JupyterLab Tabular Data Editor Documentation

Release stable

Mar 24, 2021

Getting Started

Manipulate your tabular data responsively and effectively within JupyterLab. Try it on Binder.

CHAPTER 1

Overview

Manipulate your tabular data responsively and effectively within JupyterLab.

Data is an integral part of many JupyterLab workflows, but a native data editing tool is non-existent. The JupyterLab Tabular Data Editor provides an interface to edit your data files side-by-side with Jupyter notebooks.

With this extension, you can create csv files from scratch, navigate through millions of cells smoothly, and manage your data all within JupyterLab.

This extension streamlines the editing process with a refined suite of commands that help you prepare your data for further work in machine learning, scientific computing, or other data-driven tasks.

CHAPTER 2

Installation

2.1 Requirements

JupyterLab >= 2.0

2.2 Install Nodejs

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

2.3 Install

```
jupyter labextension install jupyterlab-tabular-data-editor
```

2.4 Contributing

2.4.1 Install

The `jlpm` command is JupyterLab's pinned version of yarn that is installed with JupyterLab. You may use `yarn` or `npm` in lieu of `jlpm` below.

```
# Clone the repo to your local environment
# Move to jupyterlab-tabular-data-editor directory

# Install dependencies
jlpm
# Build Typescript source
```

(continues on next page)

(continued from previous page)

```
jlpm build
# Link your development version of the extension with JupyterLab
jupyter labextension install .
# Rebuild Typescript source after making changes
jlpm build
# Rebuild JupyterLab after making any changes
jupyter lab build
```

You can watch the source directory and run JupyterLab in watch mode to watch for changes in the extension's source and automatically rebuild the extension and application.

```
# Watch the source directory in another terminal tab
jlpm watch
# Run jupyterlab in watch mode in one terminal tab
jupyter lab --watch
```

Now every change will be built locally and bundled into JupyterLab. Be sure to refresh your browser page after saving file changes to reload the extension (note: you'll need to wait for webpack to finish, which can take 10s+ at times).

2.4.2 Uninstall

```
jupyter labextension uninstall jupyterlab-tabular-data-editor
```

CHAPTER 3

Changelog

3.1 v1.0.0

- Extension is now compatible with Jupyterlab 3.0

3.2 v0.7.5

- Bug fixes with moving shadow/line

3.3 v0.7.4

- Backspace keyboard shortcut working
- Fixes small bug with data types not updating on an undo/redo that changes the type
- Save dialog bug fix
- Created a selectCell method in the PaintedGrid
- Draw icon refactor
- Removed serializer and old model files

3.4 v0.7.3

- Datagrid styling changes
- Adjusts the position and style of the icons

3.5 v0.7.2

- Fixed the move line not accounting for scroll
- Package updated from @tde-csvviewer to @jupyterlab/csvviewer + Launcher handled in a way that we don't need to change _computeRowOffsets
- Fixed right-click column header results in move shadow

3.6 v0.7.1

- Added new files to the demo folder
- Ghost row/columns bug fixes
- Refactor data detection to format data

3.7 v0.7.0

- Can now edit headers after scrolling
- Hover feature for ghost row and column
- Clearing rows and columns bug fix
- Pointer cursor for ghost row/column
- Modified icon painting setup to work with absolute positioning rather than relative positioning
- Adding data detection icons
- Styling for data detection icons
- Replace all bug fix
- Makes the text "Column 1" appear on the column header when launching a csv file

3.8 v0.6.0

- Cell data types for the body region
- Multi insert/remove for rows/columns
- WCAG AAA approved search match colors
- Ghost row and column feature added
- Fix the header displaying the wrong value on edit
- Serialization fix for data sets larger than 500 rows
- Inserting/removing column bug fixes
- Fixed console error when searching for a match

3.9 v0.5.0

- Shadow/line fixes when moving + handler.ts refactoring
- Create a new csv file from launcher
- Reduced column header and row height
- Edit Headers
- Save keybinding

