to initialize it as a git repository and commit the current code.

```
git init
git add .
git commit -m 'Seed apod project from cookiecutter'
```

Note: This step is not technically necessary, but it is good practice to track changes in version control system in case you need to rollback to an earlier version or want to collaborate with others. For example, you can compare your work throughout this tutorial with the commits in a reference version of `jupyterlab_apod` on GitHub at https://github.com/jupyterlab/jupyterlab_apod.

33.4 Add an Astronomy Picture of the Day widget

33.4.1 Show an empty panel

The *command palette* is the primary view of all commands available to you in JupyterLab. For your first addition, you're going to add a *Random Astronomy Picture* command to the palette and get it to show an *Astronomy Picture* tab panel when invoked.

Fire up your favorite text editor and open the `src/index.ts` file in your extension project. Change the import at the top of the file to get a reference to the command palette interface and the Jupyter front end.

```
import {
  JupyterFrontEnd, JupyterFrontEndPlugin
} from '@jupyterlab/application';

import {
  ICommandPalette
} from '@jupyterlab/apputils';
```

Locate the `extension` object of type `JupyterFrontEndPlugin`. Change the definition so that it reads like so:

```
/**
 * Initialization data for the jupyterlab_apod extension.
 */
const extension: JupyterFrontEndPlugin<void> = {
```

(continues on next page)

(continued from previous page)

```
  id: 'jupyterlab_apod',
  autoStart: true,
  requires: [ICommandPalette],
  activate: (app: JupyterFrontEnd, palette: ICommandPalette) => {
    console.log('JupyterLab extension jupyterlab_apod is activated!');
    console.log('ICommandPalette:', palette);
  }
};
```

The `requires` attribute states that your plugin needs an object that implements the `ICommandPalette` interface when it starts. JupyterLab will pass an instance of `ICommandPalette` as the second parameter of `activate` in order to satisfy this requirement. Defining `palette: ICommandPalette` makes this instance available to your code in that function. The second `console.log` line exists only so that you can immediately check that your changes work.

Now you will need to install these dependencies. Run the following commands in the repository root folder to install the dependencies and save them to your *package.json*:

```
jlpm add @jupyterlab/apputils
jlpm add @jupyterlab/application
```

Finally, run the following to rebuild your extension.

```
jlpm run build
```

JupyterLab will rebuild after the extension does. You can see it's progress in the `jupyter lab --watch` window. After that finishes, return to the browser tab that opened when you started JupyterLab. Refresh it and look in the console. You should see the same activation message as before, plus the new message about the ICommandPalette instance you just added. If you don't, check the output of the build command for errors and correct your code.

```
JupyterLab extension jupyterlab_apod is activated!
ICommandPalette: Palette {_palette: CommandPalette}
```

Note that we had to run `jlpm run build` in order for the bundle to update, because it is using the compiled JavaScript files in `/lib`. If you wish to avoid running `jlpm run build` after each change, you can open a third terminal, and run the `jlpm run watch` command from your extension directory, which will automatically compile the TypeScript files as they change.

Now return to your editor. Modify the imports at the top of the file to add a few more imports:

```
import {
  ICommandPalette, MainAreaWidget
} from '@jupyterlab/apputils';

import {
  Widget
} from '@phosphor/widgets';
```

Install this new dependency as well:

```
jlpm add @phosphor/widgets
```

Then modify the `activate` function again so that it has the following code:

```
activate: (app: JupyterFrontEnd, palette: ICommandPalette) => {
  console.log('JupyterLab extension jupyterlab_apod is activated!');
```

(continues on next page)

(continued from previous page)

```
  // Create a blank content widget inside of a MainAreaWidget
  const content = new Widget();
  const widget = new MainAreaWidget({content});
  widget.id = 'apod-jupyterlab';
  widget.title.label = 'Astronomy Picture';
  widget.title.closable = true;

  // Add an application command
  const command: string = 'apod:open';
  app.commands.addCommand(command, {
    label: 'Random Astronomy Picture',
    execute: () => {
      if (!widget.isAttached) {
        // Attach the widget to the main work area if it's not there
        app.shell.add(widget, 'main');
      }
      // Activate the widget
      app.shell.activateById(widget.id);
    }
  });

  // Add the command to the palette.
  palette.addItem({command, category: 'Tutorial'});
}
```

The first new block of code creates a `MainAreaWidget` instance with an empty content `Widget` as its child. It also assigns the main area widget a unique ID, gives it a label that will appear as its tab title, and makes the tab closable by the user. The second block of code adds a new command with id `apod:open` and label *Random Astronomy Picture* to JupyterLab. When the command executes, it attaches the widget to the main display area if it is not already present and then makes it the active tab. The last new line of code uses the command id to add the command to the command palette in a section called *Tutorial*.

