Jupyter Notebook Documentation

Release 4.4.1

https://jupyter.org

Jul 03, 2017

User Documentation

1 The Jupyter Notebook 1

2 UI Components 9

3 Comms 13

4 Config file and command line options 15

5 Running a notebook server 27

6 Security in the Jupyter notebook server 31

7 Security in notebook documents 33

8 Configuring the notebook frontend 37

9 Extending the Notebook 39

10 Installing Javascript machinery 49

11 Developer FAQ 51

12 Distributing Jupyter Extensions as Python Packages 53

13 Examples and Tutorials 57

14 Jupyter notebook changelog 59

CHAPTER 1

The Jupyter Notebook

Introduction

The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results. The Jupyter notebook combines two components:

A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.

Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

See also:

See the installation guide on how to install the notebook and its dependencies.

Main features of the web application

- In-browser editing for code, with automatic syntax highlighting, indentation, and tab completion/introspection.
- The ability to execute code from the browser, with the results of computations attached to the code which generated them.
- Displaying the result of computation using rich media representations, such as HTML, LaTeX, PNG, SVG, etc. For example, publication-quality figures rendered by the matplotlib library, can be included inline.
- In-browser editing for rich text using the Markdown markup language, which can provide commentary for the code, is not limited to plain text.
- The ability to easily include mathematical notation within markdown cells using LaTeX, and rendered natively by MathJax.

Notebook documents

Notebook documents contains the inputs and outputs of a interactive session as well as additional text that accompanies the code but is not meant for execution. In this way, notebook files can serve as a complete computational record of a session, interleaving executable code with explanatory text, mathematics, and rich representations of resulting objects. These documents are internally JSON files and are saved with the `.ipynb` extension. Since JSON is a plain text format, they can be version-controlled and shared with colleagues.

Notebooks may be exported to a range of static formats, including HTML (for example, for blog posts), reStructuredText, LaTeX, PDF, and slide shows, via the nbconvert command.

Furthermore, any `.ipynb` notebook document available from a public URL can be shared via the Jupyter Notebook Viewer (nbviewer). This service loads the notebook document from the URL and renders it as a static web page. The results may thus be shared with a colleague, or as a public blog post, without other users needing to install the Jupyter notebook themselves. In effect, nbviewer is simply nbconvert as a web service, so you can do your own static conversions with nbconvert, without relying on nbviewer.

See also:

Details on the notebook JSON file format

Starting the notebook server

You can start running a notebook server from the command line using the following command:

```
jupyter notebook
```

This will print some information about the notebook server in your console, and open a web browser to the URL of the web application (by default, `http://127.0.0.1:8888`).

The landing page of the Jupyter notebook web application, the **dashboard**, shows the notebooks currently available in the notebook directory (by default, the directory from which the notebook server was started).

You can create new notebooks from the dashboard with the `New Notebook` button, or open existing ones by clicking on their name. You can also drag and drop `.ipynb` notebooks and standard `.py` Python source code files into the notebook list area.

When starting a notebook server from the command line, you can also open a particular notebook directly, bypassing the dashboard, with `jupyter notebook my_notebook.ipynb`. The `.ipynb` extension is assumed if no extension is given.

When you are inside an open notebook, the *File* | *Open...* menu option will open the dashboard in a new browser tab, to allow you to open another notebook from the notebook directory or to create a new notebook.

Note: You can start more than one notebook server at the same time, if you want to work on notebooks in different directories. By default the first notebook server starts on port 8888, and later notebook servers search for ports near that one. You can also manually specify the port with the `--port` option.

Creating a new notebook document

A new notebook may be created at any time, either from the dashboard, or using the *File* | *New* menu option from within an active notebook. The new notebook is created within the same directory and will open in a new browser tab. It will also be reflected as a new entry in the notebook list on the dashboard.

Opening notebooks

An open notebook has **exactly one** interactive session connected to an IPython kernel, which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel. In the dashboard, notebooks with an active kernel have a `Shutdown` button next to them, whereas notebooks without an active kernel have a `Delete` button in its place.

Other clients may connect to the same underlying IPython kernel. The notebook server always prints to the terminal the full details of how to connect to each kernel, with messages such as the following:

```
[NotebookApp] Kernel started: 87f7d2c0-13e3-43df-8bb8-1bd37aaf3373
```

This long string is the kernel's ID which is sufficient for getting the information necessary to connect to the kernel. You can also request this connection data by running the `%connect_info` magic. This will print the same ID information as well as the content of the JSON data structure it contains.

You can then, for example, manually start a Qt console connected to the *same* kernel from the command line, by passing a portion of the ID:

```
$ ipython qtconsole --existing 87f7d2c0
```

Without an ID, `--existing` will connect to the most recently started kernel. This can also be done by running the `%qtconsole` magic in the notebook.

See also:

Decoupled two-process model

Notebook user interface

When you create a new notebook document, you will be presented with the **notebook name**, a **menu bar**, a **toolbar** and an empty **code cell**.

notebook name: The name of the notebook document is displayed at the top of the page, next to the `IP[y]: Notebook` logo. This name reflects the name of the `.ipynb` notebook document file. Clicking on the notebook name brings up a dialog which allows you to rename it. Thus, renaming a notebook from "Untitled0" to "My first notebook" in the browser, renames the `Untitled0.ipynb` file to `My first notebook.ipynb`.

menu bar: The menu bar presents different options that may be used to manipulate the way the notebook functions.

toolbar: The tool bar gives a quick way of performing the most-used operations within the notebook, by clicking on an icon.

code cell: the default type of cell, read on for an explanation of cells

---

Note: As of notebook version 4.1, the user interface allows for multiple cells to be selected. The `quick celltype selector`, found in the menubar, will display a dash – when multiple cells are selected to indicate that the type of the cells in the selection might not be unique. The quick selector can still be used to change the type of the selection and will change the type of all the currently selected cells.

Structure of a notebook document

The notebook consists of a sequence of cells. A cell is a multi-line text input field, and its contents can be executed by using `Shift-Enter`, or by clicking either the "Play" button the toolbar, or *Cell | Run* in the menu bar. The execution behavior of a cell is determined the cell's type. There are four types of cells: **code cells**, **markdown cells**, **raw cells** and **heading cells**. Every cell starts off being a **code cell**, but its type can be changed by using a dropdown on the toolbar (which will be "Code", initially), or via *keyboard shortcuts*.

For more information on the different things you can do in a notebook, see the collection of examples.

Code cells

A *code cell* allows you to edit and write new code, with full syntax highlighting and tab completion. By default, the language associated to a code cell is Python, but other languages, such as `Julia` and `R`, can be handled using cell magic commands.

When a code cell is executed, code that it contains is sent to the kernel associated with the notebook. The results that are returned from this computation are then displayed in the notebook as the cell's *output*. The output is not limited to text, with many other possible forms of output are also possible, including `matplotlib` figures and HTML tables (as used, for example, in the `pandas` data analysis package). This is known as IPython's *rich display* capability.

See also:

Rich Output example notebook

Markdown cells

You can document the computational process in a literate way, alternating descriptive text with code, using *rich text*. In IPython this is accomplished by marking up text with the Markdown language. The corresponding cells are called *Markdown cells*. The Markdown language provides a simple way to perform this text markup, that is, to specify which parts of the text should be emphasized (italics), bold, form lists, etc.

When a Markdown cell is executed, the Markdown code is converted into the corresponding formatted rich text. Markdown allows arbitrary HTML code for formatting.

Within Markdown cells, you can also include *mathematics* in a straightforward way, using standard LaTeX notation: `$...$` for inline mathematics and `$$...$$` for displayed mathematics. When the Markdown cell is executed, the LaTeX portions are automatically rendered in the HTML output as equations with high quality typography. This is made possible by MathJax, which supports a large subset of LaTeX functionality

Standard mathematics environments defined by LaTeX and AMS-LaTeX (the *amsmath* package) also work, such as `\begin{equation}...\end{equation}`, and `\begin{align}...\end{align}`. New LaTeX macros may be defined using standard methods, such as `\newcommand`, by placing them anywhere *between math delimiters* in a Markdown cell. These definitions are then available throughout the rest of the IPython session.

See also:

Markdown Cells example notebook

Raw cells

Raw cells provide a place in which you can write *output* directly. Raw cells are not evaluated by the notebook. When passed through nbconvert, raw cells arrive in the destination format unmodified. For example, this allows you to type full LaTeX into a raw cell, which will only be rendered by LaTeX after conversion by nbconvert.

Heading cells

If you want to provide structure for your document, you can use markdown headings. Markdown headings consist of 1 to 6 hash # signs # followed by a space and the title of your section. The markdown heading will be converted to a clickable link for a section of the notebook. It is also used as a hint when exporting to other document formats, like PDF. We recommend using only one markdown header in a cell and limit the cell's content to the header text. For flexibility of text format conversion, we suggest placing additional text in the next notebook cell.

Basic workflow

The normal workflow in a notebook is, then, quite similar to a standard IPython session, with the difference that you can edit cells in-place multiple times until you obtain the desired results, rather than having to rerun separate scripts with the `%run` magic command.

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

At certain moments, it may be necessary to interrupt a calculation which is taking too long to complete. This may be done with the *Kernel* | *Interrupt* menu option, or the `Ctrl-m i` keyboard shortcut. Similarly, it may be necessary or desirable to restart the whole computational process, with the *Kernel* | *Restart* menu option or `Ctrl-m .` shortcut.

A notebook may be downloaded in either a `.ipynb` or `.py` file from the menu option *File* | *Download as*. Choosing the `.py` option downloads a Python `.py` script, in which all rich output has been removed and the content of markdown cells have been inserted as comments.

See also:

Running Code in the Jupyter Notebook example notebook

Notebook Basics example notebook

a warning about doing "roundtrip" conversions.

Keyboard shortcuts

All actions in the notebook can be performed with the mouse, but keyboard shortcuts are also available for the most common ones. The essential shortcuts to remember are the following:

- **`Shift-Enter`: run cell** Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter` is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.
- **`Ctrl-Enter`: run cell in-place** Execute the current cell as if it were in "terminal mode", where any output is shown, but the cursor *remains* in the current cell. The cell's entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.
- **`Alt-Enter`: run cell, insert below** Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). This is thus a shortcut for the sequence `Shift-Enter`, `Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.)
- **`Esc` and `Enter`: Command mode and edit mode** In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

For the full list of available shortcuts, click *Help*, *Keyboard Shortcuts* in the notebook menus.

Plotting

One major feature of the Jupyter notebook is the ability to display plots that are the output of running code cells. The IPython kernel is designed to work seamlessly with the matplotlib plotting library to provide this functionality. Specific plotting library integration is a feature of the kernel.

Installing kernels

For information on how to install a Python kernel, refer to the IPython install page.

Kernels for other languages can be found in the IPython wiki. They usually come with instruction what to run to make the kernel available in the notebook.

Signing Notebooks

To prevent untrusted code from executing on users' behalf when notebooks open, we have added a signature to the notebook, stored in metadata. The notebook server verifies this signature when a notebook is opened. If the signature stored in the notebook metadata does not match, javascript and HTML output will not be displayed on load, and must be regenerated by re-executing the cells.

Any notebook that you have executed yourself *in its entirety* will be considered trusted, and its HTML and javascript output will be displayed on load.

If you need to see HTML or Javascript output without re-executing, you can explicitly trust notebooks, such as those shared with you, or those that you have written yourself prior to IPython 2.0, at the command-line with:

```
$ jupyter trust mynotebook.ipynb [other notebooks.ipynb]
```

This just generates a new signature stored in each notebook.

You can generate a new notebook signing key with:

```
$ jupyter trust --reset
```

Browser Compatibility

The Jupyter Notebook is officially supported by the latest stable versions of the following browsers:

- Chrome
- Safari
- Firefox

The is mainly due to the notebook's usage of WebSockets and the flexible box model.

The following browsers are unsupported:

- Safari < 5
- Firefox < 6

- Chrome < 13
- Opera (any): CSS issues, but execution might work
- Internet Explorer < 10
- Internet Explorer 10 (same as Opera)

Using Safari with HTTPS and an untrusted certificate is known to not work (websockets will fail).

