Octopus

Author: Jason Maassen (J.Maassen@esciencecenter.nl)

Version: Userguide v1.0, Octopus v1.0

Last modified: 10 September 2013

Copyrights & Disclaimers

For more information on the Netherlands eScience Center see:

http://www.esciencecenter.nl

The octopus project web site can be found at:

https://github.com/NLeSC/octopus.

Third party libraries

This product includes the SLF4J library, which is Copyright (c) 2004-2013 QOS.ch See "notices/LICENSE.slf4j.txt" for the licence information of the SLF4J library.

This product includes the JSch library, which is Copyright (c) 2002-2012 Atsuhiko Yamanaka, JCraft,Inc. See "notices/LICENSE.jsch.txt" for the licence information of the JSch library.

This product includes the Logback library, which is Copyright (c) 1999-2012, QOS.ch. See "notices/LICENSE.logback.txt" for the licence information of the Logback library.

This product includes the JaCoCo library, which is Copyright (c) 2009, 2013 Mountainminds GmbH & Co. KG and Contributors. See "notices/LICENSE.jacoco.txt" for the licence information of the JaCoCo library.

This project includes the JUnit library. See "notices/LICENSE.junit.txt" for the licence information of the JUnit library.

This project includes the Mockito library, which is Copyright (c) 2007 Mockito contributors. See "notices/LICENSE.mockito.txt" for the licence information of the Mockito library.

What is it?

Octopus is a middleware abstraction library. It provides a simple Java programming interface to various pieces of software that can be used to access distributed compute and storage resources.

Why Octopus?

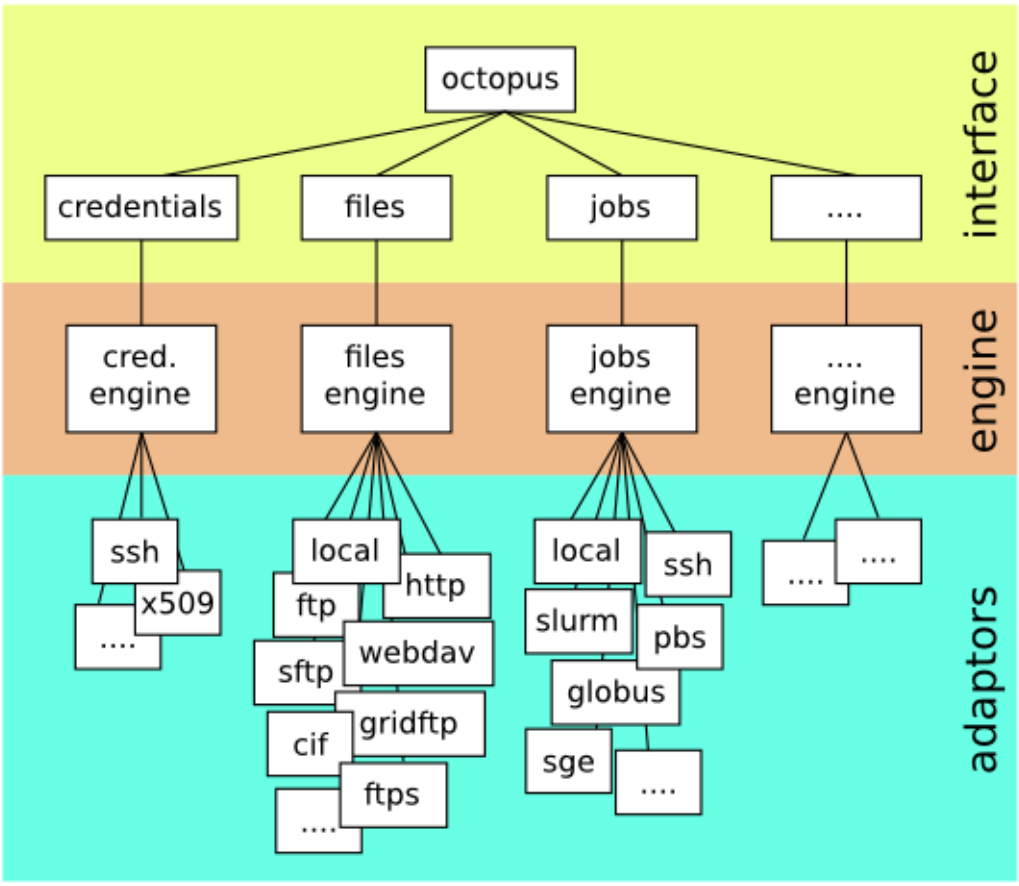
Octopus is developed by the Netherlands eScience Center as a support library for our projects. Several projects develop end-user applications that require access to distributed compute and storage resources. Octopus provides a simple API to access those resources, allowing those applications to be developed more rapidly. The experience gained during the development of these end-user applications is used to improve the Octopus API and implementation.

