jupyter_kernel_mgmt Documentation

Release 0.6.0.dev0

Jupyter Development Team

Jan 29, 2020

APPLICATION DEVELOPERS

This package provides the Python API for starting, managing and communicating with Jupyter kernels. For information on messaging with Jupyter kernels, please refer to the Jupyter Protocol documentation.

Note: This is a new interface under development, and may still change. Not all Jupyter applications use this yet. See the jupyter_client docs for the established way of discovering and managing kernels.

CHAPTER

ONE

KERNEL FINDER

The `jupyter_kernel_mgmt` package provides a means of discovering kernel types that are available for use. This is accomplished using the *KernelFinder* class.

KernelFinder instances are created in one of two ways.

1. The most common way is to call KernelFinder's class method *KernelFinder.from_entrypoints()*. This loads all registered kernel providers.
2. You can also provide a list of *kernel provider* instances to KernelFinder's constructor. This loads only those instances provided.

Once an instance of *KernelFinder* has been created, kernels can be discovered and launched via KernelFinder's instance methods, *find_kernels()* and *launch()*, respectively.

1.1 Finding kernels

Available kernel types are discovered using KernelFinder's *KernelFinder.find_kernels()* method. This method is a generator that walks the set of loaded kernel providers calling each of their *KernelProvider.find_kernels()* methods yielding each entry.

Each kernel type has an ID (e.g. `spec/python3`) and a dictionary containing information to help a program or a user select an appropriate kernel. Different providers may include different metadata in this dictionary.

1.1.1 Kernel Specifications

The main built-in kernel provider, *KernelSpecProvider*, looks for kernels described by files in certain specific folders. Each kernel is described by one directory, and the name of the directory is used in its kernel type ID. These kernel spec directories may be in a number of locations:

Type	Unix	Windows
System	`/usr/share/jupyter/kernels` `/usr/local/share/jupyter/kernels`	`%PROGRAMDATA%\jupyter\kernels`
User	`~/.local/share/jupyter/kernels` (Linux) `~/Library/Jupyter/kernels` (Mac)	`%APPDATA%\jupyter\kernels`
Env	`{sys.prefix}/share/jupyter/kernels`	

The user location takes priority over the system locations, and the case of the names is ignored, so selecting kernels works the same way whether or not the filesystem is case sensitive.

Since kernel names, and their *provider ids*, show up in URLs and other places, a kernelspec is required to have a simple name, only containing ASCII letters, ASCII numbers, and the simple separators: `-` hyphen, `.` period, `_` underscore.

Other locations may also be searched if the `JUPYTER_PATH` environment variable is set.

For IPython kernels, three types of files are presently used: `kernel.json`, `kernel.js`, and logo image files. However, different Kernel Providers can support other files and directories within the kernel directory or may not even use a directory for their kernel discovery model. That said, for kernels prior to Kernel Providers or those discovered by instances of class `KernelSpecProvider`, the most important file is **kernel.json**. This file consists of a JSON-serialized dictionary that adheres to the *kernel specification format*.

For example, the kernel.json file for the IPython kernel looks like this:

```
{
 "argv": ["python3", "-m", "IPython.kernel",
          "-f", "{connection_file}"],
 "display_name": "Python 3",
 "language": "python"
}
```

Kernel Specification Format

The information contained in each entry returned from a Kernel Provider's `find_kernels()` method consists of a dictionary containing the following keys and values:

- **display_name**: The kernel's name as it should be displayed in the UI. Unlike the kernel name used in the API, this can contain arbitrary unicode characters. This value should be provided by all kernel providers.
- **language**: The name of the language of the kernel. When loading notebooks, if no matching kernelspec key (may differ across machines) is found, a kernel with a matching *language* will be used. This allows a notebook written on any Python or Julia kernel to be properly associated with the user's Python or Julia kernel, even if they aren't listed under the same name as the author's. This value should be provided by all kernel providers.
- **metadata** (optional): A dictionary of additional attributes about this kernel. Metadata added here should be namespaced for the tool reading and writing that metadata.

Kernelspec-based providers obtain this information from a *kernel.json* file located in a directory pertaining to the kernel's name. Other fields in the kernel.json file include information used to launch and manage the kernel. As a result, you'll also find the following fields in *kernel.json* files:

- **argv**: (optional): A list of command line arguments used to start the kernel. For instances of class `KernelSpecProvider` the text `{connection_file}` in any argument will be replaced with the path to the connection file. However, subclasses of `KernelSpecProvider` may choose to provide different substitutions, especially if they don't use a connection file.
- **interrupt_mode** (optional): May be either `signal` or `message` and specifies how a client is supposed to interrupt cell execution on this kernel, either by sending an interrupt `signal` via the operating system's signalling facilities (e.g. *SIGINT* on POSIX systems), or by sending an `interrupt_request` message on the control channel (see kernel interrupt). If this is not specified `signal` mode will be used.
- **env** (optional): A dictionary of environment variables to set for the kernel. These will be added to the current environment variables before the kernel is started.

However, whether a provider exposes information used during their kernel's launch is entirely up to the provider.

1.1.2 IPython kernel provider

A second built-in kernel provider, *`IPykernelProvider`*, identifies if `ipykernel` is importable by the same Python the frontend is running on. If so, it provides exactly one kernel type, `pyimport/kernel`, which runs the IPython kernel in that same Python environment.

This may be functionally a duplicate of a kernel type discovered through an *installed kernelspec*.

1.2 Launching kernels

Launching kernels works similarly to their discovery. To launch a previously discovered kernel, the kernel's *fully qualified kernel type* is provided to KernelFinder's *`launch()`* method.

Note: A **fully qualified kernel type** includes a prefix of the kernel's *provider id* followed by a forward slash ('/'). For example, the `python3` kernel as provided by the *`KernelSpecProvider`* would have a fully qualified kernel type of `spec/python3`.