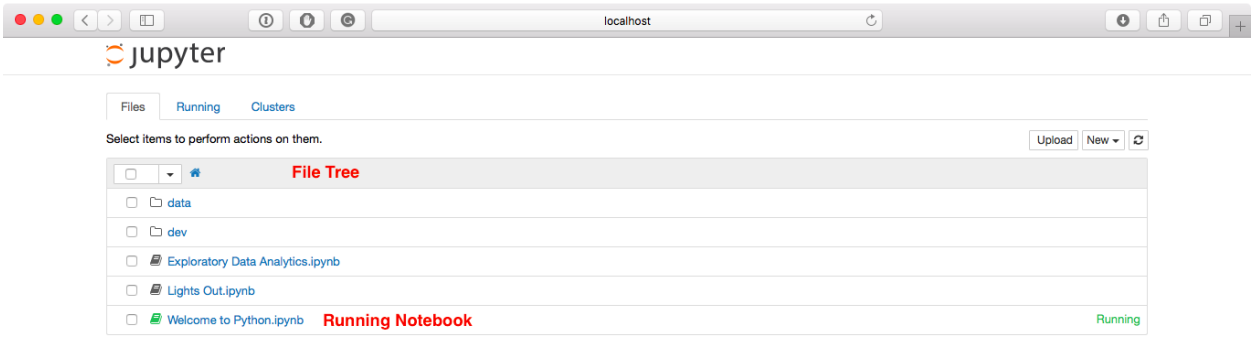
CHAPTER 2

UI Components

When opening bug reports or sending emails to the Jupyter mailing list, it is useful to know the names of different UI components so that other developers and users have an easier time helping you diagnose your problems. This section will familiarize you with the names of UI elements within the Notebook and the different Notebook modes.

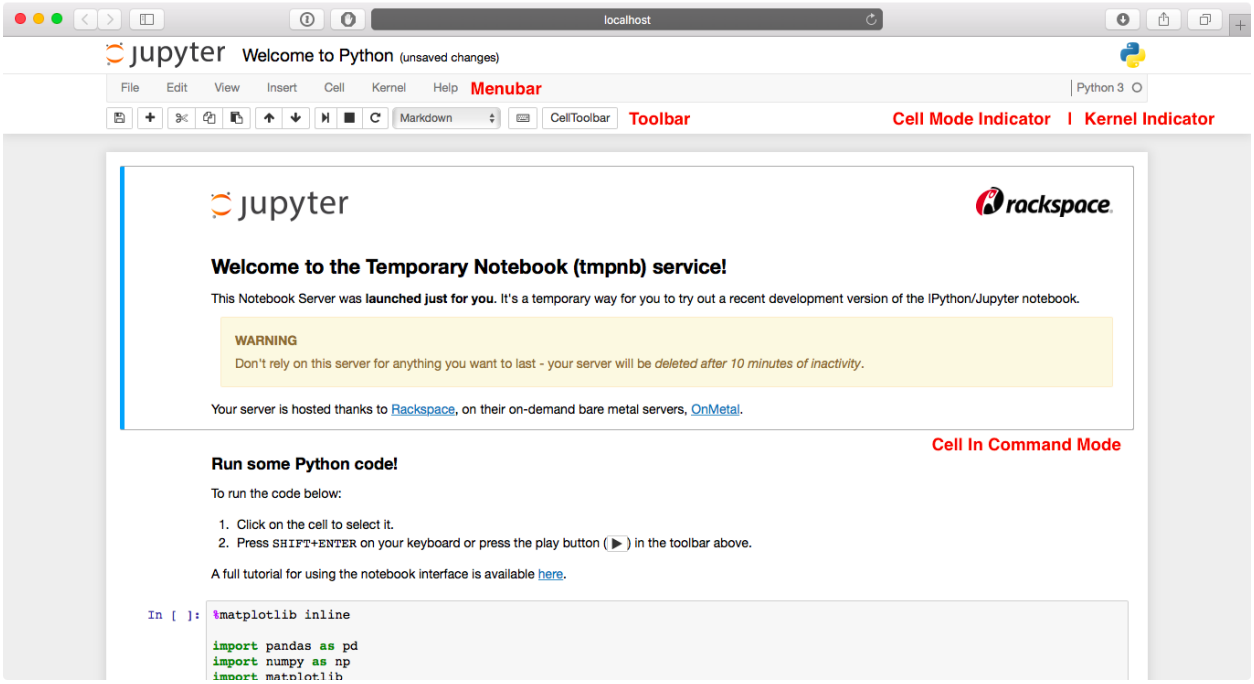
Notebook Dashboard

When you launch `jupyter notebook` the first page that you encounter is the Notebook Dashboard.



Notebook Editor

Once you've selected a Notebook to edit, the Notebook will open in the Notebook Editor.

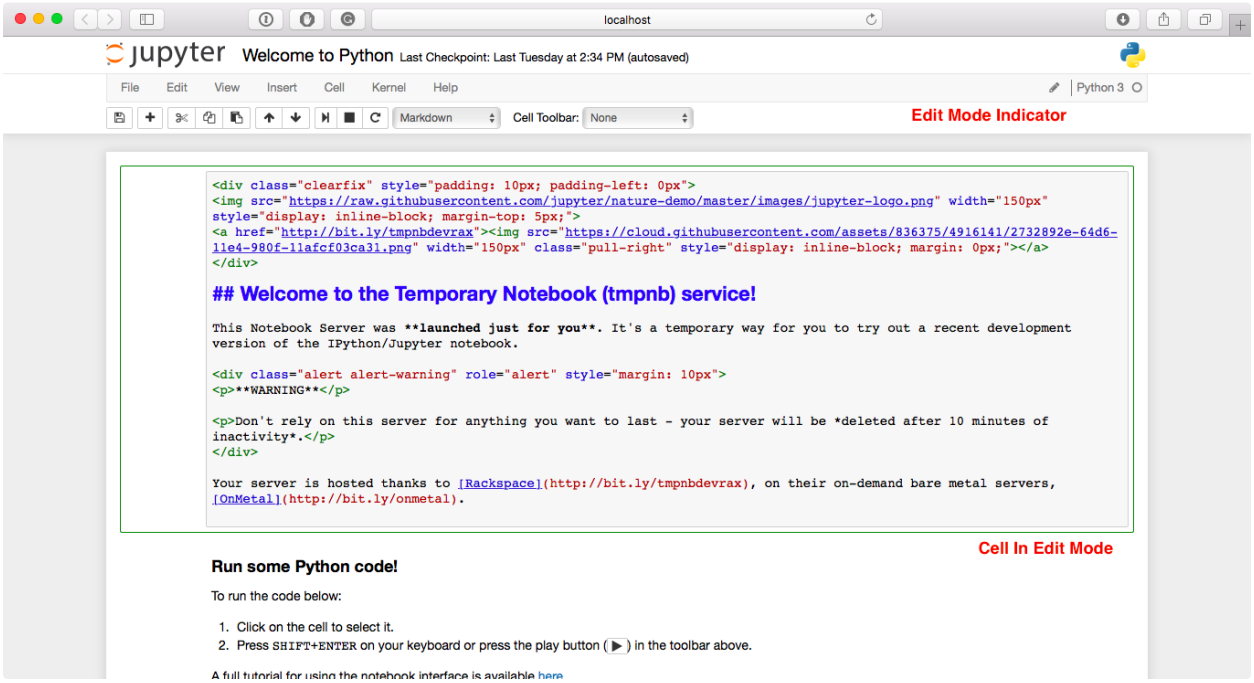


Interactive User Interface Tour of the Notebook

If you would like to learn more about the specific elements within the Notebook Editor, you can go through the User Interface Tour by selecting Help in the menubar then selecting User Interface Tour.

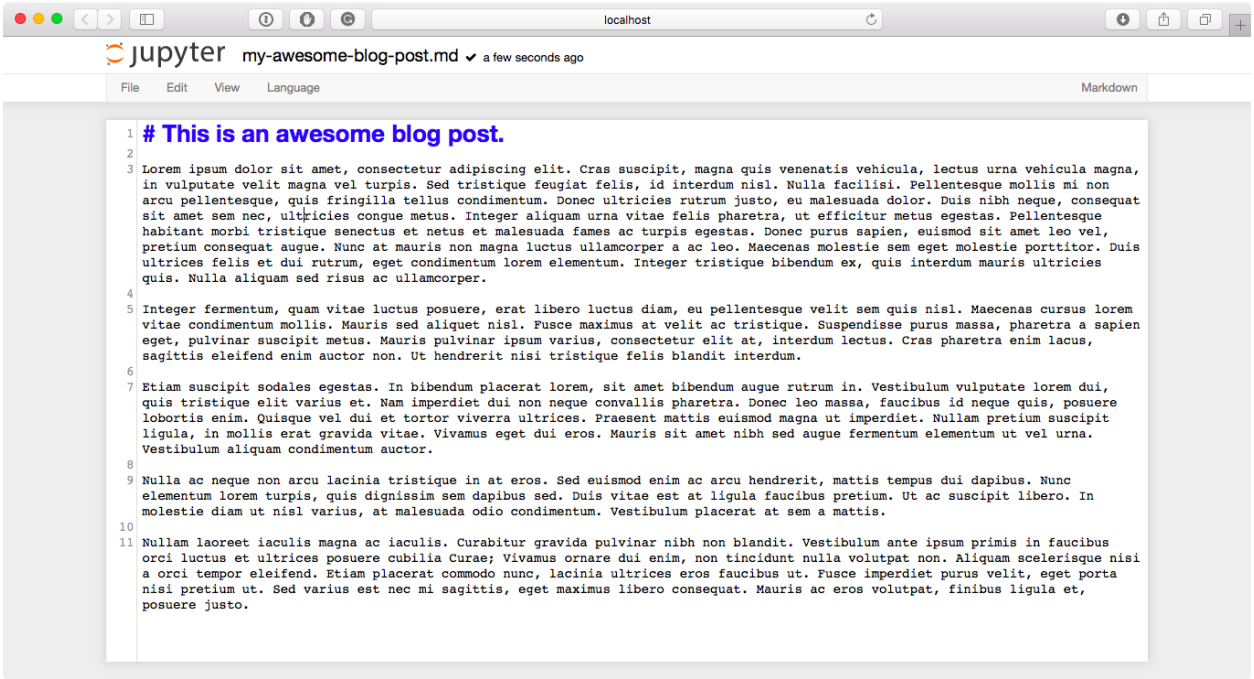
Edit Mode and Notebook Editor

When a cell is in edit mode, the Cell Mode Indicator will change to reflect the cell's state. This state is indicated by a small pencil icon on the top right of the interface. When the cell is in command mode, there is no icon in that location.



File Editor

Now let's say that you've chosen to open a Markdown file instead of a Notebook file whilst in the Notebook Dashboard. If so, the file will be opened in the File Editor.



CHAPTER 3

Comms

Comms allow custom messages between the frontend and the kernel. They are used, for instance, in ipywidgets to update widget state.

A comm consists of a pair of objects, in the kernel and the frontend, with an automatically assigned unique ID. When one side sends a message, a callback on the other side is triggered with that message data. Either side, the frontend or kernel, can open or close the comm.

See also:

Custom Messages The messaging specification section on comms

Opening a comm from the kernel

First, the function to accept the comm must be available on the frontend. This can either be specified in a *requirejs* module, or registered in a registry, for example when an *extension* is loaded. This example shows a frontend comm target registered in a registry:

```
Jupyter.notebook.kernel.comm_manager.register_target('my_comm_target',
    function(comm, msg) {
        // comm is the frontend comm instance
        // msg is the comm_open message, which can carry data

        // Register handlers for later messages:
        comm.on_msg(function(msg) {...});
        comm.on_close(function(msg) {...});
        comm.send({'foo': 0});
    });
```

Now that the frontend comm is registered, you can open the comm from the kernel:

```
from ipykernel.comm import Comm

# Use comm to send a message from the kernel
```

```
my_comm = Comm(target_name='my_comm_target', data={'foo': 1})
my_comm.send({'foo': 2})

# Add a callback for received messages.
@my_comm.on_msg
def _recv(msg):
    # Use msg['content']['data'] for the data in the message
```

This example uses the IPython kernel; it's up to each language kernel what API, if any, it offers for using comms.

Opening a comm from the frontend

This is very similar to above, but in reverse. First, a comm target must be registered in the kernel. For instance, this may be done by code displaying output: it will register a target in the kernel, and then display output containing Javascript to connect to it.

```
def target_func(comm, msg):
    # comm is the kernel Comm instance
    # msg is the comm_open message

    # Register handler for later messages
    @comm.on_msg
    def _recv(msg):
        # Use msg['content']['data'] for the data in the message

    # Send data to the frontend
    comm.send({'foo': 5})

get_ipython().kernel.comm_manager.register_target('my_comm_target', target_func)
```

This example uses the IPython kernel again; this example will be different in other kernels that support comms. Refer to the specific language kernel's documentation for comms support.

And then open the comm from the frontend:

```
comm = Jupyter.notebook.kernel.comm_manager.new_comm('my_comm_target',
                                                     {'foo': 6})
// Send data
comm.send({'foo': 7})

// Register a handler
comm.on_msg(function(msg) {
    console.log(msg.content.data.foo);
});
```

CHAPTER 4

Config file and command line options

The notebook server can be run with a variety of command line arguments. A list of available options can be found below in the *options section*.

Defaults for these options can also be set by creating a file named `jupyter_notebook_config.py` in your Jupyter folder. The Jupyter folder is in your home directory, `~/.jupyter`.

