
Kernel Gateway Documentation

Release 0.5.0.dev

Project Jupyter team

April 04, 2016

1	Getting started	3
1.1	Using pip	3
1.2	Using jupyter/minimal-kernel	3
1.3	Using another docker-stacks image	3
2	Use Cases	5
3	Features	7
4	jupyter-websocket Mode	9
4.1	HTTP Resources	9
4.2	Websocket Resources	9
5	notebook-http Mode	11
5.1	Getting the Request Data	11
5.2	Setting the Response Body	12
5.3	Setting the Response Status and Headers	12
5.4	Swagger Spec	13
5.5	Running	13
6	KernelGatewayApp configuration options	15
7	Troubleshooting	17
7.1	I can't access kernel gateway in my Docker container.	17
7.2	Kernel gateway raises an error when I use notebook-http mode.	17
8	Development Workflow	19
8.1	Prerequisites	19
8.2	Clone the repo	19
8.3	Build the development image	19
8.4	Run the tests	19
8.5	Run the gateway server	20
8.6	Build the docs	20
8.7	Update the Swagger API spec	20
9	Summary of changes	21
9.1	0.5	21
9.2	0.5.0 (2016-04-08)	21
9.3	0.4	21

9.4	0.3	22
9.5	0.2	22
9.6	0.1	22
10	Indices and tables		25

Jupyter Kernel Gateway is a web server that supports different mechanisms for spawning and communicating with Jupyter kernels, such as:

- A Jupyter Notebook server-compatible HTTP API used for requesting kernels and talking the [Jupyter kernel protocol](#) with the kernels over Websockets
- A HTTP API defined by annotated notebook cells that maps HTTP verbs and resources to code to execute on a kernel

The server launches kernels in its local process/filesystem space. It can be containerized and scaled out using common technologies like [tmprnb](#), [Cloud Foundry](#), and [Kubernetes](#).

Contents:

Getting started

This document describes some of the basics of installing and running the Jupyter Kernel Gateway.

1.1 Using pip

We make stable releases of the kernel gateway to PyPI. You can use `pip` to install the latest version along with its dependencies.

```
# install from pypi
pip install jupyter_kernel_gateway
```

Once installed, you can use the `jupyter` CLI to run the server.

```
# run it with default options
jupyter kernelgateway
```

1.2 Using jupyter/minimal-kernel

The `docker-stacks` project defines a `minimal-kernel` Docker image. The Dockerfile for this image installs the kernel gateway and sets it as the default container command. You can run an instance of this container with the following commands:

```
docker pull jupyter/minimal-kernel
docker run -it --rm -p 8888:8888 jupyter/minimal-kernel
```

1.3 Using another docker-stacks image

You can add the kernel gateway to any `docker-stacks` image to get a more feature-filled kernel environment. For example, you could define a Dockerfile like so:

```
# start from the jupyter image with R, Python, and Scala (Apache Toree) kernels pre-installed
FROM jupyter/all-spark-notebook

# install the kernel gateway
RUN pip install jupyter_kernel_gateway

# run kernel gateway on container start, not notebook server
```

```
EXPOSE 8888
CMD ["jupyter", "kernelgateway", "--KernelGatewayApp.ip=0.0.0.0", "--KernelGatewayApp.port=8888"]
```

You can then build and run it.