The application is responsible for ensuring the name passed to *`KernelFinder.launch()`* is prefixed with a provider id. For backwards compatibility with existing kernelspecs, a prefix of `spec/` is recommended in such cases so as to associate it with the `KernelSpecProvider`.

KernelFinder's launch method then locates the provider and calls the specific kernel provider's *`launch()`* method.

`KernelFinder.launch(name, cwd=None, launch_params=None)` takes two additional (and optional) arguments.

cwd (optional) specifies the current working directory relative to the notebook. Use of this value is up to the provider, as some kinds of kernels may not see the same filesystem as the process launching them.

launch_params (optional) specifies a dictionary of provider-specific name/value pairs that can can be used during the kernel's launch. What parameters are used can also be specified in the form of JSON schema embedded in the provider's kernel specification returned from its *`find_kernels()`* method. The application retrieving the kernel's information and invoking its subsequent launch, is responsible for providing appropriately relevant values.

1.2.1 Using launched kernels

A 2-tuple of connection information and the provider's *`kernel manager`* instance are returned from KernelFinder's launch method.

Although the *KernelManager* instance allows an application to manage a kernel's lifecycle, it does not provide a means of communicating with the kernel. To communicate with the kernel, an instance of *KernelClient* is required.

If the application would like to perform automatic restart operations (where the application detects the kernel is no longer running and issues a restart request) the application should establish a *KernelRestarter* instance.

CHAPTER

TWO

KERNEL MANAGER

The *kernel manager* is the primary interface through which an application controls a kernel's lifecycle upon its successful launch. Implemented by the *provider*, it exposes well-known methods to determine if the kernel is alive, as well as its interruption and shutdown, among other things. Applications like Jupyter Notebook invoke a majority of these methods indirectly via the REST API upon a user's request, while others are invoked from within the application itself.

CHAPTER

THREE

KERNEL CLIENT

To communicate with the kernel, an instance of *KernelClient* is required. This class instance takes its parameters from the application and formats them according to the Jupyter message protocol.

CHAPTER

FOUR

KERNEL RESTARTER

Applications that wish to perform automatic restart operations (where the application detects the kernel is no longer running and issues a restart request) use a *kernel restarter*. This instance is associated to the appropriate *kernel manager* instance to accomplish the restart functionality.

CHAPTER

FIVE

STANDALONE USAGE

The aim of the Kernel Management is to be integrated in larger applications such as the *Jupyter Server*.

Although, it can be used as a standalone module to, for example, launch a *Kernel Finder* from the command line and get a list of *Kernel Specifications*

```
python -m jupyter_kernel_mgmt.discovery
```

Similarly to jupyter_client, the Kernel Management can be used in different flavors and use cases. We are looking to your contributions to enrich the example use cases. You can get inspiration from the test_client.py source code.

PS: The existing separated jupyter_client can not be used in combination with the Kernel Management. The *Kernel-Client* code to use should be the one shipped by Kernel Management, not by jupyter_client.

CHAPTER

SIX

USE WITH JUPYTER SERVER

This page describes the way Jupyter Server uses the Kernel Management module.

On the Jupyter Server side, WEB handlers receive the javascript requests and are responsible for the communication with the Kernel Manager and Providers.

The MainKernelSpecHandler in *services/kernelsspecs/handlers* is reponsible to find the available Kernel Specs.

The other Handlers located in *services/kernels/handlers* are reponsible to launch and pass the message to the ZeroMQ channels:

- KernelHandler - accessible on endpoint /api/kernels
- MainKernelHandler - accessible on endpoint /api/kernels/<kernel_id>
- KernelActionHandler - accessible on endpoint /api/kernels/<kernel_id>/{interrupt,restart}
- ZMQChannelsHandler - accessible on endpoint /api/kernels/<kernel_id>/channels

Jupyter Server runs with a single *ServerApp* that initializes each of the handlers with services related to the Kernels:

- A `kernel_manager` - the default manager is *MappingKernelManager* provided by *jupyter_server*.
- A `kernel_finder` - is imported from the *jupyter_kernel_mgmt* library.
- A `session_manager` - uses a kernel_manager *MappingKernelManager*.

A single instance of MappingKernelManager is shared across all other objects (singleton pattern). The MappingKernelManager instance has a `KernelFinder` field.

The kernel_manager we are referring to in Jupyter Server should not be confused with the kernel_manager of the Kernel Manager it self. To avoid confusion, we will name the Servers's one *mapping_kernel_manager* in the next sections.

Notably, the ZMQChannelsHandler has access to the kernel's client interface via its kernel_client property.

In order to be found by a kernel_finder, Kernel Providers need to register them selves via the entrypoint mechanism.

The *included kernel providers*, *`KernelSpecProvider`* and *`IPykernelProvider`*, register by default their entrypoints.

```
entrypoints:
  jupyter_kernel_mgmt.kernel_type_providers' : [
    'spec = jupyter_kernel_mgmt.discovery:KernelSpecProvider',
    'pyimport = jupyter_kernel_mgmt.discovery:IPykernelProvider',
  ]
```

The system administrator can install additional providers. In that case, those external providers can register their own entypoints, see e.g kubernetes_kernel_provider, yarn_kernel_provider. . .

The interactions sequence between Jupyter Server and the Kernel Management is sketched here. Please note that this diagram just gives an idea of the interactions and is not aimed to reflect an exhaustive list of object constructs nor method calls…

See also:

High Level API

CHAPTER

SEVEN

KERNEL PROVIDERS

7.1 Creating a kernel provider

By writing a kernel provider, you can extend how Jupyter applications discover and start kernels. For example, you could find kernels in an environment system like conda, or kernels on remote systems which you can access.