To create a `jupyter_notebook_config.py` file, with all the defaults commented out, you can use the following command line:

```
$ jupyter notebook --generate-config
```

Options

This list of options can be generated by running the following and hitting enter:

```
$ jupyter notebook --help
```

Application.log_datefmt [Unicode] Default: `'%Y-%m-%d %H:%M:%S'`

The date format used by logging formatters for %(asctime)s

Application.log_format [Unicode] Default: `'[%(name)s]%(highlevel)s %(message)s'`

The Logging format template

Application.log_level [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: `30`

Set the log level by value or name.

Application.show_config [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

Application.show_config_json [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

JupyterApp.answer_yes [Bool] Default: `False`

Answer yes to any prompts.

JupyterApp.config_file [Unicode] Default: `''`

Full path of a config file.

JupyterApp.config_file_name [Unicode] Default: `''`

Specify a config file to load.

JupyterApp.generate_config [Bool] Default: `False`

Generate default config file.

JupyterApp.log_datefmt [Unicode] Default: `'%Y-%m-%d %H:%M:%S'`

The date format used by logging formatters for %(asctime)s

JupyterApp.log_format [Unicode] Default: `'[%(name)s]%(highlevel)s %(message)s'`

The Logging format template

JupyterApp.log_level [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: `30`

Set the log level by value or name.

JupyterApp.show_config [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

JupyterApp.show_config_json [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

NotebookApp.allow_credentials [Bool] Default: `False`

Set the Access-Control-Allow-Credentials: true header

NotebookApp.allow_origin [Unicode] Default: `''`

Set the Access-Control-Allow-Origin header

Use '*' to allow any origin to access your server.

Takes precedence over allow_origin_pat.

NotebookApp.allow_origin_pat [Unicode] Default: `''`

Use a regular expression for the Access-Control-Allow-Origin header

Requests from an origin matching the expression will get replies with:

Access-Control-Allow-Origin: origin

where *origin* is the origin of the request.

Ignored if allow_origin is set.

NotebookApp.allow_root [Bool] Default: `False`

Whether to allow the user to run the notebook as root.

NotebookApp.answer_yes [Bool] Default: `False`

Answer yes to any prompts.

NotebookApp.base_project_url [Unicode] Default: `'/'`

DEPRECATED use base_url

NotebookApp.base_url [Unicode] Default: `'/'`

The base URL for the notebook server.

Leading and trailing slashes can be omitted, and will automatically be added.

NotebookApp.browser [Unicode] Default: `''`

Specify what command to use to invoke a web browser when opening the notebook. If not specified, the default browser will be determined by the *webbrowser* standard library module, which allows setting of the BROWSER environment variable to override it.

NotebookApp.certfile [Unicode] Default: `''`

The full path to an SSL/TLS certificate file.

NotebookApp.client_ca [Unicode] Default: `''`

The full path to a certificate authority certificate for SSL/TLS client authentication.

NotebookApp.config_file [Unicode] Default: `''`

Full path of a config file.

NotebookApp.config_file_name [Unicode] Default: `''`

Specify a config file to load.

NotebookApp.config_manager_class [Type] Default: `'notebook.services.config.manager.ConfigManager'`

The config manager class to use

NotebookApp.contents_manager_class [Type] Default: `'notebook.services.contents.largefilemanager.LargeFileManager'`

The notebook manager class to use.

NotebookApp.cookie_options [Dict] Default: `{}`

Extra keyword arguments to pass to *set_secure_cookie*. See tornado's set_secure_cookie docs for details.

NotebookApp.cookie_secret [Bytes] Default: `b''`

The random bytes used to secure cookies. By default this is a new random number every time you start the Notebook. Set it to a value in a config file to enable logins to persist across server sessions.

Note: Cookie secrets should be kept private, do not share config files with cookie_secret stored in plaintext (you can read the value from a file).

NotebookApp.cookie_secret_file [Unicode] Default: `''`

The file where the cookie secret is stored.

NotebookApp.default_url [Unicode] Default: `'/tree'`

The default URL to redirect to from `/`

NotebookApp.disable_check_xsrf [Bool] Default: `False`

Disable cross-site-request-forgery protection

Jupyter notebook 4.3.1 introduces protection from cross-site request forgeries, requiring API requests to either:

- originate from pages served by this server (validated with XSRF cookie and token), or
- authenticate with a token

Some anonymous compute resources still desire the ability to run code, completely without authentication. These services can disable all authentication and security checks, with the full knowledge of what that implies.

NotebookApp.enable_mathjax [Bool] Default: `True`

Whether to enable MathJax for typesetting math/TeX

MathJax is the javascript library Jupyter uses to render math/LaTeX. It is very large, so you may want to disable it if you have a slow internet connection, or for offline use of the notebook.

When disabled, equations etc. will appear as their untransformed TeX source.

NotebookApp.extra_nbextensions_path [List] Default: `[]`

extra paths to look for Javascript notebook extensions

NotebookApp.extra_static_paths [List] Default: `[]`

Extra paths to search for serving static files.

This allows adding javascript/css to be available from the notebook server machine, or overriding individual files in the IPython

NotebookApp.extra_template_paths [List] Default: `[]`

Extra paths to search for serving jinja templates.

Can be used to override templates from notebook.templates.

NotebookApp.file_to_run [Unicode] Default: `''`

No description

NotebookApp.generate_config [Bool] Default: `False`

Generate default config file.

NotebookApp.ignore_minified_js [Bool] Default: `False`

Deprecated: Use minified JS file or not, mainly use during dev to avoid JS recompilation

NotebookApp.iopub_data_rate_limit [Float] Default: `1000000`

(bytes/sec) Maximum rate at which messages can be sent on iopub before they are limited.

NotebookApp.iopub_msg_rate_limit [Float] Default: `1000`

(msgs/sec) Maximum rate at which messages can be sent on iopub before they are limited.

NotebookApp.ip [Unicode] Default: `'localhost'`

The IP address the notebook server will listen on.

NotebookApp.jinja_environment_options [Dict] Default: `{}`

Supply extra arguments that will be passed to Jinja environment.

NotebookApp.jinja_template_vars [Dict] Default: `{}`

Extra variables to supply to jinja templates when rendering.

NotebookApp.kernel_manager_class [Type] Default: `'notebook.services.kernels.kernelmanager.MappingKernelManager'`

The kernel manager class to use.

NotebookApp.kernel_spec_manager_class [Type] Default: `'jupyter_client.kernelspec.KernelSpecManager'`

The kernel spec manager class to use. Should be a subclass of *jupyter_client.kernelspec.KernelSpecManager*.

The Api of KernelSpecManager is provisional and might change without warning between this version of Jupyter and the next stable one.

NotebookApp.keyfile [Unicode] Default: `''`

The full path to a private key file for usage with SSL/TLS.

NotebookApp.log_datefmt [Unicode] Default: `'%Y-%m-%d %H:%M:%S'`

The date format used by logging formatters for %(asctime)s

NotebookApp.log_format [Unicode] Default: `'[%(name)s]%(highlevel)s %(message)s'`

The Logging format template

NotebookApp.log_level [0|10|20|30|40|50|'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'] Default: `30`

Set the log level by value or name.

NotebookApp.login_handler_class [Type] Default: `'notebook.auth.login.LoginHandler'`

The login handler class to use.

NotebookApp.logout_handler_class [Type] Default: `'notebook.auth.logout.LogoutHandler'`

The logout handler class to use.

NotebookApp.mathjax_config [Unicode] Default: `'TeX-AMS-MML_HTMLorMML-full,Safe'`

The MathJax.js configuration file that is to be used.

NotebookApp.mathjax_url [Unicode] Default: `''`

A custom url for MathJax.js. Should be in the form of a case-sensitive url to MathJax, for example: /static/components/MathJax/MathJax.js

NotebookApp.nbserver_extensions [Dict] Default: `{}`

Dict of Python modules to load as notebook server extensions.Entry values can be used to enable and disable the loading ofthe extensions. The extensions will be loaded in alphabetical order.

NotebookApp.notebook_dir [Unicode] Default: `''`

The directory to use for notebooks and kernels.

NotebookApp.open_browser [Bool] Default: `True`

Whether to open in a browser after starting. The specific browser used is platform dependent and determined by the python standard library *webbrowser* module, unless it is overridden using the –browser (NotebookApp.browser) configuration option.

NotebookApp.password [Unicode] Default: `''`

Hashed password to use for web authentication.

To generate, type in a python/IPython shell:

```
from notebook.auth import passwd; passwd()
```

The string should be of the form type:salt:hashed-password.

NotebookApp.password_required [Bool] Default: `False`

Forces users to use a password for the Notebook server. This is useful in a multi user environment, for instance when everybody in the LAN can access each other's machine though ssh.

In such a case, server the notebook server on localhost is not secure since any user can connect to the notebook server via ssh.

NotebookApp.port [Int] Default: `8888`

The port the notebook server will listen on.

NotebookApp.port_retries [Int] Default: `50`

The number of additional ports to try if the specified port is not available.

NotebookApp.pylab [Unicode] Default: `'disabled'`

DISABLED: use %pylab or %matplotlib in the notebook to enable matplotlib.

NotebookApp.rate_limit_window [Float] Default: `3`

(sec) Time window used to check the message and data rate limits.

NotebookApp.reraise_server_extension_failures [Bool] Default: `False`

Reraise exceptions encountered loading server extensions?

NotebookApp.server_extensions [List] Default: `[]`

DEPRECATED use the nbserver_extensions dict instead

NotebookApp.session_manager_class [Type] Default: `'notebook.services.sessions.sessionmanager.SessionManager'`

The session manager class to use.

NotebookApp.show_config [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout

NotebookApp.show_config_json [Bool] Default: `False`

Instead of starting the Application, dump configuration to stdout (as JSON)

NotebookApp.ssl_options [Dict] Default: `{}`

Supply SSL options for the tornado HTTPServer. See the tornado docs for details.

NotebookApp.terminado_settings [Dict] Default: `{}`

Supply overrides for terminado. Currently only supports “shell_command”.

NotebookApp.token [Unicode] Default: `'<generated>'`

Token used for authenticating first-time connections to the server.

When no password is enabled, the default is to generate a new, random token.

Setting to an empty string disables authentication altogether, which is NOT RECOMMENDED.

NotebookApp.tornado_settings [Dict] Default: `{}`

Supply overrides for the tornado.web.Application that the Jupyter notebook uses.

NotebookApp.trust_xheaders [Bool] Default: `False`

Whether to trust or not X-Scheme/X-Forwarded-Proto and X-Real-Ip/X-Forwarded-For headerssent by the upstream reverse proxy. Necessary if the proxy handles SSL

NotebookApp.webapp_settings [Dict] Default: `{}`

DEPRECATED, use tornado_settings

NotebookApp.websocket_url [Unicode] Default: `''`

The base URL for websockets, if it differs from the HTTP server (hint: it almost certainly doesn’t).

Should be in the form of an HTTP origin: ws[s]://hostname[:port]

ConnectionFileMixin.connection_file [Unicode] Default: `''`

JSON file in which to store connection info [default: kernel-<pid>.json]

This file will contain the IP, ports, and authentication key needed to connect clients to this kernel. By default, this file will be created in the security dir of the current profile, but can be specified by absolute path.

ConnectionFileMixin.control_port [Int] Default: `0`

set the control (ROUTER) port [default: random]

ConnectionFileMixin.hb_port [Int] Default: `0`

set the heartbeat port [default: random]

ConnectionFileMixin.iopub_port [Int] Default: `0`

set the iopub (PUB) port [default: random]

ConnectionFileMixin.ip [Unicode] Default: `''`

Set the kernel's IP address [default localhost]. If the IP address is something other than localhost, then Consoles on other machines will be able to connect to the Kernel, so be careful!

ConnectionFileMixin.shell_port [Int] Default: `0`

set the shell (ROUTER) port [default: random]

ConnectionFileMixin.stdin_port [Int] Default: `0`

set the stdin (ROUTER) port [default: random]

ConnectionFileMixin.transport ['tcp'|'ipc'] Default: `'tcp'`

No description

KernelManager.autorestart [Bool] Default: `True`

Should we autorestart the kernel if it dies.

KernelManager.connection_file [Unicode] Default: `''`

JSON file in which to store connection info [default: kernel-<pid>.json]

This file will contain the IP, ports, and authentication key needed to connect clients to this kernel. By default, this file will be created in the security dir of the current profile, but can be specified by absolute path.

KernelManager.control_port [Int] Default: `0`

set the control (ROUTER) port [default: random]

KernelManager.hb_port [Int] Default: `0`

set the heartbeat port [default: random]

KernelManager.iopub_port [Int] Default: `0`

set the iopub (PUB) port [default: random]

KernelManager.ip [Unicode] Default: `''`

Set the kernel's IP address [default localhost]. If the IP address is something other than localhost, then Consoles on other machines will be able to connect to the Kernel, so be careful!

KernelManager.kernel_cmd [List] Default: `[]`

DEPRECATED: Use kernel_name instead.

