JupyterLab Documentation

Release 1.0 Beta

Project Jupyter

Jan 25, 2018

Getting Started

CHAPTER 1

Overview

JupyterLab is the next generation user interface for Project Jupyter

JupyterLab goes beyond the classic Jupyter Notebook by providing a flexible and *extensible* web application with a set of reusable components. You can arrange multiple notebooks, text editors, terminals, output areas, and custom components using tabs/panels and collapsible sidebars. These *components* are carefully designed to enable you to use them together or on their own to support new workflows (for example, you can send code from a file to a code console with a keystroke, or move cells around a notebook or between notebooks with drag-and-drop).

JupyterLab has full support for Jupyter Notebook documents. In addition, it enables other models of interactive computing, such as:

- *Code Consoles* provide transient scratchpads for running code interactively, with full support for rich output.
- *Kernel-backed documents* allow code in any text file (Markdown, Python, R, LaTeX, etc.) to be run interactively in any Jupyter kernel.

JupyterLab also offers a unified model for viewing and handling data formats. This allows data in many formats (images, CSV, JSON, Markdown, PDF, Vega, Vega-Lite, etc.) to be opened as a file or returned by a kernel as rich output. See *File and Output Formats* for more information.

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

CHAPTER 2

Installation

JupyterLab can be installed using `conda` or `pip`:

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

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

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

```
pip install jupyterlab
```

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
```

2.1 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
```

2.2 Supported browsers

The following browsers are currently known to work:

- Firefox (latest version)
- Chrome (latest version)
- Safari (latest version)

Earlier browser versions or other browsers may also work, but may not be tested.

CHAPTER 3

Starting JupyterLab

Start JupyterLab using:

```
jupyter lab
```

JupyterLab will open automatically in your browser. You may also access JupyterLab by entering the notebook server's URL (usually `http://localhost:8888`) into the browser. You can also open the classic Notebook from JupyterLab by selecting "Launch Classic Notebook" from the JupyterLab Help menu.

Because JupyterLab is a server extension of the classic Jupyter Notebook server, you can also use JupyterLab by starting the classic Jupyter Noteboook (`jupyter notebook`) and visiting the `/lab` URL (usually `http://localhost:8888/lab`) rather than the default `/tree` URL. Conversely, to go to the classic Notebook from JupyterLab, you can change the URL from `/lab` to `/tree`.

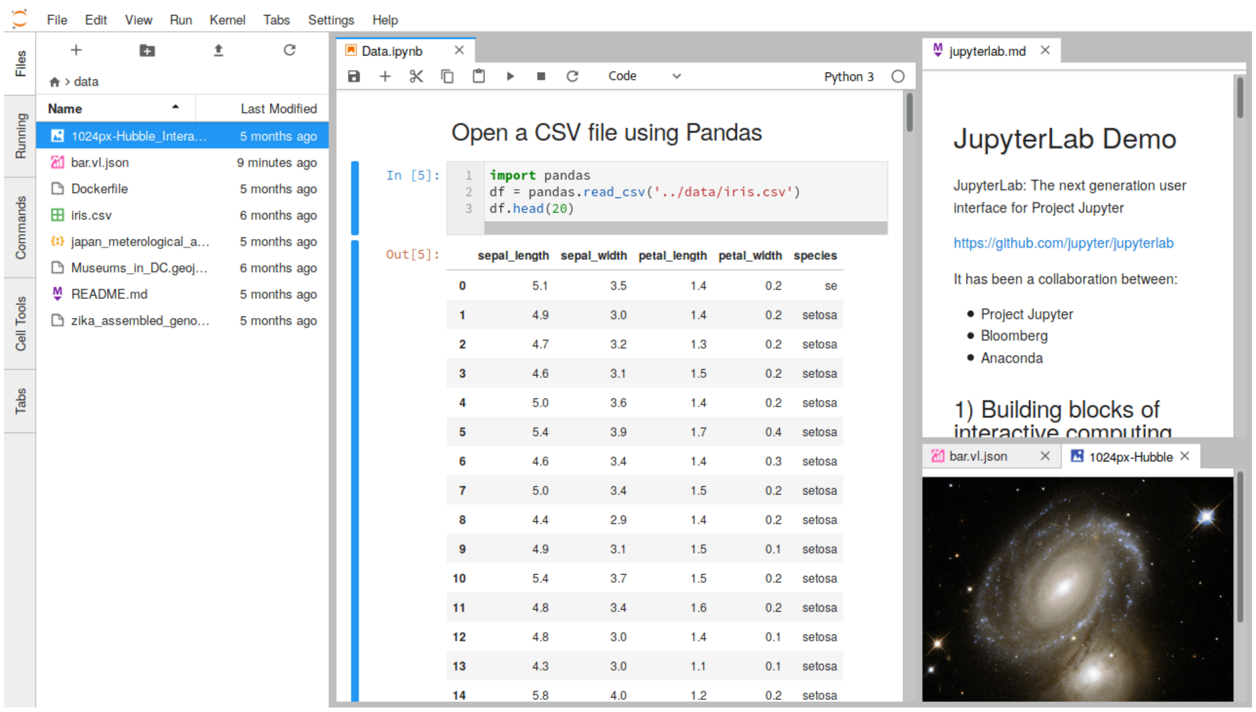
JupyterLab has the same security model as the classic Jupyter Notebook; for more information see the security section of the classic Notebook's documentation.

CHAPTER 4

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

4.1 Menu Bar

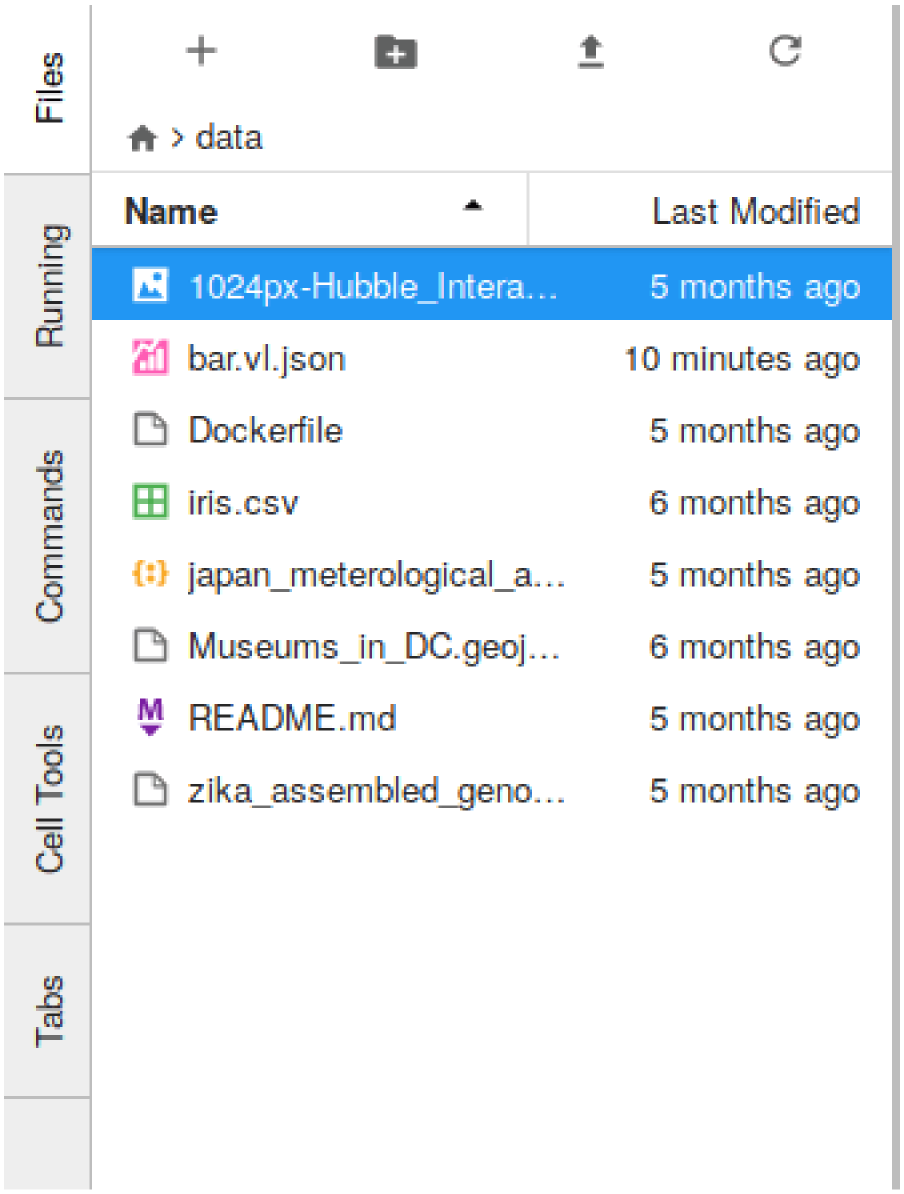
The menu bar at the top of JupyterLab has top-level menus that expose actions available in JupyterLab with their keyboard shortcuts. The default menus are:

- 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.

4.2 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 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 clicking on the active sidebar tab:

[animation]

JupyterLab extensions can add additional panels to the left sidebar.