To write a kernel provider, subclass `KernelProviderBase`, giving your provider an ID and overriding two methods.

`class MyKernelProvider`

`id`
A short string identifying this provider. Cannot contain forward slash (`/`).

`find_kernels``()`
Get the available kernel types this provider knows about. Return an iterable of 2-tuples: (name, attributes). *name* is a short string identifying the kernel type. *attributes* is a dictionary with information to allow selecting a kernel. This may also contain metadata describing other information pertaining to the kernel's launch - parameters, environment, etc.

`launch``(`*name*, *cwd=None*, *launch_params=None*`)`
Launches the kernel and returns a 2-tuple: (connection_info, kernel_manager). *connection_info* is a dictionary consisting of the connection information pertaining to the launched kernel. *kernel_manager* is a `KernelManager` instance.

name is a name returned from `find_kernels()`.

cwd is a string indicating the (optional) working directory in which the kernel should be launched.

launch_params is a dictionary of name/value pairs to be used by the provider and/or passed to the kernel as parameters.

For example, imagine we want to tell Jupyter about kernels for a new language called *oblong*:

```python
# oblong_provider.py
import asyncio
from jupyter_kernel_mgmt.discovery import KernelProviderBase
from jupyter_kernel_mgmt import KernelManager
from shutil import which

class OblongKernelProvider(KernelProviderBase):
    id = 'oblong'

    @asyncio.coroutine
    def find_kernels(self):
```

(continues on next page)

(continued from previous page)

```
        if not which('oblong-kernel'):
            return  # Check it's available

        # Two variants - for a real kernel, these could be something like
        # different conda environments.
        yield 'standard', {
            'display_name': 'Oblong (standard)',
            'language': {'name': 'oblong'},
            'argv': ['oblong-kernel'],
        }
        yield 'rounded', {
            'display_name': 'Oblong (rounded)',
            'language': {'name': 'oblong'},
            'argv': ['oblong-kernel'],
        }

    async def launch(self, name, cwd=None, launch_params=None):
        if name == 'standard':
            return await my_launch_method(cwd=cwd, launch_params=launch_params,
                                          kernel_cmd=['oblong-kernel'],
                                          extra_env={'ROUNDED': '0'})
        elif name == 'rounded':
            return await my_launch_method(cwd=cwd, launch_params=launch_params,
                                          kernel_cmd=['oblong-kernel'],
                                          extra_env={'ROUNDED': '1'})
        else:
            raise ValueError("Unknown kernel %s" % name)
```

You would then register this with an *entry point*. In your `setup.py`, put something like this:

```
setup(...
    entry_points = {
    'jupyter_kernel_mgmt.kernel_type_providers' : [
        # The name before the '=' should match the id attribute
        'oblong = oblong_provider:OblongKernelProvider',
    ]
})
```

7.2 Finding kernel types

To find and start kernels in client code, use *KernelFinder*. This uses multiple kernel providers to find available kernels. Like a kernel provider, it has methods `find_kernels` and `launch`. The kernel names it works with have the provider ID as a prefix, e.g. `oblong/rounded` (from the example above).

```
from jupyter_kernel_mgmt.discovery import KernelFinder
kf = KernelFinder.from_entrypoints()

## Find available kernel types
for name, attributes in kf.find_kernels():
    print(name, ':', attributes['display_name'])
# oblong/standard : Oblong (standard)
# oblong/rounded : Oblong(rounded)
# ...

```

(continues on next page)

(continued from previous page)

```
## Start a kernel by name
connect_info, manager = await kf.launch('oblong/standard')

client = IOLoopKernelClient(connect_info, manager=manager)
try:
    await asyncio.wait_for(client.wait_for_ready(), timeout=startup_timeout)
except RuntimeError:
    await client.shutdown_or_terminate()
    await client.close()
    await manager.kill()

# Use `manager` for lifecycle management, `client` for communication
```

7.3 Included kernel providers

`jupyter_kernel_mgmt` includes two kernel providers in its distribution.

1. *KernelSpecProvider* handles the discovery and launch of most existing kernelspec-based kernels that exist today.

2. *IPykernelProvider* handles the discover and launch of any IPython kernel that is located in the executing python's interpreter. For example, if the application is running in a virtual Python environment, this provider identifies if any IPython kernel is local to that environment and may not be identified by the path algorithm used by *KernelSpecProvider*.

7.4 Included kernel launchers

The kernel provider is responsible for launching the kernel and returning the connection information and *kernel manager* instance. Typically, a provider will implement a *launcher* to perform this action.

For those providers launching their kernels using the subprocess module's Popen class, `jupyter_kernel_mgmt` includes two kernel launcher implementations in its distribution.

1. *SubprocessKernelLauncher* launches kernels using the 'tcp' transport.

2. *SubprocessIPCKernelLauncher* launchers kernels using the 'ipc' transport (using filesystem sockets).

Both launchers return the resulting connection information and an instance of *KernelManager*, which is subsequently used to manage the rest of the kernel's lifecycle.

7.5 Glossary

Kernel instance A running kernel, a process which can accept ZMQ connections from frontends. Its state includes a namespace and an execution counter.

Kernel type The software to run a kernel instance, along with the context in which a kernel starts. One kernel type allows starting multiple, initially similar kernel instances. For instance, one kernel type may be associated with one conda environment containing `ipykernel`. The same kernel software in another environment would be a different kernel type. Another software package for a kernel, such as `IRkernel`, would also be a different kernel type.

Kernel provider A Python class to discover kernel types and allow a client to start instances of those kernel types. For instance, one kernel provider might find conda environments containing `ipykernel` and allow starting kernel instances in these environments. While another kernel provider might enable the ability to launch kernels across a Kubernetes cluster.

