Enterprise Gateway Documentation

Release 2.6.0

Project Jupyter team

Jan 31, 2022

USER DOCUMENTATION

Jupyter Enterprise Gateway is a pluggable framework that provides useful functionality for anyone supporting multiple users in a multi-cluster environment. Some of the core functionality it provides is better optimization of compute resources, improved multi-user support, and more granular security for your Jupyter notebook environment–making it suitable for enterprise, scientific, and academic implementations.

From a technical perspective, Jupyter Enterprise Gateway is a web server that enables the ability to launch kernels on behalf of remote notebooks. This leads to better resource management, as the web server is no longer the single location for kernel activity.

By default, Jupyter runs kernels locally - potentially exhausting the server of resources. By leveraging the functionality of the underlying resource management applications like Hadoop YARN, Kubernetes, etc., Jupyter Enterprise Gateway distributes kernels across the compute cluster, dramatically increasing the number of simultaneously active kernels.

Jupyter Enterprise Gateway leverages local resource managers to distribute kernels

CHAPTER

ONE

GETTING STARTED

Jupyter Enterprise Gateway requires Python (Python 3.3 or greater, or Python 2.7) and is intended to be installed on a node (typically the master node) of a managed cluster. Although its design center is for running kernels in Apache Spark 2.x clusters, clusters configured without Apache Spark are also acceptable.

The following Resource Managers are supported with the Jupyter Enterprise Gateway:

- Spark Standalone
- YARN Resource Manager - Client Mode
- YARN Resource Manager - Cluster Mode
- IBM Spectrum Conductor - Cluster Mode
- Kubernetes
- Docker Swarm

If you don't rely on a Resource Manager, you can use the Distributed mode which will connect a set of hosts via SSH.

The following kernels have been tested with the Jupyter Enterprise Gateway:

- Python/Apache Spark 2.x with IPython kernel
- Scala 2.11/Apache Spark 2.x with Apache Toree kernel
- R/Apache Spark 2.x with IRkernel

To support Scala kernels, Apache Toree is used. To support IPython kernels and R kernels, various packages have to be installed on each of the resource manager nodes. The simplest way to enable all the data nodes with required dependencies is to install Anaconda on all cluster nodes.

To take full advantage of security and user impersonation capabilities, a Kerberized cluster is recommended.

1.1 Enterprise Gateway Features

Jupyter Enterprise Gateway exposes the following features and functionality:

- Enables the ability to launch kernels on different servers thereby distributing resource utilization across the enterprise
- Pluggable framework allows for support of additional resource managers
- Secure communication from client to kernel
- Persistent kernel sessions (see Roadmap)
- Configuration profiles (see Roadmap)

- Feature parity with Jupyter Kernel Gateway's websocket-mode.
- A CLI for launching the enterprise gateway server: `jupyter enterprisegateway OPTIONS`
- A Python 2.7 and 3.3+ compatible implementation

1.2 Kernel Gateway vs. Enterprise Gateway

Enterprise Gateway was formerly built directly on Kernel Gateway. At that time, it had complete feature parity with Kernel Gateway. However, in order to address various roadmap items, Enterprise Gateway removed its dependency on Kernel Gateway, so now the question arises, when does one choose Enterprise Gateway over Kernel Gateway?

1.2.1 Use Enterprise Gateway if…

1. You have a large computer cluster consisting of limited resources (GPUs, large memory, etc) and users require those resources from notebooks
2. You have large numbers of users requiring access to a shared compute cluster
3. You require some amount of High Availability/Disaster Recovery such that another Gateway server can be spun up to service existing (and remote) kernels

1.2.2 Use Kernel Gateway if…

1. You have a small user pool where the resources of the Gateway server can be shared amongst those users (no remote kernel support)
2. You wish to configured the notebook-http mode functionality where a specific Notebook provides HTTP endpoints

Note that Enterprise Gateway also supports local kernels by default. However, HA/DR functionality won't be affective unless kernels run remotely.

1.3 Installing Enterprise Gateway

For new users, we **highly recommend** installing Anaconda. Anaconda conveniently installs Python, the Jupyter Notebook, the IPython kernel and other commonly used packages for scientific computing and data science.

Use the following installation steps:

- Download Anaconda. We recommend downloading Anaconda's latest Python version (currently Python 2.7 and Python 3.6).
- Install the version of Anaconda which you downloaded, following the instructions on the download page.
- Install the latest version of Jupyter Enterprise Gateway from PyPI or conda forge along with its dependencies.

```
# install using pip from pypi
pip install --upgrade jupyter_enterprise_gateway
```

```
# install using conda from conda forge
conda install -c conda-forge jupyter_enterprise_gateway
```

At this point, the Jupyter Enterprise Gateway deployment provides local kernel support which is fully compatible with Jupyter Kernel Gateway.

To uninstall Jupyter Enterprise Gateway…

```
#uninstall using pip
pip uninstall jupyter_enterprise_gateway
```

```
#uninstall using conda
conda uninstall jupyter_enterprise_gateway
```

1.4 Installing Kernels

To leverage the full distributed capabilities of Spark, Jupyter Enterprise Gateway has provided deep integration with various resource managers. Having said that, Enterprise Gateway also supports running in a pseudo-distributed mode utilizing for example both YARN client or Spark Standalone modes. We've also recently added Kubernetes, Docker Swarm and IBM Spectrum Conductor integrations.

Please follow the links below to learn specific details about how to enable/configure the different modes of deploying your kernels:

- *Distributed*
- *YARN Cluster Mode*
- *YARN Client Mode*
- *Spark Standalone*
- *Kubernetes*
- *Docker Swarm*
- *IBM Spectrum Conductor*
- *Standalone Remote Kernel Execution*

In each of the resource manager sections, we set the `KERNELS_FOLDER` to `/usr/local/share/jupyter/kernels` since that's one of the default locations searched by the Jupyter framework. Co-locating kernelspecs hierarchies in the same parent folder is recommended, although not required.

Depending on the resource manager, we detail in the related section the implemented kernel languages (python, scala, R…). The following kernels have been tested with the Jupyter Enterprise Gateway:

- Python/Apache Spark 2.x with IPython kernel
- Scala 2.11/Apache Spark 2.x with Apache Toree kernel
- R/Apache Spark 2.x with IRkernel

1.4.1 Important Requirements regarding the Nodes

We have three cases:

Case 1 - The kernel is run in via a container-based process proxy (Kubernetes, Docker or DockerSwarm)

In that case, the image should ensure the availability of the kernel libraries and kernelspec. The kernelspec is not necessary here, only the launcher. We talk about this in container customization.

The launch of containerized kernels via Enterprise Gateway is two-fold.

1. First, there's the argv section in the kernelspec that is processed by the server. In these cases, the command that is invoked is a python script using the target container's api (kubernetes, docker, or docker swarm) that is responsible for converting any necessary "parameters" to environment variables, etc. that are used during the actual container creation.
2. The command that is run in the container is actually the kernel launcher script. This script is responsible for taking the response address (which is now an env variable) and returning the kernel's connection information back on that response address to Enterprise Gateway. The kernel launcher does additional things - but primarily listens for interrupt and shutdown requests, which it then passes along to the actual (embedded) kernel.

So container environments have two launches - one to launch the container itself, the other to launch the kernel (within the container).

Case 2 - The kernel is run via DistributedProcessProxy

The kernelspecs are required on all nodes if using the DistributedProcessProxy - which apply to YARN Client mode, Standalone, and Distributed modes. All kernels (libraries…) and their corresponding kernelspecs must reside on each node.

The kernelspec hierarchies (i.e., paths) must be available and identical on all nodes.

IPython and IRkernel kernels must be installed on each node.

SSH passwordless is needed between the EG node and the other nodes.

Case 3 - The kernel is run via YarnClusterProcessProxy or ConductorClusterProcessProxy

With cluster process proxies, distribution of kernelspecs to all nodes besides the EG node is not required.

However, the IPython and IRkernel kernels must be installed on each node.

Note that because the Apache Toree kernel, and its supporting libraries, will be transferred to the target node via spark-submit, installation of Apache Toree (the scala kernel) is not required except on the Enterprise Gateway node itself.

1.4.2 Sample kernelspecs

We provide sample kernel configuration and launcher tar files as part of each release (e.g. jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz) that can be extracted and modified to fit your configuration.

For information about how to build your own kernel-based docker image for use by Enterprise Gateway see Custom kernel images.

1.5 Starting Enterprise Gateway

Very few arguments are necessary to minimally start Enterprise Gateway. The following command could be considered a minimal command and essentially provides functionality equal to Jupyter Kernel Gateway:

```
jupyter enterprisegateway --ip=0.0.0.0 --port_retries=0
```

where `--ip=0.0.0.0` exposes Enterprise Gateway on the public network and `--port_retries=0` ensures that a single instance will be started.

Please note that the ability to target resource-managed clusters (and use remote kernels) will require additional configuration settings depending on the resource manager. For additional information see the appropriate "Enabling . . . Support" section listed above.

We recommend starting Enterprise Gateway as a background task. As a result, you might find it best to create a start script to maintain options, file redirection, etc.

The following script starts Enterprise Gateway with `DEBUG` tracing enabled (default is `INFO`) and idle kernel culling for any kernels idle for 12 hours where idle check intervals occur every minute. The Enterprise Gateway log can then be monitored via `tail -F enterprise_gateway.log` and it can be stopped via `kill $(cat enterprise_gateway.pid)`

```
#!/bin/bash

LOG=/var/log/enterprise_gateway.log
PIDFILE=/var/run/enterprise_gateway.pid

jupyter enterprisegateway --ip=0.0.0.0 --port_retries=0 --log-level=DEBUG > $LOG 2>&1 &
if [ "$?" -eq 0 ]; then
  echo $! > $PIDFILE
else
  exit 1
fi
```

1.6 Connecting a Notebook to Enterprise Gateway

To leverage the benefits of Enterprise Gateway, it's helpful to redirect a Notebook server's kernel management to the Gateway server. This allows better separation of the user's notebooks from the managed computer cluster (Kubernetes, Hadoop YARN, Docker Swarm, etc.) on which Enterprise Gateway resides. A Notebook server can be configured to relay kernel requests to an Enterprise Gateway server in two ways - depending on the version of Notebook you're using.

1.6.1 Notebook 6.0 (and above)

With the Notebook 6.0 release, the NB2KG server extension (see next section) is built directly into the Notebook server. As a result, the steps for installing and configuring the server extension are no longer necessary.

To start the notebook server from the command line, the following will redirect kernel management request to the Gateway server running at <ENTERPRISE_GATEWAY_HOST_IP>:

```
jupyter notebook --gateway-url=http://<ENTERPRISE_GATEWAY_HOST_IP>:8888 --GatewayClient.
↪http_user=guest --GatewayClient.http_pwd=guest-password
```

If you have Notebook already in a docker image, a corresponding docker invocation would look something like this:

```
docker run -t --rm \
  -e JUPYTER_GATEWAY_URL='http://<master ip>:8888' \
  -e JUPYTER_GATEWAY_HTTP_USER=guest \
  -e JUPYTER_GATEWAY_HTTP_PWD=guest-password \
  -e JUPYTER_GATEWAY_VALIDATE_CERT='false' \
  -e LOG_LEVEL=DEBUG \
  -p 8888:8888 \
  -v ${HOME}/notebooks/:/tmp/notebooks \
  -w /tmp/notebooks \
  notebook-docker-image
```

Notebook files residing in `${HOME}/notebooks` can then be accessed via `http://localhost:8888`.

1.6.2 NB2KG Server Extension

For Notebook versions prior to 6.0, the NB2KG server extension is used to connect a Notebook from a local desktop or laptop to the Enterprise Gateway instance. Please refer to the NB2KG repository's README file for installation instructions.

Extending the notebook launch command listed on the NB2KG repo, one might use the following…

```
export KG_URL=http://<ENTERPRISE_GATEWAY_HOST_IP>:8888
export KG_HTTP_USER=guest
export KG_HTTP_PASS=guest-password
export KERNEL_USERNAME=${KG_HTTP_USER}
jupyter notebook \
  --NotebookApp.session_manager_class=nb2kg.managers.SessionManager \
  --NotebookApp.kernel_manager_class=nb2kg.managers.RemoteKernelManager \
  --NotebookApp.kernel_spec_manager_class=nb2kg.managers.RemoteKernelSpecManager
```

CHAPTER

TWO

SYSTEM ARCHITECTURE

Below are sections presenting details of the Enterprise Gateway internals and other related items. While we will attempt to maintain its consistency, the ultimate answers are in the code itself.

2.1 Enterprise Gateway Process Proxy Extensions

Enterprise Gateway is follow-on project to Jupyter Kernel Gateway with additional abilities to support remote kernel sessions on behalf of multiple users within resource-managed frameworks such as Apache Hadoop YARN or Kubernetes. Enterprise Gateway introduces these capabilities by extending the existing class hierarchies for `AsyncKernelManager` and `AsyncMultiKernelManager` classes, along with an additional abstraction known as a *process proxy*.

2.1.1 Overview

At its basic level, a running kernel consists of two components for its communication - a set of ports and a process.

Kernel Ports

The first component is a set of five zero-MQ ports used to convey the Jupyter protocol between the Notebook and the underlying kernel. In addition to the 5 ports, is an IP address, a key, and a signature scheme indicator used to interpret the key. These eight pieces of information are conveyed to the kernel via a json file, known as the connection file.

In today's Jupyter Kernel Gateway implementation, the IP address must be a local IP address meaning that the kernel cannot be remote from the kernel gateway. The enforcement of this restriction is down in the `jupyter_client` module - two levels below JKG.

This component is the core communication mechanism between the Notebook and the kernel. All aspects, including life-cycle management, can occur via this component. The kernel process (below) comes into play only when port-based communication becomes unreliable or additional information is required.

Kernel Process

When a kernel is launched, one of the fields of the kernel's associated kernel specification is used to identify a command to invoke. In today's implementation, this command information, along with other environment variables (also described in the kernel specification), is passed to `popen()` which returns a process class. This class supports four basic methods following its creation:

1. `poll()` to determine if the process is still running
2. `wait()` to block the caller until the process has terminated
3. `send_signal(signum)` to send a signal to the process
4. `kill()` to terminate the process

As you can see, other forms of process communication can be achieved by abstracting the launch mechanism.

2.1.2 Kernel Specifications

The primary vehicle for indicating a given kernel should be handled in a different manner is the kernel specification, otherwise known as the *kernel spec*. Enterprise Gateway leverages the natively extensible `metadata` stanza to introduce a new stanza named `process_proxy`.

The `process_proxy` stanza identifies the class that provides the kernel's process abstraction (while allowing for future extensions). This class then provides the kernel's lifecycle management operations relative to the managed resource or functional equivalent.

Here's an example of a kernel specification that uses the `DistributedProcessProxy` class for its abstraction:

```
{
  "language": "scala",
  "display_name": "Spark - Scala (YARN Client Mode)",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy"
    }
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "__TOREE_SPARK_OPTS__": "--master yarn --deploy-mode client --name ${KERNEL_ID:-
↪ERROR__NO__KERNEL_ID}",
    "__TOREE_OPTS__": "",
    "LAUNCH_OPTS": "",
    "DEFAULT_INTERPRETER": "Scala"
  },
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_scala_yarn_client/bin/run.sh",
    "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

See the *Process Proxy* section for more details on process proxies and those provided as part of the Enterprise Gateway release.

2.2 Remote Mapping Kernel Manager

`RemoteMappingKernelManager` is a subclass of Notebook's `MappingKernelManager` and provides two functions.

1. It provides the vehicle for making the `RemoteKernelManager` class known and available.
2. It overrides `start_kernel` to look at the target kernel's kernel spec to see if it contains a remote process proxy class entry. If so, it records the name of the class in its member variable to be made avaiable to the kernel start logic.

2.3 Remote Kernel Manager

`RemoteKernelManager` is a subclass of jupyter_client's `IOLoopKernelManager` class and provides the primary integration points for remote process proxy invocations. It implements a number of methods which allow Enterprise Gateway to circumvent functionality that might otherwise be prevented. As a result, some of these overrides may not be necessary if lower layers of the Jupyter framework were modified. For example, some methods are required because Jupyter makes assumptions that the kernel process is local.

Its primary functionality, however, is to override the `_launch_kernel` method (which is the method closest to the process invocation) and instantiates the appropriate process proxy instance - which is then returned in place of the process instance used in today's implementation. Any interaction with the process then takes place via the process proxy.

Both `RemoteMappingKernelManager` and `RemoteKernelManager` class definitions can be found in remotemanager.py

2.4 Process Proxy

Process proxy classes derive from the abstract base class `BaseProcessProxyABC` - which defines the four basic process methods. There are two immediate subclasses of `BaseProcessProxyABC` - `LocalProcessProxy` and `RemoteProcessProxy`.

`LocalProcessProxy` is essentially a pass-through to the current implementation. KernelSpecs that do not contain a `process_proxy` stanza will use `LocalProcessProxy`.

`RemoteProcessProxy` is an abstract base class representing remote kernel processes. Currently, there are six built-in subclasses of `RemoteProcessProxy` …

- `DistributedProcessProxy` - largely a proof of concept class, `DistributedProcessProxy` is responsible for the launch and management of kernels distributed across and explicitly defined set of hosts using ssh. Hosts are determined via a round-robin algorithm (that we should make pluggable someday).
- `YarnClusterProcessProxy` - is responsible for the discovery and management of kernels hosted as yarn applications within a YARN-managed cluster.
- `KubernetesProcessProxy` - is responsible for the discovery and management of kernels hosted within a Kubernetes cluster.
- `DockerSwarmProcessProxy` - is responsible for the discovery and management of kernels hosted within a Docker Swarm cluster.

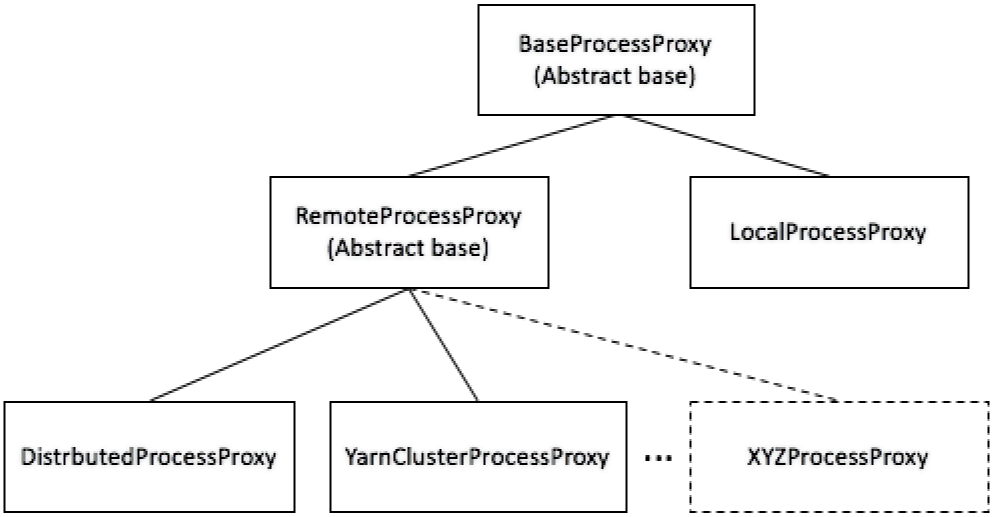
- DockerProcessProxy - is responsible for the discovery and management of kernels hosted within Docker configuration. Note: because these kernels will always run local to the corresponding Enterprise Gateway instance, these process proxies are of limited use.
- ConductorClusterProcessProxy - is responsible for the discovery and management of kernels hosted within an IBM Spectrum Conductor cluster.
- SparkOperatorProcessProxy - is responsible for the descovery and management of kernels hosted within a Kubernetes cluster but created as a SparkApplication instead of a Pod. The SparkApplication is a Kubernetes custom resource defined inside the project spark-on-k8s-operator, which makes all kinds of spark on k8s components better organized and easy to configure.

Before you run this kernel, please ensure that spark operator is installed under a certain namespace of your Kubernetes cluster.

You might notice that the last five process proxies do not necessarily control the *launch* of the kernel. This is because the native jupyter framework is utilized such that the script that is invoked by the framework is what launches the kernel against that particular resource manager. As a result, the *startup time* actions of these process proxies is more about discovering where the kernel *landed* within the cluster in order to establish a mechanism for determining lifetime. *Discovery* typically consists of using the resource manager's API to locate the kernel who's name includes its kernel ID in some fashion.

On the other hand, the DistributedProcessProxy essentially wraps the kernelspec argument vector (i.e., invocation string) in a remote shell since the host is determined by Enterprise Gateway, eliminating the discovery step from its implementation.

These class definitions can be found in the processproxies package. However, Enterprise Gateway is architected such that additonal process proxy implementations can be provided and are not required to be located within the Enterprise Gateway hierarchy - i.e., we embrace a *bring your own process proxy* model.



The process proxy constructor looks as follows:

```
def __init__(self, kernel_manager, proxy_config):
```

where

- kernel_manager is an instance of a RemoteKernelManager class.
- proxy_config is a dictionary of configuration values present in the kernel spec's json file. These values can

be used to override or amend various global configuration values on a per-kernel basis. See *Process Proxy Configuration* for more information.

```
@abstractmethod
def launch_process(self, kernel_cmd, *kw):
```

where

- kernel_cmd is a list (argument vector) that should be invoked to launch the kernel. This parameter is an artifact of the kernel manager _launch_kernel() method.
- **kw is a set key-word arguments which includes an env dictionary element consisting of the names and values of which environment variables to set at launch time.

The launch_process() method is the primary method exposed on the Process Proxy classes. It's responsible for performing the appropriate actions relative to the target type. The process must be in a running state prior to returning from this method - otherwise attempts to use the connections will not be successful since the (remote) kernel needs to have created the sockets.

All process proxy subclasses should ensure BaseProcessProxyABC.launch_process() is called - which will automatically place a variable named KERNEL_ID (consisting of the kernel's unique ID) into the corresponding kernel's environment variable list since KERNEL_ID is a primary mechanism for associating remote applications to a specific kernel instance.

```
def poll(self):
```

The poll() method is used by the Jupyter framework to determine if the process is still alive. By default, the framework's heartbeat mechanism calls poll() every 3 seconds. This method returns None if the process is still running, False otherwise (per the popen() contract).

```
def wait(self):
```

The wait() method is used by the Jupyter framework when terminating a kernel. Its purpose is to block return to the caller until the process has terminated. Since this could be a while, its best to return control in a reasonable amount of time since the kernel instance is destroyed anyway. This method does not return a value.

```
def send_signal(self, signum):
```

The send_signal() method is used by the Jupyter framework to send a signal to the process. Currently, SIGINT (2) (to interrupt the kernel) is the signal sent.

It should be noted that for normal processes - both local and remote - poll() and kill() functionality can be implemented via send_signal with signum values of 0 and 9, respectively.

This method returns None if the process is still running, False otherwise.

