
Jupyter Notebook Documentation

Release 5.0.0.dev

<https://jupyter.org>

March 20, 2016

1	The Jupyter Notebook	3
2	UI Components	9
3	Config file and command line options	13
4	Running a notebook server	21
5	Security in Jupyter notebooks	25
6	Configuring the notebook frontend	29
7	Extending the Notebook	31
8	Want to contribute?	39
9	Examples and Tutorials	43
10	Jupyter notebook changelog	45

What's New in Jupyter Notebook

Release **4.1.0**

Release Announcement

- Cell toolbar selector moved to View menu
- Restart & Run All Cells added to Kernel menu
- Multiple-cell selection and actions including cut, copy, paste and execute
- Command palette added for executing Jupyter actions
- Find and replace added to Edit menu

To upgrade to the release: `pip install notebook --upgrade` or `conda upgrade notebook`

The Jupyter Notebook

1.1 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.

1.1.1 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](#).

1.1.2 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](#)

1.2 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.

1.2.1 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.

1.2.2 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](#)

1.3 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.

1.4 Structure of a notebook document

The notebook consists of a sequence of cells. A cell is a multiline 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 drop-down 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](#).

1.4.1 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

1.4.2 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

1.4.3 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`.

1.4.4 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.

1.5 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.

1.5.1 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.

1.6 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.

1.7 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.