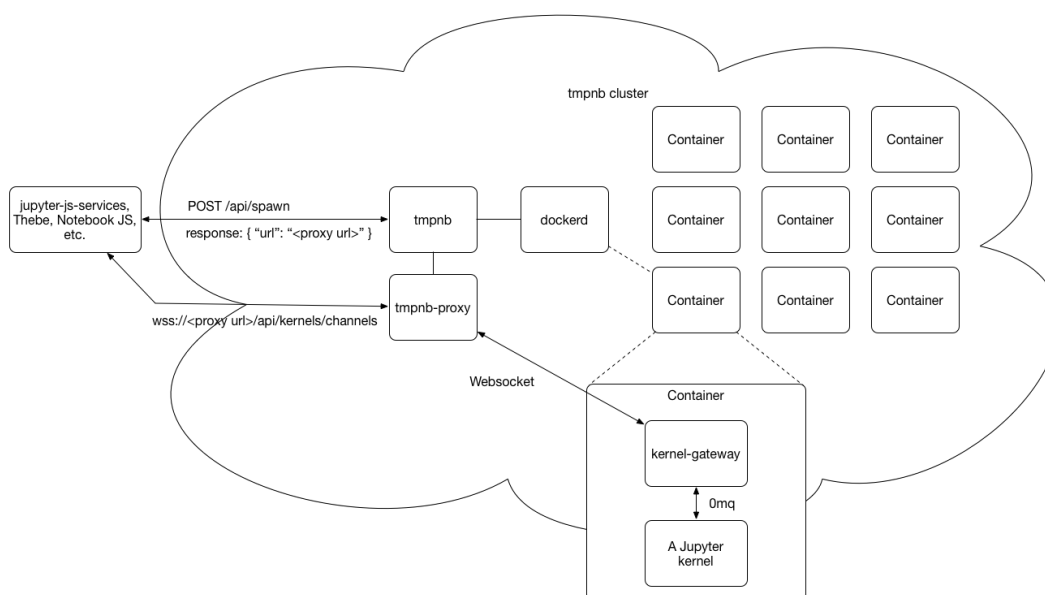
```
docker build -t my/kernel-gateway .
docker run -it --rm -p 8888:8888 my/kernel-gateway
```


Use Cases

The Jupyter Kernel Gateway makes possible the following novel uses of kernels:

- Attach a local Jupyter Notebook server to a compute cluster in the cloud running near big data (e.g., interactive gateway to Spark)
- Enable a new breed of non-notebook web clients to provision and use kernels (e.g., dashboards using [jupyter-js-services](#))
- Scale kernels independently from clients (e.g., via [tmpnb](#), [Binder](#), or your favorite cluster manager)
- Create microservices from notebooks via `notebook-http` mode

The following diagram shows how you might use `tmpnb` to deploy a pool of kernel gateway instances in Docker containers to support on-demand interactive compute:



For more inspiration, see the [jupyter/kernel_gateway_demos](#).

Features

The Jupyter Kernel Gateway has the following features:

- `jupyter-websocket` mode which provides a Jupyter Notebook server-compatible API for requesting kernels and communicating with them using Websockets
- `notebook-http` mode which maps HTTP requests to cells in annotated notebooks
- Option to set a shared authentication token and require it from clients
- Options to set CORS headers for servicing browser-based clients
- Option to set a custom base URL (e.g., for running under `tmprnb`)
- Option to limit the number kernel instances a gateway server will launch (e.g., to force scaling at the container level)
- Option to pre-spawn a set number of kernel instances
- Option to set a default kernel language to use when one is not specified in the request
- Option to pre-populate kernel memory from a notebook
- Option to serve annotated notebooks as HTTP endpoints, see *notebook-http*
- Option to allow downloading of the notebook source when running `notebook-http` mode
- Generation of [Swagger specs](#) for notebook-defined API in `notebook-http` mode
- A CLI for launching the kernel gateway: `jupyter kernelgateway OPTIONS`
- A Python 2.7 and 3.3+ compatible implementation

jupyter-websocket Mode

The `KernelGatewayApp.api` command line argument defaults to `jupyter-websocket`. In this mode, the kernel gateway exposes:

1. a superset of the HTTP API provided by the Jupyter Notebook server, and
2. the equivalent Websocket API implemented by the Jupyter Notebook server.

4.1 HTTP Resources

The HTTP API consists of kernel, session, monitoring, and metadata resources. All of these are documented in a `swagger.yaml` file. You can use the [Swagger UI](#) to interact with a running instance of the kernel gateway by pointing the tool to the `/api/swagger.json` resource.