Provider Id A simple string ([a-z,0-9,_,-,.]) that identifies the provider. Each kernel name returned from the provider's `find_kernels()` method will be prefixed by the provider id followed by a '/' separator.

CHAPTER

EIGHT

KERNEL MANAGEMENT APIS

8.1 High Level API

These functions are convenient interfaces to start and interact with Jupyter kernels.

8.1.1 Async interface

These functions are meant to be called from an asyncio event loop.

class jupyter_kernel_mgmt.**run_kernel_async**(*name*, ***kwargs*)

Context manager to run a kernel by kernel type name.

Gives an async client:

```
async with run_kernel_blocking("pyimport/kernel") as kc:
    await kc.execute("a = 6 * 7")
```

async jupyter_kernel_mgmt.**start_kernel_async**(*name*, *cwd=None*, *launch_params=None*, *finder=None*)

Start a kernel by kernel type name, return (manager, async client)

8.1.2 Blocking interface

jupyter_kernel_mgmt.**run_kernel_blocking**(*name*, ***kwargs*)

Context manager to run a kernel by kernel type name.

Gives a blocking client:

```
with run_kernel_blocking("pyimport/kernel") as kc:
    kc.execute_interactive("print(6 * 7)")
```

jupyter_kernel_mgmt.**start_kernel_blocking**(*name*, ***, *cwd=None*, *launch_params=None*, *finder=None*, *startup_timeout=60*)

Start a kernel by kernel type name, return (manager, blocking client)

8.2 Kernel Finder

The Kernel Finder API is used by applications wishing to discover, launch, and manage Jupyter kernels. `KernelFinder` is not meant to be subclassed. To make it discover additional kernels, see *Kernel Providers*.

class `jupyter_kernel_mgmt.discovery.`**`KernelFinder`**(*providers*)

Manages a collection of kernel providers to find available kernel types. *providers* should be a list of kernel provider instances.

`find_kernels`()

Iterate over available kernel types. Yields 2-tuples of (prefixed_name, attributes)

classmethod from_entrypoints()

Load all kernel providers advertised by entry points.

Kernel providers should use the "jupyter_kernel_mgmt.kernel_type_providers" entry point group.

Returns an instance of KernelFinder.

async launch(*name*, *cwd=None*, *launch_params=None*)

Launch a kernel of a given kernel type using asyncio.

8.3 Kernel Provider

The Kernel Provider API is what a third-party would implement to introduce a form of kernel management that might differ from the more common `KernelSpecProvider` kernel. For example, a kernel provider might want to launch (and manage) kernels across a Kubernetes cluster. They would then implement a provider that performs the necessary action for launch and provide a *KernelManager* instance that can perform the appropriate actions to control the kernel's lifecycle.

See also:

Kernel Providers

class `jupyter_kernel_mgmt.discovery.`**`KernelProviderBase`**

abstract find_kernels()

Return an iterator of (kernel_name, kernel_info_dict) tuples.

abstract async launch(*name*, *cwd=None*, *launch_params=None*)

Launch a kernel, returns a 2-tuple of (connection_info, kernel_manager).

name will be one of the kernel names produced by find_kernels() and known to this provider.

cwd (optional) a string that specifies the path to the current directory of the notebook as conveyed by the client. Its interpretation is provider-specific.

launch_params (optional) a dictionary consisting of the launch parameters used to launch the kernel. Its interpretation is provider-specific.

This method launches and manages the kernel in an asynchronous (non-blocking) manner.

load_config(*config=None*)

Loads the configuration corresponding to the hosting application. This method is called during KernelFinder initialization prior to any other methods. The Kernel provider is responsible for interpreting the *config* parameter (when present).

config (optional) an instance of Config consisting of the hosting application's configurable traitlets.

class `jupyter_kernel_mgmt.discovery.`**`KernelSpecProvider`**(*search_path=None*)

Offers kernel types from installed kernelspec directories.

`find_kernels`()

Return an iterator of (kernel_name, kernel_info_dict) tuples.

async **`launch`**(*name*, *cwd=None*, *launch_params=None*)

Launch a kernel, returns a 2-tuple of (connection_info, kernel_manager).

name will be one of the kernel names produced by find_kernels() and known to this provider.

cwd (optional) a string that specifies the path to the current directory of the notebook as conveyed by the client. Its interpretation is provider-specific.

launch_params (optional) a dictionary consisting of the launch parameters used to launch the kernel. Its interpretation is provider-specific.

This method launches and manages the kernel in an asynchronous (non-blocking) manner.

class `jupyter_kernel_mgmt.discovery.`**`IPykernelProvider`**

Offers a kernel type using the Python interpreter it's running in. This checks if ipykernel is importable first. If import fails, it doesn't offer a kernel type.

`find_kernels`()

Return an iterator of (kernel_name, kernel_info_dict) tuples.

async **`launch`**(*name*, *cwd=None*, *launch_params=None*)

Launch a kernel, returns a 2-tuple of (connection_info, kernel_manager).

name will be one of the kernel names produced by find_kernels() and known to this provider.

cwd (optional) a string that specifies the path to the current directory of the notebook as conveyed by the client. Its interpretation is provider-specific.

launch_params (optional) a dictionary consisting of the launch parameters used to launch the kernel. Its interpretation is provider-specific.

This method launches and manages the kernel in an asynchronous (non-blocking) manner.

8.4 Kernel Launchers

The kernel provider is responsible for launching the kernel and returning the connection information and *kernel manager* instance. For those providers choosing to use Popen as their means of launching their kernels, `jupyter_kernel_mgmt` provides *`SubprocessKernelLauncher`* for the 'tcp' transport and and *`SubprocessIPCKernelLauncher`* for the 'ipc' transport (using filesystem sockets).

class `jupyter_kernel_mgmt.subproc.launcher.`**`SubprocessKernelLauncher`**(*kernel_cmd*, *cwd*, *extra_env=None*, *ip=None*, *launch_params=None*)

Run a kernel asynchronously in a subprocess.

