

---

# JupyterLab Documentation

发布 *0.18.4*

Project Jupyter

2018 年 09 月 29 日



<b>1</b>	<b>Overview</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Starting JupyterLab</b>	<b>7</b>
<b>4</b>	<b>Frequently Asked Questions (FAQ)</b>	<b>9</b>
<b>5</b>	<b>JupyterLab Changelog</b>	<b>11</b>
<b>6</b>	<b>JupyterLab 接口 (API)</b>	<b>25</b>
<b>7</b>	<b>JupyterLab 链接(URLs)</b>	<b>31</b>
<b>8</b>	<b>File 的工作过程</b>	<b>33</b>
<b>9</b>	<b>文本编辑器</b>	<b>37</b>
<b>10</b>	<b>Notebooks</b>	<b>39</b>
<b>11</b>	<b>Code Consoles</b>	<b>41</b>
<b>12</b>	<b>终端 (Terminals)</b>	<b>43</b>
<b>13</b>	<b>管理内核和终端</b>	<b>45</b>
<b>14</b>	<b>命令调用板 (Command Palette)</b>	<b>47</b>
<b>15</b>	<b>文件和内核</b>	<b>49</b>
<b>16</b>	<b>文件和输出格式</b>	<b>51</b>
<b>17</b>	<b>扩展</b>	<b>59</b>
<b>18</b>	<b>JupyterLab 执行在 JupyterHub 里</b>	<b>63</b>
<b>19</b>	<b>General Codebase Orientation</b>	<b>65</b>
<b>20</b>	<b>Extension Developer Guide</b>	<b>67</b>

<b>21 Documents</b>	<b>75</b>
<b>22 Overview of document architecture</b>	<b>77</b>
<b>23 Notebook</b>	<b>81</b>
<b>24 Design Patterns</b>	<b>87</b>
<b>25 CSS Patterns</b>	<b>91</b>
<b>26 Writing Documentation</b>	<b>95</b>
<b>27 Virtual DOM and React</b>	<b>99</b>
<b>28 Adding Content</b>	<b>101</b>
<b>29 Examples</b>	<b>103</b>
<b>30 Terminology</b>	<b>107</b>
<b>31 Let's Make an xkcd JupyterLab Extension</b>	<b>109</b>
<b>32 Indices and Tables</b>	<b>125</b>

JupyterLab is the next-generation web-based user interface for Project Jupyter. [Try it on Binder](https://jupyter.readthedocs.io/en/latest/community/content-community.html). JupyterLab follows the Jupyter [Community Guides](https://jupyter.readthedocs.io/en/latest/community/content-community.html).

The screenshot displays the JupyterLab web interface. On the left, a sidebar shows a file browser with a list of notebooks: Data.ipynb, Fasta.ipynb, Julia.ipynb, Lorenz.ipynb (selected), R.ipynb, iris.csv, lightning.json, and lorenz.py. The main area is divided into three panes. The top pane shows the 'Lorenz.ipynb' notebook with a text cell explaining the Lorenz system of differential equations:

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

Below the equations, a text cell states: "Let's call the function once to view the solutions. For this set of parameters, we see the trajectories swirling around two points, called attractors." A code cell (In [4]) contains the following Python code:

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

The bottom-left pane shows the 'Output View' with three sliders for parameters: sigma (10.00), beta (2.67), and rho (28.00). Below the sliders is a 3D plot of the Lorenz attractor, showing a complex, swirling trajectory in a 3D space. The bottom-right pane shows the 'lorenz.py' file with the following Python code:

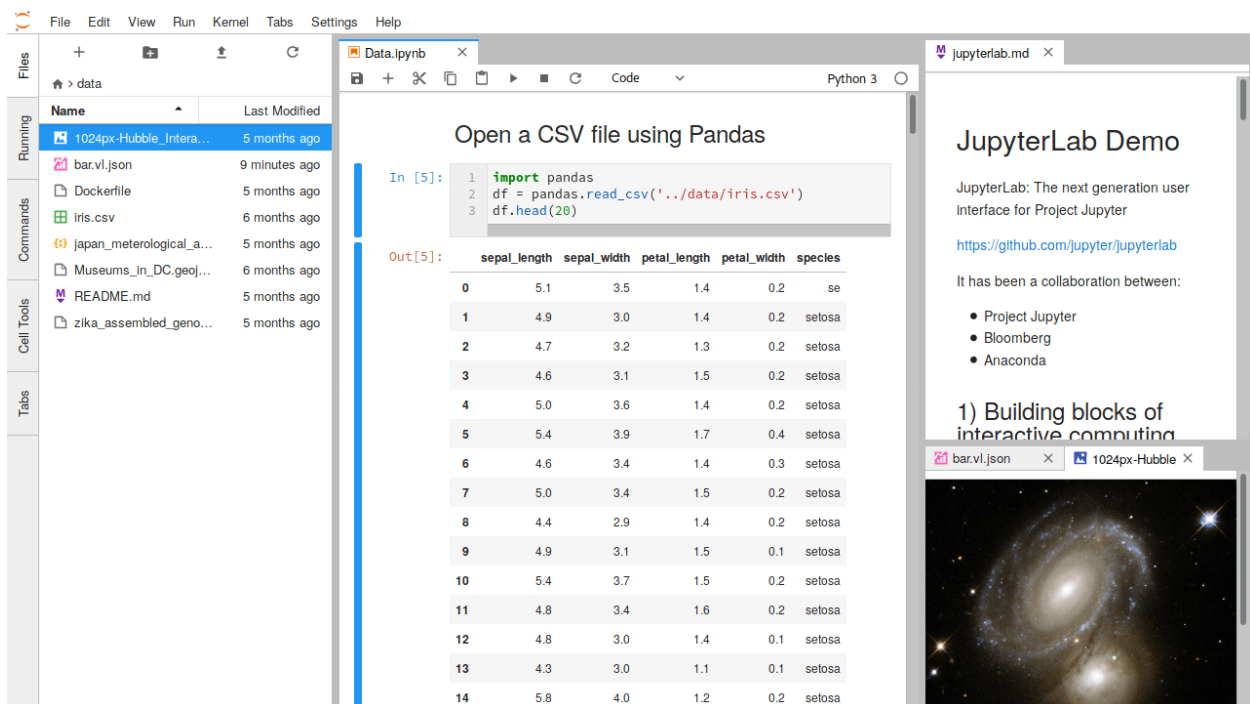
```
9 def solve_lorenz(N=10, max_time=4.0, sigma=10.0, beta=8./3, rho=28.0):
10     """Plot a solution to the Lorenz differential equations."""
11     fig = plt.figure()
12     ax = fig.add_axes([0, 0, 1, 1], projection='3d')
13     ax.axis('off')
14
15     # prepare the axes limits
16     ax.set_xlim((-25, 25))
17     ax.set_ylim((-35, 35))
18     ax.set_zlim((5, 55))
19
20     def lorenz_deriv(x,y,z, t0, sigma=sigma, beta=beta, rho=rho):
21         """Compute the time-derivative of a Lorenz system."""
22         x, y, z = x,y,z
23         return [sigma * (y - x), x * (rho - z) - y, x * y - beta * z]
24
25     # Choose random starting points, uniformly distributed from -15 to 15
26     np.random.seed(1)
27     x0 = -15 + 30 * np.random.random((N, 3))
28
```



# CHAPTER 1

## Overview

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



The screenshot displays the JupyterLab web interface. On the left is a file browser showing a directory structure with files like '1024px-Hubble\_Inter...', 'bar.vl.json', 'Dockerfile', 'Iris.csv', 'japan\_meteorological\_a...', 'Museums\_in\_DC.geoj...', 'README.md', and 'zika\_assembled\_geno...'. The main area contains a code editor titled 'Data.ipynb' with the following code:

```
In [5]: 1 import pandas
2 df = pandas.read_csv('../data/iris.csv')
3 df.head(20)
```

The output of the code is a table with 20 rows and 6 columns: 'sepal\_length', 'sepal\_width', 'petal\_length', 'petal\_width', 'species', and an unnamed column. The data is as follows:

	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	

On the right side of the interface is a sidebar titled 'JupyterLab Demo' which includes the text: 'JupyterLab: The next generation user interface for Project Jupyter', a link to 'https://github.com/jupyter/jupyterlab', and a list of collaborators: 'Project Jupyter', 'Bloomberg', and 'Anaconda'. Below this is a section titled '1) Building blocks of interactive computing' with a thumbnail image of a galaxy.

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

- *Code Consoles* provide transient scratchpads for running code interactively, with full support for rich output. A code console can be linked to a notebook kernel as a computation log from the notebook, for example.

- *Kernel-backed documents* enable code in any text file (Markdown, Python, R, LaTeX, etc.) to be run interactively in any Jupyter kernel.
- Notebook cell outputs can be *mirrored into their own tab*, side by side with the notebook, enabling simple dashboards with interactive controls backed by a kernel.
- Multiple views of documents with different editors or viewers enable live editing of documents reflected in other viewers. For example, it is easy to have live preview of *Markdown*, 分隔符分隔值, or *Vega/Vega-Lite* documents.

JupyterLab also offers a unified model for viewing and handling data formats. JupyterLab understands many file formats (images, CSV, JSON, Markdown, PDF, Vega, Vega-Lite, etc.) and can also display rich kernel output in these formats. See [文件和输出格式](#) for more information.

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

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

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

## 1.1 JupyterLab Releases

The current release of JupyterLab is suitable for general daily use. Please review the *JupyterLab Changelog* for detailed descriptions of each release.

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

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



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

### 2.1 conda

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

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

### 2.2 pip

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

```
pip install jupyterlab
```

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

### 2.3 pipenv

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

```
pipenv install jupyterlab  
pipenv shell
```

or from a git checkout:

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

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

```
pipenv shell
jupyter lab
```

## 2.4 Installing with Previous Versions of Notebook

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

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

## 2.5 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
```

## 2.6 Supported browsers

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

- Firefox
- Chrome
- Safari

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

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

## CHAPTER 3

---

### Starting JupyterLab

---

Start JupyterLab using:

```
jupyter lab
```

JupyterLab will open automatically in your browser.

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

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

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

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

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

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



---

### Frequently Asked Questions (FAQ)

---

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

#### 4.1 General

- [\*What is JupyterLab?\*](#)
- [\*Is JupyterLab ready to use?\*](#)
- [\*What will happen to the classic Jupyter Notebook?\*](#)
- [\*Where is the official online documentation for JupyterLab?\*](#)

#### 4.2 Development

- [\*How can you report a bug or provide feedback?\*](#)
- [\*How can you contribute?\*](#)
- [\*How can you extend or customize JupyterLab?\*](#)



### 5.1 v0.34.0

#### 5.1.1 August 18, 2018

See the [JupyterLab 0.34.0](#) milestone on GitHub for the full list of pull requests and issues closed.

#### 5.1.2 Key Features

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

#### 5.1.3 Changes for Developers

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

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

The theme name can also be targeted by writing CSS rules for `data-theme-name`. (#5078)

- The `IThemeManager` interface now exposes a signal for `themeChanged`, allowing extension authors to react to changes in the theme. Theme extensions must also provide a new boolean property `isLight`, declaring whether they are broadly light colored. This data allows third-party extensions to react better to the active application theme. (#5078)
- Added a patch to update the uploads for each `FileBrowserModel` instantly whenever a file upload errors. Previously, the upload that erred was only being removed from uploads upon an update. This would allow the status bar component and other extensions that use the `FileBrowserModel` to be more precise. (#5077)
- Cell IDs are now passed in the shell message as part of the cell metadata when a cell is executed. This helps in developing reactive kernels. (#5033)
- The IDs of all deleted cells since the last run cell are now passed as part of the cell metadata on execution. The IDs of deleted cells since the last run cell are stored as `deletedCells` in `NotebookModel`. This helps in developing reactive kernels. (#5037)
- The `ToolbarButton` in `apputils` has been refactored with an API change and now uses a React component `ToolbarButtonComponent` to render its children. It is now a `div` with a single `button` child, which in turn as two `span` elements for an icon and text label. Extensions that were using the `className` options should rename it as `iconClassName`. The `className` options still exists, but it used as the CSS class on the `button` element itself. The API changes were done to accommodate styling changes to the button. (#5117)
- The `Toolbar.createFromCommand` function has been replaced by a dedicated `ToolbarButton` subclass called `CommandToolbarButton`, that wraps a similarly named React component. (#5117)
- The design and styling of the right and left sidebars tabs has been improved to address #5054. We are now using icons to render tabs for the extensions we ship with JupyterLab and extension authors are encouraged to do the same (text labels still work). Icon based tabs can be used by removing `widget.caption` and adding `widget.iconClass = '<youriconclass> jp-SideBar-tabIcon';`. (#5117)
- The style of buttons in JupyterLab has been updated to a borderless design. (#5117)
- A new series of helper CSS classes for styling SVG-based icons at different sizes has been added: `jp-Icon`, `jp-Icon-16`, `jp-Icon-18`, `jp-Icon-20`.
- The rank of the default sidebar widget has been updated. The main change is giving the extension manager a rank of 1000 so that it appears at the end of the default items.
- Python 3.5+ is now required to use JupyterLab. Python 2 kernels can still be run within JupyterLab. (#5119)
- JupyterLab now uses `yarn 1.9.4` (aliased as `jlpm`), which now allows uses to use Node 10+. (#5121)
- Clean up handling of `baseUrl` and `wsURL` for `PageConfig` and `ServerConnection`. (#5111)

## 5.1.4 Other Changes

- Added the pipe (`|`) character as a CSV delimiter option. (#5112)
- Added `Open From Path...` to top level `File` menu. (#5108)
- Added a `saveState` signal to the document context object. (#5096)



- Added “Copy Download Link” to context menu for files. (#5089)
- Extensions marked as `deprecated` are no longer shown in the extension manager. (#5058)
- Remove `In` and `Out` text from cell prompts. Shrunk the prompt width from 90px to 64px. In the light theme, set the prompt colors of executed console cells to active prompt colors and reduced their opacity to 0.5. In the dark theme, set the prompt colors of executed console cells to active prompt colors and set their opacity to 1. (#5097 and #5130)

### 5.1.5 Bug Fixes

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

## 5.2 v0.33.0

### 5.2.1 July 26, 2018

See the [JupyterLab 0.33.0 milestone](#) on GitHub for the full list of pull requests and issues closed.

### 5.2.2 Key Features:

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

### 5.2.3 No longer in beta

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

## 5.2.4 Workspaces

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

## 5.2.5 Menu items

- “Activate Previously Used Tab” added to the Tab menu (`Ctrl/Cmd Shift '`) to toggle between the previously active tabs in the main area. (#4296)
- “Reload From Disk” added to the File menu to reload an open file from the state saved on disk. (#4615)
- “Save Notebook with View State” added to the File menu to persist the notebook collapsed and scrolled cell state. We now read the `collapsed`, `scrolled`, `jupyter.source_hidden` and `jupyter.outputs_hidden` notebook cell metadata when opening. `collapsed` and `jupyter.outputs_hidden` are redundant and the initial collapsed state is the union of both of them. When the state is persisted, if an output is collapsed, both will be written with the value `true`, and if it is not, both will not be written. (#3981)
- “Increase/Decrease Font Size” added to the text editor settings menu. (#4811)
- “Show in File Browser” added to a document tab’s context menu. (#4500)
- “Open in New Browser Tab” added to the file browser context menu. (#4315)
- “Copy Path” added to file browser context menu to copy the document’s path to the clipboard. (#4582)
- “Show Left Area” has been renamed to “Show Left Sidebar” for consistency (same for right sidebar). (#3818)

## 5.2.6 Keyboard shortcuts

- “Save As...” given the keyboard shortcut `Ctrl/Cmd Shift S`. (#4560)
- “Run All Cells” given the keyboard shortcut `Ctrl/Cmd Shift Enter`. (#4558)
- “notebook:change-to-cell-heading-X” keyboard shortcuts (and commands) renamed to “notebook:change-cell-to-heading-X” for  $X=1\ldots6$ . This fixes the notebook command-mode keyboard shortcuts for changing headings. (#4430)
- The console execute shortcut can now be set to either `Enter` or `Shift Enter` as a Console setting. (#4054)

## 5.2.7 Command palette items

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

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

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

### 5.2.8 Settings

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

### 5.2.9 Larger file uploads

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

### 5.2.10 Extension management and installation

- New extension manager for installing JupyterLab extensions from npm within the JupyterLab UI. You can enable this from the Advanced Settings interface. (#4682, #4925)
- Please note that to install extensions in JupyterLab, you must use NodeJS version 9 or earlier (i.e., not NodeJS version 10). We will upgrade yarn, with NodeJS version 10 support, when a [bug in yarn](#) is fixed. (#4804)

### 5.2.11 Interface changes

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

### 5.2.12 Renderers

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

### 5.2.13 Changes for developers

- A new signal for observing application dirty status state changes. (#4840)
- A new signal for observing notebook cell execution. (#4740, #4744)
- A new anyMessage signal for observing any message a kernel sends or receives. (#4437)
- A generic way for different widgets to register a “Save with extras” command that appears in the File menu under save. (#3981)

- A new API for removing groups from a JupyterLab menu. `addGroup` now returns an `IDisposable` which can be used to remove the group. `removeGroup` has been removed. (#4890)
- The `Launcher` now uses commands from the application `CommandRegistry` to launch new activities. Extension authors that add items to the launcher will need to update them to use commands. (#4757)
- There is now a top-level `addBottomArea` function in the application, allowing extension authors to add bottom panel items like status bars. (#4752)
- `Rendermime` extensions can now indicate that they are the default rendered widget factory for a file-type. For instance, the default widget for a markdown file is a text editor, but the default rendered widget is the markdown viewer. (#4692)
- Add new workspace REST endpoints to `jupyterlab_launcher` and make them available in `@jupyterlab/services`. (#4841)
- Documents created with a `mimerenderer` extension can now be accessed using an `IInstanceTracker` which tracks them. Include the token `IMimeDocumentTracker` in your plugin to access this. The `IInstanceTracker` interface has also gained convenience functions `find` and `filter` to simplify iterating over instances. (#4762)
- `RenderMime` render errors are now displayed to the user. (#4465)
- `getNotebookVersion` is added to the `PageConfig` object. (#4224)
- The session `kernelChanged` signal now contains both the old kernel and the new kernel to make it easy to unregister things from the old kernel. (#4834)
- The `connectTo` functions for connecting to kernels and sessions are now synchronous (returning a connection immediately rather than a promise). The `DefaultSession` `clone` and `update` methods are also synchronous now. (#4725)
- Kernel message processing is now asynchronous, which guarantees the order of processing even if a handler is asynchronous. If a kernel message handler returns a promise, kernel message processing is paused until the promise resolves. The kernel's `anyMessage` signal is emitted synchronously when a message is received before asynchronous message handling, and the `iopubMessage` and `unhandledMessage` signals are emitted during asynchronous message handling. These changes mean that the `comm` `onMsg` and `onClose` handlers and the kernel future `onReply`, `onIOPub`, and `onStdin` handlers, as well as the `comm` target and message hook handlers, may be asynchronous and return promises. (#4697)
- Kernel `comm` targets and message hooks now are unregistered with `removeCommTarget` and `removeMessageHook`, instead of using disposables. The corresponding `registerCommTarget` and `registerMessageHook` functions now return nothing. (#4697)
- The kernel `connectToComm` function is synchronous, and now returns the `comm` rather than a promise to the `comm`. (#4697)
- The `KernelFutureHandler` class `expectShell` constructor argument is renamed to `expectReply`. (#4697)
- The kernel future `done` returned promise now resolves to `undefined` if there is no reply message. (#4697)
- The `IDisplayDataMsg` is updated to have the optional `transient` key, and a new `IUpdateDisplayDataMsg` type was added for update display messages. (#4697)
- The `uuid` function from `@jupyterlab/coreutils` is removed. Instead import `UUID` from `@phosphor/coreutils` and use `UUID.uuid4()`. (#4604)
- Main area widgets like the launcher and console inherit from a common `MainAreaWidget` class which provides a content area (`.content`) and a toolbar (`.toolbar`), consistent focus handling and activation behavior, and a spinner displayed until the given `reveal` promise is resolved. Document widgets, like the notebook and

text editor and other documents opened from the document manager, implement the `IDocumentWidget` interface (instead of `DocumentRegistry.IReadyWidget`), which builds on `MainAreaWidget` and adds a `.context` attribute for the document context and makes dirty handling consistent. Extension authors may consider inheriting from the `MainAreaWidget` or `DocumentWidget` class for consistency. Several effects from these changes are noted below. (#3499, #4453)

- The notebook panel `.notebook` attribute is renamed to `.content`.
- The text editor is now the `.content` of a `DocumentWidget`, so the top-level editor widget has a toolbar and the editor itself is `widget.content.editor` rather than just `widget.editor`.
- Mime documents use a `MimeContent` widget embedded inside of a `DocumentWidget` now.
- Main area widgets and document widgets now have a `revealed` promise which resolves when the widget has been revealed (i.e., the spinner has been removed). This should be used instead of the `ready` promise.

Changes in the JupyterLab code infrastructure include:

- The JupyterLab TypeScript codebase is now compiled to ES2015 (ES6) using TypeScript 2.9. We also turned on the TypeScript `esModuleInterop` flag to enable more natural imports from non-es2015 JavaScript modules. With the update to ES2015 output, code generated from `async/await` syntax became much more manageable, so we have started to use `async/await` liberally throughout the codebase, especially in tests. Because we use Typedoc for API documentation, we still use syntax compatible with TypeScript 2.7 where Typedoc is used. Extension authors may have some minor compatibility updates to make. If you are writing an extension in TypeScript, we recommend updating to TypeScript 2.9 and targeting ES2015 output as well. (#4462, #4675, #4714, #4797)
- The JupyterLab codebase is now formatted using Prettier. By default the development environment installs a pre-commit hook that formats your staged changes. (#4090)
- Updated build infrastructure using webpack 4 and better typing. (#4702, #4698)
- Upgraded yarn to version 1.6. Please note that you must use NodeJS version 9 or earlier with JupyterLab (i.e., not NodeJS version 10). We will upgrade yarn, with NodeJS version 10 support, when a bug in yarn is fixed. (#4804)
- Various process utilities were moved to `jupyterlab_launcher`. (#4696)

## 5.2.14 Other fixes

- Fixed a rendering bug with the Launcher in single-document mode. (#4805)
- Fixed a bug where the native context menu could not be triggered in a notebook cell in Chrome. (#4720)
- Fixed a bug where the cursor would not show up in the dark theme. (#4699)
- Fixed a bug preventing relative links from working correctly in alternate `IDrives`. (#4613)
- Fixed a bug breaking the image viewer upon saving the image. (#4602)
- Fixed the font size for code blocks in notebook Markdown headers. (#4617)
- Prevented a memory leak when repeatedly rendering a Vega chart. (#4904)
- Support dropped terminal connection re-connecting. (#4763, #4802)
- Use `require.ensure` in `vega4-extension` to lazily load `vega-embed` and its dependencies on first render. (#4706)
- Relative links to documents that include anchor tags will now correctly scroll the document to the right place. (#4692)
- Fix default settings JSON in setting editor. (#4591, #4595)

- Fix setting editor pane layout's stretch factor. (#2971, #4772)
- Programmatically set settings are now output with nicer formatting. (#4870)
- Fixed a bug in displaying one-line CSV files. (#4795, #4796)
- Fixed a bug where JSON arrays in rich outputs were collapsed into strings. (#4480)

## 5.3 Beta 2 (v0.32.0)

### 5.3.1 Apr 16, 2018

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

- Better handling of a corrupted or invalid state database. (#3619, #3622, #3687, #4114).
- Fixing file dirty status indicator. (#3652).
- New option for whether to autosave documents. (#3734).
- More commands in the notebook context menu. (#3770, #3909)
- Defensively checking for completion metadata from kernels. (#3888)
- New “Shutdown all” button in the Running panel. (#3764)
- Performance improvements wherein non-focused documents poll the server less. (#3931)
- Changing the keyboard shortcut for singled-document-mode to something less easy to trigger. (#3889)
- Performance improvements for rendering text streams, especially around progress bars. (#4045).
- Canceling a “Restart Kernel” now functions correctly. (#3703).
- Defer loading file contents until after the application has been restored. (#4087).
- Ability to rotate, flip, and invert images in the image viewer. (#4000)
- Major performance improvements for large CSV viewing. (#3997).
- Always show the context menu in the file browser, even for an empty directory. (#4264).
- Handle asynchronous comm messages in the services library more correctly (Note: this means @jupyterlab/services is now at version 2.0!) ([#4115](https://github.com/jupyterlab/jupyterlab/issues/4115)).
- Display the kernel banner in the console when a kernel is restarted to mark the restart ([#3663](https://github.com/jupyterlab/jupyterlab/issues/3663)).
- Many tweaks to the UI, as well as better error handling.

## 5.4 Beta 1 (v0.31.0)

### 5.4.1 Jan 11, 2018

- Add a /tree handler and Copy Shareable Link to file listing right click menu: <https://github.com/jupyterlab/jupyterlab/pull/3396>
- Experimental support for saved workspaces: #3490, #3586
- Added types information to the completer: #3508

- More improvements to the top level menus: <https://github.com/jupyterlab/jupyterlab/pull/3344>
- Editor settings for notebook cells: <https://github.com/jupyterlab/jupyterlab/pull/3441>
- Simplification of theme extensions: <https://github.com/jupyterlab/jupyterlab/pull/3423>
- New CSS variable naming scheme: <https://github.com/jupyterlab/jupyterlab/pull/3403>
- Improvements to cell selection and dragging: <https://github.com/jupyterlab/jupyterlab/pull/3414>
- Style and typography improvements: <https://github.com/jupyterlab/jupyterlab/pull/3468> <https://github.com/jupyterlab/jupyterlab/pull/3457> <https://github.com/jupyterlab/jupyterlab/pull/3445> <https://github.com/jupyterlab/jupyterlab/pull/3431> <https://github.com/jupyterlab/jupyterlab/pull/3428> <https://github.com/jupyterlab/jupyterlab/pull/3408> <https://github.com/jupyterlab/jupyterlab/pull/3418>

## 5.5 v0.30.0

### 5.5.1 Dec 05, 2017

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

## 5.6 v0.29.2

### 5.6.1 Nov 17, 2017

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

## 5.7 v0.29.0

### 5.7.1 Nov 09, 2017

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

## 5.8 v0.28.0

### 5.8.1 Oct 16, 2017

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

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

## 5.9 v0.27.0

### 5.9.1 Aug 23, 2017

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

## 5.10 v0.26.0

### 5.10.1 Jul 21, 2017

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

## 5.11 v0.25.0

### 5.11.1 Jul 07, 2017

- Added a new extension type for mime renderers, with the `vega2-extension` as a built-in example. Also overhauled the `rendermime` interfaces. <https://github.com/jupyterlab/jupyterlab/pull/2488> <https://github.com/>



[jupyterlab/jupyterlab/pull/2555](https://github.com/jupyterlab/jupyterlab/pull/2555) <https://github.com/jupyterlab/jupyterlab/pull/2595>

- Finished JSON-schema based settings system, using client-side storage for now. <https://github.com/jupyterlab/jupyterlab/pull/2411>
- Overhauled the launcher design. <https://github.com/jupyterlab/jupyterlab/pull/2506> <https://github.com/jupyterlab/jupyterlab/pull/2580>
- Numerous bug fixes and style updates.

## 5.12 v0.24.0

### 5.12.1 Jun 16, 2017

- Overhaul of the launcher. #2380
- Initial implementation of client-side settings system. #2157
- Updatable outputs. #2439
- Use new Phosphor Datagrid for CSV viewer. #2433
- Added ability to enable/disable extensions without rebuilding. #2409
- Added language and tab settings for the file viewer. #2406
- Improvements to real time collaboration experience. #2387 #2333
- Compatibility checking for extensions. #2410
- Numerous bug fixes and style improvements.

## 5.13 v0.23.0

### 5.13.1 Jun 02, 2017

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

## 5.14 v0.22.0

### 5.14.1 May 18, 2017

- Export To... for notebooks. <https://github.com/jupyterlab/jupyterlab/pull/2200>
- Change kernel by clicking on the kernel name in the notebook. <https://github.com/jupyterlab/jupyterlab/pull/2195>

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

## 5.15 v0.20.0

### 5.15.1 Apr 21, 2017

Release Notes:

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

Numerous bug fixes and improvements.

## 5.16 v0.19.0

### 5.16.1 Apr 04, 2017

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

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

## 5.17 v0.18.0

### 5.17.1 Mar 21, 2017

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

- Numerous bug fixes and style updates.

## 5.18 v0.17.0

### 5.18.1 Mar 01, 2017

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

## 5.19 v0.16.0

### 5.19.1 Feb 09, 2017

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



## JupyterLab 接口 (API)

JupyterLab 为交互式探索性计算提供灵活的构建块。虽然 JupyterLab 具有传统集成开发环境 (IDE) 中的许多功能，但它仍然专注于交互式探索性计算。

JupyterLab 界面包含一个主工作区，包含文档和活动选项卡，可折叠左侧边栏和菜单栏。左侧边栏包含文件浏览器，正在运行的内核和终端列表，命令选项板，笔记本单元工具检查器和选项卡列表。

The screenshot displays the JupyterLab interface. On the left is a sidebar with sections: Files, Running, Commands, Cell Tools, and Tabs. The 'Files' section shows a file browser for the 'data' directory, listing files like '1024px-Hubble\_Intera...', 'bar.vl.json', 'Dockerfile', 'iris.csv', 'japan\_meteorological\_a...', 'Museums\_in\_DC.geoj...', 'README.md', and 'zika\_assembled\_gen...' with their last modified times. The main area is divided into two panes. The left pane, titled 'Data.ipynb', shows a code editor with the following code:

```

In [5]: 1 import pandas
        2 df = pandas.read_csv('../data/iris.csv')
        3 df.head(20)

```

The right pane shows the output of the code, which is a table of the first 15 rows of the iris dataset:

	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

The right pane is titled 'JupyterLab Demo' and contains text about JupyterLab being the next generation user interface for Project Jupyter, a link to the GitHub repository, and a list of collaborators: Project Jupyter, Bloomberg, and Anaconda. Below this is a section titled '1) Building blocks of interactive computing' with a sub-panel showing a galaxy image.

JupyterLab 会话始终驻留在工作空间中。工作区包含 JupyterLab 的状态：当前打开的文件，应用程序区域和选项卡的布局等。工作区可以使用命名工作区 URL 保存在服务器上。要了解有关 Jupyterlab 中 URL 的更多信息，请访问 [JupyterLab 链接\(URLs\)](#)。

## 6.1 菜单栏 (Menu Bar)

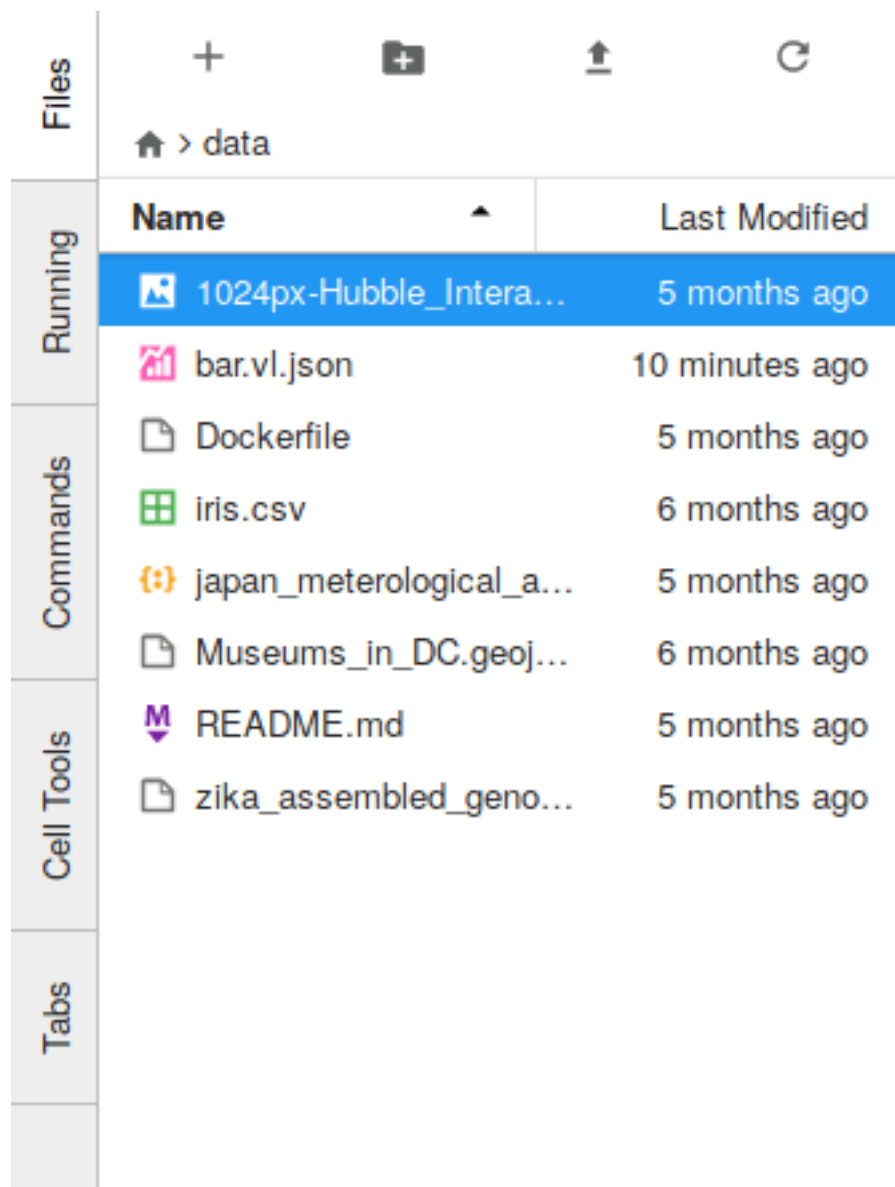
JupyterLab顶部的菜单栏有一个顶级菜单，允许您通过键盘快捷键显示 JupyterLab 中可用的操作。默认菜单是：

- **File:** 与文件和目录相关的操作
- **Edit:** 与编辑文件和其他活动有关的行动
- **View:** 改变 JupyterLab 外观的动作
- **Run:** 用于在不同活动中运行代码的操作，例如 notebooks 和 consoles
- **Kernel:** 用于管理内核的操作，内核是用于运行代码的单独进程
- **Tabs:** 停靠面板中的打开文档和活动列表
- **Settings:** 常用设置和高级设置编辑器
- **Help:** JupyterLab 和内核帮助链接列表

*JupyterLab extensions* 还可以在菜单栏中创建新的顶级菜单。

## 6.2 左侧边栏 (Left Sidebar)

左侧边栏包含许多常用选项卡，例如文件浏览器，正在运行的内核和终端列表，命令选项板以及主工作区中的选项卡列表：



通过在“视图”菜单中选择“显

示左侧边栏”或单击活动侧边栏选项卡，可以折叠或展开左侧边栏：

JupyterLab 扩展可以向左侧边栏添加其他面板。

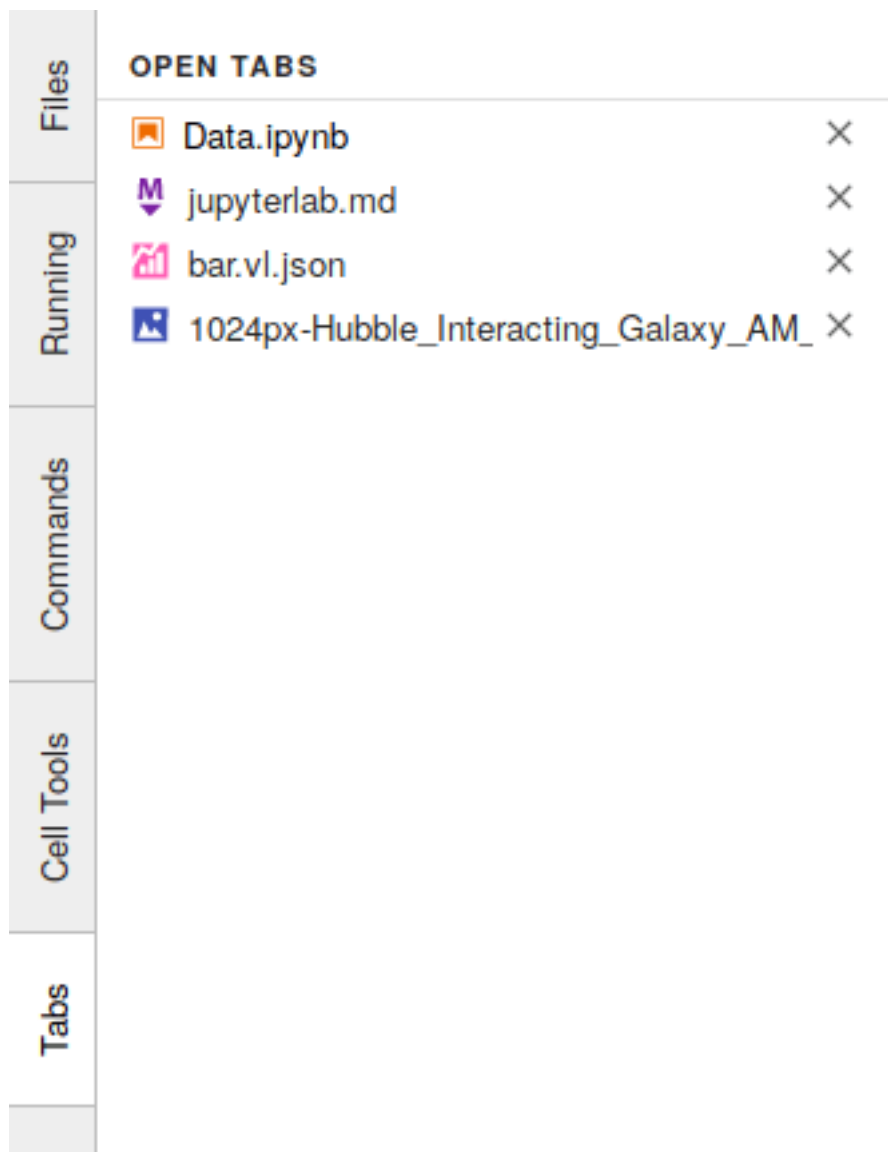
## 6.3 主工作区 (Main Work Area)

JupyterLab中的 主工作区 使您可以将文档（notebooks，文本文件等）和其他活动（terminals，code consoles等）排列到可以调整大小或细分的选项卡面板中。将选项卡拖动到选项卡面板的中心可将选项卡移动到面板。通过将选项卡拖动到面板的左侧，右侧，顶部或底部来细分选项卡面板：

工作区域只有一个当前活动。当前活动的选项卡标有彩色顶部边框（默认为蓝色）。

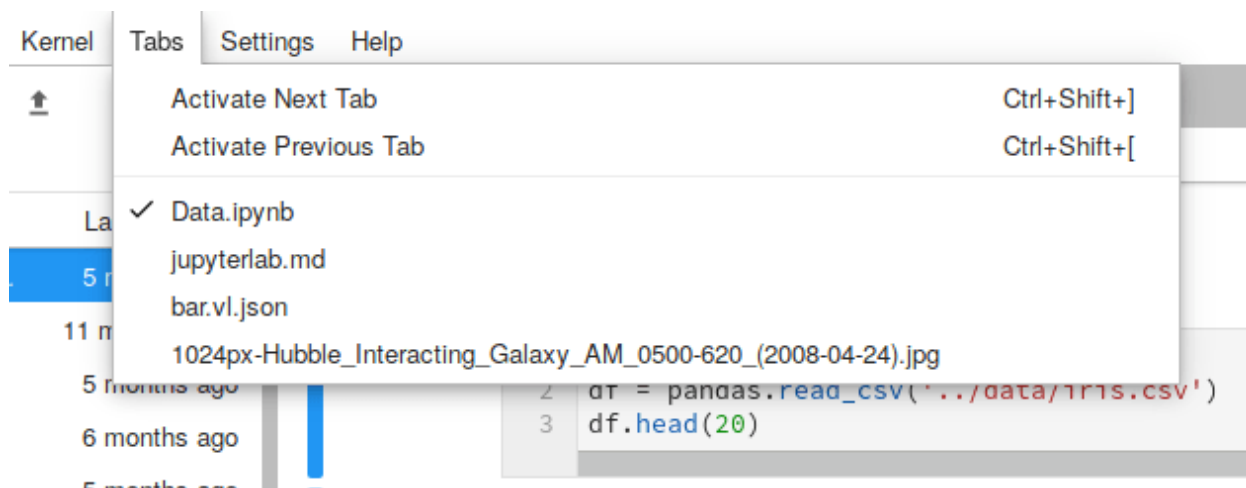
## 6.4 标签 (tabs) 和单文档模式

左侧栏中的“选项卡”面板列出了主要工作区中的打开文档或活动：



选项卡菜单中也提供了相同的信息：





在不关闭主工作区域中的其他选项卡的情况下，关注单个文档或活动通常很有用。单文档模式启用此功能，同时可以轻松返回主工作区中的多活动布局。使用“视图”菜单切换单文档模式：

离开单文档模式时，将恢复主区域的原始布局。

## 6.5 上下文菜单 (Context Menus)

JupyterLab 的许多部分（例如 notebooks，文本文件，code consoles 和选项卡）都有上下文菜单，可以通过右键单击元素来访问它们：

可以通过按住“Shift”和右键单击来访问浏览器的本机上下文菜单：

## 6.6 键盘快捷键

与经典 Notebook 一样，您可以通过键盘快捷键导航用户界面。您可以通过选择“设置”菜单中的“高级设置编辑器”项，然后在“设置”选项卡中选择“键盘快捷键”来查找和自定义当前键盘快捷键列表。

您还可以使用“设置”菜单中的“ref:‘文本编辑器<file-editor>’键映射”子菜单自定义文本编辑器以使用 vim，emacs 或 Sublime Text 键盘映射：



---

## JupyterLab 链接(URLs)

---

与经典 `notebook` 一样，JupyterLab 为用户提供了一种复制打开特定笔记本或文件的 URL 的方法。此外，JupyterLab 连接(URLs)是用户界面的高级部分，允许管理工作区。这两个函数 - 文件路径和工作空间 - 可以组合在打开特定工作空间中特定文件的 URL 中。.. `_url-tree`:

### 7.1 文件导航 `/tree`

JupyterLab 的文件导航 URL 采用经典 `notebook` 的命名法; 这些 URL 是 `/tree` URLs:

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

输入此 URL 将以 `单文档模式` 在 JupyterLab 中打开 `notebook`。

### 7.2 管理工作区

JupyterLab 会话始终驻留在工作空间中。工作区包含 JupyterLab 的状态: 当前打开的文件，应用程序区域和选项卡的布局等。刷新页面时，将还原工作区。

默认工作空间未命名，仅将其状态保存在用户的本地浏览器中:

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

命名工作区将其状态保存在服务器上，并且可以在多个用户（或浏览器）之间共享，只要他们可以访问同一服务器:

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

工作区应该一次只能在一个浏览器选项卡中打开。如果 JupyterLab 检测到同时多次打开工作空间，则会提示输入新的工作空间名称。同时也不支持在两个不同的浏览器选项卡中打开文档。

## 7.3 克隆工作区

您可以使用 `clone url` 参数将工作空间的内容复制到另一个工作空间。

要将工作区 `foo` 的内容复制到工作区 `bar` 中:

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

要将默认工作空间的内容复制到工作空间 `foo` 中:

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

要将工作空间 `foo` 的内容复制到默认工作空间:

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

## 7.4 重置工作区

使用 `reset url` 参数清除其内容的工作空间。

要重置工作空间 `foo` 的内容:

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

要重置默认工作区的内容:

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

## 7.5 组合URL功能

这些URL功能可以单独使用，如上所述，也可以组合使用。

要重置工作区 `foo` 并在以后加载特定 `notebook`:

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

要将工作区栏的内容克隆到工作区 `foo` 中，然后加载 `notebook` :

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

要重置默认工作区的内容并加载 `notebook`:

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

---

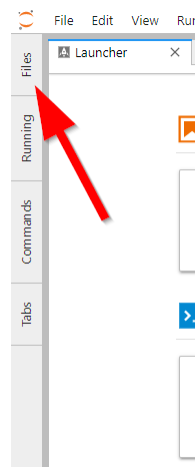
### File 的工作过程

---

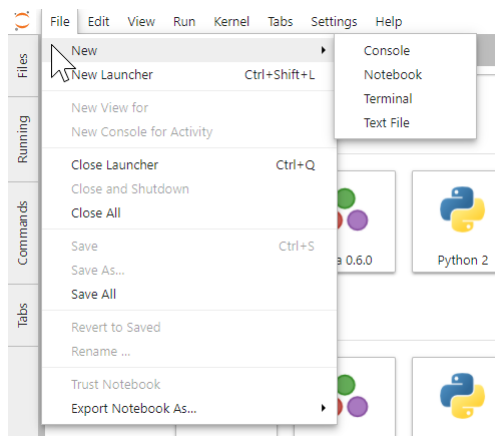
#### 8.1 打开文件

使用文件浏览器和“文件”菜单可以处理系统上的文件和目录。这包括打开，创建，删除，重命名，下载，复制和共享文件和目录。

文件浏览器位于左侧边栏“文件”选项卡中：



对文件的许多操作也可以在“文件”菜单中执行：



要打开任何文件，请在文件浏览器中双击其名称：

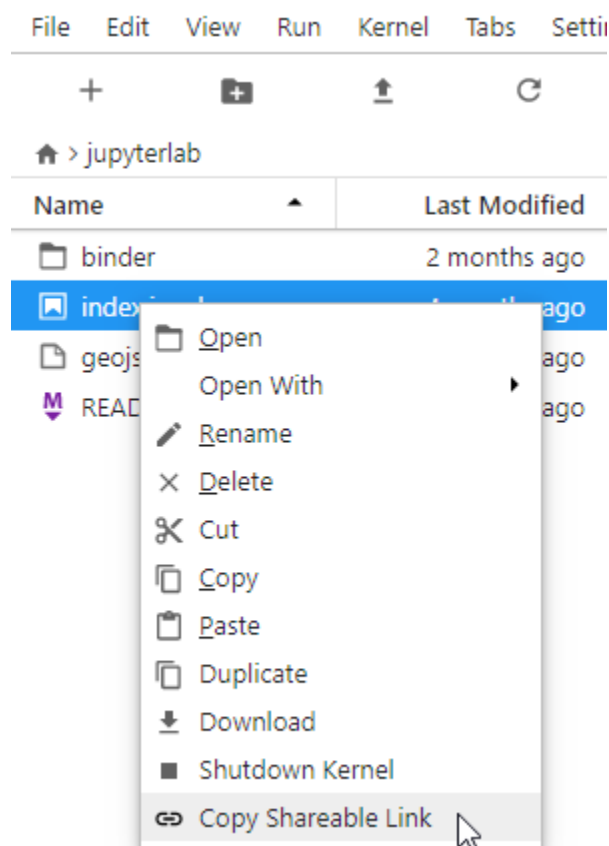
你还可以将文件拖到主工作区中以创建新选项卡：

许多文件类型都有:ref:多个视图/编辑器 <file-and-output-formats>。例如，您可以在 [文本编辑器](#) 中打开Markdown文件，也可以将其作为呈现的HTML打开。JupyterLab 扩展还可以为文件添加新的查看器/编辑器。要在非默认查看器/编辑器中打开文件，请在文件浏览器中右键单击其名称，然后使用“Open With...”子菜单选择查看器/编辑器：

可以在多个查看器/编辑器中同时打开单个文件，它们将保持同步：

可以通过双击列表中的文件夹或单击目录列表顶部的文件夹来导航文件系统：

右键单击文件或目录，然后选择“Copy Shareable Link”以复制可用于打开JupyterLab的URL，并打开该文件或目录。

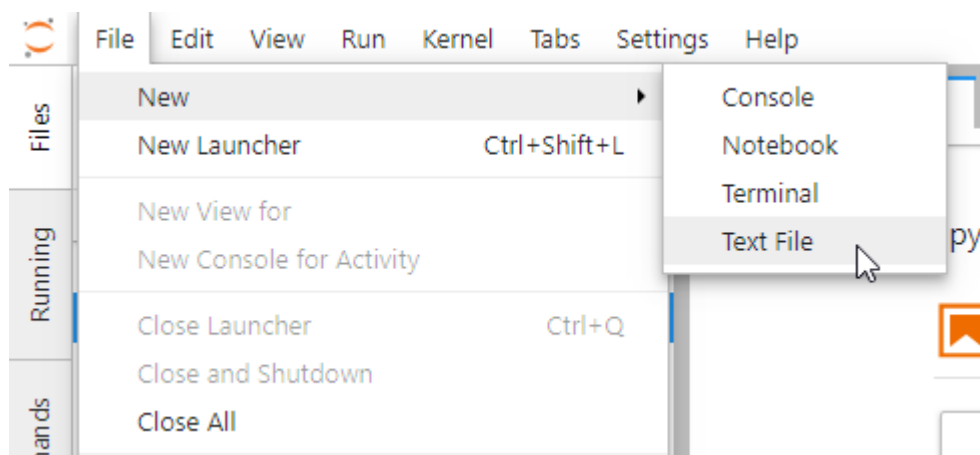


右键单击文件或目录，然后选择“Copy Path”以复制文件  
系统相对路径。这可以用于将参数传递给各种内核中调用的函数中的打开文件。

## 8.2 创建文件和活动

单击文件浏览器顶部的 + 按钮创建新文件或活动。这将在主工作区中打开一个新的Launcher选项卡，使您可以选择一个活动和内核：

您还可以使用“文件”菜单创建新文档或活动：



新活动或文档的当前工作目录将是文件浏览器中列出的目录（**Terminal**除外，它始终在文件浏览器的根目录中启动）：

使用默认名称创建新文件。通过在文件浏览器中右键单击其名称并从上下文菜单中选择“Rename”来重命名文件：

## 8.3 上传和下载

通过将文件拖放到文件浏览器上，或者通过单击文件浏览器顶部的“Upload Files”按钮，可以将文件上传到文件浏览器的当前目录：

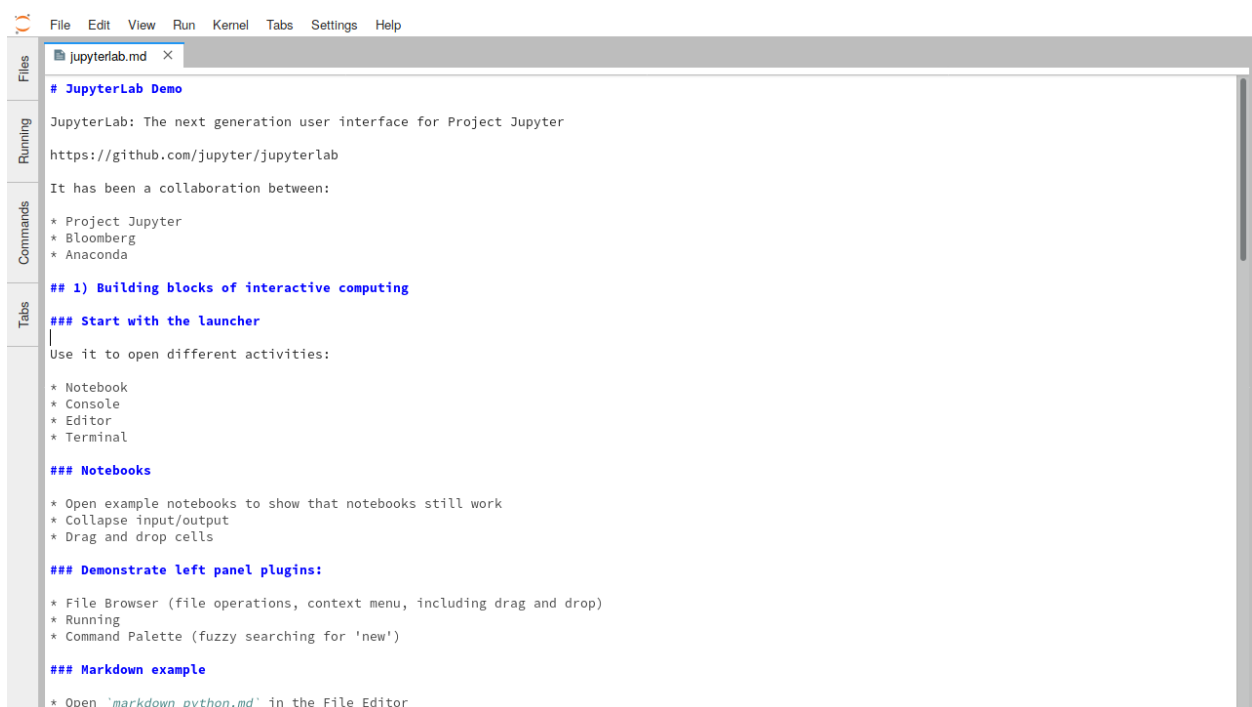
可以通过在文件浏览器中右键单击其名称并从上下文菜单中选择“Download”来下载JupyterLab中的任何文件：



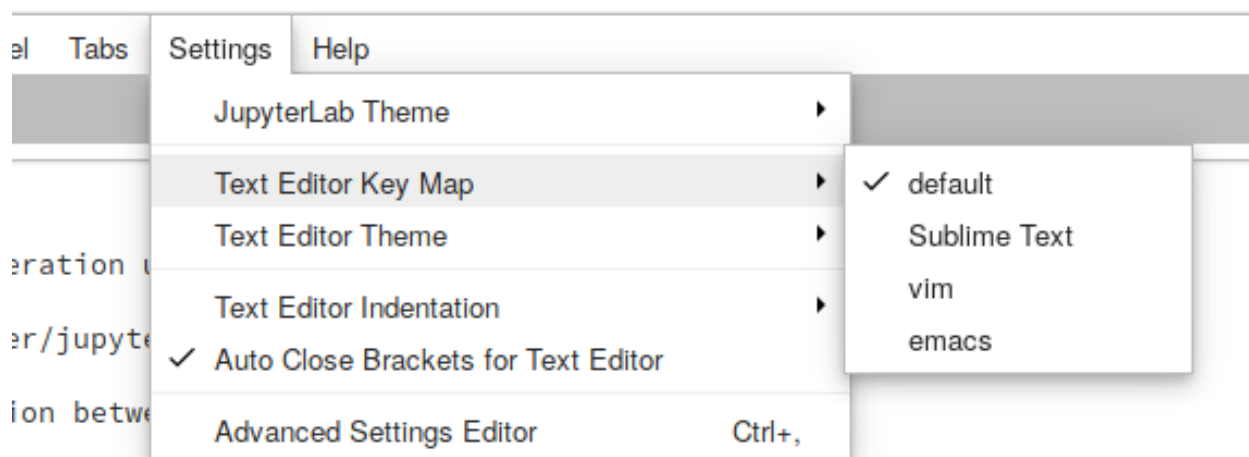


## 文本编辑器

JupyterLab中的文本编辑器使您可以在JupyterLab中编辑文本文件：



文本编辑器包括语法突出显示，可配置缩进（制表符或空格），*key maps* 和基本主题。可以在“settings”菜单中找到这些设置：



## interactive computing

### ier

#### activities:

要编辑现有文本文件，请在文件浏览器中双击其名称或将其拖到主工作区：

要在文件浏览器的当前目录中创建新的文本文件，请单击文件浏览器顶部的 + 按钮以创建新的Launcher选项卡，然后单击Launcher中的“Text Editor”卡：

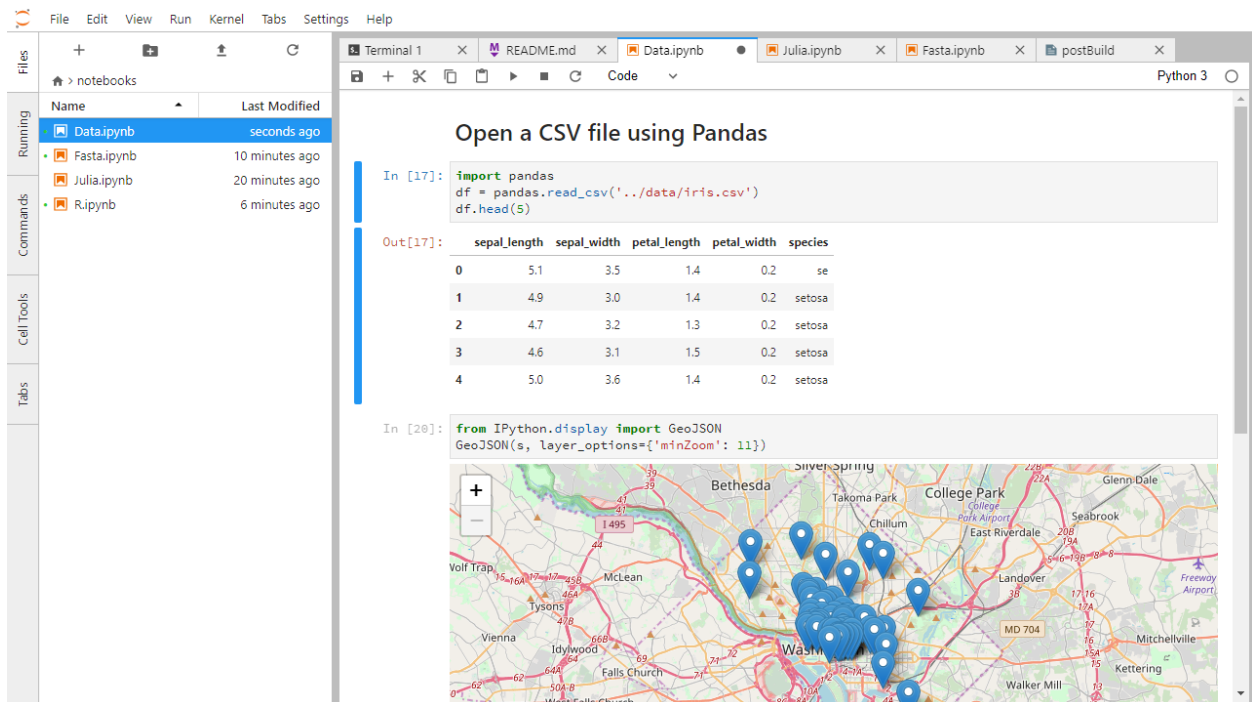
您还可以使用“文件”菜单创建新的文本文件：

使用默认名称创建新文件。通过在文件浏览器中右键单击其名称并从上下文菜单中选择“Rename”来重命名文件：

# CHAPTER 10

## Notebooks

**Jupyter notebooks** 是将实时可运行代码与描述文本（Markdown），方程式（LaTeX），images，交互式可视化和其他丰富输出相结合的文档：



**JupyterLab**完全支持**Jupyter**笔记本（.ipynb文件）。JupyterLab中使用的笔记本文档格式与经典的Jupyter Notebook相同。您现有的笔记本应该在JupyterLab中正确打开。如果他们不这样做，请在我们的 [GitHub issues](#) 页面上打开一个问题。

单击文件浏览器中的 + 按钮，然后在新的Launcher选项卡中选择内核，创建一个笔记本：

使用默认名称创建新文件。通过在文件浏览器中右键单击其名称并从上下文菜单中选择“Rename”来重命名文件：

JupyterLab 中 Notebooks 的用户界面紧跟经典 Jupyter Notebook 的用户界面。经典笔记本的键盘快捷键继续工作（使用命令和编辑模式）。但是，JupyterLab 中的笔记本电脑可能会有许多新功能。

拖放单元格以重新排列笔记本：

在笔记本之间拖动单元格以快速复制内容：

创建单个 notebook 的多个同步 views：

使用 View menu 或每个单元格左侧的蓝色伸缩按钮折叠并展开代码和输出：

通过右键单击单元格并选择“为输出启用滚动”，启用长输出滚动：

创建单元格输出的新同步视图：

选项卡完成（使用 Tab 键激活）现在可以包含有关匹配项类型的其他信息：

注意：IPython 6.3.1 暂时禁用了类型注释。要重新启用它们，请将“`c.Completer.use_jedi = True`”添加到 `ipython_config.py` 文件中。

工具提示（使用 Shift Tab 激活）包含有关对象的其他信息：

您可以将 [代码控制台](#) 连接到笔记本内核，以便按照完成它们的顺序在内核中完成计算日志。附加的代码控制台还提供了交互式检查内核状态而无需更改笔记本的位置。右键单击笔记本并选择“New Console for Notebook”：

# CHAPTER 11

---

## Code Consoles

---

**Code Consoles** 使您能够在内核中以交互方式运行代码。**Code Consoles** 的单元格显示了代码在内核中执行的顺序，而不是 **notebook** 文档中单元格的显式排序。代码控制台也显示丰富的输出，就像笔记本电脑一样。

单击 [文件浏览器](#) 中的 + 按钮并选择内核，创建一个新的代码控制台：

使用 Shift Enter 运行代码。使用向上和向下箭头浏览以前运行的代码的历史记录：

选项卡完成 (Tab) 和工具提示 (Shift Tab) 在笔记本中起作用：

通过右键单击代码控制台并选择“**Clear Console Cells**”，清除代码控制台的单元格而不重新启动内核：

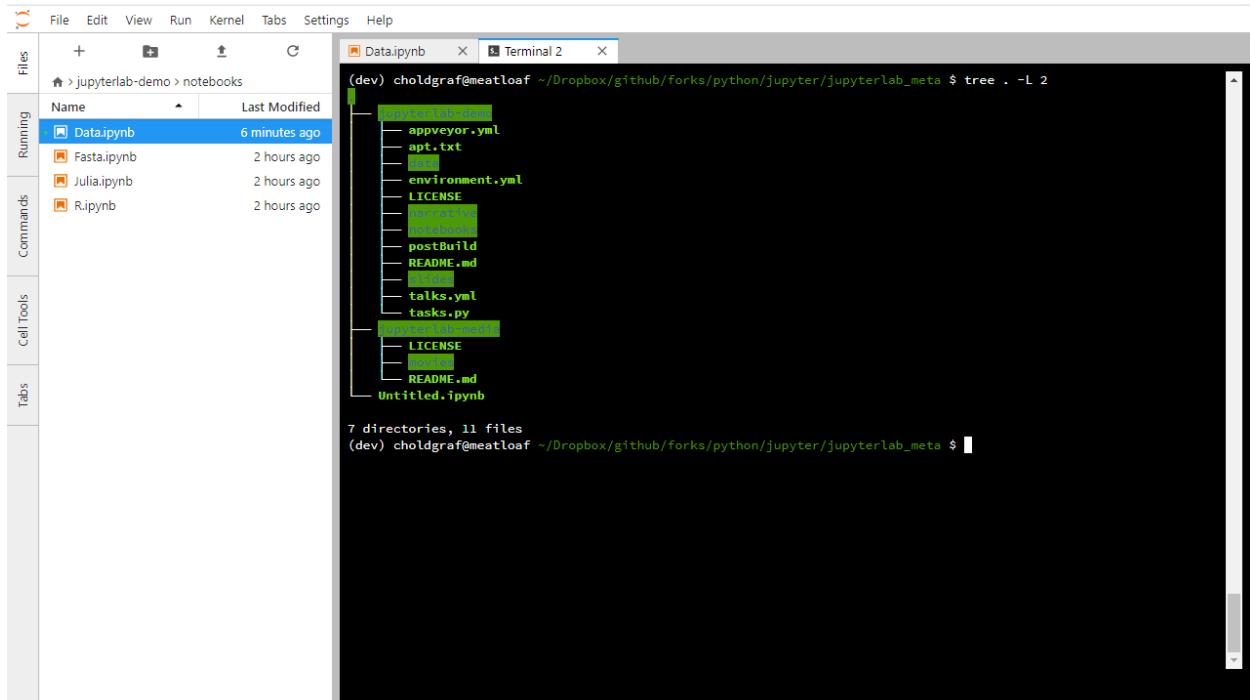
从 [file menu](#) 创建代码控制台可让您为代码控制台选择现有内核。然后代码控制台充当该内核中计算的日志，以及可以在内核中交互式检查和运行代码的位置：



# CHAPTER 12

## 终端 (Terminals)

JupyterLab 终端在Mac / Linux和Windows上的PowerShell上提供对系统shell (bash, tsch等) 的完全支持。您可以使用终端在系统shell中运行任何内容, 包括vim或emacs等程序。终端在运行Jupyter服务器的系统上运行, 具有用户的权限。因此, 如果JupyterLab安装在本地计算机上, 则JupyterLab终端将在那里运行。



要打开新终端, 请单击文件浏览器中的 + 按钮, 然后新的Launcher选项卡中选择终端:

关闭终端选项卡将使其在服务器上运行, 但您可以使用左侧栏中的运行选项卡重新打开它:

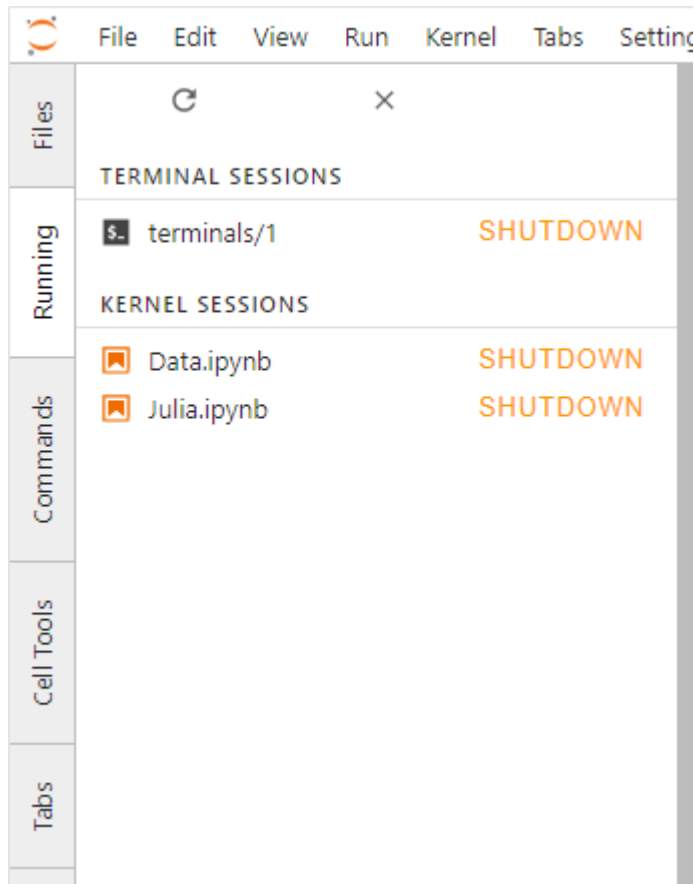




# CHAPTER 13

## 管理内核和终端

左侧栏中的“运行”面板显示当前在所有 notebooks，code consoles 和文件夹中运行的所有内核和终端的列表：



与经典的Jupyter Notebook一样，当您关闭笔记本文档，代码控制台或终端时，服务器上运行的底层内核或终端将继续运行。这使您可以执行长时间运行的操作并在以后返回。使用“运行”面板可以重新打开或聚焦链接到给定内核或终端的文档：

可以从“运行”面板关闭内核或终端:

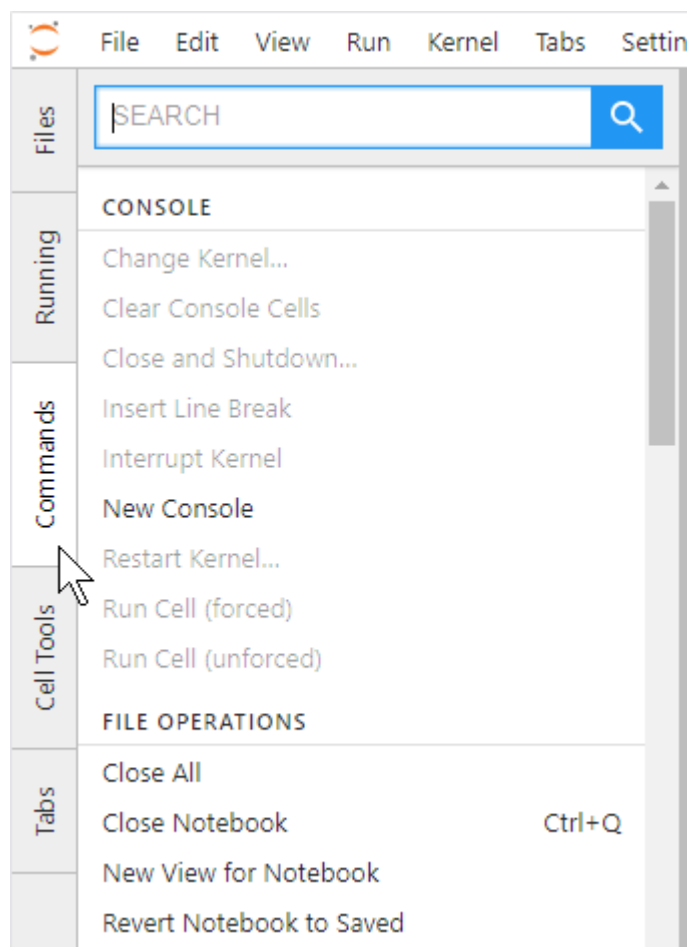
您可以通过单击 x 来关闭所有内核和终端按钮:

---

### 命令调用板 (Command Palette)

---

JupyterLab中的所有用户操作都通过集中式命令系统进行处理。这些命令在JupyterLab中共享和使用（菜单栏，上下文菜单，键盘快捷键等）。左侧栏中的命令选项板提供了一种键盘驱动的方式来搜索和运行JupyterLab命令：



可以使用键盘快捷键 Command/Ctrl Shift

c 访问命令调用板:

在Jupyter体系结构中，内核是由服务器启动的独立进程，它以不同的编程语言和环境运行代码。Jupyter-Lab使您可以将任何打开的文本文件连接到 [代码控制台和内核](#)。这意味着您可以交互式地从内核中的文本文件轻松运行代码。

右键单击文档，然后选择“**Create Console for Editor**”：

代码控制台打开后，发送一行代码或选择一段代码，然后通过按 `Shift Enter` 键将其发送到代码控制台：

在Markdown文档中，`Shift Enter` 将自动检测光标是否在代码块内，如果没有选择则运行整个块：

文本文件编辑器中的\*任何\*文本文件（Markdown，Python，R，LaTeX，C ++等）都可以这种方式连接到代码控制台和内核。



## CHAPTER 16

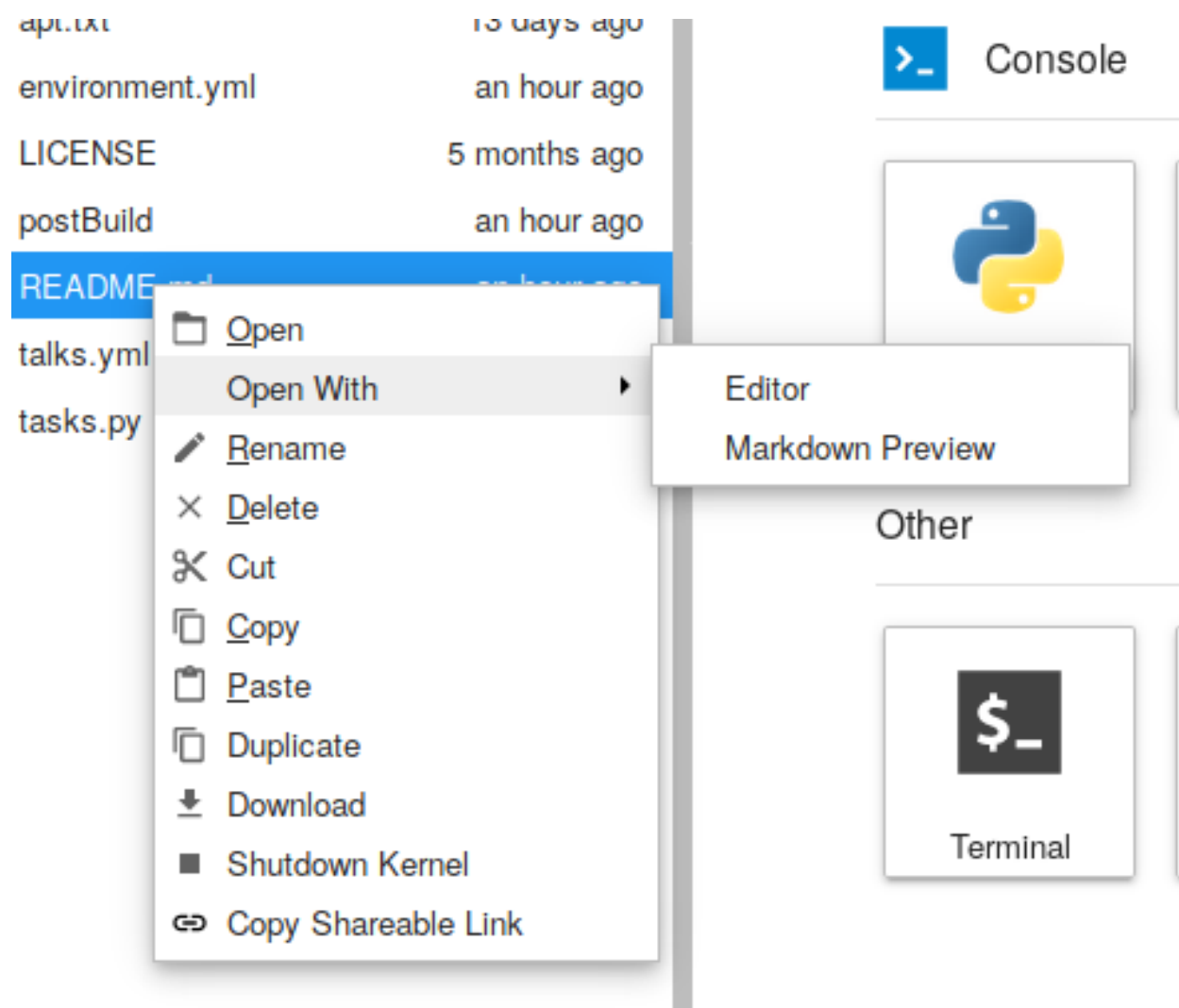
---

### 文件和输出格式

---

JupyterLab提供统一的体系结构，用于以各种格式查看和编辑数据。无论数据是在文件中还是由内核提供为笔记本或代码控制台中的富单元输出，此模型都适用。

对于文件，数据格式由文件的扩展名检测（如果没有扩展名，则检测整个文件名）。单个文件扩展名可能会注册多个编辑器或查看器。例如，**Markdown**文件（.md）可以在文件编辑器中编辑，也可以呈现为**HTML**格式。您可以通过右键单击文件浏览器中的文件名并使用“**Open With**”子菜单打开文件的不同编辑器和查看器：

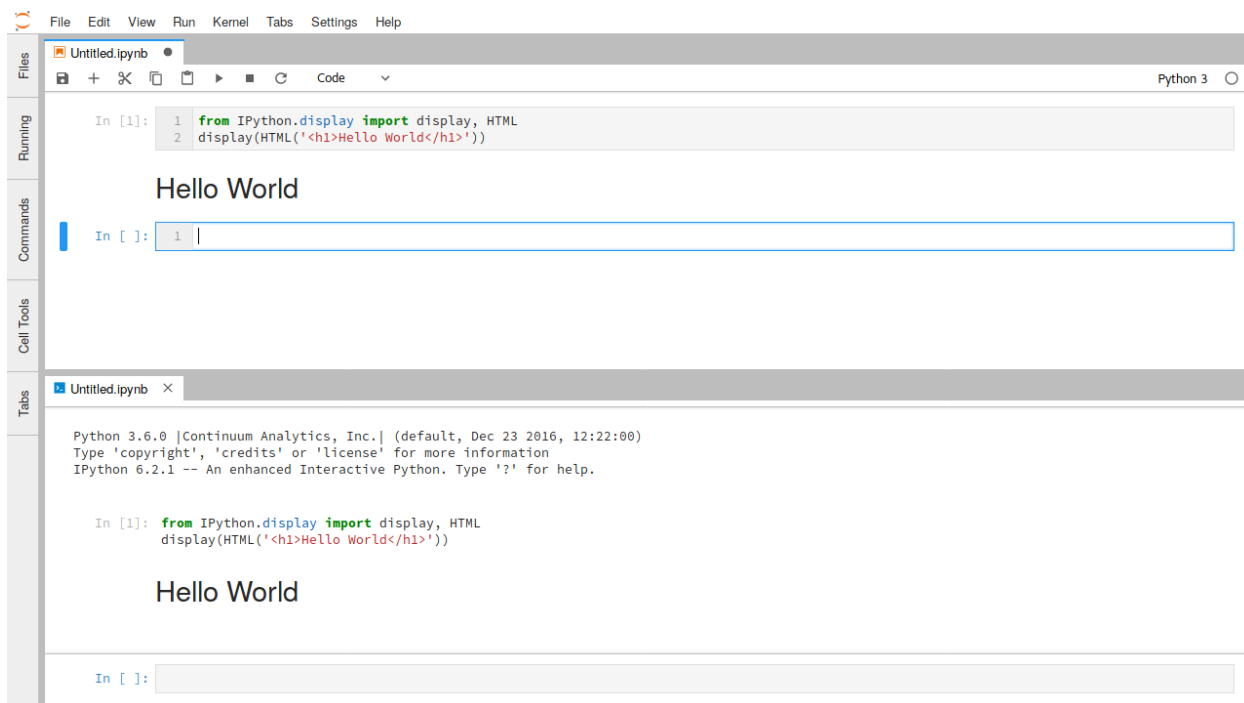


要在笔记本电脑或代码控制台中使用这些不同的数据格式作为输出，您可以使用您正在使用的内核的相关显示API。例如，IPython内核提供了各种便利类来显示丰富的输出：

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

运行此代码将在笔记本或代码控制台单元格的输出中显示HTML：





IPython显示功能还可以从密钥字典（MIME类型）和值（MIME数据）构造原始丰富输出消息：

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

其他Jupyter内核提供类似的API。

本节的其余部分将重点介绍JupyterLab默认支持的一些常见数据格式。JupyterLab扩展还可以添加对其他文件格式的支持。

## 16.1 Markdown

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

Markdown是一种简单而流行的标记语言，用于Jupyter Notebook中的文本编辑。

Markdown文档可以像文本文件一样编辑，并且可以渲染后呈现：

此模式支持的Markdown语法与Jupyter Notebook中使用的语法相同（例如，LaTeX方程式工作）。如动画中所示，对Markdown源的编辑会立即反映在渲染后的版本中。

## 16.2 图片

- File extension: .bmp, .gif, .jpeg, .jpg, .png, .svg
- MIME type: image/bmp, image/gif, image/jpeg, image/png, image/svg+xml

JupyterLab支持单元格输出图像数据以及上述格式的文件。在图像文件查看器中，您可以使用键盘快捷键（如“+”和“-”）来缩放图像，[“`”和]“`”旋转图像，使用“H”和“V”来水平和垂直翻转图像。使用“I”反转颜色，并使用“O”重置图像。

要将SVG图像编辑为文本文件，请在文件浏览器中右键单击SVG文件名，然后在“Open With”子菜单中选择“Editor”项：

## 16.3 分隔符分隔值

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

具有分隔符分隔行的文件（如CSV文件）是表格数据的常用格式。JupyterLab中这些文件的默认查看器是一个高性能数据网格查看器，它可以显示以逗号分隔，制表符分隔和分号分隔的值：

虽然可以通过网格查看器读取制表符分隔行文件，但它当前不会自动识别 `.tsv` 文件。要查看，您必须将扩展名更改为 `.csv` 并将分隔符设置为选项卡。

要将CSV文件编辑为文本文件，请在文件浏览器中右键单击该文件，然后在“Open With”子菜单中选择“Editor”项：

JupyterLab的网格查看器可以打开大文件，最大可达特定浏览器的字符串大小。下面的表格显示了我们在每个支持的浏览器中成功打开的最大测试文件的大小：

Browser	Max Size
Firefox	250MB
Chrome	730MB
Safari	1.8GB

可以成功加载的文件的实际最大大小将根据浏览器版本和文件内容而有所不同。

## 16.4 JSON

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

JavaScript Object Notation (JSON) 文件在数据科学的实践中很常见。JupyterLab支持在单元格输出中显示JSON数据或使用树视图查看JSON文件：

要将JSON编辑为文本文件，请在文件浏览器中右键单击文件名，然后在“Open With”子菜单中选择“Editor”项：

## 16.5 HTML

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

JupyterLab支持在单元格输出中呈现HTML，并在文件编辑器中将HTML文件编辑为文本。

## 16.6 LaTeX

- File extension: `.tex`

- MIME type: `text/latex`

JupyterLab支持在单元格输出LaTeX方程，并在文件编辑器中将LaTeX文件编辑为文本。

## 16.7 PDF

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

PDF是文档的通用标准文件格式。要在JupyterLab中查看PDF文件，请在文件浏览器中双击该文件：

## 16.8 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和Vega-Lite是声明性可视化语法，可以将可视化编码为JSON数据。有关更多信息，请参阅Vega或Vega-Lite的文档。JupyterLab支持在文件和单元格输出Vega 2.x和Vega-Lite 1.x数据。

可以通过双击文件浏览器中的文件来打开具有 `.vl` 或 `.vl.json` 文件扩展名的Vega-Lite 1.x文件：

也可以通过文件浏览器内容菜单中的“Open With...”子菜单在JSON查看器或文件编辑器中打开文件：

与JupyterLab中的其他文件一样，单个文件的多个视图保持同步，使您能够以交互方式编辑和渲染Vega / Vega-Lite可视化数据：

相同的工作流程也适用于Vega 2.x文件，文件扩展名为 `.vg` 或 `.vg.json`。

笔记本电脑或代码控制台中Vega / Vega-Lite的输出支持是通过第三方库提供的，例如Altair（Python），vegalite R软件包或Vegas（Scala / Spark）。



可在 [此处](#) 找到支持Vega 3.x和Vega-Lite 2.x的JupyterLab扩展。

## 16.9 Virtual DOM

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

诸如 `react.js` 之类的虚拟DOM库极大地改善了在HTML中呈现交互式内容的体验。`nteract`项目与Project Jupyter密切合作，为虚拟DOM数据创建了一个 **声明性JSON格式**。`JupyterLab`可以使用`react.js`呈现此数据。这适用于扩展名为 `.vdom` 的VDOM文件或笔记本输出中的VDOM文件。

以下是以交互方式编辑和呈现的 `.vdom` 文件的示例：

The `nteract/vdom` 库提供了一个Python API，用于创建在`nteract`和`JupyterLab`中呈现的VDOM输出：



The screenshot shows a JupyterLab window with a menu bar (File, Edit, View, Run, Kernel, Tabs, Settings, Help) and a sidebar with icons for Files, Running, Commands, Cell Tools, and Tabs. The main area displays a notebook cell with the following code:

```
In [3]: 1 from IPython.display import display
2 from vdom.helpers import h1, p, img, div, b
3
4 display(
5     div(
6         h1('Our Incredibly Declarative Example'),
7         p('Can you believe we wrote this ', b('in Python'), '?'),
8         img(src="https://media.giphy.com/media/xUPGcuWZHRC2HyBRS/giphy.gif"),
9         p('What will ', b('you'), ' create next?'),
10     )
11 )
```

The rendered output of the cell shows the text "Our Incredibly Declarative Example" in a large, bold, black font. Below it, the text "Can you believe we wrote this in Python?" is displayed, where "in Python" is in a smaller, bold, black font. To the right of the text is a small, colorful illustration of two robots. One robot is green and yellow, and the other is red and yellow. They are standing on a light blue surface against a light blue background.



从根本上说，JupyterLab的设计是可扩展的。JupyterLab扩展可以自定义或增强JupyterLab的任何部分。他们可以为 Notebook 中的丰富输出提供新的皮肤，文件查看器和编辑器或渲染器。扩展可以将项添加到菜单或命令选项板，键盘快捷键或设置系统中的设置。扩展可以为其他扩展提供API，并且可以依赖于其他扩展。事实上，整个JupyterLab 本身只是一个扩展的集合，它们不比任何自定义扩展有更多的强大。

JupyterLab 采用npm包（Javascript开发中的标准包格式）进行扩展。在GitHub上构建了许多社区开发的扩展。您可以搜索GitHub主题 [jupyterlab-extension](#) 来查找扩展名。有关开发扩展的信息，请参阅 [开发人员文档](#)。

**注解：**如果您是JupyterLab 扩展开发人员，请注意扩展开发人员API不稳定，并且会不断更新。

要安装JupyterLab扩展，您需要安装Node.js版本4或更高版本。如果您使用 conda，您可以使用：

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

如果你的操作系统是 MacOS，可以通过 [Homebrew](#) 安装：

```
brew install node
```

您也可以“Node.js website <<https://nodejs.org/>>”的官网下载 Node.js，并且安装。

## 17.1 安装扩展

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:

JupyterLab application 包含一组核心扩展，提供本用户指南（notebook，终端，文本编辑器等）中描述的功能。您可以使用以下命令在应用程序中安装新扩展：

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

其中“my-extension”是 [npm](#) 上有效的JupyterLab扩展名npm包的名称。使用my-extension @ version语法安装特定版本的扩展，例如：

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

您还可以安装未上传到npm的扩展，即“my-extension”可以是包含扩展名，gzip压缩包或gzip压缩包的URL的本地目录。

我们鼓励扩展作者将“jupyterlab-extension”添加到任何具有JupyterLab扩展名的GitHub存储库，以便于发现。您可以通过在GitHub中搜索“jupyterlab-extension <<https://github.com/search?utf8=%E2%9C%93&q=topic%3Ajupyterlab-extension&type=Repositories>>”主题来查看扩展列表。

您可以通过运行以下命令列出当前安装的扩展：

```
jupyter labextension list
```

通过运行以下命令卸载扩展：

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

其中 my-extension 是扩展名，打印在扩展名列表中。您也可以使用此命令卸载核心扩展（您可以随时重新安装核心扩展）。

安装和卸载扩展可能需要一些时间，因为它们已下载，与核心扩展捆绑在一起，整个应用程序将 *Rebuild* 。您可以通过在 install 命令后列出其名称来在同一命令中安装/卸载多个扩展。

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:

如果要在几个阶段中安装/卸载多个扩展，则可能需要在安装/卸载步骤中包含标志“--no-build”来推迟重建应用程序。准备好Rebuild后，可以运行以下命令：

```
jupyter lab build
```

## 17.2 禁用扩展程序

您可以通过运行以下命令禁用特定的JupyterLab扩展（包括核心扩展），而无需Rebuild应用程序：

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

这将阻止扩展程序在浏览器中加载，但不需要Rebuild。

您可以使用以下命令重新启用扩展：

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

## 17.3 高级用法

JupyterLab持久存储的任何信息都存储在其应用程序目录中，包括设置和构建资产。这与安装Python包的位置（例如在“site\_packages”中）是分开的，因此安装目录是不可变的。

可以在任何JupyterLab命令中使用 `--app-dir` 覆盖应用程序目录，也可以通过设置JUPYTERLAB\_DIR环境变量来覆盖应用程序目录。如果未指定，则默认为“<sys-prefix>/share/jupyter/lab”，其中<sys-prefix>是当



前Python环境的特定于站点的目录前缀。您可以通过运行 `jupyter lab path` 来查询当前的应用程序路径。

### 17.3.1 JupyterLab Build 过程

要 **rebuild app** 目录，请运行“`jupyter lab build`”。默认情况下，`jupyter labextension install` 命令构建应用程序，因此您通常不需要直接调用“`build`”。

Build 过程包括：

- 使用模板文件填充“`staging`”目录
- 处理任何本地安装的包
- 确保所有已安装的资源（`assets`）都可用
- 打包资源（`assets`）
- 将打包的资源（`assets`）复制到“`static`”目录

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

请注意，构建将始终使用最新的JavaScript包，以满足JupyterLab本身和任何已安装扩展的依赖性要求。如果您希望使用Python包附带的一组固定需求运行JupyterLab，您可以启动为“`jupyter lab --core-mode`”。

### 17.3.2 JupyterLab app 目录

JupyterLab应用程序目录包含子目录“`extensions`”，`schemas`，`settings`，`staging`，`static`，and `themes`。

#### extensions

`extensions` 目录包含应用程序的每个已安装扩展的打包`tarball`。如果应用程序目录与“`sys-prefix`”目录不同，则“`sys-prefix`”目录中安装的扩展将在`app`目录中使用。如果在“`sys-prefix`”目录中存在的`app`目录中安装了扩展，则它将隐藏“`sys-prefix`”版本。卸载扩展将首先卸载带阴影的扩展，然后再次调用时尝试卸载“`sys-prefix`”版本。如果无法卸载“`sys-prefix`”版本，则仍可使用设置中的“`ignoredPackages`”元数据忽略其插件。

#### schemas

`schemas` 目录包含描述各个扩展使用的设置的 **JSON Schemas**。用户可以使用JupyterLab设置编辑器编辑这些设置。

#### settings

`settings`“”目录包含“`page_config.json`”和“`build_config.json`”文件。

`page_config.json`

`page_config.json` 数据用于向应用程序环境提供配置数据。

`page_config.json` 文件中的两个重要字段可以控制加载哪些插件：

1. `disabledExtensions` 表示根本不应加载的扩展名。

2. 对于在某些内容需要之前不加载的扩展的“`deferredExtensions`”，无论它们是否将“`autostart`”设置为“`true`”。

每个字段的值是一个字符串数组。对“`disabledExtensions`”和“`deferredExtensions`”中的模式执行以下检查序列。

- 如果在配置值和包名称之间发生相同的字符串匹配（例如，“`@jupyterlab/apputils-extension`”），则整个包被禁用（或延迟）。
- 如果字符串值被编译为正则表达式并且对包名称测试为正（例如，“`disabledExtensions`”: [`"@jupyterlab/apputils*$"`]），则整个包被禁用（或延迟）。
- 如果在配置值和包中的单个插件ID之间发生相同的字符串匹配（例如，“`disabledExtensions`”: [`"@jupyterlab/apputils-extension: settings"`]），则该特定插件被禁用（或延迟）。
- 如果将字符串值编译为正则表达式并对包中的单个插件ID进行正向测试（例如，“`disabledExtensions`”: [`"^@jupyterlab/apputils-extension:set.*$"`]），则该特定插件被禁用（或延期）。

#### build\_config.json

`build_config.json` 文件用于跟踪使用 `jupyter labextension install <directory>` 安装的本地目录，以及已明确卸载的核心扩展。`build_config.json` 文件的示例是：

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

#### staging and static

`static` 目录包含将由JupyterLab应用程序加载的assets。`staging` 目录用于创建构建，然后填充“`staging`”目录。

运行“`jupyter lab`”将尝试在应用程序目录中运行“`static assets`”（如果存在）。您可以运行“`jupyter lab --core-mode`”来加载核心JupyterLab应用程序（即没有任何扩展的应用程序）。

#### themes

`themes` 目录包含JupyterLab Themes 扩展的资源（例如CSS和图标）。

## JupyterLab 执行在 JupyterHub 里

JupyterLab与JupyterHub一起开箱即用，甚至可以与经典Notebook并排运行。

### 18.1 默认情况下使用JupyterLab

如果在运行JupyterHub的系统上安装JupyterLab，它将立即在 `/lab` URL上可用，但默认情况下用户仍将被定向到经典Notebook (`/tree`)。要将用户的默认用户界面更改为JupyterLab，请在`jupyterhub_config.py`文件中设置以下配置选项：`jupyterhub_config.py` file:

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

在此配置中，用户仍然可以在“`/tree`”访问经典Notebook，方法是在浏览器中键入该URL，或者使用JupyterLab的“帮助”菜单中的“Launch Classic Notebook”项。

### 18.2 进一步整合

JupyterLab和JupyterHub之间的额外集成由JupyterLab的 `jupyterlab-hub` 扩展提供。它提供了一个Hub菜单，其中包含访问JupyterHub控制面板或注销集线器的项目。

安装 `jupyterlab-hub` extension, 执行:

```
jupyter labextension install @jupyterlab/hub-extension
```

`jupyterlab-hub` GitHub repository 存储库提供了进一步的说明。

### 18.3 示例配置

有关将JupyterLab与JupyterHub一起使用的完整配置示例，请参阅‘`jupyterhub-deploy-teaching`’ <<https://github.com/jupyterhub/jupyterhub-deploy-teaching>> 存储库。JupyterLab开发指南是开发 JupyterLab 扩展 (extensions) 或

开发 Jupyterlab 自身。

jupyterlab/jupyterlab 包含两个 `package` 的声明:

- 在 `repo` 的根目录中的 “`package.json`” 文件指示的是 `npm` 包
- 在 `repo` 的根目录中的 “`setup.py`” 文件指示的是 `Python` 包

`npm`包和`Python`包都命名为 `jupyterlab`.

请查看 [Contributing Guidelines](#) 开发安装说明

## 19.1 项目文件夹 (Directories) 说明

### 19.1.1 NPM package: `src/`, `lib/`, `typings/`, `buildutils/`

- `src/`: `Typescript` 源文件.
  - `jlpm run build` 源文件, 将`build`后的文件放入 `lib/` 文件夹里.
  - `jlpm run clean` 删除 `lib/` 里的文件.
- `typings/`: 外部`Typescript` 的 `library` 定义的类型 `needs`.
- `buildutils/`: 管理 `repo` 的 `build` 过程

### 19.1.2 实例: `examples/`

`examples/` 目录包含组件的独立示例, 例如页面上的简单笔记本, 控制台, 终端和文件浏览器。 `lab` 示例说明了 `JupyterLab`中使用的组件的简化组合。 此示例显示多个独立组件组合在一起以创建更复杂的应用程序。

### 19.1.3 测试: `test/`

测试存储并在 `test/` 目录中运行。源文件位于 `test/src/` 中。

### 19.1.4 Notebook extension: `jupyterlab/`

`jupyterlab/` 目录包含Jupyter服务器扩展。

服务器扩展包括一个私有的npm包，以便构建扩展服务的 **webpack bundle**。私有npm包依赖于 `repo` 根目录中的 `jupyterlab` npm包。

### 19.1.5 Git hooks: `git-hooks/`

`git-hooks/` 目录存储了一些方便的git hooks，每次你在git repo中签出或合并（通过pull请求或直接push到master）时，它会自动重建npm包和服务器扩展。

### 19.1.6 Documentation: `docs/`

在构建文档（`jlpm run docs`）之后，`docs / index.html` 就是文档的入口点。

**警告:** The extension developer API is not stable and will evolve in JupyterLab releases in the near future.

JupyterLab can be extended in four ways via:

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

See [Let's Make an xkcd JupyterLab Extension](#) to learn how to make a simple JupyterLab extension.

To understand how to wrap an **Angular** application as a JupyterLab extension, see the “[Create your own Angular JupyterLab extension](#)” guide provided by [Scripted Forms](#).

A JupyterLab application is comprised of:

- A core Application object
- Plugins

## 20.1 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 JupyterLab plugin extensions in `CommonJS` and `TypeScript`.

The default plugins in the JupyterLab application include:

- `Terminal` - Adds the ability to create command prompt terminals.
- `Shortcuts` - Sets the default set of shortcuts for the application.
- `Images` - Adds a widget factory for displaying image files.
- `Help` - Adds a side bar widget for displaying external documentation.
- `File Browser` - Creates the file browser and the document manager and the file browser to the side bar.
- `Editor` - Add a widget factory for displaying editable source files.
- `Console` - Adds the ability to launch Jupyter Console instances for interactive kernel console sessions.

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

## 20.2 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.

## 20.3 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 work area for user activity.
- A bottom area for things like status bars

## 20.4 Phosphor

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



## 20.5 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. This means that if your extension is in TypeScript you'll have to run a `jlpm run build` before the changes will be reflected in JupyterLab. To avoid this step you can also watch the TypeScript sources in your extension which is usually assigned to the `tsc -w` shortcut.

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

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

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

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

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

This will remove the package metadata from the source tree, but will **not** remove any files added by the `addsibling` script, which should be removed manually.

The package should export ECMAScript 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 ECMAScript 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`).

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

## 20.6 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.

## 20.7 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 `jupyterlab 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 in the same way as a regular extension (see [extension authoring](#)).

## 20.8 Standard (General-Purpose) Extensions

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

### 20.8.1 Core Tokens

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

- `@jupyterlab/application:ILayoutRestorer`: An interface to the application layout restoration functionality. Use this to have your activities restored across page loads.
- `@jupyterlab/application:IMimeDocumentTracker`: An instance tracker for documents rendered using a mime renderer extension. Use this if you want to list and interact with documents rendered by such extensions.
- `@jupyterlab/application:IRouter`: The URL router used by the application. Use this to add custom URL-routing for your extension (e.g., to invoke a command if the user navigates to a sub-path).
- `@jupyterlab/apputils:ICommandPalette`: An interface to the application command palette in the left panel. Use this to add commands to the palette.
- `@jupyterlab/apputils:ISplashScreen`: An interface to the splash screen for the application. Use this if you want to show the splash screen for your own purposes.
- `@jupyterlab/apputils:IThemeManager`: An interface to the theme manager for the application. Most extensions will not need to use this, as they can register a [theme extension](#).
- `@jupyterlab/apputils:IWindowResolver`: An interface to a window resolver for the application. JupyterLab workspaces are given a name, which are determined using the window resolver. Require this if you want to use the name of the current workspace.
- `@jupyterlab/codeeditor:IEditorServices`: An interface to the text editor provider for the application. Use this to create new text editors and host them in your UI elements.
- `@jupyterlab/completer:ICompletionManager`: An interface to the completion manager for the application. Use this to allow your extension to invoke a completer.
- `@jupyterlab/console:IConsoleTracker`: An instance tracker for code consoles. Use this if you want to be able to iterate over and interact with code consoles created by the application.
- `@jupyterlab/console:IContentFactory`: A factory object that creates new code consoles. Use this if you want to create and host code consoles in your own UI elements.
- `@jupyterlab/coreutils:ISettingRegistry`: An interface to the JupyterLab settings system. Use this if you want to store settings for your application. See [extension settings](#) for more information.
- `@jupyterlab/coreutils:IStateDB`: An interface to the JupyterLab state database. Use this if you want to store data that will persist across page loads. See [state database](#) for more information.
- `@jupyterlab/docmanager:IDocumentManager`: An interface to the manager for all documents used by the application. Use this if you want to open and close documents, create and delete files, and otherwise interact with the file system.
- `@jupyterlab/filebrowser:IFileBrowserFactory`: A factory object that creates file browsers. Use this if you want to create your own file browser (e.g., for a custom storage backend), or to interact with other file browsers that have been created by extensions.

- `@jupyterlab/fileeditor:IEditorTracker`: An instance tracker for file editors. Use this if you want to be able to iterate over and interact with file editors created by the application.
- `@jupyterlab/imageviewer:IImageTracker`: An instance tracker for images. Use this if you want to be able to iterate over and interact with images viewed by the application.
- `@jupyterlab/inspector:IInspector`: An interface for adding variable inspectors to widgets. Use this to add the ability to hook into the variable inspector to your extension.
- `@jupyterlab/launcher:ILauncher`: An interface to the application activity launcher. Use this to add your extension activities to the launcher panel.
- `@jupyterlab/mainmenu:IMainMenu`: An interface to the main menu bar for the application. Use this if you want to add your own menu items.
- `@jupyterlab/notebook:ICellTools`: An interface to the `Cell Tools` panel in the application left area. Use this to add your own functionality to the panel.
- `@jupyterlab/notebook:IContentFactory`: A factory object that creates new notebooks. Use this if you want to create and host notebooks in your own UI elements.
- `@jupyterlab/notebook:INotebookTracker`: An instance tracker for code consoles. Use this if you want to be able to iterate over and interact with notebooks created by the application.
- `@jupyterlab/rendermime:IRenderMimeRegistry`: An interface to the rendermime registry for the application. Use this to create renderers for various mime-types in your extension. Most extensions will not need to use this, as they can register a *[mime renderer extension](#)*.
- `@jupyterlab/rendermime:ILatexTypesetter`: An interface to the LaTeX typesetter for the application. Use this if you want to typeset math in your extension.
- `@jupyterlab/settingeditor:ISettingEditorTracker`: An instance tracker for setting editors. Use this if you want to be able to iterate over and interact with setting editors created by the application.
- `@jupyterlab/terminal:ITerminalTracker`: An instance tracker for terminals. Use this if you want to be able to iterate over and interact with terminals created by the application.
- `@jupyterlab/tooltip:ITooltipManager`: An interface to the tooltip manager for the application. Use this to allow your extension to invoke a tooltip.

## 20.8.2 Standard Extension Example

For a concrete example of a standard extension, see [How to Extend the Notebook Plugin](#). Notice that the mime renderer 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.

## 20.8.3 Storing Extension Data

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

### Extension Settings

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

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

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

See the [fileeditor-extension](#) for another example of an extension that uses settings.

## 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
};
```

## 20.8.4 Context Menus

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

```
// In `onAfterAttach()`  
this.node.addEventListener('contextmenu', this);  
  
// In `handleEvent()`  
case 'contextmenu':  
    event.stopPropagation();
```

JupyterLab can be extended in several ways:

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

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





---

### Overview of document architecture

---

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

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

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

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

The `Document Manager` uses the Document Registry to create models and widgets for documents. The Document Manager is only meant to be accessed by the File Browser itself.

## 22.1 Document Registry

*Document widget extensions* in the JupyterLab application can register:

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

### 22.1.1 Widget Factories

Create a widget for a given file.

*Example*

- The notebook widget factory that creates NotebookPanel widgets.

### 22.1.2 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).

### 22.1.3 Widget Extension Factories

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

*Examples*

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

### 22.1.4 File Types

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

### 22.1.5 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”).

### 22.1.6 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.

### 22.1.7 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.

## 22.2 Document Manager

The *Document Manager* handles:

- document models
- document contexts

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



## 23.1 Background

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

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

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

## 23.2 Structure of the Notebook plugin

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

### 23.2.1 Model

The `NotebookModel` contains an observable list of cells.

A `cell model` can be:

- a code cell
- a markdown cell
- raw cell

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

### Cell operations

The NotebookModel cell list supports single-step operations such as moving, adding, or deleting cells. Compound cell list operations, such as undo/redo, are also supported by the NotebookModel. Right now, undo/redo is only supported

on cells and is not supported on notebook attributes, such as notebook metadata. Currently, undo/redo for individual cell input content is supported by the CodeMirror editor's undo feature. (Note: CodeMirror editor's undo does not cover cell metadata changes.)

## Metadata

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

### 23.2.2 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 an `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.

## 23.3 How to extend the Notebook plugin

We'll walk through two notebook extensions:

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

### 23.3.1 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
};

/**
```

(continues on next page)

(续上页)

```

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

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

/**
 * Activate the extension.
 */
function activate(app: 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.

### 23.3.2 The *ipywidgets* third party extension

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

The *ipywidgets* extension registers a factory for a notebook *widget* extension using the [Document Registry](#). The



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

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

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



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

#### 24.1 TypeScript

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

#### 24.2 Initialization Options

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

#### 24.3 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.

#### 24.4 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.

## 24.5 Private Module Namespace

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

## 24.6 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.

## 24.7 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.

## 24.8 Signals

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

## 24.9 Models

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

## 24.10 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.

## 24.11 Data Structures

For public API, we have three options: JavaScript `Array`, `Iterator`, 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 `Iterator` 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.

## 24.12 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](#).

## 24.13 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.

## 24.14 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.

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

### 25.1 CSS checklist

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

### 25.2 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.

#### 25.2.1 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`.

### 25.2.2 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`.

### 25.2.3 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.

## 25.3 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 as `import './style/index.css';`.

## 25.4 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 enables the styling of these children in a manner that is independent of the children implementation or CSS classes they have themselves.

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

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

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

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

## 25.5 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 structure is highly recursive, the usual descendant selectors may not be specific to target only the desired children.

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

---

## Writing Documentation

---

This section provides information about writing documentation for JupyterLab. See our [Contributor Guide](#) for details on installation and testing.

### 26.1 Writing Style

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

### 26.2 User Interface Naming Conventions

#### 26.2.1 Documents, Files, and Activities

Files are referred 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.

### 26.2.2 Element Names

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

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

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

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

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

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

See *JupyterLab* 接口 (API) for descriptions of elements in the UI.

## 26.3 Keyboard Shortcuts

Typeset keyboard shortcuts as follows:

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

## 26.4 Screenshots and Animations

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

- Set screen resolution to non-hidpi (non-retina)
- Set browser viewport to 1280x720 px. The Firefox Web Developer extension and Chrome Developer Tools offer device specific rendering that enables you to set this viewport resolution.
- Capture the viewport, **not the full browser window**, using the capture software of your choice. **Do not include any of the desktop background.**
- For PNGs, reduce their size using pngquant. For movies, upload them to the IPython/Jupyter YouTube channel and embed them in the docs with an iframe. The pngs can live in the main repository. The movies should be added to the `jupyterlab-media` repository.
- Use [www.youtube-nocookie.com](http://www.youtube-nocookie.com) website, which can be found by clicking on the 'privacy-enhanced' embedding option in the Share dialog on YouTube. Add the following parameters the end of the URL `?rel=0&showinfo=0`. This disables the video title and related video suggestions.
- Screenshots or animations should be preceded by a sentence describing the content, such as "To open a file, double-click on its name in the File Browser:".
- We have custom CSS that will add box shadows, and proper sizing of screenshots and embedded YouTube videos. See examples in the documentation for how to embed these assets.

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

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

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



---

### 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` hierarchy.
- Lifecycle events (`onBeforeDetach`, `onAfterAttach`, etc.).
- Both CSS-based and absolutely positioned layouts.

In situations where these features 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.





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 [path-posix](#). 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.



---

### 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:

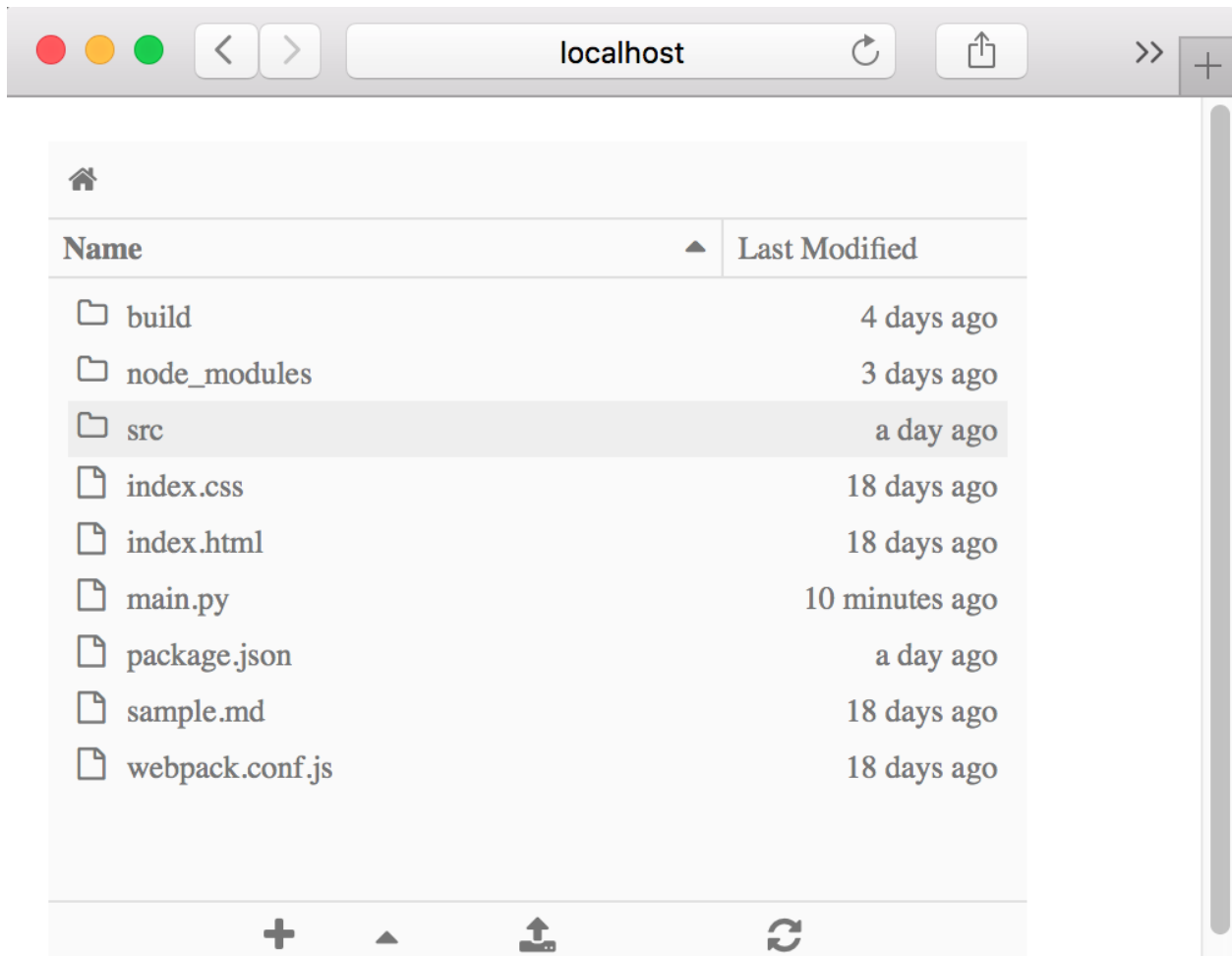
```
jlpm run build:examples
```

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

```
python main.py
```

### 29.1 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:



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

### 29.1.1 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.



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.

### 30.1 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.





---

## Let's Make an xkcd JupyterLab Extension

---

**警告:** The extension developer API is not stable and will evolve in JupyterLab releases in the near future.

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



Sound like fun? Excellent. Here we go!

## 31.1 Setup a development environment

### 31.1.1 Install conda using miniconda

Start by installing miniconda, following [Conda's installation documentation](#).

### 31.1.2 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.

```
conda activate jupyterlab-ext
```

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

## 31.2 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.

## 31.3 Create an extension project

### 31.3.1 Initialize the project from a cookiecutter

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

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

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

```
author_name [: Your Name
extension_name [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]: https://github.com/my_
↪name/jupyterlab_xkcd
```

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
```

### 31.3.2 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.

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

---

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

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

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

### 31.3.3 See the initial extension in action

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

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

- [Accessing the DevTools in Google Chrome](#)
- [Opening the Web Console in Firefox](#)

After you reload the page with the console open, you should see a message that says `JupyterLab extension jupyterlab_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.

### 31.3.4 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](https://github.com/jupyterlab/jupyterlab_xkcd).

## 31.4 Add an xkcd widget

### 31.4.1 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';
```

You will also need to install this dependency. Run the following command in the repository root folder install the dependency and save it to your *package.json*:

```
jlpm add @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.

```
jlpm run build
```

JupyterLab will rebuild after the extension does. You can see it's progress in the `jupyter lab --watch` window. After that finishes, return to the browser tab that opened when you started JupyterLab. Refresh it and look in the console. You should see the same activation message as before, plus the new message about the `ICommandPalette` instance you just added. If you don't, check the output of the build command for errors and correct your code.

```
JupyterLab extension jupyterlab_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';
```

Install this dependency as well:

```
jlpm add @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!');

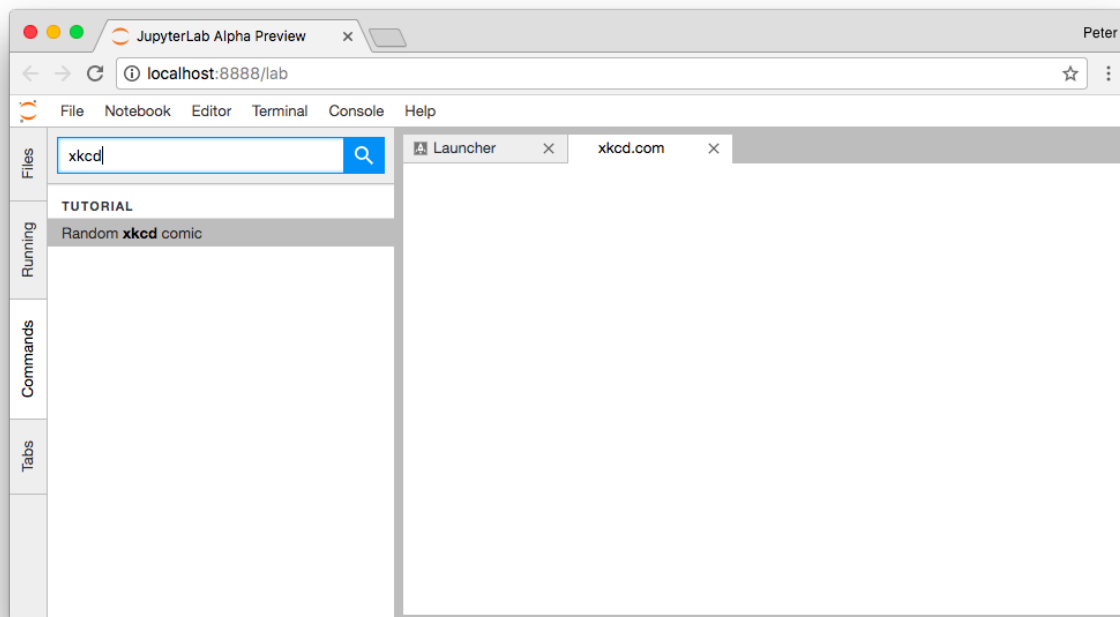
  // Create a single widget
  let widget: Widget = new Widget();
  widget.id = 'xkcd-jupyterlab';
  widget.title.label = 'xkcd.com';
  widget.title.closable = true;

  // 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 work 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 command executes, it attaches the widget to the main display area if it is not already present and then makes it the active tab. The last new line of code 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 by clicking on *Commands* and type *xkcd* in the search box. 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 command on panel'
```

### 31.4.2 Show a comic in the panel

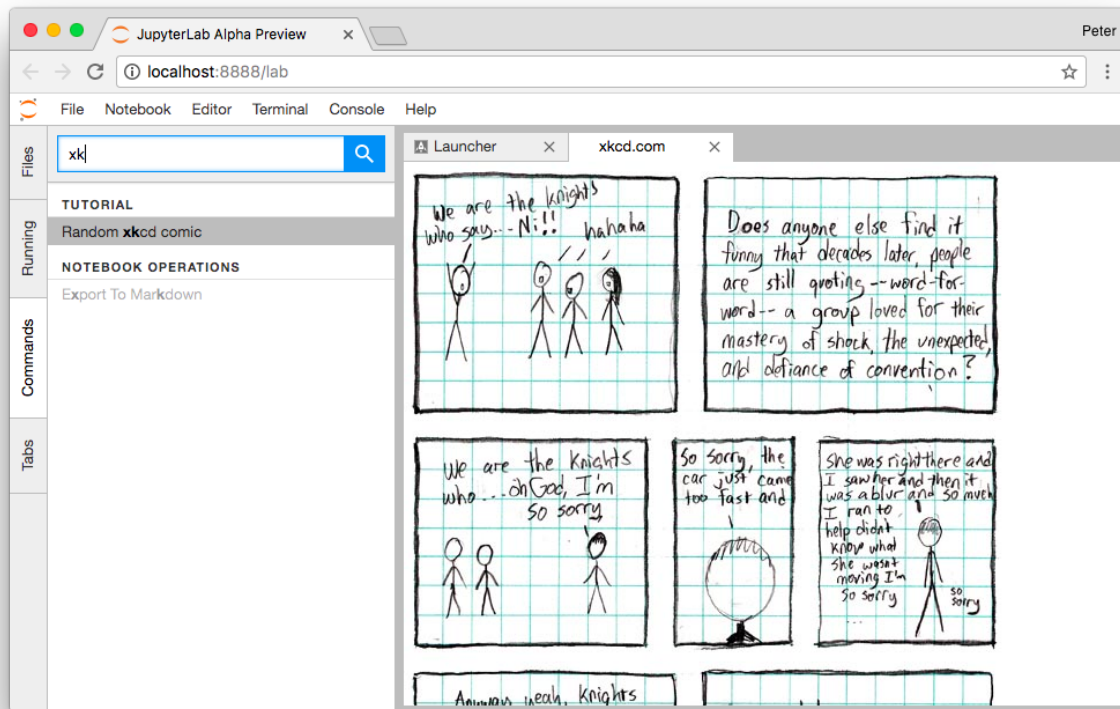
You've got an empty panel. It's time to add a comic to it. Go back to your code editor. Add the following code below the lines that create a `Widget` instance and above the lines that define the command.

```
// 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.



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'
```

## 31.5 Improve the widget behavior

### 31.5.1 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;
}
```

(continues on next page)



(续上页)

```
.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 `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">
        
      </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.



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'
```

### 31.5.2 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';
```

Install this dependency:

```
jupyterlab add @phosphor/messaging
```

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

```
/**
 * An xkcd 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">
          
        </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 xkcd 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 work 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'

```

### 31.5.3 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';
```

Install this dependency:

```
jupyterlab add @phosphor/coreutils
```

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: () => {
```

(continues on next page)

(续上页)

```

    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 work 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?

## 31.6 Publish your extension to npmjs.org

npm is both a JavaScript package manager and the de facto registry for JavaScript software. You can [sign up for an account on the npmjs.com site](#) or create an account from the command line by running `npm adduser` and entering values when prompted. Create an account now if you do not already have one. If you already have an account, login by running `npm login` and answering the prompts.

Next, open the project `package.json` file in your text editor. Prefix the name field value

with `@your-npm-username>/` so that the entire field reads `"name": "@your-npm-username/jupyterlab_xkcd"` where you've replaced the string `your-npm-username` with your real username. Review the homepage, repository, license, and [other supported package.json](#) fields while you have the file open. Then open the `README.md` file and adjust the command in the *Installation* section so that it includes the full, username-prefixed package name you just included in the `package.json` file. For example:

```
jupyter labextension install @your-npm-username/jupyterlab_xkcd
```

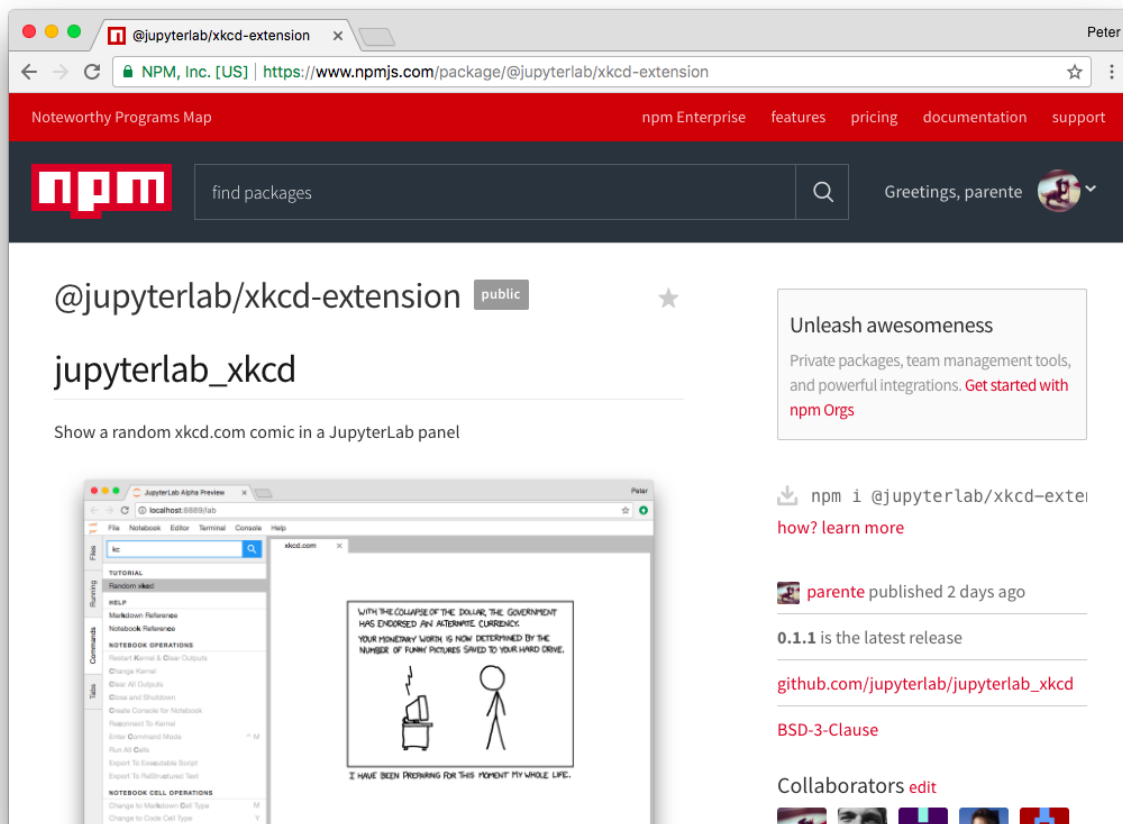
Return to your terminal window and make one more git commit:

```
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](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.



You can now try installing your extension as a user would. Open a new terminal and run the following commands, again substituting your npm username where appropriate (make sure to stop the existing `jupyter lab --watch`

command first):

```
conda create -n jupyterlab-xkcd jupyterlab nodejs
conda activate jupyterlab-xkcd
jupyter labextension install @your-npm-username/jupyterlab_xkcd
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.

## 31.7 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 32

---

### Indices and Tables

---

- `genindex`
- `modindex`
- `search`