Build your extension again using `jlpm run build` (unless you are using `jlpm run watch` already) and refresh the browser tab. Open the command palette on the left side by clicking on *Commands* and type *Astronomy* in the search box. Your *Random Astronomy Picture* command should appear. Click it or select it with the keyboard and press *Enter*. You should see a new, blank panel appear with the tab title *Astronomy Picture*. Click the *x* on the tab to close it and activate the command again. The tab should reappear. Finally, click one of the launcher tabs so that the *Astronomy Picture* panel is still open but no longer active. Now run the *Random Astronomy Picture* command one more time. The single *Astronomy Picture* tab should come to the foreground.

If your widget is not behaving, compare your code with the reference project state at the 01-show-a-panel tag. Once you've got everything working properly, git commit your changes and carry on.

```
git add .
git commit -m 'Show Astronomy Picture command in palette'
```

33.4.2 Show a picture in the panel

You now have an empty panel. It's time to add a picture to it. Go back to your code editor. Add the following code below the lines that create a `MainAreaWidget` instance and above the lines that define the command.

```
// Add an image element to the content
let img = document.createElement('img');
```

(continues on next page)

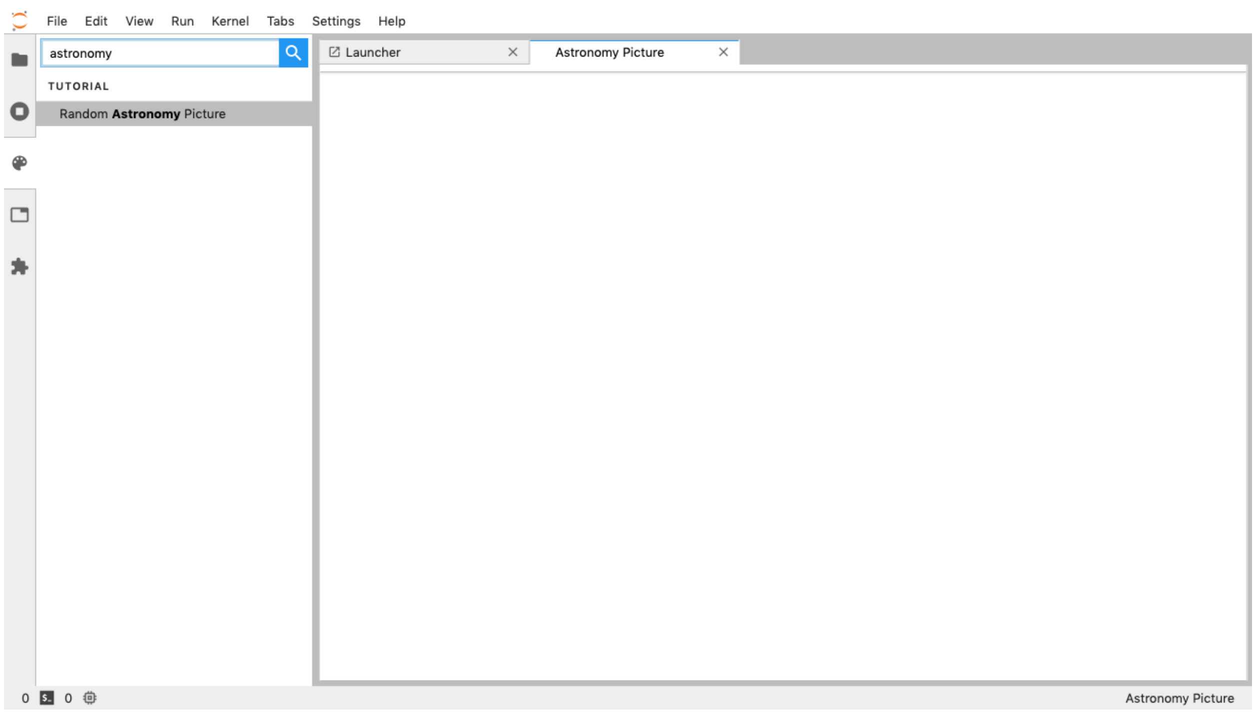


Fig. 2: The in-progress extension, showing a blank panel.