1.8 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
```

1.9 Browser Compatibility

The Jupyter Notebook is officially supported the latest stable version 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)

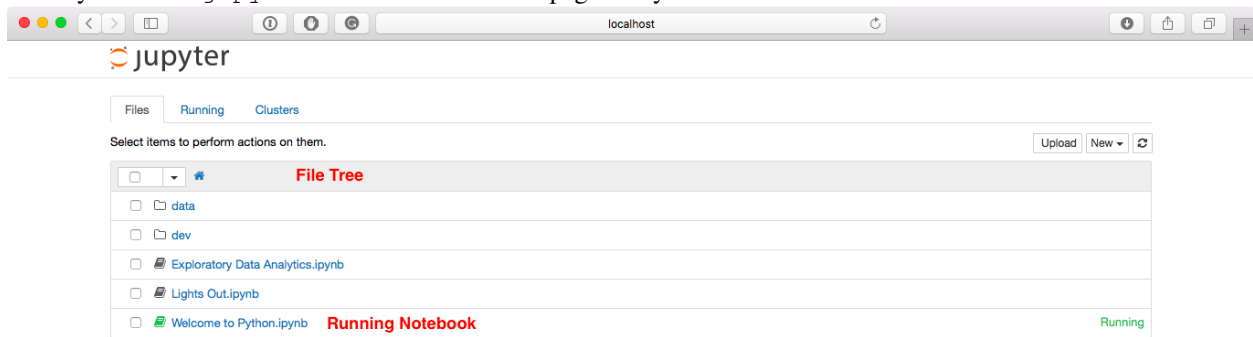
Using Safari with HTTPS and an untrusted certificate is known to not work (websockets will fail).

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.

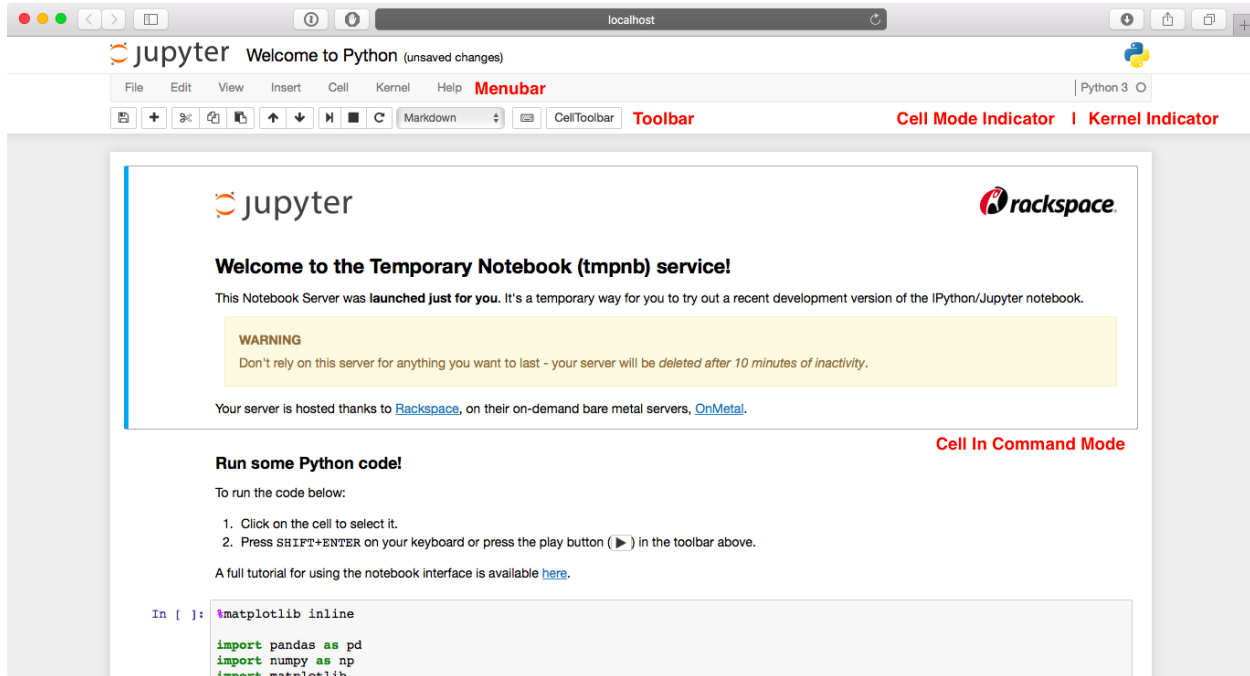
2.1 Notebook Dashboard

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