3.10 v0.4.0

- Selection UX for Undo/Redo
- Right-click selection fixes
- Litestore refactor

3.11 v0.3.0

- Multiple context menus
- Clear contents (rows, columns, selections)

3.12 v0.2.0

- Copy, cut, and paste
- Undo and redo
- Implemented Litestore
- Move columns and rows
- Theme manager (light/dark)
- Search and replace
- Command Toolbar
- Binder link setup

3.13 v0.1.0

- Editable cells
- Alphabetic column header
- Save CSV file
- Delete rows and columns
- Add rows and columns

CHAPTER 4

User Experience

The JupyterLab Tabular Data Editor provides a versatile interface to support your data editing process.

4.1 Toolbar

The toolbar has the following functionalities: save, undo, redo, cut, copy, and paste. In addition, you can format your data based on data types by toggling on *Format Data*.

4.2 Context Menus

You can access the context menu by right-clicking. Commands within the context menu adjust depending on what's selected and where you right-click on the datagrid.

4.3 Keyboard Shortcuts

You can manipulate your data and navigate the datagrid through keyboard shortcuts.

4.3.1 General extension shortcuts

Keypress	Command
Ctrl + X	Cut the selected item and copy it to the clipboard
Ctrl + C	Copy the selected item to the clipboard
Ctrl + V	Paste the contents of the clipboard
Ctrl + Z	Undo the previous action
Shift + Ctrl + Z	Redo the previous action
Ctrl + S	Save the current file
Ctrl + F	Open the Find window
Space	Edit a cell

4.3.2 Moving around in the datagrid

Keypress	Command
Left/Right Arrow	Move one cell to the left or right
Ctrl + Left/Right Arrow	Move to the farthest cell left or right in the row
Up/Down Arrow	Move one cell up or down
Ctrl + Up/Down Arrow	Move to the top or bottom cell in the column
Tab	Move one cell to the right
Shift + Tab	Move one cell to the left
Enter	Move one cell down
Shift + Enter	Move one cell up

4.3.3 Selecting cells

Keypress	Command
Shift + Left/Right Arrow	Extend the cell selection one cell to the left or right
Shift + Up/Down Arrow	Extend the cell selection one cell up or down
Shift + Ctrl + Left/Right Arrow	Extend the cell selection to the farthest cell left or right
Shift + Ctrl + Up/Down Arrow	Extend the cell selection to the farthest cell up or down

CHAPTER 5

Features

5.1 Launch new files and quickly add rows and columns

5.2 Seamlessly rearrange your data table

5.3 Insert and remove multiple rows and columns

5.4 Format your data with a click of a button

5.5 Search and replace with ease

CHAPTER 6

Codebase Orientation

This codebase was adapted from JupyterLab's csvviewer and csvviewer-extension packages.

6.1 Directories

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

6.1.1 Source Code: `src/`

This contains the primary TypeScript files for this extension, which are compiled to JavaScript.

6.1.2 Binder setup: `binder/`

This contains an environment specification for `repo2docker` which allows the repository to be tested on mybinder.org.

6.1.3 Demo: `demo/`

The `demo/` directory contains sample csv files and Jupyter notebooks that highlight some features of this extension.

6.1.4 Design: `design/`

A directory containing a series of design documents and prototypes motivating various choices made in the course of building the Tabular Data Editor.

6.1.5 Documentation: `docs/`

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

6.1.6 Styling: `style/`

This directory contains the icon assets and css styles for this extension.

6.1.7 Testing: `test/`

Tests for the TypeScript files in the `src/` directory. These test files pull in the TypeScript sources and exercise their APIs.

Run `jlpm test` from the root directory to run all tests for this extension

6.1.8 Test Utilities: `testutils/`

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

CHAPTER 7

Contributing to the Tabular Data Editor

If you're reading this section, you're probably interested in contributing to tabular Data Editor. Welcome and thanks for your interest in contributing!