```
def kill(self):
```

The kill() method is used by the Jupyter framework to terminate the kernel process. This method is only necessary when the request to shutdown the kernel - sent via the control port of the zero-MQ ports - does not respond in an appropriate amount of time.

This method returns None if the process is killed successfully, False otherwise.

2.4.1 RemoteProcessProxy

As noted above, `RemoteProcessProxy` is an abstract base class that derives from `BaseProcessProxyABC`. Subclasses of `RemoteProcessProxy` must implement two methods - `confirm_remote_startup()` and `handle_timeout()`:

```python
@abstractmethod
def confirm_remote_startup(self, kernel_cmd, **kw):
```

where

- `kernel_cmd` is a list (argument vector) that should be invoked to launch the kernel. This parameter is an artifact of the kernel manager `_launch_kernel()` method.
- `**kw` is a set key-word arguments.

`confirm_remote_startup()` is responsible for detecting that the remote kernel has been appropriately launched and is ready to receive requests. This can include gather application status from the remote resource manager but is really a function of having received the connection information from the remote kernel launcher. (See *Kernel Launchers*)

```python
@abstractmethod
def handle_timeout(self):
```

`handle_timeout()` is responsible for detecting that the remote kernel has failed to startup in an acceptable time. It should be called from `confirm_remote_startup()`. If the timeout expires, `handle_timeout()` should throw HTTP Error 500 (`Internal Server Error`).

Kernel launch timeout expiration is expressed via the environment variable `KERNEL_LAUNCH_TIMEOUT`. If this value does not exist, it defaults to the Enterprise Gateway process environment variable `EG_KERNEL_LAUNCH_TIMEOUT` - which defaults to 30 seconds if unspecified. Since all `KERNEL_` environment variables "flow" from the Notebook server, the launch timeout can be specified as a client attribute of the Notebook session.

YarnClusterProcessProxy

As part of its base offering, Enterprise Gateway provides an implementation of a process proxy that communicates with the YARN resource manager that has been instructed to launch a kernel on one of its worker nodes. The node on which the kernel is launched is up to the resource manager - which enables an optimized distribution of kernel resources.

Derived from `RemoteProcessProxy`, `YarnClusterProcessProxy` uses the `yarn-api-client` library to locate the kernel and monitor its life-cycle. However, once the kernel has returned its connection information, the primary kernel operations naturally take place over the ZeroMQ ports.

This process proxy is reliant on the `--EnterpriseGatewayApp.yarn_endpoint` command line option or the `EG_YARN_ENDPOINT` environment variable to determine where the YARN resource manager is located. To accommodate increased flexibility, the endpoint definition can be defined within the process proxy stanza of the kernelspec, enabling the ability to direct specific kernels to different YARN clusters.

In cases where the YARN cluster is configured for high availability, then the `--EnterpriseGatewayApp.alt_yarn_endpoint` command line option or the `EG_ALT_YARN_ENDPOINT` environment variable should also be defined. When set, the underlying `yarn-api-client` library will choose the active Resource Manager between the two.

In cases where the YARN cluster is configured for high availability, then the `--EnterpriseGatewayApp.alt_yarn_endpoint` command line option or the `EG_ALT_YARN_ENDPOINT` environment variable should also be defined. When set, the underlying `yarn-api-client` library will choose the active Resource Manager between the two.

Note: If Enterprise Gateway is running on an edge node of the YARN cluster and has a valid `yarn-site.xml` file in HADOOP_CONF_DIR, neither of these values are required (default = None). In such cases, the `yarn-api-client` library will choose the active Resource Manager from the configuration files.

See Enabling YARN Cluster Mode Support for details.

DistributedProcessProxy

Like `YarnClusterProcessProxy`, Enterprise Gateway also provides an implementation of a basic round-robin remoting mechanism that is part of the `DistributedProcessProxy` class. This class uses the `--EnterpriseGatewayApp.remote_hosts` command line option (or `EG_REMOTE_HOSTS` environment variable) to determine on which hosts a given kernel should be launched. It uses a basic round-robin algorithm to index into the list of remote hosts for selecting the target host. It then uses ssh to launch the kernel on the target host. As a result, all kernelspec files must reside on the remote hosts in the same directory structure as on the Enterprise Gateway server.

It should be noted that kernels launched with this process proxy run in YARN *client* mode - so their resources (within the kernel process itself) are not managed by the YARN resource manager.

Like the yarn endpoint parameter the `remote_hosts` parameter can be specified within the process proxy configuration to override the global value - enabling finer-grained kernel distributions.

See Enabling YARN Client Mode or Spark Standalone Support for details.

KubernetesProcessProxy

With the popularity of Kubernetes within the enterprise, Enterprise Gateway now provides an implementation of a process proxy that communicates with the Kubernetes resource manager via the Kubernetes API. Unlike the other offerings, in the case of Kubernetes, Enterprise Gateway is itself deployed within the Kubernetes cluster as a *Service* and *Deployment*. The primary vehicle by which this is accomplished is via the enterprise-gateway.yaml file that contains the necessary metadata to define its deployment.

See Enabling Kubernetes Support for details.

DockerSwarmProcessProxy

Enterprise Gateway provides an implementation of a process proxy that communicates with the Docker Swarm resource manager via the Docker API. When used, the kernels are launched as swarm services and can reside anywhere in the managed cluster. To leverage kernels configured in this manner, Enterprise Gateway can be deployed either as a Docker Swarm *service* or a traditional Docker container.

A similar `DockerProcessProxy` implementation has also been provided. When used, the corresponding kernel will be launched as a traditional docker container that runs local to the launching Enterprise Gateway instance. As a result, its use has limited value.

See Enabling Docker Swarm Support for details.

ConductorClusterProcessProxy

Enterprise Gateway also provides an implementation of a process proxy that communicates with an IBM Spectrum Conductor resource manager that has been instructed to launch a kernel on one of its worker nodes. The node on which the kernel is launched is up to the resource manager - which enables an optimized distribution of kernel resources.

Derived from `RemoteProcessProxy`, `ConductorClusterProcessProxy` uses Conductor's REST-ful API to locate the kernel and monitor its life-cycle. However, once the kernel has returned its connection information, the primary kernel operations naturally take place over the ZeroMQ ports.

This process proxy is reliant on the `--EnterpriseGatewayApp.conductor_endpoint` command line option or the `EG_CONDUCTOR_ENDPOINT` environment variable to determine where the Conductor resource manager is located.

See Enabling IBM Spectrum Conductor Support for details.

CustomResourceProcessProxy

Enterprise Gateway also provides a implementation of a process proxy derived from `KubernetesProcessProxy` called `CustomResourceProcessProxy`.

Instead of creating kernels based on a Kubernetes pod, `CustomResourceProcessProxy` manages kernels via a custom resource definition (CRD). For example, `SparkApplication` is a CRD that includes many components of a Spark-on-Kubernetes application.

If you are going to extend `CustomResourceProcessProxy`, just follow steps below:

- override custom resource related variables(i.e. `group`, `version` and `plural` and `get_container_status` method, wrt spark_operator.py.
- define a jinja template like sparkoperator.k8s.io-v1beta2.yaml.j2. As a generic design, the template file should be named as {crd_group}-{crd_version} so that you can reuse launch_custom_resource.py in the kernelspec.
- define a kernelspec like spark_python_operator/kernel.json.

2.4.2 Process Proxy Configuration

Each kernel.json's `process-proxy` stanza can specify an optional `config` stanza that is converted into a dictionary of name/value pairs and passed as an argument to the each process-proxy constructor relative to the class identified by the `class_name` entry.

How each dictionary entry is interpreted is completely a function of the constructor relative to that process-proxy class or its super-class. For example, an alternate list of remote hosts has meaning to the `DistributedProcessProxy` but not to its super-classes. As a result, the super-class constructors will not attempt to interpret that value.

In addition, certain dictionary entries can override or amend system-level configuration values set on the command-line, thereby allowing administrators to tune behaviors down to the kernel level. For example, an administrator might want to constrain python kernels configured to use specific resources to an entirely different set of hosts (and ports) that other remote kernels might be targeting in order to isolate valuable resources. Similarly, an administrator might want to only authorize specific users to a given kernel.

In such situations, one might find the following `process-proxy` stanza:

```
{
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy",
```

(continues on next page)

(continued from previous page)

```
      "config": {
        "remote_hosts": "priv_host1,priv_host2",
        "port_range": "40000..41000",
        "authorized_users": "bob,alice"
      }
    }
  }
}
```

In this example, the kernel associated with this kernel.json file is relegated to hosts `priv_host1` and `priv_host2` where kernel ports will be restricted to a range between `40000` and `41000` and only users `bob` and `alice` can launch such kernels (provided neither appear in the global set of `unauthorized_users` since denial takes precedence).

For a current enumeration of which system-level configuration values can be overridden or amended on a per-kernel basis see Per-kernel Configuration Overrides.

2.5 Kernel Launchers

As noted above a kernel is considered started once the `launch_process()` method has conveyed its connection information back to the Enterprise Gateway server process. Conveyance of connection information from a remote kernel is the responsibility of the remote kernel *launcher*.

Kernel launchers provide a means of normalizing behaviors across kernels while avoiding kernel modifications. Besides providing a location where connection file creation can occur, they also provide a ‘hook’ for other kinds of behaviors - like establishing virtual environments or sandboxes, providing collaboration behavior, adhering to port range restrictions, etc.

There are four primary tasks of a kernel launcher:

1. Creation of the connection file and ZMQ ports on the remote (target) system along with a *gateway listener* socket
2. Conveyance of the connection (and listener socket) information back to the Enterprise Gateway process
3. Invocation of the target kernel
4. Listen for interrupt and shutdown requests from Enterprise Gateway and carry out the action when appropriate

Kernel launchers are minimally invoked with three parameters (all of which are conveyed by the `argv` stanza of the corresponding kernel.json file) - the kernel’s ID as created by the server and conveyed via the placeholder `{kernel_id}`, a response address consisting of the Enterprise Gateway server IP and port on which to return the connection information similarly represented by the placeholder `{response_address}`, and a public-key used by the launcher to encrypt an AES key that encrypts the kernel’s connection information back to the server and respresented by the placeholder `{public_key}`.

The kernel’s id is identified by the parameter `--RemoteProcessProxy.kernel-id`. Its value (`{kernel_id}`) is essentially used to build a connection file to pass to the to-be-launched kernel, along with any other things - like log files, etc.

The response address is identified by the parameter `--RemoteProcessProxy.response-address`. Its value (`{response_address}`) consists of a string of the form `<IPV4:port>` where the IPV4 address points back to the Enterprise Gateway server - which is listening for a response on the provided port. The port’s default value is `8877`, but can be specified via the environment variable `EG_RESPONSE_PORT`.

The public key is identified by the parameter `--RemoteProcessProxy.public-key`. Its value (`{public_key}`) is used to encrypt an AES key created by the launcher to encrypt the kernel’s connection information. The server, upon receipt of the response, uses the corresponding private key to decrypt the AES key, which it then uses to decrypt the connection information. Both the public and private keys are ephemeral; created upon Enterprise Gateway's startup. They can be ephemeral because they are only needed during a kernel's startup and never again.

Here's a kernel.json file illustrating these parameters…

```
{
  "language": "python",
  "display_name": "Spark - Python (YARN Cluster Mode)",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.yarn.
↪YarnClusterProcessProxy"
    }
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "SPARK_OPTS": "--master yarn --deploy-mode cluster --name ${KERNEL_ID:-ERROR__NO__
↪KERNEL_ID} --conf spark.yarn.submit.waitAppCompletion=false",
    "LAUNCH_OPTS": ""
  },
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_python_yarn_cluster/bin/run.sh",
    "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

Other options supported by launchers include:

- `--RemoteProcessProxy.port-range {port_range}` - passes configured port-range to launcher where launcher applies that range to kernel ports. The port-range may be configured globally or on a per-kernelspec basis, as previously described.
- `--RemoteProcessProxy.spark-context-initialization-mode [lazy|eager|none]` - indicates the *timeframe* in which the spark context will be created.
  - `lazy` (default) attempts to defer initialization as late as possible - although can vary depending on the underlying kernel and launcher implementation.
  - `eager` attempts to create the spark context as soon as possible.
  - `none` skips spark context creation altogether.

  Note that some launchers may not be able to support all modes. For example, the scala launcher uses the Toree kernel - which currently assumes a spark context will exist. As a result, a mode of `none` doesn't apply. Similarly, the `lazy` and `eager` modes in the Python launcher are essentially the same, with the spark context creation occurring immediately, but in the background thereby minimizing the kernel's startup time.

Kernel.json files also include a `LAUNCH_OPTS:` section in the `env` stanza to allow for custom parameters to be conveyed in the launcher's environment. `LAUNCH_OPTS` are then referenced in the run.sh script as the initial arguments to the launcher (see launch_ipykernel.py) …

```
eval exec \
     "${SPARK_HOME}/bin/spark-submit" \
```

(continues on next page)

(continued from previous page)

```
    "${SPARK_OPTS}" \
    "${PROG_HOME}/scripts/launch_ipykernel.py" \
    "${LAUNCH_OPTS}" \
    "$@"
```

2.6 Extending Enterprise Gateway

Theoretically speaking, enabling a kernel for use in other frameworks amounts to the following:

1. Build a kernel specification file that identifies the process proxy class to be used.
2. Implement the process proxy class such that it supports the four primitive functions of `poll()`, `wait()`, `send_signal(signum)` and `kill()` along with `launch_process()`.
3. If the process proxy corresponds to a remote process, derive the process proxy class from `RemoteProcessProxy` and implement `confirm_remote_startup()` and `handle_timeout()`.
4. Insert invocation of a launcher (if necessary) which builds the connection file and returns its contents on the `{response_address}` socket and following the encryption protocol set forth in the other launchers.

CHAPTER

THREE

SECURITY FEATURES

Jupyter Enterprise Gateway does not currently perform user *authentication* but, instead, assumes that all users issuing requests have been previously authenticated. Recommended applications for this are Apache Knox or perhaps even Jupyter Hub (e.g., if gateway-enabled notebook servers were spawned targeting an Enterprise Gateway cluster).

This section introduces some of the security features inherent in Enterprise Gateway (with more to come).

KERNEL_USERNAME

In order to convey the name of the authenicated user, `KERNEL_USERNAME` should be sent in the kernel creation request via the `env:` entry. This will occur automatically within the gateway-enabled Notebook server since it propagates all environment variables prefixed with `KERNEL_`. If the request does not include a `KERNEL_USERNAME` entry, one will be added to the kernel's launch environment with the value of the gateway user.

This value is then used within the *authorization* and *impersonation* functionality.

3.1 Authorization

By default, all users are authorized to start kernels. This behavior can be adjusted when situations arise where more control is required. Basic authorization can be expressed in two ways.

3.1.1 Authorized Users

The command-line or configuration file option: `EnterpriseGatewayApp.authorized_users` can be specified to contain a list of user names indicating which users are permitted to launch kernels within the current gateway server.

On each kernel launched, the authorized users list is searched for the value of `KERNEL_USERNAME` (case-sensitive). If the user is found in the list the kernel's launch sequence continues, otherwise HTTP Error 403 (Forbidden) is raised and the request fails.

Warning: Since the `authorized_users` option must be exhaustive, it should be used only in situations where a small and limited set of users are allowed access and empty otherwise.

3.1.2 Unauthorized Users

The command-line or configuration file option: `EnterpriseGatewayApp.unauthorized_users` can be specified to contain a list of user names indicating which users are **NOT** permitted to launch kernels within the current gateway server. The `unauthorized_users` list is always checked prior to the `authorized_users` list. If the value of `KERNEL_USERNAME` appears in the `unauthorized_users` list, the request is immediately failed with the same 403 (Forbidden) HTTP Error.

From a system security standpoint, privileged users (e.g., `root` and any users allowed `sudo` privileges) should be added to this option.

3.1.3 Authorization Failures

It should be noted that the corresponding messages logged when each of the above authorization failures occur are slightly different. This allows the administrator to discern from which authorization list the failure was generated.

Failures stemming from *inclusion* in the `unauthorized_users` list will include text similar to the following:

```
User 'bob' is not authorized to start kernel 'Spark - Python (YARN Client Mode)'. Ensure
KERNEL_USERNAME is set to an appropriate value and retry the request.
```

Failures stemming from *exclusion* from a non-empty `authorized_users` list will include text similar to the following:

```
User 'bob' is not in the set of users authorized to start kernel 'Spark - Python (YARN␣
↪Client Mode)'. Ensure
KERNEL_USERNAME is set to an appropriate value and retry the request.
```

3.2 User Impersonation

The Enterprise Gateway server leverages other technologies to implement user impersonation when launching kernels. This option is configured via two pieces of information: `EG_IMPERSONATION_ENABLED` and `KERNEL_USERNAME`.

`EG_IMPERSONATION_ENABLED` indicates the intention that user impersonation should be performed and can also be conveyed via the command-line boolean option `EnterpriseGatewayApp.impersonation_enabled` (default = False).

`KERNEL_USERNAME` is also conveyed within the environment of the kernel launch sequence where its value is used to indicate the user that should be impersonated.

3.2.1 Impersonation in YARN Cluster Mode

In a cluster managed by the YARN resource manager, impersonation is implemented by leveraging kerberos, and thus require this security option as a pre-requisite for user impersonation. When user impersonation is enabled, kernels are launched with the `--proxy-user ${KERNEL_USERNAME}` which will tell YARN to launch the kernel in a container used by the provided user name.

Note that, when using kerberos in a YARN managed cluster, the gateway user (`elyra` by default) needs to be set up as a `proxyuser` superuser in hadoop configuration. Please refer to the Hadoop documentation regarding the proper configuration steps.

3.2.2 SPNEGO Authentication to YARN APIs

When kerberos is enabled in a YARN managed cluster, the administration uis can be configured to require authentication/authorization via SPENEGO. When running Enterprise Gateway in a environment configured this way, we need to convey an extra configuration to enable the proper authorization when communicating with YARN via the YARN APIs.

`YARN_ENDPOINT_SECURITY_ENABLED` indicates the requirement to use SPNEGO authentication/authorization when connecting with the YARN APIs and can also be conveyed via the command-line boolean option `EnterpriseGatewayApp.yarn_endpoint_security_enabled` (default = False)

3.2.3 Impersonation in Standalone or YARN Client Mode

Impersonation performed in standalone or YARN cluster modes tends to take the form of using `sudo` to perform the kernel launch as the target user. This can also be configured within the run.sh script and requires the following:

1. The gateway user (i.e., the user in which Enterprise Gateway is running) must be enabled to perform sudo operations on each potential host. This enablement must also be done to prevent password prompts since Enterprise Gateway runs in the background. Refer to your operating system documentation for details.
2. Each user identified by `KERNEL_USERNAME` must be associated with an actual operating system user on each host.
3. Once the gateway user is configured for `sudo` privileges it is **strongly recommended** that that user be included in the set of `unauthorized_users`. Otherwise, kernels not configured for impersonation, or those requests that do not include `KERNEL_USERNAME`, will run as the, now, highly privileged gateway user!

WARNING: Should impersonation be disabled after granting the gateway user elevated privileges, it is **strongly recommended** those privileges be revoked (on all hosts) prior to starting kernels since those kernels will run as the gateway user **regardless of the value of KERNEL_USERNAME**.

3.3 SSH Tunneling

Jupyter Enterprise Gateway is configured to perform SSH tunneling on the five ZeroMQ kernel sockets as well as the communication socket created within the launcher and used to perform remote and cross-user signalling functionality. SSH tunneling is NOT enabled by default. Tunneling can be enabled/disabled via the environment variable `EG_ENABLE_TUNNELING=False`. Note, there is no command-line or configuration file support for this variable.

Note that SSH by default validates host keys before connecting to remote hosts and the connection will fail for invalid or unknown hosts. Enterprise Gateway honors this requirement, and invalid or unknown hosts will cause tunneling to fail. Please perform necessary steps to validate all hosts before enabling SSH tunneling, such as:

- SSH to each node cluster and accept the host key properly
- Configure SSH to disable `StrictHostKeyChecking`

3.4 Using Generic Security Service (Kerberos)

Jupyter Enterprise Gateway has support for SSH connections using GSS (for example Kerberos), which enables its deployment without the use of an ssh key. The `EG_REMOTE_GSS_SSH` environment variable can be used to control this behavior.

See list of additional supported environment variables.

3.5 Securing Enterprise Gateway Server

3.5.1 Using SSL for encrypted communication

Enterprise Gateway supports Secure Sockets Layer (SSL) communication with its clients. With SSL enabled, all the communication between the server and client are encrypted and highly secure.

1. You can start Enterprise Gateway to communicate via a secure protocol mode by setting the `certfile` and `keyfile` options with the command:

   ```
   jupyter enterprisegateway --ip=0.0.0.0 --port_retries=0 --certfile=mycert.pem --keyfile=mykey.key
   ```

   As server starts up, the log should reflect the following,

   ```
   [EnterpriseGatewayApp] Jupyter Enterprise Gateway at https://localhost:8888
   ```

   Note: Enterprise Gateway server is started with `HTTPS` instead of `HTTP`, meaning server side SSL is enabled.

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

   ```
   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout mykey.key -out mycert.pem
   ```

2. With Enterprise Gateway server SSL enabled, now you need to configure the client side SSL, which is accomplished via the Gateway configuration options embedded in Notebook server.

   During Jupyter Notebook server startup, export the following environment variables where the gateway-enabled server has access during runtime:

   ```
   export JUPYTER_GATEWAY_CLIENT_CERT=${PATH_TO_PEM_FILE}
   export JUPYTER_GATEWAY_CLIENT_KEY=${PATH_TO_KEY_FILE}
   export JUPYTER_GATEWAY_CA_CERTS=${PATH_TO_SELFSIGNED_CA}
   ```

   Note: If using a self-signed certificate, you can set `JUPYTER_GATEWAY_CA_CERTS` same as `JUPYTER_GATEWAY_CLIENT_CERT`.

3.5.2 Using Enterprise Gateway configuration file

You can also utilize the Enterprise Gateway configuration file to set static configurations for the server.

1. If you do not already have a configuration file, generate a Enterprise Gateway configuration file by running the following command:

   ```
   jupyter enterprisegateway --generate-config
   ```

2. By default, the configuration file will be generated `~/.jupyter/jupyter_enterprise_gateway_config.py`.

3. By default, all the configuration fields in `jupyter_enterprise_gateway_config.py` are commented out. To enable SSL from the configuration file, modify the corresponding parameter to the appropriate value.

   ```
   s,c.KernelGatewayApp.certfile = '/absolute/path/to/your/certificate/fullchain.pem'
   s,c.KernelGatewayApp.keyfile = '/absolute/path/to/your/certificate/privatekey.key'
   ```

4. Using configuration file achieves the same result as starting the server with `--certfile` and `--keyfile`, this way provides better readability and debuggability.

After configuring the above, the communication between gateway-enabled Notebook Server and Enterprise Gateway is SSL enabled.

CHAPTER

FOUR

ANCILLARY FEATURES

This page points out some features and functionality worthy of your attention but not necessarily part of the Jupyter Enterprise Gateway implementation.

4.1 Culling idle kernels

With the adoption of notebooks and interactive development for data science, a new “resource utilization” pattern has arisen, where kernel resources are locked for a given notebook, but due to interactive development process it might be idle for a long period of time causing the cluster resources to starve. One way to workaround this problem is to enable culling of idle kernels after a specific timeout period.

Idle kernel culling is set to “off” by default. It’s enabled by setting `--MappingKernelManager.cull_idle_timeout` to a positive value representing the number of seconds a kernel must remain idle to be culled (default: 0, recommended: 43200, 12 hours).

You can also configure the interval that the kernels are checked for their idle timeouts by adjusting the setting `--MappingKernelManager.cull_interval` to a positive value. If the interval is not set or set to a non-positive value, the system uses 300 seconds as the default value: (default: 300 seconds).

There are use-cases where we would like to enable only culling of idle kernels that have no connections (e.g. the notebook browser was closed without stopping the kernel first), this can be configured by adjusting the setting `--MappingKernelManager.cull_connected` (default: False).

Here’s an updated start script that provides some default configuration to enable the culling of idle kernels:

```
#!/bin/bash

LOG=/var/log/enterprise_gateway.log
PIDFILE=/var/run/enterprise_gateway.pid

jupyter enterprisegateway --ip=0.0.0.0 --port_retries=0 --log-level=DEBUG \
   --MappingKernelManager.cull_idle_timeout=43200 --MappingKernelManager.cull_
↪interval=60 > $LOG 2>&1 &

if [ "$?" -eq 0 ]; then
  echo $! > $PIDFILE
else
  exit 1
fi
```

4.2 Installing Python modules from within notebook cell

To be able to honor user isolation in a multi-tenant world, installing Python modules using `pip` from within a Notebook Cell should be done using the `--user` command-line option as shown below:

```
!pip install --user <module-name>
```

This results in the Python module to be installed in `$USER/.local/lib/python<version>/site-packages` folder. `PYTHONPATH` environment variable defined in `kernel.json` must include `$USER/.local/lib/python<version>/site-packages` folder so that the newly installed module can be successfully imported in a subsequent Notebook Cell as shown below:

```
import <module-name>
```

CHAPTER

FIVE

USE CASES

Jupyter Enterprise Gateway addresses specific use cases for different personas. We list a few below:

- **As an administrator**, I want to fix the bottleneck on the Jupyter Kernel Gateway server due to large number of kernels running on it and the size of each kernel (spark driver) process, by deploying the Enterprise Gateway, such that kernels can be launched as managed resources within YARN, distributing the resource-intensive driver processes across the YARN cluster, while still allowing the data analysts to leverage the compute power of a large YARN cluster.
- **As an administrator**, I want to have some user isolation such that user processes are protected against each other and user can preserve and leverage their own environment, i.e. libraries and/or packages.
- **As a data scientist**, I want to run my notebook using the Enterprise Gateway such that I can free up resources on my own laptop and leverage my company's large YARN cluster to run my compute-intensive jobs.
- **As a solution architect**, I want to explore supporting a different resource manager with Enterprise Gateway, e.g. Kubernetes, by extending and implementing a new ProcessProxy class such that I can easily take advantage of specific functionality provided by the resource manager.
- **As an administrator**, I want to constrain applications to specific port ranges so I can more easily identify issues and manage network configurations that adhere to my corporate policy.
- **As an administrator**, I want to constrain the number of active kernels that each of my users can have at any given time.
- **As a solution architect**, I want to easily integrate the ability to launch remote kernels with existing platforms, so I can leverage my compute cluster in a customizable way.