The Popen Command to launch the kernel. Override this if you have a custom kernel. If kernel_cmd is specified in a configuration file, Jupyter does not pass any arguments to the kernel, because it cannot make any assumptions about the arguments that the kernel understands. In particular, this means that the kernel does not receive the option –debug if it given on the Jupyter command line.

KernelManager.shell_port [Int] Default: `0`

set the shell (ROUTER) port [default: random]

KernelManager.shutdown_wait_time [Float] Default: `5.0`

Time to wait for a kernel to terminate before killing it, in seconds.

KernelManager.stdin_port [Int] Default: `0`

set the stdin (ROUTER) port [default: random]

KernelManager.transport ['tcp'|'ipc'] Default: `'tcp'`

No description

Session.buffer_threshold [Int] Default: `1024`

Threshold (in bytes) beyond which an object's buffer should be extracted to avoid pickling.

Session.check_pid [Bool] Default: `True`

Whether to check PID to protect against calls after fork.

This check can be disabled if fork-safety is handled elsewhere.

Session.copy_threshold [Int] Default: `65536`

Threshold (in bytes) beyond which a buffer should be sent without copying.

Session.debug [Bool] Default: `False`

Debug output in the Session

Session.digest_history_size [Int] Default: `65536`

The maximum number of digests to remember.

The digest history will be culled when it exceeds this value.

Session.item_threshold [Int] Default: `64`

The maximum number of items for a container to be introspected for custom serialization. Containers larger than this are pickled outright.

Session.key [CBytes] Default: `b''`

execution key, for signing messages.

Session.keyfile [Unicode] Default: `''`

path to file containing execution key.

Session.metadata [Dict] Default: `{}`

Metadata dictionary, which serves as the default top-level metadata dict for each message.

Session.packer [DottedObjectName] Default: `'json'`

The name of the packer for serializing messages. Should be one of 'json', 'pickle', or an import name for a custom callable serializer.

Session.session [CUnicode] Default: `''`

The UUID identifying this session.

Session.signature_scheme [Unicode] Default: `'hmac-sha256'`

The digest scheme used to construct the message signatures. Must have the form 'hmac-HASH'.

Session.unpacker [DottedObjectName] Default: `'json'`

The name of the unpacker for unserializing messages. Only used with custom functions for *packer*.

Session.username [Unicode] Default: `'username'`

Username for the Session. Default is your system username.

MultiKernelManager.default_kernel_name [Unicode] Default: `'python3'`

The name of the default kernel to start

MultiKernelManager.kernel_manager_class [DottedObjectName] Default: `'jupyter_client.ioloop.IOLoopKernelManager'`

The kernel manager class. This is configurable to allow subclassing of the KernelManager for customized behavior.

MappingKernelManager.default_kernel_name [Unicode] Default: `'python3'`

The name of the default kernel to start

MappingKernelManager.kernel_manager_class [DottedObjectName] Default: `'jupyter_client.ioloop.IOLoopKernelManager'`

The kernel manager class. This is configurable to allow subclassing of the KernelManager for customized behavior.

MappingKernelManager.root_dir [Unicode] Default: `''`

No description

ContentsManager.checkpoints [Instance] Default: `None`

No description

ContentsManager.checkpoints_class [Type] Default: `'notebook.services.contents.checkpoints.Checkpoints'`

No description

ContentsManager.checkpoints_kwargs [Dict] Default: `{}`

No description

ContentsManager.hide_globs [List] Default: `['__pycache__', '*.pyc', '*.pyo', '.DS_Store', '*.so', '*.dyl...`

Glob patterns to hide in file and directory listings.

ContentsManager.pre_save_hook [Any] Default: `None`

Python callable or importstring thereof

To be called on a contents model prior to save.

This can be used to process the structure, such as removing notebook outputs or other side effects that should not be saved.

It will be called as (all arguments passed by keyword):

```
hook(path=path, model=model, contents_manager=self)
```

- model: the model to be saved. Includes file contents. Modifying this dict will affect the file that is stored.

- path: the API path of the save destination
- contents_manager: this ContentsManager instance

ContentsManager.root_dir [Unicode] Default: `'/'`

No description

ContentsManager.untitled_directory [Unicode] Default: `'Untitled Folder'`

The base name used when creating untitled directories.

ContentsManager.untitled_file [Unicode] Default: `'untitled'`

The base name used when creating untitled files.

ContentsManager.untitled_notebook [Unicode] Default: `'Untitled'`

The base name used when creating untitled notebooks.

FileManagerMixin.use_atomic_writing [Bool] Default: `True`

By default notebooks are saved on disk on a temporary file and then if succefully written, it replaces the old ones. This procedure, namely 'atomic_writing', causes some bugs on file system whitout operation order enforcement (like some networked fs). If set to False, the new notebook is written directly on the old one which could fail (eg: full filesystem or quota)

FileContentsManager.checkpoints [Instance] Default: `None`

No description

FileContentsManager.checkpoints_class [Type] Default: `'notebook.services.contents.checkpoints.Checkpoints'`

No description

FileContentsManager.checkpoints_kwargs [Dict] Default: `{}`

No description

FileContentsManager.hide_globs [List] Default: `['__pycache__', '*.pyc', '*.pyo', '.DS_Store', '*.so', '*.dyl...`

Glob patterns to hide in file and directory listings.

FileContentsManager.post_save_hook [Any] Default: `None`

Python callable or importstring thereof

to be called on the path of a file just saved.

This can be used to process the file on disk, such as converting the notebook to a script or HTML via nbconvert.

It will be called as (all arguments passed by keyword):

```
hook(os_path=os_path, model=model, contents_manager=instance)
```

- path: the filesystem path to the file just written
- model: the model representing the file
- contents_manager: this ContentsManager instance

FileContentsManager.pre_save_hook [Any] Default: `None`

Python callable or importstring thereof

To be called on a contents model prior to save.

This can be used to process the structure, such as removing notebook outputs or other side effects that should not be saved.

It will be called as (all arguments passed by keyword):

```
hook(path=path, model=model, contents_manager=self)
```

- model: the model to be saved. Includes file contents. Modifying this dict will affect the file that is stored.
- path: the API path of the save destination
- contents_manager: this ContentsManager instance

FileContentsManager.root_dir [Unicode] Default: `''`

No description

FileContentsManager.save_script [Bool] Default: `False`

DEPRECATED, use post_save_hook. Will be removed in Notebook 5.0

FileContentsManager.untitled_directory [Unicode] Default: `'Untitled Folder'`

The base name used when creating untitled directories.

FileContentsManager.untitled_file [Unicode] Default: `'untitled'`

The base name used when creating untitled files.

FileContentsManager.untitled_notebook [Unicode] Default: `'Untitled'`

The base name used when creating untitled notebooks.

FileContentsManager.use_atomic_writing [Bool] Default: `True`

By default notebooks are saved on disk on a temporary file and then if succefully written, it replaces the old ones. This procedure, namely 'atomic_writing', causes some bugs on file system whitout operation order enforcement (like some networked fs). If set to False, the new notebook is written directly on the old one which could fail (eg: full filesystem or quota)

NotebookNotary.algorithm ['md5'|'sha224'|'sha384'|'sha1'|'sha256'|'sha512'] Default: `'sha256'`

The hashing algorithm used to sign notebooks.

NotebookNotary.db_file [Unicode] Default: `''`

The sqlite file in which to store notebook signatures. By default, this will be in your Jupyter data directory. You can set it to ':memory:' to disable sqlite writing to the filesystem.

NotebookNotary.secret [Bytes] Default: `b''`

The secret key with which notebooks are signed.

NotebookNotary.secret_file [Unicode] Default: `''`

The file where the secret key is stored.

NotebookNotary.store_factory [Callable] Default: `traitlets.Undefined`

A callable returning the storage backend for notebook signatures. The default uses an SQLite database.

KernelSpecManager.ensure_native_kernel [Bool] Default: `True`

If there is no Python kernelspec registered and the IPython kernel is available, ensure it is added to the spec list.

KernelSpecManager.kernel_spec_class [Type] Default: `'jupyter_client.kernelspec.KernelSpec'`

The kernel spec class. This is configurable to allow subclassing of the KernelSpecManager for customized behavior.

KernelSpecManager.whitelist [Set] Default: `set()`

Whitelist of allowed kernel names.

By default, all installed kernels are allowed.

CHAPTER 5

Running a notebook server

The *Jupyter notebook* web application is based on a server-client structure. The notebook server uses a two-process kernel architecture based on ZeroMQ, as well as Tornado for serving HTTP requests.

Note: By default, a notebook server runs locally at 127.0.0.1:8888 and is accessible only from *localhost*. You may access the notebook server from the browser using *http://127.0.0.1:8888*.

This document describes how you can *secure a notebook server* and how to *run it on a public interface*.

Important: **This is not the multi-user server you are looking for**. This document describes how you can run a public server with a single user. This should only be done by someone who wants remote access to their personal machine. Even so, doing this requires a thorough understanding of the set-ups limitations and security implications. If you allow multiple users to access a notebook server as it is described in this document, their commands may collide, clobber and overwrite each other.

If you want a multi-user server, the official solution is JupyterHub. To use JupyterHub, you need a Unix server (typically Linux) running somewhere that is accessible to your users on a network. This may run over the public internet, but doing so introduces additional of security concerns.

Securing a notebook server

You can protect your notebook server with a simple single password by configuring the `NotebookApp.password` setting in `jupyter_notebook_config.py`.

Prerequisite: A notebook configuration file

Check to see if you have a notebook configuration file, `jupyter_notebook_config.py`. The default location for this file is your Jupyter folder in your home directory, `~/.jupyter`.

If you don't already have one, create a config file for the notebook using the following command:

```
$ jupyter notebook --generate-config
```

Preparing a hashed password

You can prepare a hashed password using the function `notebook.auth.security.passwd()`:

```
In [1]: from notebook.auth import passwd
In [2]: passwd()
Enter password:
Verify password:
Out[2]: 'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

Caution: `passwd()` when called with no arguments will prompt you to enter and verify your password such as in the above code snippet. Although the function can also be passed a string as an argument such as `passwd('mypassword')`, please **do not** pass a string as an argument inside an IPython session, as it will be saved in your input history.

Adding hashed password to your notebook configuration file

You can then add the hashed password to your `jupyter_notebook_config.py`. The default location for this file `jupyter_notebook_config.py` is in your Jupyter folder in your home directory, `~/.jupyter`, e.g.:

```
c.NotebookApp.password = u'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

Using SSL for encrypted communication

When using a password, it is a good idea to also use SSL with a web certificate, so that your hashed password is not sent unencrypted by your browser.

Important: Web security is rapidly changing and evolving. We provide this document as a convenience to the user, and recommend that the user keep current on changes that may impact security, such as new releases of OpenSSL. The Open Web Application Security Project (OWASP) website is a good resource on general security issues and web practices.

You can start the notebook to communicate via a secure protocol mode by setting the `certfile` option to your self-signed certificate, i.e. `mycert.pem`, with the command:

```
$ jupyter notebook --certfile=mycert.pem --keyfile mykey.key
```

Tip: A self-signed certificate can be generated with `openssl`. For example, the following command will create a certificate valid for 365 days with both the key and certificate data written to the same file:

```
$ openssl req -x509 -nodes -days 365 -newkey rsa:1024 -keyout mykey.key -out mycert.
↪pem
```

When starting the notebook server, your browser may warn that your self-signed certificate is insecure or unrecognized. If you wish to have a fully compliant self-signed certificate that will not raise warnings, it is possible (but rather involved) to create one, as explained in detail in this tutorial.

Running a public notebook server

If you want to access your notebook server remotely via a web browser, you can do so by running a public notebook server. For optimal security when running a public notebook server, you should first secure the server with a password and SSL/HTTPS as described in *Securing a notebook server*.

Start by creating a certificate file and a hashed password, as explained in *Securing a notebook server*.

If you don't already have one, create a config file for the notebook using the following command line:

```
$ jupyter notebook --generate-config
```

In the `~/.jupyter` directory, edit the notebook config file, `jupyter_notebook_config.py`. By default, the notebook config file has all fields commented out. The minimum set of configuration options that you should to uncomment and edit in :file:`jupyter_notebook_config.py` is the following:

```
# Set options for certfile, ip, password, and toggle off browser auto-opening
c.NotebookApp.certfile = u'/absolute/path/to/your/certificate/mycert.pem'
c.NotebookApp.keyfile = u'/absolute/path/to/your/certificate/mykey.key'
# Set ip to '*' to bind on all interfaces (ips) for the public server
c.NotebookApp.ip = '*'
c.NotebookApp.password = u'sha1:bcd259ccf...<your hashed password here>'
c.NotebookApp.open_browser = False

# It is a good idea to set a known, fixed port for server access
c.NotebookApp.port = 9999
```

You can then start the notebook using the `jupyter notebook` command.