Parameters

- **`kernel_cmd`** (*`list of str`*) – The Popen command template to launch the kernel
- **`cwd`** (*`str`*) – The working directory to launch the kernel in
- **`extra_env`** (*`dict, optional`*) – Dictionary of environment variables to update the existing environment

- **ip** (*str, optional*) – 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!

build_popen_kwargs(*connection_file*)
Build a dictionary of arguments to pass to Popen

files_to_cleanup(*connection_file*, *connection_info*)
Find files to be cleaned up after this kernel is finished.

This method is mostly to be overridden for cleaning up IPC sockets.

format_kernel_cmd(*connection_file*, *kernel_resource_dir=None*)
Replace templated args (e.g. {connection_file})

async launch()
The main method to launch a kernel.

Returns (connection_info, kernel_manager)

make_connection_file()
Generates a JSON config file, including the selection of random ports.

make_ports()
Randomly select available ports for each of port_names

class jupyter_kernel_mgmt.subproc.launcher.**SubprocessIPCKernelLauncher**(*kernel_cmd*, *cwd*, *extra_env=None*, *ip=None*, *launch_params=None*)
Start a kernel on this machine to listen on IPC (filesystem) sockets

files_to_cleanup(*connection_file*, *connection_info*)
Find files to be cleaned up after this kernel is finished.

This method is mostly to be overridden for cleaning up IPC sockets.

make_ports()
Randomly select available ports for each of port_names

8.5 Kernel Manager

The Kernel Manager API is used to manage a kernel's lifecycle. It does not provide communication support between the application and the kernel itself. Any third-parties implementing their own `KernelProvider` would likely implement their own `KernelManager` derived from the `KernelManagerABC` abstract base class. However, those providers using *Popen to launch local kernels* can use `KernelManager` directly.

class jupyter_kernel_mgmt.managerabc.**KernelManagerABC**
Abstract base class from which all KernelManager classes are derived.

kernel_id
The id associated with the kernel.

abstract async is_alive()
Check whether the kernel is currently alive (e.g. the process exists)

abstract async wait()
Wait for the kernel process to exit.

Returns True if the kernel is still alive after waiting, False if it exited (like is_alive()).

abstract async signal(*signum*)
Send a signal to the kernel.

abstract async interrupt()
Interrupt the kernel by sending it a signal or similar event

Kernels can request to get interrupts as messages rather than signals. The manager is *not* expected to handle this. `KernelClient.interrupt()` should send an interrupt_request or call this method as appropriate.

abstract async kill()
Forcibly terminate the kernel.

This method may be used to dispose of a kernel that won't shut down. Working kernels should usually be shut down by sending shutdown_request from a client and giving it some time to clean up.

async cleanup()
Clean up any resources, such as files created by the manager.

class `jupyter_kernel_mgmt.subproc.manager.`**KernelManager**(*popen*, *files_to_cleanup=None*, *win_interrupt_evt=None*)
Manages a single kernel in a subprocess on this host.

Parameters

- **popen** (`subprocess.Popen or asyncio.subprocess.Process`) – The process with the started kernel. Windows will use Popen (by default), while non-Windows will use asyncio's Process.
- **files_to_cleanup** (`list of paths, optional`) – Files to be cleaned up after terminating this kernel.
- **win_interrupt_evt** – On Windows, a handle to be used to interrupt the kernel. Not used on other platforms.

async cleanup()
Clean up resources when the kernel is shut down

async interrupt()
Interrupts the kernel by sending it a signal.

Unlike `signal_kernel`, this operation is well supported on all platforms.

Kernels may ask for interrupts to be delivered by a message rather than a signal. This method does *not* handle that. Use KernelClient.interrupt to send a message or a signal as appropriate.

async is_alive()
Is the kernel process still running?

async kill()
Kill the running kernel.

async signal(*signum*)
Sends a signal to the process group of the kernel (this usually includes the kernel and any subprocesses spawned by the kernel).

Note that since only SIGTERM is supported on Windows, this function is only useful on Unix systems.

async wait()
Wait for kernel to terminate

8.6 Kernel Client

The Kernel Client API is how an application communicates with a previously launched kernel using the Jupyter message protocol (FIXME jupyter_protocol). For applications based on IO loops, Jupyter Kernel Management provides `IOLoopKernelClient`.

class `jupyter_kernel_mgmt.client_base.`**`KernelClient`**(*connection_info*, *manager=None*, *use_heartbeat=True*)

Communicates with a single kernel on any host via zmq channels.

The messages that can be sent are exposed as methods of the client (KernelClient.execute, complete, history, etc.). These methods only send the message, they don't wait for a reply. To get results, use e.g. `get_shell_msg()` to fetch messages from the shell channel.

`close`()

Close sockets of this client.

After calling this, the client can no longer be used.

`comm_info`(*target_name=None*, *_header=None*)

Request comm info

Returns

Return type The msg_id of the message sent

`complete`(*code*, *cursor_pos=None*, *_header=None*)

Tab complete text in the kernel's namespace.

Parameters

- **`code`** (*`str`*) – The context in which completion is requested. Can be anything between a variable name and an entire cell.
- **`cursor_pos`** (*`int, optional`*) – The position of the cursor in the block of code where the completion was requested. Default: `len(code)`

Returns

Return type The msg_id of the message sent.

`execute`(*code*, *silent=False*, *store_history=True*, *user_expressions=None*, *allow_stdin=None*, *stop_on_error=True*, *_header=None*)

Execute code in the kernel.