CHAPTER

SIX

LOCAL MODE

The Local deployment can be useful for local development and is not meant to be run in production environments as it subjects the gateway server to resource exhaustion.

If you just want to try EG in a local setup, you can use the following kernelspec (no need for a launcher):

```
{
  "display_name": "Python 3 Local",
  "language": "python",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.processproxy.
↪LocalProcessProxy"
    }
  },
  "argv": [
    "python",
    "-m",
    "ipykernel_launcher",
    "-f",
    "{connection_file}"
  ]
}
```

`process_proxy` is optional (if Enterprise Gateway encounters a kernelspec without the `process_proxy` stanza, it will treat that kernelspec as if it contained `LocalProcessProxy`).

Side note: You can run a Local kernel in *Distributed mode* by setting `remote_hosts` to the localhost. Why would you do that?

1. One reason is that it decreases the window in which a port conflict can occur since the 5 kernel ports are created by the launcher (within the same process and therefore closer to the actual invocation of the kernel) rather than by the server prior to the launch of the kernel process.
2. The second reason is that auto-restarted kernels - when an issue occurs - say due to a port conflict - will create a new set of ports rather than try to re-use the same set that produced the failure in the first place. In this case, you'd want to use the per-kernel configuration approach and set `remote_hosts` in the config stanza of the `process_proxy` stanza (using the stanza instead of the global `EG_REMOTE_HOSTS` allows you to not interfere with the other resource managers configuration, e.g. Spark Standalone or YARN Client kernels - Those other kernels need to be able to continue leveraging the full cluster nodes).

CHAPTER

SEVEN

DISTRIBUTED MODE

This page describes the approach taken for integrating Enterprise Gateway into a distributed set of hosts, without cluster resource managers.

The following sample kernelspecs are currently available on Distributed mode:

- python_distributed

Install the `python_distributed` kernelspec on all nodes.

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/python_distributed/ python_distributed/
```

The `python_distributed` kernelspec uses `DistributedProcessProxy` which is responsible for the launch and management of kernels distributed across and explicitly defined set of hosts using ssh. Hosts are determined via a round-robin algorithm (that we should make pluggable someday).

The set of remote hosts are derived from two places.

- The configuration option: EnterpriseGatewayApp.remote_hosts who's default value comes from the env variable EG_REMOTE_HOSTS - which, itself, defaults to 'localhost'.
- The config option can be overridden on a per-kernel basis if the process_proxy stanza contains a config stanza where there's a remote_hosts entry. If present, this value will be used instead.

You have to ensure passwordless SSH from the EG host to all other hosts for the user under which EG is run.

CHAPTER EIGHT

YARN CLUSTER MODE

To leverage the full distributed capabilities of Jupyter Enterprise Gateway, there is a need to provide additional configuration options in a cluster deployment.

The following sample kernelspecs are currently available on YARN cluster:

- spark_R_yarn_cluster
- spark_python_yarn_client
- spark_scala_yarn_client

The distributed capabilities are currently based on an Apache Spark cluster utilizing YARN as the Resource Manager and thus require the following environment variables to be set to facilitate the integration between Apache Spark and YARN components:

- SPARK_HOME: Must point to the Apache Spark installation path

```
SPARK_HOME:/usr/hdp/current/spark2-client                            #For HDP␣
↪distribution
```

- EG_YARN_ENDPOINT: Must point to the YARN Resource Manager endpoint if remote from YARN cluster

```
EG_YARN_ENDPOINT=http://${YARN_RESOURCE_MANAGER_FQDN}:8088/ws/v1/cluster #Common to YARN␣
↪deployment
```

Note: If Enterprise Gateway is using an applicable HADOOP_CONF_DIR that contains a valid `yarn-site.xml` file, then this config value can remain unset (default = None) and the YARN client library will locate the appropriate Resource Manager from the configuration. This is also true in cases where the YARN cluster is configured for high availability.

If Enterprise Gateway is remote from the YARN cluster (i.e., no HADOOP_CONF_DIR) and the YARN cluster is configured for high availability, then the alternate endpoint should also be specified…

```
EG_ALT_YARN_ENDPOINT=http://${ALT_YARN_RESOURCE_MANAGER_FQDN}:8088/ws/v1/cluster #Common␣
↪to YARN deployment
```

8.1 Configuring Kernels for YARN Cluster mode

For each supported Jupyter Kernel, we have provided sample kernel configurations and launchers as part of the release jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz.

Considering we would like to enable the IPython Kernel that comes pre-installed with Anaconda to run on Yarn Cluster mode, we would have to copy the sample configuration folder **spark_python_yarn_cluster** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "python3" | awk '{print $2}')"
KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"
mkdir $KERNELS_FOLDER/spark_python_yarn_cluster/
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_python_yarn_cluster/ spark_python_yarn_cluster/
```

After that, you should have a `kernel.json` that looks similar to the one below:

```
{
  "language": "python",
  "display_name": "Spark - Python (YARN Cluster Mode)",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.yarn.
↪YarnClusterProcessProxy"
    }
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/conda/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.6-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.local/
↪lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/current/
↪spark2-client/python/lib/py4j-0.10.6-src.zip,PATH=/opt/conda/bin:$PATH",
    "SPARK_OPTS": "--master yarn --deploy-mode cluster --name ${KERNEL_ID:-ERROR__NO__
↪KERNEL_ID} --conf spark.yarn.submit.waitAppCompletion=false",
    "LAUNCH_OPTS": ""
  },
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_python_yarn_cluster/bin/run.sh",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

8.2 Scala Kernel (Apache Toree kernel)

We have tested the latest version of Apache Toree with Scala 2.11 support. Please note that the Apache Toree kernel is now bundled in the kernelspecs tar file for each of the Scala kernelspecs provided by Enterprise Gateway.

Follow the steps below to install/configure the Toree kernel:

Install Apache Toree Kernelspecs

Considering we would like to enable the Scala Kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_scala_yarn_cluster** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_scala_yarn_cluster/ spark_scala_yarn_cluster/
```

For more information about the Scala kernel, please visit the Apache Toree page.

8.3 Installing support for Python (IPython kernel)

The IPython kernel comes pre-installed with Anaconda and we have tested with its default version of IPython kernel.

Update the IPython Kernelspecs

Considering we would like to enable the IPython kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_python_yarn_cluster** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_python_yarn_cluster/ spark_python_yarn_cluster/
```

For more information about the IPython kernel, please visit the IPython kernel page.

8.4 Installing support for R (IRkernel)

Install IRkernel

Perform the following steps on Jupyter Enterprise Gateway hosting system as well as all YARN workers

```
conda install --yes --quiet -c r r-essentials r-irkernel r-argparse
# Create an R-script to run and install packages and update IRkernel
cat <<'EOF' > install_packages.R
install.packages(c('repr', 'IRdisplay', 'evaluate', 'git2r', 'crayon', 'pbdZMQ',
                   'devtools', 'uuid', 'digest', 'RCurl', 'curl', 'argparse'),
                   repos='http://cran.rstudio.com/')
devtools::install_github('IRkernel/IRkernel@0.8.14')
IRkernel::installspec(user = FALSE)
```

(continues on next page)

(continued from previous page)

```
EOF
# run the package install script
$ANACONDA_HOME/bin/Rscript install_packages.R
# OPTIONAL: check the installed R packages
ls $ANACONDA_HOME/lib/R/library
```

Update the IRkernel Kernelspecs

Considering we would like to enable the IRkernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_R_yarn_cluster** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_R_yarn_cluster/ spark_R_yarn_cluster/
```

For more information about the iR kernel, please visit the IRkernel page.

After making any necessary adjustments such as updating SPARK_HOME or other environment specific configuration, you now should have a new Kernel available which will use Jupyter Enterprise Gateway to execute your notebook cell contents in distributed mode on a Spark/Yarn Cluster.

CHAPTER

NINE

YARN CLIENT MODE

By default, Jupyter Enterprise Gateway provides feature parity with Jupyter Kernel Gateway's websocket-mode, which means that by installing kernels in Enterprise Gateway and using the vanilla kernelspecs created during installation you will have your kernels running in client mode with drivers running on the same host as Enterprise Gateway.

Having said that, even if you are not leveraging the full distributed capabilities of Jupyter Enterprise Gateway, client mode can still help mitigate resource starvation by enabling a pseudo-distributed mode, where kernels are started in different nodes of the cluster utilizing a round-robin algorithm. In this case, you can still experience bottlenecks on a given node that receives requests to start "large" kernels, but otherwise, you will be better off compared to when all kernels are started on a single node or as local processes, which is the default for vanilla Jupyter Notebook.

Please note also the YARN client mode can be considered as a *Distributed* mode. It just happen to use spark-submit form different nodes in the cluster but uses `DistributedProcessProxy`.

The following sample kernelspecs are currently available on YARN client:

- spark_R_yarn_client
- spark_python_yarn_client
- spark_scala_yarn_client

The pseudo-distributed capabilities are currently supported in YARN Client mode and require the following environment variables to be set:

- SPARK_HOME: Must point to the Apache Spark installation path

```
SPARK_HOME:/usr/hdp/current/spark2-client                           #For HDP␣
↪distribution
```

- EG_REMOTE_HOSTS must be set to a comma-separated set of FQDN hosts indicating the hosts available for running kernels. (This can be specified via the command line as well: `--EnterpriseGatewayApp.remote_hosts`)

```
EG_REMOTE_HOSTS=elyra-node-1.fyre.ibm.com,elyra-node-2.fyre.ibm.com,elyra-node-3.fyre.
↪ibm.com,elyra-node-4.fyre.ibm.com,elyra-node-5.fyre.ibm.com
```

For each supported Jupyter Kernel, we have provided sample kernel configurations and launchers as part of the release jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz.

Considering we would like to enable the IPython Kernel that comes pre-installed with Anaconda to run on Yarn Client mode, we would have to copy the sample configuration folder **spark_python_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "python3" | awk '{print $2}')"
KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"
tar -zxvf enterprise_gateway_kernelspecs.tar.gz --strip 1 --directory $KERNELS_FOLDER/
↪spark_python_yarn_client/ spark_python_yarn_client/
```

After that, you should have a kernel.json that looks similar to the one below:

```
{
  "language": "python",
  "display_name": "Spark - Python (YARN Client Mode)",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy"
    }
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/conda/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.6-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.local/
↪lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/current/
↪spark2-client/python/lib/py4j-0.10.6-src.zip,PATH=/opt/conda/bin:$PATH",
    "SPARK_OPTS": "--master yarn --deploy-mode client --name ${KERNEL_ID:-ERROR__NO__
↪KERNEL_ID} --conf spark.yarn.submit.waitAppCompletion=false",
    "LAUNCH_OPTS": ""
  },
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_python_yarn_client/bin/run.sh",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

After making any necessary adjustments such as updating SPARK_HOME or other environment specific configuration, you now should have a new Kernel available which will use Jupyter Enterprise Gateway to execute your notebook cell contents.

9.1 Scala Kernel (Apache Toree kernel)

We have tested the latest version of Apache Toree with Scala 2.11 support. Please note that the Apache Toree kernel is now bundled in the kernelspecs tar file for each of the Scala kernelspecs provided by Enterprise Gateway.

Follow the steps below to install/configure the Toree kernel:

Install Apache Toree Kernelspecs

Considering we would like to enable the Scala Kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_scala_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_scala_yarn_client/ spark_scala_yarn_client/
```

For more information about the Scala kernel, please visit the Apache Toree page.

9.2 Installing support for Python (IPython kernel)

The IPython kernel comes pre-installed with Anaconda and we have tested with its default version of IPython kernel.

Update the IPython Kernelspecs

Considering we would like to enable the IPython kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_python_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_python_yarn_client/ spark_python_yarn_client/
```

For more information about the IPython kernel, please visit the IPython kernel page.

9.3 Installing support for R (IRkernel)

Install IRkernel

Perform the following steps on Jupyter Enterprise Gateway hosting system as well as all YARN workers

```
conda install --yes --quiet -c r r-essentials r-irkernel r-argparse
# Create an R-script to run and install packages and update IRkernel
cat <<'EOF' > install_packages.R
install.packages(c('repr', 'IRdisplay', 'evaluate', 'git2r', 'crayon', 'pbdZMQ',
                   'devtools', 'uuid', 'digest', 'RCurl', 'curl', 'argparse'),
                   repos='http://cran.rstudio.com/')
devtools::install_github('IRkernel/IRkernel@0.8.14')
IRkernel::installspec(user = FALSE)
```

(continues on next page)

(continued from previous page)

```
EOF
# run the package install script
$ANACONDA_HOME/bin/Rscript install_packages.R
# OPTIONAL: check the installed R packages
ls $ANACONDA_HOME/lib/R/library
```

Update the IRkernel Kernelspecs

Considering we would like to enable the IRkernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_R_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_R_yarn_client/ spark_R_yarn_client/
```

For more information about the iR kernel, please visit the IRkernel page.

After making any necessary adjustments such as updating SPARK_HOME or other environment specific configuration, you now should have a new Kernel available which will use Jupyter Enterprise Gateway to execute your notebook cell contents in distributed mode on a Spark/Yarn Cluster.

CHAPTER

TEN

SPARK STANDALONE

By default, Jupyter Enterprise Gateway provides feature parity with Jupyter Kernel Gateway's websocket-mode, which means that by installing kernels in Enterprise Gateway and using the vanilla kernelspecs created during installation you will have your kernels running in client mode with drivers running on the same host as Enterprise Gateway.

Having said that, even if you are not leveraging the full distributed capabilities of Jupyter Enterprise Gateway, client mode can still help mitigate resource starvation by enabling a pseudo-distributed mode, where kernels are started in different nodes of the cluster utilizing a round-robin algorithm. In this case, you can still experience bottlenecks on a given node that receives requests to start "large" kernels, but otherwise, you will be better off compared to when all kernels are started on a single node or as local processes, which is the default for vanilla Jupyter Notebook.

The pseudo-distributed capabilities are currently supported in Spark Standalone and require the following environment variables to be set:

- SPARK_HOME: Must point to the Apache Spark installation path

```
SPARK_HOME:/usr/hdp/current/spark2-client                           #For HDP␣
↪distribution
```

- EG_REMOTE_HOSTS must be set to a comma-separated set of FQDN hosts indicating the hosts available for running kernels. (This can be specified via the command line as well: `--EnterpriseGatewayApp.remote_hosts`)

```
EG_REMOTE_HOSTS=elyra-node-1.fyre.ibm.com,elyra-node-2.fyre.ibm.com,elyra-node-3.fyre.
↪ibm.com,elyra-node-4.fyre.ibm.com,elyra-node-5.fyre.ibm.com
```

10.1 Configuring Kernels for Spark Standalone

Although Enterprise Gateway does not currently provide sample kernelspecs for Spark standalone, here are the steps necessary to convert a yarn_client kernelspec to standalone.

For each supported Jupyter Kernel, we have provided sample kernel configurations and launchers as part of the release jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz.

Considering we would like to enable the IPython Kernel that comes pre-installed with Anaconda to run on Spark Standalone, we would have to copy the sample configuration folder **spark_python_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list) and rename it to **spark_python_spark_standalone***

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "python3" | awk '{print $2}')"
KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"
```

(continues on next page)

(continued from previous page)

```
tar -zxvf enterprise_gateway_kernelspecs.tar.gz --strip 1 --directory $KERNELS_FOLDER/
↪spark_python_yarn_client/ spark_python_yarn_client/
mv $KERNELS_FOLDER/spark_python_yarn_client $KERNELS_FOLDER/spark_python_spark_standalone
```

You need to edit the kernel.json:

- Update the display_name with e.g. `Spark - Python (Spark Standalone)`.
- Update the `--master` option in the SPARK_OPTS to point to the spark master node rather than indicate `--deploy-mode client`.
- Update `SPARK_OPTS` and remove the `spark.yarn.submit.waitAppCompletion=false`.

After that, you should have a kernel.json that looks similar to the one below:

```
{
  "language": "python",
  "display_name": "Spark - Python (Spark Standalone)",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy"
    }
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/conda/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.6-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.local/
↪lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/current/
↪spark2-client/python/lib/py4j-0.10.6-src.zip,PATH=/opt/conda/bin:$PATH",
    "SPARK_OPTS": "--master spark://127.0.0.1:7077  --name ${KERNEL_ID:-ERROR__NO__
↪KERNEL_ID}",
    "LAUNCH_OPTS": ""
  },
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_python_spark_standalone/bin/run.sh",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

After making any necessary adjustments such as updating SPARK_HOME or other environment specific configuration, you now should have a new Kernel available which will use Jupyter Enterprise Gateway to execute your notebook cell contents.

10.2 Scala Kernel (Apache Toree kernel)

We have tested the latest version of Apache Toree with Scala 2.11 support. Please note that the Apache Toree kernel is now bundled in the kernelspecs tar file for each of the Scala kernelspecs provided by Enterprise Gateway.

Follow the steps below to install/configure the Toree kernel:

Install Apache Toree Kernelspecs

Considering we would like to enable the Scala Kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_scala_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_scala_yarn_client/ spark_scala_yarn_client/
mv $KERNELS_FOLDER/spark_scala_yarn_client $KERNELS_FOLDER/spark_scala_spark_standalone
```

For more information about the Scala kernel, please visit the Apache Toree page.

10.3 Installing support for Python (IPython kernel)

The IPython kernel comes pre-installed with Anaconda and we have tested with its default version of IPython kernel.

Update the IPython Kernelspecs

Considering we would like to enable the IPython kernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_python_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_python_yarn_client/ spark_python_yarn_client/
mv $KERNELS_FOLDER/spark_python_yarn_client $KERNELS_FOLDER/spark_python_spark_standalone
```

For more information about the IPython kernel, please visit the IPython kernel page.

10.4 Installing support for R (IRkernel)

Install IRkernel

Perform the following steps on Jupyter Enterprise Gateway hosting system as well as all YARN workers

```
conda install --yes --quiet -c r r-essentials r-irkernel r-argparse
# Create an R-script to run and install packages and update IRkernel
cat <<'EOF' > install_packages.R
install.packages(c('repr', 'IRdisplay', 'evaluate', 'git2r', 'crayon', 'pbdZMQ',
                   'devtools', 'uuid', 'digest', 'RCurl', 'curl', 'argparse'),
                   repos='http://cran.rstudio.com/')
```

(continues on next page)

(continued from previous page)

```
devtools::install_github('IRkernel/IRkernel@0.8.14')
IRkernel::installspec(user = FALSE)
EOF
# run the package install script
$ANACONDA_HOME/bin/Rscript install_packages.R
# OPTIONAL: check the installed R packages
ls $ANACONDA_HOME/lib/R/library
```

Update the IRkernel Kernelspecs

Considering we would like to enable the IRkernel to run on YARN Cluster and Client mode we would have to copy the sample configuration folder **spark_R_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_
↪enterprise_gateway_kernelspecs-2.5.0.tar.gz
KERNELS_FOLDER=/usr/local/share/jupyter/kernels
tar -zxvf jupyter_enterprise_gateway_kernelspecs-2.5.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_R_yarn_client/ spark_R_yarn_client/
mv $KERNELS_FOLDER/spark_R_yarn_client $KERNELS_FOLDER/spark_R_spark_standalone
```

For more information about the iR kernel, please visit the IRkernel page.

After making any necessary adjustments such as updating SPARK_HOME or other environment specific configuration, you now should have a new Kernel available which will use Jupyter Enterprise Gateway to execute your notebook cell contents in distributed mode on a Spark/Yarn Cluster.

CHAPTER

ELEVEN

KUBERNETES

This page describes the approach taken for integrating Enterprise Gateway into an existing Kubernetes cluster.

In this solution, Enterprise Gateway is, itself, provisioned as a Kubernetes *deployment* and exposed as a Kubernetes *service*. In this way, Enterprise Gateway can leverage load balancing and high availability functionality provided by Kubernetes (although HA cannot be fully realized until EG supports persistent sessions).

The following sample kernelspecs are currently available on Kubernetes:

- R_kubernetes
- python_kubernetes
- python_tf_gpu_kubernetes
- python_tf_kubernetes
- scala_kubernetes
- spark_R_kubernetes
- spark_python_kubernetes
- spark_scala_kubernetes

As with all kubernetes deployments, Enterprise Gateway is built into a docker image. The base Enterprise Gateway image is elyra/enterprise-gateway and can be found in the Enterprise Gateway dockerhub organization elyra, along with other kubernetes-based images. See Runtime Images for image details.

When deployed within a spark-on-kubernetes cluster, Enterprise Gateway can easily support cluster-managed kernels distributed across the cluster. Enterprise Gateway will also provide standalone (i.e., *vanilla*) kernel invocation (where spark contexts are not automatically created) which also benefits from their distribution across the cluster.

11.1 Enterprise Gateway Deployment

Enterprise Gateway manifests itself as a Kubernetes deployment, exposed externally by a Kubernetes service. It is identified by the name `enterprise-gateway` within the cluster. In addition, all objects related to Enterprise Gateway, including kernel instances, have the kubernetes label of `app=enterprise-gateway` applied.

The service is currently configured as type `NodePort` but is intended for type `LoadBalancer` when appropriate network plugins are available. Because kernels are stateful, the service is also configured with a `sessionAffinity` of `ClientIP`. As a result, kernel creation requests will be routed to different deployment instances (see deployment) thereby diminishing the need for a `LoadBalancer` type. Here's the service yaml entry from enterprise-gateway.yaml:

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: enterprise-gateway
  name: enterprise-gateway
  namespace: enterprise-gateway
spec:
  ports:
  - name: http
    port: 8888
    targetPort: 8888
  - name: response
    port: 8877
    targetPort: 8877
  selector:
    gateway-selector: enterprise-gateway
  sessionAffinity: ClientIP
  type: NodePort
```

The deployment yaml essentially houses the pod description. By increasing the number of `replicas` a configuration can experience instant benefits of distributing Enterprise Gateway instances across the cluster. This implies that once session persistence is provided, we should be able to provide highly available (HA) kernels. Here's the yaml portion from enterprise-gateway.yaml that defines the Kubernetes deployment and pod (some items may have changed):

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: enterprise-gateway
  namespace: enterprise-gateway
  labels:
    gateway-selector: enterprise-gateway
    app: enterprise-gateway
    component: enterprise-gateway
spec:
# Uncomment/Update to deploy multiple replicas of EG
#  replicas: 1
  selector:
    matchLabels:
      gateway-selector: enterprise-gateway
  template:
    metadata:
      labels:
        gateway-selector: enterprise-gateway
        app: enterprise-gateway
        component: enterprise-gateway
    spec:
      # Created above.
      serviceAccountName: enterprise-gateway-sa
      containers:
      - env:
        - name: EG_PORT
          value: "8888"
```

(continues on next page)

(continued from previous page)

```
    - name: EG_RESPONSE_PORT
      value: "8877"

      # Created above.
    - name: EG_NAMESPACE
      value: "enterprise-gateway"

      # Created above.  Used if no KERNEL_NAMESPACE is provided by client.
    - name: EG_KERNEL_CLUSTER_ROLE
      value: "kernel-controller"

      # All kernels reside in the EG namespace if True, otherwise KERNEL_NAMESPACE
      # must be provided or one will be created for each kernel.
    - name: EG_SHARED_NAMESPACE
      value: "False"

    - name: EG_TUNNELING_ENABLED
      value: "False"
    - name: EG_CULL_IDLE_TIMEOUT
      value: "600"
    - name: EG_LOG_LEVEL
      value: "DEBUG"
    - name: EG_KERNEL_LAUNCH_TIMEOUT
      value: "60"
    - name: EG_KERNEL_WHITELIST
      value: "'r_kubernetes','python_kubernetes','python_tf_kubernetes','scala_
↪kubernetes','spark_r_kubernetes','spark_python_kubernetes','spark_scala_kubernetes'"
    - name: EG_DEFAULT_KERNEL_NAME
      value: "python_kubernetes"
    # Optional authorization token passed in all requests (see --
↪EnterpriseGatewayApp.auth_token)
    #- name: EG_AUTH_TOKEN
    #  value: <configured-auth-token>

    # Ensure the following VERSION tag is updated to the version of Enterprise␣
↪Gateway you wish to run
    image: elyra/enterprise-gateway:VERSION
    # k8s will only pull :latest all the time.
    # the following line will make sure that :VERSION is always pulled
    # You should remove this if you want to pin EG to a release tag
    imagePullPolicy: Always
    name: enterprise-gateway
    args: ["--gateway"]
    ports:
    - containerPort: 8888
    - containerPort: 8877
```

11.1.1 Namespaces

A best practice for Kubernetes applications running in an enterprise is to isolate applications via namespaces. Since Enterprise Gateway also requires isolation at the kernel level, it makes sense to use a namespace for each kernel, by default.