2.2 Notebook Editor

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

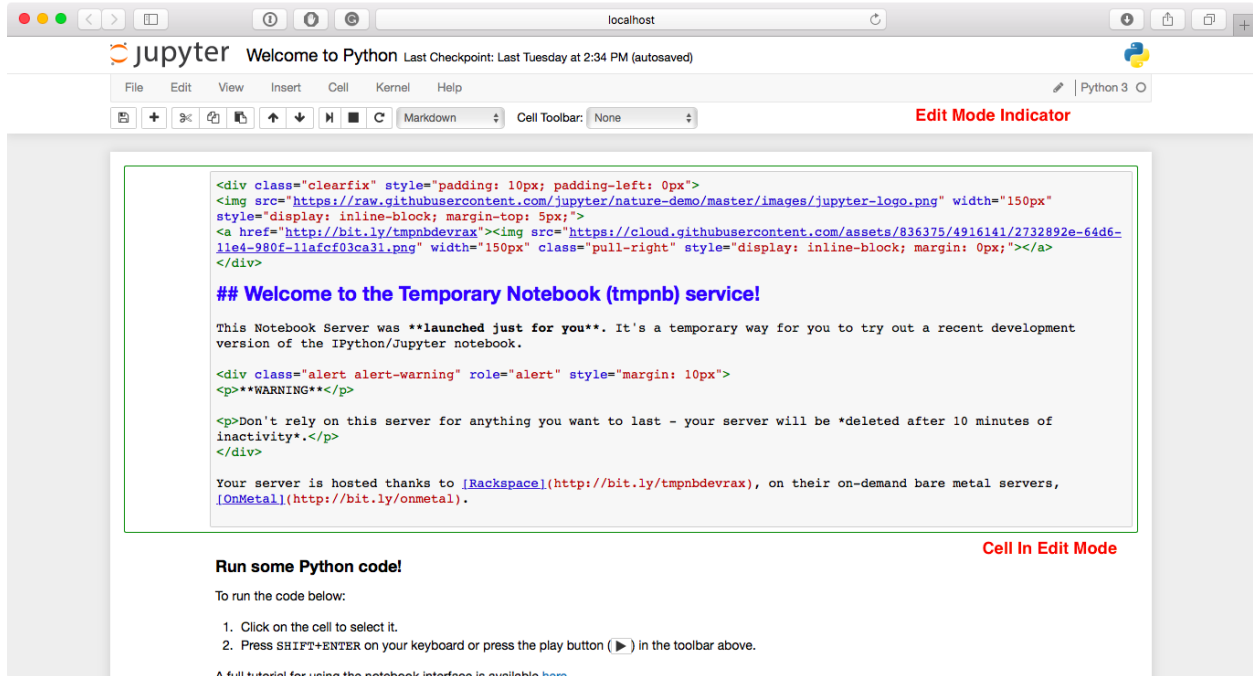


2.3 Interactive User Interface Tour of the Notebook

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.

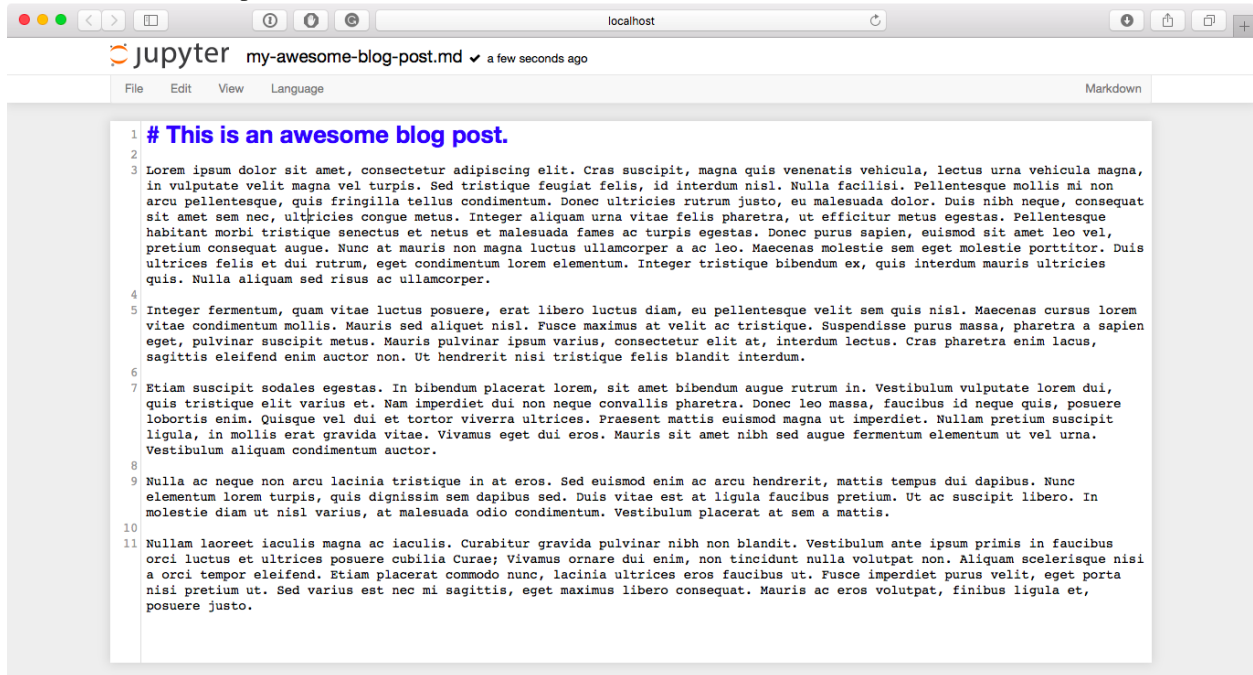
2.3.1 Edit Mode and Notebook Editor

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.



2.4 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.



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
```