4.2 Websocket Resources

The `/api/kernels/{kernel_id}/channels` resource multiplexes the [Jupyter kernel messaging protocol](#) over a single Websocket connection.

See the [NodeJS](#) and [Python](#) client demos for two simple examples of using these resources to send code to kernels for interactive evaluation.

notebook-http Mode

NOTE This mode may one day become a plug-in to the Jupyter Kernel Gateway or split into a separate project. The concepts documented here, however, will survive any move.

The `KernelGatewayApp.api` command line argument can be set to `notebook-http`. In this mode, the kernel gateway exposes annotated cells in the `KernelGatewayApp.seed_uri` notebook as HTTP resources.

To turn a notebook cell into a HTTP handler, you must prefix it with a single line comment. The comment describes the HTTP method and resource, as in the following Python example:

```
# GET /hello/world
print("hello world")
```

The annotation above declares the cell contents as the code to execute when the kernel gateway receives a HTTP GET request for the path `/hello/world`. For other languages, the comment prefix may change, but the rest of the annotation remains the same.

Multiple cells may share the same annotation. Their content is concatenated to form a single code segment at runtime. This facilitates typical, iterative development in notebooks with lots of short, separate cells: The notebook author does not need to merge all of the cells into one, or refactor to use functions.

```
# GET /hello/world
print("I'm cell #1")

# GET /hello/world
print("I'm cell #2")
```

5.1 Getting the Request Data

Before the gateway invokes an annotated cell, it sets the value of a global notebook variable named `REQUEST` to a JSON string containing information about the request. You may parse this string to access the request properties.

For example, in Python:

```
# GET /hello/world
req = json.loads(REQUEST)
# do something with req
```

You may specify path parameters when registering an endpoint by prepending a `:` to a path segment. For example, a path with parameters `firstName` and `lastName` would be defined as the following in a Python comment:

```
# GET /hello/:firstName/:lastName
```

The `REQUEST` object currently contains the following properties:

- `body` - The value of the body, see the [Body And Content Type](#Request Content-Type and Request Body Processing) section below
- `args` - An object with keys representing query parameter names and their associated values. A query parameter name may be specified multiple times in a valid URL, and so each value is a sequence (e.g., list, array) of strings from the original URL.
- `path` - An object of key-value pairs representing path parameters and their values.
- `headers` - An object of key-value pairs where a key is a HTTP header name and a value is the HTTP header value. If there are multiple values are specified for a header, the value will be an array.

5.1.1 Request Content-Type and Request Body Processing

If the HTTP request to the kernel gateway has a `Content-Type` header the value of `REQUEST.body` may change. Below is the list of outcomes for various mime-types:

- `application/json` - The `REQUEST.body` will be an object of key-value pairs representing the request body
- `multipart/form-data` and `application/x-www-form-urlencoded` - The `REQUEST.body` will be an object of key-value pairs representing the parameters and their values. Files are currently not supported for `multipart/form-data`
- `text/plain` - The `REQUEST.body` will be the string value of the body
- All other types will be sent as strings

5.2 Setting the Response Body

The response from an annotated cell may be set in one of two ways:

1. Writing to stdout in a notebook cell
2. Emitting output in a notebook cell

The first method is preferred because it is explicit: a cell writes to stdout using the appropriate language statement or function (e.g. Python `print`, Scala `println`, R `print`, etc.). The kernel gateway collects all bytes from kernel stdout and returns the entire byte string verbatim as the response body.

The second approach is used if nothing appears on stdout. This method is dependent upon language semantics, kernel implementation, and library usage. The response body will be the `content.data` structure in the Jupyter `execute_result` message.