4.3 Main area

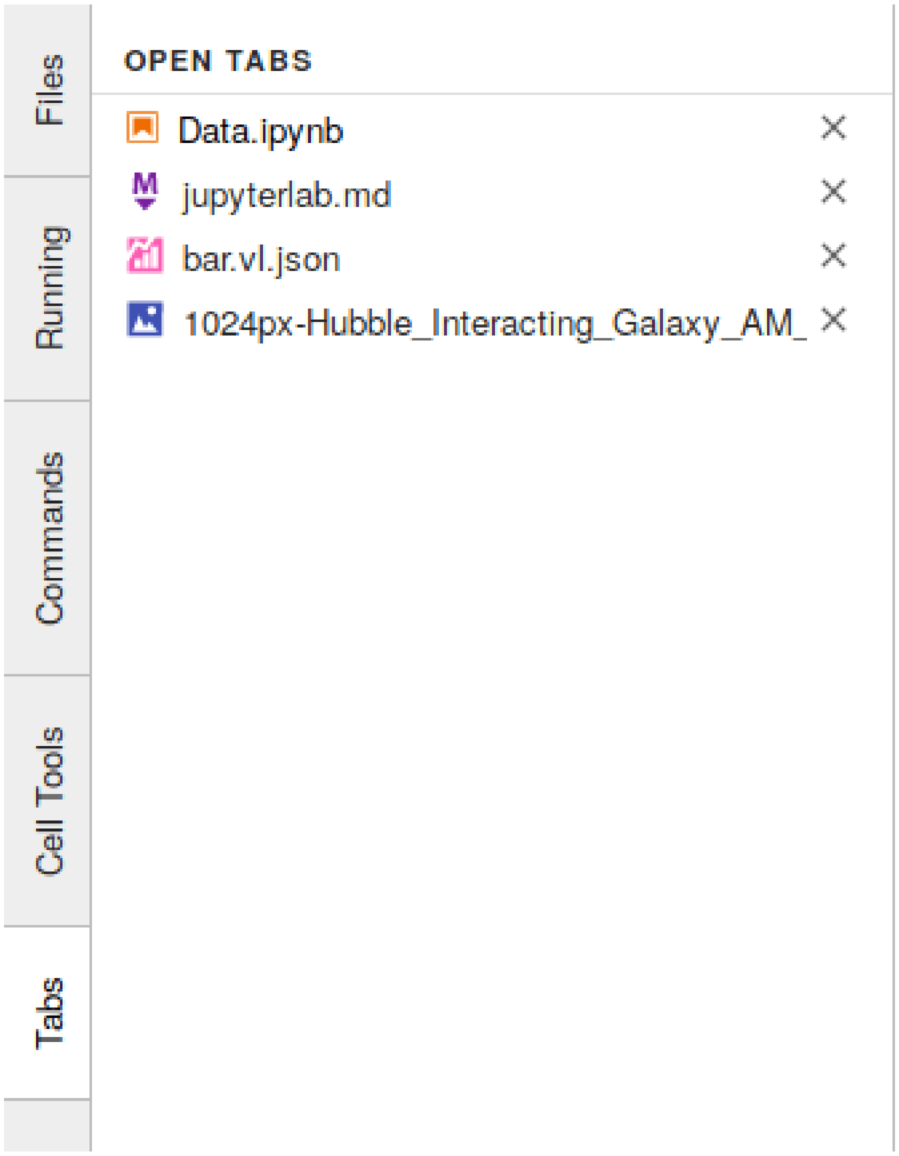
The main work area in JupyterLab allows you to arrange documents (notebooks, text files, etc.) and other activities (terminals, code consoles, etc.) into panels of tabs that can be resized or subdivided:

[animation]

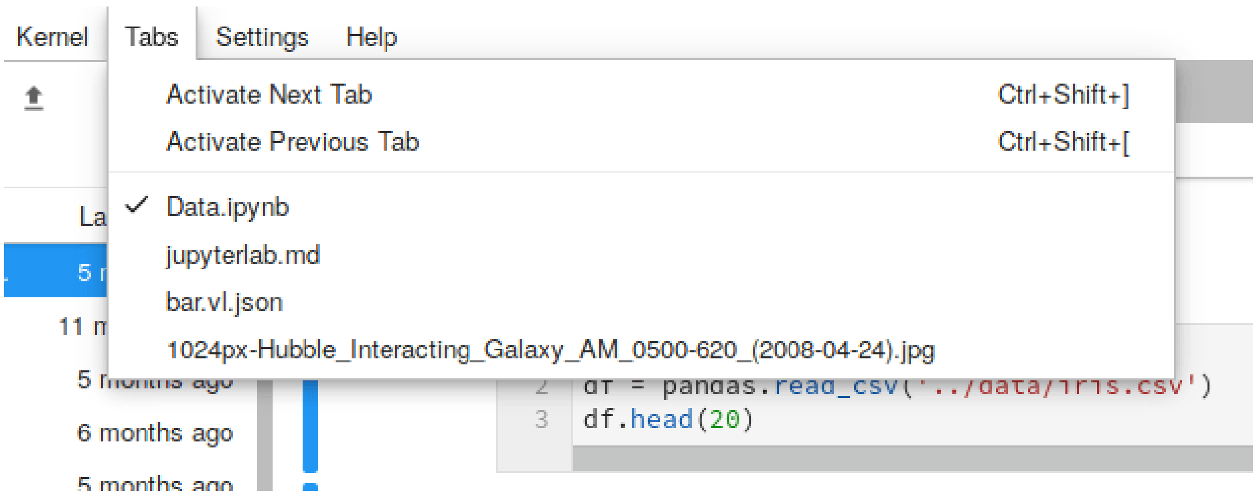
The main area has a single current activity. The tab for this activity is marked with a colored top border (blue by default).

4.4 Tabs and Single Document Mode

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

```
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 area. Single Document Mode toggles the view of the main area to show only a single tab at a time:

[animation]

When you leave Single Document Mode, the original layout of the main area is restored.

4.5 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:

[animation]

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

[animation]

CHAPTER 5

Working with Files

5.1 Opening Files

The file browser and File menu allow 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:

[screenshot]

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

[screenshot]

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

[animation]

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

[animation]

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 non-default viewer/editor, right-click on its name in the file browser and use the "Open With…" submenu to select the viewer/editor:

[animation]

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

[animation]

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:

[animation]

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.

[screenshot]

5.2 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 area, which allows you to pick an activity and kernel:

[animation]

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

[screenshot]

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):

[animation]

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:

[animation]

5.3 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:

[animation]

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

[animation]

CHAPTER 6

File Editor

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

[screenshot]

The file editor includes syntax highlighting, configurable indentation (tabs or spaces), different key maps (vim, emacs, Sublime Text) and basic theming. These settings can be found in the Settings menu:

[screenshot]

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

[animation]

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:

[animation]

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

[animation]

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:

[animation]

CHAPTER 7

Notebooks

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

[screenshot]

Jupyter Notebook (.ipynb files) are fully supported in JupyterLab. Furthermore, the notebook document format used in JupyterLab is the same as in the classic 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:

[animation]

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:

[animation]

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:

[animation]

Drag cells between notebooks to quickly copy content:

[animation]

Create multiple synchronized views of a single notebook:

[animation]

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

[animation]

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

[animation]

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

[animation]

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

[animation]

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

[animation]

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 "Create Console for Notebook":

[animation]

CHAPTER 8

Code Consoles

Code consoles allow 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:

[animation]

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

[animation]

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

[animation]

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

[animation]

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:

[animation]

CHAPTER 9

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.

[screenshot]

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

[animation]

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:

[animation]

CHAPTER 10

Managing Kernels and Terminals

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

[screenshot]

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 allows you to perform long-running actions and return later. The Running tab allows you to re-open or focus the document linked to a given kernel or terminal:

[animation]

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

[animation]

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

[animation]

CHAPTER 11

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 any JupyterLab command:

[screenshot]

The Command Palette can be accessed using the keyboard shortcut Command-Shift-C:

[animation]

CHAPTER 12

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 allows 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":

[animation]

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`:

[animation]

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:

[animation]

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 13

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:

[screenshot]

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:

```python
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:

[screenshot]

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

```python
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.

13.1 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:

[animation showing opening a markdown document editor and renderer side-by-side, and changes in the editor being reflected in the renderer]

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.

13.2 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 `0` to reset the zoom level. 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:

[animation]

13.3 CSV

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

Files with rows of comma-separated values (CSV files) are a common format for tabular data. The default viewer for CSV files in JupyterLab is a high-performance data grid viewer (which can also handle tab- and semicolon-separated values):

[animation]

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:

[animation]

13.4 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:

[animation showing both rendering JSON as cell output and viewing a JSON file]

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:

[animation]

13.5 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.

13.6 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.

13.7 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:

[animation]

13.8 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 allow 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:

[animation]

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

[animation]

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

[animation]

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).

[screenshot]

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

13.9 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:

[animation]

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

[screenshot or animation]

CHAPTER 14

Extensions

JupyterLab extensions add functionality to the JupyterLab application. They can provide new file viewer types, launcher activities, and output renderers, among many other things. JupyterLab extensions are npm packages (the standard package format in Javascript development). For information about developing extensions, see the *developer documentation*.

In order to install JupyterLab extensions, you need to have Node.js version 4+ 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
```