3.1 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.

JupyterApp.answer_yes [Bool] Default: `False`

Answer yes to any prompts.

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

Full path of a config file.

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

Specify a config file to load.

JupyterApp.generate_config [Bool] Default: `False`

Generate default config file.

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.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: `u''`

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: `u''`

The full path to an SSL/TLS certificate file.

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

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

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.filemanager.FileContentsManager'`

The notebook manager class to use.

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

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: `u''`

The file where the cookie secret is stored.

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

The default URL to redirect to from /

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.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: `0`

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

NotebookApp.iopub_msg_rate_limit [Float] Default: `0`

(msg/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.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_url [Unicode] Default: `''`

The url for MathJax.js.

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 through 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 [Integer] Default: `8888`

The port the notebook server will listen on.

NotebookApp.port_retries [Integer] 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: `1.0`

(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: []

Python modules to load as notebook server extensions. This is an experimental API, and may change in future releases.

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

The session manager class to use.

NotebookApp.ssl_options [Dict] Default: {}

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

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 headers sent 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 [Integer] Default: 0

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

ConnectionFileMixin.hb_port [Integer] Default: 0

set the heartbeat port [default: random]

ConnectionFileMixin.iopub_port [Integer] Default: 0

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

ConnectionFileMixin.ip [Unicode] Default: u''

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 [Integer] Default: 0

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

ConnectionFileMixin.stdin_port [Integer] Default: 0

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

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

No description

KernelManager.autorestart [Bool] Default: `True`

Should we autorestart the kernel if it dies.

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.

Session.buffer_threshold [Integer] 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 [Integer] 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 [Integer] Default: `65536`

The maximum number of digests to remember.

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

Session.item_threshold [Integer] 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: `''`

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: `u''`

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: `u'username'`

Username for the Session. Default is your system username.

MultiKernelManager.default_kernel_name [Unicode] Default: `'python2'`

The name of the default kernel to start

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

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

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

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: `[u'__pycache__', '*.pyc', '*.pyo', '.DS_Store', '*.so', '*.dy...'`

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.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 successfully written, it replaces the old ones. This procedure, namely `'atomic_writing'`, causes some bugs on file system without 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.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.root_dir [Unicode] Default: u' '

No description

FileContentsManager.save_script [Bool] Default: False

DEPRECATED, use post_save_hook. Will be removed in Notebook 5.0

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

The hashing algorithm used to sign notebooks.

NotebookNotary.cache_size [Integer] Default: 65535

The number of notebook signatures to cache. When the number of signatures exceeds this value, the oldest 25% of signatures will be culled.

NotebookNotary.db_file [Unicode] Default: u' '

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

NotebookNotary.secret [Bytes] Default: ''

The secret key with which notebooks are signed.

NotebookNotary.secret_file [Unicode] Default: u' '

The file where the secret key is stored.

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.

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](#).

4.1 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`.

4.1.1 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
```

4.1.2 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.

4.1.3 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'
```

4.1.4 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](#).

4.2 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.

4.2.1 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.

4.3 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 sub-directory 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/'
```

4.4 Embedding the notebook in another website

Sometimes you may want to embed the notebook somewhere on your website, e.g. in an `IFrame`. To do this, you may need to override the Content-Security-Policy to allow embedding. Assuming your website is at `https://mywebsite.example.com`, you can embed the notebook on your website with the following configuration setting in `jupyter_notebook_config.py`:

```
c.NotebookApp.tornado_settings = {
    'headers': {
        'Content-Security-Policy': "frame-ancestors 'https://mywebsite.example.com' 'self' "
    }
}
```

When embedding the notebook in a website using an `iframe`, consider putting the notebook in single-tab mode. Since the notebook opens some links in new tabs by default, single-tab mode keeps the notebook from opening additional tabs. Adding the following to `~/ .jupyter/custom/custom.js` will enable single-tab mode:

```
define(['base/js/namespace'], function(Jupyter){
  Jupyter._target = '_self';
});
```

4.5 Known issues

4.5.1 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.

4.5.2 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"]
```

Security in Jupyter notebooks

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 introduces a security model to prevent execution of untrusted code without explicit user input.

5.1 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.

5.2 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?”