In both cases, the response defaults to status 200 OK and `Content-Type: text/plain` if cell execution completes without error. If an error occurs, the status is 500 Internal Server Error. If the HTTP request method is not one supported at the given path, the status is 405 Not Supported. If you wish to return custom status or headers, see the next section.

See the [api_intro.ipynb](#) notebook for basic request and response examples.

5.3 Setting the Response Status and Headers

Annotated cells may have an optional metadata companion cell that sets the HTTP response status and headers. Consider this Python cell that creates a person entry in a database table and returns the new row ID in a JSON object:


```
# POST /person
req = json.loads(REQUEST)
row_id = person_table.insert(req['body'])
res = {'id' : row_id}
print(json.dumps(res))
```

Now consider this companion cell which runs after the cell above and sets a custom response header and status:

```
# ResponseInfo GET /hello/world
print(json.dumps({
    "headers" : {
        "Content-Type" : "application/json"
    },
    "status" : 201
})))
```

Currently, `headers` and `status` are the only fields supported. `headers` should be an object of key-value pairs mapping header names to header values. `status` should be an integer value. Both should be printed to stdout as JSON.

Given the two cells above, a POST request to `/person` produces a HTTP response like the following from the kernel gateway, assuming no errors occur:

```
HTTP/1.1 200 OK
Content-Type: application/json

{"id": 123}
```

See the [setting_response_metadata.ipynb](#) notebook for examples of setting response metadata.

5.4 Swagger Spec

The resource `/_api/spec/swagger.json` is automatically generated from the notebook used to define the HTTP API. The response is a simple Swagger spec which can be used with the [Swagger editor](#), a [Swagger ui](#), or with any other Swagger-aware tool.

Currently, every response is listed as having a status of 200 OK.

5.5 Running

The minimum number of arguments needed to run in HTTP mode are `--KernelGatewayApp.api=notebook-http` and `--KernelGatewayApp.seed_uri=some/notebook/file.ipynb`

If you development, you can run the kernel gateway in `notebook-http` mode using the Makefile in this repository:

```
make dev ARGS="--KernelGatewayApp.api='notebook-http' \
--KernelGatewayApp.seed_uri=/srv/kernel_gateway/etc/api_examples/api_intro.ipynb"
```

With the above Make command, all of the notebooks in `etc/api_examples` are mounted into `/srv/kernel_gateway/etc/api_examples/` and can be run in HTTP mode.

The `notebook-http` mode will honor the `prespawn_count` command line argument. This will start the specified number of kernels and execute the `seed_uri` notebook on each one. Requests will be distributed across the pool of prespawned kernels, providing a minimal layer of scalability. An example which starts a pool of 5 kernels follows:

```
make dev ARGS="--KernelGatewayApp.api='notebook-http' \  
--KernelGatewayApp.seed_uri=/srv/kernel_gateway/etc/api_examples/api_intro.ipynb" \  
--KernelGatewayApp.prespawn_count=5
```