14.1 Installing Extensions

The base JupyterLab application includes a core set of extensions, which provide the features described in this User Guide (Notebook, Terminal, Text Editor, etc.) You can install new extensions into the application using the command:

```
jupyter labextension install <foo>
```

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

```
jupyter labextension install <foo>@1.2.3
```

You can also install an extension that is not uploaded to npm, i.e., `<foo>` 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-extensions` 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 <bar>
```

where `<bar>` is the name of the extension, as printed in the extension list. You can also uninstall core extensions using this command (which can later be re-installed).

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
```

14.2 Disabling Extensions

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

```
jupyter labextension disable <bar>
```

where `<bar>` is the name of the 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 <foo>
```

14.3 Advanced Usage

The JupyterLab application directory (where the application assets are built and the settings reside) 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`.

14.3.1 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 assets to the `static` directory

14.3.2 JupyterLab Application Directory

The JupyterLab application directory contains the subdirectories `extensions`, `schemas`, `settings`, `staging`, `static`, and `themes`.

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 contains `page_config.json` and `build_config.json` files.

page_config.json

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

Two important fields in the `page_config.json` file allow control of which plugins load:

1. `disabledExtensions` for extensions that should not load at all.
2. `deferredExtensions` for extensions that do not load until they are required by something, irrespective of whether they set `autostart` to `true`.

The value for each field is 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).

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 JuptyerLab 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.

CHAPTER 15

General Codebase Orientation

The `jupyterlab/jupyterlab` repository contains two packages:

- an npm package indicated by a `package.json` file in the repo's root directory
- a Python package indicated by a `setup.py` file in the repo's root directory

Th npm package and the Python package are both named `jupyterlab`.

See the Contributing Guidelines for developer installation instructions.

15.1 Directories

15.1.1 NPM package: `src/`, `lib/`, `typings/`, `buildutils/`

- `src/`: the source typescript files.
  - `jlpm run build` builds the source files into javascript files in `lib/`.
  - `jlpm run clean` deletes the `lib/` directory.
- `typings/`: type definitions for external libraries that typescript needs.
- `buildutils/`: Utilities for managing the repo

    75b8171af. . . Fix clean:slate script and update docs

15.1.2 Examples: `examples/`

The `examples/` directory contains stand-alone examples of components, such as a simple notebook on a page, a console, terminal, and a filebrowser. The `lab` example illustrates a simplified combination of components used in JupyterLab. This example shows multiple stand-alone components combined to create a more complex application.

15.1.3 Testing: `test/`

The tests are stored and run in the `test/` directory. The source files are in `test/src/`.

15.1.4 Notebook extension: `jupyterlab/`

The `jupyterlab/` directory contains the Jupyter server extension.

The server extension includes a private npm package in order to build the **webpack bundle** which the extension serves. The private npm package depends on the `jupyterlab` npm package found in the repo's root directory.

15.1.5 Git hooks: `git-hooks/`

The `git-hooks/` directory stores some convenience git hooks that automatically rebuild the npm package and server extension every time you check out or merge (via pull request or direct push to master) in the git repo.

15.1.6 Documentation: `docs/`

After building the docs (`jlpm run docs`), `docs/index.html` is the entry point to the documentation.

CHAPTER 16

Extension Developer Guide

JupyterLab can be extended in three ways via:

- **application plugins (top level):** Application plugins extend the functionality of JupyterLab itself.
- **mime renderer extension (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.
- document widget extensions (lower level): Document widget extensions extend the functionality of document widgets added to the application, and we cover them in the "Documents" tutorial.

See *Let's Make an xkcd JupyterLab Extension* to learn how to make a simple JupyterLab extension.

A JupyterLab application is comprised of:

- A core Application object
- Plugins

16.1 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 cookie cutters to create JuptyerLab plugin extensions in *CommonJS <https://github.com/jupyterlab/extension-cookiecutter-js>* and *TypeScript <https://github.com/jupyterlab/extension-cookiecutter-ts>*.

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.

A dependency graph for the core JupyterLab plugins (along with links to their source) is shown here:

16.2 Application Object

The JupyterLab Application object is given to each plugin in its `activate()` function. The Application object has a:

- commands - used to add and execute commands in the application.
- keymap - used to add keyboard shortcuts to the application.
- shell - a JupyterLab shell instance.

16.3 JupyterLab Shell

The JupyterLab shell is used to add and interact with content in the application. The application consists of:

- A top area for things like top level menus and toolbars
- Left and right side bar areas for collapsible content
- A main area for user activity.
- A bottom area for things like status bars

16.4 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 allows information like resize events to flow through the widget hierarchy in the application. **Phosphor signals** are a *one-to-many* interaction that allow listeners to react to changes in an observed object.

16.5 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"`).

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.

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. 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 filesystem, 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, but wil **not** remove any files added by the `addsibling` script, which should be removed manually.

The package should export EMCAScript 5 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. If your JavaScript is written in any other dialect than EMCAScript 5 it must be converted using an appropriate tool.

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`).

16.6 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 cookiecutters for Mime render extensions in JavaScript and TypeScript.

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 `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.

16.7 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 CSS file in the theme directory. 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 assets is specified `package.json` under the `"jupyterlab"` key as `"themeDir"`. See the JupyterLab Light Theme for an example. Ensure that the theme files are included in the `"files"` metadata in package.json. A theme can optionally specify an `embed.css` file that can be consumed outside of a JupyterLab application.

To quickly create a theme based on the JupyterLab Light Theme, follow the instructions in the contributing guide and then run `jlpm run create:theme` from the repository root directory. Once you select a name, title and a description, a new theme folder will be created in the current directory. You can move that new folder to a location of your choice, and start making desired changes.

The theme extension is installed the same as a regular extension (see [extension authoring](#Extension Authoring)).

16.8 Standard (General-Purpose) Extensions

See the example, How to Extend the Notebook Plugin. Notice that the mime renderer and themes extensions above use a limited, simplified interface to JupyterLab's extension system. Modifying the notebook plugin requires the full, general-purpose interface to the extension system.

16.9 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.

16.10 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 allows for default setting values and user overrides.

16.10.1 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: JupyterLabPlugin<IFoo> = {
  id,
  requires: [IStateDB],
  provides: IFoo,
  activate: (app: JupyterLab, 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
};
```

16.10.2 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 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();
```

CHAPTER 17

Documents

JupyterLab can be extended in two ways via:

- Extensions (top level): Application extensions extend the functionality of JupyterLab itself, and we cover them in the Extensions developer tutorial.
- **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).

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 is only meant to be accessed by the File Browser itself.

17.1 Document Registry

Document widget extensions in the JupyterLab application can register:

- widget factories
- model factories
- widget extension factories
- file types
- file creators

17.1.1 Widget Factories

Create a widget for a given file.

Example

- The notebook widget factory that creates NotebookPanel widgets.

17.1.2 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).

17.1.3 Widget Extension Factories

Adds additional functionality to a widget type. An extension instance is created for each widget instance, allowing 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.

17.1.4 File Types

Intended to be used in a "Create New" dialog, providing a list of known file types.

17.1.5 File Creators

Intended for create quick launch file creators.

The default use will be for the "create new" dropdown in the file browser, giving list of items that can be created with default options (e.g. "Python 3 Notebook").

17.1.6 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.

17.1.7 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.

17.2 Document Manager

The *Document Manager* handles:

- document models
- document contexts

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

CHAPTER 18

Notebook

18.1 Background

JupyterLab Walkthrough June 16, 2016 YouTube video

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.

18.2 Structure of the Notebook plugin

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

18.2.1 Model

The **‘NotebookModel <http://jupyterlab.github.io/jupyterlab/classes/_notebook_src_model_.notebookmodel.html>‘__** contains an observable list of cells.

A **‘cell model <http://jupyterlab.github.io/jupyterlab/modules/_cells_src_model_.html>‘__** 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.)

Cursors and metadata

The notebook model and the cell model (i.e. notebook cells) support getting and setting metadata through cursors. You may request a cursor to write to a specific metadata key from a notebook model or a cell model.

18.2.2 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.

18.3 How to extend the Notebook plugin

We'll walk through two notebook extensions:

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

18.3.1 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 {
  JupyterLab, JupyterLabPlugin
} 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: JupyterLabPlugin<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 {
    let callback = () => {
      NotebookActions.runAll(panel.notebook, context.session);
    };
    let button = new ToolbarButton({
      className: 'myButton',
      onClick: callback,
      tooltip: 'Run All'
    });

    let i = document.createElement('i');
    i.classList.add('fa', 'fa-fast-forward');
    button.node.appendChild(i);

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

/**
 * Activate the extension.
 */
function activate(app: JupyterLab) {
  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.

18.3.2 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] (http://jupyterlab.github.io/jupyterlab/interfaces/_docregistry_src_registry_.documentregistry.icontext.html). 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 19

Design Patterns

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

19.1 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.

19.2 Initialization Options

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

19.3 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.

19.4 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.

19.5 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 allows the entire section to be collapsed in an editor if desired.

19.6 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.

19.7 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.

19.8 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 allow 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 allows the connection to be properly cleared by `clearSignalData(this)`. Using a private method avoids allocating a closure for each connection.

19.9 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 allow consumers to handle a change in model. The reason to allow 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.

19.10 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.

19.11 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.

19.12 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.

19.13 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.

19.14 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 20

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 dfined in a main theme package.

20.1 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 `src/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 where ever possible. Individual packages should not npm-depend on these packages though, to allow the theme to be swapped out.
- Additional public/private CSS variables are defined by plugins sparingly and in accordance with the conventions described below.

20.2 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.

20.2.1 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`.

20.2.2 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`.

20.2.3 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.

20.3 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.
- Multiple CSS files may be used and organized as needed, but they should be imported into a single `index.css` at the top-level of the plugin.

20.4 CSS class names

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 allows 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

20.5 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 stucture 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.

CHAPTER 21

Writing Documentation

This section provide information about writing documentation for JupyterLab.

21.1 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.

21.2 User Interface Naming Conventions

21.2.1 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.

21.2.2 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 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

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

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.

21.3 Keyboard Shortcuts

Typeset keyboard shortcuts as follows:

- Monospace typeface, with no 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.

21.4 Screenshots and Animations

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

- Take screenshots and animations at the standard browser font sizes, 100% browser zoom.
- It is often helpful to have a colored background to highlight the content of an animation or screenshot. If a colored background is needed, use Material Design Grey 500 (`#9e9e9e`).
- Screenshots and animations should be styled in the documentation with a `max-width: 100%` property. Never stretch them wider than the original.
- Screenshots and animations taken on high resolution screens (retina) may need to be saved at half resolution to be consistent.
- Screenshots or animations should be proceeded by a sentence describing the content, such as "To open a file, double-click on its name in the File Browser:".
- Ideally each screenshot or animation should have a drop shadow, but it is unclear if we should do that beforehand or in CSS (the xkcd extension tutorial has images with shadowns already - let's think through this.)

  To help us organize them, let's name the files like this:

  ```
  sourcefile.md
  sourcefile_filebrowser.png
  sourcefile_editmenu.png
  ```

  This will help us track the images next to the content they are used in.

CHAPTER 22

Virtual DOM and React

JupyterLab is based on PhosphorJS, which provides a flexible `Widget` class that handles the following:

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

In situations where these feature are needed, we recommend using Phosphor's `Widget` class directly.

The idea of virtual DOM rendering, which became popular in the React community, is a very elegant and efficient way of rendering and updating DOM content in response to model/state changes.

Phosphor's `Widget` class integrates well with ReactJS and we are now using React in JupyterLab to render leaf content when the above capabilities are not needed.

An example of using React with Phosphor can be found in the launcher of JupyterLab.

CHAPTER 23

Adding Content

As an example: Add a leaflet viewer plugin for geoJSON files.

- Go to npm: search for leaflet (success!).
- Go to `jupyterlab` top level source directory: `jlpm add leaflet`. This adds the file to the `dependencies` in `package.json`.
- Next we see if there is a typing declaration for leaflet: `jlpm add --dev @types/leaflet`. Success!
- If there are no typings, we must create our own. An example typings file that exports functions is ansi_up. An example with a class is xterm.
- Add a reference to the new library in `src/typings.d.ts`.
- Create a folder in `src` for the plugin.
- Add `index.ts` and `plugin.ts` files.
- If creating CSS, import them in `src/default-themes/index.css`.
- The `index.ts` file should have the core logic for the plugin. In this case, it should create a widget and widget factory for rendering geojson files (see Documents).
- The `plugin.ts` file should create the extension and add the content to the application. In this case registering the widget factory using the document registry.
- Run `jlpm run build` from `jupyterlab/jupyterlab`
- Run `jupyter lab` and verify changes.

CHAPTER 24

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:

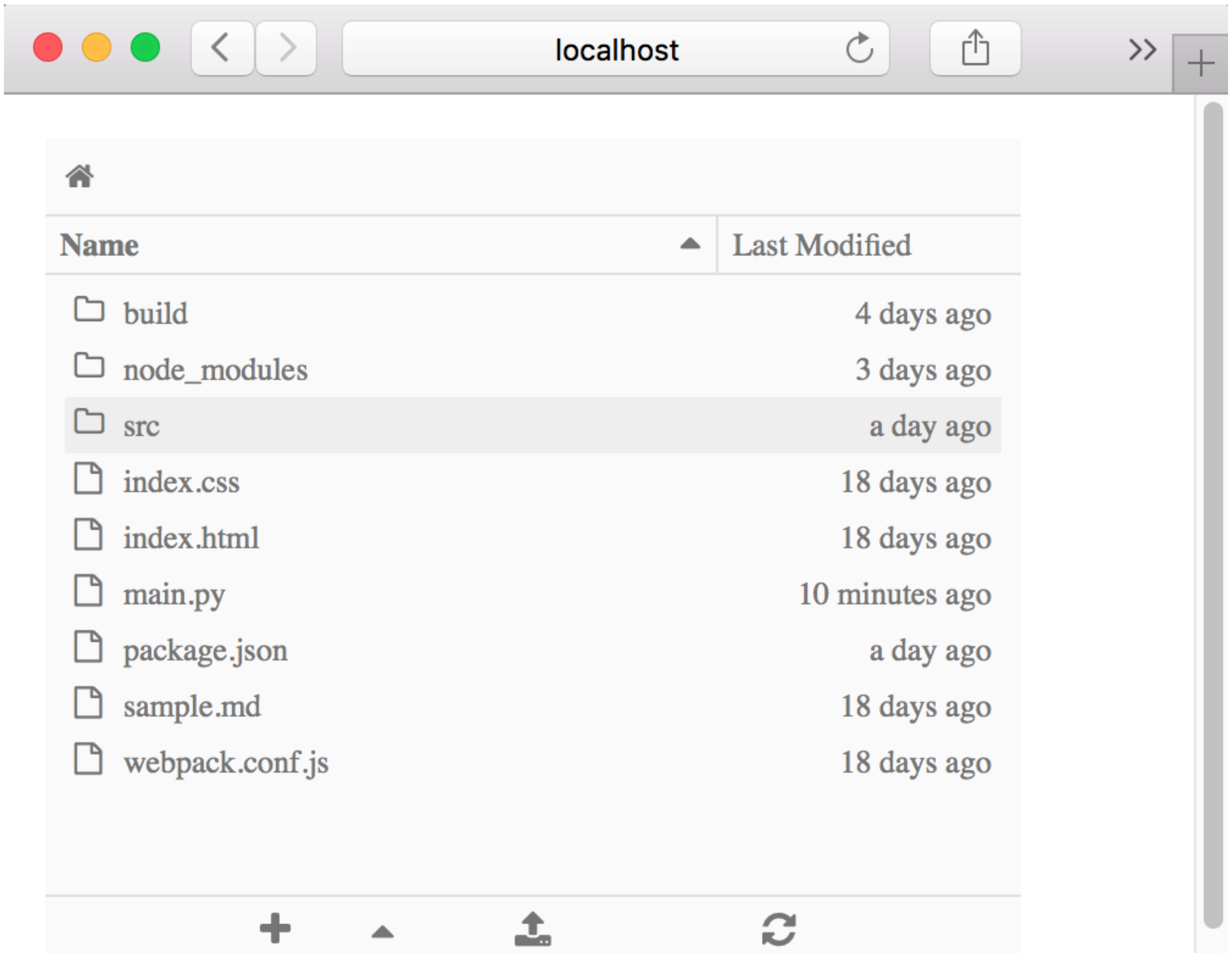
```
jlpm run build:examples
```

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

```
python main.py
```

24.1 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:

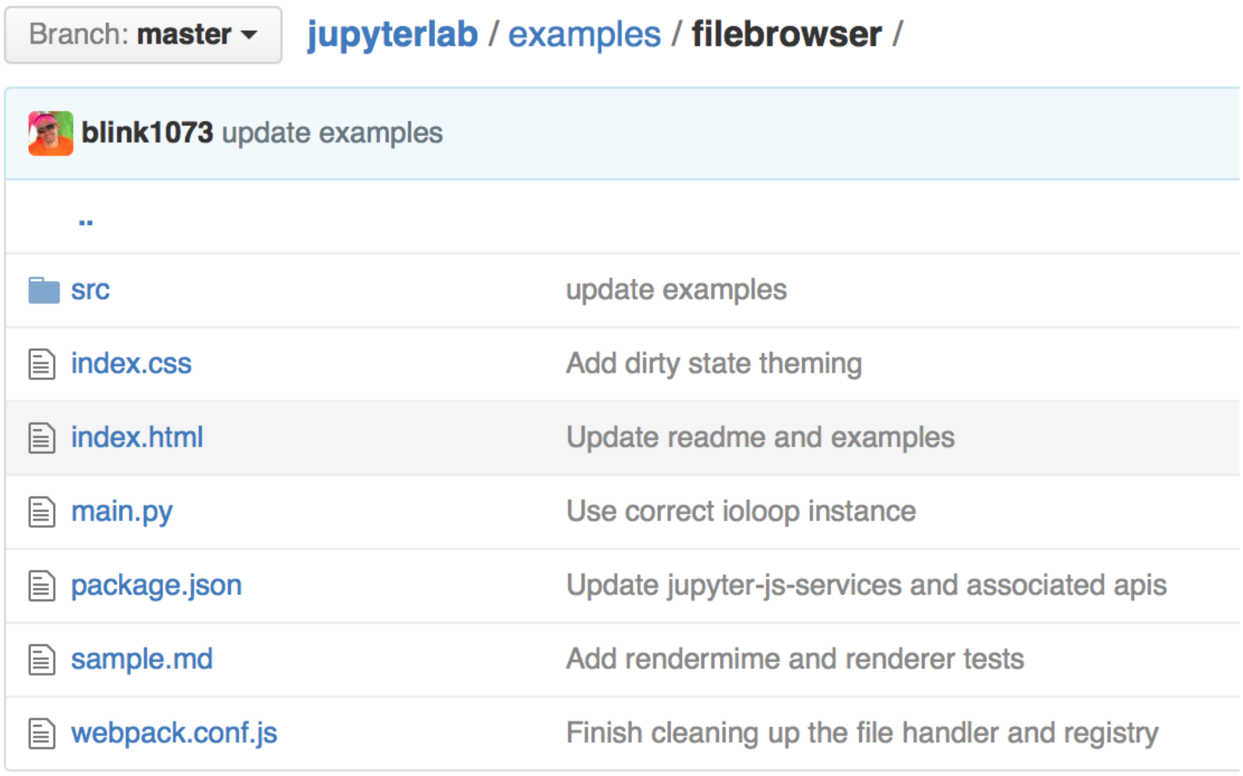


Name	Last Modified
build	4 days ago
node_modules	3 days ago
src	a day ago
index.css	18 days ago
index.html	18 days ago
main.py	10 minutes ago
package.json	a day ago
sample.md	18 days ago
webpack.conf.js	18 days ago

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

24.1.1 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 25

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.

25.1 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.
- 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 26

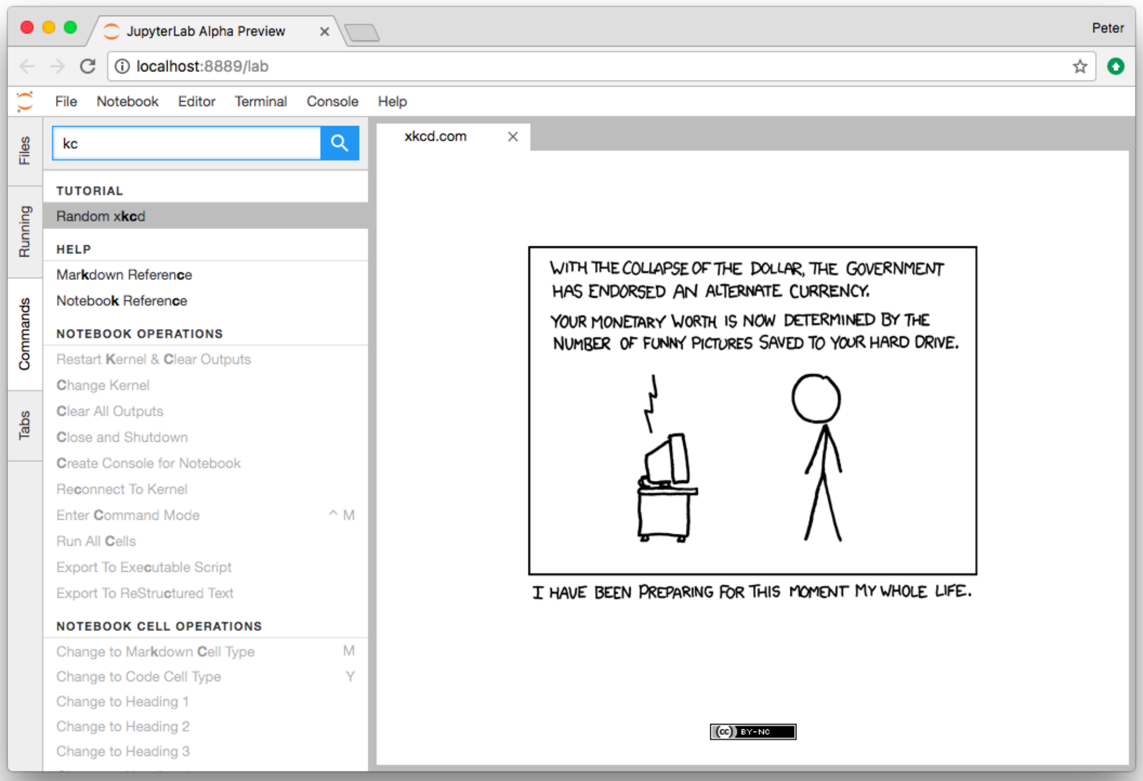
Let's Make an xkcd JupyterLab Extension

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 xkcd comic" command to the *command palette* sidebar
- Fetches the comic image and metadata when activated
- Shows the image and metadata in a tab panel

By working through this tutorial, you'll learn:

- How to setup an extension development environment from scratch on a Linux or OSX machine.
  - Windows users: You'll need to modify the commands slightly.
- 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


JupyterLab Alpha Preview
Peter
localhost:8889/lab
File
Notebook
Editor
Terminal
Console
Help
Files
Running
Commands
Tabs
kc
TUTORIAL
Random xkcd
HELP
Markdown Reference
Notebook Reference
NOTEBOOK OPERATIONS
Restart Kernel & Clear Outputs
Change Kernel
Clear All Outputs
Close and Shutdown
Create Console for Notebook
Reconnect To Kernel
Enter Command Mode ^ M
Run All Cells
Export To Executable Script
Export To ReStructured Text
NOTEBOOK CELL OPERATIONS
Change to Markdown Cell Type M
Change to Code Cell Type Y
Change to Heading 1
Change to Heading 2
Change to Heading 3
xkcd.com
WITH THE COLLAPSE OF THE DOLLAR, THE GOVERNMENT HAS ENDORSED AN ALTERNATE CURRENCY.
YOUR MONETARY WORTH IS NOW DETERMINED BY THE NUMBER OF FUNNY PICTURES SAVED TO YOUR HARD DRIVE.
I HAVE BEEN PREPARING FOR THIS MOMENT MY WHOLE LIFE.
BY-NC

Sound like fun? Excellent. Here we go!

26.1 Setup a development environment

26.1.1 Install conda using miniconda

Start by opening your web browser and downloading the latest Python 3.x Miniconda installer to your home directory. When the download completes, open a terminal and create a root conda environment by running this command.

```
bash Miniconda3*.sh -b -p ~/miniconda
```

Now activate the conda environment you just created so that you can run the `conda` package manager.

```
source ~/miniconda/bin/activate
```

26.1.2 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
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

It's best practice to leave the root conda environment, the one 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 nodejs jupyterlab cookiecutter git -c conda-forge
```

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

```
source ~/miniconda/bin/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.

26.2 Create a repository

Create a new repository for your extension. For example, on Github. This is an optional step but highly recommended if you want to share your extension.

26.3 Create an extension project

26.3.1 Initialize the project from a cookiecutter

Next use cookiecutter to create a new project for your extension.

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

When prompted, enter values like the following for all of the cookiecutter prompts.

```
author_name []: Your Name
extension_name [jupyterlab_myextension]: jupyterlab_xkcd
project_short_description [A JupyterLab extension.]: Show a random xkcd.com comic in␣
↪a JupyterLab panel
repository [https://github.com/my_name/jupyterlab_myextension]: Your repository
url
```

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

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

```
cd jupyterlab_xkcd
ls
```

You should see a list like the following.

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

26.3.2 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.

```
npm install
npm run build
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.

```
source ~/miniconda/bin/activate jupyterlab-ext
jupyter lab --watch
```

26.3.3 See the initial extension in action

JupyterLab should appear momentarily in your default web browser. If all goes well, the last bunch of messages you should see in your terminal should look something like the following:

```
Webpack is watching the files...

Hash: 1c15fc765a97c45c075c
Version: webpack 2.7.0
Time: 6423ms
                                 Asset     Size  Chunks                    Chunk Names
  674f50d287a8c48dc19ba404d20fe713.eot   166 kB          [emitted]
af7ae505a9eed503f8b8e6982036873e.woff2  77.2 kB          [emitted]
 fee66e712a8a08eef5805a46892932ad.woff    98 kB          [emitted]
  b06871f281fee6b241d60582ae9369b9.ttf   166 kB          [emitted]
  912ec66d7572ff821749319396470bde.svg   444 kB          [emitted]  [big]
                           0.bundle.js   890 kB       0  [emitted]  [big]
                        main.bundle.js  6.82 MB       1  [emitted]  [big]  main
                       0.bundle.js.map  1.08 MB       0  [emitted]
                    main.bundle.js.map  8.19 MB       1  [emitted]         main
  [27] ./~/@jupyterlab/application/lib/index.js 5.66 kB {1} [built]
 [427] ./~/@jupyterlab/application-extension/lib/index.js 6.14 kB {1} [optional]␣
↪[built]
 [443] ./~/@jupyterlab/pdf-extension/lib/index.js 4.98 kB {1} [optional] [built]
 [445] ./~/@jupyterlab/settingeditor-extension/lib/index.js 2.67 kB {1} [optional]␣
↪[built]
 [446] ./~/@jupyterlab/shortcuts-extension/lib/index.js 3.75 kB {1} [optional] [built]
 [447] ./~/@jupyterlab/tabmanager-extension/lib/index.js 1.8 kB {1} [optional] [built]
 [448] ./~/@jupyterlab/terminal-extension/lib/index.js 7.33 kB {1} [optional] [built]
 [449] ./~/@jupyterlab/theme-dark-extension/lib/index.js 800 bytes {1} [optional]␣
↪[built]
 [450] ./~/@jupyterlab/theme-light-extension/lib/index.js 804 bytes {1} [optional]␣
↪[built]
 [451] ./~/@jupyterlab/tooltip-extension/lib/index.js 5.61 kB {1} [optional] [built]
 [452] ./~/@jupyterlab/vega2-extension/lib/index.js 6.19 kB {1} [optional] [built]
 [453] ./~/es6-promise/auto.js 179 bytes {1} [built]
 [454] /Users/foo/workspace/xkcd/lib/index.js 353 bytes {1} [optional] [built]
 [455] ./~/font-awesome/css/font-awesome.min.css 892 bytes {1} [built]
 [860] ./build/index.out.js 35.2 kB {1} [built]
    + 1114 hidden modules
```

Return to the browser. Open the JavaScript console in the JupyterLab tab by following the instructions for your browser:

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

You should see a message that says `JupyterLab extension jupyterlab_xkcd is activated!` in the console. If you do, congrats, you're ready to start modifying the 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.

26.3.4 Commit what you have to git

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

```
git init
git add .
git commit -m 'Seed xkcd 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_xkcd` on GitHub at https://github.com/jupyterlab/jupyterlab_xkcd.

26.4 Add an xkcd widget

26.4.1 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 xkcd comic* command to the palette and get it to show an *xkcd* tab panel when invoked.

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

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

Locate the `extension` object of type `JupyterLabPlugin`. Change the definition so that it reads like so:

```
/**
 * Initialization data for the jupyterlab_xkcd extension.
 */
const extension: JupyterLabPlugin<void> = {
  id: 'jupyterlab_xkcd',
  autoStart: true,
  requires: [ICommandPalette],
  activate: (app: JupyterLab, palette: ICommandPalette) => {
    console.log('JupyterLab extension jupyterlab_xkcd 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.

Run the following to rebuild your extension.

```
npm run build
```

When the build completes, 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_xkcd is activated!
ICommandPalette: Palette {_palette: CommandPalette}
```

Note that we had to run `npm 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 `npm run build` after each change, you can open a third terminal, and run the `npm run watch` command from your extension directory, which will automatically compile the TypeScript files as they change.

Now return to your editor. Add the following additional import to the top of the file.

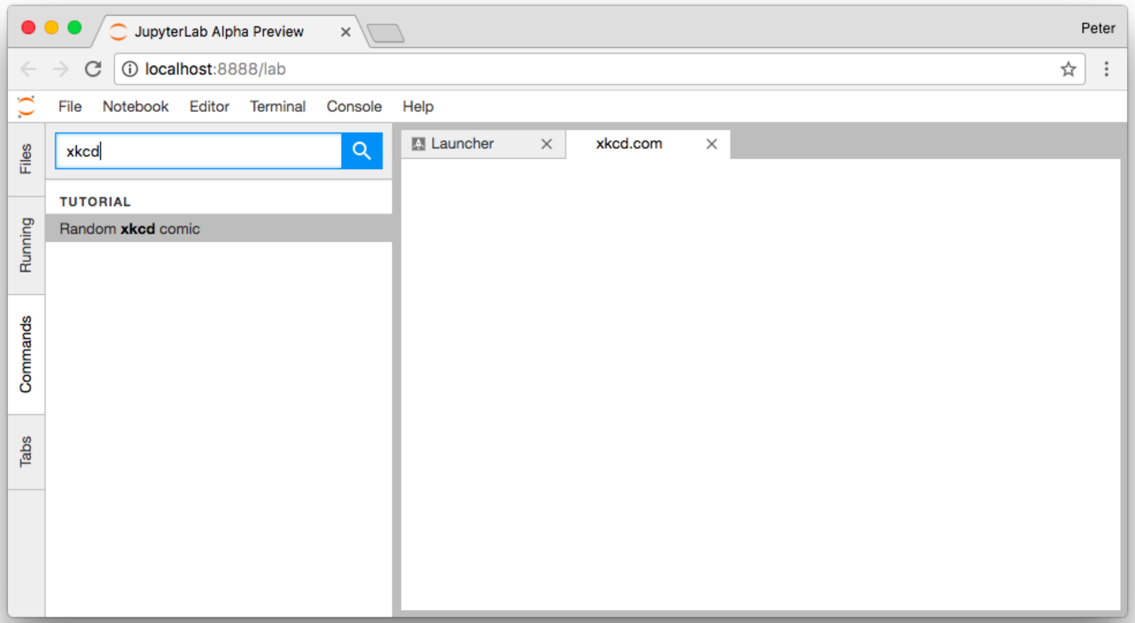
```
import {
  Widget
} from '@phosphor/widgets';
```

Then modify the `activate` function again so that it has the following code:

```
activate: (app: JupyterLab, palette: ICommandPalette) => {
  console.log('JupyterLab extension jupyterlab_xkcd is activated!');

  // Create a single widget
  let widget: Widget = new Widget();
  widget.id = 'xkcd-jupyterlab';
  widget.title.label = 'xkcd.com';
  widget.title.closable = true;

  // Add an application command
  const command: string = 'xkcd:open';
  app.commands.addCommand(command, {
    label: 'Random xkcd comic',
    execute: () => {
      if (!widget.isAttached) {
        // Attach the widget to the main area if it's not there
        app.shell.addToMainArea(widget);
      }
      // 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 `Widget` instance, assigns it 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 add a new command labeled *Random xkcd comic* to JupyterLab. When the comm and 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 adds the command to the command palette in a section called *Tutorial*.

Build your extension again using `npm run build` (unless you are using `npm run watch` already) and refresh the browser tab. Open the command palette on the left side and type *xkcd*. Your *Random xkcd comic* 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 *xkcd.com*. 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 *xkcd.com* panel is still open but no longer active. Now run the *Random xkcd comic* command one more time. The single *xkcd.com* 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 xkcd panel on command'
```

26.4.2 Show a comic in the panel

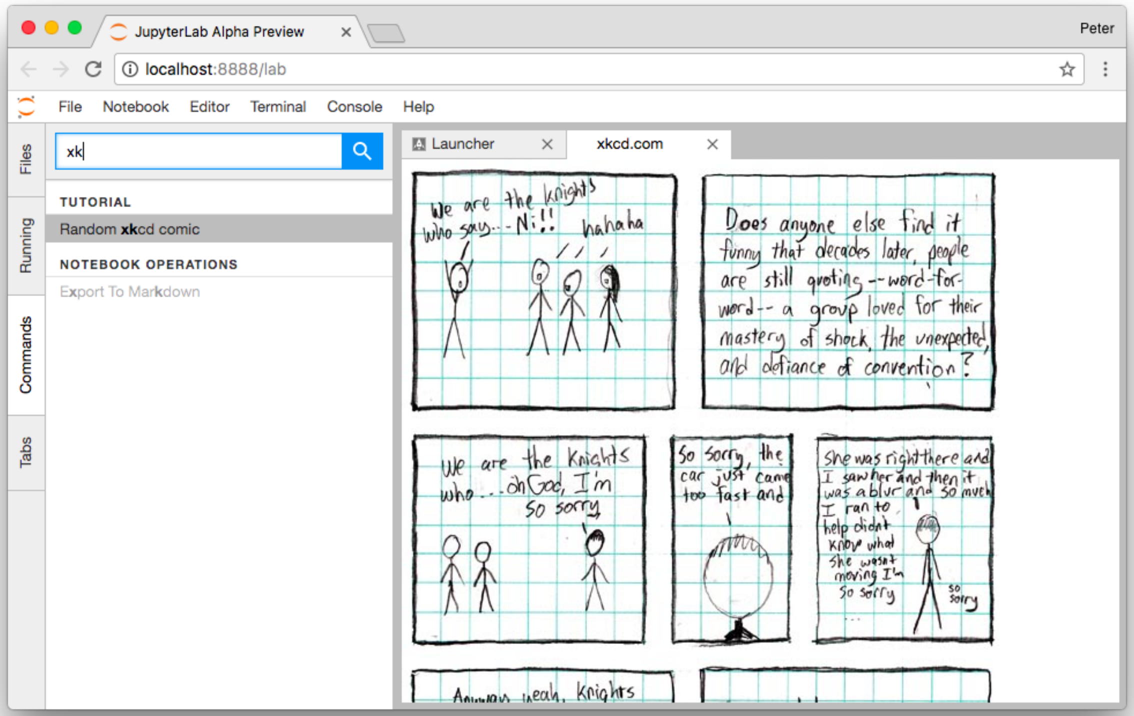
You've got an empty panel. It's time to add a comic to it. Go back to your code editor. Add the following code below the lines that create a `Widget` instance and above the lines that define the command.

```
// Add an image element to the panel
let img = document.createElement('img');
widget.node.appendChild(img);
```

```
// Fetch info about a random comic
fetch('https:////egszlpbmle.execute-api.us-east-1.amazonaws.com/prod').then(response␣
↪=> {
  return response.json();
}).then(data => {
  img.src = data.img;
  img.alt = data.title;
  img.title = data.alt;
});
```

The first two lines create a new HTML `<img>` element and add it to the widget DOM node. The next lines make a request using the HTML fetch API that returns information about a random xkcd comic, and set the image source, alternate text, and title attributes based on the response.

Rebuild your extension if necessary (`npm run build`), refresh your browser tab, and run the *Random xkcd comic* command again. You should now see a comic in the xkcd.com panel when it opens.


JupyterLab Alpha Preview
Peter
localhost:8888/lab
File
Notebook
Editor
Terminal
Console
Help
Files
Running
Commands
Tabs
xk
TUTORIAL
Random xkcd comic
NOTEBOOK OPERATIONS
Export To Markdown
Launcher
xkcd.com
We are the knights who say... Ni!!
hahaha
Does anyone else find it funny that decades later, people are still quoting --word-for-word-- a group loved for their mastery of shock, the unexpected, and defiance of convention?
We are the Knights who... oh God, I'm so sorry
So sorry, the car just came too fast and
She was right there and I saw her and then it was a blur and so much I ran to help didn't know what she wasn't moving I'm so sorry
so sorry
Anyway, yeah, knights

Note that the comic is not centered in the panel nor does the panel scroll if the comic is larger than the panel area. Also note that the comic 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 comic at all, compare your code with the 02-show-a-comic tag in the reference project. When it's working, make another git commit.

```
git add .
git commit -m 'Show a comic in the panel'
```

26.5 Improve the widget behavior

26.5.1 Center the comic and add attribution

Open `style/index.css` in our extension project directory for editing. Add the following lines to it.

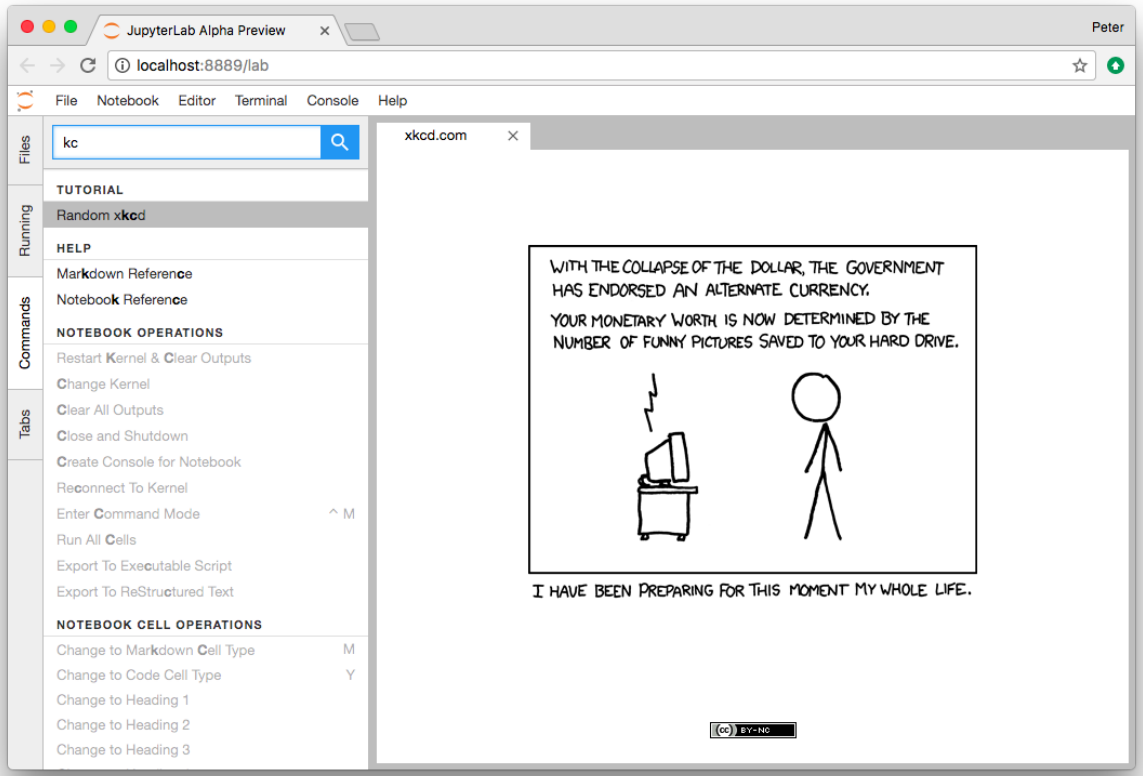
```
.jp-xkcdWidget {
    display: flex;
    flex-direction: column;
    overflow: auto;
}

.jp-xkcdCartoon {
    margin: auto;
}

.jp-xkcdAttribution {
    margin: 20px auto;
}
```

The first rule stacks content vertically within the widget panel and lets the panel scroll when the content overflows. The other rules center the cartoon and attribution badge horizontally and space them out vertically.

Return to the `index.ts` file. Note that there is already an import of the CSS file in the `index.ts` file. Modify the the `activate` function to apply the CSS classes and add the attribution badge markup. The beginning of the function should read like the following:

```
activate: (app: JupyterLab, palette: ICommandPalette) => {
  console.log('JupyterLab extension jupyterlab_xkcd is activated!');

  // Create a single widget
  let widget: Widget = new Widget();
  widget.id = 'xkcd-jupyterlab';
  widget.title.label = 'xkcd.com';
  widget.title.closable = true;
  widget.addClass('jp-xkcdWidget'); // new line

  // Add an image element to the panel
  let img = document.createElement('img');
  img.className = 'jp-xkcdCartoon'; // new line
  widget.node.appendChild(img);

  // New: add an attribution badge
  img.insertAdjacentHTML('afterend',
    `<div class="jp-xkcdAttribution">
      <a href="https://creativecommons.org/licenses/by-nc/2.5/" class="jp-
↪xkcdAttribution" target="_blank">
        <img src="https://licensebuttons.net/l/by-nc/2.5/80x15.png" />
      </a>
    </div>`
  );

  // Keep all the remaining fetch and command lines the same
  // as before from here down ...
```

Build your extension if necessary (`npm run build`) and refresh your JupyterLab browser tab. Invoke the *Random xkcd comic* command and confirm the comic is centered with an attribution badge below it. Resize the browser window or the panel so that the comic is larger than the available area. Make sure you can scroll the panel over the entire area of the comic.


JupyterLab Alpha Preview
Peter
localhost:8889/lab
File
Notebook
Editor
Terminal
Console
Help
Files
Running
Commands
Tabs
kc
TUTORIAL
Random xkcd
HELP
Markdown Reference
Notebook Reference
NOTEBOOK OPERATIONS
Restart Kernel & Clear Outputs
Change Kernel
Clear All Outputs
Close and Shutdown
Create Console for Notebook
Reconnect To Kernel
Enter Command Mode ^ M
Run All Cells
Export To Executable Script
Export To ReStructured Text
NOTEBOOK CELL OPERATIONS
Change to Markdown Cell Type M
Change to Code Cell Type Y
Change to Heading 1
Change to Heading 2
Change to Heading 3
xkcd.com
WITH THE COLLAPSE OF THE DOLLAR, THE GOVERNMENT HAS ENDORSED AN ALTERNATE CURRENCY.
YOUR MONETARY WORTH IS NOW DETERMINED BY THE NUMBER OF FUNNY PICTURES SAVED TO YOUR HARD DRIVE.
I HAVE BEEN PREPARING FOR THIS MOMENT MY WHOLE LIFE.

If anything is misbehaving, 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'
```

26.5.2 Show a new comic on demand

The `activate` function has grown quite long, and there's still more functionality to add. You should refactor the code into two separate parts:

1. An `XkcdWidget` that encapsulates the xkcd 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 comic should refresh

Start by refactoring the widget code into the new `XkcdWidget` class. Add the following additional import to the top of the file.

```
import {
  Message
} from '@phosphor/messaging';
```

Then add the class just below the import statements in the `index.ts` file.

```
/**
 * An xckd comic viewer.
 */
class XkcdWidget extends Widget {
  /**
   * Construct a new xkcd widget.
   */
  constructor() {
    super();

    this.id = 'xkcd-jupyterlab';
    this.title.label = 'xkcd.com';
    this.title.closable = true;
    this.addClass('jp-xkcdWidget');

    this.img = document.createElement('img');
    this.img.className = 'jp-xkcdCartoon';
    this.node.appendChild(this.img);

    this.img.insertAdjacentHTML('afterend',
      `<div class="jp-xkcdAttribution">
        <a href="https://creativecommons.org/licenses/by-nc/2.5/" class="jp-
↪xkcdAttribution" target="_blank">
          <img src="https://licensebuttons.net/l/by-nc/2.5/80x15.png" />
        </a>
      </div>`
    );
  }

  /**
   * The image element associated with the widget.
   */
  readonly img: HTMLImageElement;

  /**
   * Handle update requests for the widget.
   */
  onUpdateRequest(msg: Message): void {
    fetch('https://egszlpbmle.execute-api.us-east-1.amazonaws.com/prod').
↪then(response => {
      return response.json();
    }).then(data => {
      this.img.src = data.img;
      this.img.alt = data.title;
      this.img.title = data.alt;
    });
  }
};
```

You've written all of the code before. All you've done is restructure it to use instance variables and move the comic request to its own function.