(continued from previous page)

```
content.node.appendChild(img);

// Get a random date string in YYYY-MM-DD format
function randomDate() {
  const start = new Date(2010, 1, 1);
  const end = new Date();
  const randomDate = new Date(start.getTime() + Math.random()*(end.getTime() - start.
↪getTime()));
  return randomDate.toISOString().slice(0, 10);
}

// Fetch info about a random picture
const response = await fetch(`https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY&
↪date=${randomDate()}`);
const data = await response.json() as APODResponse;

if (data.media_type === 'image') {
  // Populate the image
  img.src = data.url;
  img.title = data.title;
} else {
  console.log('Random APOD was not a picture.');
}
```

The first two lines create a new HTML `<img>` element and add it to the widget DOM node. The next lines define a function get a random date in the form `YYYY-MM-DD` format, and then the function is used to make a request using the HTML fetch API that returns information about the Astronomy Picture of the Day for that date. Finally, we set the image source and title attributes based on the response.

Now define the `APODResponse` type that was introduced in the code above. Put this definition just under the imports at the top of the file.

```
interface APODResponse {
  copyright: string;
  date: string;
  explanation: string;
  media_type: 'video' | 'image';
  title: string;
  url: string;
};
```

And update the `activate` method to be `async` since we are now using `await` in the method body.

```
activate: async (app: JupyterFrontEnd, palette: ICommandPalette) =>
```

Rebuild your extension if necessary (`jlpm run build`), refresh your browser tab, and run the *Random Astronomy Picture* command again. You should now see a picture in the panel when it opens (if that random date had a picture and not a video).

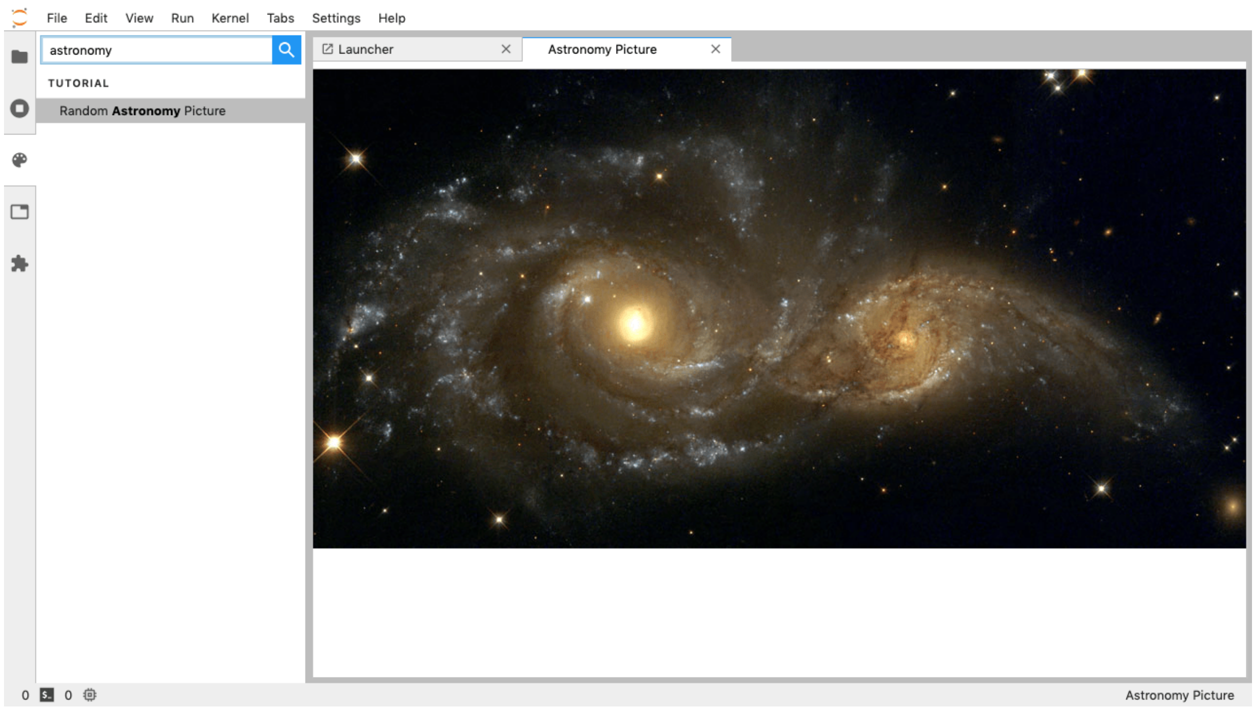


Fig. 3: The in-progress extension, showing the Astronomy Picture of the Day for 19 Jan 2014.