Installation

The installation procedure and dependencies of the octopus library can be found in the file "INSTALL.md" in the octopus distribution.

Design

Octopus is designed with extensibility in mind. It uses a modular and layer design as shown in the figure below:



Octopus consists of three layers, an *interface layer*, an *engine layer* and an *adaptor layer*.

The *interface layer* is used by the application using octopus. It contains several specialized interfaces:

- Octopus: this is the main entry point used to retrieve the other interfaces.
- Files: contains functionality related to files, e.g., creation, deletion, copying, reading, writing, obtaining directory listings, etc.
- Jobs: contains functionality related to job submission, e.g., submitting, polling status, cancelling, etc.
- Credentials: contains functionality related to credentials. Credentials (such as a username password combination) are often needed to gain access to files or to submit jobs.

The modular design of octopus allows us to add additional interfaces in later versions, e.g., a Clouds interface to manage virtual machines, or a Networks interface to manage bandwidth-on-demand networks.

The *adaptor layer* contains the adaptors for the each of the middlewares that octopus supports. An *adaptor* offers a middleware specific implementation for the functionality offered by one of the interfaces in octopus.

For example, an adaptor may provide an *sftp* specific implementation of the functions in the octopus *file interface* (such as *copy* or *delete*) by translating each of these functions to *sftp* specific code and commands.

For each interface in octopus there may be multiple adaptors translating its functionality to different middlewares. To distinguises between these adaptors octopus uses the *scheme* they support, such as "sftp", "http" or "ssh". There can be only one adaptor for each scheme.

The *engine layer* of octopus contains the "glue" that connects each interface to the adaptors that implement its functionality. When a function of the interface layer is invoked, the call will be forwarded to the engine layer. It is then the responsibility of the engine layer to forward this call to the right adaptor.

To perform this selection, the engine layer matches the *scheme* of the object on which the operation needs to be performed, to the *schemes* supported by each of the adaptors. When the schemes match, the adaptor is selected.

Interfaces and datatypes

This section will briefly explain each of the interfaces and related datatypes. Detailed information about Octopus can be found in the online Javadoc at:

http://nlesc.github.io/octopus/javadoc/

Package Structure

The octopus API uses the following package structure:

- `nl.esciencecenter.octopus` Entry point into octopus.
- `nl.esciencecenter.octopus.credentials` Credential interface.
- `nl.esciencecenter.octopus.files` Files interface.
- `nl.esciencecenter.octopus.jobs` Jobs interface.
- `nl.esciencecenter.octopus.exeptions` Exceptions used in octopus.
- `nl.esciencecenter.octopus.util` Various utilty classes.

We will now briefly describe the most important classes and interfaces of these packages.

Octopus factory and interface

The `nl.esciencecenter.octopus` package contains the entry point into the octopus library: **OctopusFactory**

```
public class OctopusFactory {
   public static Octopus newOctopus(Map<String,String> properties)
   public static void endOctopus(Octopus octopus)
   public static void endAll()
}
```

The **newOctopus** method can be used to create a new octopus instance, while the **endOctopus** method can be used to release the octopus instance once it is no longer needed. It is important to end the octopus when it is no longer needed, as this allows it to release any resources it has obtained.

When creating an octopus using **newOctopus**, the *properties* parameter can be used to configure the octopus instance. If no configuration is necessary, `null` can be used. Properties consist of a set of key-value pairs. In octopus all keys **must** start with "octopus.". To configure the adaptors, properties of the form "octopus.adaptors.(name).(property)" can be used, where "(name)" is the name of the adaptor (for example "local" or "ssh") and "(property)" is the name of the property to be configured. Note that this name can be futher qualified, for example "octopus.adaptors.local.a.b.c". The available properties can be found in the documentation of the individual adaptors (see Appendix A).

A call to **newOctopus** will return an **Octopus**:

```
public interface Octopus {
    Files files()
    Jobs jobs()
    Credentials credentials()
    Map<String,String> getProperties()
    AdaptorStatus getAdaptorStatus(String adaptorName)
    AdaptorStatus[] getAdaptorStatuses()
}
```

The **files**, **jobs** and **credentials** methods in this interface can be used to retrieve various interfaces that the octopus library offers. They will be described in more detail below.

The **getProperties** method can be used to retrieve the properties used when the octopus was created. Most objects created by octopus contain such a **getProperties** method. For brevity, we will not explain these further.

The **getAdaptorStatus** method can be used to retrieve information about the adaptors. This information is returned in an **AdaptorStatus**:

```
public interface AdaptorStatus {
    String getName()
    String getDescription()
    String[] getSupportedSchemes()
    OctopusPropertyDescription[] getSupportedProperties()
    Map<String, String> getAdaptorSpecificInformation()
}
```

An **AdaptorStatus** contains **getName** to retrieve the name of an adaptor, **getDescription** to get a human readable description of what functionality it has to offer and **getSupportedSchemes** to retrieve a list of the schemes it supports.

The **getSupportedProperties** method can be used to retrieve a list of configuration options the adaptor supports. Each returned **OctopusPropertyDescription** gives a full description of a single property, including its name (of the form "octopus.adaptors.(name).(property)"), the expected type of its value, a human readable description of its purpose, etc. More information on the supported properties can be found in Appendix A.

Finally, **getAdaptorSpecificInformation** can be used to retrieve status information from the adaptor. Each key contains a property of the form described above.

Credentials interface

The `nl.esciencecenter.octopus.credentials` package contains the **Credentials** interface of octopus:

```
public interface Credentials {
    Credential newCertificateCredential(String scheme, String keyfile,
        String certfile, String username, char [] password,
        Map<String,String> properties)

    Credential newPasswordCredential(String scheme, String username,
        char [] password, Map<String,String> properties)

    Credential getDefaultCredential(String scheme)
    void close(Credential credential)
}
```

The **Credentials** interface contains various methods for creating credentials, based on certificates or passwords. For each method, the desired *scheme* needs to be provided as a parameter (for example, "ssh" or "sftp"). This allows octopus to forward the call to the correct adaptor. Note that some types of credentials may not be supported by all adaptors. An exception will be thrown when an unsupported **new**Credential** methods is invoked.

Additional configuration can also be provides using the *properties* parameter, which use the same form as described in the *Octopus factory and interface* section above. If no additional configuration is needed, `null` can be used. The **getDefaultCredential** method returns the default credential for the given scheme. All adaptors are guarenteed to support this method.

All **new**Credential** methods return a **Credential** that contains the following methods:

```
public interface Credential {
   String getAdaptorName()
   Map<String,String> getProperties()
}
```

The **getAdaptorName** method can be used to retrieve the name of the adaptor that created the credential. Many adaptor specific objects returned by octopus contain this method. For brevity we will not explain this further.

When a **Credential** is no longer used, it **must** be closed using **close**. This releases any resources held by the **Credential**. The **isOpen** method can be used to check if a **Credential** is open or closed.

Files interface

The `nl.esciencecenter.octopus.files` package contains the **Files** interface of octopus. For readability we will split the explanation of **Files** into several parts:

```
public interface Files {
   FileSystem newFileSystem(String scheme, String location,
            Credential credential, Map<String,String> properties)

   void close(FileSystem filesystem)
   boolean isOpen(FileSystem filesystem)
   // ... more follows
}
```

The **Files** interface contains several method for creating and closing a **FileSystem**. A **FileSystem** provides an abstraction for a (possibly remote) file system.

To create a **FileSystem** the **newFileSystem** method can be used. As before, the desired **scheme** must be provided as a parameter. In addition, the *location* parameter provides information on the location of the file system using an adaptor specific string. For local file systems, the location must contain the root of the file system to access, such as "/" on Linux or "C:" on Windows. For remote file systems, the location typically contains the host name of the machine to connect to. The exact format of accepted location strings can be found in the adaptor documentation.

The following are all valid combinations of file system schemes and locations:

```
"file"  "/"                       connect to the local file system on Linux
"file"  "C:"                      connect to the local C: drive on Windows
"sftp"  "example.com"             connect to example.com using sftp
"sftp"  "test@example.com:44"     connect to example.com using sftp on port 44
                                  with "test" as user name.
```

The **newFileSystem** method also has a *credential* parameter to provide the credential needed to access the file system. If this parameter is set to `null` the default credentials will be used for the scheme. The *properties* parameter can be used to provide additional configuration properties. Again, `null` can be used if no additional configuration is required. The returned **FileSystem** contains the following methods:

```
public interface FileSystem {
    // ...
    String getScheme()
    String getLocation()
    Path getEntryPath()
}
```

The **getScheme** and **getLocation** methods returns the scheme and location strings used to create the **FileSystem**. The **getEntryPath** method returns the *path at which the file system was entered*. For example, when accessing a file system using "sftp" it is customary (but not manditory) to enter the file system at the users' home directory. Therefore, the entry path of the **FileSystem** will be similar to "/home/(username)". For local file systems the entry path is typically set to the root of the file system (such as "/" or "C:").

When a **FileSystem** is no longer used, it **must** be closed using **close**. This releases any resources held by the **FileSystem**. The **isOpen** method can be used to check if a **FileSystem** is open or closed. Once a **FileSystem** is created, it can be used to access files:

```
public interface Files {
   Path newPath(FileSystem filesystem, RelativePath location)
   void createFile(Path path)
   void createDirectories(Path dir)
   void createDirectory(Path dir)
   boolean exists(Path path)
   void delete(Path path)
   FileAttributes getAttributes(Path path)
   // ... more follows
}
```

The **newPath** method can be used to create a new **Path**. An **Path** represents a path on a specific **FileSystem**. This path does not necessarily exists. To create an **Path**, both the target **FileSystem** and a **RelativePath** are needed. A **RelativePath** contains a sequence of strings separated using a special *separator* character, which is used to identify a location on a file system (for example "/tmp/dir"). **RelativePath** contains many utility methods for manipulating these string sequences. The details can be found in the Javadoc of **RelativePath**.

Files contains several methods to create and delete files and directories. When creating files and directories octopus checks if the target already exists. If so, an exception will be thrown. Similary, an exception is thrown when attempting to delete non-existing file or a directory that is not empty. The **exists** method can be used to check if a path exists.

Using the **getAttributes** method the attributes of a file can be retrieved. The returned **FileAttributes** contains information on the type of file (regular file, directory, link, etc), it size, creation time, access rights, etc.

To list directories, the following methods are available:

```
public interface Files {
   DirectoryStream<Path> newDirectoryStream(Path dir)
   DirectoryStream<PathAttributesPair>
         newAttributesDirectoryStream(Path dir)

   // ... more follows
}
```

Both **newDirectoryStream** and **newAttributesDirectoryStream** return a **DirectoryStream** which can be used to iterate over the contents of a directory. For the latter, the **FileAttributes** for each of the files are also included. alternatively, these methods are also available with an extra *filter* parameter, which can be used to filter the stream in advance.

To read or write files, the following methods are available:

```
public interface Files {
   InputStream newInputStream(Path path)
   OutputStream newOutputStream(Path path, OpenOption... options)
}
```

Using these methods, an **InputStream** can be created to read a file, and an **OutputStream** can be created to write a file. The **newOutputStream** method requires a *options* parameter to specify how the file should be opened for writing (for example, should the data be append or should the file be truncated first). These options are describe in more detail in the Javadoc.

To copy files, the following methods are available:

```
public interface Files {
   Copy copy(Path source, Path target, CopyOption... options)
   CopyStatus getCopyStatus(Copy copy)
   CopyStatus cancelCopy(Copy copy)
}
```

The **copy** method supports various copy operations such as a regular copy, a resume or an append. The *options* parameter can be used to specify the desired operation. Normally, **copy** performs its operation *synchronously*, that is, the call blocks until the copy is completed. However, *asynchronous* operations are also supported by providing the option **CopyOption.ASYNCHRONOUS**. In that case a **Copy** object is returned that can be used to retrieve the status of the copy (using **getCopyStatus**) or cancel it (using **cancelCopy**). The details of the available copy operations can be found in the Javadoc of **CopyOption**.

Jobs interface

The `nl.esciencecenter.octopus.job` package contains the **Jobs** interface of octopus. For readability we will split the explanation of **Jobs** into several parts:

```
public interface Jobs {

    Scheduler newScheduler(String scheme, String location,
       Credential credential, Map<String,String> properties)

    void close(Scheduler scheduler)
    boolean isOpen(Scheduler scheduler)
    // ... more follows
}
```

The **Jobs** interface contains the **newScheduler** method that can be used to create a **Scheduler**. A **Scheduler** provides an abstraction for a (possibly remote) scheduler that can be used to run jobs. The **newScheduler** method has **scheme** and **location** parameters that specify how to access the scheduler. As with **newFileSystem** the **location** is adaptor specific. To access the local scheduler, passing `null` or an empty string is sufficient. To access remote schedulers, the location typically contains the host name of the machine to connect to. The exact format of accepted location strings can be found in the adaptor documentation.

The following are valid examples of scheduler schemes and locations:

```
"local" ""                    the local scheduler
"ssh"   "example.com"         connect to a remote scheduler at example.com using SSH
"slurm" ""                    connect to a local slurm scheduler
"slurm" "test@example.com:44" connect to a remote slurm scheduler at example.com by
                              using SSH on port 44 with "test" as user name.
```

When a **Scheduler** is no longer used, is **must** be closed using the **close** method. The **isOpen** method can be use to check if a **Scheduler** is open or closed. A **Scheduler** contains the following:

```
public interface Scheduler {
    String[] getQueueNames()
    boolean isOnline()
    boolean supportsInteractive()
    boolean supportsBatch()
    // ...
}
```

Each **Scheduler** contains one or more queues to which jobs can be submitted. Each queue has a name that is unique to the **Scheduler**. The **getQueueNames** method can be used to retrieve all queue names.

The **isOnline** method can be used to determine if the **Scheduler** is an *online scheduler* or an *offline scheduler*. Online schedulers need to remain active for their jobs to run. Closing an online scheduler will kill all jobs that were submitted to it. Offline schedulers do not need to remains active for their jobs to run. A submitted job will typically be handed over to some external server that will manage the job for the rest of its lifetime.

The **supportsInteractive** and **supportsBatch** method can be use to check if the **Scheduler** supports interactive and/or batch jobs. Interactive jobs are jobs where the user gets direct control over the standard streams of the job (the *stdin*, *stdout* and *stderr* streams). The user **must** retrieve these streams using the **getStreams** method in **Jobs** and then provide input and output, or close the streams. Failing to do so may cause the job to block indefinately. Batch jobs are jobs where the standard streams are redirected from and to files. The location of these files must be set before the job is started, as will be explained below.

Once a **Scheduler** is created, **Jobs** contains several methods to retrieve information about the **Scheduler**:

```
public interface Jobs {
    String getDefaultQueueName(Scheduler scheduler)
    QueueStatus getQueueStatus(Scheduler scheduler, String queueName)
    QueueStatus[] getQueueStatuses(Scheduler scheduler, String...queueNames)
    Job[] getJobs(Scheduler scheduler, String... queueNames)
    // ... more follows
}
```

The **getQueueStatuses** method can be used to retrieve information about a queue. If no queue names are provided as a parameter, information on all queues in the scheduler will be returned. Using the **getDefaultQueueName** the default queue can be retrieved for the **Scheduler**. The **getJobs** method can be used to retrieve information on all jobs in a queue. Note that this may also include jobs from other users.

To submit and manage jobs, the **Jobs** interface contains the following methods:

```
public interface Jobs {
    Job submitJob(Scheduler scheduler, JobDescription description)
    Streams getStreams(Job job)
    JobStatus getJobStatus(Job job)
    JobStatus[] getJobStatuses(Job... jobs)
    JobStatus waitUntilRunning(Job job, long timeout)
    JobStatus waitUntilDone(Job job, long timeout)
    JobStatus cancelJob(Job job)
}
```

The **submitJob** method can be used to submit a job to a **Scheduler**. A **JobDescription** must be provided as parameter. A **JobDescription** contains all necessary information on how to start the job, for example, the location of the executable, any command line arguments that are required, the working directory, if the job is an interactive of batch job, the location of the files for stream redirection (in case of a batch job), etc. See the Javadoc for details of the **JobDescription**.

Once a job is submitted, a **Job** object is returned that can be used with **getJobStatus** to retrieve the status of the job, and with **cancelJob** to cancel it. This **Job** contains the following:

```
public interface Job {
    JobDescription getJobDescription()
    Scheduler getScheduler()
    String getIdentifier()
    boolean isInteractive()
    boolean isOnline()
}
```

Besides methods for retrieveing the **JobDescription** and **Scheduler** that created it, each **Job** also contains the **isInteractive** method to determine if the **Job** is interactive, and the **isOnline** method to determine if the job is running on an *online scheduler* (explained above).

After submitting a job, **waitUntilRunning** can be used to wait until a job is no longer waiting in the queue and **waitUntilDone** can be used to wait until the job has finished.

For all methods returning a **JobStatus**, the following rule applies: after a job has finished, the status is only guarenteed to be returned *once*. Any subsequent calls to a method that returns a **JobStatus** *may* throw an exception stating that the job does not exist. Some adaptors may return a result however.

Exceptions

The `nl.esciencecenter.octopus.exceptions` package contains the exceptions that may be thrown by octopus. See the Javadoc for the available exceptions.

Utilities classes

The `nl.esciencecenter.octopus.util` package contains various utility classes. The main entry points are **Utils**, **Sandbox** and **JavaJobDescription**.

In **Utils** various utility methods can be found that make it easier to use Octopus. Many methods provide simple shortcuts to often used code constructs. Some examples are shown below:

```
public class Utils {
    // Create a new local Scheduler.
    public static Scheduler getLocalScheduler(Jobs jobs)

    // Create a new Scheduler without Credentials or properties.
    public static Scheduler newScheduler(Jobs jobs, String scheme)

    // Create a Path that represents the home directory of the current user.
    public static Path getLocalHome(Files files)

    // Create a Path that represents the current working directory.
    public static Path getLocalCWD(Files files)

    // Convert a String containing a local path into a Path.
    public static Path fromLocalPath(Files files, String path)

    // Retrieve all local file systems.
    public static FileSystem [] getLocalFileSystems(Files files)

    //  Are we running on a Linux machine ?
    public static boolean isLinux()

    // Are we running on a Windows machine ?
    public static boolean isWindows()

    // Are we running on a OSX machine ?
    public static boolean isOSX()
}
```

In addition many methods are provided for reading data from files or streams to various output targets, writing data to files or streams from various input sources, recursive copying, recursive deletion, etc. See the Javadoc of **Utils** for details.

A **Sandbox** is a utility class that makes is it easier to create a (possibly remote) temporary directory and transfer files to and from this directory. A Sandbox is often used in when submitting jobs that require input files and / or produce output files. Sandbox contains the following methods:

```
public class Sandbox {
   Sandbox(Files files, Path root, String sandboxName)
   void addUploadFile(Path src, String dest)
   void addDownloadFile(String src, Path dest)
   void upload(CopyOption... options)
   void download(CopyOption... options)
   void delete()
}
```

Creating a Sandbox requires an octopus **Files** interface and a **root** directory. The Sandbox will then create a temporary directory **sandboxName** in **root**. If **sandboxName** is `null`, a random name will be generated. Using **addUploadFile** files can be added to the upload queue. These files will be transferred to the Sandbox directory when **upload** is invoked. Similarly, using **addDownloadFile**, files can be added to the download queue. They will be downloaded from the Sandbox directory when **download** is invoked. Finally, **delete** can be used to delete the Sandbox directory.

A **JavaJobDescription** is a utility class that makes is it easier to create a **JobDescription** for running a Java application. In addition to the command line arguments used by the application, Java applications typically require a number of *special* command line argument for the Java Virtual Machine (JVM), such as a *class path*, *system properties*, and *JVM options*.

The JavaJobDescription class extends the regular JobDescription with support for these additional arguments. When a Job a submitted to an octopus Scheduler that uses a JavaJobDescription, the various types of command line arguments will be merged automatically into a single arguments list. See the Javadoc of **JavaJobDescription** for details.

Examples

Examples of how to use octopus can be found in the examples directory. We will list the examples here in order of increasing complexity, and with a short description of each example.

Initializing Octopus

Creating an **Octopus**: CreatingOctopus.java

Creating an **Octopus** with configuration properties: CreatingOctopusWithProperties.java

Creating Credentials

Creating a password and default **Credential**: CreatingCredential.java

File Access

Creating a local **FileSystem**: CreateLocalFileSystem.java

Checking if a local file exists: LocalFileExists.java

Creating a **FileSystem** based on a URI. CreateFileSystem.java

Checking if a (possibly remote) file exists: FileExists.java

Listing a directory: DirectoryListing.java

Listing the attributes of a file: ShowFileAttributes.java

Copying a file: CopyFile.java

Job Submission

Creating a **Scheduler** and retrieving the status of its queues: ListQueueStatus.java

Creating a **Scheduler** and retrieving the jobs: ListJobs.java

Listing the status of a Job: ListJobStatus.java

Submitting a batch job without output: SubmitSimpleBatchJob.java

Submitting a batch job with output: SubmitBatchJobWithOutput.java

Submitting an interactive job with output: SubmitInteractiveJobWithOutput.java

Appendix A: Adaptor Documentation

This section contains the adaptor documentation which is generated from the information provided by the adaptors themselves.