Parameters

- **`code`** (*`str`*) – A string of code in the kernel's language.
- **`silent`** (*`bool, optional (default False)`*) – If set, the kernel will execute the code as quietly as possible, and will force store_history to be False.
- **`store_history`** (*`bool, optional (default True)`*) – If set, the kernel will store command history. This is forced to be False if silent is True.
- **`user_expressions`** (*`dict, optional`*) – A dict mapping names to expressions to be evaluated in the user's dict. The expression values are returned as strings formatted using `repr()`.
- **`allow_stdin`** (*`bool, optional (default self.allow_stdin)`*) – Flag for whether the kernel can send stdin requests to frontends.

  Some frontends (e.g. the Notebook) do not support stdin requests. If raw_input is called from code executed from such a frontend, a StdinNotImplementedError will be raised.

- **`stop_on_error`** (*`bool, optional (default True)`*) – Flag whether to abort the execution queue, if an exception is encountered.

Returns

Return type The msg_id of the message sent.

`history`(*raw=True*, *output=False*, *hist_access_type='range'*, *_header=None*, ***kwargs*)
Get entries from the kernel's history list.

Parameters

- **`raw`** (*`bool`*) – If True, return the raw input.
- **`output`** (*`bool`*) – If True, then return the output as well.
- **`hist_access_type`** (*`str`*) –

  'range' (fill in session, start and stop params), 'tail' (fill in n) or 'search' (fill in pattern param).
- **`session`** (*`int`*) – For a range request, the session from which to get lines. Session numbers are positive integers; negative ones count back from the current session.
- **`start`** (*`int`*) – The first line number of a history range.
- **`stop`** (*`int`*) – The final (excluded) line number of a history range.
- **`n`** (*`int`*) – The number of lines of history to get for a tail request.
- **`pattern`** (*`str`*) – The glob-syntax pattern for a search request.

Returns

Return type The ID of the message sent.

`input`(*string*, *parent=None*)
Send a string of raw input to the kernel.

This should only be called in response to the kernel sending an `input_request` message on the stdin channel.

`inspect`(*code*, *cursor_pos=None*, *detail_level=0*, *_header=None*)
Get metadata information about an object in the kernel's namespace.

It is up to the kernel to determine the appropriate object to inspect.

Parameters

- **`code`** (*`str`*) – The context in which info is requested. Can be anything between a variable name and an entire cell.
- **`cursor_pos`** (*`int, optional`*) – The position of the cursor in the block of code where the info was requested. Default: `len(code)`
- **`detail_level`** (*`int, optional`*) – The level of detail for the introspection (0-2)

Returns

Return type The msg_id of the message sent.

`interrupt`(*_header=None*)
Send an interrupt message/signal to the kernel

`is_complete`(*code*, *_header=None*)
Ask the kernel whether some code is complete and ready to execute.

kernel_info(*_header=None*)
Request kernel info

Returns

Return type The msg_id of the message sent

property owned_kernel
True if this client 'owns' the kernel, i.e. started it.

shutdown(*restart=False*, *_header=None*)
Request an immediate kernel shutdown.

Upon receipt of the (empty) reply, client code can safely assume that the kernel has shut down and it's safe to forcefully terminate it if it's still alive.

The kernel will send the reply via a function registered with Python's atexit module, ensuring it's truly done as the kernel is done with all normal operation.

Returns

Return type The msg_id of the message sent

class `jupyter_kernel_mgmt.client.`**IOLoopKernelClient**(*connection_info*, *manager=None*)
Uses a zmq/tornado IOLoop to handle received messages and fire callbacks.

Use ClientInThread to run this in a separate thread alongside your application.

add_handler(*handler*, *channels*)
Add a callback for received messages on one or more channels.

Parameters

- **handler** (*function*) – Will be called for each message received with the message dictionary as a single argument.
- **channels** (*set or str*) – Channel names: 'shell', 'iopub', 'stdin' or 'control'

close()
Close the client's sockets & streams.

This does not close the IOLoop.

comm_info(*target_name=None*, *_header=None*)
Request comm info

Returns

Return type The msg_id of the message sent

complete(*code*, *cursor_pos=None*, *_header=None*)
Tab complete text in the kernel's namespace.

Parameters

- **code** (*str*) – The context in which completion is requested. Can be anything between a variable name and an entire cell.
- **cursor_pos** (*int, optional*) – The position of the cursor in the block of code where the completion was requested. Default: `len(code)`

Returns

Return type The msg_id of the message sent.

execute(*code*, *silent=False*, *store_history=True*, *user_expressions=None*, *allow_stdin=None*, *stop_on_error=True*, *interrupt_timeout=None*, *idle_timeout=None*, *raise_on_no_idle=False*, *_header=None*)
Execute code in the kernel.

Parameters

- **code** (*str*) – A string of code in the kernel's language.
- **silent** (*bool, optional (default False)*) – If set, the kernel will execute the code as quietly as possible, and will force store_history to be False.
- **store_history** (*bool, optional (default True)*) – If set, the kernel will store command history. This is forced to be False if silent is True.
- **user_expressions** (*dict, optional*) – A dict mapping names to expressions to be evaluated in the user's dict. The expression values are returned as strings formatted using `repr()`.
- **allow_stdin** (*bool, optional (default self.allow_stdin)*) – Flag for whether the kernel can send stdin requests to frontends.

  Some frontends (e.g. the Notebook) do not support stdin requests. If raw_input is called from code executed from such a frontend, a StdinNotImplementedError will be raised.
- **stop_on_error** (*bool, optional (default True)*) – Flag whether to abort the execution queue, if an exception is encountered.

Returns

Return type The msg_id of the message sent.

history(*raw=True*, *output=False*, *hist_access_type='range'*, *_header=None*, ***kwargs*)
Get entries from the kernel's history list.

