Enterprise Gateway Documentation

Release 1.1.1

Project Jupyter team

May 03, 2019

USER DOCUMENTATION

Jupyter Enterprise Gateway is a web server (built directly on Jupyter Kernel Gateway) that enables the ability to launch kernels on behalf of remote notebooks throughout your enterprise compute cluster. This enables better resource management since the web server is no longer the single location for kernel activity which, in Big Data environments, can result in large processes that together deplete your single node of its resources.

Fig. 1: 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, increasing the number of simultaneously active kernels dramatically.

Fig. 2: 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 Apache Spark 2.x cluster.

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

- Spark Standalone
- YARN Resource Manager - Client Mode
- YARN Resource Manager - Cluster Mode

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 must be installed. To support IPython kernels and R kernels to run in YARN containers, various packages have to be installed on each of the YARN data 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
- A CLI for launching the enterprise gateway server: `jupyter enterprisegateway OPTIONS`
- A Python 2.7 and 3.3+ compatible implementation

1.2 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 using `pip`(part of Anaconda) 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
```

1.3 Installing Kernels

Please follow the link below to learn more specific details about how to install/configure specific kernels with Jupyter Enterprise Gateway:

- *Installing and Configuring kernels*

1.4 Configuring Spark Resource Managers

To leverage the full distributed capabilities of Spark, Jupyter Enterprise Gateway has provided deep integrarion with YARN resource manager. Having said that, EG also supports running in pseudo-distributed utilizing both YARN client or Spark Standalone modes.

Please follow the links below to learn more specific details about how to enable/configure the different modes:

- *Enabling Cluster Mode support*
- *Enabling Client Mode/Standalone support*

1.5 Starting Enterprise Gateway

Very few arguments are necessary to minimally start Enterprise Gateway. The following command could be considered a minimal command:

```
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.

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

NB2KG is used to connect a Notebook from a local desktop or laptop to the Enterprise Gateway instance on the Spark/YARN cluster. We strongly recommend that NB2KG v0.1.0 be used as our team has provided some security enhancements to enable for conveying the notebook user (for configurations when Enterprise Gateway is running behind a secured gateway) and allowing for increased request timeouts (due to the longer kernel startup times when interacting with the resource manager or distribution operations).

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 KG_REQUEST_TIMEOUT=30
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
```

For your convenience, we have also built a docker image (elyra/nb2kg) with Jupyter Notebook, Jupyter Lab and NB2KG which can be launched by the command below:

```
docker run -t --rm \
  -e KG_URL='http://<master ip>:8888' \
  -e KG_HTTP_USER=guest \
  -e KG_HTTP_PASS=guest-password \
  -p 8888:8888 \
  -e VALIDATE_KG_CERT='no' \
  -e LOG_LEVEL=DEBUG \
  -e KG_REQUEST_TIMEOUT=40 \
  -e KG_CONNECT_TIMEOUT=40 \
  -v ${HOME}/notebooks/:/tmp/notebooks \
  -w /tmp/notebooks \
  elyra/nb2kg
```

To invoke Jupyter Lab, simply add `lab` as the last option following the image name (e.g., `elyra/nb2kg lab`).

CHAPTER

TWO

ENABLING CLUSTER MODE SUPPORT

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

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

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

Configuring Kernels for YARN Cluster mode

For each supported Jupyter Kernel, we have provided sample kernel configurations and launchers as part of the release e.g. jupyter_enterprise_gateway_kernelspecs-1.1.0.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-incubator/enterprise_gateway/releases/download/v1.1.0/
↪enterprise_gateway_kernelspecs.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_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)",
  "process_proxy": {
    "class_name": "enterprise_gateway.services.processproxies.yarn.
↪YarnClusterProcessProxy"
  },
```

(continues on next page)

(continued from previous page)

```
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/anaconda3/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.4-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.
↪local/lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/
↪current/spark2-client/python/lib/py4j-0.10.4-src.zip,PATH=/opt/anaconda2/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",
    "{connection_file}",
    "--RemoteProcessProxy.response-address",
    "{response_address}"
  ]
}
```

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 THREE

ENABLING CLIENT MODE/STANDALONE SUPPORT

Jupyter Enterprise Gateway extends Jupyter Kernel Gateway and is 100% compatible with JKG, which means that by installing kernels in Enterprise Gateway and using the vanila 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 YARN Client mode or using vanilla 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
```

Configuring Kernels for YARN Client mode

For each supported Jupyter Kernel, we have provided sample kernel configurations and launchers as part of the release e.g. jupyter_enterprise_gateway_kernelspecs-1.1.0.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-incubator/enterprise_gateway/releases/download/v1.1.0/
↪enterprise_gateway_kernelspecs.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)",
  "process_proxy": {
    "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy"
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/anaconda3/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.4-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.
↪local/lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/
↪current/spark2-client/python/lib/py4j-0.10.4-src.zip,PATH=/opt/anaconda2/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",
    "{connection_file}",
    "--RemoteProcessProxy.response-address",
    "{response_address}"
  ]
}
```

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.

Configuring Kernels for Spark Standalone mode

The main difference between YARN Client and Standalone is the values used in SPARK_OPTS for the `--master` parameter.

Please see below how a kernel.json would look like for integrating with Spark Standalone:

```
{
  "language": "python",
  "display_name": "Spark - Python (YARN Client Mode)",
  "process_proxy": {
    "class_name": "enterprise_gateway.services.processproxies.distributed.
↪DistributedProcessProxy"
  },
  "env": {
    "SPARK_HOME": "/usr/hdp/current/spark2-client",
    "PYSPARK_PYTHON": "/opt/anaconda3/bin/python",
    "PYTHONPATH": "${HOME}/.local/lib/python3.6/site-packages:/usr/hdp/current/spark2-
↪client/python:/usr/hdp/current/spark2-client/python/lib/py4j-0.10.4-src.zip",
    "SPARK_YARN_USER_ENV": "PYTHONUSERBASE=/home/yarn/.local,PYTHONPATH=${HOME}/.
↪local/lib/python3.6/site-packages:/usr/hdp/current/spark2-client/python:/usr/hdp/
↪current/spark2-client/python/lib/py4j-0.10.4-src.zip,PATH=/opt/anaconda2/bin:$PATH",
    "SPARK_OPTS": "--master spark://127.0.0.1:7077  --name ${KERNEL_ID:-ERROR__NO__
↪KERNEL_ID} --conf spark.yarn.submit.waitAppCompletion=false",
    "LAUNCH_OPTS": ""
  },
  "argv": [
```

(continues on next page)

(continued from previous page)

```
    "/usr/local/share/jupyter/kernels/spark_python_yarn_client/bin/run.sh",
    "{connection_file}",
    "--RemoteProcessProxy.response-address",
    "{response_address}"
  ]
}
```

CHAPTER FOUR

ADVANCED FEATURES

4.1 Culling idle kernels

With the adoption of notebooks and iteractive development for data science, a new "resource utilization" pattern has arrised, where kernel resources are locked for a given notebook, but due to iteractive development process it might be idle for a long period of time causing the cluster resources to starve and a 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). Positive values less than 300 (5 minutes) will be adjusted to 300.

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).

Below, see an updated start script that provides some default configuration to enable 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

SUPPORTED KERNELS

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

We provide sample kernel configuration and launcher tar files as part of each release (e.g. jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz) that can be extracted and modified to fit your configuration.

Please find below more details on specific kernels:

5.1 Scala kernel (Apache Toree kernel)

We have tested the latest version of Apache Toree with Scala 2.11 support.

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

Install Apache Toree

```
# pip-install the Apache Toree installer
pip install https://dist.apache.org/repos/dist/release/incubator/toree/0.2.0-
↪incubating/toree-pip/toree-0.2.0.tar.gz

# install a new Toree Scala kernel which will be updated with Enterprise Gateway's␣
↪custom kernel scripts
jupyter toree install --spark_home="${SPARK_HOME}" --kernel_name="Spark 2.x" --
↪interpreters="Scala"
```

Update the 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** and **spark_scala_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter-incubator/enterprise_gateway/releases/download/v1.1.0/
↪jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz

SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "spark_scala" | awk '{print $2}
↪')"

KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"
```

(continues on next page)

(continued from previous page)

```
tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_scala_yarn_cluster/ spark_scala_yarn_cluster/

tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_scala_yarn_client/ spark_scala_yarn_client/

cp $KERNELS_FOLDER/spark_scala/lib/*.jar  $KERNELS_FOLDER/spark_scala_yarn_cluster/lib
```

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

5.2 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** and **spark_python_yarn_client** to where the Jupyter kernels are installed (e.g. jupyter kernelspec list)

```
wget https://github.com/jupyter-incubator/enterprise_gateway/releases/download/v1.1.0/
↪jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz

SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "spark_scala" | awk '{print $2}
↪')"

KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"

tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_python_yarn_cluster/ spark_python_yarn_cluster/

tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.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.

5.3 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
Rscript -e 'install.packages("argparser", repos="https://cran.rstudio.com")'


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

(continues on next page)

(continued from previous page)

```
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 iR Kernelspecs

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

```
wget https://github.com/jupyter-incubator/enterprise_gateway/releases/download/v1.1.0/
↪jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz

SCALA_KERNEL_DIR="$(jupyter kernelspec list | grep -w "spark_scala" | awk '{print $2}
↪')"

KERNELS_FOLDER="$(dirname "${SCALA_KERNEL_DIR}")"

tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.tar.gz --strip 1 --directory
↪$KERNELS_FOLDER/spark_R_yarn_cluster/ spark_R_yarn_cluster/

tar -zxvf jupyter_enterprise_gateway_kernelspecs-1.1.0.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 iR kernel page.

CHAPTER

SIX

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 nb2kg-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 NB2KG 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.

6.1 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.

6.1.1 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.

6.1.2 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.

6.1.3 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.
```

6.2 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.

6.2.1 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.

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)

6.2.2 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**.

6.3 SSH Tunneling

Jupyter Enterprise Gateway is now 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`

6.4 Securing Enterprise Gateway Server

6.4.1 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 NB2KG serverextension.

   During Jupyter notebook server startup, export the following environment variables where NB2KG will access during runtime:

```
export KG_CLIENT_CERT=${PATH_TO_PEM_FILE}
export KG_CLIENT_KEY=${PATH_TO_KEY_FILE}
export KG_CLIENT_CA=${PATH_TO_SELFSIGNED_CA}
```

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

6.4.2 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 NB2KG and Enterprise Gateway is SSL enabled.

CHAPTER

SEVEN

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 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.

CHAPTER

EIGHT

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.

8.1 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 YARN. Enterprise Gateway introduces these capabilities by extending the existing class hierarchies for `KernelManager`, `MultiKernelManager` and `KernelSpec` classes, along with an additional abstraction known as a *process proxy*.

8.1.1 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 JKG 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.

8.1.2 Remote Kernel Spec

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 introduces a new subclass of KernelSpec named `RemoteKernelSpec`.

The `RemoteKernelSpec` class provides support for a new (and optional) stanza within the kernelspec file. This stanza is named `process_proxy` and identifies the class that provides the kernel's process abstraction (while allowing for future extensions).

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)",
  "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",
    "--profile",
    "{connection_file}",
    "--RemoteProcessProxy.response-address",
    "{response_address}"
  ]
}
```

The `RemoteKernelSpec` class definition can be found in remotekernelspec.py

See the *Process Proxy* section for more details.

8.2 Remote Mapping Kernel Manager

`RemoteMappingKernelManager` is a subclass of JKG's existing `SeedingMappingKernelManager` 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.

8.3 Remote Kernel Manager

`RemoteKernelManager` is a subclass of JKG's existing `KernelGatewayIOLoopKernelManager` 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.

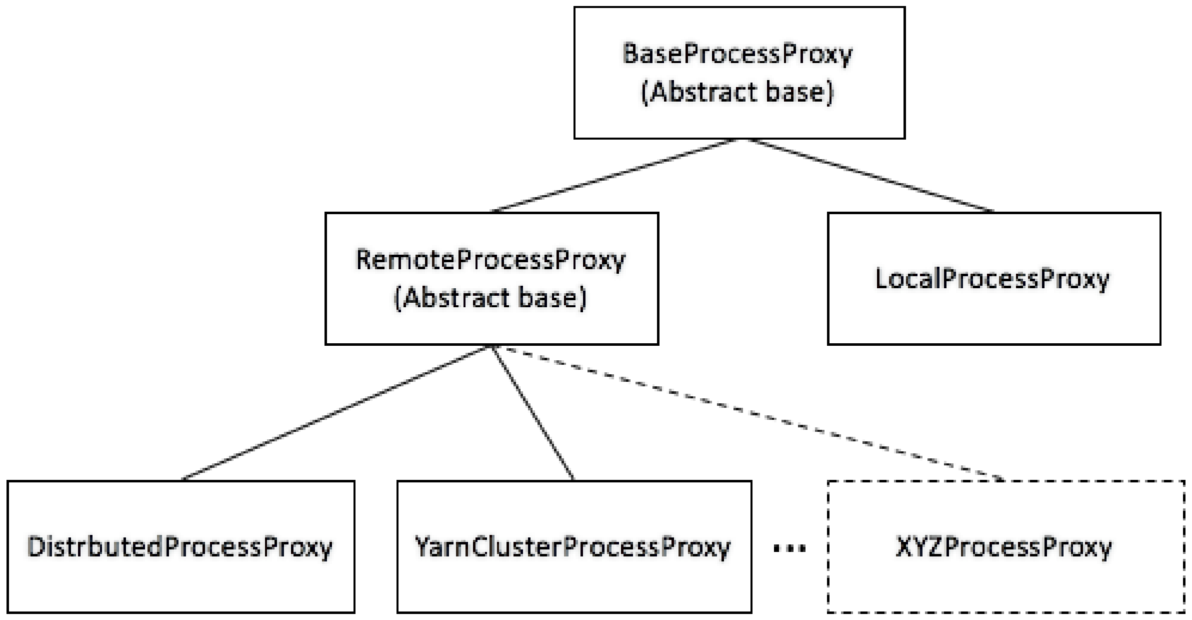
Both `RemoteMappingKernelManager` and `RemoteKernelManager` class definitions can be found in remotemanager.py

8.4 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 two built-in subclasses of `RemoteProcessProxy` - `DistributedProcessProxy` - representing a proof of concept class that remotes a kernel via ssh and `YarnClusterProcessProxy` - representing the design target of launching kernels hosted as yarn applications via yarn/cluster mode. These class definitions can be found in the processproxies package.



Process Class Hierarchy

The process proxy constructor looks as follows:

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

where

- kernel_manager is an instance of a RemoteKernelManager class that is associated with the corresponding RemoteKernelSpec instance.
- 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.

8.4.1 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()`:

```
@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 *Launchers*)

```
@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 `NB2KG`, 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 launcher 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.

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.

8.4.2 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:

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

8.5 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*.

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 three primary tasks of a launcher:

1. Creation of the connection file on the remote (target) system
2. Conveyance of the connection information back to the Enterprise Gateway process
3. Invocation of the target kernel

Launchers are minimally invoked with two parameters (both of which are conveyed by the `argv` stanza of the corresponding kernel.json file) - a path to the ***non-existent*** connection file represented by the placeholder `{connection_file}` and 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}`.

Depending on the target kernel, the connection file parameter may or may not be identified by an argument name. However, 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.

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

```
{
  "language": "python",
  "display_name": "Spark - Python (YARN Cluster Mode)",
  "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",
    "{connection_file}",
    "--RemoteProcessProxy.response-address",
    "{response_address}"
  ]
}
```

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 \
     "${SPARK_HOME}/bin/spark-submit" \
     "${SPARK_OPTS}" \
     "${PROG_HOME}/scripts/launch_ipykernel.py" \
     "${LAUNCH_OPTS}" \
     "$@"
```

8.6 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.

CHAPTER

NINE

CONFIGURATION OPTIONS

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

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

Note that because Enterprise Gateway is built on Kernel Gateway, all of the `KernelGatewayApp` options can be specified as `EnterpriseGatewayApp` options. In addition, the `KG_` prefix of inherited environment variables has also been preserved, while those variables introduced by Enterprise Gateway will be prefixed with `EG_`.

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> (KernelGatewayApp.ip)
    Default: '127.0.0.1'
    IP address on which to listen (KG_IP env var)
--port=<Int> (KernelGatewayApp.port)
    Default: 8888
    Port on which to listen (KG_PORT env var)
--port_retries=<Int> (KernelGatewayApp.port_retries)
    Default: 50
    Number of ports to try if the specified port is not available
    (KG_PORT_RETRIES env var)
--api=<Unicode> (KernelGatewayApp.api)
    Default: 'kernel_gateway.jupyter_websocket'
    Controls which API to expose, that of a Jupyter notebook server, the seed
    notebook's, or one provided by another module, respectively using values
    'kernel_gateway.jupyter_websocket', 'kernel_gateway.notebook_http', or
    another fully qualified module name (KG_API env var)
--seed_uri=<Unicode> (KernelGatewayApp.seed_uri)
    Default: None
    Runs the notebook (.ipynb) at the given URI on every kernel launched. No
    seed by default. (KG_SEED_URI env var)
--keyfile=<Unicode> (KernelGatewayApp.keyfile)
    Default: None
    The full path to a private key file for usage with SSL/TLS. (KG_KEYFILE env
    var)
--certfile=<Unicode> (KernelGatewayApp.certfile)
    Default: None
    The full path to an SSL/TLS certificate file. (KG_CERTFILE env var)
--client-ca=<Unicode> (KernelGatewayApp.client_ca)
    Default: None
    The full path to a certificate authority certificate for SSL/TLS client
    authentication. (KG_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 options
----------------------------
--EnterpriseGatewayApp.allow_credentials=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Credentials header. (KG_ALLOW_CREDENTIALS env
    var)
--EnterpriseGatewayApp.allow_headers=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Headers header. (KG_ALLOW_HEADERS env var)
--EnterpriseGatewayApp.allow_methods=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Methods header. (KG_ALLOW_METHODS env var)
--EnterpriseGatewayApp.allow_origin=<Unicode>
    Default: ''
    Sets the Access-Control-Allow-Origin header. (KG_ALLOW_ORIGIN env var)
--EnterpriseGatewayApp.answer_yes=<Bool>
```

(continues on next page)

(continued from previous page)

```
    Default: False
    Answer yes to any prompts.
--EnterpriseGatewayApp.api=<Unicode>
    Default: 'kernel_gateway.jupyter_websocket'
    Controls which API to expose, that of a Jupyter notebook server, the seed
    notebook's, or one provided by another module, respectively using values
    'kernel_gateway.jupyter_websocket', 'kernel_gateway.notebook_http', or
    another fully qualified module name (KG_API env var)
--EnterpriseGatewayApp.auth_token=<Unicode>
    Default: ''
    Authorization token required for all requests (KG_AUTH_TOKEN env var)
--EnterpriseGatewayApp.authorized_users=<Set>
    Default: set()
    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)
--EnterpriseGatewayApp.base_url=<Unicode>
    Default: '/'
    The base path for mounting all API resources (KG_BASE_URL env var)
--EnterpriseGatewayApp.certfile=<Unicode>
    Default: None
    The full path to an SSL/TLS certificate file. (KG_CERTFILE env var)
--EnterpriseGatewayApp.client_ca=<Unicode>
    Default: None
    The full path to a certificate authority certificate for SSL/TLS client
    authentication. (KG_CLIENT_CA env var)
--EnterpriseGatewayApp.conductor_endpoint=<Unicode>
    Default: None
    The http url for accessing the Conductor REST API. (EG_CONDUCTOR_ENDPOINT
    env var)
--EnterpriseGatewayApp.config_file=<Unicode>
    Default: ''
    Full path of a config file.
--EnterpriseGatewayApp.config_file_name=<Unicode>
    Default: ''
    Specify a config file to load.
--EnterpriseGatewayApp.default_kernel_name=<Unicode>
    Default: ''
    Default kernel name when spawning a kernel (KG_DEFAULT_KERNEL_NAME env var)
--EnterpriseGatewayApp.env_process_whitelist=<List>
    Default: []
    Environment variables allowed to be inherited from the spawning process by
    the kernel
--EnterpriseGatewayApp.expose_headers=<Unicode>
    Default: ''
    Sets the Access-Control-Expose-Headers header. (KG_EXPOSE_HEADERS env var)
--EnterpriseGatewayApp.force_kernel_name=<Unicode>
    Default: ''
    Override any kernel name specified in a notebook or request
    (KG_FORCE_KERNEL_NAME env var)
--EnterpriseGatewayApp.generate_config=<Bool>
    Default: False
    Generate default config file.
--EnterpriseGatewayApp.impersonation_enabled=<Bool>
```

(continues on next page)

(continued from previous page)

```
    Default: False
    Indicates whether impersonation will be performed during kernel launch.
    (EG_IMPERSONATION_ENABLED env var)
--EnterpriseGatewayApp.ip=<Unicode>
    Default: '127.0.0.1'
    IP address on which to listen (KG_IP env var)
--EnterpriseGatewayApp.kernel_manager_class=<Type>
    Default: 'enterprise_gateway.services.kernels.remotemanager.RemoteMapp...
    The kernel manager class to use. Should be a subclass of
    `notebook.services.kernels.MappingKernelManager`.
--EnterpriseGatewayApp.kernel_spec_manager_class=<Type>
    Default: 'enterprise_gateway.services.kernelspecs.remotekernelspec.Rem...
    The kernel spec manager class to use. Should be a subclass of
    `jupyter_client.kernelspec.KernelSpecManager`.
--EnterpriseGatewayApp.keyfile=<Unicode>
    Default: None
    The full path to a private key file for usage with SSL/TLS. (KG_KEYFILE env
    var)
--EnterpriseGatewayApp.log_datefmt=<Unicode>
    Default: '%Y-%m-%d %H:%M:%S'
    The date format used by logging formatters for %(asctime)s
--EnterpriseGatewayApp.log_format=<Unicode>
    Default: '[%(name)s]%(highlevel)s %(message)s'
    The Logging format template
--EnterpriseGatewayApp.log_level=<Enum>
    Default: 30
    Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')
    Set the log level by value or name.
--EnterpriseGatewayApp.max_age=<Unicode>
    Default: ''
    Sets the Access-Control-Max-Age header. (KG_MAX_AGE env var)
--EnterpriseGatewayApp.max_kernels=<Int>
    Default: None
    Limits the number of kernel instances allowed to run by this gateway.
    Unbounded by default. (KG_MAX_KERNELS env var)
--EnterpriseGatewayApp.max_kernels_per_user=<Int>
    Default: -1
    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)
--EnterpriseGatewayApp.port=<Int>
    Default: 8888
    Port on which to listen (KG_PORT env var)
--EnterpriseGatewayApp.port_range=<Unicode>
    Default: '0..0'
    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)
--EnterpriseGatewayApp.port_retries=<Int>
    Default: 50
    Number of ports to try if the specified port is not available
    (KG_PORT_RETRIES env var)
--EnterpriseGatewayApp.prespawn_count=<Int>
    Default: None
    Number of kernels to prespawn using the default language. No prespawn by
```

(continues on next page)

(continued from previous page)

```
    default. (KG_PRESPAWN_COUNT env var)
--EnterpriseGatewayApp.remote_hosts=<List>
    Default: ['localhost']
    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)
--EnterpriseGatewayApp.seed_uri=<Unicode>
    Default: None
    Runs the notebook (.ipynb) at the given URI on every kernel launched. No
    seed by default. (KG_SEED_URI env var)
--EnterpriseGatewayApp.trust_xheaders=<CBool>
    Default: False
    Use x-* header values for overriding the remote-ip, useful when application
    is behing a proxy. (KG_TRUST_XHEADERS env var)
--EnterpriseGatewayApp.unauthorized_users=<Set>
    Default: {'root'}
    Comma-separated list of user names (e.g., ['root','admin']) against which
    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)
--EnterpriseGatewayApp.yarn_endpoint=<Unicode>
    Default: 'http://localhost:8088/ws/v1/cluster'
    The http url for accessing the YARN Resource Manager. (EG_YARN_ENDPOINT env
    var)
--EnterpriseGatewayApp.yarn_endpoint_security_enabled=<Bool>
    Default: False
    Is YARN Kerberos/SPNEGO Security enabled (True/False).
    (EG_YARN_ENDPOINT_SECURITY_ENABLED env var)

NotebookHTTPPersonality options
-------------------------------
--NotebookHTTPPersonality.allow_notebook_download=<Bool>
    Default: False
    Optional API to download the notebook source code in notebook-http mode,
    defaults to not allow
--NotebookHTTPPersonality.cell_parser=<Unicode>
    Default: 'kernel_gateway.notebook_http.cell.parser'
    Determines which module is used to parse the notebook for endpoints and
    documentation. Valid module names include
    'kernel_gateway.notebook_http.cell.parser' and
    'kernel_gateway.notebook_http.swagger.parser'. (KG_CELL_PARSER env var)
--NotebookHTTPPersonality.comment_prefix=<Dict>
    Default: {'scala': '//', None: '#'}
    Maps kernel language to code comment syntax
--NotebookHTTPPersonality.static_path=<Unicode>
    Default: None
    Serve static files on disk in the given path as /public, defaults to not
    serve

JupyterWebsocketPersonality options
-----------------------------------
--JupyterWebsocketPersonality.env_whitelist=<List>
    Default: []
    Environment variables allowed to be set when a client requests a new kernel
--JupyterWebsocketPersonality.list_kernels=<Bool>
    Default: False
    Permits listing of the running kernels using API endpoints /api/kernels and
```

(continues on next page)

(continued from previous page)

```
    /api/sessions (KG_LIST_KERNELS env var). Note: Jupyter Notebook allows this
    by default but kernel gateway does not.
```

9.1 Addtional supported environment variables

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

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_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_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_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.

EG_LOCAL_IP_BLACKLIST=''
    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_LOCAL_IP_BLACKLIST=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
```

The following environment variables may be useful for troubleshooting:

```
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
```

(continues on next page)

(continued from previous page)

```
    value can be adjusted in situations where YARN troubleshooting may be warranted.

  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_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.
```

9.2 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.

CHAPTER

TEN

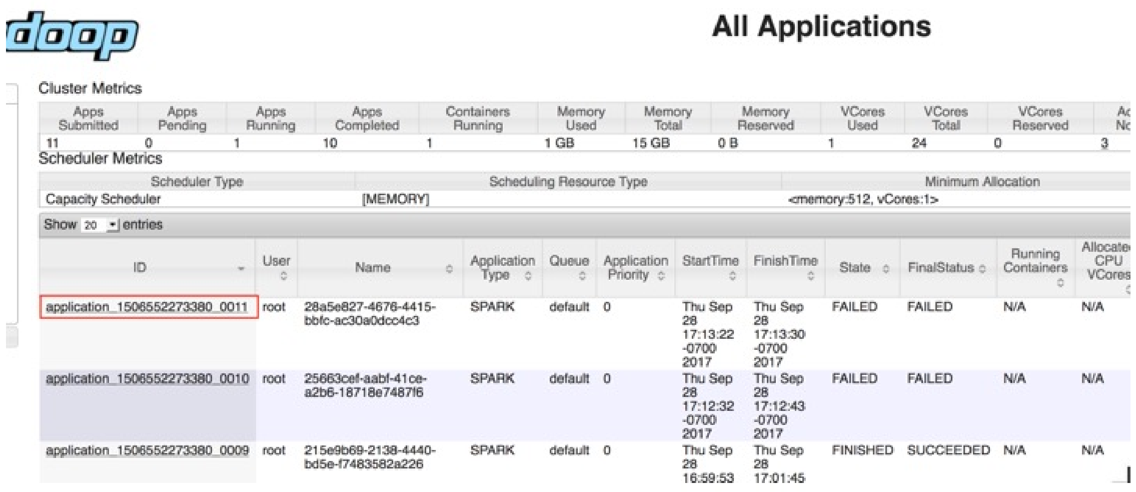
TROUBLESHOOTING

- **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

YARN ResourceManager UI

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/anaconda2/lib/python2.7/site-packages/tornado/web.py", line 1512, in␣
↪_execute
    result = yield result
  File "/opt/anaconda2/lib/python2.7/site-packages/tornado/gen.py", line 1055, in␣
↪run
    value = future.result()
  ....
  ....
  ....
  File "/opt/anaconda2/lib/python2.7/site-packages/enterprise_gateway/services/
↪kernels/remotemanager.py", line 125, in _launch_kernel
    return self.process_proxy.launch_process(kernel_cmd, **kw)
  File "/opt/anaconda2/lib/python2.7/site-packages/enterprise_gateway/services/
↪processproxies/yarn.py", line 63, in launch_process
    self.confirm_remote_startup(kernel_cmd, **kw)
  File "/opt/anaconda2/lib/python2.7/site-packages/enterprise_gateway/services/
↪processproxies/yarn.py", line 174, in confirm_remote_startup
    ready_to_connect = self.receive_connection_info()
  File "/opt/anaconda2/lib/python2.7/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 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.

- **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.

- **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/anaconda2/lib/python2.7/site-packages/tornado/web.py", line 1511, in␣
↪_execute
    result = yield result
  File "/opt/anaconda2/lib/python2.7/site-packages/tornado/gen.py", line 1055, in␣
↪run
    value = future.result()
  ....
  ....
  File "/opt/anaconda2/lib/python2.7/site-packages/enterprise_gateway/services/
↪processproxies/processproxy.py", line 478, in __init__
    super(RemoteProcessProxy, self).__init__(kernel_manager, proxy_config)
  File "/opt/anaconda2/lib/python2.7/site-packages/enterprise_gateway/services/
↪processproxies/processproxy.py", line 87, in __init__
    self._validate_port_range(proxy_config)
  File "/opt/anaconda2/lib/python2.7/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].

CHAPTER

ELEVEN

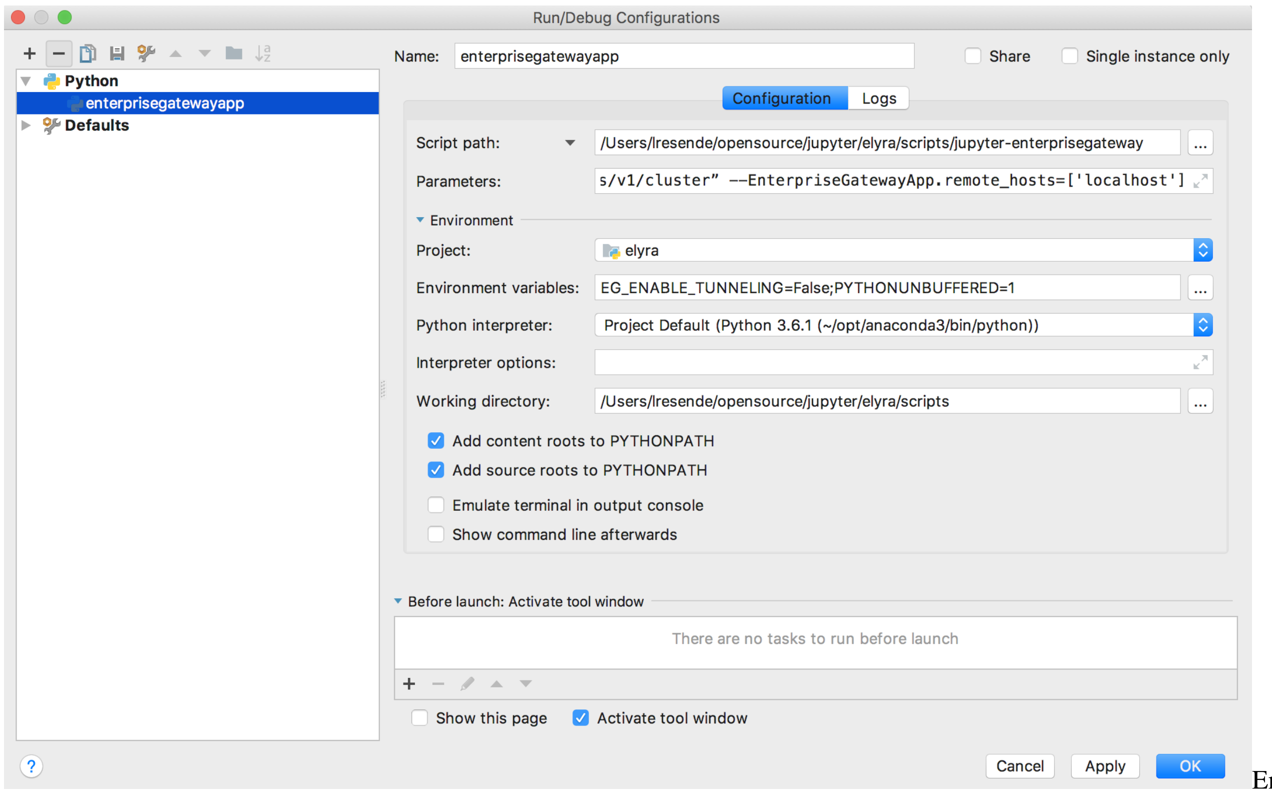
DEBUGGING JUPYTER ENTERPRISE GATEWAY

11.1 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.

11.1.1 Creating a new Debug Configuration

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



Run/Debug Configurations

Python
enterprisegatewayapp
Defaults

Name: enterprisegatewayapp
Share
Single instance only

Configuration | Logs

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

Cancel | Apply | OK

Enterprise Gateway debug configuration

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
```

11.1.2 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

TWELVE

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

THIRTEEN

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.

13.1 Prerequisites

Install miniconda and GNU make on your system.

13.2 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-incubator/enterprise_gateway.git
```

13.3 Make

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

```
activate                       eval `make activate`
clean                          Make a clean source tree
dev                            Make a server in jupyter_websocket mode
dist                           Make binary and source distribution to dist folder
docker-clean-enterprise-gateway Remove elyra/enterprise-gateway:dev docker image
docker-clean-nb2kg             Remove elyra/nb2kg:dev docker image
docker-clean-yarn-spark        Remove elyra/yarn-spark:2.1.0 docker image
docker-clean                   Remove docker images
docker-image-enterprise-gateway Build elyra/enterprise-gateway:dev docker image
docker-image-nb2kg             Build elyra/nb2kg:dev docker image
docker-image-yarn-spark        Build elyra/yarn-spark:2.1.0 docker image
docker-images                  Build docker images
docs                           Make HTML documentation
```

(continues on next page)

(continued from previous page)

```
env                             Make a dev environment
install                         Make a conda env with dist/*.whl and dist/*.tar.gz␣
↪installed
itest                           Run integration tests (optionally) against docker␣
↪container
kernelspecs                     Create an archive with sample kernelspecs
nuke                            Make clean + remove conda env
release                         Make a wheel + source release on PyPI
test                            Run unit tests
```

Some of the more useful commands are listed below.

13.4 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.

13.5 Build the wheel file

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

```
make dist
```

13.6 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.

13.7 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.

13.8 Build the docs

Run Sphinx to build the HTML documentation.

```
make docs
```

13.9 Run the unit tests

Run the unit test suite.

```
make test
```

13.10 Run the integration tests

Run the integration tests suite. T

hese 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
```

13.11 Build the docker images

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

```
make docker-images
```

CHAPTER

FOURTEEN

DOCKER IMAGES

The project produces three docker images to make both testing and general usage easier:

1. elyra/yarn-spark
2. elyra/enterprise-gateway
3. elyra/nb2kg

All 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`.

14.1 elyra/yarn-spark

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

The tag of this image reflects the version of Spark included in the image and we don't anticipate the need to update this image regularly.

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 docker-image-yarn-spark`. Note: the tag for this image will still be `:2.1.0` since `elyra/enterprise-gateway` depends on this image/tag.

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.

14.2 elyra/enterprise-gateway

Image elyra/enterprise-gateway is the primary image produced by this repository. Built on elyra/yarn-spark, it 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 `elyra`. This user is enabled for `sudo` so that it can emulate other users where necessary. Other users included in this image are `jovyan` (the user common in most Jupyter-based images, with UID=`1000` and GID=`100`), `bob` and `alice` (names commonly used in security-based examples).

We plan on producing one image per release to the enterprise-gateway docker repo where the image's tag reflects the corresponding release. Over time, we may integrate with docker hub to produce this image on every build - where the commit hash becomes the tag.

To build a local image, run `make docker-image-enterprise-gateway`. Because this is a development build, the the tag for this image will be `:dev`.

14.3 elyra/nb2kg

Image elyra/nb2kg is a simple image built on jupyter/minimal-notebook along with the latest release of NB2KG. The image also sets some of the new variables that pertain to enterprise gateway (e.g., `KG_REQUEST_TIMEOUT`, `KG_HTTP_USER`, `KERNEL_USERNAME`, etc.).

To build a local image, run `make docker-image-nb2kg`. Because this is a development build, the tag for this image will be `:dev`.

CHAPTER FIFTEEN

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
  - Kubernetes
  - Platform Conductor
- User Environments
- High Availability
- Batch REST API

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