Important: Use 'https'. Keep in mind that when you enable SSL support, you must access the notebook server over `https://`, not over plain `http://`. The startup message from the server prints a reminder in the console, but *it is easy to overlook this detail and think the server is for some reason non-responsive*.

When using SSL, always access the notebook server with 'https://'.

You may now access the public server by pointing your browser to `https://your.host.com:9999` where `your.host.com` is your public server's domain.

Firewall Setup

To function correctly, the firewall on the computer running the jupyter notebook server must be configured to allow connections from client machines on the access port `c.NotebookApp.port` set in :file:`jupyter_notebook_config.py` port to allow connections to the web interface. The firewall must also allow connections from 127.0.0.1 (localhost) on ports from 49152 to 65535. These ports are used by the server to communicate with the notebook kernels. The kernel communication ports are chosen randomly by ZeroMQ, and may require multiple connections per kernel, so a large range of ports must be accessible.

Running the notebook with a customized URL prefix

The notebook dashboard, which is the landing page with an overview of the notebooks in your working directory, is typically found and accessed at the default URL `http://localhost:8888/`.

If you prefer to customize the URL prefix for the notebook dashboard, you can do so through modifying `jupyter_notebook_config.py`. For example, if you prefer that the notebook dashboard be located with a subdirectory that contains other ipython files, e.g. `http://localhost:8888/ipython/`, you can do so with configuration options like the following (see above for instructions about modifying `jupyter_notebook_config.py`):

```
c.NotebookApp.base_url = '/ipython/'
c.NotebookApp.webapp_settings = {'static_url_prefix':'/ipython/static/'}
```

Known issues

Proxies

When behind a proxy, especially if your system or browser is set to autodetect the proxy, the notebook web application might fail to connect to the server's websockets, and present you with a warning at startup. In this case, you need to configure your system not to use the proxy for the server's address.

For example, in Firefox, go to the Preferences panel, Advanced section, Network tab, click 'Settings...', and add the address of the notebook server to the 'No proxy for' field.

Docker CMD

Using `jupyter notebook` as a Docker CMD results in kernels repeatedly crashing, likely due to a lack of PID reaping. To avoid this, use the tini `init` as your Dockerfile *ENTRYPOINT*:

```
# Add Tini. Tini operates as a process subreaper for jupyter. This prevents
# kernel crashes.
ENV TINI_VERSION v0.6.0
ADD https://github.com/krallin/tini/releases/download/${TINI_VERSION}/tini /usr/bin/
↪tini
RUN chmod +x /usr/bin/tini
ENTRYPOINT ["/usr/bin/tini", "--"]

EXPOSE 8888
CMD ["jupyter", "notebook", "--port=8888", "--no-browser", "--ip=0.0.0.0"]
```

CHAPTER 6

Security in the Jupyter notebook server

Since access to the Jupyter notebook server means access to running arbitrary code, it is important to restrict access to the notebook server. For this reason, notebook 4.3 introduces token-based authentication that is **on by default**.

Note: If you enable a password for your notebook server, token authentication is not enabled by default, and the behavior of the notebook server is unchanged from from versions earlier than 4.3.

When token authentication is enabled, the notebook uses a token to authenticate requests. This token can be provided to login to the notebook server in three ways:

- in the `Authorization` header, e.g.:

```
Authorization: token abcdef...
```

- In a URL parameter, e.g.:

```
https://my-notebook/tree/?token=abcdef...
```

- In the password field of the login form that will be shown to you if you are not logged in.

When you start a notebook server with token authentication enabled (default), a token is generated to use for authentication. This token is logged to the terminal, so that you can copy/paste the URL into your browser:

```
[I 11:59:16.597 NotebookApp] The Jupyter Notebook is running at: http://
↪localhost:8888/?token=c8de56fa4deed24899803e93c227592aef6538f93025fe01
```

If the notebook server is going to open your browser automatically (the default, unless `--no-browser` has been passed), an *additional* token is generated for launching the browser. This additional token can be used only once, and is used to set a cookie for your browser once it connects. After your browser has made its first request with this one-time-token, the token is discarded and a cookie is set in your browser.

At any later time, you can see the tokens and URLs for all of your running servers with **`jupyter notebook list`**:

```
$ jupyter notebook list
Currently running servers:
http://localhost:8888/?token=abc... :: /home/you/notebooks
https://0.0.0.0:9999/?token=123... :: /tmp/public
http://localhost:8889/ :: /tmp/has-password
```

For servers with token-authentication enabled, the URL in the above listing will include the token, so you can copy and paste that URL into your browser to login. If a server has no token (e.g. it has a password or has authentication disabled), the URL will not include the token argument. Once you have visited this URL, a cookie will be set in your browser and you won't need to use the token again, unless you switch browsers, clear your cookies, or start a notebook server on a new port.

You can disable authentication altogether by setting the token and password to empty strings, but this is **NOT RECOMMENDED**, unless authentication or access restrictions are handled at a different layer in your web application:

```
c.NotebookApp.token = ''
c.NotebookApp.password = ''
```

CHAPTER 7

Security in notebook documents

As Jupyter notebooks become more popular for sharing and collaboration, the potential for malicious people to attempt to exploit the notebook for their nefarious purposes increases. IPython 2.0 introduced a security model to prevent execution of untrusted code without explicit user input.

The problem

The whole point of Jupyter is arbitrary code execution. We have no desire to limit what can be done with a notebook, which would negatively impact its utility.

Unlike other programs, a Jupyter notebook document includes output. Unlike other documents, that output exists in a context that can execute code (via Javascript).

The security problem we need to solve is that no code should execute just because a user has **opened** a notebook that **they did not write**. Like any other program, once a user decides to execute code in a notebook, it is considered trusted, and should be allowed to do anything.

Our security model

- Untrusted HTML is always sanitized
- Untrusted Javascript is never executed
- HTML and Javascript in Markdown cells are never trusted
- **Outputs** generated by the user are trusted
- Any other HTML or Javascript (in Markdown cells, output generated by others) is never trusted
- The central question of trust is “Did the current user do this?”

The details of trust

When a notebook is executed and saved, a signature is computed from a digest of the notebook's contents plus a secret key. This is stored in a database, writable only by the current user. By default, this is located at:

```
~/.local/share/jupyter/nbsignatures.db  # Linux
~/Library/Jupyter/nbsignatures.db       # OS X
%APPDATA%/jupyter/nbsignatures.db       # Windows
```

Each signature represents a series of outputs which were produced by code the current user executed, and are therefore trusted.

When you open a notebook, the server computes its signature, and checks if it's in the database. If a match is found, HTML and Javascript output in the notebook will be trusted at load, otherwise it will be untrusted.

Any output generated during an interactive session is trusted.

Updating trust

A notebook's trust is updated when the notebook is saved. If there are any untrusted outputs still in the notebook, the notebook will not be trusted, and no signature will be stored. If all untrusted outputs have been removed (either via `Clear Output` or re-execution), then the notebook will become trusted.

While trust is updated per output, this is only for the duration of a single session. A newly loaded notebook file is either trusted or not in its entirety.

Explicit trust

Sometimes re-executing a notebook to generate trusted output is not an option, either because dependencies are unavailable, or it would take a long time. Users can explicitly trust a notebook in two ways:

- At the command-line, with:

```
jupyter trust /path/to/notebook.ipynb
```

- After loading the untrusted notebook, with `File / Trust Notebook`

These two methods simply load the notebook, compute a new signature, and add that signature to the user's database.

Reporting security issues

If you find a security vulnerability in Jupyter, either a failure of the code to properly implement the model described here, or a failure of the model itself, please report it to security@ipython.org.

If you prefer to encrypt your security reports, you can use `this PGP public key`.

Affected use cases

Some use cases that work in Jupyter 1.0 became less convenient in 2.0 as a result of the security changes. We do our best to minimize these annoyances, but security is always at odds with convenience.

Javascript and CSS in Markdown cells

While never officially supported, it had become common practice to put hidden Javascript or CSS styling in Markdown cells, so that they would not be visible on the page. Since Markdown cells are now sanitized (by Google Caja), all Javascript (including click event handlers, etc.) and CSS will be stripped.

We plan to provide a mechanism for notebook themes, but in the meantime styling the notebook can only be done via either `custom.css` or CSS in HTML output. The latter only have an effect if the notebook is trusted, because otherwise the output will be sanitized just like Markdown.

Collaboration

When collaborating on a notebook, people probably want to see the outputs produced by their colleagues' most recent executions. Since each collaborator's key will differ, this will result in each share starting in an untrusted state. There are three basic approaches to this:

- re-run notebooks when you get them (not always viable)
- explicitly trust notebooks via `jupyter trust` or the notebook menu (annoying, but easy)
- share a notebook signatures database, and use configuration dedicated to the collaboration while working on the project.

To share a signatures database among users, you can configure:

```
c.NotebookNotary.data_dir = "/path/to/signature_dir"
```

to specify a non-default path to the SQLite database (of notebook hashes, essentially). We are aware that SQLite doesn't work well on NFS and we are working out better ways to do this.

CHAPTER 8

Configuring the notebook frontend

Note: The ability to configure the notebook frontend UI and preferences is still a work in progress.

This document is a rough explanation on how you can persist some configuration options for the notebook JavaScript.

There is no exhaustive list of all the configuration options as most options are passed down to other libraries, which means that non valid configuration can be ignored without any error messages.

How front end configuration works

The frontend configuration system works as follows:

- get a handle of a configurable JavaScript object.
- access its configuration attribute.
- update its configuration attribute with a JSON patch.

Example - Changing the notebook's default indentation

This example explains how to change the default setting `indentUnit` for CodeMirror Code Cells:

```
var cell = Jupyter.notebook.get_selected_cell();
var config = cell.config;
var patch = {
      CodeCell:{
        cm_config:{indentUnit:2}
      }
    }
config.update(patch)
```

You can enter the previous snippet in your browser's JavaScript console once. Then reload the notebook page in your browser. Now, the preferred indent unit should be equal to two spaces. The custom setting persists and you do not need to reissue the patch on new notebooks.

`indentUnit`, used in this example, is one of the many CodeMirror options which are available for configuration.

Example - Restoring the notebook's default indentation

If you want to restore a notebook frontend preference to its default value, you will enter a JSON patch with a `null` value for the preference setting.

For example, let's restore the indent setting `indentUnit` to its default of four spaces. Enter the following code snippet in your JavaScript console:

```
var cell = Jupyter.notebook.get_selected_cell();
var config = cell.config;
var patch = {
      CodeCell:{
        cm_config:{indentUnit: null} # only change here.
      }
    }
config.update(patch)
```

Reload the notebook in your browser and the default indent should again be two spaces.

Persisting configuration settings

Under the hood, Jupyter will persist the preferred configuration settings in `~/.jupyter/nbconfig/<section>.json`, with `<section>` taking various value depending on the page where the configuration is issued. `<section>` can take various values like `notebook`, `tree`, and `editor`. A `common` section contains configuration settings shared by all pages.

CHAPTER 9

Extending the Notebook

Certain subsystems of the notebook server are designed to be extended or overridden by users. These documents explain these systems, and show how to override the notebook's defaults with your own custom behavior.

Contents API

The Jupyter Notebook web application provides a graphical interface for creating, opening, renaming, and deleting files in a virtual filesystem.

The `ContentsManager` class defines an abstract API for translating these interactions into operations on a particular storage medium. The default implementation, `FileContentsManager`, uses the local filesystem of the server for storage and straightforwardly serializes notebooks into JSON. Users can override these behaviors by supplying custom subclasses of ContentsManager.

This section describes the interface implemented by ContentsManager subclasses. We refer to this interface as the **Contents API**.

Data Model

Filesystem Entities

ContentsManager methods represent virtual filesystem entities as dictionaries, which we refer to as **models**.

Models may contain the following entries:

Key	Type	Info
name	unicode	Basename of the entity.
path	unicode	Full (*API-style*) path to the entity.
type	unicode	The entity type. One of `"notebook"`, `"file"` or `"directory"`.
created	datetime	Creation date of the entity.
last_modified	datetime	Last modified date of the entity.
content	variable	The "content" of the entity. (*See Below*)
mimetype	unicode or `None`	The mimetype of `content`, if any. (*See Below*)
format	unicode or `None`	The format of `content`, if any. (*See Below*)

Certain model fields vary in structure depending on the `type` field of the model. There are three model types: **notebook**, **file**, and **directory** .

- **`notebook` models**
  - The `format` field is always `"json"`.
  - The `mimetype` field is always `None`.
  - The `content` field contains a `nbformat.notebooknode.NotebookNode` representing the .ipynb file represented by the model. See the NBFormat documentation for a full description.
- **`file` models**
  - The `format` field is either `"text"` or `"base64"`.
  - The `mimetype` field is `text/plain` for text-format models and `application/octet-stream` for base64-format models.
  - The `content` field is always of type `unicode`. For text-format file models, `content` simply contains the file's bytes after decoding as UTF-8. Non-text (`base64`) files are read as bytes, base64 encoded, and then decoded as UTF-8.
- **`directory` models**
  - The `format` field is always `"json"`.
  - The `mimetype` field is always `None`.
  - The `content` field contains a list of *content-free* models representing the entities in the directory.