Please take a look at the Contributor documentation, familiarize yourself with using JupyterLab, and introduce yourself to the community (on the mailing list or discourse) and share what area of the project you are interested in working on. Please also see the Jupyter Community Guides.

We have labeled some issues as good first issue or help wanted that we believe are good examples of small, self-contained changes. We encourage those that are new to the code base to implement and/or ask questions about these issues.

Table of contents

7.1 General Guidelines for Contributing

For general documentation about contributing to Jupyter projects, see the Project Jupyter Contributor Documentation and Code of Conduct.

All source code is written in TypeScript. See the Style Guide.

All source code is formatted using prettier. When code is modified and committed, all staged files will be automatically formatted using pre-commit git hooks (with help from the lint-staged and husky libraries). The benefit of using a code formatter like prettier is that it removes the topic of code style from the conversation when reviewing pull requests, thereby speeding up the review process.

You may also use the prettier npm script (e.g. `npm run prettier` or `yarn prettier` or `jlpm prettier`) to format the entire code base. We recommend installing a prettier extension for your code editor and configuring it to format your code with a keyboard shortcut or automatically on save.

7.2 Submitting a Pull Request Contribution

Generally, an issue should be opened describing a piece of proposed work and the issues it solves before a pull request is opened.

7.2.1 Issue Management

Opening an issue lets community members participate in the design discussion, makes others aware of work being done, and sets the stage for a fruitful community interaction. A pull request should reference the issue it is addressing. Once the pull request is merged, the issue related to it will also be closed. If there is additional discussion around implemementation the issue may be re-opened.

7.3 Setting Up a Development Environment

You can launch a binder with the latest JupyterLab Tabular Editor to test something (this may take a few minutes to load):

7.3.1 Installing Node.js and jlpm

Building JupyterLab from its GitHub source code requires Node.js. The development version requires Node.js version 10+, as defined in the `engines` specification in dev_mode/package.json.

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

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

If you use Homebrew on Mac OS X:

```
brew install node
```

You can also use the installer from the Node.js website.

To check which version of Node.js is installed:

```
node -v
```

7.4 Installing JupyterLab

JupyterLab requires Jupyter Notebook version 4.3 or later.

If you use `conda`, you can install notebook using:

```
conda install -c conda-forge notebook
```

You may also want to install `nb_conda_kernels` to have a kernel option for different conda environments

```
conda install -c conda-forge nb_conda_kernels
```

If you use `pip`, you can install notebook using:

```
pip install notebook
```

Fork the JupyterLab repository.

Once you have installed the dependencies mentioned above, use the following steps:

```
git clone https://github.com/<your-github-username>/jupyterlab.git
cd jupyterlab
pip install -e .
jlpm install
jlpm run build  # Build the dev mode assets (optional)
jlpm run build:core  # Build the core mode assets (optional)
jupyter lab build  # Build the app dir assets (optional)
```

Notes:

- A few of the scripts will run "python". If your target python is called something else (such as "python3") then parts of the build will fail. You may wish to build in a conda environment, or make an alias.
- Some of the packages used in the development environment require Python 3.0 or higher. If you encounter an ImportError during the installation, make sure Python 3.0+ is installed. Also, try using the Python 3.0+ version of `pip` or `pip3 install -e .` command to install JupyterLab from the forked repository.
- The `jlpm` command is a JupyterLab-provided, locked version of the yarn package manager. If you have `yarn` installed already, you can use the `yarn` command when developing, and it will use the local version of `yarn` in `jupyterlab/yarn.js` when run in the repository or a built application directory.
- If you decide to use the `jlpm` command and encounter the `jlpm: command not found` error, try adding the user-level bin directory to your `PATH` environment variable. You already installed `jlpm` along with JupyterLab in the previous command, but `jlpm` might not be accessible due to `PATH` environment variable related issues. If you are using a Unix derivative (FreeBSD, GNU / Linux, OS X), you can achieve this by using `export PATH="$HOME/.local/bin:$PATH"` command.
- At times, it may be necessary to clean your local repo with the command `npm run clean:slate`. This will clean the repository, and re-install and rebuild.
- If `pip` gives a `VersionConflict` error, it usually means that the installed version of `jupyterlab_server` is out of date. Run `pip install --upgrade jupyterlab_server` to get the latest version.

- To install JupyterLab in isolation for a single conda/virtual environment, you can add the `--sys-prefix` flag to the extension activation above; this will tie the installation to the `sys.prefix` location of your environment, without writing anything in your user-wide settings area (which are visible to all your envs):
- You can run `jlpm run build:dev:prod` to build more accurate sourcemaps that show the original Typescript code when debugging. However, it takes a bit longer to build the sources, so is used only to build for production by default.

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 --sys-prefix jupyterlab
```

For installation instructions to write documentation, please see *Writing Documentation*

7.4.1 Run JupyterLab

Start JupyterLab:

```
jupyter lab --watch
```

7.4.2 Run the Tests

```
jlpm test
```

We use `jest` for all tests, so standard `jest` workflows apply. Tests can be debugged in either VSCode or Chrome. It can help to add an `it.only` to a specific test when debugging. All of the `test*` scripts in each package accept `jest` cli options.

VSCode Debugging

To debug in VSCode, open a package folder in VSCode. We provide a launch configuration in each package folder. In a terminal, run `jlpm test:debug:watch`. In VSCode, select "Attach to Jest" from the "Run" sidebar to begin debugging. See VSCode docs on debugging for more details.

Chrome Debugging

To debug in Chrome, run `jlpm test:debug:watch` in the terminal. Open Chrome and go to `chrome://inspect/`. Select the remote device and begin debugging.

Testing Utilities

There are some helper functions in `testutils` (which is a public npm package called `@jupyterlab/testutils`) that are used by many of the tests.

For tests that rely on `@jupyterlab/services` (starting kernels, interacting with files, etc.), there are two options. If a simple interaction is needed, the `Mock` namespace exposed by `testutils` has a number of mock implmentations (see `testutils/src/mock.ts`). If a full server interaction is required, use the `JupyterServer` class.

We have a helper function called `testEmission` to help with writing tests that use `Lumino` signals, as well as a `framePromise` function to get a `Promise` for a `requestAnimationFrame`. We sometimes have to set

a sentinel value inside a `Promise` and then check that the sentinel was set if we need a promise to run without blocking.

7.5 Performance Testing

If you are making a change that might affect how long it takes to load JupyterLab in the browser, we recommend doing some performance testing using Lighthouse. It let's you easily compute a number of metrics, like page load time, for the site.

To use it, first build JupyterLab in dev mode:

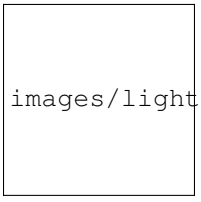
```
jlpm run build:dev
```

Then, start JupyterLab using the dev build:

```
jupyter lab --dev --NotebookApp.token=''  --no-browser
```

Now run Lighthouse against this local server and show the results:

```
jlpm run lighthouse --view
```



7.5.1 Using throttling

Lighthouse recommends using the system level comcast tool to throttle your network connection and emulate different scenarios. To use it, first install that tool using `go`:

```
go get github.com/tylertreat/comcast
```

Then, before you run Lighthouse, enable the throttling (this requires sudo):

```
run lighthouse:throttling:start
```

This enables the "WIFI (good)" preset of comcast, which should emulate loading JupyterLab over a local network.

Then run the lighthouse tests:

```
jlpm run lighthouse [...]
```

Then disable the throttling after you are done:

```
jlpm run lighthouse:throttling:stop
```

7.5.2 Comparing results

Performance results are usually only useful in comparison to other results. For that reason, we have included a comparison script that can take two lighthouse results and show the changes between them.

Let's say we want to compare the results of the production build of JupyterLab with the normal build. The production build minifies all the JavaScript, so should load a bit faster.

First, we build JupyterLab normally, start it up, profile it and save the results:

```
jlpm build:dev
jupyter lab --dev --NotebookApp.token='' --no-browser

# in new window
jlpm run lighthouse --output json --output-path normal.json
```

Then rebuild with the production build and retest:

```
jlpm run build:dev:prod
jupyter lab --dev --NotebookApp.token='' --no-browser

# in new window
jlpm run lighthouse --output json --output-path prod.json
```

Now we can use compare the two outputs:

```
jlpm run lighthouse:compare normal.json prod.json
```

This gives us a report of the relative differences between the audits in the two reports:

Resulting Output

`normal.json` -> `prod.json`

First Contentful Paint
- -62% Δ
- 1.9 s -> 0.7 s
- First Contentful Paint marks the time at which the first text or image is painted. Learn more.

First Meaningful Paint
- -50% Δ
- 2.5 s -> 1.3 s
- First Meaningful Paint measures when the primary content of a page is visible. Learn more.

Speed Index
- -48% Δ
- 2.6 s -> 1.3 s
- Speed Index shows how quickly the contents of a page are visibly populated. Learn more.

Estimated Input Latency
- 0% Δ
- 20 ms -> 20 ms
- Estimated Input Latency is an estimate of how long your app takes to respond to user input, in milliseconds, during the busiest 5s window of page load. If your latency is higher than 50 ms, users may perceive your app as laggy. Learn more.

Max Potential First Input Delay
- 9% Δ
- 200 ms -> 210 ms
- The maximum potential First Input Delay that your users could experience is the duration, in milliseconds, of the longest task. Learn more.

First CPU Idle
- -50% Δ
- 2.5 s -> 1.3 s
- First CPU Idle marks the first time at which the page's main thread is quiet enough to handle input. Learn more.

Time to Interactive
- -52% Δ
- 2.5 s -> 1.2 s
- Time to interactive is the amount of time it takes for the page to become fully interactive. Learn more.

Avoid multiple page redirects
- -2% Δ
- Potential savings of 10 ms -> Potential savings of 10 ms
- Redirects introduce additional delays before the page can be loaded. Learn more.

Minimize main-thread work
- -54% Δ
- 2.1 s -> 1.0 s
- Consider reducing the time spent parsing, compiling and executing JS. You may find delivering smaller JS payloads helps with this.

JavaScript execution time
- -49% Δ
- 1.1 s -> 0.6 s
- Consider reducing the time spent parsing, compiling, and executing JS. You may find delivering smaller JS payloads helps with this. Learn more.

Preload key requests
- -100% Δ
- Potential savings of 240 ms ->
- Consider using <link rel=preload> to prioritize fetching resources that are currently requested later in page load. Learn more.

Uses efficient cache policy on static assets
- 0% Δ
- 1 resource found -> 1 resource found
- A long cache lifetime can speed up repeat visits to your page. Learn more.

Avoid enormous network payloads
- -86% Δ
- Total size was 30,131 KB -> Total size was 4,294 KB
- Large network payloads cost users real money and are highly correlated with long load times. Learn more.

Minify JavaScript
- -100% Δ
- Potential savings of 23,041 KB ->
- Minifying JavaScript files can reduce payload sizes and script parse time. Learn more.

Enable text compression
- -86% Δ
- Potential savings of 23,088 KB -> Potential savings of 3,112 KB
- Text-based resources should be served with compression (gzip, deflate or brotli) to minimize total network bytes. Learn more.

Avoid an excessive DOM size
- 0% Δ
- 1,268 elements -> 1,268 elements
- Browser engineers recommend pages contain fewer than ~1,500 DOM elements. The sweet spot is a tree depth < 32 elements and fewer than 60 children/parent element. A large DOM can increase memory usage, cause longer style calculations, and produce costly layout reflows. Learn more.

7.6 Debugging in the Browser

All methods of building JupyterLab produce source maps. The source maps should be available in the source files view of your browser's development tools under the `webpack://` header.

When running JupyterLab normally, expand the ~ header to see the source maps for individual packages.

When running in `--dev-mode`, the core packages are available under `packages/`, while the third party libraries are available under ~. Note: it is recommended to use `jupyter lab --watch --dev-mode` while debugging.

When running a test, the packages will be available at the top level (e.g. `application/src`), and the current set of test files available under `/src`. Note: it is recommended to use `jlpm run watch` in the test folder while debugging test options. See *above* for more info.

7.7 Writing Documentation

Documentation is written in Markdown and reStructuredText. In particular, the documentation on our Read the Docs page is written in reStructuredText. To ensure that the Read the Docs page builds, you'll need to install the documentation dependencies with `pip`:

```
pip install -r docs/requirements.txt
```

To test the docs run:

```
py.test --check-links -k .md . || py.test --check-links -k .md --lf .
```

The Read the Docs pages can be built using `make`:

```
cd docs
make html
```

Or with `jlpm`:

```
jlpm run docs
```

7.7.1 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.

7.7.2 User Interface Naming Conventions

Documents, Files, and Activities

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

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

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

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

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 interface for descriptions of elements in the UI.

7.8 Testing Changes to External Packages

7.8.1 Linking/Unlinking Packages to JupyterLab

If you want to make changes to one of JupyterLab’s external packages (for example, Lumino and test them out against your copy of JupyterLab, you can easily do so using the `link` command:

1. Make your changes and then build the external package
2. Register a link to the modified external package
    - navigate to the external package dir and run `jlpm link`
3. Link JupyterLab to modded package
    - navigate to top level of your JupyterLab repo, then run `jlpm link "<package-of-interest>"`

You can then (re)build JupyterLab (eg `jlpm run build`) and your changes should be picked up by the build.

To restore JupyterLab to its original state, you use the `unlink` command:

1. Unlink JupyterLab and modded package
    - navigate to top level of your JupyterLab repo, then run `jlpm unlink "<package-of-interest>"`
2. Reinstall original version of the external package in JupyterLab
    - run `jlpm install --check-files`

You can then (re)build JupyterLab and everything should be back to default.

7.8.2 Possible Linking Pitfalls

If you're working on an external project with more than one package, you'll probably have to link in your copies of every package in the project, including those you made no changes to. Failing to do so may cause issues relating to duplication of shared state.

Specifically, when working with Lumino, you'll probably have to link your copy of the `"@lumino/messaging"` package (in addition to whatever packages you actually made changes to). This is due to potential duplication of objects contained in the `MessageLoop` namespace provided by the `messaging` package.

7.9 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.

7.10 Screenshots and Animations

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

- Make sure the screenshot does not contain copyrighted material (preferable), or the license is allowed in our documentation and clearly stated.
- If taking a png screenshot, use the Firefox or Chrome developer tools to do the following:
    - set the browser viewport to 1280x720 pixels
    - set the device pixel ratio to 1:1 (i.e., non-hidpi, non-retina)
    - screenshot the entire *viewport* using the browser developer tools. Screenshots should not include any browser elements such as the browser address bar, browser title bar, etc., and should not contain any desktop background.
- If creating a movie, adjust the settings as above (1280x720 viewport resolution, non-hidpi) and use a screen capture utility of your choice to capture just the browser viewport.

- For PNGs, reduce their size using `pngquant --speed 1 <filename>`. The resulting filename will have `-fs8` appended, so make sure to rename it and use the resulting file. Commit the optimized png file to the main repository. Each png file should be no more than a few hundred kilobytes.
- For movies, upload them to the IPython/Jupyter YouTube channel and add them to the jupyterlab-media repository. To embed a movie in the documentation, use the `www.youtube-nocookie.com` website, which can be found by clicking on the 'privacy-enhanced' embedding option in the Share dialog on YouTube. Add the following parameters the end of the URL `?rel=0&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.

7.11 Notes

- The npm modules are fully compatible with Node/Babel/ES6/ES5. Simply omit the type declarations when using a language other than TypeScript.