KernelGatewayApp configuration options

Run `jupyter kernelgateway --help-all` after installation to see the full set of server options. A snapshot of this help appears below:

```
KernelGatewayApp options
-----
--KernelGatewayApp.allow_credentials=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Credentials header. (KG_ALLOW_CREDENTIALS env
    var)
--KernelGatewayApp.allow_headers=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Headers header. (KG_ALLOW_HEADERS env var)
--KernelGatewayApp.allow_methods=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Methods header. (KG_ALLOW_METHODS env var)
--KernelGatewayApp.allow_notebook_download=<Bool>
    Default: False
    Optional API to download the notebook source code in notebook-http mode,
    defaults to not allow
--KernelGatewayApp.allow_origin=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Origin header. (KG_ALLOW_ORIGIN env var)
--KernelGatewayApp.answer_yes=<Bool>
    Default: False
    Answer yes to any prompts.
--KernelGatewayApp.api=<Unicode>
    Default: 'jupyter-websocket'
    Controls which API to expose, that of a Jupyter kernel or the seed
    notebook's, using values "jupyter-websocket" or "notebook-http" (KG_API env
    var)
--KernelGatewayApp.auth_token=<Unicode>
    Default: ''
    Authorization token required for all requests (KG_AUTH_TOKEN env var)
--KernelGatewayApp.base_url=<Unicode>
    Default: ''
    The base path on which all API resources are mounted (KG_BASE_URL env var)
--KernelGatewayApp.config_file=<Unicode>
    Default: ''
    Full path of a config file.
--KernelGatewayApp.config_file_name=<Unicode>
    Default: ''
    Specify a config file to load.
--KernelGatewayApp.default_kernel_name=<Unicode>
```

```
Default: ''
The default kernel name to use when spawning a kernel
(KG_DEFAULT_KERNEL_NAME env var)
--KernelGatewayApp.expose_headers=<Unicode>
Default: ''
Sets the Access-Control-Expose-Headers header. (KG_EXPOSE_HEADERS env var)
--KernelGatewayApp.generate_config=<Bool>
Default: False
Generate default config file.
--KernelGatewayApp.ip=<Unicode>
Default: ''
IP address on which to listen (KG_IP env var)
--KernelGatewayApp.list_kernels=<Bool>
Default: False
Enables listing the running kernels through /api/kernels and /api/sessions
(KG_LIST_KERNELS env var). Note: Jupyter Notebook allows this by default but
kernel gateway does not .
--KernelGatewayApp.log_datefmt=<Unicode>
Default: '%Y-%m-%d %H:%M:%S'
The date format used by logging formatters for %(asctime)s
--KernelGatewayApp.log_format=<Unicode>
Default: '[%(name)s]%(highlevel)s %(message)s'
The Logging format template
--KernelGatewayApp.log_level=<Enum>
Default: 30
Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')
Set the log level by value or name.
--KernelGatewayApp.max_age=<Unicode>
Default: ''
Sets the Access-Control-Max-Age header. (KG_MAX_AGE env var)
--KernelGatewayApp.max_kernels=<Int>
Default: 0
Limits the number of kernel instances allowed to run by this gateway.
(KG_MAX_KERNELS env var)
--KernelGatewayApp.port=<Int>
Default: 0
Port on which to listen (KG_PORT env var)
--KernelGatewayApp.prespawn_count=<Int>
Default: None
Number of kernels to prespawn using the default language. (KG_PRESPAWN_COUNT
env var)
--KernelGatewayApp.seed_uri=<Unicode>
Default: ''
Runs the notebook (.ipynb) at the given URI on every kernel launched.
(KG_SEED_URI env var)
```

Troubleshooting

7.1 I can't access kernel gateway in my Docker container.

The kernel gateway server listens on port 8888 by default. Make sure this internal port is exposed on an external port when starting the container. For example, if you run:

```
docker run -it --rm -p 9000:8888 jupyter/minimal-kernel
```

you can access your kernel gateway on the IP address of your Docker host on port 9000.

7.2 Kernel gateway raises an error when I use `notebook-http` mode.

The `notebook-http` mode publishes a web API defined by annotations and code in a notebook. Make sure you are specifying a path or URL of a notebook (`*.ipynb`) file when you launch the kernel gateway in this mode. Set the `--KernelGatewayApp.seed_uri` command line parameter or `KG_SEED_URI` environment variable to do so.

Development Workflow

This document includes instructions for setting up Dockerized development environment for the Jupyter Kernel Gateway. It also includes common steps in the developer workflow such as running tests, building doc, updating specs, etc.