Note: In certain circumstances, we don't need the full content of an entity to complete a Contents API request. In such cases, we omit the `mimetype`, `content`, and `format` keys from the model. This most commonly occurs when listing a directory, in which circumstance we represent files within the directory as content-less models to avoid having to recursively traverse and serialize the entire filesystem.

Sample Models

```
# Notebook Model with Content
{
    'content': {
        'metadata': {},
        'nbformat': 4,
        'nbformat_minor': 0,
        'cells': [
            {
                'cell_type': 'markdown',
                'metadata': {},
                'source': 'Some **Markdown**',
            },
```

```
        ],
    },
    'created': datetime(2015, 7, 25, 19, 50, 19, 19865),
    'format': 'json',
    'last_modified': datetime(2015, 7, 25, 19, 50, 19, 19865),
    'mimetype': None,
    'name': 'a.ipynb',
    'path': 'foo/a.ipynb',
    'type': 'notebook',
    'writable': True,
}

# Notebook Model without Content
{
    'content': None,
    'created': datetime.datetime(2015, 7, 25, 20, 17, 33, 271931),
    'format': None,
    'last_modified': datetime.datetime(2015, 7, 25, 20, 17, 33, 271931),
    'mimetype': None,
    'name': 'a.ipynb',
    'path': 'foo/a.ipynb',
    'type': 'notebook',
    'writable': True
}
```

API Paths

ContentsManager methods represent the locations of filesystem resources as **API-style paths**. Such paths are interpreted as relative to the root directory of the notebook server. For compatibility across systems, the following guarantees are made:

- Paths are always `unicode`, not `bytes`.
- Paths are not URL-escaped.
- Paths are always forward-slash (/) delimited, even on Windows.
- Leading and trailing slashes are stripped. For example, `/foo/bar/buzz/` becomes `foo/bar/buzz`.
- The empty string (`""`) represents the root directory.

Writing a Custom ContentsManager

The default ContentsManager is designed for users running the notebook as an application on a personal computer. It stores notebooks as .ipynb files on the local filesystem, and it maps files and directories in the Notebook UI to files and directories on disk. It is possible to override how notebooks are stored by implementing your own custom subclass of `ContentsManager`. For example, if you deploy the notebook in a context where you don't trust or don't have access to the filesystem of the notebook server, it's possible to write your own ContentsManager that stores notebooks and files in a database.

Required Methods

A minimal complete implementation of a custom `ContentsManager` must implement the following methods:

Continued on next page

Table 9.1 – continued from previous page

ContentsManager.get(path[, content, type, ...])	Get a file or directory model.
ContentsManager.save(model, path)	Save a file or directory model to path.
ContentsManager.delete_file(path)	Delete the file or directory at path.
ContentsManager.rename_file(old_path, new_path)	Rename a file or directory.
ContentsManager.file_exists([path])	Does a file exist at the given path?
ContentsManager.dir_exists(path)	Does a directory exist at the given path?
ContentsManager.is_hidden(path)	Is path a hidden directory or file?

Customizing Checkpoints

TODO:

Testing

`notebook.services.contents.tests` includes several test suites written against the abstract Contents API. This means that an excellent way to test a new ContentsManager subclass is to subclass our tests to make them use your ContentsManager.

Note: PGContents is an example of a complete implementation of a custom `ContentsManager`. It stores notebooks and files in PostgreSQL and encodes directories as SQL relations. PGContents also provides an example of how to re-use the notebook's tests.

File save hooks

You can configure functions that are run whenever a file is saved. There are two hooks available:

- `ContentsManager.pre_save_hook` runs on the API path and model with content. This can be used for things like stripping output that people don't like adding to VCS noise.
- `FileContentsManager.post_save_hook` runs on the filesystem path and model without content. This could be used to commit changes after every save, for instance.

They are both called with keyword arguments:

```
pre_save_hook(model=model, path=path, contents_manager=cm)
post_save_hook(model=model, os_path=os_path, contents_manager=cm)
```

Examples

These can both be added to `jupyter_notebook_config.py`.

A pre-save hook for stripping output:

```
def scrub_output_pre_save(model, **kwargs):
    """scrub output before saving notebooks"""
    # only run on notebooks
    if model['type'] != 'notebook':
```

```
        return
    # only run on nbformat v4
    if model['content']['nbformat'] != 4:
        return

    for cell in model['content']['cells']:
        if cell['cell_type'] != 'code':
            continue
        cell['outputs'] = []
        cell['execution_count'] = None

c.FileContentsManager.pre_save_hook = scrub_output_pre_save
```

A post-save hook to make a script equivalent whenever the notebook is saved (replacing the `--script` option in older versions of the notebook):

```
import io
import os
from notebook.utils import to_api_path

_script_exporter = None

def script_post_save(model, os_path, contents_manager, **kwargs):
    """convert notebooks to Python script after save with nbconvert

    replaces `ipython notebook --script`
    """
    from nbconvert.exporters.script import ScriptExporter

    if model['type'] != 'notebook':
        return

    global _script_exporter
    if _script_exporter is None:
        _script_exporter = ScriptExporter(parent=contents_manager)
    log = contents_manager.log

    base, ext = os.path.splitext(os_path)
    py_fname = base + '.py'
    script, resources = _script_exporter.from_filename(os_path)
    script_fname = base + resources.get('output_extension', '.txt')
    log.info("Saving script /%s", to_api_path(script_fname, contents_manager.root_
↪dir))
    with io.open(script_fname, 'w', encoding='utf-8') as f:
        f.write(script)
c.FileContentsManager.post_save_hook = script_post_save
```

This could be a simple call to `jupyter nbconvert --to script`, but spawning the subprocess every time is quite slow.

Custom request handlers

The notebook webserver can be interacted with using a well defined RESTful API. You can define custom RESTful API handlers in addition to the ones provided by the notebook. As described below, to define a custom handler you need to first write a notebook server extension. Then, in the extension, you can register the custom handler.

Writing a notebook server extension

The notebook webserver is written in Python, hence your server extension should be written in Python too. Server extensions, like IPython extensions, are Python modules that define a specially named load function, `load_jupyter_server_extension`. This function is called when the extension is loaded.

```
def load_jupyter_server_extension(nb_server_app):
    """
    Called when the extension is loaded.

    Args:
        nb_server_app (NotebookWebApplication): handle to the Notebook webserver␣
↪instance.
    """
    pass
```

To get the notebook server to load your custom extension, you'll need to add it to the list of extensions to be loaded. You can do this using the config system. `NotebookApp.server_extensions` is a config variable which is an array of strings, each a Python module to be imported. Because this variable is notebook config, you can set it two different ways, using config files or via the command line.

For example, to get your extension to load via the command line add a double dash before the variable name, and put the Python array in double quotes. If your package is "mypackage" and module is "mymodule", this would look like `jupyter notebook --NotebookApp.server_extensions="['mypackage.mymodule']"`. Basically the string should be Python importable.

Alternatively, you can have your extension loaded regardless of the command line args by setting the variable in the Jupyter config file. The default location of the Jupyter config file is `~/.jupyter/profile_default/jupyter_notebook_config.py`. Then, inside the config file, you can use Python to set the variable. For example, the following config does the same as the previous command line example [1].

```
c = get_config()
c.NotebookApp.server_extensions = [
    'mypackage.mymodule'
]
```

Before continuing, it's a good idea to verify that your extension is being loaded. Use a print statement to print something unique. Launch the notebook server and you should see your statement printed to the console.

Registering custom handlers

Once you've defined a server extension, you can register custom handlers because you have a handle to the Notebook server app instance (`nb_server_app` above). However, you first need to define your custom handler. To declare a custom handler, inherit from `notebook.base.handlers.IPythonHandler`. The example below[1] is a Hello World handler:

```
from notebook.base.handlers import IPythonHandler

class HelloWorldHandler(IPythonHandler):
    def get(self):
        self.finish('Hello, world!')
```

The Jupyter Notebook server use Tornado as its web framework. For more information on how to implement request handlers, refer to the Tornado documentation on the matter.

After defining the handler, you need to register the handler with the Notebook server. See the following example:

```
web_app = nb_server_app.web_app
host_pattern = '.*$'
route_pattern = url_path_join(web_app.settings['base_url'], '/hello')
web_app.add_handlers(host_pattern, [(route_pattern, HelloWorldHandler)])
```

Putting this together with the extension code, the example looks like the following:

```
from notebook.utils import url_path_join
from notebook.base.handlers import IPythonHandler

class HelloWorldHandler(IPythonHandler):
    def get(self):
        self.finish('Hello, world!')

def load_jupyter_server_extension(nb_server_app):
    """
    Called when the extension is loaded.

    Args:
        nb_server_app (NotebookWebApplication): handle to the Notebook webserver␣
↪instance.
    """
    web_app = nb_server_app.web_app
    host_pattern = '.*$'
    route_pattern = url_path_join(web_app.settings['base_url'], '/hello')
    web_app.add_handlers(host_pattern, [(route_pattern, HelloWorldHandler)])
```

References: 1. Peter Parente's Mindtrove

Custom front-end extensions

This describes the basic steps to write a JavaScript extension for the Jupyter notebook front-end. This allows you to customize the behaviour of the various pages like the dashboard, the notebook, or the text editor.

The structure of a front-end extension

Note: The notebook front-end and Javascript API are not stable, and are subject to a lot of changes. Any extension written for the current notebook is almost guaranteed to break in the next release.

A front-end extension is a JavaScript file that defines an AMD module which exposes at least a function called `load_ipython_extension`, which takes no arguments. We will not get into the details of what each of these terms consists of yet, but here is the minimal code needed for a working extension:

```
// file my_extension/main.js

define(function(){

    function load_ipython_extension(){
        console.info('this is my first extension');
    }

    return {
```

```
        load_ipython_extension: load_ipython_extension
    };
});
```

Note: Although for historical reasons the function is called `load_ipython_extension`, it does apply to the Jupyter notebook in general, and will work regardless of the kernel in use.

If you are familiar with JavaScript, you can use this template to require any Jupyter module and modify its configuration, or do anything else in client-side Javascript. Your extension will be loaded at the right time during the notebook page initialisation for you to set up a listener for the various events that the page can trigger.

You might want access to the current instances of the various Jupyter notebook components on the page, as opposed to the classes defined in the modules. The current instances are exposed by a module named `base/js/namespace`. If you plan on accessing instances on the page, you should `require` this module rather than accessing the global variable `Jupyter`, which will be removed in future. The following example demonstrates how to access the current notebook instance:

```
// file my_extension/main.js

define([
    'base/js/namespace'
], function(
    Jupyter
) {
    function load_ipython_extension() {
        console.log(
            'This is the current notebook application instance:',
            Jupyter.notebook
        );
    }

    return {
        load_ipython_extension: load_ipython_extension
    };
});
```

Modifying key bindings

One of the abilities of extensions is to modify key bindings, although once again this is an API which is not guaranteed to be stable. However, custom key bindings are frequently requested, and are helpful to increase accessibility, so in the following we show how to access them.

Here is an example of an extension that will unbind the shortcut `0,0` in command mode, which normally restarts the kernel, and bind `0,0,0` in its place:

```
// file my_extension/main.js

define([
    'base/js/namespace'
], function(
    Jupyter
) {

    function load_ipython_extension() {
```

```
        Jupyter.keyboard_manager.command_shortcuts.remove_shortcut('0,0');
        Jupyter.keyboard_manager.command_shortcuts.add_shortcut('0,0,0', 'jupyter-
↪notebook:restart-kernel');
    }

    return {
        load_ipython_extension: load_ipython_extension
    };
});
```

Note: The standard keybindings might not work correctly on non-US keyboards. Unfortunately, this is a limitation of browser implementations and the status of keyboard event handling on the web in general. We appreciate your feedback if you have issues binding keys, or have any ideas to help improve the situation.

You can see that I have used the **action name** `jupyter-notebook:restart-kernel` to bind the new shortcut. There is no API yet to access the list of all available *actions*, though the following in the JavaScript console of your browser on a notebook page should give you an idea of what is available:

```
Object.keys(require('base/js/namespace').actions._actions);
```

In this example, we changed a keyboard shortcut in **command mode**; you can also customize keyboard shortcuts in **edit mode**. However, most of the keyboard shortcuts in edit mode are handled by CodeMirror, which supports custom key bindings via a completely different API.

