Jupyter Notebook Documentation

Release 4.0.0.dev

https://jupyter.org

January 12, 2016

CHAPTER 1

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 documentation for directions 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

1.4 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.

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:

'Basic Output'_ example notebook

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

You can provide a conceptual structure for your computational document as a whole using different levels of headings; there are 6 levels available, from level 1 (top level) down to level 6 (paragraph). These can be used later for constructing tables of contents, etc. As with Markdown cells, a heading cell is replaced by a rich text rendering of the heading when the cell is executed.

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 on the following browsers:

- Chrome 13
- Safari 5
- Firefox 6

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

Config

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
```

2.1 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: `''`

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.

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.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.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.FileContents`

The notebook manager class to use.

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.enable_mathjax [Bool] Default: `True`

Whether to enable MathJax for typesetting math/TeX

MathJax is the javascript library IPython 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.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.MappingKerne`

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 IPython 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.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.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.SessionMan`

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 IPython 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.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 [Int] Default: `1024`

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

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.IOLoopKernelMan`

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.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.

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: `''`

No description

FileContentsManager.save_script [Bool] Default: `False`

DEPRECATED, use post_save_hook

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

The hashing algorithm used to sign notebooks.

NotebookNotary.cache_size [Int] 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: `''`

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: `b''`

The secret key with which notebooks are signed.

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

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.

CHAPTER 3

Running a notebook server

The Jupyter notebook web-application is based on a server-client structure. This server uses a two-process kernel architecture based on ZeroMQ, as well as Tornado for serving HTTP requests. By default, a notebook server runs on http://127.0.0.1:8888/ and is accessible only from *localhost*. This document describes how you can *secure a notebook server* and how to *run it on a public interface*.

3.1 Securing a notebook server

You can protect your notebook server with a simple single password by setting the `NotebookApp.password` configurable. 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'
```

Note: `passwd()` can also take the password as a string argument. **Do not** pass it as an argument inside an IPython session, as it will be saved in your input history.

You can then add this to your `jupyter_notebook_config.py`, e.g.:

```
# Password to use for web authentication
c = get_config()
c.NotebookApp.password =
u'sha1:67c9e60bb8b6:9ffede0825894254b2e042ea597d771089e11aed'
```

When using a password, it is a good idea to also use SSL, so that your password is not sent unencrypted by your browser. You can start the notebook to communicate via a secure protocol mode using a self-signed certificate with the command:

```
$ ipython notebook --certfile=mycert.pem
```

Note: 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 mycert.pem -out mycert.pem
```

Your browser will warn you of a dangerous certificate because it is self-signed. If you want to have a fully compliant certificate that will not raise warnings, it is possible (but rather involved) to obtain one, as explained in detail in this tutorial.

Keep in mind that when you enable SSL support, you will need to access the notebook server over `https://`, not over plain `http://`. The startup message from the server prints this, but it is easy to overlook and think the server is for some reason non-responsive.

3.2 Running a public notebook server

If you want to access your notebook server remotely via a web browser, you can do the following.

Start by creating a certificate file and a hashed password, as explained above. Then, 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 file has all fields commented; the minimum set you need to uncomment and edit is the following:

```
c = get_config()

# Notebook config
c.NotebookApp.certfile = u'/absolute/path/to/your/certificate/mycert.pem'
c.NotebookApp.ip = '*'
c.NotebookApp.open_browser = False
c.NotebookApp.password = u'sha1:bcd259ccf...[your hashed password here]'
# It is a good idea to put it on a known, fixed port
c.NotebookApp.port = 9999
```

You can then start the notebook and access it later by pointing your browser to `https://your.host.com:9999` with `jupyter notebook`.

3.2.1 Firewall Setup

To function correctly, the firewall on the computer running the ipython server must be configured to allow connections from client machines on the `c.NotebookApp.port` 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.

3.3 Running with a different URL prefix

The notebook dashboard (the landing page with an overview of the notebooks in your working directory) typically lives at the URL `http://localhost:8888/`. If you prefer that it lives, together with the rest of the notebook, under a sub-directory, 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/'}
```

3.4 Known issues

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.

CHAPTER 4

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.

4.1 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, an 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.

4.2 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?”

4.3 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.

4.3.1 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.

4.3.2 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.

4.4 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`.

4.5 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.

4.5.1 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.

4.5.2 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 an Jupyter profile dedicated to the collaboration while working on the project.

4.5.3 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.

CHAPTER 5

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.

5.1 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**.

5.1.1 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	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.

5.1.2 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:

`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?

5.1.3 Customizing Checkpoints

TODO:

5.1.4 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.

CHAPTER 6

Development

6.1 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.

View the original notebook on nbviewer

CHAPTER 7

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.

7.1 Tutorials

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

7.2 Examples

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

View the original notebook on nbviewer

CHAPTER 8

Jupyter notebook changelog

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

8.1 4.0.2

- Fix launching the notebook on Windows
- Fix the path searched for frontend config
- Fix nbextension-install on Python 2

8.2 4.0.0

First release of the notebook as a standalone package.