Note that the image is not centered in the panel nor does the panel scroll if the image is larger than the panel area. Also note that the image does not update no matter how many times you close and reopen the panel. You'll address both of these problems in the upcoming sections.

If you don't see a image at all, compare your code with the 02-show-an-image tag in the reference project. When it's working, make another git commit.

```
git add .
git commit -m 'Show a picture in the panel'
```

33.5 Improve the widget behavior

33.5.1 Center the image, add attribution, and error messaging

Open `style/index.css` in our extension project directory for editing. Add the following lines to it.

```
.my-apodWidget {
  display: flex;
  flex-direction: column;
  align-items: center;
  overflow: auto;
}
```

This CSS stacks content vertically within the widget panel and lets the panel scroll when the content overflows. This CSS file is included on the page automatically by JupyterLab because the `package.json` file has a `style` field pointing to it. In general, you should import all of your styles into a single CSS file, such as this `index.css` file, and put the path to that CSS file in the `package.json` file `style` field.

Return to the `index.ts` file. Modify the `activate` function to apply the CSS classes, the copyright information, and error handling for the API response. The beginning of the function should read like the following:

```
activate: async (app: JupyterFrontEnd, palette: ICommandPalette) => {
  console.log('JupyterLab extension jupyterlab_apod is activated!');

  // Create a blank content widget inside of a MainAreaWidget
  const content = new Widget();
  content.addClass('my-apodWidget'); // new line
  const widget = new MainAreaWidget({content});
  widget.id = 'apod-jupyterlab';
  widget.title.label = 'Astronomy Picture';
  widget.title.closable = true;

  // Add an image element to the content
  let img = document.createElement('img');
  content.node.appendChild(img);

  let summary = document.createElement('p');
  content.node.appendChild(summary);

  // Get a random date string in YYYY-MM-DD format
  function randomDate() {
    const start = new Date(2010, 1, 1);
    const end = new Date();
    const randomDate = new Date(start.getTime() + Math.random()*(end.getTime() -␣
↪start.getTime()));
    return randomDate.toISOString().slice(0, 10);
  }

  // Fetch info about a random picture
  const response = await fetch(`https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY&
↪date=${randomDate()}`);
  if (!response.ok) {
    const data = await response.json();
    if (data.error) {
      summary.innerText = data.error.message;
    } else {
      summary.innerText = response.statusText;
```

(continues on next page)

(continued from previous page)

```
    }
  } else {
    const data = await response.json() as APODResponse;

    if (data.media_type === 'image') {
      // Populate the image
      img.src = data.url;
      img.title = data.title;
      summary.innerText = data.title;
      if (data.copyright) {
        summary.innerText += ` (Copyright ${data.copyright})`;
      }
    } else {
      summary.innerText = 'Random APOD fetched was not an image.';
    }
  }

// Keep all the remaining fetch and command lines the same
// as before from here down ...
```

Build your extension if necessary (`jlpm run build`) and refresh your JupyterLab browser tab. Invoke the *Random Astronomy Picture* command and confirm the image is centered with the copyright information below it. Resize the browser window or the panel so that the image is larger than the available area. Make sure you can scroll the panel over the entire area of the image.

If anything is not working correctly, compare your code with the reference project 03-style-and-attribute tag. When everything is working as expected, make another commit.

```
git add .
git commit -m 'Add styling, attribution, error handling'
```

33.5.2 Show a new image on demand

The `activate` function has grown quite long, and there's still more functionality to add. Let's refactor the code into two separate parts:

1. An `APODWidget` that encapsulates the Astronomy Picture panel elements, configuration, and soon-to-be-added update behavior
2. An `activate` function that adds the widget instance to the UI and decide when the picture should refresh

Start by refactoring the widget code into the new `APODWidget` class. Add the following additional import to the top of the file.

```
import {
  Message
} from '@phosphor/messaging';
```