Defining and registering your own actions

As part of your front-end extension, you may wish to define actions, which can be attached to toolbar buttons, or called from the command palette. Here is an example of an extension that defines a (not very useful!) action to show an alert, and adds a toolabr button using the full action name:

```
// file my_extension/main.js

define([
    'base/js/namespace'
], function(
    Jupyter
) {
    function load_ipython_extension() {

        var handler = function () {
            alert('this is an alert from my_extension!');
        };

        var action = {
            icon: 'fa-comment-o', // a font-awesome class used on buttons, etc
            help    : 'Show an alert',
            help_index : 'zz',
            handler : handler
        };
        var prefix = 'my_extension';
        var action_name = 'show-alert';

        var full_action_name = Jupyter.actions.register(action, name, prefix); //␣
↪returns 'my_extension:show-alert'
```

```
        Jupyter.toolbar.add_buttons_group([full_action_name]);
    }

    return {
        load_ipython_extension: load_ipython_extension
    };
});
```

Every action needs a name, which, when joined with its prefix to make the full action name, should be unique. Built-in actions, like the `jupyter-notebook:restart-kernel` we bound in the earlier *Modifying key bindings* example, use the prefix `jupyter-notebook`. For actions defined in an extension, it makes sense to use the extension name as the prefix. For the action name, the following guidelines should be considered:

- First pick a noun and a verb for the action. For example, if the action is "restart kernel," the verb is "restart" and the noun is "kernel".
- Omit terms like "selected" and "active" by default, so "delete-cell", rather than "delete-selected-cell". Only provide a scope like "-all-" if it is other than the default "selected" or "active" scope.
- If an action has a secondary action, separate the secondary action with "-and-", so "restart-kernel-and-clear-output".
- Use above/below or previous/next to indicate spatial and sequential relationships.
- Don't ever use before/after as they have a temporal connotation that is confusing when used in a spatial context.
- For dialogs, use a verb that indicates what the dialog will accomplish, such as "confirm-restart-kernel".

Installing and enabling extensions

You can install your nbextension with the command:

```
jupyter nbextension install path/to/my_extension/ [--user|--sys-prefix]
```

The default installation is system-wide. You can use `--user` to do a per-user installation, or `--sys-prefix` to install to Python's prefix (e.g. in a virtual or conda environment). Where my_extension is the directory containing the Javascript files. This will copy it to a Jupyter data directory (the exact location is platform dependent - see Data files).

For development, you can use the `--symlink` flag to symlink your extension rather than copying it, so there's no need to reinstall after changes.

To use your extension, you'll also need to **enable** it, which tells the notebook interface to load it. You can do that with another command:

```
jupyter nbextension enable my_extension/main [--sys-prefix]
```

The argument refers to the Javascript module containing your `load_ipython_extension` function, which is `my_extension/main.js` in this example. There is a corresponding `disable` command to stop using an extension without uninstalling it.

Changed in version 4.2: Added `--sys-prefix` argument

CHAPTER 10

Installing Javascript machinery

Running the Notebook from the source code on GitHub requires some JavaScript tools to build/minify the CSS and JavaScript components. We do these steps when making releases, so there's no need for these tools when installing released versions of the Notebook.

First, install Node.js. The installers on the Node.js website also include Node's package manager, *npm*. Alternatively, install both of these from your package manager. For example, on Ubuntu or Debian:

```
sudo apt-get install nodejs-legacy npm
```

You can then build the JavaScript and CSS by running:

```
python setup.py css js
```

This will automatically fetch the remaining dependencies (bower, less) and install them in a subdirectory.

For quick iteration on the Notebook's JavaScript you can deactivate the use of the bundled and minified JavaScript by using the option `--NotebookApp.ignore_minified_js=True`. This might though highly increase the number of requests that the browser make to the server, but can allow to test JavaScript file modification without going through the compilation step that can take up to 30 sec.

Making a notebook release

Make sure you have followed the step above and have all the tools to generate the minified JavaScript and CSS files.

Make sure the repository is clean of any file that could be problematic. You can remove all non-tracked files with:

```
$ git clean -xfdi
```

This would ask you for confirmation before removing all untracked files. Make sure the `dist/` folder is clean and avoid stale build from previous attempts.

1. Update version number in `notebook/_version.py`.

2. Run `$ python setup.py jsversion`. It will modify (at least) `notebook/static/base/js/namespace.js` to make the notebook version available from within JavaScript.

3 . Commit and tag the release with the current version number:

```
git commit -am "release $VERSION"
git tag $VERSION
```

4. You are now ready to build the `sdist` and `wheel`:

```
$ python setup.py sdist --formats=zip,gztar
$ python setup.py bdist_wheel
```

5. You can now test the `wheel` and the `sdist` locally before uploading to PyPI. Make sure to use twine to upload the archives over SSL.

```
$ twine upload dist/*
```

6. If all went well, change the `notebook/_version.py` back adding the `.dev` suffix.

7. Push directly on master, not forgetting to push `--tags`.

CHAPTER 11

Developer FAQ

1. How do I install a pre-release version such as a beta or release candidate?

   ```
   python -m pip install notebook --pre --upgrade
   ```

View the original notebook on nbviewer

CHAPTER 12

Distributing Jupyter Extensions as Python Packages

Overview

How can the notebook be extended?

The Jupyter Notebook client and server application are both deeply customizable. Their behavior can be extended by creating, respectively:

- nbextension: a notebook extension
  - a single JS file, or directory of JavaScript, Cascading StyleSheets, etc. that contain at minimum a JavaScript module packaged as an AMD modules that exports a function `load_ipython_extension`
- server extension: an importable Python module
  - that implements `load_jupyter_server_extension`

Why create a Python package for Jupyter extensions?

Since it is rare to have a server extension that does not have any frontend components (an nbextension), for convenience and consistency, all these client and server extensions with their assets can be packaged and versioned together as a Python package with a few simple commands. This makes installing the package of extensions easier and less error-prone for the user.

Installation of Jupyter Extensions

Install a Python package containing Jupyter Extensions

There are several ways that you may get a Python package containing Jupyter Extensions. Commonly, you will use a package manager for your system:

```
pip install helpful_package
# or
conda install helpful_package
# or
apt-get install helpful_package

# where 'helpful_package' is a Python package containing one or more Jupyter␣
↪Extensions
```

Enable a Server Extension

The simplest case would be to enable a server extension which has no frontend components.

A `pip` user that wants their configuration stored in their home directory would type the following command:

```
jupyter serverextension enable --py helpful_package
```

Alternatively, a `virtualenv` or `conda` user can pass `--sys-prefix` which keeps their environment isolated and reproducible. For example:

```
# Make sure that your virtualenv or conda environment is activated
[source] activate my-environment

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

Install the nbextension assets

If a package also has an nbextension with frontend assets that must be available (but not neccessarily enabled by default), install these assets with the following command:

```
jupyter nbextension install --py helpful_package # or --sys-prefix if using␣
↪virtualenv or conda
```

Enable nbextension assets

If a package has assets that should be loaded every time a Jupyter app (e.g. lab, notebook, dashboard, terminal) is loaded in the browser, the following command can be used to enable the nbextension:

```
jupyter nbextension enable --py helpful_package # or --sys-prefix if using virtualenv␣
↪or conda
```

Did it work? Check by listing Jupyter Extensions.

After running one or more extension installation steps, you can list what is presently known about nbextensions or server extension. The following commands will list which extensions are available, whether they are enabled, and other extension details:

```
jupyter nbextension list
jupyter serverextension list
```

Additional resources on creating and distributing packages

Of course, in addition to the files listed, there are number of other files one needs to build a proper package. Here are some good resources: - The Hitchhiker's Guide to Packaging - Repository Structure and Python by Kenneth Reitz

How you distribute them, too, is important: - Packaging and Distributing Projects - conda: Building packages

Here are some tools to get you started: - generator-nbextension

Example - Server extension

Creating a Python package with a server extension

Here is an example of a python module which contains a server extension directly on itself. It has this directory structure:

```
- setup.py
- MANIFEST.in
- my_module/
  - __init__.py
```

Defining the server extension

This example shows that the server extension and its `load_jupyter_server_extension` function are defined in the `__init__.py` file.

my_module/__init__.py

```
def _jupyter_server_extension_paths():
    return [{
        "module": "my_module"
    }]


def load_jupyter_server_extension(nbapp):
    nbapp.log.info("my module enabled!")
```

Install and enable the server extension

Which a user can install with:

```
jupyter serverextension enable --py my_module [--sys-prefix]
```

Example - Server extension and nbextension

Creating a Python package with a server extension and nbextension

Here is another server extension, with a front-end module. It assumes this directory structure:

```
- setup.py
- MANIFEST.in
- my_fancy_module/
  - __init__.py
  - static/
    index.js
```

Defining the server extension and nbextension

This example again shows that the server extension and its `load_jupyter_server_extension` function are defined in the `__init__.py` file. This time, there is also a function `_jupyter_nbextension_path` for the nbextension.

my_fancy_module/__init__.py

```
def _jupyter_server_extension_paths():
    return [{
        "module": "my_fancy_module"
    }]

# Jupyter Extension points
def _jupyter_nbextension_paths():
    return [dict(
        section="notebook",
        # the path is relative to the `my_fancy_module` directory
        src="static",
        # directory in the `nbextension/` namespace
        dest="my_fancy_module",
        # _also_ in the `nbextension/` namespace
        require="my_fancy_module/index")]

def load_jupyter_server_extension(nbapp):
    nbapp.log.info("my module enabled!")
```

Install and enable the server extension and nbextension

The user can install and enable the extensions with the following set of commands:

```
jupyter nbextension install --py my_fancy_module [--sys-prefix|--user]
jupyter nbextension enable --py my_fancy_module [--sys-prefix|--system]
jupyter serverextension enable --py my_fancy_module [--sys-prefix|--system]
```

View the original notebook on nbviewer

View the original notebook on nbviewer

CHAPTER 13

Examples and Tutorials

This portion of the documentation was generated from notebook files. You can download the original interactive notebook files using the links at the tops and bottoms of the pages.

Tutorials

- What is the Jupyter Notebook
- Notebook Basics
- Running Code
- Working With Markdown Cells
- Custom Keyboard Shortcuts
- JavaScript Notebook Extensions

Examples

- Importing Notebooks
- Connecting with the Qt Console
- Typesetting Equations

View the original notebook on nbviewer

CHAPTER 14

Jupyter notebook changelog

A summary of changes in the Jupyter notebook. For more detailed information, see GitHub.

Tip: Use `pip install notebook --upgrade` or `conda upgrade notebook` to upgrade to the latest release.

4.4.0

- Allow override of output callbacks to redirect output messages. This is used to implement the ipywidgets Output widget, for example.
- Fix an async bug in message handling by allowing comm message handlers to return a promise which halts message processing until the promise resolves.

See the 4.4 milestone on GitHub for a complete list of issues and pull requests involved in this release.

4.3.2

4.3.2 is a patch release with a bug fix for CodeMirror and improved handling of the "editable" cell metadata field.

- Monkey-patch for CodeMirror that resolves #2037 without breaking #1967
- Read-only (`"editable": false`) cells can be executed but cannot be split, merged, or deleted

See the 4.3.2 milestone on GitHub for a complete list of issues and pull requests involved in this release.

4.3.1

4.3.1 is a patch release with a security patch, a couple bug fixes, and improvements to the newly-released token authentication.

Security fix:

- CVE-2016-9971. Fix CSRF vulnerability, where malicious forms could create untitled files and start kernels (no remote execution or modification of existing files) for users of certain browsers (Firefox, Internet Explorer / Edge). All previous notebook releases are affected.

Bug fixes:

- Fix carriage return handling
- Make the font size more robust against fickle browsers
- Ignore resize events that bubbled up and didn't come from window
- Add Authorization to allowed CORS headers
- Downgrade CodeMirror to 5.16 while we figure out issues in Safari

Other improvements:

- Better docs for token-based authentication
- Further highlight token info in log output when autogenerated

See the 4.3.1 milestone on GitHub for a complete list of issues and pull requests involved in this release.

4.3.0

4.3 is a minor release with many bug fixes and improvements. The biggest user-facing change is the addition of token authentication, which is enabled by default. A token is generated and used when your browser is opened automatically, so you shouldn't have to enter anything in the default circumstances. If you see a login page (e.g. by switching browsers, or launching on a new port with `--no-browser`), you get a login URL with the token from the command `jupyter notebook list`, which you can paste into your browser.

Highlights:

- API for creating mime-type based renderer extensions using `OutputArea.register_mime_type` and `Notebook.render_cell_output` methods. See mimerender-cookiecutter for reference implementations and cookiecutter.
- Enable token authentication by default. See *Security in the Jupyter notebook server* for more details.
- Update security docs to reflect new signature system
- Switched from term.js to xterm.js

Bug fixes:

- Ensure variable is set if exc_info is falsey
- Catch and log handler exceptions in `events.trigger`
- Add debug log for static file paths
- Don't check origin on token-authenticated requests
- Remove leftover print statement
- Fix highlighting of Python code blocks

- `json_errors` should be outermost decorator on API handlers
- Fix remove old nbserver info files
- Fix notebook mime type on download links
- Fix carriage symbol bahvior
- Fix terminal styles
- Update dead links in docs
- If kernel is broken, start a new session
- Include cross-origin check when allowing login URL redirects

Other improvements:

- Allow JSON output data with mime type "application/*+json"
- Allow kernelspecs to have spaces in them for backward compat
- Allow websocket connections from scripts
- Allow `None` for post_save_hook
- Upgrade CodeMirror to 5.21
- Upgrade xterm to 2.1.0
- Docs for using comms
- Set `dirty` flag when output arrives
- Set `ws-url` data attribute when accessing a notebook terminal
- Add base aliases for nbextensions
- Include `@` operator in CodeMirror IPython mode
- Extend mathjax_url docstring
- Load nbextension in predictable order
- Improve the error messages for nbextensions
- Include cross-origin check when allowing login URL redirects

See the 4.3 milestone on GitHub for a complete list of issues and pull requests involved in this release.

4.2.3

4.2.3 is a small bugfix release on 4.2.

Highlights:

- Fix regression in 4.2.2 that delayed loading custom.js until after `notebook_loaded` and `app_initialized` events have fired.
- Fix some outdated docs and links.

See also:

4.2.3 on GitHub.

4.2.2

4.2.2 is a small bugfix release on 4.2, with an important security fix. All users are strongly encouraged to upgrade to 4.2.2.

Highlights:

- **Security fix**: CVE-2016-6524, where untrusted latex output could be added to the page in a way that could execute javascript.
- Fix missing POST in OPTIONS responses.
- Fix for downloading non-ascii filenames.
- Avoid clobbering ssl_options, so that users can specify more detailed SSL configuration.
- Fix inverted load order in nbconfig, so user config has highest priority.
- Improved error messages here and there.

See also:

4.2.2 on GitHub.

4.2.1

4.2.1 is a small bugfix release on 4.2. Highlights:

- Compatibility fixes for some versions of ipywidgets
- Fix for ignored CSS on Windows
- Fix specifying destination when installing nbextensions

See also:

4.2.1 on GitHub.

4.2.0

Release 4.2 adds a new API for enabling and installing extensions. Extensions can now be enabled at the system-level, rather than just per-user. An API is defined for installing directly from a Python package, as well.

See also:

Distributing Jupyter Extensions as Python Packages

Highlighted changes:

- Upgrade MathJax to 2.6 to fix vertical-bar appearing on some equations.
- Restore ability for notebook directory to be root (4.1 regression)
- Large outputs are now throttled, reducing the ability of output floods to kill the browser.
- Fix the notebook ignoring cell executions while a kernel is starting by queueing the messages.
- Fix handling of url prefixes (e.g. JupyterHub) in terminal and edit pages.
- Support nested SVGs in output.

And various other fixes and improvements.

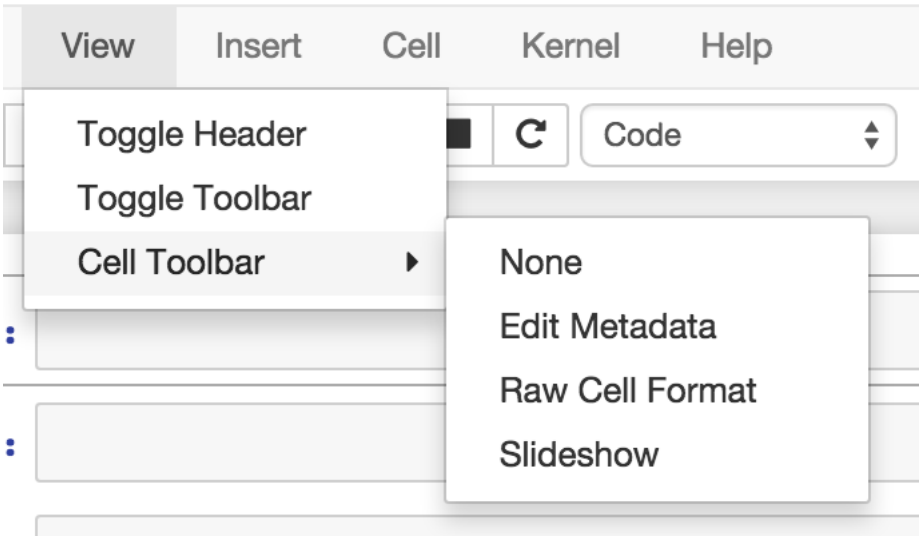
4.1.0

Bug fixes:

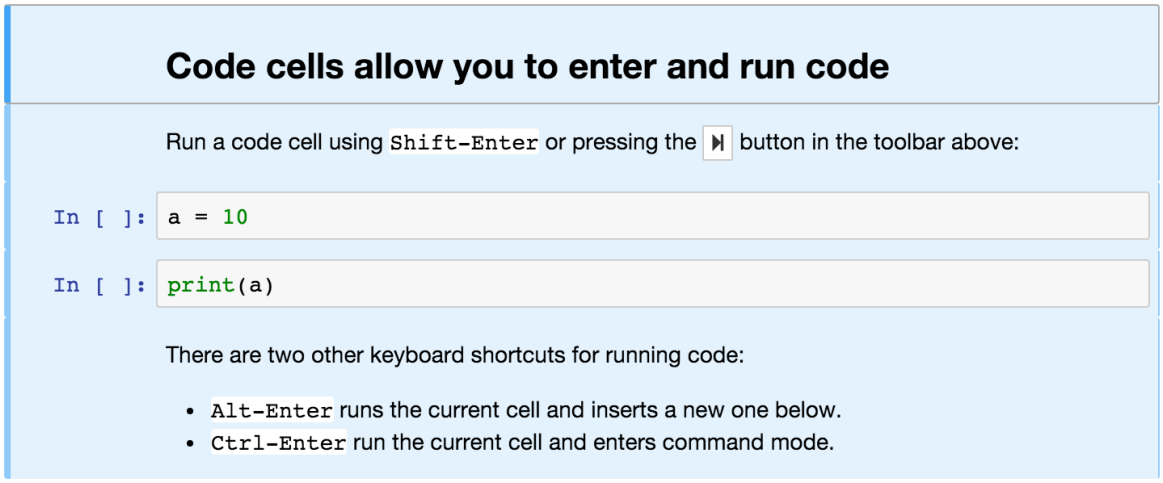
- Properly reap zombie subprocesses
- Fix cross-origin problems
- Fix double-escaping of the base URL prefix
- Handle invalid unicode filenames more gracefully
- Fix ANSI color-processing
- Send keepalive messages for web terminals
- Fix bugs in the notebook tour

UI changes:

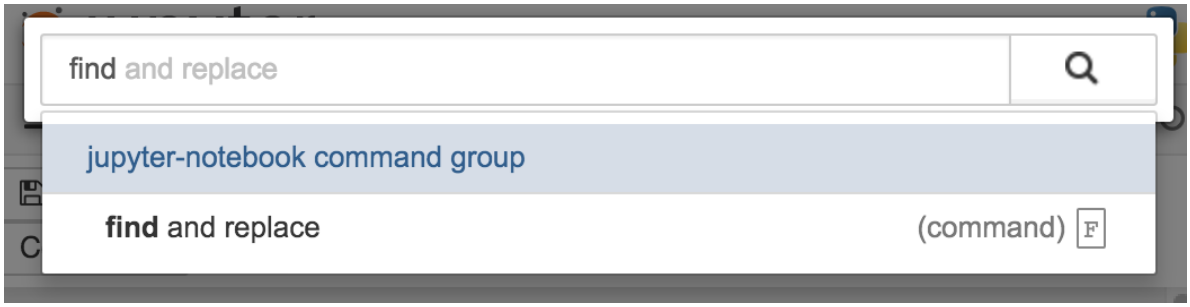
- Moved the cell toolbar selector into the *View* menu. Added a button that triggers a "hint" animation to the main toolbar so users can find the new location. (Click here to see a screencast)



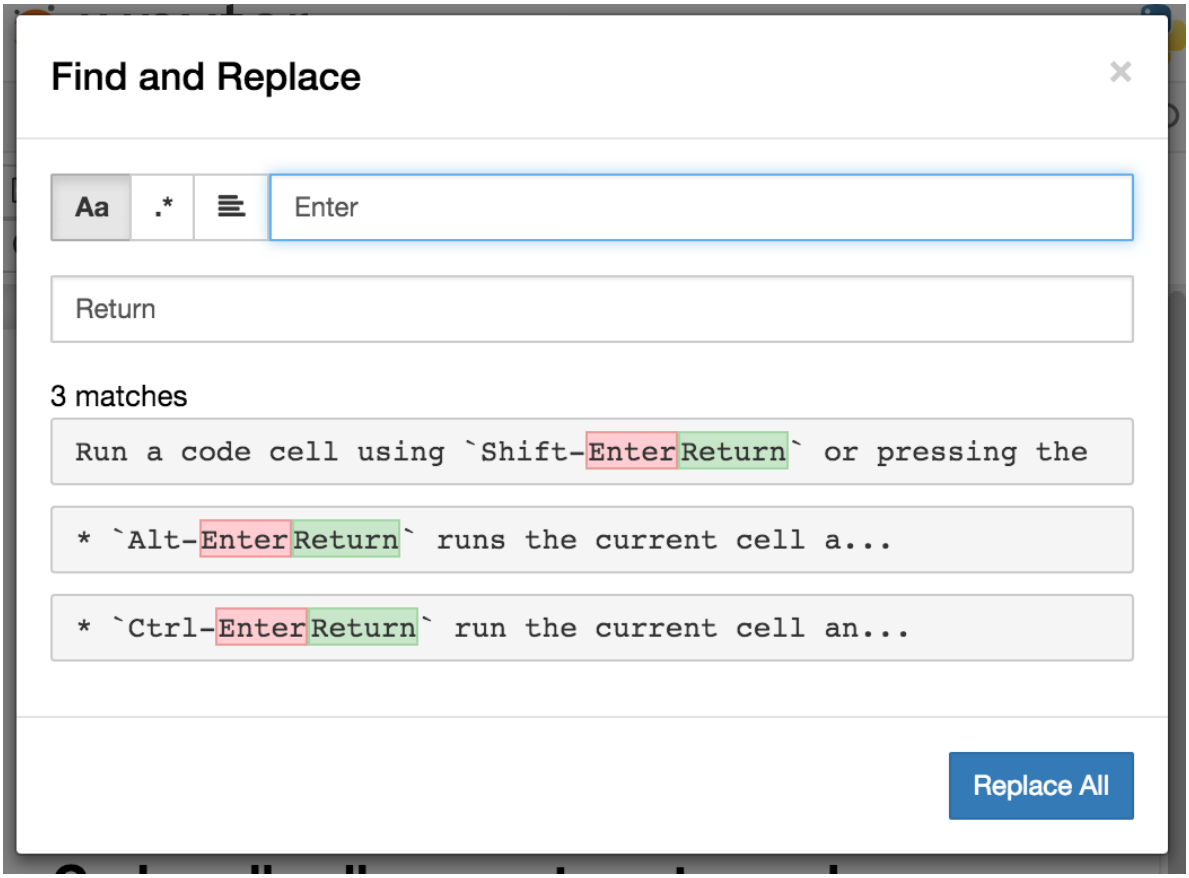
- Added *Restart & Run All* to the *Kernel* menu. Users can also bind it to a keyboard shortcut on action `restart-kernel-and-run-all-cells`.
- Added multiple-cell selection. Users press `Shift-Up/Down` or `Shift-K/J` to extend selection in command mode. Various actions such as cut/copy/paste, execute, and cell type conversions apply to all selected cells.



- Added a command palette for executing Jupyter actions by name. Users press `Cmd/Ctrl-Shift-P` or click the new command palette icon on the toolbar.



- Added a *Find and Replace* dialog to the *Edit* menu. Users can also press `F` in command mode to show the dialog.



Other improvements:

- Custom KernelManager methods can be Tornado coroutines, allowing async operations.
- Make clearing output optional when rewriting input with `set_next_input(replace=True)`.
- Added support for TLS client authentication via `--NotebookApp.client-ca`.
- Added tags to `jupyter/notebook` releases on DockerHub. `latest` continues to track the master branch.

See the 4.1 milestone on GitHub for a complete list of issues and pull requests handled.

4.0.x

4.0.6

- fix installation of mathjax support files
- fix some double-escape regressions in 4.0.5
- fix a couple of cases where errors could prevent opening a notebook

4.0.5

Security fixes for maliciously crafted files.

- CVE-2015-6938: malicious filenames

- CVE-2015-7337: malicious binary files in text editor.

Thanks to Jonathan Kamens at Quantopian and Juan Broullón for the reports.

4.0.4

- Fix inclusion of mathjax-safe extension

4.0.2

- Fix launching the notebook on Windows
- Fix the path searched for frontend config

4.0.0

First release of the notebook as a standalone package.