Parameters

- **raw** (*bool*) – If True, return the raw input.
- **output** (*bool*) – If True, then return the output as well.
- **hist_access_type** (*str*) –

  'range' (fill in session, start and stop params), 'tail' (fill in n) or 'search' (fill in pattern param).
- **session** (*int*) – For a range request, the session from which to get lines. Session numbers are positive integers; negative ones count back from the current session.
- **start** (*int*) – The first line number of a history range.
- **stop** (*int*) – The final (excluded) line number of a history range.
- **n** (*int*) – The number of lines of history to get for a tail request.
- **pattern** (*str*) – The glob-syntax pattern for a search request.

Returns

Return type The ID of the message sent.

input(*string*, *parent=None*)
Send a string of raw input to the kernel.

This should only be called in response to the kernel sending an `input_request` message on the stdin channel.

`inspect`(*code*, *cursor_pos=None*, *detail_level=0*, *_header=None*)
Get metadata information about an object in the kernel's namespace.

It is up to the kernel to determine the appropriate object to inspect.

Parameters

- **code** (*str*) – The context in which info is requested. Can be anything between a variable name and an entire cell.
- **cursor_pos** (*int, optional*) – The position of the cursor in the block of code where the info was requested. Default: `len(code)`
- **detail_level** (*int, optional*) – The level of detail for the introspection (0-2)

Returns

Return type The msg_id of the message sent.

`async interrupt`(*_header=None*)
Send an interrupt message/signal to the kernel

`is_complete`(*code*, *_header=None*)
Ask the kernel whether some code is complete and ready to execute.

`kernel_info`(*_header=None*)
Request kernel info

Returns

Return type The msg_id of the message sent

`property owned_kernel`
True if this client 'owns' the kernel, i.e. started it.

`remove_handler`(*handler*, *channels=None*)
Remove a previously registered callback.

`shutdown`(*restart=False*, *_header=None*)
Request an immediate kernel shutdown.

Upon receipt of the (empty) reply, client code can safely assume that the kernel has shut down and it's safe to forcefully terminate it if it's still alive.

The kernel will send the reply via a function registered with Python's atexit module, ensuring it's truly done as the kernel is done with all normal operation.

Returns

Return type The msg_id of the message sent

`shutdown_or_terminate`(*timeout=5.0*)
Ask the kernel to shut down, and terminate it if it takes too long.

The kernel will be given up to timeout seconds to respond to the shutdown message, then the same timeout to terminate.

8.7 Kernel Restarter

The Kernel Restarter API is used by applications wishing to perform automatic kernel restarts upon detection of the kernel's unexpected termination. `jupyter_kernel_mgmt` provides `KernelRestarterBase` and provides an implementation of that class for Tornado-based applications via `TornadoKernelRestarter`.

class `jupyter_kernel_mgmt.restarter.`**`KernelRestarterBase`**(*kernel_manager*, *kernel_type*, *kernel_finder=None*, ***kw*)

Monitor and autorestart a kernel.

`debug`
Whether to include every poll event in debugging output. Has to be set explicitly, because there will be *a lot* of output.

`time_to_dead`
Kernel heartbeat interval in seconds.

`restart_limit`
The number of consecutive autorestarts before the kernel is presumed dead.

`start`()
Start monitoring the kernel.

`stop`()
Stop monitoring.

`add_callback`(*f*, *event*)
Register a callback to fire on a particular event.

Possible values for event: 'died': the monitored kernel has died

'restarted': a restart has been attempted (this does not necessarily mean that the new kernel is usable).

'failed': *restart_limit* attempts have failed in quick succession, and the restarter is giving up.

`remove_callback`(*f*, *event*)
Unregister a callback from a particular event

Possible values for *event* are the same as in `add_callback()`.

`async do_restart`(*auto=False*)
Called when the kernel has died

`async poll`()

class `jupyter_kernel_mgmt.restarter.`**`TornadoKernelRestarter`**(*kernel_manager*, *kernel_type*, *kernel_finder=None*, ***kw*)

Monitor a kernel using the tornado ioloop.

`add_callback`(*f*, *event*)
Register a callback to fire on a particular event.

Possible values for event: 'died': the monitored kernel has died

'restarted': a restart has been attempted (this does not necessarily mean that the new kernel is usable).

'failed': *restart_limit* attempts have failed in quick succession, and the restarter is giving up.

`add_traits`(***traits*)
Dynamically add trait attributes to the HasTraits instance.

classmethod class_config_rst_doc()
Generate rST documentation for this class' config options.

Excludes traits defined on parent classes.

classmethod class_config_section()
Get the config class config section

classmethod class_get_help(*inst=None*)
Get the help string for this class in ReST format.

If *inst* is given, it's current trait values will be used in place of class defaults.

classmethod class_get_trait_help(*trait*, *inst=None*)
Get the help string for a single trait.

If *inst* is given, it's current trait values will be used in place of the class default.

classmethod class_own_trait_events(*name*)
Get a dict of all event handlers defined on this class, not a parent.

Works like `event_handlers`, except for excluding traits from parents.

classmethod class_own_traits(***metadata*)
Get a dict of all the traitlets defined on this class, not a parent.

Works like *class_traits*, except for excluding traits from parents.

classmethod class_print_help(*inst=None*)
Get the help string for a single trait and print it.

classmethod class_trait_names(***metadata*)
Get a list of all the names of this class' traits.

This method is just like the `trait_names()` method, but is unbound.

classmethod class_traits(***metadata*)
Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

This method is just like the `traits()` method, but is unbound.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

property cross_validation_lock
A contextmanager for running a block with our cross validation lock set to True.

At the end of the block, the lock's value is restored to its value prior to entering the block.

async do_restart(*auto=False*)
Called when the kernel has died

has_trait(*name*)
Returns True if the object has a trait with the specified name.

hold_trait_notifications()
Context manager for bundling trait change notifications and cross validation.