Install this dependency:

```
jlpm add @phosphor/messaging
```

Then add the class just below the import statements in the `index.ts` file.

```
class APODWidget extends Widget {
  /**
  * Construct a new APOD widget.
  */
  constructor() {
    super();

    this.addClass('my-apodWidget');

    // Add an image element to the panel
    this.img = document.createElement('img');
    this.node.appendChild(this.img);

    // Add a summary element to the panel
    this.summary = document.createElement('p');
    this.node.appendChild(this.summary);
  }

  /**
  * The image element associated with the widget.
  */
  readonly img: HTMLImageElement;

  /**
  * The summary text element associated with the widget.
  */
  readonly summary: HTMLParagraphElement;

  /**
  * Handle update requests for the widget.
  */
  async onUpdateRequest(msg: Message): Promise<void> {

    const response = await fetch(`https://api.nasa.gov/planetary/apod?api_key=DEMO_
↪KEY&date=${this.randomDate()}`);

    if (!response.ok) {
      const data = await response.json();
      if (data.error) {
        this.summary.innerText = data.error.message;
      } else {
        this.summary.innerText = response.statusText;
      }
      return;
    }

    const data = await response.json() as APODResponse;

    if (data.media_type === 'image') {
      // Populate the image
      this.img.src = data.url;
      this.img.title = data.title;
      this.summary.innerText = data.title;
      if (data.copyright) {
        this.summary.innerText += ` (Copyright ${data.copyright})`;
      }
    } else {
```

(continues on next page)

(continued from previous page)

```
      this.summary.innerText = 'Random APOD fetched was not an image.';
    }
  }

  /**
  * Get a random date string in YYYY-MM-DD format.
  */
  randomDate(): string {
    const start = new Date(2010, 1, 1);
    const end = new Date();
    const randomDate = new Date(start.getTime() + Math.random()*(end.getTime() -␣
↪start.getTime()));
    return randomDate.toISOString().slice(0, 10);
  }
}
```

You've written all of the code before. All you've done is restructure it to use instance variables and move the image request to its own function.

Next move the remaining logic in `activate` to a new, top-level function just below the `APODWidget` class definition. Modify the code to create a widget when one does not exist in the main JupyterLab area or to refresh the image in the exist widget when the command runs again. The code for the `activate` function should read as follows after these changes:

```
/**
* Activate the APOD widget extension.
*/
function activate(app: JupyterFrontEnd, palette: ICommandPalette) {
  console.log('JupyterLab extension jupyterlab_apod is activated!');

  // Create a single widget
  const content = new APODWidget();
  const widget = new MainAreaWidget({content});
  widget.id = 'apod-jupyterlab';
  widget.title.label = 'Astronomy Picture';
  widget.title.closable = true;

  // Add an application command
  const command: string = 'apod:open';
  app.commands.addCommand(command, {
    label: 'Random Astronomy Picture',
    execute: () => {
      if (!widget.isAttached) {
        // Attach the widget to the main work area if it's not there
        app.shell.add(widget, 'main');
      }
      // Refresh the picture in the widget
      content.update();
      // Activate the widget
      app.shell.activateById(widget.id);
    }
  });

  // Add the command to the palette.
  palette.addItem({ command, category: 'Tutorial' });
}
```

Remove the `activate` function definition from the `JupyterFrontEndPlugin` object and refer instead to the top-level function like this:

```
const extension: JupyterFrontEndPlugin<void> = {
  id: 'jupyterlab_apod',
  autoStart: true,
  requires: [ICommandPalette],
  activate: activate
};
```

Make sure you retain the `export default extension;` line in the file. Now build the extension again and refresh the JupyterLab browser tab. Run the *Random Astronomy Picture* command more than once without closing the panel. The picture should update each time you execute the command. Close the panel, run the command, and it should both reappear and show a new image.

If anything is not working correctly, compare your code with the 04-refactor-and-refresh tag to debug. Once it is working properly, commit it.

```
git add .
git commit -m 'Refactor, refresh image'
```

33.5.3 Restore panel state when the browser refreshes

You may notice that every time you refresh your browser tab, the Astronomy Picture panel disappears, even if it was open before you refreshed. Other open panels, like notebooks, terminals, and text editors, all reappear and return to where you left them in the panel layout. You can make your extension behave this way too.

Update the imports at the top of your `index.ts` file so that the entire list of import statements looks like the following:

```
import {
  ILayoutRestorer, JupyterFrontEnd, JupyterFrontEndPlugin
} from '@jupyterlab/application';

import {
  ICommandPalette, MainAreaWidget, WidgetTracker
} from '@jupyterlab/apputils';

import {
  Message
} from '@phosphor/messaging';

import {
  Widget
} from '@phosphor/widgets';
```

Install this dependency:

```
jlpm add @phosphor/coreutils
```

Then add the `ILayoutRestorer` interface to the `JupyterFrontEndPlugin` definition. This addition passes the global `LayoutRestorer` as the third parameter of the `activate` function.

```
const extension: JupyterFrontEndPlugin<void> = {
  id: 'jupyterlab_apod',
  autoStart: true,
```

(continues on next page)

(continued from previous page)

```
  requires: [ICommandPalette, ILayoutRestorer],
  activate: activate
};
```

Finally, rewrite the `activate` function so that it:

1. Declares a widget variable, but does not create an instance immediately.
2. Constructs a `WidgetTracker` and tells the `ILayoutRestorer` to use it to save/restore panel state.
3. Creates, tracks, shows, and refreshes the widget panel appropriately.

```
function activate(app: JupyterFrontEnd, palette: ICommandPalette, restorer:␣
↪ILayoutRestorer) {
  console.log('JupyterLab extension jupyterlab_apod is activated!');

  // Declare a widget variable
  let widget: MainAreaWidget<APODWidget>;

  // Add an application command
  const command: string = 'apod:open';
  app.commands.addCommand(command, {
    label: 'Random Astronomy Picture',
    execute: () => {
      if (!widget) {
        // Create a new widget if one does not exist
        const content = new APODWidget();
        widget = new MainAreaWidget({content});
        widget.id = 'apod-jupyterlab';
        widget.title.label = 'Astronomy Picture';
        widget.title.closable = true;
      }
      if (!tracker.has(widget)) {
        // Track the state of the widget for later restoration
        tracker.add(widget);
      }
      if (!widget.isAttached) {
        // Attach the widget to the main work area if it's not there
        app.shell.add(widget, 'main');
      }
      widget.content.update();

      // Activate the widget
      app.shell.activateById(widget.id);
    }
  });

  // Add the command to the palette.
  palette.addItem({ command, category: 'Tutorial' });

  // Track and restore the widget state
  let tracker = new WidgetTracker<MainAreaWidget<APODWidget>>({
    namespace: 'apod'
  });
  restorer.restore(tracker, {
    command,
    name: () => 'apod'
  });
```

(continues on next page)

(continued from previous page)

```
}
```

Rebuild your extension one last time and refresh your browser tab. Execute the *Random Astronomy Picture* command and validate that the panel appears with an image in it. Refresh the browser tab again. You should see an Astronomy Picture panel reappear immediately without running the command. Close the panel and refresh the browser tab. You should then not see an Astronomy Picture tab after the refresh.

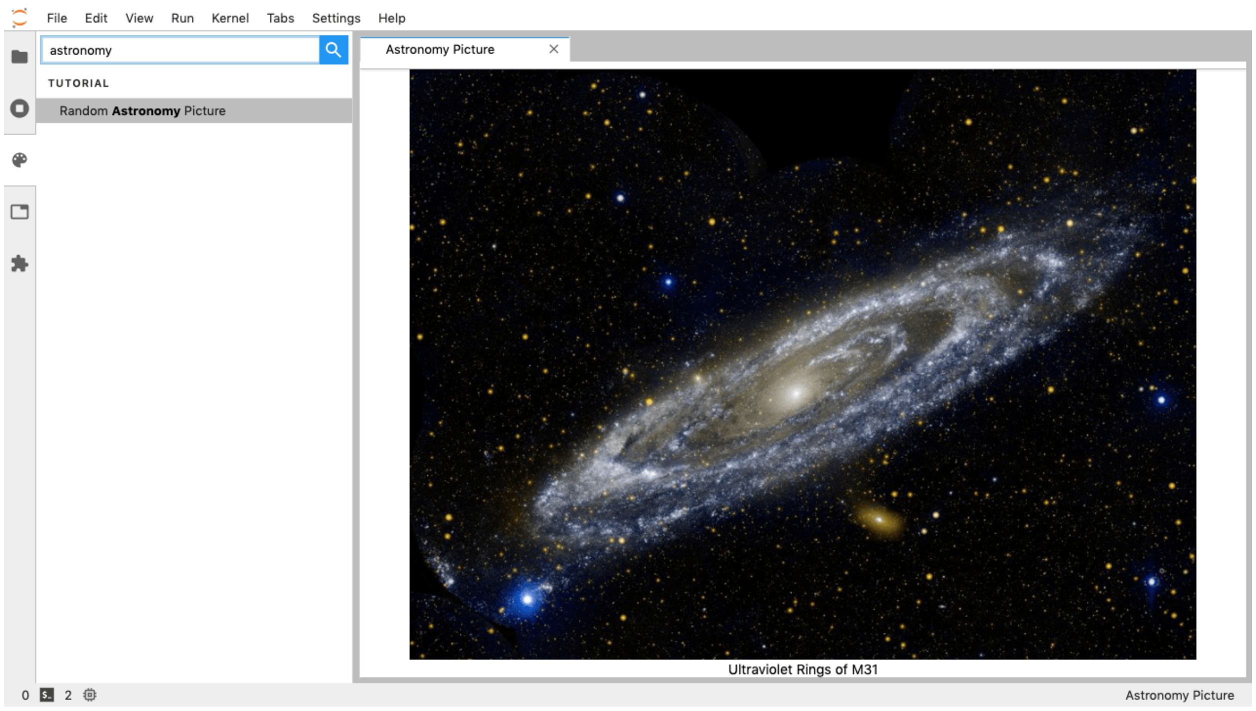


Fig. 4: The completed extension, showing the Astronomy Picture of the Day for 24 Jul 2015.

Refer to the 05-restore-panel-state tag if your extension is not working correctly. Make a commit when the state of your extension persists properly.

```
git add .
git commit -m 'Restore panel state'
```

Congratulations! You've implemented all of the behaviors laid out at the start of this tutorial. Now how about sharing it with the world?

33.6 Publish your extension to npmjs.org

npm is both a JavaScript package manager and the de facto registry for JavaScript software. You can sign up for an account on the npmjs.com site or create an account from the command line by running `npm adduser` and entering values when prompted. Create an account now if you do not already have one. If you already have an account, login by running `npm login` and answering the prompts.

Next, open the project `package.json` file in your text editor. Prefix the `name` field value with `@your-npm-username>/` so that the entire field reads `"name": "@your-npm-username/jupyterlab_apod"` where you've replaced the string `your-npm-username` with your real username. Review the homepage, repository, license, and other supported package.json fields while you have the file open. Then open the README.md file and adjust the command in the *Installation* section so that it includes the full, username-prefixed package name you just included in the package.json file. For example:

```
jupyter labextension install @your-npm-username/jupyterlab_apod
```

Return to your terminal window and make one more git commit:

```
git add .
git commit -m 'Prepare to publish package'
```

Now run the following command to publish your package:

```
npm publish --access=public
```

Check that your package appears on the npm website. You can either search for it from the homepage or visit https://www.npmjs.com/package/@your-username/jupyterlab_apod directly. If it doesn't appear, make sure you've updated the package name properly in the package.json and run the npm command correctly. Compare your work with the state of the reference project at the 06-prepare-to-publish tag for further debugging.

You can now try installing your extension as a user would. Open a new terminal and run the following commands, again substituting your npm username where appropriate (make sure to stop the existing jupyter lab --watch command first):

```
conda create -n jupyterlab-apod jupyterlab nodejs
conda activate jupyterlab-apod
jupyter labextension install @your-npm-username/jupyterlab_apod
jupyter lab
```

You should see a fresh JupyterLab browser tab appear. When it does, execute the *Random Astronomy Picture* command to prove that your extension works when installed from npm.

33.7 Learn more

You've completed the tutorial. Nicely done! If you want to keep learning, here are some suggestions about what to try next:

- Add the image description that comes in the API response to the panel.
- Assign a default hotkey to the *Random Astronomy Picture* command.
- Make the image a link to the picture on the NASA website (URLs are of the form https://apod.nasa.gov/apod/apYYMMDD.html).
- Make the image title and description update after the image loads so that the picture and description are always synced.
- Give users the ability to pin pictures in separate, permanent panels.
- Add a setting for the user to put in their API key so they can make many more requests per hour than the demo key allows.
- Push your extension git repository to GitHub.
- Learn how to write *other kinds of extensions*.

CHAPTER

THIRTYFOUR

INDICES AND TABLES

- genindex
- modindex
- search