5.3 The details of trust

Jupyter notebooks store a signature in metadata, which is used to answer the question “Did the current user do this?”

This signature is a digest of the notebooks contents plus a secret key, known only to the user. The secret key is a user-only readable file in the Jupyter profile’s security directory. By default, this is:

```
~/.jupyter/profile_default/security/notebook_secret
```

Note: The notebook secret being stored in the profile means that loading a notebook in another profile results in it being untrusted, unless you copy or symlink the notebook secret to share it across profiles.

When a notebook is opened by a user, the server computes a signature with the user's key, and compares it with the signature stored in the notebook's metadata. If the signature matches, 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.

5.3.1 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 notebook file on disk is either trusted or not in its entirety.

5.3.2 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 with the user's key, and then store the newly signed notebook.

5.4 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.

5.5 Affected use cases

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

5.5.1 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.

5.5.2 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 secret, and use a Jupyter profile dedicated to the collaboration while working on the project.

5.5.3 Multiple profiles or machines

Since the notebook secret is stored in a profile directory by default, opening a notebook with a different profile or on a different machine will result in a different key, and thus be untrusted. The only current way to address this is by sharing the notebook secret. This can be facilitated by setting the configurable:

```
c.NotebookApp.secret_file = "/path/to/notebook_secret"
```

in each profile, and only sharing the secret once per machine.

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.

6.1 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.

6.2 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.

6.3 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.

6.4 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.

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.

7.1 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**.

7.1.1 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	Base name of the entity.
path	unicode	Full (<i>API-style</i>) 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. (<i>See Below</i>)
mimetype	unicode or None	The mimetype of content, if any. (<i>See Below</i>)
format	unicode or None	The format of content, if any. (<i>See Below</i>)

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.

7.1.2 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:

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

7.1.3 Customizing Checkpoints

TODO:

7.1.4 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.

7.2 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)
```

7.2.1 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.

7.3 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.

7.3.1 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.

7.3.2 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](#)

Want to contribute?

Here are some resources to help you get started with setting up a development environment, how to contribute, and technical aspects of contributing.

8.1 Notebook project developer guides

These guides provide information about specific topics related to developing the Notebook.

8.1.1 Installing JavaScript machinery

Note: This section is prepared for contributors to the Notebook source code. Users of the released Notebook do not need to install the JavaScript tools.

Building the Notebook from its GitHub source code requires some tools to create and minify JavaScript components and the CSS. These tools and the following steps are used when making a Notebook release.

1. Install [Node.js](#) and **npm**.

- Using the installers on [Node.js](#) website: Select a pre-built installer on the [Node.js](#) website. The installer will include Node.js and Node's package manager, **npm**.
- Using system's package manager: Install Node.js and **npm** using the system's package manager. For example, the command for Ubuntu or Debian is:

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

2. Install the notebook:

In the notebook repo, do a development install:

```
pip install -e .
```

3. Rebuild JavaScript and CSS

There is a build step for the JavaScript and CSS in the notebook. You will need to run this command whenever there are changes to JavaScript or LESS sources:

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

This command will automatically fetch any missing dependencies (bower, less) and install them in a subdirectory.

Prototyping tip

When doing prototyping which needs quick iteration of the Notebook's JavaScript, run `npm run build:watch` in the root of the repository. This will cause WebPack to monitor the files you edit and recompile them on the fly.

8.1.2 Making a notebook release

This document guides a contributor through creating a release of the Jupyter notebook.

Check installed tools

Review *Installing JavaScript machinery*. Make sure all the tools needed to generate the minified JavaScript and CSS files are properly installed.

Clean the repository

You can remove all non-tracked files with:

```
git clean -xdi
```

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.

Create the release

1. Update version number in `notebook/_version.py`.
2. Run this command:

```
python setup.py jsversion
```

It will modify (at least) `notebook/static/base/js/namespace.js` which makes 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` too.

8.1.3 Developer FAQ

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

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

2. What are the basic steps for a development install?

```
git clone https://github.com/jupyter/notebook
cd notebook
python setup.py js css
pip install -e .
```

8.2 Jupyter developer guides

The Project Jupyter organization has more general documentation about contributing to Jupyter projects. Currently, this is available in the Jupyter documentation in the [Developer Documentation](#) section.

Some of the topics include:

- [Submitting a Bug](#)
- [Submitting an Enhancement Proposal](#)
- [Contributing to the Documentation](#)
- [Git and Github Resources](#)
- [Contributing to the Code](#)

[View the original notebook on nbviewer](#)

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.

9.1 Tutorials

- [What is the Jupyter Notebook](#)
- [Notebook Basics](#)
- [Running Code](#)
- [Working With Markdown Cells](#)
- [Custom Keyboard Shortcuts](#)
- [JavaScript Notebook Extensions](#)

9.2 Examples

- [Importing Notebooks](#)
- [Connecting with the Qt Console](#)
- [Typesetting Equations](#)

[View the original notebook on nbviewer](#)

Jupyter notebook changelog

A summary of changes in the Jupyter notebook. For more detailed information, see [GitHub](#).

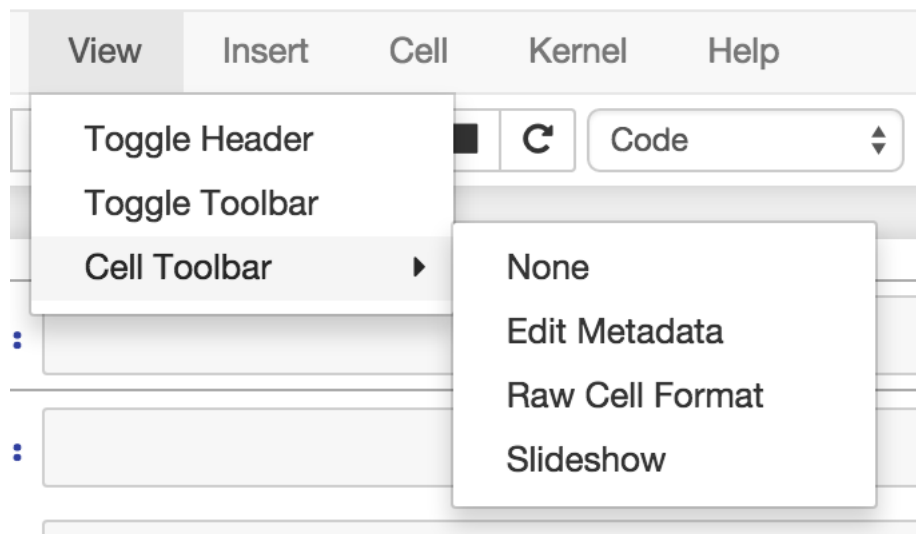
10.1 4.1.0

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.

Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the  button in the toolbar above:

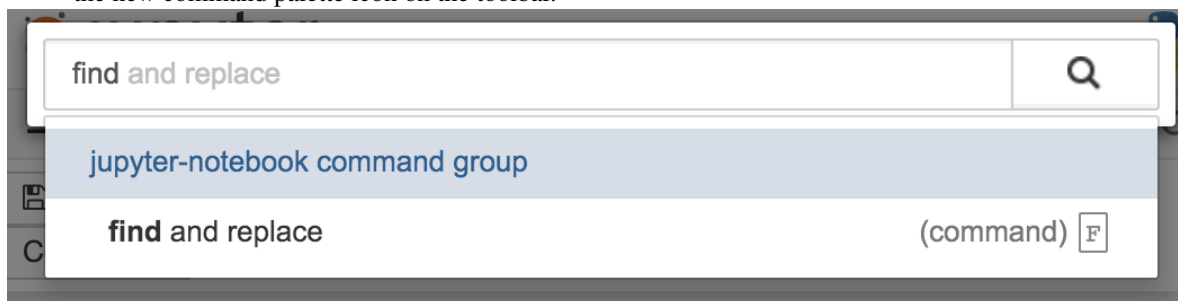
```
In [ ]: a = 10
```

```
In [ ]: print(a)
```

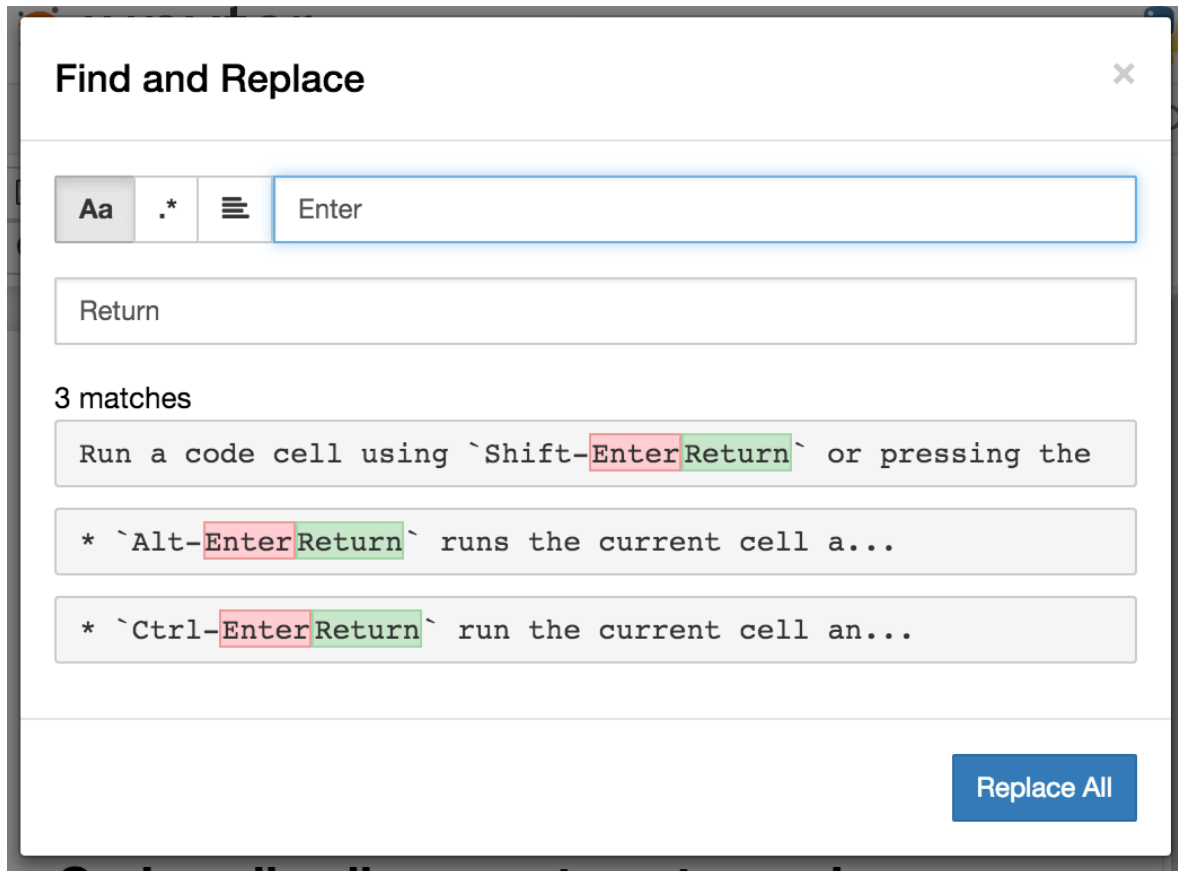
There are two other keyboard shortcuts for running code:

- `Alt-Enter` runs the current cell and inserts a new one below.
- `Ctrl-Enter` run the current cell and enters command mode.

- 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.

10.2 4.0.x

10.2.1 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

10.2.2 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.

10.2.3 4.0.4

- Fix inclusion of mathjax-safe extension

10.2.4 4.0.2

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

10.2.5 4.0.0

First release of the notebook as a standalone package.