Next move the remaining logic in `activate` to a new, top-level function just below the `XkcdWidget` class definition. Modify the code to create a widget when one does not exist in the main JupyterLab area or to refresh the comic in the exist widget when the command runs again. The code for the `activate` function should read as follows after these changes:

```
/**
 * Activate the xckd widget extension.
 */
function activate(app: JupyterLab, palette: ICommandPalette) {
  console.log('JupyterLab extension jupyterlab_xkcd is activated!');

  // Create a single widget
  let widget: XkcdWidget = new XkcdWidget();

  // Add an application command
  const command: string = 'xkcd:open';
  app.commands.addCommand(command, {
    label: 'Random xkcd comic',
    execute: () => {
      if (!widget.isAttached) {
        // Attach the widget to the main area if it's not there
        app.shell.addToMainArea(widget);
      }
      // Refresh the comic in the widget
      widget.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 `JupyterLabPlugin` object and refer instead to the top-level function like so:

```
const extension: JupyterLabPlugin<void> = {
  id: 'jupyterlab_xkcd',
  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 xkcd comic* command more than once without closing the panel. The comic should update each time you execute the command. Close the panel, run the command, and it should both reappear and show a new comic.

If anything is amiss, compare your code with the 04-refactor-and-refresh tag to debug. Once it's working properly, commit it.

```
git add .
git commit -m 'Refactor, refresh comic'
```

26.5.3 Restore panel state when the browser refreshes

You may notice that every time you refresh your browser tab, the xkcd 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 {
  JupyterLab, JupyterLabPlugin, ILayoutRestorer // new
} from '@jupyterlab/application';

import {
  ICommandPalette, InstanceTracker // new
} from '@jupyterlab/apputils';

import {
  JSONExt // new
} from '@phosphor/coreutils';

import {
  Message
} from '@phosphor/messaging';

import {
  Widget
} from '@phosphor/widgets';

import '../style/index.css';
```

Then, add the `ILayoutRestorer` interface to the `JupyterLabPlugin` definition. This addition passes the global `LayoutRestorer` to the third parameter of the `activate`.

```
const extension: JupyterLabPlugin<void> = {
  id: 'jupyterlab_xkcd',
  autoStart: true,
  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 an `InstanceTracker` 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: JupyterLab, palette: ICommandPalette, restorer:␣
↪ILayoutRestorer) {
  console.log('JupyterLab extension jupyterlab_xkcd is activated!');

  // Declare a widget variable
  let widget: XkcdWidget;

  // Add an application command
  const command: string = 'xkcd:open';
  app.commands.addCommand(command, {
    label: 'Random xkcd comic',
    execute: () => {
      if (!widget) {
        // Create a new widget if one does not exist
        widget = new XkcdWidget();
        widget.update();
      }
      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 area if it's not there
        app.shell.addToMainArea(widget);
      } else {
        // Refresh the comic in the widget
        widget.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 InstanceTracker<Widget>({ namespace: 'xkcd' });
  restorer.restore(tracker, {
    command,
    args: () => JSONExt.emptyObject,
    name: () => 'xkcd'
  });
};
```

Rebuild your extension one last time and refresh your browser tab. Execute the *Random xkcd comic* command and validate that the panel appears with a comic in it. Refresh the browser tab again. You should see an xkcd panel appear immediately without running the command. Close the panel and refresh the browser tab. You should not see an xkcd tab after the refresh.

Refer to the 05-restore-panel-state tag if your extension is misbehaving. Make a commit when the state of your extension persists properly.

```
git add .
git commit -m 'Restore panel state'
```

Congrats! You've implemented all of the behaviors laid out at the start of this tutorial. Now how about sharing it with the world?

26.6 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/xkcd-extension"` 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/xkcd-extension
```

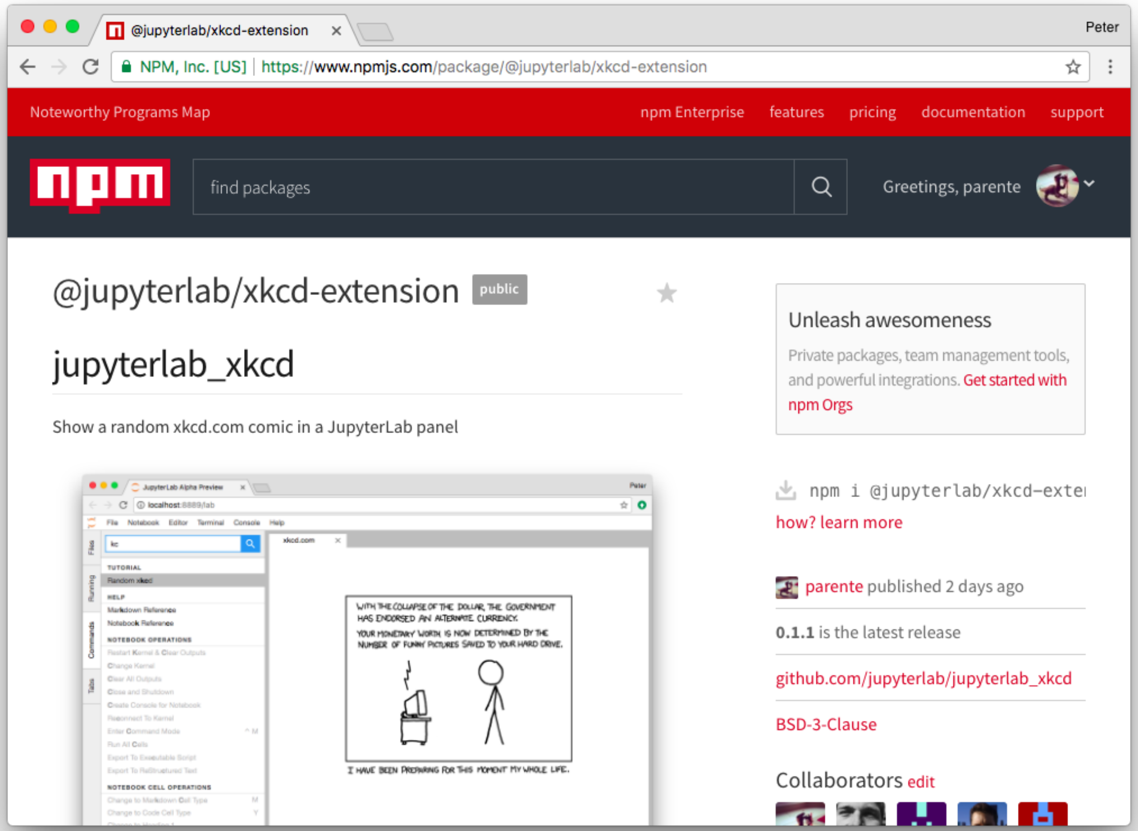
Return to your terminal window and make one more git commit:

```
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_xkcd` 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.


@jupyterlab/xkcd-extension
Peter
NPM, Inc. [US] https://www.npmjs.com/package/@jupyterlab/xkcd-extension
Noteworthy Programs Map
npm Enterprise
features
pricing
documentation
support
npm
find packages
Greetings, parente
WITH THE COLLAPSE OF THE DOLLAR, THE GOVERNMENT HAS ENDORSED AN ALTERNATE CURRENCY. YOUR MONETARY WORTH IS NOW DETERMINED BY THE NUMBER OF FUNNY PICTURES SAVED TO YOUR HARD DRIVE.
I HAVE BEEN PREPARING FOR THIS MOMENT MY WHOLE LIFE.

@jupyterlab/xkcd-extension public

jupyterlab_xkcd

Show a random xkcd.com comic in a JupyterLab panel

Unleash awesomeness

Private packages, team management tools, and powerful integrations. Get started with npm Orgs

npm i @jupyterlab/xkcd-exte

how? learn more

parente published 2 days ago

0.1.1 is the latest release

github.com/jupyterlab/jupyterlab_xkcd

BSD-3-Clause

Collaborators edit

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:

```
conda create -n jupyterlab-xkcd jupyterlab nodejs
source activate jupyterlab-xkcd
jupyter labextension install @your-npm-username/xkcd-extension
jupyter lab
```

You should see a fresh JupyterLab browser tab appear. When it does, execute the *Random xkcd comic* command to prove that your extension works when installed from npm.

26.7 Learn more

You've completed the tutorial. Nicely done! If you want to keep learning, here are some suggestions about what to try next:

- Assign a hotkey to the *Random xkcd comic* command.
- Make the image a link to the comic on https://xkcd.com.
- Push your extension git repository to GitHub.
- Give users the ability to pin comics in separate, permanent panels.
- Learn how to write other kinds of extensions.

CHAPTER 27

Indices and Tables

- genindex
- modindex
- search