RStudio Launcher Plugin Software Development Kit 99.9.9

QuickStart Guide

2021-12-17

Contents

Chapter 1

Preface

This is the quick start guide for writing RStudio Launcher Plugins using the RStudio Launcher Plugin SDK. It will cover the minimum steps required to create a working RStudio Launcher Plugin for a job scheduling system. For more detailed information and advanced topics, please see the RStudio Launcher Plugin SDK Developer's Guide.

Chapter 2

Introduction

The purpose of the RStudio Launcher is to provide a generic interface between the RStudio IDE and an arbitrary job scheduling system. Each RStudio Launcher Plugin will allow the RStudio IDE to launch jobs, such as R Sessions, R scripts, and Jupyter Notebooks, to a job scheduling system without requiring any modification to the RStudio IDE itself.

The goal of an RStudio Launcher Plugin is to implement each RStudio Launcher API in a way that the intended action of the API will be completed by the job scheduling system with which it interfaces. Some features of the job scheduling system may not be exposed, and some limitations of the job scheduling system may need to be worked around.

Figure 2.1 depicts the architecture of the RStudio Launcher, using three existing RStudio Launcher Plugins as examples. The following communication will occur if a user creates a new R session in the Kubernetes cluster:

1. RStudio Server Pro will build the command and arguments to run the R Session.
2. RStudio Server Pro will send a job submission request to the RStudio Launcher over HTTP, including the R Session command, arguments, the user-requested cluster, and any resource constraints set by the user.
3. The RStudio Launcher will verify that a cluster matching the requested Kubernetes cluster exists.
4. Upon finding a valid Kubernetes cluster, the RStudio Launcher will forward the request to the RStudio Kubernetes Launcher Plugin via the JSON RStudio Launcher Plugin API.

5. The Kubernetes Plugin will create a pod with the requested resource restrictions and launch the R Session with the specified command and arguments, via the Kubernetes HTTP API.
6. The Kubernetes Plugin will report either success or failure to the RStudio Launcher via the JSON RStudio Launcher Plugin API, based on the response from the Kubernetes HTTP API.
7. The RStudio Launcher will forward the result to RStudio Server Pro via the RStudio Launcher HTTP API.

Many of the steps listed above are the same regardless of the job scheduling system in use. The RStudio Launcher Plugin SDK aims to reduce the development time and maintenance cost of RStudio Launcher Plugins by implementing common functionality and providing a straightforward framework for implementing the job scheduling system specific components of the RStudio Launcher Plugin.

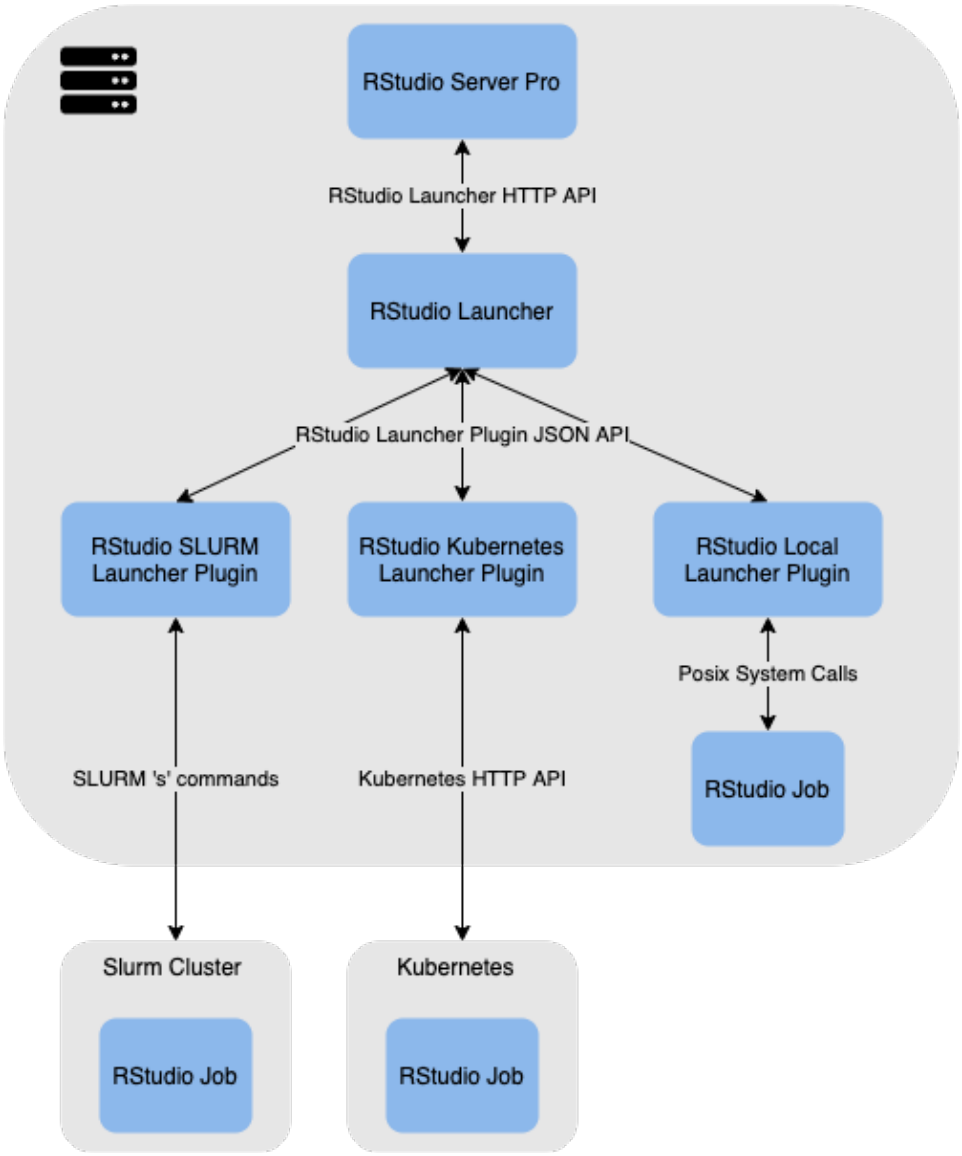

Figure 2.1: RStudio Launcher Architecture

Chapter 3

Getting Started

3.1 Prerequisites

3.1.1 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.

3.1.2 Build Dependencies

To get started developing an RStudio Launcher Plugin using the RStudio Launcher Plugin SDK, it is necessary to install the development tools and dependencies required for building and running RStudio Launcher 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 RStudio Launcher Plugin 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/

3.1.3 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.

3.1.4 Job Scheduling System

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

3.2 Folder Structure

The following describes the purpose of the files and folders which can be found within the RStudio Launcher Plugin SDK.

- `CMakeGlobals.txt` - Defines global values required for the RStudio Launcher Plugin SDK cmake project.
- `CMakeLists.txt` - The CMakeLists definition for the entire RStudio Launcher Plugin SDK cmake project. This file may be opened in any IDE which supports cmake projects for a view of the entire SDK, including the SDK itself, and the sample and QuickStart plugins.
- `dependencies` - This folder contains scripts which may be useful for installing RStudio Launcher Plugin SDK
- `docs` - Formal documentation for the RStudio Launcher Plugin SDK, including the QuickStart Guide, the Developer's Guide and the source code Doxygen documentation.

- `plugins` - The sample Local RStudio Launcher Plugin and the QuickStart RStudio Launcher Plugin.
- `Local` - The source code for the Local RStudio Launcher Plugin.
  - `CMakeLists.txt` - The cmake project definition for the Local RStudio Launcher Plugin. This file may be opened in any IDE which supports cmake projects for a view of only the Local RStudio Launcher Plugin.
- `QuickStart` - The source code for the QuickStart RStudio Launcher Plugin.
  - `CMakeLists.txt` - The cmake project definition for the QuickStart RStudio Launcher Plugin. This file may be opened in any IDE which supports cmake projects for a view of only the QuickStart RStudio Launcher Plugin.
- `sdk` - The source code for the RStudio Launcher Plugin SDK.
- `tools` - This folder contains scripts which are useful when developing and testing the SDK.

3.3 Development Process

The RStudio Launcher Plugin SDK provides the QuickStart RStudio Launcher Plugin to help reduce the complexity of developing an RStudio Launcher Plugin using the SDK. The QuickStart Plugin includes a series of `TODO` items. This document will walk the developer through the steps of each `TODO`. At the end of this document, the developer should have a functional RStudio Launcher Plugin.

If any feature requires a more complex implementation than is possible by following the `TODO` for that feature, please refer to the RStudio Launcher Plugin SDK Developer's Guide.

Chapter 4

TODO Items

This section will walk the developer through each of the QuickStart `TODO` items. As each `TODO` is completed, it is recommended to remove it from the code.

Throughout this chapter, an example Plugin Developer who is part of the "Orchid Organization" and developing a Launcher Plugin for the "Mars" job scheduling system will be used. The theoretical Mars job scheduling provides a C/C++ interface, which will be referred to as the `mars_api`. It provides a set of free functions which allow the caller to get details from the Mars job scheduling system. Any `mars_api` call may throw a `mars_exception`, which needs to be handled appropriately.

4.1 TODO #1: Change Namespaces

TODO Location	Impact
QuickStartMain.cpp	All source files

The classes in the QuickStart RStudio Launcher Plugin are all in the `rstudio::launcher_plugins::quickstart` namespace. As desired, these namespaces may be changed.

4.1.1 Example

The Plugin Developer may change the following namespace definitions

```
namespace rstudio {
namespace launcher_plugins {
namespace quickstart {
  ...
} // namespace quickstart
} // namespace launcher_plugins
} // namespace rstudio
```

to

```
namespace orchid {
namespace mars {
  ...
} // namespace mars
} // namespace orchid
```

4.2 TODO #2: Rename Classes

TODO Location	Impact
QuickStartMain.cpp	All source files

4.2.1 Example

The developer may now wish to rename all the classes from `QuickStart*` to `Mars*`. For example, `QuickStartMain` would become `MarsMain`.

In addition, the developer may also wish to modify the include guards to match their new namespace and class names. For example,

```
#ifndef QUICKSTART_QUICK_START_PLUGIN_API
#define QUICKSTART_QUICK_START_PLUGIN_API
```

might become

```
#ifndef ORCHID_MARS_PLUGIN_API
#define ORCHID_MARS_PLUGIN_API
```

The naming schemes described here are only recommendations. The exact naming scheme is completely at the discretion of the developer.

4.3 TODO #3: Change the Plugin Name

TODO Location	Impact
QuickStartMain.cpp	QuickStartMain.cpp

The plugin name is used to generate the program ID, which will be used for logging, and the configuration file name. By default, the program ID will be `"rstudio-" + getPluginName() + "-launcher"`. The configuration file name will be `"launcher." + getPluginName() + ".conf"` and must be located in `/etc/rstudio`.

4.3.1 Example

The developer might now change

```
std::string getPluginName() const override
{
  // TODO #3: Change the Plugin Name
  return "quickstart";
}
```

to

```
std::string getPluginName() const override
{
  return "mars";
}
```

4.4 TODO #4: Optionally Change the Program ID

TODO Location	Impact
QuickStartMain.cpp	QuickStartMain.cpp

The program ID will be used to uniquely identify logs from the plugin that are sent to the system log. By default, the value will be `"rstudio-" + getPluginName() + "-launcher"`, however it is possible to override `AbstractMain::getProgramID()` to set a custom program ID.

4.4.1 Example

The developer might add

```
std::string getProgramId() const override
{
  return "orchid-" + getProgramName() + "-launcher";
}
```

to `MarsMain.cpp`.

4.5 TODO #5: Add Options

TODO Location	Impact
QuickStartOptions.cpp	QuickStartOptions.cpp, QuickStartOptions.hpp

The QuickStartOptions class demonstrates how to add job scheduling specific options to the Plugin. It may not be immediately obvious what options will be required for the Plugin, but when it does become clear that an option is needed it can be added here.

4.5.1 Example

Suppose that the Mars job scheduling system can be controlled using an HTTP API, and it can be set up at a custom URL and port. The Orchid developer might then add a required option for the URL, an optional option for whether SSL should be used, and an optional option for the port. Additionally, the Mars Launcher Plugin will need to be able to make requests (such as listing all of the jobs in the system) as an administrator user. In that case, the developer might modify QuickStartOptions.hpp and QuickStartOptions.cpp in the following way:

QuickStartOptions.hpp

```
class QuickStartOptions : public Noncopyable
{
public:
// ...

  bool getSampleOption() const;

// ...

private:
// ...

  bool m_sampleOption;
}
```

QuickStartOptions.cpp

```
bool QuickStartOptions::getSampleOption() const
{
  return m_sampleOption;
}

void QuickStartOptions::initialize()
{
   // TODO #5: Add options, as necessary.
   using namespace rstudio::launcher_plugins::options;
   Options& options = Options::getInstance();
```

```
   options.registerOptions()
      ("sample-option",
       Value<bool>(m_sampleOption).setDefaultValue(true),
       "sample option to demonstrate how to register options");
}
```

MarsOptions.hpp

```
namespace orchid {
namespace mars {
namespace options {

class MarsOptions : public Noncopyable
{
public:
// ...

  const std::string& getMarsServiceUser() const;

  const std::string& getHost() const;

  unsigned int getPort() const;

  bool getUseSsl() const;

// ...

private:
// ...

  std:string m_marsServiceUser;

  std::string m_host;

  unsigned int m_port;

  bool m_useSsl;
}
```

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

MarsOptions.cpp

```
#include "options/MarsOptions.hpp"

#include <options/Options.hpp>

namespace orchid {
namespace mars {
namespace options {

const std::string& MarsOptions::getHost() const
{
  return m_host;
}

unsigned int MarsOptions::getPort() const
{
  if (m_port != 0)
    return m_port;

  if (m_useSsl)
    return 443;

  return 80;
}

bool MarsOptions::getUseSsl() const
{
  return m_useSsl;
}

void MarsOptions::initialize()
{
  // TODO #5: Add options, as necessary.
  using namespace rstudio::launcher_plugins::options;
```

```
  Options& options = Options::getInstance();
  options.registerOptions()
    ("mars-service-user",
     Value<std::string>(m_host).setDefaultValue("mars"),
     "a user which can make service requests to the Mars server")
    ("host",
     Value<std::string>(m_host),
     "the IP address or hostname of the Mars server")
    ("port",
     Value<unsigned int>(m_port).setDefaultValue(0),
     "the port to use for connecting to the Mars server")
    ("use-ssl",
     Value<bool>(m_useSsl).setDefaultValue(false),
     "whether to use HTTPS (true) or HTTP (false, default) when connecting to the Mar
}

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

4.6 TODO #6: Initialize Communication

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

The next step is to initialize communication with the job scheduling system. This might involve opening a TCP connection to the job scheduling system which will remain open for the duration of the Plugin's lifetime, or it might involve testing that it is possible to connect to the job scheduling system. In the case of the sample Local Plugin, provided with this SDK, no communication test is necessary since jobs will be run on the local machine.

Any other tasks which need to be completed before the Plugin is ready to launch jobs must be completed here. For example, in the RStudio SLURM Launcher Plugin it was also necessary to check which timezone the SLURM cluster is using, since job timestamps need to be compared later on. If the

Plugin implementation will be tied to a specific version, or set of versions, of the job scheduling system, that should be validated here as well.

4.6.1 Example

The Plugin Developer may now change

```
#include <QuickStartJobSource.hpp>

#include <Error.hpp>

namespace rstudio {
namespace launcher_plugins {
namespace quickstart {

Error QuickStartJobSource::initialize()
{
   // TODO #6: Initialize communication with the job scheduling system. If communicat
   return Success();
}

} // namespace quickstart
} // namespace launcher_plugins
} // namespace rstudio
```

in `QuickStartJobSource.cpp` to

```
#include "MarsJobSource.hpp"

#include <Error.hpp>
#include "options/MarsOptions.hpp"

namespace orchid {
namespace mars {

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

```
  try
  {
    mars_api::init(opts.host(), opts.port(), opts.useSsl());
  }
  catch (const mars_api::mars_exception& e)
  {
    return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
  }

  return Success();
}

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

4.7 TODO #7: Define Cluster Configuration

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

RStudio applications which make use of the Launcher can determine the configuration and capabilities of each job scheduling system by submitting a Cluster Info request to the Launcher. Each Plugin is responsible for returning a complete and correct list of the configuration and capabilities of the job scheduling system with which it integrates. For more information about the Cluster Info request, see the RStudio Launcher Plugin SDK Developer Guide.

Some aspects of the configuration and capabilities may be independent of the end user or the Launcher, and others may differ by user. If any configuration or capability values should differ based on the end user, it is likely that these rules should be configurable by the system admin. For the convenience of the Plugin Developer, an `AbstractUserProfiles` class has been provided to facilitate the implementation of User Profiles. For more details about implementing this feature, please refer to the 'User Profiles' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer Guide.

This section will describe the questions which need to be answered to correctly define the configuration and capabilities of a job scheduling system, and what value(s) of the `JobSourceConfiguration` structure should be modified in the overridden implementation of `IJobSource::getConfigurations`.

- Does the job scheduling system support or require containers (such as Docker or Singularity containers)? If yes:
  - set `JobSourceConfiguration::ContainerConfig::SupportsContainers` to `true`. Note that if `JobSourceConfiguration::ContainerConfig::SupportsContainers` is set to `false`, all other values of `JobSourceConfiguration::ContainerConfig` will be ignored.
  - decide how the list of available container images should be determined and add each container image name to `JobSourceConfiguration::ContainerConfig::ContainerImages`.
  - decide how the default container image should be determined, if any, and then set the value of `JobSourceConfiguration::ContainerConfig::DefaultImag`
  - decide whether unknown images should be allowed and set `JobSourceConfiguration::ContainerConfig::AllowUnknownImages` to `true` or `false` as desired. > Info: The RStudio Kubernetes Launcher Plugin allows system administrators to control the list of available container images, the default image, and whether unknown images should be allowed on a per-user or per-group basis through the User Profiles feature. For an example of what that configuration looks like, see the sample `launcher.kubernetes.profiles.conf` file in the Job Launcher Plugin Configuration section of the RStudio Job Launcher Administrator's Guide.

- Does the job scheduling system support job queues? A job queue is a subset of machines within the job scheduling system. When a job is submitted to the job scheduling system, a specific queue may be requested and the job should be run on one or more of the machines in the requested queue. In some job scheduling systems, this may also be known as a "partition". If yes:
  - pull down the list of queues from the job scheduling system, or otherwise determine the list available queues, and add each queue to `JobSourceConfiguration::Queues`.

- Does the job scheduling system support setting resource limits on a job?
  - determine the type of resource limits that are supported by the job scheduling system. The SDK supports the following resource types by default: Cpu Count, Cpu Time, Memory, and Memory Swap. More types can be added by setting the `ResourceLimit::ResourceType` field to the name of the desired type. For each type add a `ResourceLimit` object of that type to `JobSourceConfiguration::ResourceLimits`, optionally setting the maximum and default values of that resource limit type.

    Info: The existing RStudio Launcher Plugins allow system administrators to configure the maximum and default values for each resource type on a per-user or per-group basis through the User Profiles feature. From an exampleof what that looks like, see a sample `launcher.<plugin name>.profiles.conf` Job Launcher Plugin Configuration section of the RStudio Job Launcher Administrator's Guide.

- Does the job scheduling system support other placement constraints that are important to surface? It is not necessary to surface every placement constraint supported by the job scheduling system; however if any constraints are commonly used or otherwise important to the end user, they may be surfaced through this feature. Examples of such constraints may be other types of resource limits (e.g. disk space, GPU type, Processor Type, etc.), or other types of constraints entirely (e.g. disk type, region of a cloud service, etc.). Placement constraints can either be an enumeration style set of possible values or a free-form text value. If yes:
  - for each enumeration-style placement constraint, add one `PlacementConstraint` object to `JobSourceConfiguration::PlacementConstraints` for each possible value of that constraint.
  - for each free-form placement constraint, add one `PlacementConstraint` object to `JobSourceConfiguration::PlacementConstraints` with no value.

Important: Whether or not to use the `system::User` parameter provided to each of the capabilities methods is at the discretion

of the Plugin Developer. If the Plugin Developer does not wish to support User Profiles and does not have another method to determine per-user or per-group rules, the `system::User` parameter may simply be ignored.

If there are any job configuration values which cannot be covered by one of the above categories, refer to the 'Custom Job Source Configuration' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer's Guide.

4.7.1 Example

Assume that the Mars scheduling system has the following capabilities:

- container support
- a way to list available container images that are available to a given user
- no default container image
- no way to specify unknown images
- a way to set a limit on the number of CPUs, the amount of Memory, and the number of GPUs, and a way to check the maximum possible values that can be requested
- a way to choose between the processor type: `x86` or `ARM`
- optionally, a way to select which host(s) the Mars scheduler can use to run the job
- job queues

The Plugin developer can report this support by changing

```
Error QuickStartJobSource::getConfiguration(
  const system::User& in_user,
  api::JobSourceConfiguration& out_configuration) const
{
  // TODO #7: Define cluster configuration.
  return Success();
}
```

in `QuickStartJobSource.cpp` to

```
Error MarsJobSource::getConfiguration(
  const system::User& in_user,
  api::JobSourceConfiguration& out_configuration) const
{
  api::ContainerConfiguration& containerConfig = out_configuration.ContainerConfig;
  containerConfig.SupportsContainers = true;

  int maxGpuCount = 0, maxCpuCount = 0, maxMemoryMB = 0;
  try
  {
    // Get the container images and the queues.
    containerConfig.ContainerImages = mars_api::get_images(in_user.getUsername());
    out_configuration.Queues  = mars_api::get_queues());

    // Get the maximum resource limit and placement constraint values.
    maxGpuCount = mars_api::get_total_gpus();
    maxCpuCount = mars_api::get_total_cpus();
    maxMemoryMB = mars_api::get_max_memory();
  }
  catch (const mars_api::mars_exception& e)
  {
    return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
  }

  // Populate the resource limits.
  out_configuration.ResourceLimits.emplace_back(
    api::ResourceLimit::Type::CPU_COUNT,
    std::to_string(maxCpuCount));
  out_configuration.ResourceLimits.emplace_back(
    api::ResourceLimit::Type::MEMORY,
    std::to_string((double)maxMemoryMB));
  out_configuration.ResourceLimits.emplace_back(
    "GPUs",
    std::to_string(maxGpuCount),
    "0");

  // Populate the placement constraints
  out_configuration.PlacementConstraints.emplace_back("Processor Type", "x86");
  out_configuration.PlacementConstraints.emplace_back("Processor Type", "ARM");
```

```
  out_configuration.PlacementConstraints.emplace_back("Processor Type", "Any");

  out_configuration.PlacementConstraints.emplace_back("Hostname");