The initial namespace is created in the `enterprise-gateway.yaml` file using a default name of `enterprise-gateway`. This name is communicated to the EG application via the env variable `EG_NAMESPACE`. All Enterprise Gateway components reside in this namespace.

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: enterprise-gateway
  namespace: enterprise-gateway
```

By default, kernel namespaces are created when the respective kernel is launched. At that time, the kernel namespace name is computed from the kernel username (`KERNEL_USERNAME`) and its Id (`KERNEL_ID`) just like the kernel pod name. Upon a kernel's termination, this namespace - provided it was created by Enterprise Gateway - will be deleted.

Installations wishing to pre-create the kernel namespace can do so by conveying the name of the kernel namespace via `KERNEL_NAMESPACE` in the `env` portion of the kernel creation request. (They must also provide the namespace's service account name via `KERNEL_SERVICE_ACCOUNT_NAME` - see next section.) When `KERNEL_NAMESPACE` is set, Enterprise Gateway will not attempt to create a kernel-specific namespace, nor will it attempt its deletion. As a result, kernel namespace lifecycle management is the user's responsibility.

Although **not recommended**, installations requiring everything in the same namespace - Enterprise Gateway and all its kernels - can do so by setting env `EG_SHARED_NAMESPACE` to `True`. When set, all kernels will run in the enterprise gateway namespace, essentially eliminating all aspects of isolation between kernel instances.

11.1.2 Role-Based Access Control (RBAC)

Another best practice of Kubernetes applications is to define the minimally viable set of permissions for the application. Enterprise Gateway does this by defining role-based access control (RBAC) objects for both Enterprise Gateway and kernels.

Because the Enterprise Gateway pod must create kernel namespaces, pods, services (for Spark support) and rolebindings, a cluster-scoped role binding is required. The cluster role binding `enterprise-gateway-controller` also references the subject, `enterprise-gateway-sa`, which is the service account associated with the Enterprise Gateway namespace and also created by the yaml file.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: enterprise-gateway-sa
  namespace: enterprise-gateway
  labels:
    app: enterprise-gateway
    component: enterprise-gateway
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: enterprise-gateway-controller
  labels:
```

(continues on next page)

(continued from previous page)

```
    app: enterprise-gateway
    component: enterprise-gateway
rules:
  - apiGroups: [""]
    resources: ["pods", "namespaces", "services", "configmaps", "secrets",
↪"persistentvolumnes", "persistentvolumeclaims"]
    verbs: ["get", "watch", "list", "create", "delete"]
  - apiGroups: ["rbac.authorization.k8s.io"]
    resources: ["rolebindings"]
    verbs: ["get", "list", "create", "delete"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: enterprise-gateway-controller
  labels:
    app: enterprise-gateway
    component: enterprise-gateway
subjects:
  - kind: ServiceAccount
    name: enterprise-gateway-sa
    namespace: enterprise-gateway
roleRef:
  kind: ClusterRole
  name: enterprise-gateway-controller
  apiGroup: rbac.authorization.k8s.io
```

The `enterprise-gateway.yaml` file also defines the minimally viable roles for a kernel pod - most of which are required for Spark support. Since kernels, by default, reside within their own namespace created upon their launch, a cluster role is used within a namespace-scoped role binding created when the kernel's namespace is created. The name of the kernel cluster role is `kernel-controller` and, when Enterprise Gateway creates the namespace and role binding, is also the name of the role binding instance.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kernel-controller
  labels:
    app: enterprise-gateway
    component: kernel
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "watch", "list", "create", "delete"]
```

As noted above, installations wishing to pre-create their own kernel namespaces should provide the name of the service account associated with the namespace via `KERNEL_SERVICE_ACCOUNT_NAME` in the `env` portion of the kernel creation request (along with `KERNEL_NAMESPACE`). If not provided, the built-in namespace service account, `default`, will be referenced. In such circumstances, Enterprise Gateway will **not** create a role binding on the name for the service account, so it is the user's responsibility to ensure that the service account has the capability to perform equivalent operations as defined by the `kernel-controller` role.

Here's an example of the creation of a custom namespace (`kernel-ns`) with its own service account (`kernel-sa`) and role binding (`kernel-controller`) that references the cluster-scoped role (`kernel-controller`) and includes appropriate labels to help with administration and analysis:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: kernel-ns
  labels:
    app: enterprise-gateway
    component: kernel
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: kernel-sa
  namespace: kernel-ns
  labels:
    app: enterprise-gateway
    component: kernel
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kernel-controller
  namespace: kernel-ns
  labels:
    app: enterprise-gateway
    component: kernel
subjects:
  - kind: ServiceAccount
    name: kernel-sa
    namespace: kernel-ns
roleRef:
  kind: ClusterRole
  name: kernel-controller
  apiGroup: rbac.authorization.k8s.io
```

11.1.3 Kernel Image Puller

Because kernels now reside within containers and its typical for the first reference of a container to trigger its pull from a docker repository, kernel startup requests can easily timeout whenever the kernel image is first accessed on any given node. To mitigate this issue, Enterprise Gateway deployment includes a DaemonSet object named `kernel-image-puller` or KIP. This object is responsible for polling Enterprise Gateway for the current set of configured kernelspecs, picking out any configured image name references, and pulling those images to the node on which KIP is running. Because its a daemon set, this will also address the case when new nodes are added to a configuration.

The Kernel Image Puller can be configured for the interval at which it checks for new kernelspecs (`KIP_INTERVAL`), the number of puller threads it will utilize per node (`KIP_NUM_PULLERS`), the number of retries it will attempt for a given image (`KIP_NUM_RETRIES`), and the pull policy (`KIP_PULL_POLICY`) - which essentially dictates whether it will attempt to pull images that its already encoutnered (`Always`) vs. only pulling the image if it hasn't seen it yet (`IfNotPresent`).

If the Enterprise Gateway defines an authentication token (`EG_AUTH_TOKEN`) then that same token should be configured here as (`KIP_AUTH_TOKEN`) so that the puller can correctly authenticate its requests.

The Kernel Image Puller also supports multiple container runtimes since Docker is no longer configured by default in Kubernetes. KIP currently supports Docker and Containerd runtimes. If another runtime is encountered, KIP will try to proceed using the Containerd client `crictl` against the configured socket. As a result, it is import that the `criSocket` value be appropriately configured relative to the container runtime. If the runtime is something other than Docker or Containerd and `crictl` isn't able to pull images, it may be necessary to manually pre-seed images or incur kernel start timeouts the first time a given node is asked to start a kernel associated with a non-resident image.

KIP also supports the notion of a *default container registry* whereby image names that do not specify a registry (e.g., `docker.io` or `quay.io`) KIP will apply the configured default. Ideally, the image name should be fully qualified.

Here's what the Kernel Image Puller looks like in the yaml...

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kernel-image-puller
  namespace: enterprise-gateway
spec:
  selector:
    matchLabels:
      name: kernel-image-puller
  template:
    metadata:
      labels:
        name: kernel-image-puller
        app: enterprise-gateway
        component: kernel-image-puller
    spec:
      containers:
      - name: kernel-image-puller
        image: elyra/kernel-image-puller:VERSION
           - name: KIP_LOG_LEVEL
             value: {{ .Values.logLevel }}
           - name: KIP_GATEWAY_HOST
             value: "http://enterprise-gateway.{{ .Release.Namespace }}:{{ .Values.port }
↪}"
           - name: KIP_INTERVAL
             value: !!str {{ .Values.kip.interval }}
           - name: KIP_PULL_POLICY
             value: {{ .Values.kip.pullPolicy }}
           - name: KIP_CRI_ENDPOINT
             value: "unix://{{ .Values.kip.criSocket }}"
           - name: KIP_DEFAULT_CONTAINER_REGISTRY
             value: {{ .Values.kip.defaultContainerRegistry }}
         # Optional authorization token passed in all requests (should match EG_AUTH_
↪TOKEN)
              {{- if .Values.authToken }}
        - name: KIP_AUTH_TOKEN
          value: {{ .Values.authToken }}
              {{- end }}
        volumeMounts:
           - name: cri-socket
             mountPath: !!str {{ .Values.kip.criSocket }}  # see env KIP_CRI_ENDPOINT
      volumes:
```

(continues on next page)

(continued from previous page)

```
        - name: cri-socket
          hostPath:
            path: {{ .Values.kip.criSocket }}
```

11.1.4 Kernelspec Modifications

One of the more common areas of customization we see occurs within the kernelspec files located in `/usr/local/share/jupyter/kernels`. To accommodate the ability to customize the kernel definitions, you have two different options: NFS mounts, or custom container images. The two options are mutually exclusive, because they mount kernelspecs into the same location in the Enterprise Gateway pod.

Via NFS

The kernels directory can be mounted as an NFS volume into the Enterprise Gateway pod, thereby making the kernelspecs available to all EG pods within the Kubernetes cluster (provided the NFS mounts exist on all applicable nodes).

As an example, we have included the necessary entries for mounting an existing NFS mount point into the Enterprise Gateway pod. By default, these references are commented out as they require the system administrator configure the appropriate NFS mounts and server IP. If you are deploying Enterprise Gateway via the Helm chart (see Deploying Enterprise Gateway, below), you can enable NFS directly via Helm values.

Here you can see how `enterprise-gateway.yaml` references use of the volume (via `volumeMounts` for the container specification and `volumes` in the pod specification):

```
    spec:
      containers:
      - env:
        - name: EG_NAMESPACE
          value: "enterprise-gateway"
        - name: EG_KERNEL_CLUSTER_ROLE
          value: "kernel-controller"
        - name: EG_SHARED_NAMESPACE
          value: "False"
        - name: EG_TUNNELING_ENABLED
          value: "False"
        - name: EG_CULL_IDLE_TIMEOUT
          value: "600"
        - name: EG_LOG_LEVEL
          value: "DEBUG"
        - name: EG_KERNEL_LAUNCH_TIMEOUT
          value: "60"
        - name: EG_KERNEL_WHITELIST
          value: "'r_kubernetes','python_kubernetes','python_tf_kubernetes','python_tf_
↪gpu_kubernetes','scala_kubernetes','spark_r_kubernetes','spark_python_kubernetes',
↪'spark_scala_kubernetes'"
        image: elyra/enterprise-gateway:VERSION
        name: enterprise-gateway
        args: ["--gateway"]
        ports:
        - containerPort: 8888
# Uncomment to enable NFS-mounted kernelspecs
```

(continues on next page)

(continued from previous page)

```
        volumeMounts:
        - name: kernelspecs
          mountPath: "/usr/local/share/jupyter/kernels"
      volumes:
      - name: kernelspecs
        nfs:
          server: <internal-ip-of-nfs-server>
          path: "/usr/local/share/jupyter/kernels"
```

Note that because the kernel pod definition file, kernel-pod.yaml, resides in the kernelspecs hierarchy, customizations to the deployments of future kernel instances can now also take place. In addition, these same entries can be added to the kernel-pod.yaml definitions if access to the same or other NFS mount points are desired within kernel pods. (We'll be looking at ways to make modifications to per-kernel configurations more manageable.)

Use of more formal persistent volume types must include the Persistent Volume and corresponding Persistent Volume Claim stanzas.

Via Custom Container Image

If you are deploying Enterprise Gateway via the Helm chart (see Deploying Enterprise Gateway, below), then instead of using NFS, you can build your custom kernelspecs into a container image that Enterprise Gateway consumes. Here's an example Dockerfile for such a container:

```
FROM alpine:3.9

COPY kernels /kernels
```

This assumes that your source contains a `kernels/` directory with all of the kernelspecs you'd like to end up in the image, e.g. `kernels/python_kubernetes/kernel.json` and any associated files.

Once you build your custom kernelspecs image and push it to a container registry, you can refer to it from your Helm deployment. For instance:

```
helm upgrade --install --atomic --namespace enterprise-gateway enterprise-gateway etc/kubernetes/helm --set kernelspecs.image=your-custom-image:latest
```

. . . where `your-custom-image:latest` is the image name and tag of your kernelspecs image. Once deployed, the Helm chart copies the data from the `/kernels` directory of your container into the `/usr/local/share/jupyter/kernels` directory of the Enterprise Gateway pod. Note that when this happens, the built-in kernelspecs are no longer available. So include all kernelspecs that you want to be available in your container image.

Also, you should update the Helm chart `kernel_whitelist` value with the name(s) of your custom kernelspecs.

11.2 Kubernetes Kernel Instances

There are essentially two kinds of kernels (independent of language) launched within an Enterprise Gateway Kubernetes cluster - *vanilla* and *spark-on-kubernetes* (if available).

When *vanilla* kernels are launched, Enterprise Gateway is responsible for creating the corresponding pod. On the other hand, *spark-on-kubernetes* kernels are launched via `spark-submit` with a specific `master` URI - which then creates the corresponding pod(s) (including executor pods). Images can be launched using both forms provided they have the appropriate support for Spark installed.

Here's the yaml configuration used when *vanilla* kernels are launched. As noted in the KubernetesProcessProxy section below, this file (kernel-pod.yaml.j2) serves as a template where each of the tags surrounded with {{ and }} represent variables that are substituted at the time of the kernel's launch. All {{ kernel_xxx }} parameters correspond to KERNEL_XXX environment variables that can be specified from the client in the kernel creation request's json body.

```
apiVersion: v1
kind: Pod
metadata:
  name: "{{ kernel_pod_name }}"
  namespace: "{{ kernel_namespace }}"
  labels:
    kernel_id: "{{ kernel_id }}"
    app: enterprise-gateway
    component: kernel
spec:
  restartPolicy: Never
  serviceAccountName: "{{ kernel_service_account_name }}"
  {% if kernel_uid is defined or kernel_gid is defined %}
  securityContext:
    {% if kernel_uid is defined %}
    runAsUser: {{ kernel_uid | int }}
    {% endif %}
    {% if kernel_gid is defined %}
    runAsGroup: {{ kernel_gid | int }}
    {% endif %}
    fsGroup: 100
  {% endif %}
  containers:
  - env:
    - name: EG_RESPONSE_ADDRESS
      value: "{{ eg_response_address }}"
    - name: EG_PUBLIC_KEY
      value: "{{ eg_public_key }}"
    - name: KERNEL_LANGUAGE
      value: "{{ kernel_language }}"
    - name: KERNEL_SPARK_CONTEXT_INIT_MODE
      value: "{{ kernel_spark_context_init_mode }}"
    - name: KERNEL_NAME
      value: "{{ kernel_name }}"
    - name: KERNEL_USERNAME
      value: "{{ kernel_username }}"
    - name: KERNEL_ID
      value: "{{ kernel_id }}"
    - name: KERNEL_NAMESPACE
      value: "{{ kernel_namespace }}"
    image: "{{ kernel_image }}"
    name: "{{ kernel_pod_name }}"
```

There are a number of items worth noting:

1. Kernel pods can be identified in three ways using kubectl:

    1. By the global label app=enterprise-gateway - useful when needing to identify all related objects (e.g., kubectl get all -l app=enterprise-gateway)

   2. By the *kernel_id* label `kernel_id=<kernel_id>` - useful when only needing specifics about a given kernel. This label is used internally by enterprise-gateway when performing its discovery and lifecycle management operations.

   3. By the *component* label `component=kernel` - useful when needing to identity only kernels and not other enterprise-gateway components. (Note, the latter can be isolated via `component=enterprise-gateway`.)

   Note that since kernels run in isolated namespaces by default, it's often helpful to include the clause `--all-namespaces` on commands that will span namespaces. To isolate commands to a given namespace, you'll need to add the namespace clause `--namespace <namespace-name>`.

2. Each kernel pod is named by the invoking user (via the `KERNEL_USERNAME` env) and its kernel_id (env `KERNEL_ID`). This identifier also applies to those kernels launched within `spark-on-kubernetes`.

3. Kernel pods use the specified `securityContext`. If env `KERNEL_UID` is not specified in the kernel creation request a default value of `1000` (the jovyan user) will be used. Similarly for `KERNEL_GID`, whose default is `100` (the users group). In addition, Enterprise Gateway enforces a lists of prohibited UID and GID values. By default, this list is initialized to the 0 (root) UID and GID. Administrators can configure the `EG_PROHIBITED_UIDS` and `EG_PROHIBITED_GIDS` environment variables via the enterprise-gateway.yaml file with comma-separated values to alter the set of user and group ids to be prevented.

4. As noted above, if `KERNEL_NAMESPACE` is not provided in the request, Enterprise Gateway will create a namespace using the same naming algorithm for the pod. In addition, the `kernel-controller` cluster role will be bound to a namespace-scoped role binding of the same name using the namespace's default service account as its subject. Users wishing to use their own kernel namespaces must provide **both** `KERNEL_NAMESPACE` and `KERNEL_SERVICE_ACCOUNT_NAME` as these are both used in the `kernel-pod.yaml.j2` as `{{ kernel_namespace }}` and `{{ kernel_service_account_name }}`, respectively.

5. Kernel pods have restart policies of `Never`. This is because the Jupyter framework already has built-in logic for auto-restarting failed kernels and any other restart policy would likely interfere with the built-in behaviors.

6. The parameters to the launcher that is built into the image are communicated via environment variables as noted in the `env:` section above.

11.3 Unconditional Volume Mounts

Unconditional volume mounts can be added in the `kernel-pod.yaml.j2` template. An example of these unconditional volume mounts can be found when extending docker shared memory. For some I/O jobs the pod will need more than the default `64mb` of shared memory on the `/dev/shm` path.

```
volumeMounts:
# Define any "unconditional" mounts here, followed by "conditional" mounts that vary per client
{% if kernel_volume_mounts is defined %}
  {% for volume_mount in kernel_volume_mounts %}
- {{ volume_mount }}
  {% endfor %}
{% endif %}
volumes:
# Define any "unconditional" volumes here, followed by "conditional" volumes that vary per client
{% if kernel_volumes is defined %}
{% for volume in kernel_volumes %}
- {{ volume }}
```

(continues on next page)

(continued from previous page)

```
{% endfor %}
{% endif %}
```

The conditional volumes are handled by the loops inside of the yaml file. Any unconditional volumes can be added before these conditions. In the scenario where the `/dev/shm` will need to be expanded the following mount has to be added.

```
volumeMounts:
# Define any "unconditional" mounts here, followed by "conditional" mounts that vary per client
- mountPath: /dev/shm
  name: dshm
{% if kernel_volume_mounts is defined %}
  {% for volume_mount in kernel_volume_mounts %}
- {{ volume_mount }}
  {% endfor %}
{% endif %}
volumes:
# Define any "unconditional" volumes here, followed by "conditional" volumes that vary per client
- name: dshm
emptyDir:
  medium: Memory
{% if kernel_volumes is defined %}
{% for volume in kernel_volumes %}
- {{ volume }}
{% endfor %}
{% endif %}
```

11.4 Kubernetes Resource Quotas

When deploying kernels on a Kubernetes cluster a best practice is to define request and limit quotas for CPUs, GPUs, and Memory. These quotas can be defined from the client via KERNEL_-prefixed environment variables which are passed through to the kernel at startup.

- `KERNEL_CPUS` - CPU Request by Kernel
- `KERNEL_MEMORY` - MEMORY Request by Kernel
- `KERNEL_GPUS` - GPUS Request by Kernel
- `KERNEL_CPUS_LIMIT` - CPU Limit
- `KERNEL_MEMORY_LIMIT` - MEMORY Limit
- `KERNEL_GPUS_LIMIT` - GPUS Limit

Memory and CPU units are based on the Kubernetes Official Documentation while GPU is using the NVIDIA `nvidia.com/gpu` parameter. The desired units should be included in the variable's value.

When defined, these variables are then substituted into the appropriate location of the corresponding kernel-pod.yaml.j2 template.

```
{% if kernel_cpus is defined or kernel_memory is defined or kernel_gpus is defined or␣
↪kernel_cpus_limit is defined or kernel_memory_limit is defined or kernel_gpus_limit is␣
↪defined %}
  resources:
    {% if kernel_cpus is defined or kernel_memory is defined or kernel_gpus is defined %}
    requests:
      {% if kernel_cpus is defined %}
      cpu: "{{ kernel_cpus }}"
      {% endif %}
      {% if kernel_memory is defined %}
      memory: "{{ kernel_memory }}"
      {% endif %}
      {% if kernel_gpus is defined %}
      nvidia.com/gpu: "{{ kernel_gpus }}"
      {% endif %}
    {% endif %}
    {% if kernel_cpus_limit is defined or kernel_memory_limit is defined or kernel_gpus_
↪limit is defined %}
    limits:
      {% if kernel_cpus_limit is defined %}
      cpu: "{{ kernel_cpus_limit }}"
      {% endif %}
      {% if kernel_memory_limit is defined %}
      memory: "{{ kernel_memory_limit }}"
      {% endif %}
      {% if kernel_gpus_limit is defined %}
      nvidia.com/gpu: "{{ kernel_gpus_limit }}"
      {% endif %}
    {% endif %}
  {% endif %}
```

11.5 KubernetesProcessProxy

To indicate that a given kernel should be launched into a Kubernetes configuration, the kernel.json file's `metadata` stanza must include a `process_proxy` stanza indicating a `class_name:` of `KubernetesProcessProxy`. This ensures the appropriate lifecycle management will take place relative to a Kubernetes environment.

Along with the `class_name:` entry, this process proxy stanza should also include a proxy configuration stanza which specifies the docker image to associate with the kernel's pod. If this entry is not provided, the Enterprise Gateway implementation will use a default entry of `elyra/kernel-py:VERSION`. In either case, this value is made available to the rest of the parameters used to launch the kernel by way of an environment variable: `KERNEL_IMAGE`.

(Please note that the use of `VERSION` in docker image tags is a placeholder for the appropriate version-related image tag. When kernelspecs are built via the Enterprise Gateway Makefile, `VERSION` is replaced with the appropriate version denoting the target release. A full list of available image tags can be found in the dockerhub repository corresponding to each image.)

```
{
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.k8s.
↪KubernetesProcessProxy",
```

(continues on next page)

(continued from previous page)

```
      "config": {
        "image_name": "elyra/kernel-py:VERSION"
      }
    }
  }
}
```

As always, kernels are launched by virtue of the `argv:` stanza in their respective kernel.json files. However, when launching *vanilla* kernels in a kubernetes environment, what gets invoked isn't the kernel's launcher, but, instead, a python script that is responsible for using the Kubernetes Python API to create the corresponding pod instance. The pod is *configured* by applying the values to each of the substitution parameters into the kernel-pod.yaml file previously displayed. This file resides in the same `scripts` directory as the kubernetes launch script - `launch_kubernetes.py` - which is referenced by the kernel.json's `argv:` stanza:

```
{
  "argv": [
    "python",
    "/usr/local/share/jupyter/kernels/python_kubernetes/scripts/launch_kubernetes.py",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

By default, *vanilla* kernels use a value of `none` for the spark context initialization mode so no context will be created automatically.

When the kernel is intended to target *Spark-on-kubernetes*, its launch is very much like kernels launched in YARN *cluster mode*, albeit with a completely different set of parameters. Here's an example `SPARK_OPTS` string value which best conveys the idea:

```
  "SPARK_OPTS": "--master k8s://https://${KUBERNETES_SERVICE_HOST}:${KUBERNETES_SERVICE_
↪PORT} --deploy-mode cluster --name ${KERNEL_USERNAME}-${KERNEL_ID} --conf spark.
↪kubernetes.driver.label.app=enterprise-gateway --conf spark.kubernetes.driver.label.
↪kernel_id=${KERNEL_ID} --conf spark.kubernetes.executor.label.app=enterprise-gateway --
↪conf spark.kubernetes.executor.label.kernel_id=${KERNEL_ID} --conf spark.kubernetes.
↪driver.docker.image=${KERNEL_IMAGE} --conf spark.kubernetes.executor.docker.
↪image=kubespark/spark-executor-py:v2.5.0-kubernetes-0.5.0 --conf spark.kubernetes.
↪submission.waitAppCompletion=false",
```

Note that each of the labels previously discussed are also applied to the *driver* and *executor* pods.

For these invocations, the `argv:` is nearly identical to non-kubernetes configurations, invoking a `run.sh` script which essentially holds the `spark-submit` invocation that takes the aforementioned `SPARK_OPTS` as its primary parameter:

```
{
  "argv": [
    "/usr/local/share/jupyter/kernels/spark_python_kubernetes/bin/run.sh",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
```

(continues on next page)

(continued from previous page)

```
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}",
    "--RemoteProcessProxy.spark-context-initialization-mode",
    "lazy"
  ]
}
```

11.6 Deploying Enterprise Gateway on Kubernetes

Once the Kubernetes cluster is configured and `kubectl` is demonstrated to be working on the master node, it is time to deploy Enterprise Gateway. There a couple of different deployment options - kubectl or helm.

11.6.1 Option 1: Deploying with kubectl

Choose this deployment option if you want to deploy directly from Kubernetes template files with kubectl, rather than using a package manager like Helm.

Create the Enterprise Gateway kubernetes service and deployment

From the master node, create the service and deployment using the yaml file from a source release or the git repository:

```
kubectl apply -f etc/kubernetes/enterprise-gateway.yaml

service "enterprise-gateway" created
deployment "enterprise-gateway" created
```

Uninstalling Enterprise Gateway

To shutdown Enterprise Gateway issue a delete command using the previously mentioned global label `app=enterprise-gateway`

```
kubectl delete all -l app=enterprise-gateway
```

or simply delete the namespace

```
kubectl delete ns enterprise-gateway
```

A kernel's objects can be similarly deleted using the kernel's namespace…

```
kubectl delete ns <kernel-namespace>
```

Note that this should not imply that kernels be "shutdown" using a the `kernel_id=` label. This will likely trigger Jupyter's auto-restart logic - so its best to properly shutdown kernels prior to kubernetes object deletions.

Also note that deleting the Enterprise Gateway namespace will not delete cluster-scoped resources like the cluster roles `enterprise-gateway-controller` and `kernel-controller` or the cluster role binding `enterprise-gateway-controller`. The following commands can be used to delete these:

```
kubectl delete clusterrole -l app=enterprise-gateway
kubectl delete clusterrolebinding -l app=enterprise-gateway
```

11.6.2 Option 2: Deploying with Helm

Choose this option if you want to deploy via a Helm chart. If Ingress is desired see *this section* before deploying with helm.

Create the Enterprise Gateway kubernetes service and deployment

Ensure the `enterprise-gateway` namespace exists:

```
kubectl create namespace enterprise-gateway
```

From anywhere with Helm cluster access, create the service and deployment by running Helm from a source release or the git repository:

```
helm upgrade --install --atomic --namespace enterprise-gateway enterprise-gateway etc/
↪kubernetes/helm/enterprise-gateway
```

the helm chart tarball is also accessible as an asset on our release page:

```
helm install --name enterprise-gateway --atomic --namespace enterprise-gateway https://
↪github.com/jupyter/enterprise_gateway/releases/download/v2.5.0/jupyter_enterprise_
↪gateway_helm-2.5.0.tgz
```

Configuration

Here are all of the values that you can set when deploying the Helm chart. You can override them with Helm's `--set` or `--values` options.

Uninstalling Enterprise Gateway

When using Helm, you can uninstall Enterprise Gateway with the following command:

```
helm delete --purge enterprise-gateway
```

11.6.3 Confirm deployment and note the service port mapping

```
kubectl get all --all-namespaces -l app=enterprise-gateway

NAME                           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/enterprise-gateway      1         1         1            1           2h

NAME                                  DESIRED   CURRENT   READY     AGE
rs/enterprise-gateway-74c46cb7fc      1         1         1         2h

NAME                                        READY     STATUS    RESTARTS   AGE
```

(continues on next page)

(continued from previous page)

```
po/enterprise-gateway-74c46cb7fc-jrkl7   1/1       Running   0          2h

NAME                         TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
svc/enterprise-gateway       NodePort   10.110.253.220   <none>        8888:32422/TCP   2h
```

Of particular importance is the mapping to port 8888 (e.g.,32422). If you are performing this on the same host as where the notebook will run, then you will need to note the cluster-ip entry (e.g.,10.110.253.220).

(Note: if the number of replicas is > 1, then you will see two pods listed with different five-character suffixes.)

Tip: You can avoid the need to point at a different port each time EG is launched by adding an externalIPs: entry to the spec: section of the enterprise-gateway.yaml file. The file is delivered with this entry commented out. Of course, you'll need to change the IP address to that of your kubernetes master node once the comments characters have been removed.

```
# Uncomment in order to use <k8s-master>:8888
#  externalIPs:
#  - 9.30.118.200
```

However, if using Helm, see the section above about how to set the k8sMasterPublicIP.

The value of the JUPYTER_GATEWAY_URL used by the gateway-enabled Notebook server will vary depending on whether you choose to define an external IP or not. If and external IP is defined, you'll set JUPYTER_GATEWAY_URL=<externalIP>:8888 else you'll set JUPYTER_GATEWAY_URL=<k8s-master>:32422 **but also need to restart clients each time Enterprise Gateway is started.** As a result, use of the externalIPs: value is highly recommended.

11.7 Setting up a Kubernetes Ingress for use with Enterprise Gateway

To setup an ingress with Enterprise Gateway, you'll need an ingress controller deployed on your kubernetes cluster. We recommend either NGINX or Traefik. Installation and configuration instructions can be found at the following :

- NGINX-Ingress-Controller
- Traefik

Example - Here the NGINX Ingress Controller is deployed as a LoadBalancer with NodePort 32121 and 30884 open for http and https traffic respectively.

```
$ kubectl get services --all-namespaces
NAMESPACE            NAME                                            TYPE           ␣
↪CLUSTER-IP        EXTERNAL-IP   PORT(S)                        AGE
default              service/kubernetes                              ClusterIP      10.
↪96.0.1         <none>        443/TCP                        23h
default              service/my-nginx-nginx-ingress-controller       LoadBalancer   10.
↪105.234.155    <pending>     80:32121/TCP,443:30884/TCP     22h
default              service/my-nginx-nginx-ingress-default-backend  ClusterIP      10.
↪107.13.85      <none>        80/TCP                         22h
enterprise-gateway   service/enterprise-gateway                      NodePort       10.
↪97.127.52      <none>        8888:30767/TCP                 27m
kube-system          service/kube-dns                                ClusterIP      10.
↪96.0.10        <none>        53/UDP,53/TCP,9153/TCP         23h
kube-system          service/tiller-deploy                           ClusterIP      10.
↪101.96.215     <none>        44134/TCP                      23h
```

Once you have a Ingress controller installed, you can use the `Ingress` resource in kubernetes to direct traffic to your Enterprise Gateway service. The EG helm chart is configured with an ingress template, which can be found at here for Enterprise Gateway.

Example - Enable ingress and edit etc/kubernetes/helm/values.yaml to the desired configurations and install EG as normal via Helm.

```
ingress:
  enabled: true             # Ingress is disabled by default
  annotations:              # Annotations to be used, changes depend on which ingress␣
↪controller you have deployed # default is nginx
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/force-ssl-redirect: "false"
  hostName: ""              # whether to expose by setting a host-based ingress rule,␣
↪default is *
  path: /gateway/?(.*)      # URL context used to expose EG
```

A quick look at our ingress resource after deploying EG with Helm :

```
$ kubectl describe ingress enterprise-gateway-ingress -n enterprise-gateway
Name:             enterprise-gateway-ingress
Namespace:        enterprise-gateway
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *
        /gateway/?(.*)   enterprise-gateway:8888 (<none>)
Annotations:
  kubectl.kubernetes.io/last-applied-configuration:  {"apiVersion":"extensions/v1beta1",
↪"kind":"Ingress","metadata":
  {"annotations":{"kubernetes.io/ingress.class":"nginx","nginx.ingress.kubernetes.io/
↪force-ssl-redirect":"false",
  "nginx.ingress.kubernetes.io/rewrite-target":"/$1","nginx.ingress.kubernetes.io/ssl-
↪redirect":"false"},
  "name":"enterprise-gateway-ingress","namespace":"enterprise-gateway"},"spec":{"rules":[
↪{"http":{"paths":[{
  "backend":{"serviceName":"enterprise-gateway","servicePort":8888},"path":"/gateway/?(.
↪*)"}]}}]}}

  kubernetes.io/ingress.class:                     nginx
  nginx.ingress.kubernetes.io/force-ssl-redirect:  false
  nginx.ingress.kubernetes.io/rewrite-target:      /$1
  nginx.ingress.kubernetes.io/ssl-redirect:        false
Events:                                            <none>
```

This will expose the Enterprise Gateway service at

```
http://KUBERNETES_HOSTNAME:PORT/gateway
```

where `PORT` is the ingress controller's http `NodePort` we referenced earlier. **NOTE:** `PORT` may be optional depending on how your environment/infrastructure is configured.

11.8 Kubernetes Tips

The following items illustrate some useful commands for navigating Enterprise Gateway within a kubernetes environment.

- All objects created on behalf of Enterprise Gateway can be located using the label `app=enterprise-gateway`. You'll probably see duplicated entries for the deployments(deploy) and replication sets (rs) - I didn't include the duplicates here.

```
kubectl get all -l app=enterprise-gateway --all-namespaces

NAME                        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/enterprise-gateway   1         1         1            1           3h

NAME                               DESIRED   CURRENT   READY     AGE
rs/enterprise-gateway-74c46cb7fc   1         1         1         3h

NAME                                             READY     STATUS    RESTARTS   AGE
po/alice-5e755458-a114-4215-96b7-bcb016fc7b62   1/1       Running   0          8s
po/enterprise-gateway-74c46cb7fc-jrkl7          1/1       Running   0          3h
```

- All objects related to a given kernel can be located using the label `kernel_id=<kernel_id>`

```
kubectl get all -l kernel_id=5e755458-a114-4215-96b7-bcb016fc7b62 --all-namespaces

NAME                                             READY     STATUS    RESTARTS   AGE
po/alice-5e755458-a114-4215-96b7-bcb016fc7b62   1/1       Running   0          28s
```

Note: because kernels are, by default, isolated to their own namespace, you could also find all objects of a given kernel using only the `--namespace <kernel-namespace>` clause.

- To enter into a given pod (i.e., container) in order to get a better idea of what might be happening within the container, use the exec command with the pod name

```
kubectl exec -it enterprise-gateway-74c46cb7fc-jrkl7 /bin/bash
```

- Logs can be accessed against the pods or deployment (requires the object type prefix (e.g., `po/`))

```
kubectl logs -f po/alice-5e755458-a114-4215-96b7-bcb016fc7b62
```

Note that if using multiple replicas, commands against each pod are required.

- The Kubernetes dashboard is useful as well. Its located at port `30000` of the master node

```
https://elyra-kube1.foo.bar.com:30000/dashboard/#!/overview?namespace=default
```

From there, logs can be accessed by selecting the `Pods` option in the left-hand pane followed by the *lined* icon on the far right.

- User "system:serviceaccount:default:default" cannot list pods in the namespace "default"

On a recent deployment, Enterprise Gateway was not able to create or list kernel pods. Found the following command was necessary. (Kubernetes security relative to Enterprise Gateway is still under construction.)

```
kubectl create clusterrolebinding add-on-cluster-admin --clusterrole=cluster-admin  --
↪serviceaccount=default:default
```

CHAPTER

TWELVE

DOCKER SWARM

This page describes the approach taken for integrating Enterprise Gateway into an existing Docker Swarm cluster.

In this solution, Enterprise Gateway is, itself, provisioned as a Docker Swarm *service*. In this way, Enterprise Gateway can leverage load balancing and high availability functionality provided by Swarm (although HA cannot be fully realized until EG supports persistent sessions).

The base Enterprise Gateway image is elyra/enterprise-gateway and can be found in the Enterprise Gateway dockerhub organization elyra, along with other images. See Runtime Images for image details.

The following sample kernelspecs are currently available on Docker:

- R_docker
- python_docker
- python_tf_docker
- python_tf_gpu_docker
- scala_docker

12.1 Enterprise Gateway Deployment

Enterprise Gateway manifests itself as a Docker Swarm service. It is identified by the name `enterprise-gateway` within the cluster. In addition, all objects related to Enterprise Gateway, including kernel instances, have a label of `app=enterprise-gateway` applied.

The current deployment uses a compose stack definition, docker-compose.yml which creates an overlay network intended for use solely by Enterprise Gateway and any kernel-based services it launches.

To deploy the stack to a swarm cluster from a manager node, use:

```
docker stack deploy -c docker-compose.yml enterprise-gateway
```

More information about deploying and managing stacks can be found here.

Since Swarm's support for session-based affinity has not been investigated at this time, the deployment script configures a single replica. Once session affinity is available, the number of replicas can be increased.

An alternative deployment of Enterprise Gateway in docker environments is to deploy Enterprise Gateway as a traditional docker container. This can be accomplished via the docker-compose.yml file. However, keep in mind that in choosing this deployment approach, one loses leveraging swarm's monitoring/restart capabilities. That said, choosing this approach does not preclude one from leveraging swarm's scheduling capabilities for launching kernels. As noted below, kernel instances, and how they manifest as docker-based entities (i.e., a swarm service or a docker container), is purely a function of the process proxy class to which they're associated.

To start the stack using compose:

```
docker-compose up
```

The documentation for managing a compose stack can be found here.

12.1.1 Kernelspec Modifications

One of the more common areas of customization we see occur within the kernelspec files located in /usr/local/share/jupyter/kernels. To accommodate the ability to customize the kernel definitions, the kernels directory can be exposed as a mounted volume thereby making it available to all containers within the swarm cluster.

As an example, we have included the necessary commands to mount these volumes, both in the deployment script and in the launch_docker.py file used to launch docker-based kernels. By default, these references are commented out as they require the system administrator to ensure the directories are available throughout the cluster.

Note that because the kernel launch script, launch_docker.py, resides in the kernelspecs hierarchy, updates or modifications to docker-based kernel instances can now also take place. (We'll be looking at ways to make modifications to per-kernel configurations more manageable.)

12.2 Docker Swarm Kernel Instances

Enterprise Gateway currently supports launching of *vanilla* (i.e., non-spark) kernels within a Docker Swarm cluster. When kernels are launched, Enterprise Gateway is responsible for creating the appropriate entity. The kind of entity created is a function of the corresponding process proxy class.

When the process proxy class is `DockerSwarmProcessProxy` the `launch_docker.py` script will create a Docker Swarm *service*. This service uses a restart policy of `none` meaning that its configured to go away upon failures or completion. In addition, because the kernel is launched as a swarm service, the kernel can "land" on any node of the cluster.

When the process proxy class is `DockerProcessProxy` the `launch_docker.py` script will create a traditional docker *container*. As a result, the kernel will always reside on the same host as the corresponding Enterprise Gateway.

Items worth noting:

1. The Swarm service or Docker container name will be composed of the launching username (`KERNEL_USERNAME`) and kernel-id.
2. The service/container will have 3 labels applied: "kernel_id=", "component=kernel", and "app=enterprise-gateway" - similar to Kubernetes.
3. The service/container will be launched within the same docker network as Enterprise Gateway.

12.3 DockerSwarmProcessProxy

To indicate that a given kernel should be launched as a Docker Swarm service into a swarm cluster, the kernel.json file's `metadata` stanza must include a `process_proxy` stanza indicating a `class_name:` of `DockerSwarmProcessProxy`. This ensures the appropriate lifecycle management will take place relative to a Docker Swarm environment.

Along with the `class_name:` entry, this process proxy stanza should also include a proxy configuration stanza which specifies the docker image to associate with the kernel's service container. If this entry is not provided, the Enterprise Gateway implementation will use a default entry of `elyra/kernel-py:VERSION`. In either case, this value is made available to the rest of the parameters used to launch the kernel by way of an environment variable: `KERNEL_IMAGE`.

(Please note that the use of `VERSION` in docker image tags is a placeholder for the appropriate version-related image tag. When kernelspecs are built via the Enterprise Gateway Makefile, `VERSION` is replaced with the appropriate version denoting the target release. A full list of available image tags can be found in the dockerhub repository corresponding to each image.)

```
{
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.docker_swarm.
↪DockerSwarmProcessProxy",
      "config": {
        "image_name": "elyra/kernel-py:VERSION"
      }
    }
  },
}
```

As always, kernels are launched by virtue of the `argv:` stanza in their respective kernel.json files. However, when launching kernels in a docker environment, what gets invoked isn't the kernel's launcher, but, instead, a python script that is responsible for using the Docker Python API to create the corresponding instance.

```
{
  "argv": [
    "python",
    "/usr/local/share/jupyter/kernels/python_docker/scripts/launch_docker.py",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

12.4 DockerProcessProxy

Running containers in Docker Swarm versus traditional Docker are different enough to warrant having separate process proxy implementations. As a result, the kernel.json file could reference the `DockerProcessProxy` class and, accordingly, a traditional docker container (as opposed to a swarm *service*) will be created. The rest of the kernel.json file, image name, argv stanza, etc. is identical.

```
{
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.docker_swarm.
↪DockerProcessProxy",
      "config": {
        "image_name": "elyra/kernel-py:VERSION"
      }
    }
  },
  "argv": [
```

(continues on next page)

(continued from previous page)

```
    "python",
    "/usr/local/share/jupyter/kernels/python_docker/scripts/launch_docker.py",
     "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

Upon invocation, the invoked process proxy will set a "docker mode" environment variable (`EG_DOCKER_MODE`) to either `swarm` or `docker`, depending on the process proxy instance, that the `launch_docker.py` script uses to determine whether a *service* or *container* should be created, respectively.

It should be noted that each of these forms of process proxy usage does **NOT** need to match to the way in which the Enterprise Gateway instance was deployed. For example, if Enterprise Gateway was deployed using `enterprise-gateway-swarm.sh` and a `DockerProcessProxy` is used, that corresponding kernel will be launched as a traditional docker container and will reside on the same host as wherever the Enterprise Gateway (swarm) service is running. Similarly, if Enterprise Gateway was deployed using `enterprise-gateway-docker.sh` and a `DockerSwarmProcessProxy` is used (and assuming a swarm configuration is present), that corresponding kernel will be launched as a docker swarm service and will reside on whatever host the Docker Swarm scheduler decides is best.

CHAPTER THIRTEEN

IBM SPECTRUM CONDUCTOR

This information will be added shortly. The configuration is similar to that of *YARN Cluster mode* with the `ConductorClusterProcessProxy` used in place of `YARNClusterProcessProxy`.

The following sample kernelspecs are currently available on Conductor:

- spark_R_conductor_cluster
- spark_python_conductor_cluster
- spark_scala_conductor_cluster

CHAPTER

FOURTEEN

STANDALONE REMOTE KERNEL EXECUTION

14.1 a.k.a "Library Mode"

Remote kernels can be executed by using the `RemoteKernelManager` class directly. This enables running kernels using `ProcessProxys` without the Enterprise Gateway Webapp.

This can be useful in niche situations, for example, using nbconvert or nbclient to execute a kernel on a remote cluster.

Sample code using nbclient 0.2.0:

```
import nbformat
from nbclient import NotebookClient
from enterprise_gateway.services.kernels.remotemanager import RemoteKernelManager

with open("my_notebook.ipynb") as fp:
    test_notebook = nbformat.read(fp, as_version=4)

client = NotebookClient(nb=test_notebook, kernel_manager_class=RemoteKernelManager)
client.execute()
```

The above code will execute the notebook on a kernel using the configured `ProcessProxy`.

CHAPTER

FIFTEEN

CONFIGURATION OPTIONS

Jupyter Enterprise Gateway adheres to Jupyter's common configuration approach . You can configure an instance of Enterprise Gateway using:

1. A configuration file (recommended)
2. Command line parameters
3. Environment variables

See *Dynamic Configurables* for additional information.

To generate a template configuration file, run the following:

```
jupyter enterprisegateway --generate-config
```

To see the same configuration options at the command line, run the following:

```
jupyter enterprisegateway --help-all
```

A snapshot of this help appears below for ease of reference on the web.

```
Jupyter Enterprise Gateway

Provisions remote Jupyter kernels and proxies HTTP/Websocket traffic to them.

Options
-------

Arguments that take values are actually convenience aliases to full
Configurables, whose aliases are listed on the help line. For more information
on full configurables, see '--help-all'.

--debug
    set log level to logging.DEBUG (maximize logging output)
--generate-config
    generate default config file
-y
    Answer yes to any questions instead of prompting.
--log-level=<Enum> (Application.log_level)
    Default: 30
    Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')
    Set the log level by value or name.
--config=<Unicode> (JupyterApp.config_file)
```

(continues on next page)

(continued from previous page)

```
    Default: ''
    Full path of a config file.
--ip=<Unicode> (EnterpriseGatewayApp.ip)
    Default: '127.0.0.1'
    IP address on which to listen (EG_IP env var)
--port=<Int> (EnterpriseGatewayApp.port)
    Default: 8888
    Port on which to listen (EG_PORT env var)
--port_retries=<Int> (EnterpriseGatewayApp.port_retries)
    Default: 50
    Number of ports to try if the specified port is not available
    (EG_PORT_RETRIES env var)
--keyfile=<Unicode> (EnterpriseGatewayApp.keyfile)
    Default: None
    The full path to a private key file for usage with SSL/TLS. (EG_KEYFILE env
    var)
--certfile=<Unicode> (EnterpriseGatewayApp.certfile)
    Default: None
    The full path to an SSL/TLS certificate file. (EG_CERTFILE env var)
--client-ca=<Unicode> (EnterpriseGatewayApp.client_ca)
    Default: None
    The full path to a certificate authority certificate for SSL/TLS client
    authentication. (EG_CLIENT_CA env var)

Class parameters
----------------

Parameters are set from command-line arguments of the form:
`--Class.trait=value`. This line is evaluated in Python, so simple expressions
are allowed, e.g.:: `--C.a='range(3)'` For setting C.a=[0,1,2].

EnterpriseGatewayApp(EnterpriseGatewayConfigMixin, JupyterApp) options
----------------------------------------------------------------------
--EnterpriseGatewayApp.allow_credentials=<Unicode>
    Sets the Access-Control-Allow-Credentials header. (EG_ALLOW_CREDENTIALS env
    var)
    Default: ''
--EnterpriseGatewayApp.allow_headers=<Unicode>
    Sets the Access-Control-Allow-Headers header. (EG_ALLOW_HEADERS env var)
    Default: ''
--EnterpriseGatewayApp.allow_methods=<Unicode>
    Sets the Access-Control-Allow-Methods header. (EG_ALLOW_METHODS env var)
    Default: ''
--EnterpriseGatewayApp.allow_origin=<Unicode>
    Sets the Access-Control-Allow-Origin header. (EG_ALLOW_ORIGIN env var)
    Default: ''
--EnterpriseGatewayApp.alt_yarn_endpoint=<Unicode>
    The http url specifying the alternate YARN Resource Manager.  This value
    should be set when YARN Resource Managers are configured for high
    availability.  Note: If both YARN endpoints are NOT set, the YARN library
    will use the files within the local HADOOP_CONFIG_DIR to determine the
    active resource manager. (EG_ALT_YARN_ENDPOINT env var)
```

(continues on next page)

(continued from previous page)

```
    Default: None
--EnterpriseGatewayApp.answer_yes=<Bool>
    Answer yes to any prompts.
    Default: False
--EnterpriseGatewayApp.auth_token=<Unicode>
    Authorization token required for all requests (EG_AUTH_TOKEN env var)
    Default: ''
--EnterpriseGatewayApp.authorized_users=<set-item-1>...
    Comma-separated list of user names (e.g., ['bob','alice']) against which
    KERNEL_USERNAME will be compared.  Any match (case-sensitive) will allow the
    kernel's launch, otherwise an HTTP 403 (Forbidden) error will be raised.
    The set of unauthorized users takes precedence. This option should be used
    carefully as it can dramatically limit who can launch kernels.
    (EG_AUTHORIZED_USERS env var - non-bracketed, just comma-separated)
    Default: set()
--EnterpriseGatewayApp.authorized_origin=<Unicode>
    Hostname (e.g. 'localhost', 'reverse.proxy.net') which the handler will
    match against the request's SSL certificate.  An HTTP 403 (Forbidden) error
    will be raised on a failed match.  This option requires TLS to be enabled.
    It does not support IP addresses. (EG_AUTHORIZED_ORIGIN env var)
    Default: ''
--EnterpriseGatewayApp.base_url=<Unicode>
    The base path for mounting all API resources (EG_BASE_URL env var)
    Default: '/'
--EnterpriseGatewayApp.certfile=<Unicode>
    The full path to an SSL/TLS certificate file. (EG_CERTFILE env var)
    Default: None
--EnterpriseGatewayApp.client_ca=<Unicode>
    The full path to a certificate authority certificate for SSL/TLS client
    authentication. (EG_CLIENT_CA env var)
    Default: None
--EnterpriseGatewayApp.conductor_endpoint=<Unicode>
    The http url for accessing the Conductor REST API. (EG_CONDUCTOR_ENDPOINT
    env var)
    Default: None
--EnterpriseGatewayApp.config_file=<Unicode>
    Full path of a config file.
    Default: ''
--EnterpriseGatewayApp.config_file_name=<Unicode>
    Specify a config file to load.
    Default: ''
--EnterpriseGatewayApp.default_kernel_name=<Unicode>
    Default kernel name when spawning a kernel (EG_DEFAULT_KERNEL_NAME env var)
    Default: ''
--EnterpriseGatewayApp.dynamic_config_interval=<Int>
    Specifies the number of seconds configuration files are polled for changes.
    A value of 0 or less disables dynamic config updates.
    (EG_DYNAMIC_CONFIG_INTERVAL env var)
    Default: 0
--EnterpriseGatewayApp.env_process_whitelist=<list-item-1>...
    Environment variables allowed to be inherited from the spawning process by
    the kernel. (EG_ENV_PROCESS_WHITELIST env var)
```

(continues on next page)

(continued from previous page)

```
    Default: []
--EnterpriseGatewayApp.env_whitelist=<list-item-1>...
    Environment variables allowed to be set when a client requests a new kernel.
    Use '*' to allow all environment variables sent in the request.
    (EG_ENV_WHITELIST env var)
    Default: []
--EnterpriseGatewayApp.expose_headers=<Unicode>
    Sets the Access-Control-Expose-Headers header. (EG_EXPOSE_HEADERS env var)
    Default: ''
--EnterpriseGatewayApp.generate_config=<Bool>
    Generate default config file.
    Default: False
--EnterpriseGatewayApp.impersonation_enabled=<Bool>
    Indicates whether impersonation will be performed during kernel launch.
    (EG_IMPERSONATION_ENABLED env var)
    Default: False
--EnterpriseGatewayApp.ip=<Unicode>
    IP address on which to listen (EG_IP env var)
    Default: '127.0.0.1'
--EnterpriseGatewayApp.kernel_headers=<list-item-1>...
    Request headers to make available to kernel launch framework.
    (EG_KERNEL_HEADERS env var)
    Default: []
--EnterpriseGatewayApp.kernel_manager_class=<Type>
    The kernel manager class to use. Must be a subclass of
    `enterprise_gateway.services.kernels.RemoteMappingKernelManager`.
    Default: 'enterprise_gateway.services.kernels.remotemanager.RemoteMapp...
--EnterpriseGatewayApp.kernel_session_manager_class=<Type>
    The kernel session manager class to use. Must be a subclass of
    `enterprise_gateway.services.sessions.KernelSessionManager`.
    Default: 'enterprise_gateway.services.sessions.kernelsessionmanager.Fi...
--EnterpriseGatewayApp.kernel_spec_cache_class=<Type>
    The kernel spec cache class to use. Must be a subclass of
    `enterprise_gateway.services.kernelspecs.KernelSpecCache`.
    Default: 'enterprise_gateway.services.kernelspecs.kernelspec_cache.Ker...
--EnterpriseGatewayApp.kernel_spec_manager_class=<Type>
    The kernel spec manager class to use. Must be a subclass of
    `jupyter_client.kernelspec.KernelSpecManager`.
    Default: 'jupyter_client.kernelspec.KernelSpecManager'
--EnterpriseGatewayApp.keyfile=<Unicode>
    The full path to a private key file for usage with SSL/TLS. (EG_KEYFILE env
    var)
    Default: None
--EnterpriseGatewayApp.list_kernels=<Bool>
    Permits listing of the running kernels using API endpoints /api/kernels and
    /api/sessions. (EG_LIST_KERNELS env var) Note: Jupyter Notebook allows this
    by default but Jupyter Enterprise Gateway does not.
    Default: False
--EnterpriseGatewayApp.log_datefmt=<Unicode>
    The date format used by logging formatters for %(asctime)s
    Default: '%Y-%m-%d %H:%M:%S'
--EnterpriseGatewayApp.log_format=<Unicode>
```

(continues on next page)

(continued from previous page)

```
    The Logging format template
    Default: '[%(name)s]%(highlevel)s %(message)s'
--EnterpriseGatewayApp.log_level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL']
    Default: 30
--EnterpriseGatewayApp.max_age=<Unicode>
    Sets the Access-Control-Max-Age header. (EG_MAX_AGE env var)
    Default: ''
--EnterpriseGatewayApp.max_kernels=<Int>
    Limits the number of kernel instances allowed to run by this gateway.
    Unbounded by default. (EG_MAX_KERNELS env var)
    Default: None
--EnterpriseGatewayApp.max_kernels_per_user=<Int>
    Specifies the maximum number of kernels a user can have active
    simultaneously.  A value of -1 disables enforcement.
    (EG_MAX_KERNELS_PER_USER env var)
    Default: -1
--EnterpriseGatewayApp.port=<Int>
    Port on which to listen (EG_PORT env var)
    Default: 8888
--EnterpriseGatewayApp.port_range=<Unicode>
    Specifies the lower and upper port numbers from which ports are created. The
    bounded values are separated by '..' (e.g., 33245..34245 specifies a range
    of 1000 ports to be randomly selected). A range of zero (e.g., 33245..33245
    or 0..0) disables port-range enforcement.  (EG_PORT_RANGE env var)
    Default: '0..0'
--EnterpriseGatewayApp.port_retries=<Int>
    Number of ports to try if the specified port is not available
    (EG_PORT_RETRIES env var)
    Default: 50
--EnterpriseGatewayApp.remote_hosts=<list-item-1>...
    Bracketed comma-separated list of hosts on which DistributedProcessProxy
    kernels will be launched e.g., ['host1','host2']. (EG_REMOTE_HOSTS env var -
    non-bracketed, just comma-separated)
    Default: ['localhost']
--EnterpriseGatewayApp.show_config=<Bool>
    Instead of starting the Application, dump configuration to stdout
    Default: False
--EnterpriseGatewayApp.show_config_json=<Bool>
    Instead of starting the Application, dump configuration to stdout (as JSON)
    Default: False
--EnterpriseGatewayApp.ssl_version=<Int>
    Sets the SSL version to use for the web socket connection. (EG_SSL_VERSION
    env var)
    Default: None
--EnterpriseGatewayApp.trust_xheaders=<CBool>
    Use x-* header values for overriding the remote-ip, useful when application
    is behing a proxy. (EG_TRUST_XHEADERS env var)
    Default: False
--EnterpriseGatewayApp.unauthorized_users=<set-item-1>...
    Comma-separated list of user names (e.g., ['root','admin']) against which
```

(continues on next page)

(continued from previous page)

```
    KERNEL_USERNAME will be compared.  Any match (case-sensitive) will prevent
    the kernel's launch and result in an HTTP 403 (Forbidden) error.
    (EG_UNAUTHORIZED_USERS env var - non-bracketed, just comma-separated)
    Default: {'root'}
--EnterpriseGatewayApp.ws_ping_interval=<Int>
    Specifies the ping interval(in seconds) that should be used by zmq port
     associated withspawned kernels.Set this variable to 0 to disable ping mechanism.
    (EG_WS_PING_INTERVAL_SECS env var)
    Default: 30
--EnterpriseGatewayApp.yarn_endpoint=<Unicode>
    The http url specifying the YARN Resource Manager. Note: If this value is
    NOT set, the YARN library will use the files within the local
    HADOOP_CONFIG_DIR to determine the active resource manager.
    (EG_YARN_ENDPOINT env var)
    Default: None
--EnterpriseGatewayApp.yarn_endpoint_security_enabled=<Bool>
    Is YARN Kerberos/SPNEGO Security enabled (True/False).
    (EG_YARN_ENDPOINT_SECURITY_ENABLED env var)
    Default: False

KernelSpecCache(SingletonConfigurable) options
----------------------------------------------
--KernelSpecCache.cache_enabled=<CBool>
    Enable Kernel Specification caching. (EG_KERNELSPEC_CACHE_ENABLED env var)
    Default: False

FileKernelSessionManager(KernelSessionManager) options
------------------------------------------------------
--FileKernelSessionManager.enable_persistence=<Bool>
    Enable kernel session persistence (True or False). Default = False
    (EG_KERNEL_SESSION_PERSISTENCE env var)
    Default: False
--FileKernelSessionManager.persistence_root=<Unicode>
    Identifies the root 'directory' under which the 'kernel_sessions' node will
    reside.  This directory should exist.  (EG_PERSISTENCE_ROOT env var)
    Default: ''

RemoteMappingKernelManager(AsyncMappingKernelManager) options
-------------------------------------------------------------
--RemoteMappingKernelManager.allowed_message_types=<list-item-1>...
    White list of allowed kernel message types. When the list is empty, all
    message types are allowed.
    Default: []
--RemoteMappingKernelManager.buffer_offline_messages=<Bool>
    Whether messages from kernels whose frontends have disconnected should be
    buffered in-memory. When True (default), messages are buffered and replayed
    on reconnect, avoiding lost messages due to interrupted connectivity.
    Disable if long-running kernels will produce too much output while no
    frontends are connected.
    Default: True
--RemoteMappingKernelManager.cull_busy=<Bool>
    Whether to consider culling kernels which are busy. Only effective if
```

(continues on next page)

(continued from previous page)

```
    cull_idle_timeout > 0.
    Default: False
--RemoteMappingKernelManager.cull_connected=<Bool>
    Whether to consider culling kernels which have one or more connections. Only
    effective if cull_idle_timeout > 0.
    Default: False
--RemoteMappingKernelManager.cull_idle_timeout=<Int>
    Timeout (in seconds) after which a kernel is considered idle and ready to be
    culled. Values of 0 or lower disable culling. Very short timeouts may result
    in kernels being culled for users with poor network connections.
    Default: 0
--RemoteMappingKernelManager.cull_interval=<Int>
    The interval (in seconds) on which to check for idle kernels exceeding the
    cull timeout value.
    Default: 300
--RemoteMappingKernelManager.default_kernel_name=<Unicode>
    The name of the default kernel to start
    Default: 'python3'
--RemoteMappingKernelManager.kernel_info_timeout=<Float>
    Timeout for giving up on a kernel (in seconds). On starting and restarting
    kernels, we check whether the kernel is running and responsive by sending
    kernel_info_requests. This sets the timeout in seconds for how long the
    kernel can take before being presumed dead. This affects the
    MappingKernelManager (which handles kernel restarts) and the
    ZMQChannelsHandler (which handles the startup).
    Default: 60
--RemoteMappingKernelManager.kernel_manager_class=<DottedObjectName>
    The kernel manager class.  This is configurable to allow subclassing of the
    AsyncKernelManager for customized behavior.
    Default: 'jupyter_client.ioloop.AsyncIOLoopKernelManager'
--RemoteMappingKernelManager.root_dir=<Unicode>
    Default: ''
--RemoteMappingKernelManager.shared_context=<Bool>
    Share a single zmq.Context to talk to all my kernels
    Default: True
```

15.1 Additional supported environment variables

The following environment variables can be used to influence functionality and are not tied to command-line options:

```
  EG_DEFAULT_KERNEL_SERVICE_ACCOUNT_NAME=default
    Kubernetes only.  This value indicates the default service account name to use for
    kernel namespaces when the Enterprise Gateway needs to create the kernel's namespace
    and KERNEL_SERVICE_ACCOUNT_NAME has not been provided.

  EG_DOCKER_NETWORK=enterprise-gateway or bridge
    Docker only. Used by the docker deployment and launch scripts, this indicates the
    name of the docker network docker network to use.  The start scripts default this
    value to 'enterprise-gateway' because they create the network.  The docker kernel
    launcher (launch_docker.py) defaults this value to 'bridge' only in cases where it
```

(continues on next page)

(continued from previous page)

```
    wasn't previously set by the deployment script.

  EG_ENABLE_TUNNELING=False
    Indicates whether tunneling (via ssh) of the kernel and communication ports
    is enabled (True) or not (False).

  EG_KERNEL_CLUSTER_ROLE=kernel-controller or cluster-admin
    Kubernetes only.  The role to use when binding with the kernel service account.
    The enterprise-gateway.yaml script creates the cluster role 'kernel-controller'
    and conveys that name via EG_KERNEL_CLUSTER_ROLE.  Should the deployment script
    not set this valuem, Enterprise Gateway will then use 'cluster-admin'.  It is
    recommended this value be set to something other than 'cluster-admin'.

  EG_KERNEL_LAUNCH_TIMEOUT=30
    The time (in seconds) Enterprise Gateway will wait for a kernel's startup
    completion status before deeming the startup a failure, at which time a second
    startup attempt will take place.  If a second timeout occurs, Enterprise
    Gateway will report a failure to the client.

  EG_KERNEL_LOG_DIR=/tmp
    The directory used during remote kernel launches of DistributedProcessProxy
    kernels.  Files in this directory will be of the form kernel-<kernel_id>.log.

  EG_KERNEL_SESSION_PERSISTENCE=False
    **Experimental** Enables kernel session persistence.  Currently, this is purely
    experiemental and writes kernel session information to a local file.  Should
    Enterprise Gateway terminate with running kernels, a subsequent restart of
    Enterprise Gateway will attempt to reconnect to the persisted kernels.  See
    also EG_KERNEL_SESSION_LOCATION and --KernelSessionManager.enable_persistence.

  EG_KERNEL_SESSION_LOCATION=<JupyterDataDir>
    **Experimental** The location in which the kernel session information is persisted.
    By default, this is located in the configured JupyterDataDir.  See also
    EG_KERNEL_SESSION_PERSISTENCE.

  EG_MAX_PORT_RANGE_RETRIES=5
    The number of attempts made to locate an available port within the specified
    port range.  Only applies when --EnterpriseGatewayApp.port_range
    (or EG_PORT_RANGE) has been specified or is in use for the given kernel.

  EG_MIN_PORT_RANGE_SIZE=1000
    The minimum port range size permitted when --EnterpriseGatewayApp.port_range
    (or EG_PORT_RANGE) is specified or is in use for the given kernel.  Port ranges
    reflecting smaller sizes will result in a failure to launch the corresponding
    kernel (since port-range can be specified within individual kernel specifications).

  EG_MIRROR_WORKING_DIRS=False
    Containers only.  If True, kernel creation requests that specify KERNEL_WORKING_DIR
    will set the kernel container's working directory to that value.  See also
    KERNEL_WORKING_DIR.

  EG_NAMESPACE=enterprise-gateway or default
```

(continues on next page)

(continued from previous page)

```
  Kubernetes only.  Used during Kubernetes deployment, this indicates the name of
  the namespace in which the Enterprise Gateway service is deployed.  The
  enterprise-gateway.yaml file creates this namespace, then sets EG_NAMESPACE dring
  deployment. This value is then used within Enterprise Gateway to coordinate kernel
  configurations. Should this value not be set during deployment, Enterprise Gateway
  will default its value to namespace 'default'.

EG_PROHIBITED_GIDS=0
  Containers only.  A comma-separated list of group ids (GID) whose values are not
  allowed to be referenced by KERNEL_GID.  This defaults to the root group id (0).
  Attempts to launch a kernel where KERNEL_GID's value is in this list will result
  in an exception indicating error 403 (Forbidden).  See also EG_PROHIBITED_UIDS.

EG_PROHIBITED_LOCAL_IPS=''
  A comma-separated list of local IPv4 addresses (or regular expressions) that
  should not be used when determining the response address used to convey connection
  information back to Enterprise Gateway from a remote kernel.  In some cases, other
  network interfaces (e.g., docker with 172.17.0.*) can interfere - leading to
  connection failures during kernel startup.
  Example: EG_PROHIBITED_LOCAL_IPS=172.17.0.*,192.168.0.27 will eliminate the use of
  all addresses in 172.17.0 as well as 192.168.0.27

EG_PROHIBITED_UIDS=0
  Containers only.  A comma-separated list of user ids (UID) whose values are not
  allowed to be referenced by KERNEL_UID.  This defaults to the root user id (0).
  Attempts to launch a kernel where KERNEL_UID's value is in this list will result
  in an exception indicating error 403 (Forbidden).  See also EG_PROHIBITED_GIDS.

EG_RESPONSE_IP=None
  Experimental.  The IP address to use to formulate the response address (with
  `EG_RESPONSE_PORT`).  By default, the server's IP is used.  However, we may find
  it necessary to use a different IP in cases where the target kernels are external
  to the Enterprise Gateway server (for example).  It's value may also need to be
  set in cases where the computed (default) is not correct for the current topology.

EG_RESPONSE_PORT=8877
  The single response port used to receive connection information
  from launched kernels.

EG_RESPONSE_PORT_RETRIES=10
  The number of retries to attempt when the original response port
  (EG_RESPONSE_PORT) is found to be in-use.  This value should be
  set to 0 (zero) if no port retries are desired.

EG_SHARED_NAMESPACE=False
  Kubernetes only. This value indicates whether (True) or not (False) all kernel pods
  should reside in the same namespace as Enterprise Gateway.  This is not a recommended
  configuration.

EG_SSH_PORT=22
  The port number used for ssh operations for installations choosing to
  configure the ssh server on a port other than the default 22.
```

(continues on next page)

(continued from previous page)

```
EG_REMOTE_PWD=None
  The password to use to ssh to remote hosts

EG_REMOTE_USER=None
  The username to use when connecting to remote hosts (default to `getpass.getuser()`
  when not set).

EG_REMOTE_GSS_SSH=False
  Use gss instead of EG_REMOTE_USER and EG_REMOTE_PWD to connect to remote host via␣
↪SSH.
  Case insensitive. 'True' to enable, 'False', '' or unset to disable.
  Any other value will error.

EG_YARN_CERT_BUNDLE=<custom_truststore_path>
  The path to a .pem or any other custom truststore used as a CA bundle in yarn-api-
↪client.
```

15.2 Environment variables that assist in troubleshooting

The following environment variables may be useful for troubleshooting:

```
EG_DOCKER_LOG_LEVEL=WARNING
  By default, the docker client library is too verbose for its logging.  This
  value can be adjusted in situations where docker troubleshooting may be warranted.

EG_KUBERNETES_LOG_LEVEL=WARNING
  By default, the kubernetes client library is too verbose for its logging.  This
  value can be adjusted in situations where kubernetes troubleshooting may be
  warranted.

EG_LOG_LEVEL=10
  Used by remote launchers and gateway listeners (where the kernel runs), this
  indicates the level of logging used by those entities.  Level 10 (DEBUG) is
  recommended since they don't do verbose logging.

EG_MAX_POLL_ATTEMPTS=10
  Polling is used in various places during life-cycle management operations - like
  determining if a kernel process is still alive, stopping the process, waiting
  for the process to terminate, etc.  As a result, it may be useful to adjust
  this value during those kinds of troubleshooting scenarios, although that
  should rarely be necessary.

EG_POLL_INTERVAL=0.5
  The interval (in seconds) to wait before checking poll results again.

EG_REMOVE_CONTAINER=True
  Used by launch_docker.py, indicates whether the kernel's docker container should be
  removed following its shutdown.  Set this value to 'False' if you want the container
  to be left around in order to troubleshoot issues.  Remember to set back to 'True'
```

(continues on next page)

(continued from previous page)

```
    to restore normal operation.

  EG_SOCKET_TIMEOUT=5.0
    The time (in seconds) the enterprise gateway will wait on its connection
    file socket waiting on return from a remote kernel launcher.  Upon timeout, the
    operation will be retried immediately, until the overall time limit has been
    exceeded.

  EG_SSH_LOG_LEVEL=WARNING
    By default, the paramiko ssh library is too verbose for its logging.  This
    value can be adjusted in situations where ssh troubleshooting may be warranted.

  EG_YARN_LOG_LEVEL=WARNING
    By default, the yarn-api-client library is too verbose for its logging.  This
    value can be adjusted in situations where YARN troubleshooting may be warranted.
```

15.3 System-owned environment variables

The following environment variables are managed by Enterprise Gateway and listed here for completeness. WARNING: Manually setting these variables could adversely affect operations.

```
  EG_DOCKER_MODE
    Docker only.  Used by launch_docker.py to determine if the kernel container
    should be created using the swarm service API or the regular docker container
    API.  Enterprise Gateway sets this value depending on whether the kernel is
    using the DockerSwarmProcessProxy or DockerProcessProxy.

  EG_RESPONSE_ADDRESS
    This value is set during each kernel launch and resides in the environment of
    the kernel launch process. Its value represents the address to which the remote
    kernel's connection information should be sent.  Enterprise Gateway is listening
    on that socket and will associate that connnection information with the responding
    kernel.
```

15.4 Per-kernel Configuration Overrides

As mentioned in the overview of Process Proxy Configuration capabilities, it's possible to override or amend specific system-level configuration values on a per-kernel basis. The following enumerates the set of per-kernel configuration overrides:

- `remote_hosts`: This process proxy configuration entry can be used to override `--EnterpriseGatewayApp.remote_hosts`. Any values specified in the config dictionary override the globally defined values. These apply to all `DistributedProcessProxy` kernels.
- `yarn_endpoint`: This process proxy configuration entry can be used to override `--EnterpriseGatewayApp.yarn_endpoint`. Any values specified in the config dictionary override the globally defined values. These apply to all `YarnClusterProcessProxy` kernels. Note that you'll likely be required to specify a different `HADOOP_CONF_DIR` setting in the kernel.json's `env` stanza in order of the `spark-submit` command to target the appropriate YARN cluster.

- `authorized_users`: This process proxy configuration entry can be used to override `--EnterpriseGatewayApp.authorized_users`. Any values specified in the config dictionary override the globally defined values. These values apply to **all** process-proxy kernels, including the default `LocalProcessProxy`. Note that the typical use-case for this value is to not set `--EnterpriseGatewayApp.authorized_users` at the global level, but then restrict access at the kernel level.
- `unauthorized_users`: This process proxy configuration entry can be used to ***amend*** `--EnterpriseGatewayApp.unauthorized_users`. Any values specified in the config dictionary are **added** to the globally defined values. As a result, once a user is denied access at the global level, they will *always be denied access at the kernel level*. These values apply to **all** process-proxy kernels, including the default `LocalProcessProxy`.
- `port_range`: This process proxy configuration entry can be used to override `--EnterpriseGatewayApp.port_range`. Any values specified in the config dictionary override the globally defined values. These apply to all `RemoteProcessProxy` kernels.

15.5 Per-kernel Environment Overrides

In some cases, it is useful to allow specific values that exist in a kernel.json `env` stanza to be overridden on a per-kernel basis. For example, if the kernelspec supports resource limitations you may want to allow some requests to have access to more memory or GPUs than another. Enterprise Gateway enables this capability by honoring environment variables provided in the json request over those same-named variables in the kernel.json `env` stanza.

Environment variables for which this can occur are any variables prefixed with `KERNEL_` (with the exception of the internal `KERNEL_GATEWAY` variable) as well as any variables listed in the `--JupyterWebsocketPersonality.env_whitelist` command-line option (or via the `KG_ENV_WHITELIST` variable). Locally defined variables listed in `KG_PROCESS_ENV_WHITELIST` are also honored.

The following kernel-specific environment variables are used by Enterprise Gateway. As mentioned above, all `KERNEL_` variables submitted in the kernel startup request's json body will be available to the kernel for its launch.

```
KERNEL_GID=<from user> or 100
  Containers only. This value represents the group id in which the container will run.
  The default value is 100 representing the users group - which is how all kernel␣
↪images
  produced by Enterprise Gateway are built.  See also KERNEL_UID.
  Kubernetes: Warning - If KERNEL_GID is set it is strongly recommened that feature-
↪gate
  RunAsGroup be enabled, otherwise, this value will be ignored and the pod will run as
  the root group id.  As a result, the setting of this value into the Security Context
  of the kernel pod is commented out in the kernel-pod.yaml file and must be enabled
  by the administrator.
  Docker: Warning - This value is only added to the supplemental group ids.  As a␣
↪result,
  if used with KERNEL_UID, the resulting container will run as the root group with this
  value listed in its supplemental groups.

KERNEL_EXECUTOR_IMAGE=<from kernel.json process-proxy stanza> or KERNEL_IMAGE
  Kubernetees Spark only. This indicates the image that Spark on Kubernetes will use
  for the its executors.  Although this value could come from the user, its strongly
  recommended that the process-proxy stanza of the corresponding kernel's kernelspec
  (kernel.json) file be updated to include the image name.  If no image name is
  provided, the value of KERNEL_IMAGE will be used.
```

(continues on next page)

(continued from previous page)

```
KERNEL_EXTRA_SPARK_OPTS=<from user>
  Spark only. This variable allows users to add additional spark options to the
  current set of options specified in the corresponding kernel.json file.  This
  variable is purely optional with no default value.  In addition, it is the
  responsibility of the user setting this value to ensure the options passed
  are appropriate relative to the target environment.  Because this variable contains
  space-separate values, it requires appropriate quotation.  For example, to use with
  the notebook docker image jupyterhub/k8s-singleuser-sample , the environment␣
↪variable would look something like
  this:

  docker run ... -e KERNEL_EXTRA_SPARK_OPTS=\"--conf spark.driver.memory=2g
  --conf spark.executor.memory=2g\" ... jupyterhub/k8s-singleuser-sample

KERNEL_ID=<from user> or <system generated>
  This value represents the identifier used by the Jupyter framework to identify
  the kernel.  Although this value could be provided by the user, it is recommended
  that it be generated by the system.

KERNEL_IMAGE=<from user> or <from kernel.json process-proxy stanza>
  Containers only. This indicates the image to use for the kernel in containerized
  environments - Kubernetes or Docker.  Although it can be provided by the user, it
  is strongly recommended that the process-proxy stanza of the corresponding kernel's
  kernelspec (kernel.json) file be updated to include the image name.

KERNEL_LAUNCH_TIMEOUT=<from user> or EG_KERNEL_LAUNCH_TIMEOUT
  Indicates the time (in seconds) to allow for a kernel's launch.  This value should
  be submitted in the kernel startup if that particular kernel's startup time is
  expected to exceed that of the EG_KERNEL_LAUNCH_TIMEOUT set when Enterprise
  Gateway starts.

KERNEL_NAMESPACE=<from user> or KERNEL_POD_NAME or EG_NAMESPACE
  Kubernetes only.  This indicates the name of the namespace to use or create on
  Kubernetes in which the kernel pod will be located.  For users wishing to use a
  pre-created namespace, this value should be submitted in the kernel startup
  request.  In such cases, the user must also provide KERNEL_SERVICE_ACCOUNT_NAME.
  If not provided, Enterprise Gateway will create a new namespace for the kernel
  whose value is derived from KERNEL_POD_NAME.  In rare cases where
  EG_SHARED_NAMESPACE is True, this value will be set to the value of EG_NAMESPACE.

  Note that if the namespace is created by Enterprise Gateway, it will be removed
  upon the kernel's termination.  Otherwise, the Enterprise Gateway will not
  remove the namespace.

KERNEL_POD_NAME=<from user> or KERNEL_USERNAME-KERNEL_ID
  Kubernetes only. By default, Enterprise Gateway will use a kernel pod name whose
  value is derived from KERNEL_USERNAME and KERNEL_ID separated by a hyphen
  ('-').  This variable is typically NOT provided by the user, but, in such
  cases, Enterprise Gateway will honor that value.  However, when provided,
  it is the user's responsibility that KERNEL_POD_NAME is unique relative to
  any pods in the target namespace.  In addition, the pod must NOT exist -
  unlike the case if KERNEL_NAMESPACE is provided.
```

(continues on next page)

(continued from previous page)

```
KERNEL_SERVICE_ACCOUNT_NAME=<from user> or EG_DEFAULT_KERNEL_SERVICE_ACCOUNT_NAME
  Kubernetes only.  This value represents the name of the service account that
  Enterprise Gateway should equate with the kernel pod.  If Enterprise Gateway
  creates the kernel's namespace, it will be associated with the cluster role
  identified by EG_KERNEL_CLUSTER_ROLE.  If not provided, it will be derived
  from EG_DEFAULT_KERNEL_SERVICE_ACCOUNT_NAME.

KERNEL_UID=<from user> or 1000
  Containers only. This value represents the user id in which the container will run.
  The default value is 1000 representing the jovyan user - which is how all kernel␣
↪images
  produced by Enterprise Gateway are built.  See also KERNEL_GID.
  Kubernetes: Warning - If KERNEL_UID is set it is strongly recommened that feature-
↪gate
  RunAsGroup be enabled and KERNEL_GID also be set, otherwise, the pod will run as
  the root group id. As a result, the setting of this value into the Security Context
  of the kernel pod is commented out in the kernel-pod.yaml file and must be enabled
  by the administrator.

KERNEL_USERNAME=<from user> or <enterprise-gateway-user>
  This value represents the logical name of the user submitted the request to
  start the kernel. Of all the KERNEL_ variables, KERNEL_USERNAME is the one that
  should be submitted in the request. In environments in which impersonation is
  used it represents the target of the impersonation.

KERNEL_WORKING_DIR=<from user> or None
  Containers only.  This value should model the directory in which the active
  notebook file is running.   It is intended to be used in conjunction with␣
↪appropriate volume
  mounts in the kernel container such that the user's notebook filesystem exists
  in the container and enables the sharing of resources used within the notebook.
  As a result, the primary use case for this is for Jupyter Hub users running in
  Kubernetes.  When a value is provided and EG_MIRROR_WORKING_DIRS=True, Enterprise
  Gateway will set the container's working directory to the value specified in
  KERNEL_WORKING_DIR.  If EG_MIRROR_WORKING_DIRS is False, KERNEL_WORKING_DIR will
  not be available for use during the kernel's launch.  See also EG_MIRROR_WORKING_
↪DIRS.
```

The following kernel-specific environment variables are managed within Enterprise Gateway, but there's nothing preventing them from being set by the client. As a result, caution should be used if setting these variables manually.

```
KERNEL_LANGUAGE=<from language entry of kernel.json>
  This indicates the language of the kernel.  It comes from the language entry
  of the corresponding kernel.json file.  This value is used within the start
  script of the kernel containers, in conjunction with KERNEL_LAUNCHERS_DIR, in
  order to determine which launcher and kernel to start when the container is
  started.

KERNEL_LAUNCHERS_DIR=/usr/local/bin
  Containers only.  This value is used within the start script of the kernel
  containers, in conjunction with KERNEL_LANGUAGE, to determine where the
```

(continues on next page)

(continued from previous page)

```
  appropriate kernel launcher is located.

KERNEL_SPARK_CONTEXT_INIT_MODE=<from argv stanza of kernel.json> or none
  Spark containers only.  This variables exists to convey to the kernel container's
  launch script the mode of Spark context intiatilization it should apply when
  starting the spark-based kernel container.
```

15.6 Dynamic Configurables

Enterprise Gateway now supports the ability to update configuration variables without having to restart Enterprise Gateway. This enables the ability to do things like enable debug logging or adjust the maximum number of kernels per user, all without having to restart Enterprise Gateway.

To enable dynamic configurables configure `EnterpriseGatewayApp.dynamic_config_interval` to a positive value (default is 0 or disabled). Since this is the number of seconds to poll Enterprise Gateway's configuration files, a value greater than 60 (1 minute) is recommended. This functionality works for most configuration values, but does have the following caveats:

1. Any configuration variables set on the command line (CLI) or via environment variables are NOT eligible for dynamic updates. This is because Jupyter gives those values priority over file-based configuration variables.
2. Any configuration variables tied to background processing may not reflect their update if the variable is not *observed* for changes. For example, the code behind `MappingKernelManager.cull_idle_timeout` may not reflect changes to the timeout period if that variable is not monitored (i.e., observed) for changes.
3. Only `Configurables` registered by Enterprise Gateway are eligible for dynamic updates. Currently, that list consists of the following (and their subclasses): EnterpriseGatewayApp, MappingKernelManager, KernelSpecManager, and KernelSessionManager.

As a result, administrators are encouraged to configure Enterprise Gateway via configuration files with only static values configured via the command line or environment.

Note that if `EnterpriseGatewayApp.dynamic_config_interval` is configured with a positive value via the configuration file (i.e., is eligible for updates) and is subsequently set to 0, then dynamic configuration updates will be disabled until Enterprise Gateway is restarted with a positive value. Therefore, we recommend `EnterpriseGatewayApp.dynamic_config_interval` be configured via the command line or environment.

CHAPTER

SIXTEEN

TROUBLESHOOTING

This page identifies scenarios we've encountered when running Enterprise Gateway. We also provide instructions for setting up a debug environment on our *Debugging Jupyter Enterprise Gateway* page.

- **None of the scenarios on this page match or resolve my issue, what do I do next?**

  If you are unable to resolve your issue, take a look at our open issues list to see if there is an applicable scenario already reported. If found, please add a comment to the issue so that we can get a sense of urgency (although all issues are important to us). If not found, please provide the following information if possible.

  1. Describe the issue in as much detail as possible. This should include configuration information about your environment.
  2. Gather and *attach* the following files to the issue. If possible, archiving the files first and attaching the archive is preferred.
     1. The **complete** Enterprise Gateway log file. If possible, please enable `DEBUG` logging that encompasses the issue. You can refer to this section of our Getting Started page for redirection and `DEBUG` enablement.
     2. The log file(s) produced from the corresponding kernel. This is primarily a function of the underlying resource manager.
        - For containerized installations like Kubernetes or Docker Swarm, kernel log output can be captured by running the appropriate `logs` command against the pod or container, respectively. The names of the corresponding pod/container can be found in the Enterprise Gateway log.
        - For `YARN` environments, you'll need to navigate to the appropriate log directory relative the application ID associated with the kernel. The application ID can be located in the Enterprise Gateway log. If you have access to an administrative console, you can usually navigate to the application logs much more easily.
     3. Although unlikely, the notebook log may also be helpful. If we find that the issue is more client-side related, we may ask for `DEBUG` logging here as well.
  3. If you have altered or created new kernelspecs files, the files corresponding to the failing kernels would be helpful. These files could also be added to the attached archive or attached separately.

  Please know that we understand that some information cannot be provided due to its senstivity. In such cases, just let us know and we'll be happy to approach resolution of your issue from a different angle.

- **I just installed Enterprise Gateway but nothing happens, how do I proceed?**

  Because Enterprise Gateway is one element of a networked application, there are various *touch points* that should be validated independently. The following items can be used as a checklist to confirm general operability.

  1. Confirm that Enterprise Gateway is servicing general requests. This can be accomplished using the following `curl` command, which should produce the json corresponding to the configured kernelspecs:

```
curl http://<gateway_server>:<gateway_port>/api/kernelspecs
```

2. Independently validate any resource manager you're running against. Various resource managers usually provide examples for how to go about validating their configuration.

3. Confirm that the Enterprise Gateway arguments for contacting the configured resource manager are in place. These should be covered in our Getting Started topics.

4. If using a Notebook server as your front-end, ensure that the Gateway configuration options or NB2KG extension settings are properly configured. Once the notebook has started, a refresh on the tree view should issue the same `kernelspecs` request in step 1 and the drop-down menu items for available kernels should reflect an entry for each kernelspec returned.

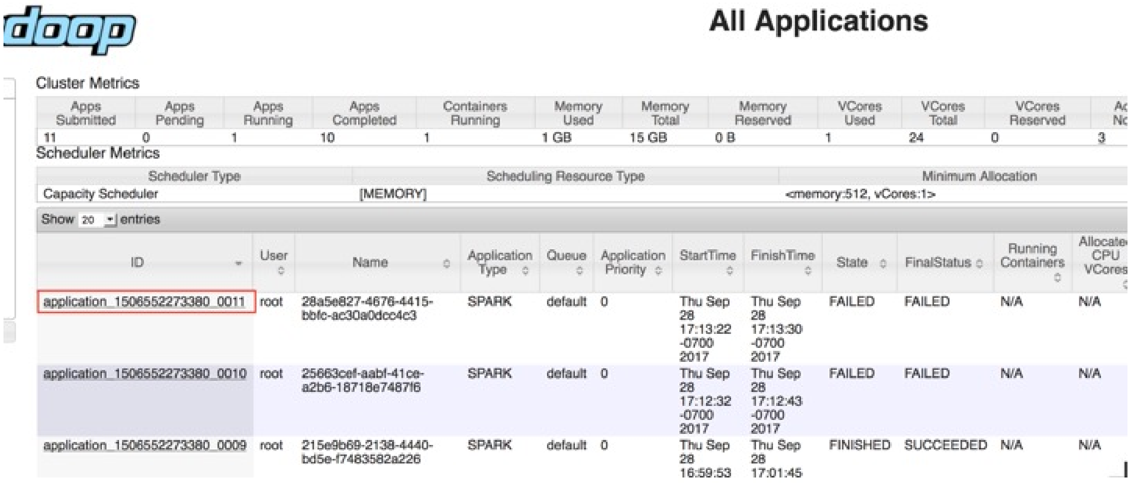
5. **Always** consult your Enterprise Gateway log file. If you have not redirected `stdout` and `stderr` to a file you are highly encouraged to do so. In addition, you should enable `DEBUG` logging at least until your configuration is stable. Please note, however, that you may be asked to produce an Enterprise Gateway log with `DEBUG` enabled when reporting issues. An example of output redirection and `DEBUG` logging is also provided on our Getting Started page.

- **I'm trying to launch a (Python/Scala/R) kernel in YARN Cluster Mode but it failed with a "Kernel error" and State: 'FAILED'.**

  1. Check the output from Enterprise Gateway for an error message. If an applicationId was generated, make a note of it. For example, you can locate the applicationId `application_1506552273380_0011` from the following snippet of message:

```
[D 2017-09-28 17:13:22.675 EnterpriseGatewayApp] 13: State: 'ACCEPTED', Host:
↪'burna2.yourcompany.com', KernelID: '28a5e827-4676-4415-bbfc-ac30a0dcc4c3',␣
↪ApplicationID: 'application_1506552273380_0011'
17/09/28 17:13:22 INFO YarnClientImpl: Submitted application application_
↪1506552273380_0011
17/09/28 17:13:22 INFO Client: Application report for application_1506552273380_
↪0011 (state: ACCEPTED)
17/09/28 17:13:22 INFO Client:
    client token: N/A
    diagnostics: AM container is launched, waiting for AM container to Register␣
↪with RM
    ApplicationMaster host: N/A
    ApplicationMaster RPC port: -1
    queue: default
    start time: 1506644002471
    final status: UNDEFINED
    tracking URL: http://burna1.yourcompany.com:8088/proxy/application_
↪1506552273380_0011/
```

  2. Lookup the YARN log for that applicationId in the YARN ResourceManager UI:



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Ac Nc
11	0	1	10	1	1 GB	15 GB	0 B	1	24	0	3

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation
Capacity Scheduler	[MEMORY]	<memory:512, vCores:1>

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocate CPU VCores
application_1506552273380_0011	root	28a5e827-4676-4415-bbfc-ac30a0dcc4c3	SPARK	default	0	Thu Sep 28 17:13:22 -0700 2017	Thu Sep 28 17:13:30 -0700 2017	FAILED	FAILED	N/A	N/A
application_1506552273380_0010	root	25663cef-aabf-41ce-a2b6-18718e7487f6	SPARK	default	0	Thu Sep 28 17:12:32 -0700 2017	Thu Sep 28 17:12:43 -0700 2017	FAILED	FAILED	N/A	N/A
application_1506552273380_0009	root	215e9b69-2138-4440-bd5e-f7483582a226	SPARK	default	0	Thu Sep 28 16:59:53	Thu Sep 28 17:01:45	FINISHED	SUCCEEDED	N/A	N/A

3. Drill down from the applicationId to find logs for the failed attempts and take appropriate actions. For example, for the error below,

```
Traceback (most recent call last):
 File "launch_ipykernel.py", line 7, in <module>
   from ipython_genutils.py3compat import str_to_bytes
 ImportError: No module named ipython_genutils.py3compat
```

   Simply running "pip install ipython_genutils" should fix the problem. If Anaconda is installed, make sure the environment variable for Python, i.e. PYSPARK_PYTHON, is properly configured in the kernelspec and matches the actual Anaconda installation directory.

- **I'm trying to launch a (Python/Scala/R) kernel in YARN Client Mode but it failed with a "Kernel error" and an AuthenticationException.**

```
[E 2017-09-29 11:13:23.277 EnterpriseGatewayApp] Exception 'AuthenticationException
↪' occurred
when creating a SSHClient connecting to 'xxx.xxx.xxx.xxx' with user 'elyra',
message='Authentication failed.'.
```

  This error indicates that the password-less ssh may not be properly configured. Password-less ssh needs to be configured on the node that the Enterprise Gateway is running on to all other worker nodes.

  You might also see an SSHException indicating a similar issue.

```
[E 2017-09-29 11:13:23.277 EnterpriseGatewayApp] Exception 'SSHException' occurred
when creating a SSHClient connecting to 'xxx.xxx.xxx.xxx' with user 'elyra',
message='No authentication methods available.'.
```

  In general, you can look for more information in the kernel log for YARN Client kernels. The default location is /tmp with a filename of kernel-<kernel_id>.log. The location can be configured using the environment variable EG_KERNEL_LOG_DIR during Enterprise Gateway start up.

  See Starting Enterprise Gateway for an example of starting the Enterprise Gateway from a script and Supported Environment Variables for a list of configurable environment variables.

- **I'm trying to launch a (Python/Scala/R) kernel in YARN Client Mode with SSH tunneling enabled but it failed with a "Kernel error" and a SSHException.**

  ```
  [E 2017-10-26 11:48:20.922 EnterpriseGatewayApp] The following exception occurred␣
  ↪waiting
  for connection file response for KernelId 'da3d0dde-9de1-44b1-b1b4-e6f3cf52dfb9' on␣
  ↪host
  'remote-host-name': The authenticity of the host can't be established.
  ```

  This error indicates that fingerprint for the ECDSA key of the remote host has not been added to the list of known hosts from where the SSH tunnel is being established.

  For example, if the Enterprise Gateway is running on `node1` under service-user `jdoe` and environment variable `EG_REMOTE_HOSTS` is set to `node2,node3,node4`, then the Kernels can be launched on any of those hosts and a SSH tunnel will be established between `node1` and any of the those hosts.

  To address this issue, you need to perform a one-time step that requires you to login to `node1` as `jdoe` and manually SSH into each of the remote hosts and accept the fingerprint of the ECDSA key of the remote host to be added to the list of known hosts as shown below:

  ```
  [jdoe@node1 ~]$ ssh node2
  The authenticity of host 'node2 (172.16.207.191)' can't be established.
  ECDSA key fingerprint is SHA256:Mqi3txf4YiRC9nXg8a/4gQq5vC4SjWmcN1V5Z0+nhZg.
  ECDSA key fingerprint is MD5:bc:4b:b2:39:07:98:c1:0b:b4:c3:24:38:92:7a:2d:ef.
  Are you sure you want to continue connecting (yes/no)? yes
  Warning: Permanently added 'node2,172.16.207.191' (ECDSA) to the list of known␣
  ↪hosts.
  [jdoe@node2 ~] exit
  ```

  Repeat the aforementioned step as `jdoe` on `node1` for each of the hosts listed in `EG_REMOTE_HOSTS` and restart Enterprise Gateway.

- **I'm trying to launch a (Python/Scala/R) kernel but it failed with `TypeError:  Incorrect padding`.**

  ```
  Traceback (most recent call last):
    File "/opt/conda/lib/python3.7/site-packages/tornado/web.py", line 1512, in _
  ↪execute
      result = yield result
    File "/opt/conda/lib/python3.7/site-packages/tornado/gen.py", line 1055, in run
      value = future.result()
    ....
    ....
    ....
    File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/kernels/
  ↪remotemanager.py", line 125, in _launch_kernel
      return self.process_proxy.launch_process(kernel_cmd, **kw)
    File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
  ↪processproxies/yarn.py", line 63, in launch_process
      self.confirm_remote_startup(kernel_cmd, **kw)
    File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
  ↪processproxies/yarn.py", line 174, in confirm_remote_startup
      ready_to_connect = self.receive_connection_info()
    File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
  ↪processproxies/processproxy.py", line 565, in receive_connection_info
      raise e
  TypeError: Incorrect padding
  ```

To address this issue, first ensure that the launchers used for each kernel are derived from the same release as the Enterprise Gateway server. Next ensure that `pycrypto 2.6.1` or later is installed on all hosts using either `pip install` or `conda install` as shown below:

```
[jdoe@node1 ~]$ pip uninstall pycrypto
[jdoe@node1 ~]$ pip install pycrypto
```

or

```
[jdoe@node1 ~]$ conda install pycrypto
```

This should be done on the host running Enterprise Gateway as well as all the remote hosts on which the kernel is launched.

- **I'm trying to launch a (Python/Scala/R) kernel with port range but it failed with `RuntimeError: Invalid port range .`**

```
Traceback (most recent call last):
  File "/opt/conda/lib/python3.7/site-packages/tornado/web.py", line 1511, in _
↪execute
    result = yield result
  File "/opt/conda/lib/python3.7/site-packages/tornado/gen.py", line 1055, in run
    value = future.result()
  ....
  ....
  File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
↪processproxies/processproxy.py", line 478, in __init__
    super(RemoteProcessProxy, self).__init__(kernel_manager, proxy_config)
  File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
↪processproxies/processproxy.py", line 87, in __init__
    self._validate_port_range(proxy_config)
  File "/opt/conda/lib/python3.7/site-packages/enterprise_gateway/services/
↪processproxies/processproxy.py", line 407, in _validate_port_range
    "port numbers is (1024, 65535).".format(self.lower_port))
RuntimeError: Invalid port range '1000..2000' specified. Range for valid port␣
↪numbers is (1024, 65535).
```

To address this issue, make sure that the specified port range does not overlap with TCP's well-known port range of (0, 1024].

- **I'm trying to launch a (Python/Scala/R) kernel but it times out and the YARN application status remain `ACCEPTED`.**

Enterprise Gateway log from server will look like the one below, and will complain that there are no resources: `launch timeout due to: YARN resources unavailable`

```
State: 'ACCEPTED', Host: '', KernelID: '3181db50-8bb5-4f91-8556-988895f63efa',␣
↪ApplicationID: 'application_1537119233094_0001'
State: 'ACCEPTED', Host: '', KernelID: '3181db50-8bb5-4f91-8556-988895f63efa',␣
↪ApplicationID: 'application_1537119233094_0001'
...
...
SIGKILL signal sent to pid: 19690
YarnClusterProcessProxy.kill, application ID: application_1537119233094_0001,␣
↪kernel ID: 3181db50-8bb5-4f91-8556-988895f63efa, state: ACCEPTED
KernelID: '3181db50-8bb5-4f91-8556-988895f63efa' launch timeout due to: YARN␣
↪resources unavailable after 61.0 seconds for app application_1537119233094_0001,␣
↪launch timeout: 60.0!  Check YARN configuration.
```

(continues on next page)

(continued from previous page)

The most common cause for this is that YARN Resource Managers are failing to start and the cluster see no resources available. Make sure YARN Resource Managerss are running ok. We have also noticed that, in Kerberized environment, sometimes there are issues with directory access right that cause the YARN Resource Managers to fail to start and this can be corrected by validating the existence of `/hadoop/yarn` and that it's owned by `yarn: hadoop`.

- **The Kernel keeps dying when processing jobs that require large amount of resources (e.g. large files)**

  This is usually seen when you are trying to use more resources then what is available for your kernel. To address this issue, increase the amount of memory available for your YARN application or another Resource Manager managing the kernel.

- **PySpark 2.4.x fails on Python 3.8**

  PySpark 2.4.x fails on Python 3.8 as described in SPARK-29536. Use Python 3.7.x as the issue only seems to have been resolved on Spark 3.0.

- **I'm trying to use a notebook with user impersonation on a Kerberos enabled cluster but it fails to authenticate.**

  When using user impersonation in a YARN cluster with Kerberos authentication, if Kerberos is not setup properly you will usually see the following warning that will keep a notebook from connecting:

  ```
  WARN Client: Exception encountered while connecting to the server : javax.security.sasl.SaslException: GSS initiate failed
  [Caused by GSSException: No valid credentials provided (Mechanism level: Failed to find any Kerberos tgt)]
  ```

  The most common cause for this WARN is when the user that started Enterprise Gateway is not authenticated with Kerberos. This can happen when the user has either not run `kinit` or their previous ticket has expired.

- **Running Jupyter Enterprise Gateway on OpenShift Kubernetes Environment fails trying to create /home/jovyan/.local**

  As described in the OpenShift Admin Guide there is a need to issue the following command to enable running with `USER` in Dockerfile.

  ```
  oc adm policy add-scc-to-group anyuid system:authenticated
  ```

CHAPTER

SEVENTEEN

DEBUGGING JUPYTER ENTERPRISE GATEWAY

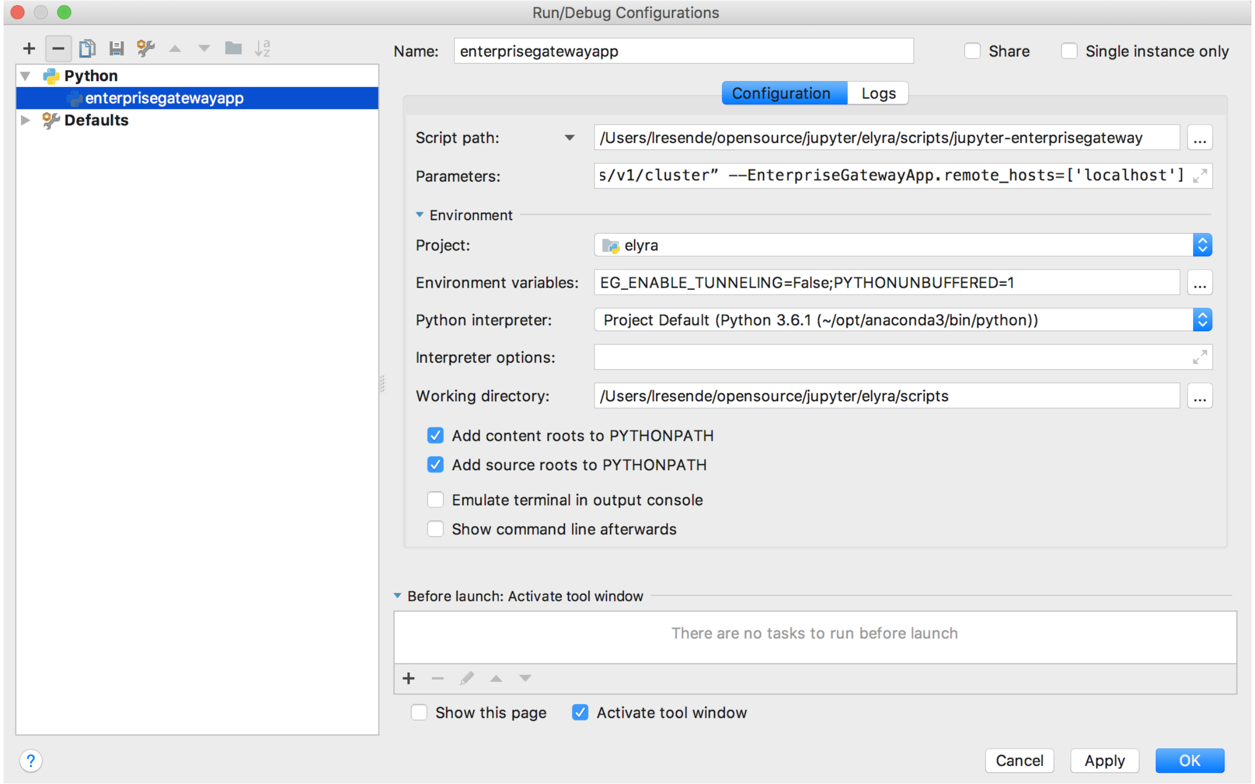
This page discusses how to go about debugging Enterprise Gateway. We also provide troubleshooting information on our *Troubleshooting* page.

17.1 Configuring your IDE for debugging Jupyter Enterprise Gateway

While your mileage may vary depending on which IDE you are using, the steps below (which was created using PyChar as an example) should be useful for configuring a debuging session for EG with minimum adjustments for different IDEs.

17.1.1 Creating a new Debug Configuration

Go to Run->Edit Configuration and create a new python configuration with the following settings:



Run/Debug Configurations

Name: enterprisegatewayapp

□ Share □ Single instance only

Script path: /Users/lresende/opensource/jupyter/elyra/scripts/jupyter-enterprisegateway

Parameters: `s/v1/cluster” --EnterpriseGatewayApp.remote_hosts=['localhost']`

Environment

Project: elyra

Environment variables: EG_ENABLE_TUNNELING=False;PYTHONUNBUFFERED=1

Python interpreter: Project Default (Python 3.6.1 (~/opt/anaconda3/bin/python))

Interpreter options:

Working directory: /Users/lresende/opensource/jupyter/elyra/scripts

☑ Add content roots to PYTHONPATH

☑ Add source roots to PYTHONPATH

□ Emulate terminal in output console

□ Show command line afterwards

Before launch: Activate tool window

There are no tasks to run before launch

□ Show this page ☑ Activate tool window

Script Path:

```
/Users/lresende/opensource/jupyter/elyra/scripts/jupyter-enterprisegateway
```

Parameters:

```
--ip=0.0.0.0
--log-level=DEBUG
--EnterpriseGatewayApp.yarn_endpoint=“http://elyra-fyi-node-1.fyre.ibm.com:8088/ws/v1/
↪cluster”
--EnterpriseGatewayApp.remote_hosts=['localhost']
```

Environment Variables:

```
EG_ENABLE_TUNNELING=False
```

Working Directotry:

```
/Users/lresende/opensource/jupyter/elyra/scripts
```

17.1.2 Running in debug mode

Now that you have handled the necessary configuration, use Run-Debug and select the debug configuration you just created and happy debuging.

CHAPTER

EIGHTEEN

CONTRIBUTING TO JUPYTER ENTERPRISE GATEWAY

Thank you for your interest in Jupyter Enterprise Gateway! If you would like to contribute to the project please first take a look at the Project Jupyter Contributor Documentation.

Prior to your contribution, we strongly recommend getting acquainted with Enterprise Gateway by checking out the *Development Workflow* and *System Architecture* pages.

CHAPTER

NINETEEN

DEVELOPMENT WORKFLOW

Here are instructions for setting up a development environment for the Jupyter Enterprise Gateway server. It also includes common steps in the developer workflow such as building Enterprise Gateway, running tests, building docs, packaging kernelspecs, etc.

19.1 Prerequisites

Install miniconda and GNU make on your system.

19.2 Clone the repo

Clone this repository in a local directory.

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

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

19.3 Make

Enterprise Gateway's build environment is centered around `make` and the corresponding `Makefile`.Entering `make` with no parameters yields the following:

```
activate                       Activate the virtualenv (default: enterprise-gateway-dev)
clean-images                   Remove docker images (includes kernel-based images)
clean-kernel-images            Remove kernel-based images
clean                          Make a clean source tree
dev                            Make a server in jupyter_websocket mode
dist                           Make source, binary and kernelspecs distribution to dist␣
↪folder
docker-images                  Build docker images (includes kernel-based images)
docs                           Make HTML documentation
env                            Make a dev environment
install                        Make a conda env with dist/*.whl and dist/*.tar.gz␣
↪installed
```

(continues on next page)

(continued from previous page)

```
itest-docker                   Run integration tests (optionally) against docker swarm
itest-yarn                     Run integration tests (optionally) against docker demo␣
↪(YARN) container
kernel-images                  Build kernel-based docker images
kernelspecs                    Create archives with sample kernelspecs
nuke                           Make clean + remove conda env
publish-images                 Push docker images to docker hub
release                        Make a wheel + source release on PyPI
test                           Run unit tests
```

Some of the more useful commands are listed below.

19.4 Build a conda environment

Build a Python 3 conda environment containing the necessary dependencies for running the enterprise gateway server, running tests, and building documentation.

```
make env
```

By default, the env built will be named `enterprise-gateway-dev`. To produce a different conda env, you can specify the name via the `ENV=` parameter.

```
make ENV=my-conda-env env
```

> **Note:** If using a non-default conda env, all `make` commands should include the `ENV=` parameter, otherwise the command will use the default environment.

19.5 Build the wheel file

Build a wheel file that can then be installed via `pip install`

```
make bdist
```

19.6 Build the kernelspec tar file

Enterprise Gateway includes two sets of kernelspecs for each of the three primary kernels: `IPython`,`IR`, and `Toree` to demonstrate remote kernels and their corresponding launchers. One set uses the `DistributedProcessProxy` while the other uses the `YarnClusterProcessProxy`. The following makefile target produces a tar file (`enterprise_gateway_kernelspecs.tar.gz`) in the `dist` directory.

```
make kernelspecs
```

Note: Because the scala launcher requires a jar file, `make kernelspecs` requires the use of `sbt` to build the scala launcher jar. Please consult the sbt site for directions to install/upgrade `sbt` on your platform. We currently prefer the use of 1.0.3.

19.7 Build distribution files

Builds the files necessary for a given release: the wheel file, the source tar file, and the kernelspecs tar file. This is essentially a helper target consisting of the `bdist` `sdist` and `kernelspecs` targets.

```
make dist
```

19.8 Run the Enterprise Gateway server

Run an instance of the Enterprise Gateway server.

```
make dev
```

Then access the running server at the URL printed in the console.

19.9 Build the docs

Run Sphinx to build the HTML documentation.

```
make docs
```

19.10 Run the unit tests

Run the unit test suite.

```
make test
```

19.11 Run the integration tests

Run the integration tests suite.

These tests will bootstrap a docker image with Apache Spark using YARN resource manager and Jupyter Enterprise Gateway and perform various tests for each kernel in both YARN client and YARN cluster mode.

```
make itest
```

19.12 Build the docker images

The following can be used to build all docker images used within the project. See *docker images* for specific details.

```
make docker-images
```

CHAPTER TWENTY

DOCKER IMAGES

All docker images can be pulled from docker hub's elyra organization and their docker files can be found in the github repository in the appropriate directory of etc/docker.

Local images can also be built via `make docker-images`.

CHAPTER

TWENTYONE

RUNTIME IMAGES

The following sections describe the docker images used within Kubernetes and Docker Swarm environments - all of which can be pulled from the Enterprise Gateway organization on dockerhub.

21.1 elyra/enterprise-gateway

The primary image for Kubernetes and Docker Swarm support, elyra/enterprise-gateway contains the Enterprise Gateway server software and default kernelspec files. For Kubernetes it is deployed using the enterprise-gateway.yaml file. For Docker Swarm, deployment can be accomplished using enterprise-gateway-swarm.sh although we should convert this to a docker compose yaml file at some point.

We recommend that a persistent/mounted volume be used so that the kernelspec files can be accessed outside of the container since we've found those to require post-deployment modifications from time to time.

21.2 elyra/kernel-py

Image elyra/kernel-py contains the IPython kernel. It is currently built on the jupyter/scipy-notebook image with additional support necessary for remote operation.

21.3 elyra/kernel-spark-py

Image elyra/kernel-spark-py is built on elyra/kernel-py and includes the Spark 2.4 distribution for use in Kubernetes clusters. Please note that the ability to use the kernel within Spark within a Docker Swarm configuration probably won't yield the expected results.

21.4 elyra/kernel-tf-py

Image elyra/kernel-tf-py contains the IPython kernel. It is currently built on the jupyter/tensorflow-notebook image with additional support necessary for remote operation.

21.5 elyra/kernel-scala

Image elyra/kernel-scala contains the Scala (Apache Toree) kernel and is built on elyra/spark which is, itself, built using the scripts provided by the Spark 2.4 distribution for use in Kubernetes clusters. As a result, the ability to use the kernel within Spark within a Docker Swarm configuration probably won't yield the expected results.

Since Toree is currently tied to Spark, creation of a *vanilla* mode Scala kernel is not high on our current set of priorities.

21.6 elyra/kernel-r

Image elyra/kernel-r contains the IRKernel and is currently built on the jupyter/r-notebook image.

21.7 elyra/kernel-spark-r

Image elyra/kernel-spark-r also contains the IRKernel but is built on elyra/kernel-r and includes the Spark 2.4 distribution for use in Kubernetes clusters.

CHAPTER TWENTYTWO

CUSTOM KERNEL IMAGES

This section presents information needed for how a custom kernel image could be built for your own uses with Enterprise Gateway. This is typically necessary if one desires to extend the existing image with additional supporting libraries or an image that encapsulates a different set of functionality altogether.

22.1 Extending Existing Kernel Images

A common form of customization occurs when the existing kernel image is serving the fundamentals but the user wishes it be extended with additional libraries so as to prevent the need of their imports within the Notebook interactions. Since the image already meets the basic requirements, this is really just a matter of referencing the existing image in the `FROM` statement and installing additional libraries. Because the EG kernel images do not run as the `root` user, you may need to switch users to perform the update.

```
FROM elyra/kernel-py:VERSION

USER root  # switch to root user to perform installation (if necessary)

RUN pip install my-libraries

USER $NB_UID  # switch back to the jovyan user
```

22.2 Bringing Your Own Kernel Image

Users that do not wish to extend an existing kernel image must be cognizant of a couple things.

1. Requirements of a kernel-based image to be used by Enterprise Gateway.
2. Is the base image one from Jupyter Docker-stacks?

22.2.1 Requirements for Custom Kernel Images

Custom kernel images require some support files from the Enterprise Gateway repository. These are packaged into a tar file for each release starting in `2.5.0`. This tar file (named `jupyter_enterprise_gateway_kernel_image_files-VERSION.tar.gz`) is composed of a few files - one bootstrap script and a kernel launcher (one per kernel type).

Bootstrap-kernel.sh

Enterprise Gateway provides a single bootstrap-kernel.sh script that handles the three kernel languages supported out of the box - Python, R, and Scala. When a kernel image is started by Enterprise Gateway, parameters used within the bootstrap-kernel.sh script are conveyed via environment variables. The bootstrap script is then responsible for validating and converting those parameters to meaningful arguments to the appropriate launcher.

Kernel Launcher

The kernel launcher, as discussed here does a number of things. In particular, it creates the connection ports and conveys that connection information back to Enterprise Gateway via the socket identified by the response address parameter. Although not a requirement for container-based usage, it is recommended that the launcher be written in the same language as the kernel. (This is more of a requirement when used in applications like YARN.)

22.2.2 About Jupyter Docker-stacks Images

Most of what is presented assumes the base image for your custom image is derived from the Jupyter Docker-stacks repository. As a result, it's good to cover what makes up those assumptions so you can build your own image independently from the docker-stacks repository.

All of the images produced from the docker-stacks repository come with a certain user configured. This user is named `jovyan` and is mapped to a user id (UID) of `1000` and a group id (GID) of `100` - named `users`.

The various startup scripts and commands typically reside in `/usr/local/bin` and we recommend trying to adhere to that policy.

The base jupyter image, upon which most all images from docker-stacks are built, also contains a `fix-permissions` script that is responsible for *gracefully* adjusting permissions based on its given parameters. By only changing the necessary permissions, use of this script minimizes the size of the docker layer in which that command is invoked durnig the build of the docker image.

22.2.3 Sample Dockerfiles for Custom Kernel Images

Below we provide two working Dockerfiles that produce custom kernel images. One based on an existing image from Jupyter docker-stacks, the other from an independent base image.

Custom Kernel Image Built on Jupyter Image

Here's an example Dockerfile that installs the minimally necessary items for a python-based kernel image built on the docker-stack image `jupyter/scipy-notebook`. Note: the string `VERSION` must be replaced with the appropriate value.

```
# Choose a base image.  Preferrably one from https://github.com/jupyter/docker-stacks
FROM jupyter/scipy-notebook:61d8aaedaeaf

# Switch user to root since, if from docker-stacks, its probably jovyan
USER root

# Install any packages required for the kernel-wrapper.  If the image
# does not contain the target kernel (i.e., IPython, IRkernel, etc.,
# it should be installed as well.
RUN pip install pycrypto

# Download and extract the enterprise gateway kernel launchers and bootstrap
# files and deploy to /usr/local/bin. Change permissions to NB_UID:NB_GID.
RUN wget https://github.com/jupyter/enterprise_gateway/releases/download/vVERSION/
↪jupyter_enterprise_gateway_kernel_image_files-VERSION.tar.gz &&\
        tar -xvf jupyter_enterprise_gateway_kernel_image_files-VERSION.tar.gz -C /usr/
↪local/bin &&\
        rm -f jupyter_enterprise_gateway_kernel_image_files-VERSION.tar.gz &&\
        fix-permissions /usr/local/bin

# Switch user back to jovyan and setup language and default CMD
USER $NB_UID
ENV KERNEL_LANGUAGE python
CMD /usr/local/bin/bootstrap-kernel.sh
```

Independent Custom Kernel Image

If your base image is not from docker-stacks, it is recommended that you NOT run the image as USER `root` and create an *image user* that is not UID 0. For this example, we will create the `jovyan` user with UID `1000` and a primary group of `users`, GID `100`. Note that Enterprise Gateway makes no assumption relative to the user in which the kernel image is running.

Aside from configuring the image user, all other aspects of customization are the same. In this case, we'll use the tensorflow-gpu image and convert it to be usable via Enterprise Gateway as a custom kernel image. Note that because this image didn't have `wget` we used `curl` to download the supporting kernel-image files.

```
FROM tensorflow/tensorflow:2.5.0-gpu-jupyter

USER root

# Install OS dependencies required for the kernel-wrapper. Missing
# packages can be installed later only if container is running as
# privileged user.
RUN apt-get update && apt-get install -yq --no-install-recommends \
    build-essential \
    libsm6 \
    libxext-dev \
```

(continues on next page)

(continued from previous page)

```
    libxrender1 \
    netcat \
    python3-dev \
    tzdata \
    unzip \
    && rm -rf /var/lib/apt/lists/*

# Install any packages required for the kernel-wrapper.  If the image
# does not contain the target kernel (i.e., IPython, IRkernel, etc.,
# it should be installed as well.
RUN pip install pycrypto

# Download and extract the enterprise gateway kernel launchers and bootstrap
# files and deploy to /usr/local/bin. Change permissions to NB_UID:NB_GID.
RUN curl -L https://github.com/jupyter/enterprise_gateway/releases/download/vVERSION/
↪jupyter_enterprise_gateway_kernel_image_files-VERSION.tar.gz | \
    tar -xz -C /usr/local/bin

RUN adduser --system --uid 1000 --gid 100 jovyan && \
    chown jovyan:users /usr/local/bin/bootstrap-kernel.sh && \
    chmod 0755 /usr/local/bin/bootstrap-kernel.sh && \
    chown -R jovyan:users /usr/local/bin/kernel-launchers

ENV NB_UID 1000
ENV NB_GID 100
USER jovyan
ENV KERNEL_LANGUAGE python
CMD /usr/local/bin/bootstrap-kernel.sh
```

22.3 Deploying Your Custom Kernel Image

The final step in deploying a customer kernel image is creating a corresponding kernelspec directory that is avaiable to Enterprise Gateway. Since Enterprise Gateway is also running in a container, its import that its kernelspecs folder either be mounted externally or a new EG image is created with the appropriate kernelspecs directory in place. For the purposes of this discussion, we'll assume the kernelspecs directory, `/usr/local/share/jupyter/kernels` is externally mounted.

- Find a similar kernelspec directory from which to create your custom kernelspec. The most important aspect to this is matching the language of your kernel since it will use the same kernel launcher. Another important question is whether or not your custom kernel uses Spark, because those kernelspecs will vary significantly since many of the spark options reside in the kernel.json's `env` stanza. Since our examples use *vanilla* (non-Spark) python kernels we'll use the `python_kubernetes` kernelspec as our basis.

```
cd /usr/local/share/jupyter/kernels
cp -r python_kubernetes python_myCustomKernel
```

- Edit the `kernel.json` file and change the `display_name:`, `image_name:` and path to `launch_kubernetes.py` script.

```
{
  "language": "python",
```

(continues on next page)

(continued from previous page)

```
  "display_name": "My Custom Kernel",
  "metadata": {
    "process_proxy": {
      "class_name": "enterprise_gateway.services.processproxies.k8s.
↪KubernetesProcessProxy",
      "config": {
        "image_name": "myDockerHub/myCustomKernelImage:myTag"
      }
    }
  },
  "env": {
  },
  "argv": [
    "python",
    "/usr/local/share/jupyter/kernels/python_myCustomKernel/scripts/launch_kubernetes.py
↪",
    "--RemoteProcessProxy.kernel-id",
    "{kernel_id}",
    "--RemoteProcessProxy.response-address",
    "{response_address}",
    "--RemoteProcessProxy.public-key",
    "{public_key}"
  ]
}
```

- If using a whitelist (`EG_KERNEL_WHITELIST`), be sure to update it with the new kernelspec directory name (e.g., `python_myCustomKernel`) and restart/redeploy Enterprise Gateway.
- Launch or refresh your Notebook session and confirm `My Custom Kernel` appears in the *new kernel* drop-down.
- Create a new notebook using `My Custom Kernel`.

CHAPTER

TWENTYTHREE

ANCILLARY DOCKER IMAGES

The project produces two docker images to make testing easier: elyra/demo-base and elyra/enterprise-gateway-demo.

23.1 elyra/demo-base

The elyra/demo-base image is considered the base image upon which elyra/enterprise-gateway-demo is built. It consist of a Hadoop (YARN) installation that includes Spark, Java, miniconda and various kernel installations.

The primary use of this image is to quickly build elyra/enterprise-gateway images for testing and development purposes. To build a local image, run `make demo-base`.

As of the 0.9.0 release, this image can be used to start a separate YARN cluster that, when combined with another instance of elyra/enterprise-gateway can better demonstrate remote kernel functionality.

23.2 elyra/enterprise-gateway-demo

Built on elyra/demo-base, elyra/enterprise-gateway-demo also includes the various example kernelspecs contained in the repository.

By default, this container will start with enterprise gateway running as a service user named `jovyan`. This user is enabled for `sudo` so that it can emulate other users where necessary. Other users included in this image are `elyra`, `bob` and `alice` (names commonly used in security-based examples).

We plan on producing one image per release to the enterprise-gateway-demo docker repo where the image's tag reflects the corresponding release.

To build a local image, run `make docker-image-enterprise-gateway-demo`. Because this is a development build, the tag for this image will not reflect the value of the VERSION variable in the root Makefile but will be 'dev'.

CHAPTER TWENTYFOUR

PROJECT ROADMAP

We have plenty to do, now and in the future. Here's where we're headed:

- Kernel Configuration Profile
  - Enable client to request different resource configurations for kernels (e.g. small, medium, large)
  - Profiles should be defined by Administrators and enabled for users and/or groups.
- Administration UI
  - Dashboard with running kernels
  - Lifecycle management
  - Time running, stop/kill, Profile Management, etc
- Support for other resource managers
- User Environments
- High Availability
  - Session persistence

We'd love to hear any other use cases you might have and look forward to your contributions to Jupyter Enterprise Gateway.