8.1 Prerequisites

Install `docker` and `docker-machine` on your system. Create and activate a Docker VM for development.

```
docker-machine create -d virtualbox dev
eval "$(docker-machine env dev)"
```

8.2 Clone the repo

Clone this repository in a local directory that your local Docker VM can volume mount:

```
# make a directory under ~ to put source
mkdir -p ~/projects
cd !$

# clone this repo
git clone https://github.com/jupyter/kernel_gateway.git
```

8.3 Build the development image

Build a Docker image containing additional dev libraries.

```
make image
```

8.4 Run the tests

Run the tests suites under Python 2 and 3.

```
make test-python3
make test-python2
```

8.5 Run the gateway server

Run an instance of the kernel gateway server.

```
make dev
```

To access the running server:

1. Run `docker-machine ls` and note the IP of the dev machine.
2. Visit `http://THAT_IP:8888/api` in your browser where `THAT_IP` is the IP address returned from the previous step.

8.6 Build the docs

Run Sphinx to build the HTML documentation.

```
make docs
```

8.7 Update the Swagger API spec

After modifying any of the APIs in `jupyter-websocket` mode, you must update the project's Swagger API specification.

1. Load the current `swagger.yaml` file into the [Swagger editor](#).
2. Make your changes.
3. Export both the `swagger.json` and `swagger.yaml` files.
4. Place the files in `kernel_gateway/services/api`.
5. Add, commit, and PR the changes.

Summary of changes

See `git log` for a more detailed summary of changes.

9.1 0.5

9.2 0.5.0 (2016-04-08)

- Support multiple cells per path in `notebook-http` mode
- Add a Swagger specification of the `jupyter-websocket` API
- Add `KERNEL_GATEWAY=1` to all kernel environments
- Support environment variables in `POST /api/kernels`
- `numpydoc` format docstrings on everything
- Convert README to Sphinx/ReadTheDocs site
- Convert `ActivityManager` to a `traitlets LoggingConfigurable`
- Fix `base_url` handling for all paths
- Fix unbounded growth of ignored kernels in `ActivityManager`
- Fix caching of Swagger spec in `notebook-http` mode
- Fix failure to install due to whitespace in `setup.py` version numbers
- Fix call to kernel manager base class when starting a kernel
- Fix test fixture hangs

9.3 0.4

9.3.1 0.4.0 (2016-02-17)

- Enable `/_api/activity` resource with stats about kernels in `jupyter-websocket` mode
- Enable `/api/sessions` resource with in-memory name-to-kernel mapping for non-notebook clients that want to look-up kernels by associated session name
- Fix prespawn kernel logic regression for `jupyter-websocket` mode

- Fix all handlers so that they return application/json responses on error
- Fix missing output from cells that emit display data in `notebook-http` mode

9.4 0.3

9.4.1 0.3.1 (2016-01-25)

- Fix CORS and auth token headers for `/_api/spec/swagger.json` resource
- Fix `allow_origin` handling for non-browser clients
- Ensure base path is prefixed with a forward slash
- Filter stderr from all responses in `notebook-http` mode
- Set Tornado logging level and Jupyter logging level together with `--log-level`

9.4.2 0.3.0 (2016-01-15)

- Support setting of status and headers in `notebook-http` mode
- Support automatic, minimal Swagger doc generation in `notebook-http` mode
- Support download of a notebook in `notebook-http` mode
- Support CORS and token auth in `notebook-http` mode
- Expose HTTP request headers in `notebook-http` mode
- Support multipart form encoding in `notebook-http` mode
- Fix request value JSON encoding when passing requests to kernels
- Fix kernel name handling when pre-spawning
- Fix lack of access logs in `notebook-http` mode

9.5 0.2

9.5.1 0.2.0 (2015-12-15)

- Support notebook-defined HTTP APIs on a pool of kernels
- Disable kernel instance list by default

9.6 0.1

9.6.1 0.1.0 (2015-11-18)

- Support Jupyter Notebook kernel CRUD APIs and Jupyter kernel protocol over Websockets
- Support shared token auth
- Support CORS headers

- Support base URL
- Support seeding kernels code from a notebook at a file path or URL
- Support default kernel, kernel pre-spawning, and kernel count limit
- First PyPI release

Indices and tables

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