Use this when doing multiple trait assignments (init, config), to avoid race conditions in trait notifiers requesting other trait values. All trait notifications will fire after all values have been assigned.

observe (*handler*, *names=traitlets.All*, *type='change'*)
Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Parameters

- **handler** (*callable*) – A callable that is called when a trait changes. Its signature should be `handler(change)`, where `change` is a dictionary. The change dictionary at least holds a 'type' key. * `type`: the type of notification. Other keys may be passed depending on the value of 'type'. In the case where type is 'change', we also have the following keys: * `owner` : the HasTraits instance * `old` : the old value of the modified trait attribute * `new` : the new value of the modified trait attribute * `name` : the name of the modified trait attribute.
- **names** (*list, str, All*) – If names is All, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **type** (*str, All (default: 'change')*) – The type of notification to filter by. If equal to All, then all notifications are passed to the observe handler.

on_trait_change (*handler=None*, *name=None*, *remove=False*)
DEPRECATED: Setup a handler to be called when a trait changes.

This is used to setup dynamic notifications of trait changes.

Static handlers can be created by creating methods on a HasTraits subclass with the naming convention '_[traitname]_changed'. Thus, to create static handler for the trait 'a', create the method _a_changed(self, name, old, new) (fewer arguments can be used, see below).

If *remove* is True and *handler* is not specified, all change handlers for the specified name are uninstalled.

Parameters

- **handler** (*callable, None*) – A callable that is called when a trait changes. Its signature can be handler(), handler(name), handler(name, new), handler(name, old, new), or handler(name, old, new, self).
- **name** (*list, str, None*) – If None, the handler will apply to all traits. If a list of str, handler will apply to all names in the list. If a str, the handler will apply just to that name.
- **remove** (*bool*) – If False (the default), then install the handler. If True then unintall it.

remove_callback (*f*, *event*)
Unregister a callback from a particular event

Possible values for *event* are the same as in `add_callback()`.

classmethod section_names ()
return section names as a list

set_trait (*name*, *value*)
Forcibly sets trait attribute, including read-only attributes.

setup_instance (**args*, ***kwargs*)
This is called **before** self.__init__ is called.

start ()
Start the polling of the kernel.

stop()
Stop the kernel polling.

classmethod trait_events(*name=None*)
Get a `dict` of all the event handlers of this class.

Parameters name (*str (default: None)*) – The name of a trait of this class. If name is `None` then all the event handlers of this class will be returned instead.

Returns

Return type The event handlers associated with a trait name, or all event handlers.

trait_metadata(*traitname*, *key*, *default=None*)
Get metadata values for trait by key.

trait_names(***metadata*)
Get a list of all the names of this class' traits.

traits(***metadata*)
Get a `dict` of all the traits of this class. The dictionary is keyed on the name and the values are the TraitType objects.

The TraitTypes returned don't know anything about the values that the various HasTrait's instances are holding.

The metadata kwargs allow functions to be passed in which filter traits based on metadata values. The functions should take a single value as an argument and return a boolean. If any function returns False, then the trait is not included in the output. If a metadata key doesn't exist, None will be passed to the function.

unobserve(*handler*, *names=traitlets.All*, *type='change'*)
Remove a trait change handler.

This is used to unregister handlers to trait change notifications.

Parameters

- **handler** (*callable*) – The callable called when a trait attribute changes.
- **names** (*list, str, All (default: All)*) – The names of the traits for which the specified handler should be uninstalled. If names is All, the specified handler is uninstalled from the list of notifiers corresponding to all changes.
- **type** (*str or All (default: 'change')*) – The type of notification to filter by. If All, the specified handler is uninstalled from the list of notifiers corresponding to all types.

unobserve_all(*name=traitlets.All*)
Remove trait change handlers of any type for the specified name. If name is not specified, removes all trait notifiers.

update_config(*config*)
Update config and load the new values

CHAPTER

NINE

CHANGES IN JUPYTER KERNEL MGMT

9.1 0.5.1

- Enable support for python 3.5

9.2 0.5.0

- Tolerate missing status on kernel-info-reply
- Add some missing dependencies and min versions
- Force pin of pyzmq for travis builds, increase timeout
- Close kernel client in start_new_kernel tests
- Give kernel providers opportunity to load configuration
- Add support for kernel launch parameters
- Add support for native coroutines (async def)
- Significant documentation updates
- Fix operations on Windows
- Rework high-level APIs

9.3 0.4.0

- Remove unused import
- Make test fail because of idle-handling bug
- Fix watching for idle after execution
- Allow multiple providers to use kernelspec-based configurations
- Allow multiple providers to use kernelspec-based configurations

9.4 0.3.0

- Expose kernel_info_dict on blocking kernel client
- Translate tornado TimeoutError to base Python TimeoutError for blocking client
- Normalise contents of kernel info dicts
- Remove configurable kernel spec class and whitelist
- Remove deprecated method to install IPython kernel spec
- Remove special kernelspec handling for IPython
- Get rid of LoggingConfigurable base class
- Allow passing a different search path for KernelSpecProvider
- Catch unparseable kernelspecs when finding all
- Rework restarter events
- Don't try to relaunch a dead kernel with the same ports
- Rework message handlers API
- Use tornado event loop to dispatch restart callbacks
- Allow restarter to be used for manual restarts
- Support Python 3.4, missing JSONDecodeError

9.5 0.2.0

- Add kernel_info_dict attribute
- Don't use prerelease versions of test dependencies
- Return message even if status='error'
- Remove ErrorInKernel exception

9.6 0.1.1

- Initial experimental implementation.

Note: Because the code in this repository originated from jupyter_client you may also want to look at its changelog history.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX