RStudio Launcher Plugin Software Development Kit 99.9.9

Developer's Guide

2021-12-17

Contents

Chapter 1

Introduction

This document is the RStudio Launcher Plugin SDK Developer's Guide. It covers the architecture of the RStudio Launcher, the architecture of the RStudio Launcher Plugin SDK, and the advanced features of the RStudio Launcher Plugin SDK. To implement a RStudio Launcher Plugin using the provided QuickStart RStudio Launcher Plugin, please refer to the RStudio Launcher Plugin SDK QuickStart guide.

For the remainder of this document, the RStudio Launcher may alternately be referred to as "the Launcher", RStudio Launcher Plugins as "Plugins" (plural) or "Plugin" (singular), and the RStudio Launcher Plugin SDK as "the SDK".

Chapter 2

Prerequisites

2.1 Platform Support

Plugins developed with the SDK require the RStudio Launcher to work. The SDK is currently compatible with the version of the RStudio Launcher that is shipped with RStudio Server Pro v1.4. As such, the SDK will be supported on all of the platforms supported by RStudio Server Pro v1.4, which can be found on the Requirements for RStudio Server Pro page.

2.2 Build Dependencies

To get started developing a Plugin using the SDK, it is necessary to install the development tools and dependencies required for building and running Plugins.

The SDK provides utilities for installing many of the required libraries, however these scripts require super user access and install packages globally, so they should be used with discretion. Before running the `dependencies/install-dependencies.sh` script, review it carefully to ensure that the default installation locations will be acceptable. Otherwise, it is possible to install the dependencies manually.

The SDK requires the following tools and libraries:

Name	Minimum Version	Reference
gcc/g++	4.8	https://gcc.gnu.org/
cmake	3.15.5	https://cmake.org/
Boost	1.70.0	https://www.boost.org/

2.3 RStudio Server Pro

This version of the RStudio Launcher Plugin SDK is compatible with RStudio Server Pro version 1.4, or greater. To properly test the integration between a Launcher Plugin and RStudio Server Pro, an RStudio Server Pro license, with the launcher feature enabled, is required. To acquire a license, please contact sales@rstudio.com.

2.4 Job Scheduling System

Each Plugin is intended to interface with a single job scheduling system. In order to test the Plugin implementation, access to a test environment with the desired job scheduling system will be required.

Chapter 3

SDK Release Notes

3.1 12.17.2021

- The debug logs are now written to `/var/log/rstudio/launcher` by default, instead of the plugin's scratch path.

Chapter 4

The RStudio Launcher

The purpose of the RStudio Launcher is to provide a generic interface to integrate job scheduling systems with RStudio Server Pro and other RStudio products. The Launcher itself is not specific to R processes, and can launch arbitrary work through any job scheduling system for which a Plugin exists. This section will describe the way that RStudio products may integrate with the Launcher, the way the Launcher communicates with and manages Plugins, and network architecture requirements.

4.1 Public Interface

On the front-end, the Launcher exposes an HTTP API. RStudio products which wish to integrate with the Launcher send HTTP requests, such as `GET /jobs` or `POST /jobs` to the Launcher. The Launcher handles authentication and authorization for each of these requests, distills the necessary information for a Plugin, and forwards those details to the appropriate Plugin. As a result, there is no requirement for the Launcher to be running on the same machine as the RStudio product which will make use of it. For more information about network architecture requirements, see section 4.3.

4.2 Plugin Management

When the Launcher is started, it reads `/etc/rstudio/launcher.conf` which contains a `[cluster]` section for each Plugin which should be used.

The Launcher will start a child process for each Plugin and request that the Plugin initialize itself with a `Bootstrap` request. During this request, the Plugin is responsible for ensuring that it has an accurate list of the jobs currently in its respective job scheduling system. If any configured Plugin fails to bootstrap correctly, the Launcher will fail to start.

During the lifetime of the Launcher, it will forward requests to the Plugins as appropriate, via the JSON Launcher Plugin API. In the current version, the Launcher is only able to communicate with Plugins over standard input and output. As a result, the Launcher and each Plugin must be run on the same machine.

When the Launcher is terminated, it will send a termination signal to each of the Plugins it is managing.

Receiving requests, formatting and sending responses, and listening for signals are all handled by the SDK before the Plugin specific implementations are invoked. Thus, the Plugin developer can focus solely on the parts of the implementation that will integrate the Plugin with the job scheduling system.

4.3 Network Architecture Requirements

As mentioned in section 4.1, it is not necessary for the Launcher and the RStudio product which will make use of it to be running on the same machine. The RStudio product which will use the Launcher needs to be able to communicate with the launcher over HTTP or HTTPS on the configured port. The default Launcher port is `5559`, but it can be modified in `launcher.rstudio.conf`. In figure 4.1, a very simple network architecture is shown in which RStudio Server Pro is running on Host A and the Launcher and its Plugins are running on Host B. The Plugins must also be able to reach and communicate with their respective job scheduling systems.

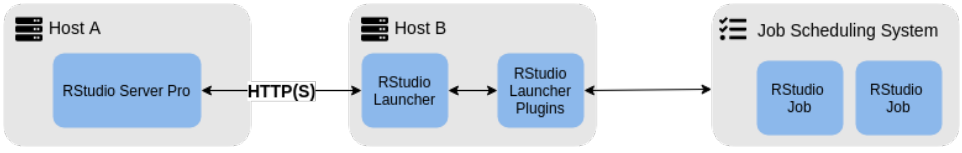

Figure 4.1: Simple Launcher Network Architecture

It is possible to load balance multiple instances of the Launcher for improved

request throughput. An example with two Launcher instances is depicted in figure 4.2, below.

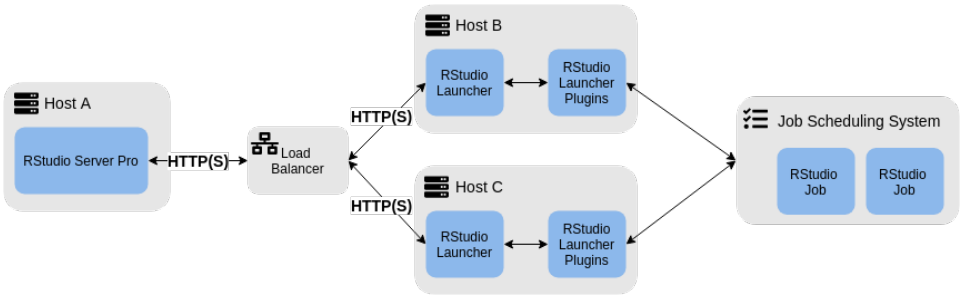

Figure 4.2: Load Balanced Launcher Network Architecture

In order to work correctly with a load balanced architecture, the Plugin implementation must be able to return the same output from any Plugin which is part of the Launcher cluster. As each API call is described in detail in the following sections, requirements for an implementation of that API call which supports load balancing will be discussed.

4.3.1 RStudio Server Pro Integration

When integrating the Launcher with RStudio Server Pro, there are two additional cases which must be considered when configuring the network: using Launcher Sessions, and using Launcher Jobs. Regardless of whether the Launcher is load balanced the network requirements stay the same, so the diagrams in this section will be based on the architecture describbned in figure 4.1, but may be applied to the architecture described in figure 4.2.

When using Launcher Sessions, the RStudio Server Pro process must be able to communicate with the R, Jupyter, or other session over TCP on an arbitrary port.

When using Launcher Jobs without Launcher Sessions, there are no special network considerations. However, when using Launcher Jobs with Launcher Sessions the R sessions must be able to communicate with RStudio Server Pro over HTTP(S), in the same way as an RStudio IDE user would.

For more details about configuring the RStudio Server and the Launcher, see the RStudio Server Pro Administration Guide.

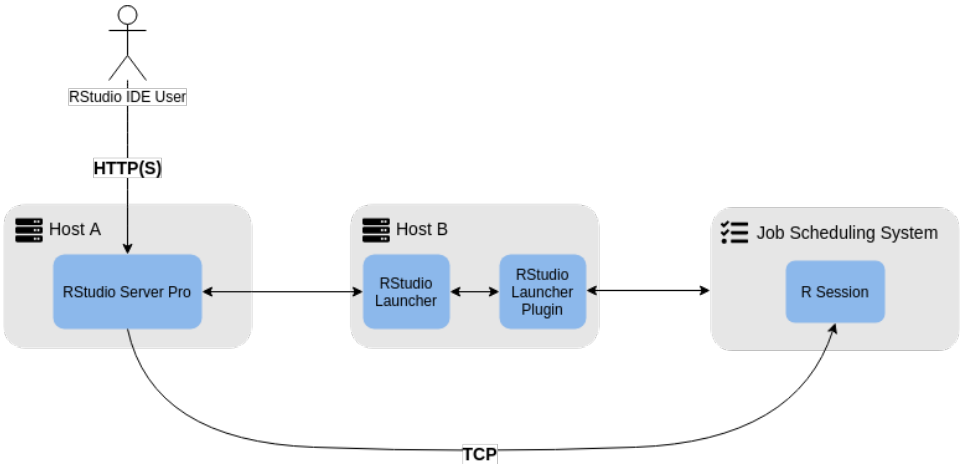

Figure 4.3: Launcher Sessions

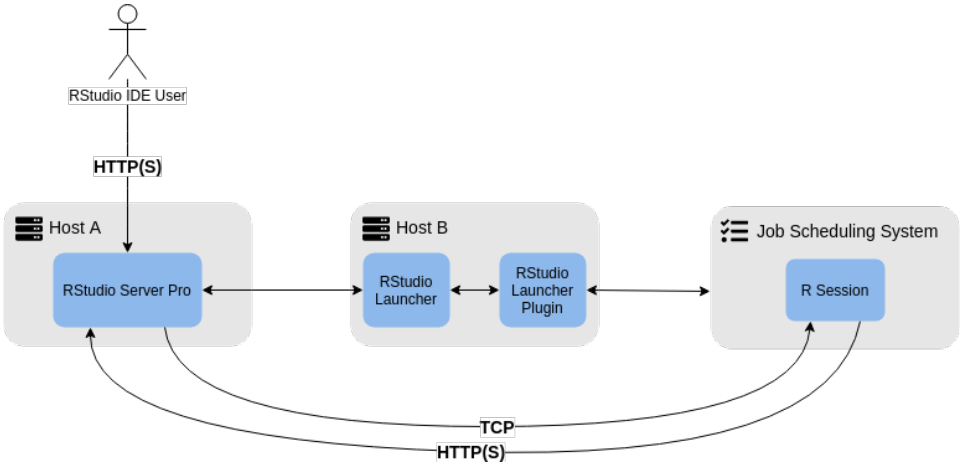

Figure 4.4: Launcher Jobs with Launcher Sessions

Chapter 5

Launcher Plugin API

The Launcher Plugin API is a set of requests and responses which may be sent from the Launcher to the Plugin (requests) or from the Plugin to the Launcher (responses). This section describes the API calls supported by this version of the RStudio Launcher Plugin SDK. Documentation for the latest version the Launcher Plugin API may be found here.

For each API call, this section will go into detail on how the request corresponds to methods on the `IJobSource` interface, or on other components of the SDK.

5.1 Common Fields

The following fields are common to all requests and responses:

Request

Field Name	Description
messageType	The type ID of the message. The ID for a given message type can be found in the section for the given message.
requestId	The monotonically increasing request ID for this request. This should be saved by the plugin for use in responses.

Response

Field Name	Description
messageType	The type ID of the message. The ID for a given message type can be found in the section for the given message.
requestId	The ID of the request that this response is answering. This is used by the Launcher to determine which response belongs to which request.
responseId	The monotonically increasing response ID for this response. The first response ID must be 0.

The SDK manages all of these fields for each request and response, so the Plugin developer does not need to worry about them.

In addition, many requests may also include `username` and `requestUsername` fields, described in the table below. These fields are passed to the appropriate `IJobSource` method as they impact the behaviour of the Plugin.

Field Name	Description
username	The username of the user that initiated the request. Used for auditing/security purposes. Some requests have the ability to specify '*'for this value, indicating all users' information should be returned. This can occur if a super user (admin user) makes a request, or if authorization is disabled.
requestUsername	The actual username used when the request was submitted. This should be ignored in most plugins, and the username value (above) should be used. This field is provided for additional information if needed.

5.2 Requests and Responses

5.2.1 Error

For any request, the Plugin may return an Error response. To return an Error response to the server, the `IJobSource` implementation should return an `Error`. The Plugin API component will generally construct the appropriate Error response based on the `Error` returned by the `IJobSource` call. For the most part, the Plugin developer does not need to be concerned with Error responses.

Error Response

Field Name	Description	Value
messageType	See above.	`Response::Type::ERROR` (-1)
requestId	See above.	Int
responseId	See above.	Int
errorCode	The error code. More details in the Error Codes section.	Int
errorMessage	The error message. More details in the Error Codes section.	Int

5.2.1.1 Error Codes

The code of the Error response must be one of the error codes supported by the Launcher Plugin API. The message can be an arbitrary string which provides more context for the Launcher user.

Code Name	Description	Value
UnknownError	The request failed for an undetermined reason. Used when the Plugin cannot determine an appropriate error code for the error.	0

Code Name	Description	Value
RequestNotSupported	The request is not supported by the Plugin. The Plugin API component may return this if the Launcher sends a request that is not part of the Plugin's supported Launcher Plugin API version.	1
InvalidRequest	The request is malformed. The Plugin API component may return this if it receives an unexpected message from the Launcher.	2
JobNotFound	The job does not exist in the scheduling system. The Plugin should return this if the user specified job ID does not exist.	3
PluignRestarted	The request could not be completed because the Plugin had to restart.	4
Timeout	The request timed out while waiting for a response from the job scheduling system.	5
JobNotRunning	The job exists in the job scheduling system but is not in the running state.	6
JobOutputNotFound	The job does not have output.	7
InvalidJobState	The job has an invalid job state for the requested action.	8
JobControlFailure	The job control action failed.	9
UnsupportedVersion	The Launcher is using a Launcher Plugin API version that is not supported by the Plugin.	10

5.2.2 Heartbeat

When the Launcher starts the Plugin, it provides the configured `heartbeat-interval-seconds` value to the Plugin. If the Plugin does not send the Heartbeat Response every configured amount of time, the Launcher will restart the Plugin. This request is completely handled by the SDK and doesn't require any action on the part of the Plugin developer. However, when debugging issues in the Plugin, it can become problematic if the Launcher restarts the Plugin while getting the debugger set up. To avoid this, `heartbeat-interval-seconds` can be set to `0` in `/etc/rstudio/launcher.conf`, which will disable heartbeating. This is not a recommended configuration for production because unresponsive Plugins will not be handled correctly.

Despite the fact that there will be many heartbeats sent to and from the

Launcher, every Heartbeat Request and Response will have request ID and response IDs of 0. This is because it is Heartbeat Response do not strictly respond to a particular request – they must merely be sent to the Launcher every configured time interval.

Heartbeat Request

Field Name	Description	Value
messageType	See above.	`Request::Type::HEARTBEAT` (0)
requestId	See above.	0

Heartbeat Response

Field Name	Description	Value
messageType	See above.	`Response::Type::HEARTBEAT` (0)
requestId	See above.	0
responseId	See above.	0

5.2.3 Bootstrap

This request is sent to the Plugin once, immediately after the Plugin is launched. During this request, the SDK will verify version compatibility with the Launcher and then trigger the population of existing jobs by invoking `IJobSource::getJobs`.

Bootstrap Request

Field Name	Description	Value
messageType	See above.	`Request::Type::BOOTST`
requestId	See above.	0
version	The Launcher Plugin API version in use by the Launcher.	Version

Bootstrap Response

Field Name	Description	Value
messageType	See above.	`Response::Type::BOOTSTRA`
requestId	See above.	0
responseId	See above.	0
version	The Launcher Plugin API version in use by the Plugin.	Version

5.2.4 Jobs

There are two API requests which return a Job State Response: the Submit Job and Job State requests. The SDK maintains an internal cache of all jobs in the system. When a Job State request is received, the SDK searches the cache for any jobs that match the request and sends them back to the Launcher. The SDK will also manage filtering the jobs and restricting the job fields in the response, if necessary. For more information about how the job cache is maintained, see the Customizing the Job Repository section of the Advanced Features chapter.

Submit Job Request

Field Name	Description	Value
messageType	See above.	`Request::Type::SUBMIT`
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String
job	The job object which describes the job to be launched.	Job

Job State Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_JOB` (3)

Field Name	Description	Value
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String
jobId	The ID of the job to be retrieved. If this field is '*' all jobs should be retrieved and filters may be used.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String
tags	If present, the set of tags by which to filter the jobs. Only jobs which have all these tags will be returned.	String Array
startTime	If present, only jobs which were submitted after this UTC time will be returned. Format: `YYYY-MM-DDThh:mm:ss`.	String
endTime	If present, only jobs which were submitted before this UTC time will be returned. Format: `YYYY-MM-DDThh:mm:ss`.	String
statuses	If present, only jobs which have one of the specified statuses will be returned.	JobState Array
fields	If present, only the job fields included in this list will be included in the response, excepted 'id', which will always be returned.	String Array

Job State Response

Field Name	Description	Value
messageType	See above.	`Response::Type::JOB_STATE` (2)
jobs	The list of jobs that met the request criteria.	Job Array

5.2.5 Job Status Stream

This API call is responsible for streaming Job status updates as they occur. The SDK maintains an internal cache of all the Jobs in the Job Scheduling System and uses the `JobStatusNotifier` to allow components of the SDK to be notified when a Job's status is updated. Components which wish to be notified of a Job status update may subscribe to the notifier, and components

which wish to notify other components about new updates should post the updates to the `JobStatusNotifier`.

The SDK provides `AbstractJobStatusWatcher` and `AbstractTimedJobStatusWatcher` to help facilitate the implementation of keeping Job statuses updated within the Plugin. For more details about implementing Job status updates, see the Job Status Updates of the Advanced Features chapter, or TODOs #'s 9 - 11 of the RStudio Launcher Plugin SDK QuickStart Guide.

The SDK will manage all Job status streams, including closing them on request. The Plugin developer is only responsible for posting correct Job status updates to the `JobStatusNotifier` either directly or indirectly.

Rather than sending a response per open stream, the Plugin sends the Launcher a response with a field that indicates which stream requests are answered by this response. A sequence ID is kept per request to ensure the Launcher can keep the responses in the right order.

Job Status Stream Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_JOB_STA` (4)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String
jobId	The ID of the job to be for which to stream statuses. If this field is '*' the statuses of all jobs will be streamed.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String
cancel	Whether the stream should be started (`false`) or canceled (`true`). Default: `false`.	Boolean

Job Status Stream Response

Field Name	Description	Value
messageType	See above.	`Response::Type::JOB_STATUS` (3)
sequences	An array of requests which are waiting for this stream response and the sequence ID of this response, relative to each request ID.	StreamSequence Array
id	The ID of the job.	String
name	The name of the job.	String
status	The status of the job.	JobState
statusMessage	The status message of the job, if any.	String

5.2.6 Control Job

This API call can be used to control the state of a job. Currently supported operations are cancel, stop, kill, suspend, and resume. The Job must be in the correct state for a given operation to be valid (respectively, pending, running, running, running, and suspended). The SDK will handle Job State validation. The Plugin is only responsible for a best-effort attempt to perform the given operation on the job. If the Plugin indicates that the operation is not supported, the SDK will return an Error Response.

Control Job Request

Field Name	Description	Value
messageType	See above.	`Request::Type::CONTROL_JOB` (5)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String
jobId	The ID of the job to control. This field cannot be '*'.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String
operation	The operation to be performed on the Job.	JobOperation

Control Job Response

Field Name	Description	Value
messageType	See above.	`Response::Type::CONTROL_JOB` (4)
requestId	See above.	Int
responseId	See above.	Int
statusMessage	A message describing the status of the control job operation, if any.	String
operationCompleted	Whether the control job operation completed successfully or not.	Boolean

5.2.7 Job Output Stream

This API call is responsible for streaming a Job's output data. The caller may request the type of output they wish to see (Standard Out, Standard Error, or both) and the Plugin must do its best to report only that type of data.

The SDK will manage all the output streams. The Plugin developer needs to implement `IJobSource::createOutputStream` and may either use the provided `FileOutputStream` class or implement a custom concrete `AbstractOutputStream`. An example using the `FileOutputStream` can be found in 'TODO #8' of the 'RStudio Launcher Plugin SDK QuickStart Guide'. For more details on implementing a custom `AbstractOutputStream` class, see the Custom Output Streams section.

Job Output Stream Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_JOB_OUT` (6)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String

Field Name	Description	Value
jobId	The ID of the job to be for which to stream output.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String
outputType	The type of output to be streamed. Standard out (0), Standard error (1), or both (2).	Int: 0, 1, or 2
cancel	Whether the stream should be started (`false`) or canceled (`true`). Default: `false`.	Boolean

Job Output Stream Response

Field Name	Description	Value
messageType	See above.	`Response::Type::JOB_OUTPUT` (5)
requestId	See above.	Int
responseId	See above.	Int
seqId	The ID of this response in the stream of output responses.	Int
outputType	The type of output to in the response, if any. Standard out, Standard error, or mixed.	String: "stdout", "stderr", or "mixed"
output	The job output, if any.	String
complete	Whether the output stream is complete.	Boolean

5.2.8 Job Resource Utilization Stream

This API call is responsible for streaming a Job's resource utilization metrics. Resource utilization metrics may only be streamed when the

Job is in the `Running` state. If the Job is not running, a `JobNotRunning` error will be returned by the SDK. Otherwise, the SDK will invoke `AbstactResourceStream::initialize` and the Plugin should commence streaming resource utilization metrics.

The SDK will manage all the resource utilization streams. The Plugin developer needs to implement `AbstactResourceStream::initialize` and may either extend `AbstractResourceStream` directly, or extend `AbstractTimedResourceStream` if resource utilization data must be polled. An example of extending `AbstractResourceStream` directly can be found in 'TODO #16' of the 'RStudio Launcher Plugin SDK QuickStart Guide'. The `LocalResourceStream` provided with the sample Local Plugin provides an example of extending `AbstractTimedResourceStream`.

Job Resource Utilization Stream Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_JOB_RES` (7)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String
jobId	The ID of the job for which to stream resource utilization metrics.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String
cancel	Whether the stream should be started (`false`) or canceled (`true`). Default: `false`.	Boolean

Job Resource Utilization Stream Response

Field Name	Description	Value
messageType	See above.	`Response::Type::JOB_RESOURCE_` (6)

Field Name	Description	Value
requestId	See above.	Int
responseId	See above.	Int
seqId	The ID of this response in the stream of resource utilization responses.	Int
cpuPercent	The percentage of CPU utilization of the Job, if available. Optional.	String
cpuSeconds	The total run time of the Job, in seconds, if available. Optional.	String
virtualMemory	The current size of virtual memory in use by the Job, in MB, if available. Optional.	String
residentMemory	The current size of the resident set in use by the Job, in MB, if available. Optional.	String
complete	Whether the resource utilization stream is complete.	Boolean

5.2.9 Job Network

This API call is responsible for providing network information about the machine running the specified job to the caller. When this API call is received, the SDK will invoke `IJobSource::getNetworkInfo`. The Plugin is responsible for resolving the hostname and the IP addresses of the machine running the job.

For an example of how to implement `IJobSource::getNetworkInfo`, see 'TODO #14' of the 'RStudio Launcher Plugin SDK QuickStart Guide'.

Job Network Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_JOB_NET` (8)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String

Field Name	Description	Value
jobId	The ID of the job to be for which to retreive network information.	String
encodedJobId	The Launcher generated encoded job ID which contains extra metadata. Provides extra information only.	String

Job Network Response

Field Name	Description	Value
messageType	See above.	`Response::Type::GET_JOB_NETWO` (7)
requestId	See above.	Int
responseId	See above.	Int
host	The hostname of the machine running the job.	String
ipAddresses	An array of the non-local IP addresses for the machine running the job.	String Array

5.2.10 Cluster Info

This API call is responsible for reporting the configuration and capabilities of the job scheduling system. RStudio applications use this call to determine what UI to surface to the end user when they launch jobs. When this request is received, the SDK will invoke the following method to compose the correct response:

- `IJobSource::getConfiguration`

The Plugin may return an error from this method if it encounters any issues populating the fields on the `JobSourceConfiguration` object. Additionally, the method is provided with a `system::User` object, which represents the user who initiated the Cluster Info request. This optionally allows the Plugin Developer to provide different results on a per-user or per-group basis. This may be useful to allow system administrator to set different maximum

and default Resource Limits for different groups of user via the User Profiles feature. It also may be useful if the job scheduling system allows administrators to make similar rule sets.

For a simple example of implementing this method, see 'TODO #7' in the 'RStudio Launcher Plugin SDK QuickStart Guide'.

Cluster Info Request

Field Name	Description	Value
messageType	See above.	`Request::Type::GET_CLUSTER_INFO` (9)
requestId	See above.	Int
username	See above.	String
requestUsername	See above.	String

Cluster Info Response

Field Name	Description	Value
messageType	See above.	`Response::Type::CLUSTER_INFO` (8)
requestId	See above.	Int
responseId	See above.	Int
supportsContainers	Whether or not the job scheduling system supports containers.	Boolean
config	Any custom configuration values which may be set per Job.	JobConfig Array
placementConstraints	Any custom placement constraints which may be requested for a Job.	PlacementConstraint Array
queues	The queues which are available to run Jobs, if the job scheduling system supports queues.	String Array
resourceLimits	The types of resource limits which may be requested for a Job, including default and maximum values, if any.	ResourceLimit Array

Field Name	Description	Value
images	If the job scheduling system supports containers, the list of container images which may be used when launching a job.	String Array
defaultImage	If the job scheduling system supports containers, the default container image to use if none is selected.	String
allowUnknownImages	If the job scheduling system supports containers, whether to allow Jobs to be run on images which are not known.	Boolean

5.3 Complex API Types

This section describes the details of API types which are not simple types, such as String, Int, or Boolean.

5.3.1 Container

Field Name	Description	Value
image	The name of the container image to use.	String
runAsUserId	The ID of the user to run the container as. Optional.	Int
runAsGroupId	The ID of the group to run the container as. Optional.	Int
supplementalGroupIds	The list of additional group IDs to be added to the run-as user in the container. Optional.	Int Array

5.3.2 Environment

Field Name	Description	Value
name	The name of the environment variable.	String

Field Name	Description	Value
value	The value of the environment variable.	String

5.3.3 ExposedPort

Field Name	Description	Value
targetPort	The target port, within the container.	Int
publishedPort	The published port, if different from the container port.	Int
protocol	The network protocol to use. Default: TCP.	String

5.3.4 Job

Field Name	Description	Value
args	The arguments of the 'command' or 'exe' of the Job.	String Array
cluster	The cluster of the Job.	String
command	The shell command of the Job. Mutually exclusive with 'exe'.	String
config	The custom configuration values of the Job.	JobConfig Array
container	The container configuration of the Job, if the Cluster supports containers.	Container Array
environment	The environment variables for the Job.	Environment Array
exe	The executable of the Job. Mutually exclusive with 'command'.	String
exitCode	The exit code of the 'command' or 'exe' of the Job.	Int
exposedPorts	The exposed ports of the Job, if containers are used.	ExposedPort Array
host	The host on which the Job was (or is being) run.	String

Field Name	Description	Value
id	The unique ID of the Job.	String
lastUpdateTime	The time of the last update to the Job.	String (ISO 8601 format DateTime)
mounts	The file system mounts to apply when the Job is run.	Mount Array
name	The name of the Job.	String
pid	The process ID of the Job, if applicable.	Int
id	The unique ID of the Job.	String
placementConstraints	The list of placement constraints that were selected for the Job.	PlacementConstraint Array
queues	The list of queues that may be used to launch the Job, or the queue that was used to run the Job.	String Array
stdin	The standard input to be passed to the 'command' or 'exe' of the Job.	String
stderr	The location of the file which contains the standard error output of the Job.	String
stdout	The location of the file which contains the standard output of the Job.	String
status	The current status of the Job.	JobState
statusMessage	The message or reason of the current status of the Job.	String
submissionTime	The time at which the Job was submitted to the Cluster.	String (ISO 8601 format DateTime)
tags	The tags that were set for the Job. Used for filtering Jobs.	String Array
user	The username of the user who launched the Job.	String
workingDirectory	The directory to use as the working directory for the 'command' or 'exe'.	String

5.3.5 JobConfig

Field Name	Description	Value
name	The name of the custom configuration value.	String
valueType	The type of the custom configuration value. Optional.	“string”, “int”, “float”, or “enum”
value	The value of the custom configuration value. Optional.	String (convertible to valueType)

5.3.6 JobOperation

This is a JSON Integer value with a limited set of values. The possible values of a JobOperation field and their meanings are listed in the table below.

Value	Description
0	The Job should be suspended. Only valid if the Job is currently in the “Running” JobState.
1	The Job should be resumed. Only valid if the Job is currently in the “Suspended” JobState.
2	The Job should be stopped. Only valid if the Job is currently in the “Running” JobState.
3	The Job should be killed. Only valid if the Job is currently in the “Running” JobState.
4	The Job should be canceled. Only valid if the Job is currently in the “Pending” JobState.

5.3.7 JobState

This type is a JSON String value with limited set of valid values. The possible values of a JobState field and their meanings are listed in the table below.

Value	Description
Canceled	The job was canceled by the user before it began to run.

Value	Description
Failed	The job could not be launched due to an error. This status does not refer to jobs where the process exited with a non-zero exit code.
Finished	The job was launched and finished executing. This includes jobs where the process exited with a non-zero exit code.
Killed	The job was forcibly killed while it was running, i.e. the job process received `SIGKILL`.
Pending	The job was successfully submitted to the job scheduling system but has not started running yet.
Running	The job is currently running.
Suspended	The job was running, but execution was paused and may be resumed at a later time.

5.3.8 Mount

Field Name	Description	Value
mountPath	The destination path of the mount.	String
readOnly	Whether the source path should be mounted with write permissions (`false`) or not (`true`). Default: `false`.	Boolean
type	The type of mount. The default supported options are 'azureFile', 'cephFs', 'glusterFs', 'host', and 'nfs'. The 'passthrough' value or a custom value may be used for other mount types.	String
source	The mount source description. Must match the specified mount type.	MountSource

5.3.9 MountSource

This object will have a different schema depending on the type of mount specified in the 'type' field of the Mount object. Below, the source object for each of the default supported mount types will be described. If the mount type is 'passthrough' or a custom value, the object definition is determined by the Plugin.

5.3.9.1 AzureFileMountSource

This represents an Azure File source. If the 'type' field in the Mount object is 'azureFile' and the 'source' object follows this schema, the Plugin should attempt to mount the requested share from the Azure File account specified in the secret.

Field Name	Description	Value
secretName	The name of the secret that contains both the Azure storage account name and the key.	String
shareName	The share name to be used.	String

5.3.9.2 CephFsMountSource

This represents a Ceph File System source. If the 'type' field in the Mount object is 'cephFs' and the 'source' object follows this schema, the Plugin should attempt to mount the requested path as the specified user from the list of Ceph monitor addresses.

Field Name	Description	Value
monitors	A comma-separated list of Ceph monitor addresses. For example: `192.168.1.200:8765,192.168.1.200:8766`	String Array
path	The path within the Ceph filesystem to mount.	String
user	The Ceph username to use.	String
secretFile	The file which contains the Ceph keyring for authentication.	String
secretRef	Reference to Ceph authentication secrets, which overrides SecretFile if specified.	String

5.3.9.3 GlusterFsMountSource

This represents a Glusterfs mount source. If the 'type' field in the Mount object is 'glusterFs' and the 'source' object follows this schema, the Plugin should attempt to mount the requested path from the specified Glusterfs endpoints.

Field Name	Description	Value
endpoints	The name of the endpoints object that represents a Gluster cluster configuration.	String
path	The name of the GlusterFs volume.	String

5.3.9.4 HostMountSource

This represents a host mount source. If the 'type' field in the Mount object is 'host' and the 'source' object follows this schema, the Plugin should attempt to mount the requested path from the Launcher host, or reject the request on the basis that mounting host paths is not supported.

Field Name	Description	Value
path	The source path to be mounted.	String

5.3.9.5 NfsMountSource

This represents an NFS server mount source. If the 'type' field in the Mount object is 'nfs' and the 'source' object follows this schema, the Plugin should attempt to mount the requested path from the specified NFS server, or reject the request on the basis that mounting NFS paths is not supported.

Field Name	Description	Value
host	The host of the NFS server.	String
path	The source path of the mount, on the NFS server.	String

5.3.10 PlacementConstraint

Field Name	Description	Value
name	The name of the placement constraint.	String
value	One of the possible values of the placement constraint. Optional.	String

5.3.11 ResourceLimit

Field Name	Description	Value
limitType	The type of the resource.	“cpuCount”, “cpuTime”, “memory”, or “memorySwap”
defaultValue	The default value of the resource type.	String
maxValue	The maximum value of the resource type.	String
value	The requested value of the resource.	String

5.3.12 StreamSequence

Field Name	Description	Value
requestId	The ID of the request for which this streamed response is being sent.	Int
sequenceId	The ordered ID of this stream response in the sequence of stream response for the given ‘requestId’.	Int

5.3.13 Version

Field Name	Description	Value
major	The major component of the supported version.	Int
minor	The minor component of the supported version.	Int
patch	The patch component of the supported version.	Int

Chapter 6

Advanced Features

This section describes optional, advanced features of the SDK which are not covered in the RStudio Launcher Plugin SDK QuickStart Guide.

6.1 Error Handling

The RStudio Launcher Plugin SDK QuickStart Guide creates a new Error with an arbitrary name and code at each location that an error would occur. It is advisable to take a more systematic approach to error handling. The specific implementation is completely at the discretion of the Plugin developer, however this section will discuss one possible organizational strategy.

For each category of error that may occur, create an enum which represents the specific types of errors in that category, and a function (or a few functions) which create `Error` objects with the correct category name.

Error codes must not begin at 0 as that would be considered a 'Success' error code (i.e. not an error). Error codes may be reused across categories.

6.1.1 Example

Suppose that the Orchid Organization's developer determines that there are only two category of error that cannot be covered by a system or unknown error (available in `Error.hpp`): Mars API errors, and internal errors.

The developer might create the file `MarsError.hpp` as follows:

```
#ifndef ORCHID_MARS_ERROR_HPP_
#define ORCHID_MARS_ERROR_HPP_

namespace rstudio {
namespace launcher_plugins {

class Error;

} // namespace launcher_plugins
} // namespace rstudio


namespace orchid {
namespace mars {

enum class InternalError
{
  SUCCESS = 0,
  UNKNOWN = 1,
  CONVERSION_FAILURE = 2,
  JOB_NOT_FOUND = 3
};

/** Enum which represents a Mars API Error. */
enum class MarsError
{
  SUCCESS = 0,
  UNKNOWN = 1,
  CONN_TIMEOUT = 2,
  NOT_AUTHORIZED = 3,
  UNSUPPORTED_VERSION = 4
};

Error createMarsError(
    const mars_api::mars_exception& in_exception,
    const ErrorLocation& in_location);

Error createMarsError(
    const mars_api::mars_exception& in_exception,
```

```
    const Error& in_cause,
    const ErrorLocation& in_location);

Error createVersionError(
    const std::string& in_supportedVersion,
    const std::string& in_actualVersion,
    const ErrorLocation& in_location);

Error createInternalError(
    InternalError in_code,
    const std::string& in_message,
    const ErrorLocation& in_location);

Error createInternalError(
    InternalError in_code,
    const std::string& in_message,
    const Error& in_cause,
    const ErrorLocation& in_location);

} // namespace mars
} // namespace orchid

#endif
```

Then `MarsError.cpp` might look like this:

```
#include "MarsError.hpp"

#include <Error.hpp>

namespace orchid {
namespace mars {

namespace {

constexpr const char* s_internalErrorName = "InternalPluginError";
constexpr const char* s_marsErrorName = "MarsApiError";

} // anonymous namespace
```

```
Error createMarsError(
    const mars_api::mars_exception& in_exception,
    const ErrorLocation& in_location)
{
  return createMarsError(in_exeption, Success(), in_location);
}

Error createMarsError(
    const mars_api::mars_exception& in_exception,
    const Error& in_cause,
    const ErrorLocation& in_location)
{
  MarsError code = MarsError::UNKNOWN;
  if (in_exception.code() == 14) // Connection timeout
    code = MarsError::CONN_TIMEOUT;
  else if (in_exception.code() == 52) // Not authorized
    code = MarsError::NOT_AUTHORIZED;

  if (in_cause == Success())
    return Error(
      s_marsErrorName,
      static_cast<int>(code),
      in_exception.what(),
      in_location);

  return Error(
    s_marsErrorName,
    static_cast<int>(code),
    in_exception.what(),
    in_cause,
    in_location);
}

Error createVersionError(
    const std::string& in_supportedVersion,
    const std::string& in_actualVersion,
    const ErrorLocation& in_location)
{
  return Error(
```

```
    s_marsErrorName,
    static_cast<int>(MarsError::UNSUPPORTED_VERSION),
    "Mars API is version \"" +
      in_actualVersion +
      "\" which is not supported. Supported version(s): "
      + in_supportedVersion,
    in_location);
}

Error createInternalError(
    InternalError in_code,
    const std::string& in_message,
    const ErrorLocation& in_location)
{
  return Error(
    s_internalErrorName,
    static_cast<int>(in_code),
    in_message,
    in_location);
}

Error createInternalError(
    InternalError in_code,
    const std::string& in_message,
    const Error& in_cause,
    const ErrorLocation& in_location)
{
  return Error(
    s_internalErrorName,
    static_cast<int>(in_code),
    in_message,
    in_cause,
    in_location);
}

} // namespace mars
} // namespace orchid
```

As an example of how these functions might be used, consider the exam-

ple in TODO #10 of the RStudio Launcher Plugin SDK QuickStart Guide. With this new error handling, the Plugin developer may change the implementation to the following:

```
Error MarsJobStatusWatcher::getJobDetails(const std::string& in_jobId, api::JobPtr& o
{
  const options::MarsOptions& opts = options::MarsOptions::getInstance();
  mars_api::job marsJob;
  try
  {
    unsigned long id = std::strtoul(in_jobId);
    marsJob = mars_api::list_job(opts.getMarsServiceUser(), id);
  }
  catch (const std::invalid_argument& e)
  {
    return createInternalError(
      InternalError::CONVERSION_FAILURE,
      "Job ID " +
        in_jobID +
        " could not be converted to unsigned long: " +
        e.what(),
      ERROR_LOCATION);
  }
  catch (const std::out_of_range& e)
  {
    return createInternalError(
      InternalError::CONVERSION_FAILURE,
      "Job ID " +
        in_jobID +
        " is out of range to be converted to unsigned long: " +
        e.what(),
      ERROR_LOCATION);
  }
  catch (const mars_api::mars_exception& e)
  {
    return createMarsError(e, ERROR_LOCATION);
  }

  // This should only be invoked for Launcher Jobs because of the filtering in pollJo
  // so return an error if somehow it's not a Launcher Job.
```

```
  if (job._name.find("[RStudio Launcher]") == std::string::npos)
    return createInternalError(
      InternalError::JOB_NOT_FOUND,
      "Job " + in_jobId + " is not an RStudio Launcher Job.",
      ERROR_LOCATION);

  out_job = marsJobToJob(marsJob);
  return Success();
}
```

With this approach, the Plugin developer has created much more consistent and informative errors.

6.2 Date-Time Support

The SDK provides a utility class for working with `DateTime` objects (`include/system/DateTime.hpp`). Dates and times can be converted from string to `system::DateTime` using `system::DateTime::fromString`, and to string using `system::DateTime::toString`. When converting from string, the default expected time string format will be `YYYY-MM-DDThh:mm:ss.ssssssTZ`, or `%Y-%m-%dT%H:%M:%S%F%ZP` using the supported format specification. This is the ISO 8601 extended time format. When converting to string, the default output format is `YYYY-MM-DDThh:mm:ss.ssssssZ`, or `%Y-%m-%dT%H:%M:%S%FZ`. Note that the default output string is in UTC time.

To use a non-default format specification, a custom format specification may be provided to the conversion function. Below is a table which describes the possible format values. As with most string formats, characters which are not prefixed by `%` will be included in the output (or should be included in the input) verbatim.

Value	Description	Example
%a	Short weekday	“Mon”, “Tue”
%A	Long weekday	“Monday”, “Tuesday”
%b	Short month	“Nov”, “Dec”
%B	Long month	“November”, “December

Value	Description	Example
%d	Numerical day of the month	“01” through “31”, as appropriate for the month
%f	Fractional seconds	“04:01:33.000000”, “04:52:16.598763”
%F	Fractional seconds, if non-zero	“04:01:33”, “04:52:16.598763”
%H	Hour value, on the 24 hour clock	“00” through “23”
%j	Numerical day of the year	“001” through “365” (or “366” on leap years)
%m	Numerical month	“01” through “12”
%M	Minute value	“00” through “59”
%s	Seconds with fractional seconds	“28.003251”
%S	Seconds	“28”
%U	Numerical week of the year, starting on a Sunday	“00” through “53”, where the first Sunday in January is the first day of week 01
%w	Numerical day of the week, starting from 0	“0” through “6”
%W	Numerical week of the year, starting on a Monday	“00” through “53”, where the first Monday in January is the first day of week 01
%y	Two digit year	2019 would be “19”
%Y	Four digit year	“2019”
%ZP	Posix time zone string	“-07:00”, “PST-08PDT+01,M4.1.0/02:00,M10.5.0/02:00”, “Z”

More advanced formatting flags and additional documentation regarding the parsing and formatting of `DateTime` objects can be found in Boost’s Date Time I/O documentation.

6.3 User Profiles

It may be useful to allow system administrators to set default or maximum values for certain features on a per-user or per-group basis. For ex-

ample, if a job scheduling system supports requesting an amount of memory for a job, system administrators may wish to give different memory levels to different groups of users. For more examples, see the sample `/etc/rstudio/launcher.kubernetes.profiles.conf` in the Job Launcher Plugin Configuration section of the RStudio Job Launcher Guide.

For the convenience of the Plugin Developer, the `AbstractUserProfiles` class may be overridden to quickly implement support for user profiles.

`AbstractUserProfiles` contains protected templated functions for getting a value by name. The templates are defined in the CPP file, and are declared for the following types:

- `std::string`
- `uint32_t`
- `int32_t`
- `uint64_t`
- `int64_t`
- `double`
- `bool`
- `std::set<U>`, where `U` is one of the above types.
- `std::vector<U>`, where `U` is one of the above types, except `std::set`.
- `std::map<U, V>`, where `U` and `V` are any pair of the above types.

If a custom type is needed, retrieve the value as a string and then parse it as needed. For an example, see the `TestUserProfiles` class in `sdk/src/options/tests/UserProfilesTests.cpp`.

The minimum requirements to implement `AbstractUserProfiles` are:

- A public constructor which sets the plugin name via the `AbstractUserProfiles(const std::string& in_pluginName)` constructor. Alternately, a private constructor with a public static `getInstance` method may be used to implement the singleton pattern. This will prevent the need to read the configuration file multiple times.
- An implementation of `AbstractUserProfiles::getValidFieldNames` which returns a set of all supported values that may be set via the user profiles configuration file.
- An implementation of `AbstractUserProfiles::validateValues` which calls one of the two protected `AbstractUserProfiles::validateValue` methods for each valid field, with the appropriate template parameter.

If the above criteria are met, the expected location of the user profiles configuration file will be `/etc/rstudio/launcher.<plugin name>.profiles.conf`.

The `AbstractUserProfiles::validateValues` method is called by `AbstractUserProfiles::initialize` after the user profiles configuration file has been read and parsed to ensure that any configuration mistakes within the file will be caught early. The `AbstractUserProfiles::initialize` method should be called from the `IJobSource::initialize` method to ensure that the user profiles configuration file has been read into memory and parsed before the Plugin enters normal operation mode. If the user profiles `initialize` method returns an error, the `IJobSource::initialize` method should also return an error.

6.3.1 Example

This example continues the examples started in the RStudio Launcher Plugin SDK QuickStart Guide. Assume that the Mars job scheduling system supports requesting a CPU count and an amount of memory, in MB. For simplicity, this examples implements the `MarsUserProfiles` class completely within the `hpp` file.

MarsUserProfiles.hpp

```
#include <options/AbstractUserProfiles.hpp>

#include <Error.hpp>
#include <system/User.hpp>

namespace orchid {
namespace mars {
namespace options {

using namespace rstudio::launcher_plugins;

class MarsUserProfiles : public options::AbstractUserProfiles
{
  public:
    static MarsUserProfiles& getInstance()
    {
```

```
    static MarsUserProfiles userProfiles;
    return userProfiles;
  }

  uint32_t getDefaultCpus(const system::User& in_user) const
  {
    // Default value is 1.
    uint32_t defaultCpus = 1;
    Error error = getValueForUser("default-cpus", in_user, defaultCpus);

    // It shouldn't be possible to get any Error except a not-found error here beca
    // validateValues. If it somehow occurred in release, just return the default v
    assert(!error || isValueNotFoundError(error));

    return defaultCpus;
  }

  uint32_t getMaxCpus(const system::User& in_user) const
  {
    // Default value is 1.
    uint32_t maxCpus = 1;
    Error error = getValueForUser("max-cpus", in_user, maxCpus);

    // It shouldn't be possible to get any Error except a not-found error here beca
    // validateValues. If it somehow occurred in release, just return the default v
    assert(!error || isValueNotFoundError(error));

    return maxCpus;
  }

  uint32_t getMaxMemory(const system::User& in_user) const
  {
    // Default value is 10 MB.
    uint32_t maxMemory = 10;
    Error error = getValueForUser("max-memory-mb", in_user, maxMemory);

    // It shouldn't be possible to get any Error except a not-found error here beca
    // validateValues. If it somehow occurred in release, just return the default v
    assert(!error || isValueNotFoundError(error));
```

```
    return maxMemory;
  }

  uint32_t getDefaultCpus(const system::User& in_user) const
  {
    // Default value is 5 MB.
    uint32_t defaultMemory = 5;
    Error error = getValueForUser("default-memory-mb", in_user, defaultMemory);

    // It shouldn't be possible to get any Error except a not-found error here beca
    // validateValues. If it somehow occurred in release, just return the default v
    assert(!error || isValueNotFoundError(error));

    return defaultMemory;
  }

private:
  MarsUserProfiles() :
    AbstractUserProfiles("mars")
  {
    m_validFieldNames.insert("max-cpus");
    m_validFieldNames.insert("default-cpus");
    m_validFieldNames.insert("max-mem-mb");
    m_validFieldNames.insert("default-mem-mb");
  }

  const std::set<std::string>& getValidFieldNames() const override
  {
    return m_validFieldNames;
  }

  Error validateValues() const override
  {
    // For supported types, validateValue will attempt to parse every occurrence of
    // specified type. If a custom type is desired, use
    // AbstractValidateValue::validateValue(
    //    const std::string& in_value,
    //    const CustomValueValidator& in_validator) const;
    // method to supply a custom validator instead. in_validator should parse the v
```

```
      // supplied and return an error if parsing fails.
      Error error = validateValue<uint32_t>("default-cpus");
      if (error)
        return error;

      error = validateValue<uint32_t>("max-cpus");
      if (error)
        return error;

      error = validateValue<uint32_t>("default-memory-mb");
      if (error)
        return error;

      return validateValue<uint32_t>("max-cpus");
    }

    std::set<std::string> m_validFieldNames;
}

} // namespace options
} // namespace mars
} // namespace orchid
```

MarsJobSource.cpp

```
// Other includes...

#include <options/MarsOptions.hpp>
#include <options/MarsUserProfiles.hpp>

// Other methods...

Error MarsJobSource::initialize()
{
  const options::MarsOptions& opts = options::MarsOptions::getInstance();

  try
  {
    mars_api::init(opts.host(), opts.port(), opts.useSsl());
```

```
  }
  catch (const mars_expcetion& e)
  {
    return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
  }

  const options::MarsUserProfiles& userProfiles = options::MarsUserProfiles::getInsta
  return userProfiles.initialize();
}

// Other methods..
```

6.4 Custom Job Source Configuration

> Important: This feature is not exposed through the RStudio Server Pro job launching UI. The use of this feature will require a feature request to the RStudio IDE project. This feature should only be used when there are no other alternatives.

The Cluster Info Response is used to report the configuration and capabilities of the Plugin. The RStudio Launcher Plugin SDK QuickStart Guide describes how the Plugin Developer may declare support for various types of resource limits, containers, custom job placement constraints, and job queues. In the event that there is some job configuration setting that is not covered by one of those built-in job settings, the Job Config feature may used.

A `JobConfig` value consists of the name of the configuration setting, its type, and optionally its value. It may have one of four types: `string`, `int`, `enum`, or `float`.

To declare support for a custom job configuration value, create a `JobConfig` object that represents the name and type of that value and add it to the `JobSourceConfiguration::CustomConfig` vector in the overridden implementation of `IJobSource::getConfiguration`.

When a job is submitted, any custom configuration values that were set on the job can be found on `Job::Config`.

6.5 Job Status Updates

The Plugin needs to keep an accurate record of all of the Jobs that were submitted to the Job Scheduling System by the Launcher. This can be implemented in any way that suits the Job Scheduling System as long as `JobStatusNotifier::updateJobStatus` is invoked each time the status of a job changes. The `JobStatusNotifier::updateJobStatus` validates that the current status really is an update, so there is no need for the Plugin implementation to do that check.

The two most common ways to implemented this feature are streaming the Job statuses and polling the Job statuses. Both methods can be implemented with the help of the `AbstractJobStatusWatcher` base class; however, streaming is the preferred method, as it should be more efficient than polling.

The `AbstractTimedJobStatusWatcher` class, which extends the `AbstractJobStatusWatcher` class, implements common functionality for polling job statuses. For more details about implementing Job status updates via polling, see TODO #'s 9 - 11 in the RStudio Launcher Plugin SDK QuickStart Guide.

6.5.1 Streaming

Streaming is the preferred method for implementing job status updates, as it can be more efficient than polling job statuses. This is because polling requires making a job status request of the Job Scheduling System every interval of time, and may result in reading the same status multiple times before a change is observed. If the Job Scheduling System provides an API that streams Job status changes, the Plugin should only have to process each status change once.

6.5.1.1 Example

Suppose that the Mars API provides a `stream_statuses` function which takes a callback function as a parameter with the signature `std::function<void(const mars_api::job_status&)>`. Assume that the `mars_api::job_status` structure has the Job ID, the Job name, the Job Status, the last modification time, and the reason for the current Job

status. Then the Plugin developer might change the implementation of `MarsJobStatusWatcher` to the following:

`MarsJobStatusWatcher.hpp`:

```
#ifndef ORCHID_MARS_MARS_JOB_STATUS_WATCHER_HPP
#define ORCHID_MARS_MARS_JOB_STATUS_WATCHER_HPP

#include <jobs/AbstractJobStatusWatcher.hpp>

#include <memory>

namespace orchid {
namespace mars {

class MarsJobStatusWatcher :
  public jobs::AbstractJobStatusWatcher,
  public std::enable_shared_from_this<MarsJobStatusWatcher>
{
public:
   /**
    * @brief Constructor.
    *
    * @param in_jobRepository           The job repository, from which to look-up jo
    * @param in_jobStatusNotifier       The job status notifier to which to post job
    */
   MarsJobStatusWatcher(
      jobs::JobRepositoryPtr in_jobRepository,
      jobs::JobStatusNotifierPtr in_jobStatusNotifier);

  /**
   * @brief Starts the Job status watcher.
   *
   * @return Success if the Job status watcher could be started; Error otherwise.
   */
  Error start();

  /**
   * @brief Stops the Job status watcher.
   */
```

```
  void stop();

private:
   /**
    * @brief Handles a change in job status when it is reported by the Mars Job Sche
    *
    * @param in_jobStatus     The new job status.
    */
   void onJobStatusUpdate(const mars_api::job_status& in_jobStatus);

   /**
    * @brief Gets the job details for the specified job.
    *
    * @param in_jobId   The ID of the job to retrieve.
    * @param out_job    The populated Job object.
    *
    * @return Success if the job details could be retrieved and parsed; Error otherw
    */
   Error getJobDetails(const std::string& in_jobId, api::JobPtr& out_job) const overr

   // The Job status stream.
   std::unique_ptr<mars_api::status_stream> m_jobStream;
};

/** Convenience typedef. */
typedef std::shared_ptr<MarsJobStatusWatcher> MarsJobStatusWatcherPtr;

} // namespace mars
} // namespace orchid

#endif
```

MarsJobStatusWatcher.cpp:

```
#include "MarsJobStatusWatcher.hpp"

namespace orchid {
namespace mars {
```

```
typedef std::shared_ptr<MarsJobStatusWatcher> SharedThis;
typedef std::weak_ptr<MarsJobStatusWatcher> WeakThis;

MarsJobStatusWatcher::MarsJobStatusWatcher(
   jobs::JobRepositoryPtr in_jobRepository,
   jobs::JobStatusNotifierPtr in_jobStatusNotifier) :
      jobs::AbstractJobStatusWatcher(
         std::move(in_jobRepository),
         std::move(in_jobStatusNotifier))
{
}

Error MarsJobStatusWatcher::start()
{
  WeakThis weakThis = shared_from_this();

  const options::MarsOptions& opts = options::MarsOptions::getInstance();
  try
  {
    m_jobStream = std::move(mars_api::stream_statuses(
      [weakThis](const mars_api::job_status& in_jobStatus)
      {
        if (SharedThis sharedThis = weakThis.lock)
          sharedThis->onJobStatusUpdate(in_jobStatus);
      }));
  }
  catch (const mars_api::mars_exception& e)
  {
    return createMarsError(e, ERROR_LOCATION);
  }
}

void MarsJobStatusWatcher::stop()
{
  m_JobStream.reset();
}

void MarsJobStatusWatcher::onJobStatusUpdate(const mars_api::job_status& in_jobStatus
{
```

```
   if (in_jobStatus._name.find("[RStudio Launcher]") != std::string::npos)
   {
      system::DateTime lastModified;
      Error error = system::DateTime::fromString(job._last_modified, lastModified);
      if (error)
      {
        // Use the current time as the invocation time instead, but log an error.
        logging::logError(error, ERROR_LOCATION);
        error = updateJobStatus(
          std::to_string(job._id),
          marsStatusToStatus(job._status),
          job._reason);
      }
      else
      {
        error = updateJobStatus(
          std::to_string(job._id),
          marsStatusToStatus(job._status),
          job._reason,
          lastModified);
      }

      if (error)
        logging::logError(error, ERROR_LOCATION);
   }
}

Error MarsJobStatusWatcher::getJobDetails(const std::string& in_jobId, api::JobPtr& o
{
  const options::MarsOptions& opts = options::MarsOptions::getInstance();
  mars_api::job marsJob;
  try
  {
    unsigned long id = std::strtoul(in_jobId);
    marsJob = mars_api::list_job(opts.getMarsServiceUser(), id);
  }
  catch (const std::invalid_argument& e)
  {
    return createInternalError(
```

```
      InternalError::CONVERSION_FAILURE,
      "Job ID " +
        in_jobID +
        " could not be converted to unsigned long: " +
        e.what(),
      ERROR_LOCATION);
  }
  catch (const std::out_of_range& e)
  {
    return createInternalError(
      InternalError::CONVERSION_FAILURE,
      "Job ID " +
        in_jobID +
        " is out of range to be converted to unsigned long: " +
        e.what(),
      ERROR_LOCATION);
  }
  catch (const mars_api::mars_exception& e)
  {
    return createMarsError(e, ERROR_LOCATION);
  }

  // This should only be invoked for Launcher Jobs because of the filtering in pollJo
  // so return an error if somehow it's not a Launcher Job.
  if (job._name.find("[RStudio Launcher]") == std::string::npos)
    return createInternalError(
      InternalError::JOB_NOT_FOUND,
      "Job " + in_jobId + " is not an RStudio Launcher Job.",
      ERROR_LOCATION);

  out_job = marsJobToJob(marsJob);
  return Success();
}

} // namespace mars
} // namespace orchid
```

6.5.2 Other Methods

It is possible that neither streaming nor polling are the best solution for keeping job statuses up to date. The use of an `AbstractJobStatusWatcher` is completely optional, and the Plugin developer may choose to implement this feature in any way that best suits the Job Scheduling System. For example, the provided sample Local Plugin does not use an `AbstractJobStatusWatcher`. Jobs are launched on the local system by forking a new processes and running the requested command in that process. The Local Plugin receives notifications when the child process writes to standard out, standard error, or exits. When the process exits, the Job state is transitioned from `Job::State::RUNNING` to `Job::State::FINISHED`. The Local Plugin also keeps track of when the process should transition from `Job::State::PENDING` to `Job::State::RUNNING` by checking whether the executable name has changed from `rsandbox` (a utility for launching processes in an isolated environment provided with the RStudio Server Pro installation) to the name of the actual executable for the Job.

6.6 Customizing the Job Repository

In 'TODO #8' of the 'RStudio Launcher Plugin SDK QuickStart Guide', the Plugin developer implemented the `AbstractJobRepository::loadJobs` method to populate the Job Repository on bootstrap. In the case that the Plugin needs to do special processing when a Job is added or removed from the repository, it can do so by overriding the other virtual methods on `AbstractJobRepository`.

An example of when this may be necessary is if the Plugin needs to do additional Job state persistence, beyond what the Job Scheduling System will save. A common case of this is Job output. If the user does not specify an output file the Job Scheduling System may not persist the Job output; however, it must be available to the Launcher until the Job expires according to the Launcher's configured `job-expiry-hours`.

There are three additional virtual methods on `AbstractJobRepository` that allow the Plugin developer to customize the behavior of the Job Repository:

- `AbstractJobRepository::onJobAdded`: this method will be invoked when a job is first added to the repository, immediately after successful submission.

- `AbstractJobRepository::onJobRemoved`: this method will be invoked when an expired Job is removed from the system. Any files or other persistent data that were created by the Plugin should be cleaned up in this method.
- `AbstractJobRepository::onInitialize`: this method will be invoked once, when the Job Repository is initialized during bootstrap. The Plugin may do any extra initialization steps that are required and is responsible for returning an `Error` if any necessary initialization steps fail.

The provided sample Local Launcher Plugin manages Job persistence completely within the Plugin. The `LocalJobRepository` implementation may be used as an example for the implementation of all three virtual methods on `AbstractJobRepository`.

6.7 Process Launching

Depending on the API exposed by the Job Scheduling System, it may be necessary to launch child processes to perform actions on the Job Scheduling System, such as running a job or listing the jobs in the system. The SDK provides a number of classes and functions for launching child processes in the `system/Process.hpp` header file.

Child processes launched through the SDK provided process module are run through the `rsandbox` process by default. This is done to ensure that the child process will be run in an isolated environment, however it prevents the parent process from continuing to write standard input to the child process. If this is needed by the Plugin, it is possible to launch the child process directly and keep the standard input stream open by setting `system::process::ProcessOptions::UseSandbox = false` and `system::process::ProcessOptions::CloseStdIn = false` respectively.

The SDK process launching module will escape the command, arguments, standard input, the standard output and standard error files, and environment variables as appropriate. The command, arguments and environment values will be treated literally. Bash expansion of them will not take place. Bash expansion may take place within the standard input, however.

6.8 Custom Output Streams

To create a custom output stream, the Plugin developer must create a class which inherits `api::AbstractOuptutStream` and implements the `api::AbstractOutputStream::start` and `api::AbstractOutputStream::stop` methods.

In the `api::AbstractOutputStream::start` method, the output stream implementation should begin reporting the Job's output. To report output, the implementation must invoke the protected method `api::AbstractOutputStream::reportData` specifying the data and the type of output. The output type will be `OutputType::STDOUT` for standard output, `OutputType::STDERR` for standard error, or if it is not possible to tell `OutputType::BOTH`. It may not be possible to tell the output type if the Job specified the same output location for both standard output and standard error output.

When the stream has completed, the output stream implementation should invoke the protected method `api::AbstractOututStream::setStreamComplete`. The stream is complete when the Job has finished emitting all output. This can only happen if the Job is in a completed state, which can be tested with `api::Job::isCompleted`. Even if a Job has completed, some Job Scheduling Systems buffer job output, so it may take a few seconds after the Job has completed for the remainder of the job output to be emitted.

If `api::AbstractOutputStream::stop` has been invoked, the output stream implementation should stop streaming data, even if the stream has not been completed.

For an example of a correct and complete implementation of an `api::AbstractOutputStream` child class, please refer to `api::FileOutputStream`.

6.8.1 Customizing the File Output Stream

It is possible that the Plugin will be able to read Job output from a file, but it will need to process the Job output in some way before surfacing the output to the user. For example, the RStudio Slurm Launcher Plugin emits one line of output at the start of each Job that represents extra Job metadata that it needs, and one line at the end of each Job to indicate that all output has been emitted. In that case, the Plugin may customize the behavior of the `api::FileOutputStream` class by

inheriting from it and overriding `api::FileOutputStream::onOutput` and/or `api::FileOutputStream::waitForStreamEnd`. By default `api::FileOutputStream::onOutput` emits every line of output and `api::FileOutputStream::waitForStreamEnd` waits for a fixed short period of time after the Job enters a completed state before ending the stream.

The RStudio Slurm Launcher Plugin would override `api::FileOutputStream::onOutput` to skip the first and last lines of output, and to notify a condition variable when the last line of output is emitted. It would override `api::FileOutputStream::waitForStreamEnd` to wait on the aforementioned condition variable instead of waiting for a fixed period of time.

Chapter 7

RStudio Launcher Plugin SDK Architecture

The RStudio Launcher Plugin SDK is designed to allow the Plugin developer to implement as little code as possible in order to get a working Plugin. Figure 7.1 describes the architecture of the SDK at a high, component level. The Plugin developer only needs to be concerned with those components which interface directly with the job scheduling system: Job Status Watcher, Job Source, Resource Utilization Stream, and Output Stream. Nevertheless, it can be useful to have a more full understanding of the workings of the SDK when making complex implementation decisions.

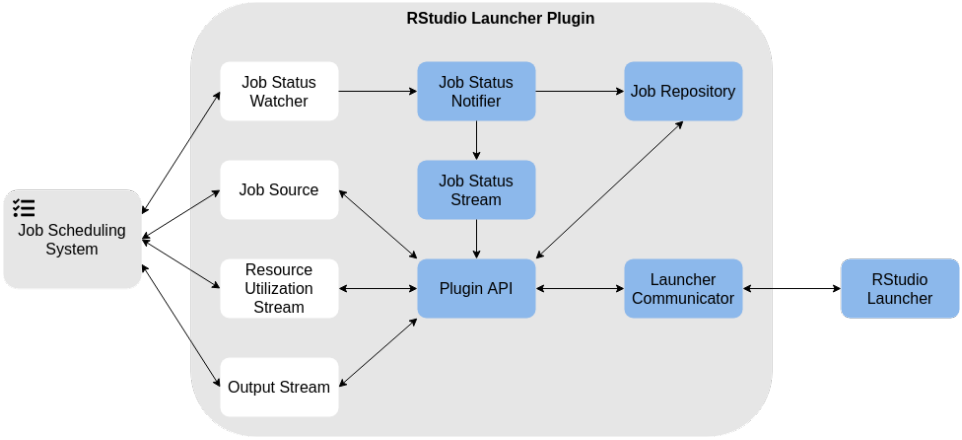

Figure 7.1: High Level Architecture

7.1 Launcher Communicator Component

The Launcher Communicator component is responsible for receiving and interpreting requests from the Launcher, and translating and sending responses to the Launcher.

The Launcher Communicator listens for data on a background thread. When data is received, the Launcher Communicator component parses and validates the data and converts each request it finds into the appropriate Request object. As parsing a request is an expensive operation, the Launcher Communicator also performs this task on a background thread. The Request object is then passed to the Plugin API component to be fulfilled.

When the Plugin API has the response to a request, it posts the Response object to the Launcher Communicator. The Launcher Communicator then formats the response in a way the Launcher will understand and sends it to the Launcher on a background thread.

This component is fully implemented by the SDK, and requires no action from the Plugin developer.

7.2 Components

7.2.1 Plugin API Component

Where the Launcher Communicator is responsible for translating the requests from the Launcher, the Plugin API component is responsible for understanding the meaning of them. Given a particular request, the Plugin API's responsibility is to dispatch the correct action from the Job Source and then convert the output to the appropriate Response object.

Chapter 5 will discuss each API call and how the Plugin API component translates those requests into actionable items for the Job Source or other components.

Each Request that is received is processed on its own thread. As a result, classes which are implemented by the Plugin developer need to be thread safe. If the limit of threads for the process is reached, Requests will be queued until a thread is available.

7.2.2 Job Source Component

The Job Source component is the main point of contact between the job scheduling system and the Plugin. The majority of the implementation work for the Plugin developer will be within this component. There are a few Launcher requests which require streamed responses. These streamed responses are the only ones which aren't covered by the Job Source.

In the majority of cases, when the Plugin API receives a request from the Launcher, it will invoke one or more methods on the Job Source and then compose a response for the Launcher based on the returned data.

There is only one instance of the Job Source per instance of the Plugin. Because the Launcher can make multiple requests to a single Plugin without receiving their responses, it is possible that the Plugin API will invoke the same method on the Job Source for the same resource concurrently. In that case, the Job Source may perform the requests sequentially, if necessary. There is no requirement that Launcher requests are responded to in any particular order. For the sake of performance, however, it is ideal to perform any tasks concurrently which may be safely performed concurrently.

7.2.3 Job Repository Component

The Job Repository component maintains a store of jobs. Jobs which have been finished for a configurable period of time will be purged from the system. The Job Repository is mainly managed by SDK implemented classes. To ensure proper job status records, the Plugin developer only needs to be concerned with implementing a Job Status Watcher.

The Job Status Watcher is responsible for interfacing with the job scheduling system to keep track of job status changes. For the convenience of the Plugin developer, there are two base classes for the Job Status Watcher which may be inherited from: `AbstractJobStatusWatcher` and `AbstractTimedJobStatusWatcher`.

The `AbstractTimedJobStatusWatcher` has a pure virtual method which is responsible for getting the statuses of all the jobs each time it is invoked. On a configurable timer, the `AbstractTimedJobStatusWatcher` will invoke said method, and then use the results to update the Job Status Notifier component, which will in turn update any components which are interested in hearing about job status changes, such as all open Job Status Streams.

On the other hand, the `AbstractJobStatusWatcher` has a pure virtual method which is responsible for streaming job statuses until the stream is canceled. The Plugin developer may choose to use either base class, depending on whether the job scheduling system has a way to stream job statuses or not.

7.2.4 Stream Components

The remaining components described in the figure 7.1 are all related to streamed responses. There are three types of requests which require streamed responses: Job Status Stream, Job Output Stream, and Job Resource Utilization Stream.

All three stream components are constructed by the Plugin API upon receiving the same request type, and are expected to be able to stream their respective data until canceled or completed. Each stream component has different requirements of the Plugin developer.

7.2.4.1 Job Status Stream Component

The Job Status Stream is completely implemented by the SDK and only requires a working Job Status Watcher implementation on the part of the Plugin developer. Each time a Job Status Stream request comes in from the Launcher, the Plugin API will construct a Job Status Stream object with the appropriate parameters and it will subscribe to the job status notifier and emit job status updates to the Plugin API as necessary.

7.2.4.2 Resource Utilization Stream Component

The Resource Utilization Stream component is responsible for streaming whatever resource utilization details are available from the job scheduling system for a particular job. Upon construction, the Resource Utilization Stream is responsible for streaming resource utilization updates from the job scheduling system and emitting them to the Plugin API as they are available. The stream should continue until it is canceled by the Plugin API.

If the job scheduling system does not support streaming resource utilization metrics, the Plugin developer may inherit from `AbstractTimedResourceUtilStream`

rather than `AbstractResourceUtilStream`. The difference between the two is the same as the difference between `AbstractTimedJobStatusWatcher` and `AbstractJobStatusWatcher` as described in section 4.2.3.

Not all job scheduling systems may expose the same resource utilization metrics, and so it may not be possible for every Plugin to fulfill this request in the same way. While this feature is useful for server administrators to track the utilization of their server resources, it is not necessary for the main functionality of the Plugin, and may be considered a best-effort feature.

7.2.4.3 Output Stream Component

The Output Stream component is responsible for streaming output data for a particular job, given that the job is either running or finished. Each Output Stream must be able to stream standard output, error output, or both simultaneously.

Like the other streams, the Output Stream should emit the requested job's output data until it is canceled; however, it is also possible for an Output Stream to finish. When a job stops executing, either because it is killed or it finishes, it should no longer write output. In that case, the Output Stream may emit the remainder of output data and then self terminate. That being said, some job scheduling systems may buffer output data and write it shortly after the job enters a finished state. If that happens, the Plugin developer must find a way to determine when job output is truly finished. For example, when implementing the RStudio SLURM Launcher Plugin it was necessary to emit a recognizable but very likely unique string as the last step of each job. This way, the Output Stream for the RStudio SLURM Launcher Plugin could remain open until it read that string. The string itself is filtered from the data which is emitted to the Plugin API.

7.2.4.4 Stream Concurrency

Because the Plugin API will construct a new Resource Utilization Stream and Output Stream for each request of those types it receives, it is not necessary that the implementation of those classes be thread safe. Each instance should only be operating on a single thread. However, it is necessary that multiple Output Streams can be constructed and concurrently stream job output data for a single job.

7.3 Plugin Startup

When the Plugin is first launched it will initialize a number of components before entering its normal operating mode. When implementing advanced features of the SDK, it may be useful for the developer to know the order of operations prior to normal operating mode. The ordered list is below.

1. The program ID is set for the whole process. This is used for logging.
2. A system log destination is created and attached to the process at the INFO log level.
3. A standard error log destination is created and attached at the INFO level to capture any issues with reading options.
4. Default options are initialized.
5. The main process is initialized. Custom options should be initialized here.
6. The options are read and validated.
7. The standard error log is detached.
8. If higher level logging was requested, a file log destination is created using the log-dir location.
9. The Launcher Communicator is constructed.
10. Signal handlers are configured and core dumps are enabled.
11. The Launcher Plugin API is created and initialized.
12. The worker thread pool is created with the configured number of threads.
13. The Launcher Communicator is started.
14. The Plugin enters normal operation mode until it is killed or hits an unrecoverable error.

7.4 Plugin Tear Down

When the Plugin receives a termination signal from the Launcher, sit starts the tear down process, which is described below:

1. The Launcher Communicator is stopped.
2. The worker thread pool is stopped. This cancels all background thread work, including active streams (such as job status streams or output streams).
3. The Plugin waits for all activity to fully stop.
4. The Plugin exits with exit code 0.

Chapter 8

Smoke Test Utility

The Smoke Test utility is provided with the RStudio Launcher Plugin SDK to provide the Plugin developer a way to exercise the basic functionality of the Plugin, particularly in a way that allows the developer to easily attach a debugger to the Plugin. The Smoke Test utility is not intended to replace integration testing with the RStudio Launcher and the RStudio product which will use the launcher.

8.1 Plugin Options

The Smoke Test utility will start the Plugin with the following options, which should not be changed:

`--heartbeat-interval-seconds=0`

`--enable-debug-logging=1`

`--unprivileged=1` if the Smoke Test utility is launched without root privileges

Other options may be changed via the Plugin's configuration file, which is usually located at `/etc/rstudio/launcher.<plugin-name>.conf`.

When running the Smoke Test utility in unprivileged mode (see Starting the Tester for more details) it is advisable to change the `scratch-path` option to a location that is readable and writable by the run-as user.

8.2 Starting the Tester

The Smoke Test utility takes two required arguments: the full or relative path to the Plugin to test, and the user to test as. It can be run in two modes: privileged and unprivileged. To run the Smoke Test utility in privileged mode, start it with root privileges. For example:

`sudo ./rlps-smoke-test ../plugins/MyPlugin/rstudio-myplugin-launcher someUser`.

To run the Smoke test utility in unprivileged mode, start it as the user specified in the second argument.

Privileged mode is the most common running environment for a Plugin; however, there are some situations where the Plugin cannot be run in privileged mode, such as within a Docker container, so it is advisable to test both scenarios.

When the Smoke Test utility starts, it will send the Plugin a Bootstrap Request and print the response. The Bootstrap response must be successful, or future requests will fail. Next the utility will enter a loop of printing the menu, waiting for input, and performing the requested action, until the user explicitly exits the utility. At any time, the user can provide `q` or `Q` as input to exit the utility.

8.3 Using the Tester

Each time the Smoke Test utility completes a requested action, it will print a menu of available actions. After an action is chosen, the utility will perform the action and print any responses from the Plugin. These responses must be manually verified, as their expected values may vary between Plugin implementations.

This section documents what each menu item will do. For an example of how a Plugin might respond to each of these tests, the Smoke Test utility may be run using the provided sample Local Launcher Plugin.

8.3.1 1. Get cluster info

This menu item will send a Cluster Info request to the Plugin and print the response. It should include all of the maximum and default resource

limits, any available placement constraints, the container information, and the available queues. If any of these values are not configured or are not supported by the Plugin, they will not be included in the response.

8.3.2 2. Get all jobs

This menu item will send a Job State request to the plugin with the `jobId` field set to `'*'` and print the response. It should show all RStudio jobs in the underlying Job Scheduling System, regardless of state.

8.3.3 3. Get filtered jobs

This menu item will send a Job State request to the plugin with the `tags` field set to `[ "filter job" ]` and print the response. It should show any jobs that were launched with menu item 7 or menu item 10.

8.3.4 4. Get running jobs

This menu item will send a Job State request to the plugin with the `statuses` field set to `[ "Running" ]` and print the response. It should show any RStudio jobs in the underlying Job Scheduling System that are currently running. It may be useful to run menu item 10 immediately prior to running this menu item in order to have a Job that will still be running.

8.3.5 5. Get finished jobs

This menu item will send a Job State request to the Plugin with the `statuses` field set to `[ "Finished" ]` and print the response. It should show any RStudio jobs in the underlying Job Scheduling System that have finished running without a failure. Note that jobs which were killed (e.g. via menu item 17) will not be displayed in this response, but jobs which were stopped (e.g. via menu item 18) should be.

8.3.6 6. Get job statuses

This menu item will send a Job Status Stream request to the Plugin and print the responses. It will attempt to wait for at least one response for

each job that was submitted to the Plugin via the Smoke Test utility since it last started, before canceling the stream request. All RStudio Jobs may be included in this response, but it is possible some may not be displayed as the request could be canceled early if there were many RStudio Jobs in the system before the Smoke Test utility was last started.

8.3.7 7. Submit quick job (matches filter)

This menu item submits a basic job to the Job Scheduling System via the Submit Job request and prints the response. This job will match the filter provided to the Job State request sent in menu item 3. The Job name is `Quick Job 1`.

If this menu item is followed by menu item 11 or menu item 12 the `stdout` output should be `This is an environment variable!` and the stderr output should be empty.

8.3.8 8. Submit quick job 2 (doesn't match filter)

This menu item submits another basic job to the Job Scheduling System via the Submit Job request and prints the response. This job will not match the filter provided to the Job State request sent in menu item 3. The Job name is `Quick Job 2`.

If this menu item is followed by menu item 11 or menu item 12 the `stdout` output should be `This is a shell command.` and the `stderr` output should be empty.

8.3.9 9. Submit stderr job (doesn't match filter)

This menu item submits a job to the Job Scheduling System which will emit `stderr` output via the Submit Job request and prints the response. This job will not match the filter provided to the Job State request sent in menu item 3. The Job name is `Stderr job`.

If this menu item is followed by menu item 11 or menu item 13 the `stdout` output should be empty and the `stderr` output should include an error about the incorrect usage of `grep`.

8.3.10 10. Submit long job (matches filter)

This menu item submits a longer running job to the Job Scheduling System via the Submit Job request and prints the response. This job will match the filter provided to the Job State request sent in menu item 3. The Job name is `Slow job`.

This job will iterate from `1` to `11`, echoing `${i}...` and then sleeping for `${i}` seconds at each iteration of the loop. It may be useful to use menu item 11 or menu item 12 immediately after this menu item in order to validate that output streaming works correctly when the output is still being written after the stream has been opened.

8.3.11 11. Stream last job's output (stdout and stderr)

This menu item sends a Job Output Stream request for both `stdout` and `stderr` output to the Plugin for the last Job that was submitted to the Plugin via the Smoke Test utility. If no Job has been submitted to the Plugin since the Smoke Test utility was started, an error will be emitted.

Even if there is no `stdout` or `stderr` output for the last job, there should still be a response from the Plugin indicating that the output is empty.

8.3.12 12. Stream last job's output (stdout)

This menu item sends a Job Output Stream request for only `stdout` output to the Plugin for the last Job that was submitted to the Plugin via the Smoke Test utility. If no Job has been submitted to the Plugin since the Smoke Test utility was started, an error will be emitted.

Even if there is no `stdout` output for the last job, there should still be a response from the Plugin indicating that the output is empty.

8.3.13 13. Stream last job's output (stderr)

This menu item sends a Job Output Stream request for only `stderr` output to the Plugin for the last Job that was submitted to the Plugin via the Smoke Test utility. If no Job has been submitted to the Plugin since the Smoke Test utility was started, an error will be emitted.

Even if there is no `stderr` output for the last job, there should still be a response from the Plugin indicating that the output is empty.

8.3.14 14. Stream last jobs' resource utilization (must be running)

This menu item sends a Job Resource Utilization Stream request for the last Job that was submitted to the Plugin via the Smoke Test utility. If no Job has been submitted to the Plugin since the Smoke Test utility was started, an error will be emitted.

The Job must currently be running in order for resource utilization data to be returned. If the Job has already completed, there should be still be a response from the Plugin indicating that the stream is complete.

Running menu item 10 immediately before this menu item may provide the best possibility of observing resource utilization stream responses from the Plugin.

8.3.15 15. Get last job's network information

This menu item sends a Job Network request to the Plugin for the last Job that was submitted to the Plugin via the Smoke Test utility. If no Job has been submitted to the Plugin since the Smoke Test utility was started, an error will be emitted.

The network information displayed in the response should match the network information of the machine which ran the Job, according to the underlying Job Scheduling system.

8.3.16 16. Submit a slow job and then cancel it

This menu item submits a slow job to the Job Scheduling System via the Submit Job request and then attempts to immediately cancel it via the Control Job request with the `operation` set to `4` (see the JobOperation section). Depending on the speed of the Job Scheduling system, it is possible that the Job will enter the `Running` state before the Cancel request is received. Because of this, it is not unexpected that an Error Response with an error code of InvalidJobState may be returned by the Plugin.

If the cancel operation completed successfully, invoking menu item 11, menu item 12, or menu item 13 should show that the Job emitted no output.

8.3.17 17. Submit a slow job and then kill it

This menu item submits a slow job to the Job Scheduling System via the Submit Job request, waits for one second, and then attempts to kill it via the Control Job request with the `operation` set to `3` (see the JobOperation section). Depending on the speed of the Job Scheduling system, it is possible that the Job will not have entered the `Running` state before the Kill request is received. Because of this, it is not unexpected that an Error Response with an error code of InvalidJobState may be returned by the Plugin.

If the kill operation completed successfully, invoking menu item 11, menu item 12, or menu item 13 should show that the Job emitted no output.

8.3.18 18. Submit a slow job and then stop it

This menu item submits a slow job to the Job Scheduling System via the Submit Job request, waits for one second, and then attempts to stop it via the Control Job request with the `operation` set to `2` (see the JobOperation section). Depending on the speed of the Job Scheduling system, it is possible that the Job will not have entered the `Running` state before the Kill request is received. Because of this, it is not unexpected that an Error Response with an error code of InvalidJobState may be returned by the Plugin.

If the kill operation completed successfully, invoking menu item 11, menu item 12, or menu item 13 should show that the Job emitted no output.

8.3.19 19. Submit a slow job, suspend it, and then resume it

This menu item submits a slow job to the Job Scheduling System via the Submit Job request, waits for one second, and then attempts to suspend it via the Control Job request with the `operation` set to `1` (see the JobOperation section). Next the utility waits for one second and then attempts to resume the job via the Control Job request with the `operation` set to `0` (see the JobOperation section).

Depending on the speed of the Job Scheduling system, it is possible that the Job will not have entered the `Running` state before the Suspend request is received. Because of this, it is not unexpected that an Error Response with an error code of InvalidJobState may be returned by the Plugin.

If the kill operation completed successfully, invoking menu item 11 or menu item 12 should show that the Job emitted `"Done."` to standard output after the Job finishes.