Octopus currently supports 4 adaptors: local, ssh, gridengine, slurm.

Adaptor: local

The local adaptor implements all functionality with standard java classes such as java.lang.Process and java.nio.file.Files.

Supported schemes:

local, file

Supported locations:

(null), (empty string), “/”

Supported properties:

`octopus.adaptors.local.queue.pollingDelay`

The polling delay for monitoring running jobs (in milliseconds).

- Expected type: INTEGER
- Default value: 1000
- Valid for: [OCTOPUS]

`octopus.adaptors.local.queue.multi.maxConcurrentJobs`

The maximum number of concurrent jobs in the multiq..

- Expected type: INTEGER
- Default value: 4
- Valid for: [OCTOPUS]

Adaptor: ssh

The SSH adaptor implements all functionality with remove ssh servers.

Supported schemes:

ssh, sftp

Supported locations:

“[user@]host[:port]”

Supported properties:

`octopus.adaptors.ssh.autoAddHostKey`

Automatically add unknown host keys to known_hosts.

- Expected type: BOOLEAN
- Default value: true
- Valid for: [SCHEDULER, FILESYSTEM]

`octopus.adaptors.ssh.strictHostKeyChecking`

Enable strict host key checking.

- Expected type: BOOLEAN
- Default value: true
- Valid for: [SCHEDULER, FILESYSTEM]

`octopus.adaptors.ssh.loadKnownHosts`

Load the standard known_hosts file.

- Expected type: BOOLEAN
- Default value: true
- Valid for: [OCTOPUS]

octopus.adaptors.ssh.queue.pollingDelay

The polling delay for monitoring running jobs (in milliseconds).

- Expected type: LONG
- Default value: 1000
- Valid for: [SCHEDULER]

octopus.adaptors.ssh.queue.multi.maxConcurrentJobs

The maximum number of concurrent jobs in the multiq..

- Expected type: INTEGER
- Default value: 4
- Valid for: [SCHEDULER]

octopus.adaptors.ssh.gateway

The gateway machine used to create an SSH tunnel to the target.

- Expected type: STRING
- Default value: null
- Valid for: [SCHEDULER, FILESYSTEM]

Adaptor: gridengine

The SGE Adaptor submits jobs to a (Sun/Ocacle/Univa) Grid Engine scheduler. This adaptor uses either the local or the ssh adaptor to gain access to the scheduler machine.

Supported schemes:

ge, sge

Supported locations:

(all locations supported by local), (all locations supported by ssh)

Supported properties:

`octopus.adaptors.gridengine.ignore.version`

Skip version check is skipped when connecting to remote machines. WARNING: it is not recommended to use this setting in production environments!

- Expected type: BOOLEAN
- Default value: false
- Valid for: [SCHEDULER]

`octopus.adaptors.gridengine.accounting.grace.time`

Number of milliseconds a job is allowed to take going from the queue to the qacct output.

- Expected type: LONG
- Default value: 60000
- Valid for: [SCHEDULER]

`octopus.adaptors.gridengine.poll.delay`

Number of milliseconds between polling the status of a job.

- Expected type: LONG
- Default value: 1000
- Valid for: [SCHEDULER]

Adaptor: slurm

The Slurm Adaptor submits jobs to a Slurm scheduler. This adaptor uses either the local or the ssh adaptor to gain access to the scheduler machine.

Supported schemes:

slurm

Supported locations:

(all locations supported by local), (all locations supported by ssh)

Supported properties:

`octopus.adaptors.slurm.ignore.version`

Skip version check is skipped when connecting to remote machines. WARNING: it is not recommended to use this setting in production environments!

- Expected type: BOOLEAN
- Default value: false
- Valid for: [SCHEDULER]

`octopus.adaptors.slurm.disable.accounting.usage`

Do not used accounting info of slurm, even when available. Mostly for testing purposes

- Expected type: BOOLEAN
- Default value: false
- Valid for: [SCHEDULER]

`octopus.adaptors.slurm.poll.delay`

Number of milliseconds between polling the status of a job.

- Expected type: LONG
- Default value: 1000
- Valid for: [SCHEDULER]