  return Success();
}
```

in `MarsJobSource.cpp`.

4.8 TODO #8: Pull All Jobs

TODO Location	Impact
QuickStartJobRepository.cpp	QuickStartJobRepository.cpp

When the Launcher starts the Plugin, it will send a Bootstrap request to ensure that the Plugin is initialized. During this request, the Plugin SDK will initialize the `AbstractJobRepository` which will invoke the concrete implementation of `AbstractJobRepostiory::loadJobs`. The Plugin developer needs to override `AbstractJobRepostiory::loadJobs` to synchronously pull down the list jobs that are already in the Job Scheduling System. The `IJobSource` will be initialized successfully before the Job Repository is initialized.

Any jobs which were not launched by the Plugin should not be included in the output. To distinguish between jobs that were launched by the Plugin and jobs that were launched either manually or by another tool, it may be useful to preface the name of the job with a unique tag (e.g. `'[RStudio Launcher] <job name>`). Alternately, if the Job Scheduling System supports setting custom fields on the job, a custom field may be used to indicate that the job was launched through the Plugin. Note that if a unique prefix is added to Launcher jobs by the Plugin, it should be removed from the job name before being returned to the Launcher.

Each job detail must be converted from the format in the Job Scheduling System to the format understood by the Launcher. For example, if dates are represented as a string in the Job Scheduling System, they must be parsed as a `system::DateTime` before the associated `Job` object can be updated.

For more information about parsing dates from strings, see the 'Date-Time Support' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer's Guide.

4.8.1 Job Fields

The `Job` object has a field for each detail that the Launcher requires. This document will discuss some of the fields which require special consideration. Because jobs may remain in the system between Plugin restarts, the Plugin must be able to populate each field of the `Job` object exactly as it would have been when it was submitted to the Plugin based on the record of the job in the Job Scheduling System. If any fields cannot be retrieved from the Job Scheduling System, it will most likely be easier to ignore them during this TODO and design a solution during TODO #12. For more information about each field, see the RStudio Launcher Plugin SDK API Reference.

4.8.1.1 Command vs. Exe vs. Container

Depending on whether a Plugin supports containers, there are two or three possible ways a job may be submitted to the Plugin. Without container support, a job may be submitted wither either the `Job::Command` or the `Job::Exe` field set. The difference is that a job submitted with the `Job::Command` field populated should be executed as a shell command (e.g. `/bin/sh -c <Command>`), and a job submitted with the `Job::Exe` field populated should be executed directly. With container support, the job may be submitted with the `Command` and `Container` fields set, with the `Exe` and `Container` fields set, or just with the `Container` field set. In that case, the `Container` field describes the `Container` that should run either the shell command or the executable, or if no command or executable was set, the `Container` itself is the job to be run. Job submission is discussed in more detail in TODO #12.

The Plugin must be able to determine from the Job Scheduling System's record of the job which of the `Job::Command`, `Job::Exe`, and `Job::Container` fields should be populated, and must be able to repopulate those fields exactly as they were when they were submitted to the Plugin.

4.8.1.2 Job Status

Another special consideration is how to convert from the Job Scheduling System's job status to the RStudio Launcher Plugin SDK job status. To help with the conversion, below is a table which describes all of the `Job::State` enumeration values and their meaning with respect to the state of the job.

`Job::State` Value	Meaning
`CANCELED`	The job was canceled by the user before it transitioned to the `RUNNING` state.
`FAILED`	The job could not be launched by the Job Scheduling System. This does not include a job that runs and exits with a non-zero exit code.
`FINISHED`	The job was launched by the Job Scheduling System and has exited. This includes jobs which exit with a non-zero exit code.
`KILLED`	The job was forcibly stopped, using `SIGKILL`.
`PENDING`	The job has been submitted to the Job Scheduling System but has not started running yet.
`RUNNING`	The job is currently running.
`SUSPENDED`	The job has been paused by the user, and may be resumed later.
`UNKNOWN`	This is not a job status - it represents the default value of the job state, before it has been submitted to the Plugin. Jobs populated by the Plugin should not have this status.

4.8.2 Example

Assume the following:

- The Mars Job Scheduling System API has a `list_jobs` function which takes a username. It retrieves the jobs the specified user has permission to see.
- The Orchid organization's developer has added a `marsJobToJob` helper function which converts a `mars_api::job` structure to an `rstudio::launcher_plugins::api::Job` structure and

has the signature `rstudio::launcher_plugins::api::JobPtr marsJobToJob(const mars_api::job& in_marsJob)`.
- The Plugin developer plans to append `"[RStudio Launcher]"` to the Job name on submission.

In that case, the Plugin developer might change

```
Error QuickStartJobRepository::loadJobs(api::JobList& out_jobs) const
{
  // TODO #8: Pull all RStudio jobs from the job scheduling system synchronously.
  return Success();
}
```

in `QuickStartJobRepository.cpp` to

```
Error MarsJobRepository::loadJobs(api::JobList &out_jobs) const
{
  const options::MarsOptions& opts = options::MarsOptions::getInstance();
  std::vector<mars_api::job> marsJobs;
  try
  {
    marsJobs = mars_api::list_jobs(opts.getMarsServiceUser());
  }
  catch (const mars_api::mars_exception& e)
  {
    return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
  }

  for (const mars_api::job& job: marsJobs)
    if (job._name.find("[RStudio Launcher]") != std::string::npos)
      out_jobs.push_back(marsJobTojob(job));

  return Success();
}
```

in `MarsJobRepository.cpp`.

4.9 TODO #9: Poll Job Statuses

TODO Location	Impact
QuickStartJobStatusWatcher.cpp	QuickStartJobStatusWatcher.cpp

Along with the next two TODOs (#10 and #11), this TODO will walk through how to set up a Job Status Watcher that polls the Job Scheduling System for Job status updates. If it is possible to stream Job status updates from the Job Scheduling System, follow the instructions in the 'Streaming' subsection of the 'Job Status Updates' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer Guide and then skip to TODO #12.

Note that using a Job Status Watcher is optional. If neither polling nor streaming Job status updates is suitable, the Plugin developer may choose to implement the updating of Job statuses however best suits the Job Scheduling System's available functionality. More details can be found in the 'Job Status Updates' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer Guide.

The `AbstractTimedJobStatusWatcher` base class will invoke `pollJobStatus` once every configurable amount of time. When this method is invoked, the Plugin should poll the Job Scheduling System for all the current Job statuses. The SDK will handle verifying whether a status is new, as long as correct information is passed to the `AbstractJobStatusWatcher::updateJobStatus` method. The necessary information is as follows:

- The ID of the Job.
- The current status of the Job.
- The current status message of the Job, if any.
- The current time, or the time at which the Job was last modified, if available.

If clock skew between the Job Scheduling System and the Plugin is a concern, or if the Job Scheduling System provides the last modification time of the Job, it is advisable to provide the last modified time to the `in_invocationTime` parameter of `updateJobStatus`.

The `pollJobStatus` should return an error if and only if it hits an unrecoverable error, such as failing to communicate with the Job Scheduling System. If there is an issue handling a single Job's status update, it may be sufficient to log an error and try again the next time that `pollJobStatus` is invoked.

The Launcher supports load balancing, which means that it is possible that jobs can be submitted to the Job Scheduling System by another instance of the Plugin. Each instance of the Plugin should be able to correctly report all Jobs that were submitted to the Job Scheduling System through the Launcher. Usually, the most straightforward way to implement this is to pick up the new Jobs while polling or streaming Job status updates. If `AbstractJobStatusWatcher::updateJobStatus` is invoked for a Job that is not currently in the Job Repository, it will invoke `AbstractJobStatusWatcher::getJobDetails`. For more details about implementing `AbstractJobStatusWatcher::getJobDetails`, see TODO #10.

4.9.1 Example

In addition to the assumptions from previous examples, assume the following:

- The `mars_api::job` structure has a `last_modified` field which is a string in the format `YYYY-MM-DDTHH:mm:ss.sssss[+-]HH:mm`.
- The `mars_api::job` structure has a `status` field which is a string, and the Plugin developer has added a `marsStatusToStatus` method which converts the Mars job status to it's applicable SDK status equivalent.
- The `mars_api::job` structure has a `reason` field which is a string describing the reason for the current status.
- The `mars_api::job` structure has an `id` field which is an unsigned integer, that the Plugin developer has mapped to an SDK `Job::Id` field by converting the integer value to a string.

In that case, the plugin developer might change

```
Error QuickStartJobStatusWatcher::pollJobStatus()
{
   // TODO #9: Poll the Job Scheduling System for job status updates. Invoke Abstract
```

```
   //          for each updated job.
  return Success();
}
```

in QuickStartJobStatusWatcher.cpp to

```
Error MarsJobStatusWatcher::pollJobStatus()
{
  const options::MarsOptions& opts = options::MarsOptions::getInstance();
  std::vector<mars_api::job> marsJobs;
  try
  {
    marsJobs = mars_api::list_jobs(opts.getMarsServiceUser());
  }
  catch (const mars_api::mars_exception& e)
  {
    return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
  }

  for (const mars_api::job& job: marsJobs)
  {
    if (job._name.find("[RStudio Launcher]") == std::string::npos)
      continue; // Skip non-Launcher jobs.

    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::logErrorMessage("Failed to update job " + std::to_string(job.id), ERRO
      logging::logError(error, ERROR_LOCATION);
    }
  }

  return Success();
}
```

in `MarsJobStatusWatcher.cpp`.

4.10 TODO #10: Get the Missing Job Details

TODO Location	Impact
QuickStartJobStatusWatcher.cpp	QuickStartJobStatusWatcher.cpp

If `AbstractJobStatusWatcher::updateJobStatus` is invoked for a Job which is not in the Job Repository, the SDK will call `AbstractJobStatusWatcher::getJobDetails` to populate a `Job` object with all of the details of the Job, and then add it to the Job Repository. Since `AbstractJobStatusWatcher::getJobDetails` should get the full details of the Job, just like `IJobSource::getJobs()` does, it may be advisable to make a shared implementation for getting and/or parsing Jobs from the Job Scheduling System.

This method should return an error if it is unable to populate the Job details.

4.10.1 Example

In addition to the assumptions from previous examples, assume the following:

- The `mars_api` has a `list_job` function which lists a specific Job by ID.

In that case, the plugin developer might change

```
Error QuickStartJobStatusWatcher::getJobDetails(const std::string& in_jobId, api::Job
{
   // TODO #10: Get the full details of the requested job from the Job Scheduling Sys
   //           error below.
   return Error(
      "NotImplemented",
      1,
      "Method QuickStartJobStatusWatcher::getJobDetails is not implemented.", ERROR_L
}
```

in `QuickStartJobStatusWatcher.cpp` to

```
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();
    marsJob = mars_api::list_job(opts.getMarsServiceUser(), id);
  }
  catch (const std::invalid_argument& e)
  {
    return Error("InternalError", 1, e.what(), ERROR_LOCATION);
  }
  catch (const std::out_of_range& e)
  {
    return Error("InternalError", 2, e.what(), ERROR_LOCATION);
  }
  catch (const mars_api::mars_exception& e)
  {
    return Error("MarsApiError", e.code(), e.what(), 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 Error("JobNotFound", 1, "Job " + in_jobId + " is not an RStudio Launcher J

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

4.11 TODO #11: Adjust the Polling Frequency

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

The `QuickStartJobStatusWatcher` doesn't actually update any Job statuses, as there is no QuickStart Job Scheduling System to pull jobs from. To reduce the amount of busy-work the QuickStart Plugin will do when run, the polling frequency of the Job Status Watcher is set to 1 minute. In reality, Job statuses should be kept much more up to date, on the order of seconds rather than minutes.

The Plugin developer may choose a fixed interval, such as 5 seconds, or allow the RStudio Server Pro administrator to decide the number of seconds using a configuration option.

4.11.1 Example

The Orchid Organization's Plugin developer may choose to change

```
QuickStartJobSource::QuickStartJobSource(
   const jobs::JobRepositoryPtr& in_jobRepository,
   const jobs::JobStatusNotifierPtr& in_jobStatusNotifier) :
   api::IJobSource(in_jobRepository, in_jobStatusNotifier),
   m_jobStatusWatcher(
         new QuickStartJobStatusWatcher(
```

```
                system::TimeDuration::Minutes(1),
                in_jobRepository,
                in_jobStatusNotifier))
{
   // TODO #11: Adjust the job status watcher frequency.
}
```

in `QuickStartJobSource.cpp` to

```
MarsJobSource::MarsJobSource(
   const jobs::JobRepositoryPtr& in_jobRepository,
   const jobs::JobStatusNotifierPtr& in_jobStatusNotifier) :
   api::IJobSource(in_jobRepository, in_jobStatusNotifier),
   m_jobStatusWatcher(
         new MarsJobStatusWatcher(
                system::TimeDuration::Seconds(5),
                in_jobRepository,
                in_jobStatusNotifier))
{
}
```

in `MarsJobSource.cpp`.

4.12 TODO #12: Submit a Job

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

When a Job is submitted to the Plugin, the Plugin will be provided with an `Job` object which represents the details of the job. The Plugin must pass the necessary details of the `Job` object to the Job Scheduling System. In addition, the Plugin is responsible for making the following updates to the `Job` object:

- Set the ID of the Job. If the Job Scheduling System does not generate an ID, the Plugin must generate a unique ID for the Job and correctly associate it with the same Job in the Job Scheduling System.

- Set the `SubmissionTime` field of the `Job` object. This may be either the time that the Plugin invoked the Job Scheduling System's submission method or the time that the Job Scheduling System reports that the Job was submitted, if available.

The SDK will set the initial Job Status to `Job::State::PENDING` if no `Error` is returned from `IJobSource::submitJob`. Additional Job Status updates must be implemented by the Plugin. Ideally, this would be implemented through Job Status Streaming, which is discussed in detail in the 'Job Status Updates' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer's Guide.

When a Job is updated in the `JobStatusNotifier` for the first time, it gets added to the `AbstractJobRepository`. If the Plugin requires any special handling for newly added Jobs, or for Jobs that are being pruned, this functionality can be added in the concrete implementation of the `AbstractJobRepositry`. For more detail see the 'Customizing the Job Repository' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer's Guide.

4.12.1 Error Handling

The Plugin should also validate the `Job` object before submitting the Job to the Job Scheduling System. For example, the Job Scheduling System may only support the NFS mount type. If there are any Host mounts set on the `Job` it will not be rejected by the SDK layer, as the SDK has no knowledge of what mount types are supported by the Job Scheduling System. In that case, the Plugin should set the `out_wasInvalidRequest` parameter to `true`, since the user requested an unsupported mount type, and return an `Error` object. The `Error`'s category name, error code, and message will be returned to the Launcher to be sent back to the caller.

If the Job object was valid but it fails to launch for another reason the Plugin should not set `out_wasInvalidRequest`. The SDK layer sets the initial value of the variable to `false`.

4.12.2 Example

Recall from Example #7 that the Mars scheduling system supports containers, queues, resource limits (CPU, Memory, and GPU), a way to select the

processor type (`x86` or `ARM`), and a way to select which specific host(s) may run the job. Assume that the Mars job scheduling system also supports mounting an NFS path on job start-up. Finally, assume that the `mars_api` exposes a function with the following signature:

```
mars_api::job submit_job(
   const char* exe,
   const char** args,
   size_t argc,
   const char* stdin,
   mars_api::job_opts opts);
```

Where `mars_api::job_opts` is defined as follows:

```
enum proc_type
{
  PROC_TYPE_X86,
  PROC_TYPE_ARM,
  PROC_TYPE_ANY
};

struct mount
{
   std::string source;
   std::string dest;
   std::string host;
   bool read_only;
};

struct job_opts
{
   job_opts() :
     max_gpus(0),
     min_gpus(0),
     max_cpus(0),
     min_cpus(0),
     max_mem_mb(0),
     min_mem_mb(0),
     processor_type(PROC_TYPE_ANY)
```

```
   {}

   std::string name;
   std::string comment;
   std::string stdout_file;
   std::string stderr_file;
   std::vector<std::string> queues;
   std::vector<std::string> hosts;
   std::vector<mount> mounts;
   int max_gpus;
   int min_gpus;
   int max_cpus;
   int min_cpus;
   int max_mem_mb;
   int min_mem_mb;
   proc_type processor_type;
};
```

The Orchid organization's Plugin developer knows from the Mars API documentation that a resource limit with a value of `0` means that there is no limit. The Plugin developer also knows that the comment field can be any UTF-8 text and does not have a maximum length. Then the developer might change

```
Error QuickStartJobSource::submitJob(api::JobPtr io_job, bool& out_wasInvalidRequest)
{
   // TODO #12: Submit and then update the job.
   out_wasInvalidRequest = true;
   return Error(
      "NotImplemented",
      1,
      "Method QuickStartJobSource::submitJob is not implemented.",
      ERROR_LOCATION);
}
```

in `QuickStartJobSource.cpp` to

```
void replaceSpecialChars(std::string& io_str)
{
   // Replace % first so that added %s won't also be replaced.
   std::regex_replace(io_str, std::regex("%"), "%p");
   std::regex_replace(io_str, std::regex("="), "%e");
   std::regex_replace(io_str, std::regex(","), "%c");
   std::regex_replace(io_str, std::regex(";"), "%s");
   std::regex_replace(io_str, std::regex("\n"), "%n");
   std::regex_replace(io_str, std::regex("\r"), "%r");
   std::regex_replace(io_str, std::regex("|"), "%b");
}

Error MarsJobSource::submitJob(api::JobPtr io_job, bool& out_wasInvalidRequest) const
{
   // Create the job_opts
   mars_api::job_opts opts;
   opts.name = io_job->Name;

   std::string stdInp = "#!/bin/sh\n"

   // Convert the mounts. If any are not NFS mounts, exit with an error.
   for (const auto& mount: io_job->Mounts)
   {
      if (!mount.NfsMountSource)
      {
         out_wasRequestInvalid = true;
         return Error("MarsPluginError", 1, "Mars Launcher Plugin only supports NFS M
      }

      // Because of the check above, .getValueOr is guaranteed to return a non-empty
      NfsMountSource mountSrc = mount.NfsSourcePath.getValueOr(NfsMountSource());

      mars_api::mount marsMount;
      marsMount.dest = mount.DestinationPath;
      marsMount.read_only = mount.IsReadOnly;
      marsMount.source = mountSrc.Path;
      marsMount.host = mountSrc.Host;

      opts.mounts.push_back(mount);
```

```
}

// Set up the resource limits. There shouldn't be any unsupported types, but
// if there are, exit with an error.
for (const auto& limit: io_job->ResourceLimits)
{
   // For each resource type, request exactly the amount set by the user.
   if (limit.ResourceType == ResourceLimit::Type::CPU_COUNT)
   {
      opts.min_cpus = std::stoi(limit.Value);
      opts.max_cpus = opts.min_cpus;
   }
   else if (limit.ResourceType == ResourceLimit::Type::MEMORY)
   {
      opts.min_mem_mb = std::stoi(limit.Value);
      opts.max_mem_mb = opts.min_mem_mb;
   }
   else if (limit.ResourceType == "GPUs")
   {
      opts.min_gpus = std::stoi(limit.Value);
      opts.max_gpus = opts.min_gpus;
   }
   else
   {
      out_wasRequestInvalid = true;
      return Error(
         "MarsPluginError",
         2,
         "Mars Launcher Plugin does not support the requested ResourceLimit type:
         ERROR_LOCATION);
   }
}

// Copy queues.
std::copy(io_job->Queues.begin(), io_job->Queues.end(), std::back_inserter(opts.qu

// Save the environment and the tags in the job comment.
std::string tagStr;
for (std::string tag: io_job->Tags)
```

```
{
   if (!tagStr.empty())
      tagStr.append(",");

   replaceSpecialChars(tag);
   tagStr.append(tag);
}

if (!tagStr.empty())
   opts.comment.append("tags=").append(tagStr).append(";");

// Also set up the export lines for the shell script that will be passed to
// stdin.
std::string envStr;
for (const auto& env: io_job->Environment)
{
   if (!envStr.empty())
     envStr.append(",");

   std::string key = env.first;
   std::string val = env.second;
   stdInp.append("export ").append(key).append("=").append(val).append("\n");

   replaceSpecialChars(key);
   replaceSpecialChars(val);
   envStr.append(key).append("|").append(val);
}

if (!envStr.empty())
   opts.comment.append("env=").append(envStr).append(";");

// Get the hosts
std::string hostsStr = io_job->getJobConfigValue("Hostname").getValueOr("");
std::sregex_token_iterator itr(hostStr.begin(), hostStr.end(), std::regex(","), -1
std::copy(itr, std::sregex_token_iterator(), std::back_inserter(opts.hosts));

// Get the processor type.
std::string procType = io_job->getJobConfigValue("Processor Type").getValueOr("");
if (procType == "x86")
```

```
    opts.processor_type = PROC_TYPE_X86;
  else if (procType == "ARM")
    opts.processor_type = PROC_TYPE_ARM;

  // Set the output files.
  opts.stdout_file = io_job->StandardOutFile;
  opts.stderr_file = io_job->StandardErrFile;

  // Add the command/exe and arguments to the script. Jobs with both or no
  // command/exe will be rejected by the SDK layer.
  if (!io_job->Command.empty())
     stdInp.append(system::process::shellEscape(io_job->Command));
  else
     stdInp.append(system::process::shellEscape(io_job->Exe));

  for (const std::string& arg: io_job->Arguments)
    stdInp.append(" ").append(system::process::shellEscape(arg));

  // Now add the real stdin, if any. Escape the here-document start with " to
  // ensure the stdin is interpreted literally.
  if (!io_job->StandardIn.empty())
    stdInp.append("<< \"EOF\"\n").append(io_job->StandardIn).append("\nEOF\n");

  LOCK_JOB(io_job)
  {
     // Now submit the job!
     io_job->SubmissionTime = system::DateTime(); // Set the submission time to now
     mars_api::job marsJob;
     try
     {
        marsJob = mars_api::submit_job("/bin/sh", { }, 0, stdInp.c_str(), opts);
     }
     catch(mars_api::mars_exception& e)
     {
        return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
     }

     // Success! Update the ID and status and then exit.
     // Set the job ID
```

```
      io_job->Id = marsJob.id;
      io_job->Status = marsStatusToStatus(marsJob.state);
   }
   END_LOCK_JOB

   return Success();
}
```

in `MarsJobSource.cpp`.

Notice that the Mars API did not provide a way to set environment variables for the job. To compensate, the Plugin developer had to wrap the requested command in a here-document which set the requested environment variables.

4.13 TODO #13: Create an Output Stream

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

When a request is made to stream job output, the SDK will retrieve the relevant `Job` object, validate the authorization of the user to access the specified Job, and then invoke `IJobSource::createOutputStream` with the appropriate parameters. The Plugin is then responsible for creating an `AbstractOuptutStream` object which will stream the correct type of job output. If the job output will always be written to a file on a shared location, the Plugin may simply construct a `FileOutputStream` object and return it to the SDK.

If there is need to skip some lines of output or in some way further process the Job output, but it can still be read from a file, it is possible to customize the behavior of the base `FileOutputStream` class by inheriting from it. For more details, please refer to the 'Customizing the File Output Stream' subsection of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer Guide.

If it is not possible to stream the output from a file, please refer to the 'Custom Output Streams' section of the 'Advanced Features' chapter of the RStudio Launcher Plugin SDK Developer Guide for details on how to create a completely custom output stream class.

4.13.1 Example

Suppose that the Mars Job Scheduling System job output is being written to a file and that the Mars Launcher Plugin will have a requirement that job output be written to a shared location. In that case the Plugin developer may decide to create a `FileOutputStream` object to stream the job output. They may change

```
Error QuickStartJobSource::createOutputStream(
   api::OutputType in_outputType,
   api::JobPtr in_job,
   api::AbstractOutputStream::OnOutput in_onOutput,
   api::AbstractOutputStream::OnComplete in_onComplete,
   api::AbstractOutputStream::OnError in_onError,
   api::OutputStreamPtr& out_outputStream)
{
   // TODO #13: Create an output stream.
   return Error(
      "NotImplemented",
      2,
      "Method QuickStartJobSource::createOutputStream is not implemented.",
      ERROR_LOCATION);
}
```

in `QuickStartJobSource.cpp` to

```
Error Mars::createOutputStream(
   api::OutputType in_outputType,
   api::JobPtr in_job,
   api::AbstractOutputStream::OnOutput in_onOutput,
   api::AbstractOutputStream::OnComplete in_onComplete,
   api::AbstractOutputStream::OnError in_onError,
   api::OutputStreamPtr& out_outputStream)
{
   out_outputStream.reset(new api::FileOutputStream(in_outputType, in_job, in_onOutpu
   return Success();
}
```

in `MarsJobSource.cpp`.

4.14 TODO #14: Get Network Information

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

Some use cases of the Launcher may require the ability to communicate with a remotely launched job. For example, when using the Launcher to run RStudio R Sessions, the RStudio Server needs to be able to communicate with the R Session over TCP. To facilitate this, the Plugin must be able to provide network information for each Job.

The Plugin developer should implement `IJobSource::getNetworkInfo` so that it populates the `out_networkInfo` object with the hostname and IP addresses of the host running the specified Job.

Note that the `in_job` object provided to the `IJobSource::getNetworkInfo` should already have the `Host` field populated because a Job must have been successfully submitted to the Job Scheduling System in order for a network info request to be valid.

4.14.1 Example

Assume that the Plugin developer set the `Host` field on the Job object to the name of the Mars node running the Job, and that the name of the Mars node may or may not be the actual hostname of the machine running the job. Also assume that there is a `mars_api` method that returns information about a node with the following signature:

```
struct node
{
  std::string hostname;
  std::vector<std::string> addresses;
  int avail_cpus;
  int avail_mem_mb;
  int avail_gpus;
  proc_type processor_type;
}

mars_api::node node_info(const char* node_name);
```

In that case, the developer might change

```
Error QuickStartJobSource::getNetworkInfo(api::JobPtr in_job, api::NetworkInfo& out_n
{
   // TODO #14: Get the network information of the specified job.
   return Success();
}
```

in `QuickStartJobSource.cpp` to

```
Error MarsJobSource::getNetworkInfo(api::JobPtr in_job, api::NetworkInfo& out_network
{
   mars_api::node nodeInfo;
   try
   {
      nodeInfo = mars_api::node_info(in_job->Host.c_str());
   }
   catch(mars_api::mars_exception& e)
   {
      return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
   }

   out_networkInfo.Hostname = nodeInfo.hostname;
   std::copy(nodeInfo.addresses.begin(), nodeInfo.addresses.end(), std::back_inserter

   return Success();
}
```

in `MarsJobSource.cpp`

4.15 TODO #15: Control Jobs

TODO Location	Impact
QuickStartJobSource.cpp	QuickStartJobSource.cpp

When a request is made to control the state of a Job, the SDK will first validate the state of the job and then invoke the appropriate method on

`IJobSource`. If the Job is not in an appropriate state for the control operation that was requested, the SDK will return an error to the Launcher without invoking any methods on the `IJobSource` object.

The following table describes the possible control job operations, the method that will be invoked for each, the equivalent POSIX signal, and the list of states that a Job can be in for that operation to be valid.

Operation	`IJobSource` method	Equivalent Posix Signal	Valid Job States
Cancel	`cancelJob`	N/A	`PENDING`
Kill	`killJob`	`SIGKILL`	`RUNNING`
Resume	`resumeJob`	`SIGCONT`	`SUSPENDED`
Stop	`stopJob`	`SIGTERM`	`RUNNING`
Suspend	`suspendJob`	`SIGSTOP`	`RUNNING`

The Plugin is not required to support any of these operations. For any of the methods that the Plugin does not support, the Plugin should return `false`. If the Plugin returns `false` for one of these methods, the SDK will provide a default `"Operation not supported"` error message. However, the Plugin developer may choose to provide a more detailed error message by setting the `out_statusMessage` variable.

If the Plugin does support a given operation, the Plugin should return `true` from the method. The Plugin can indicate the success of the operation via the `out_isComplete` parameter by setting it to `true` on a successful operation and `false` on a failed operation. In any case, the Plugin may provide more details about the success or failure of the operation by setting the `out_statusMessage` variable.

Note that the SDK will acquire the `JobLock` of the relevant `Job` object before invoking the appropriate `IJobSource` method, so the Plugin implementation is free to modify the state of the job without acquiring the `JobLock` itself. The Plugin should modify the state of the `Job` when the operation is finished. In some cases, that may mean immediately modifying the state of the Job after the operation is invoked. In some cases, that may mean updating the state of the Job when the change is reflected in the Job Scheduling System.

4.15.1 Example

Assume that the `mars_api` provides a way to cancel a job via the function `int cancel_job(const char* id)`, a way to suspend a running job via the function `int suspend_job(const char* id)`, a way to resume a suspended job via `int resume_job(const char* id)`, and a way to stop a job via `int stop_job(const char* id, bool force = false)`. In that case, the developer might change

```
bool QuickStartJobSource::cancelJob(api::JobPtr in_job, bool& out_isComplete, std::st
{
   // TODO #15: Cancel a pending job.
   out_isComplete = false;
   out_statusMessage = "Cancel job is not supported.";
   return false;
}

// ...

bool QuickStartJobSource::killJob(api::JobPtr in_job, bool& out_isComplete, std::stri
{
   // TODO #15: Kill a running job.
   out_isComplete = false;
   out_statusMessage = "Kill job is not supported.";
   return false;
}

bool QuickStartJobSource::resumeJob(api::JobPtr in_job, bool& out_isComplete, std::st
{
   // TODO #15: Resume a suspended job.
   out_isComplete = false;
   out_statusMessage = "Resume job is not supported.";
   return false;
}

bool QuickStartJobSource::stopJob(api::JobPtr in_job, bool& out_isComplete, std::stri
{
   // TODO #15: Stop a running job.
   out_isComplete = false;
   out_statusMessage = "Stop job is not supported.";
```

```
   return false;
}

bool QuickStartJobSource::suspendJob(api::JobPtr in_job, bool& out_isComplete, std::s
{
   // TODO #15: Suspend a running job.
   out_isComplete = false;
   out_statusMessage = "Suspend job is not supported.";
   return false;
}
```

in QuickStartJobSource.cpp to

```
bool MarsJobSource::cancelJob(api::JobPtr in_job, bool& out_isComplete, std::string&
{
   int ret = -1;
   try
   {
      ret = mars_api::cancel_job(in_job->Id);
      out_isComplete = (ret == 0);

      // The job status watcher will update the job status to CANCELED when it shows
      if (!out_isComplete)
        out_statusMessage = "Cancel job " + in_job->Id + " failed with code " + std::
   }
   catch (const mars_api::mars_exception& e)
   {
      out_isComplete = false;
      out_statusMessage = "Cancel job " + in_job->Id + " failed " + e.what();
   }

   return true;
}

// ...

bool MarsJobSource::killJob(api::JobPtr in_job, bool& out_isComplete, std::string& ou
{
   int ret = -1;
```

```
   try
   {
      // Stop the job forcefully.
      ret = mars_api::stop_job(in_job->Id, true);
      out_isComplete = (ret == 0);

      // Set the job status to KILLED so that it doesn't incorrectly show as FINISHED
      if (out_isComplete)
        in_job->Status = api::Job::State::KILLED;
      else
        out_statusMessage = "Kill job " + in_job->Id + " failed with code " + std::to
   }
   catch (const mars_api::mars_exception& e)
   {
      out_isComplete = false;
      out_statusMessage = "Kill job " + in_job->Id + " failed " + e.what();
   }

   return true;
}

bool MarsJobSource::resumeJob(api::JobPtr in_job, bool& out_isComplete, std::string&
{
   int ret = -1;
   try
   {
      ret = mars_api::resume_job(in_job->Id);
      out_isComplete = (ret == 0);

      // The job status watcher will update the job status to RUNNING when it shows a
      if (!out_isComplete)
        out_statusMessage = "Resume job " + in_job->Id + " failed with code " + std::
   }
   catch (const mars_api::mars_exception& e)
   {
      out_isComplete = false;
      out_statusMessage = "Resume job " + in_job->Id + " failed " + e.what();
   }
```

```
   return true;
}

bool MarsJobSource::stopJob(api::JobPtr in_job, bool& out_isComplete, std::string& ou
{
   int ret = -1;
   try
   {
      ret = mars_api::stop_job(in_job->Id);
      out_isComplete = (ret == 0);

      // The job status watcher will update the job status to FINISHED when it shows
      if (!out_isComplete)
        out_statusMessage = "Stop job " + in_job->Id + " failed with code " + std::to
   }
   catch (const mars_api::mars_exception& e)
   {
      out_isComplete = false;
      out_statusMessage = "Stop job " + in_job->Id + " failed " + e.what();
   }

   return true;
}

bool MarsJobSource::suspendJob(api::JobPtr in_job, bool& out_isComplete, std::string&
{
   int ret = -1;
   try
   {
      ret = mars_api::suspend_job(in_job->Id);
      out_isComplete = (ret == 0);

      // The job status watcher will update the job status to SUSPENDED when it shows
      if (!out_isComplete)
        out_statusMessage = "Suspend job " + in_job->Id + " failed with code " + std:
   }
   catch (const mars_api::mars_exception& e)
   {
      out_isComplete = false;
```

```
      out_statusMessage = "Suspend job " + in_job->Id + " failed " + e.what();
   }

   return true;
}
```

in `MarsJobSource.cpp`.

4.16 TODO #16: Create a Resource Utilization Stream

TODO Location	Impact
QuickStartResourceStream.cpp	QuickStartResourceStream.cpp, QuickStartResourceStream.hpp

When a request is made to stream resource utilization metrics for a Job, the SDK will retrieve the relevant `Job` object, validate the authorization of the user to access the specified Job, and then invoke `IJobSource::createResourceStream` with the appropriate parameters. The Plugin is then responsible for creating an `AbstractResourceStream` object which will stream any resource utilization metrics that it is able to access. Supported resources are CPU Percentage, elapsed CPU Time in seconds, Resident Set size (or Physical Memory size) in MB, and Virtual Memory size in MB.

Streaming resource utilization metrics is a best effort operation. If any of the resource utilization metrics cannot be retrieved through the Job Scheduling System, the Plugin may simply omit them from each response. If it is not possible to retrieve any resource utilization metrics through the Job Scheduling System, the Plugin may simply not post any updates. When the Job enters a completed state, the SDK will send a response to the Launcher to indicate that the stream has completed. The provided `QuickStartResourceStream` class already fulfills that requirement for a Job Scheduling System that does not support resource utilization measurement.

The provided sample Local Launcher Plugin implements resource utilization by inheriting from `AbstractTimedResourceStream` because the data must be polled from the Operating System periodically. The example below will show how the `AbstractResourceStream` might be inherited when

it is possible to stream the necessary information from the Job Scheduling System.

4.16.1 Example

Assume that the `mars_api` exposes a function with the following signature:

```
struct stream_handle;

std::shared_ptr<stream_handle> job_stat_stream(
   const char* id,
   int freq,
   std::function<void(int, const mars_api::job_stat*)> on_update);
```

Where `id` is the ID of the job for which to stream job resource statistics, `freq` is the frequency (in seconds) at which updates should be posted, and `on_update` is the function that will be called every `freq` seconds. The return value of `mars_api::job_stat_stream` is an opaque stream handle that is owned by the caller. The stream will stop if the handle is destroyed.

The first parameter of the `on_update` function is an error code (`0` on success) and the second parameter is the Job resource utilization statistics. Like all `mars_api` functions, `mars_api::job_stat_stream` may throw a `mars_api::mars_exception`. Additionally, `mars_api::job_stat` is defined as follows:

```
struct job_stat
{
  uint64_t rs_size; // resident set size (bytes)
  uint64_t vm_size; // virtual memory size (bytes)
  uint64_t job_seconds; // total run time of the job (seconds)
  uint64_t cpu_seconds; // total CPU usage time of the job (seconds)
  double cpu_freq; // CPU frequency of the job (kHz)
  uint64_t disk_read; // total bytes read from disk by the job
  uint64_t disk_write; // total bytes written to disk by the job
  int node_count; // number of compute nodes in use by the job
  int num_cpus; // number of CPU cores allocated to the job
};
```

In that case, the Plugin the developer might change

```
namespace rstudio {
namespace launcher_plugins {
namespace quickstart {

class QuickStartResourceStream : public api::AbstractResourceStream
{
public:
   /**
    * @brief Constructor.
    *
    * @param in_job                    The job for which resource utilization metric
    * @param in_launcherCommunicator   The communicator through which messages may b
    */
   QuickStartResourceStream(
      const api::ConstJobPtr& in_job,
      comms::AbstractLauncherCommunicatorPtr in_launcherCommunicator);

   /**
    * @brief Initializes the resource utilization stream.
    *
    * @return Success if resource utilization streaming was started correctly; Error
    */
   Error initialize() override;
};

} // namespace quickstart
} // namespace launcher_plugins
} // namespace rstudio
```

in QuickStartResourceStream.hpp to

```
namespace orchid {
namespace mars {

#include <memory>

class MarsResourceStream :
   public api::AbstractResourceStream,
   public std::enable_shared_from_this<MarsResourceStream>
```

```
{
public:
   /**
    * @brief Constructor.
    *
    * @param in_job                    The job for which resource utilization metric
    * @param in_launcherCommunicator   The communicator through which messages may b
    */
   MarsResourceStream(
      const api::ConstJobPtr& in_job,
      comms::AbstractLauncherCommunicatorPtr in_launcherCommunicator);

   /**
    * @brief Initializes the resource utilization stream.
    *
    * @return Success if resource utilization streaming was started correctly; Error
    */
   Error initialize() override;

private:
   typedef std::shared_ptr<MarsResourceStream> SharedThis;
   typedef std::weak_ptr<MarsResourceStream> WeakThis;

   // An opaque handle to the Job statistics stream.
   std::shared_ptr<mars_api::stream_handle> m_streamHandle;

   // The last measured number of CPU seconds.
   int m_lastCpuSeconds;

   // The last measured number of Job seconds.
   int m_lastJobSeconds;

   // Calculates CPU percent from the CPU and Job time.
   double calcCpuPercent(int in_cpuSeconds, int in_jobSeconds);

   // Callback function when Job statistic data is available.
   static void onData(
      WeakThis in_weakThis,
      bool in_success,
```

```
      const mars_api::job_stat* in_stats);
};

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

in MarsResourceStream.hpp, and

```
#include <QuickStartResourceStream.hpp>

namespace rstudio {
namespace launcher_plugins {
namespace quickstart {

QuickStartResourceStream::QuickStartResourceStream(
   const api::ConstJobPtr& in_job,
   comms::AbstractLauncherCommunicatorPtr in_launcherCommunicator) :
      api::AbstractResourceStream(in_job, in_launcherCommunicator)
{
}

Error QuickStartResourceStream::initialize()
{
   // TODO #16: Create a resource utilization stream.
   return Success();
}

} // namespace quickstart
} // namespace launcher_plugins
} // namespace rstudio
```

in QuickStartResourceStream.cpp to

```
#include <MarsResourceStream.hpp>

#include <Error.hpp>

namespace orchid {
```

```
namespace mars {

MarsResourceStream::MarsResourceStream(
   const api::ConstJobPtr& in_job,
   comms::AbstractLauncherCommunicatorPtr in_launcherCommunicator) :
      api::AbstractResourceStream(in_job, in_launcherCommunicator),
      m_lastCpuSeconds(0),
      m_lastJobSeconds(0)
{
}

Error MarsResourceStream::initialize()
{
   try
   {
      m_streamHandle = mars_api::job_stat_stream(
         m_job->Id.c_str(),
         3, // Get updates reasonably often, but don't use too many resources
         std::bind(&MarsResourceStream::onData, weak_from_this(), std::placeholders::
   }
   catch (const mars_api::mars_exception& e)
   {
      return Error("MarsApiError", e.code(), e.what(), ERROR_LOCATION);
   }

   return Success();
}

double MarsResourceStream::calcCpuPercent(int in_cpuSeconds, int in_jobSeconds)
{
   // Calculate CPU percent by dividing the change in the CPU seconds by the change i
   double percent = -1.0;

   // Avoid diving by zero.
   if (in_jobSeconds > m_lastJobSeconds)
   {
      double cpuChange = in_cpuSeconds - m_lastCpuSeconds;
      double jobChange = in_jobSecodns - m_lastJobSeconds;
```

```
      percent = (cpuChange / jobChange) * 100.0;

      m_lastCpuSeconds = in_cpuSeconds;
      m_lastJobSeconds = in_jobSeconds;
   }

   return percent;
}

static void MarsResourceStream::onData(
   WeakThis in_weakThis,
   int in_errorCode,
   const mars_api::job_stat* in_stats)
{
   if (SharedThis sharedThis = in_weakThis.lock())
   {
      // Handle the error case first.
      if (in_errorCode != 0)
      {
         logging::logWarningMessage(
            "Failed to retrieve Job resource utilization data from Mars with error co
            ERROR_LOCATION);
      }

      // Otherwise populate the resource utilization data and post it.
      rstudio::launcher_plugins::api::ResourceUtilData data;

      double percent = sharedThis->calcCpuPercent(in_stats->cpu_seconds, in_stats->jo
      if (percent >= 0.0)
        data.CpuPercent = percent;

      data.CpuSeconds = in_stats->cpuSeconds;

      // Convert memory from B to MB.
      data.VirtualMem = in_stats->vm_size / (1024 * 1024);
      data.ResidentMem = in_stats->rs_size / (1024 * 1024);

      sharedThis->reportData(data);
   }
```

```
}

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