XebiaLabs
the deployment automation company

Deployit Command Line Interface Manual

Version 3.9.0

Table of Contents

Preface

Deployit ships with a rich *Command Line Interface* (CLI) that makes it possible to control and administer most of its features. The aim of this manual is to describe and teach how to use the CLI.

Discovering middleware topology, setting up environments, importing packages and performing deployments are just some examples of what you can do from the CLI.

The CLI has been designed to allow it to be programmed by using the Python programming language. Python is a well-known scripting language and will be familiar to most system administrators and developers.

Throughout this manual, examples will show how to use the objects to perform various tasks when using the CLI.

The *Deployit Reference Manual* contains background information on Deployit, an overview of its features and a short explanation of basic deployment concepts. Together with this manual, it will present you with enough information to enable you to use the CLI to its full potential.

CLI examples

This manual contains example CLI snippets to illustrate how to use the Deployit CLI. These snippets are built using the *demo-plugin*, a sample plugin that shows some of Deployit's core features. The demo plugin can be downloaded from the Deployit community plugins website at GitHub:

https://github.com/xebialabs/community-plugins/downloads

Using the CLI

The Deployit CLI connects to the Deployit Server using the standard HTTP / HTTPS protocol. This makes it possible to use the CLI remotely without firewall issues.

Before you start using the Deployit CLI, make sure the Deplyit Server is running. Please refer to the *Deployit System Administrator Manual* for more information on starting the Deployit Server.

Environment variables

After installing the Deployit CLI, it's good practice to set an environment variable named `DEPLOYIT_CLI_HOME` which points to the root directory where the CLI has been installed. In the remainder of this manual, the `DEPLOYIT_CLI_HOME` environment variable will be assumed to be set and to refer to the Deployit CLI installation directory.

A second environment variable, `DEPLOYIT_CLI_OPTS`, can be used to provide JVM options for the Deployit CLI process. For example, to set the initial Java heap size to 512 megabytes of memory, and the maximum Java heap size to 2 gigabytes of memory, the environment variable would be set as follows:

- Unix platforms : `export DEPLOYIT_CLI_OPTIONS="-Xms512m -Xmx2g"`
- Windows platforms : `set DEPLOYIT_CLI_OPTIONS="-Xms512m -Xmx2g"`

If not set, the CLI startup scripts provide sensible defaults.

Starting the Deployit CLI

To access the Deployit CLI, open a terminal window or command shell and change to the `DEPLOYIT_CLI_HOME/bin` directory.

The Deployit CLI can be started by entering the command `./cli.sh` on Unix, or `cli.cmd` on Windows.

When the CLI starts, it will prompt the user for a `username` and `password` combination and once these have been entered, it will attempt to connect to the Deployit Server on `localhost` running on Deployit's standard port of `4516`.

Once connected to the Deployit Server, you can start sending commands. Note that access restrictions may apply, see the section on security below.

Starting up with different settings

By starting the CLI with the `-h` flag, the following message is shown which lists all possible options that may be used when starting up the CLI. Whenever `./cli.sh` is mentioned, it's understood to be read as `cli.cmd` on Windows platforms:

```
./cli.sh [options] [[--] arguments]

options:

-configuration VAL : Specify the location of the configuration file
-context VAL       : Context url Deployit is running at, defaults to ''
-f (-source) VAL   : Execute a specified python source file
-help              : Prints this usage message
-host VAL          : Connect to a specified host, defaults to 127.0.0.1
-password VAL      : Connect with the specified password
-port N            : Connect to a specified port, defaults to 4516
-q (-quiet)        : Suppresses the welcome banner
-secure            : Use https to connect to the Deployit server
-username VAL      : Connect as the specified user
```

Following is a short explanation on how to use the available options:

- `-configuration config_file`

  This option is used to pass the location of the Deployit CLI configuration file. This properties file supports the following options: * `cli.username` - Connect as the specified user * `cli.password` - Connect with the specified password.

- `-context newcontext`

  If provided, the `context` value will be added top the Deployit Server connection URL. For example, if 'newcontext' is given, the CLI will attempt to connect to the Deployit Server REST API at `http://host:port/newcontext/deployit`. Note that the leading slash and REST API endpoint ('deployit') will automatically be added if they are omitted from the parameter.

- `-f Python_script_file`

  Starts the CLI in batch mode to run the provided Python file. Once the script completes, the CLI will terminate.

  Note: the Deployit CLI can load and run Python script files of up to 100K in size.

- `-source Python_script_file`

  Alternative for the `-f` option.

- `-host myhost.domain.com`

  Specifies the host the Deployit Server is running on. The default host is `127.0.0.1`, i.e. `localhost`.

- `-port 1234`

  Specifies the port at which to connect to the Deployit Server. If the port is not specified, it will default to Deployit's default port `4516`.

- `-secure`

  This will make the CLI connect to the Deployit Server using HTTPS. By default it will connect to the secure port 4517, unless another port is specified with the `-port` option. In order to be able to connect, the Deployit Server must have been started using this secured port (on by the default).

- `-username myusername`

Specifies the username to be used for login. If the username is not specified, the CLI will enter interactive mode and prompt the user.

- `-password mypassword`

  Specifies the password to be used for login. If the password is not specified, the CLI will enter interactive mode and prompt the user.

- `-q`

  Suppresses display of the welcome banner.

- `-quiet`

  Alternative for the `-q` option.

One example of using options might be:

```
./cli.sh -username User -password UserPassword -host deployit.local
```

This will connect the CLI as 'User' with password 'UserPassword' to the Deployit Server running on the host `deployit.local` and listening on port 4516.

Passing arguments to CLI commands or script

After the options it's also possible to pass arguments from the command line to the CLI. It's not mandatory to specify any options in order to pass arguments.

Here's an example of passing arguments, without specifying options:

```
./cli.sh these are four arguments
```

and with options:

```
./cli.sh -username User -port 8443 -secure again with four arguments
```

It is possible to begin an argument with the character '-'. In order for the CLI not to try to interpret it as an option instead of an argument, use the '`--`' seperator between the option list and the argument list:

```
./cli.sh -username User -- -some-argument there are six arguments -one
```

This seperator only needs to be used in case one or more of the arguments begin with '-'.

The arguments may now be used in commands given on the CLI or be used in a script passed with the `-f` option, by using the `sys.argv[index]` method, whereby the index runs from 0 to the number of arguments. Index 0 of the array contains the name of the passed script, or is empty when the CLI was started in interactive mode. The first argument has index 1, the second argument index 2, and so forth. Given the command line in the first example presented above, the commands

```
import sys
print sys.argv
```

would yield as a result:

```
['', '-some-argument', 'there', 'are', 'six', 'arguments', '-one']
```

Logging in and Logging out

The CLI will attempt to connect to the Deployit Server with the provided credentials and connection settings. If a successful connection can not be established, an error message will be displayed and the CLI will terminate.

If a successful connection with the Deployit Server is established, a welcome message will

be printed and the CLI is ready to accept commands.

In interactive mode, the Deployit CLI can be ended by typing `quit` at the prompt. This will log out the user and terminate the CLI process. In batch mode (when a script is provided), the CLI terminates automatically after finishing the script.

CLI extensions

It is possible to install CLI extensions - 'extensions' for short - which are loaded during CLI startup. Extensions are Python scripts, for example with Python class definitions, that will be available in commands or scripts run from the CLI. This feature can be combined with arguments given on the command line when starting up the CLI.

To install CLI extensions follow these steps:

1. **Create a directory called `ext`** This directory should be created in the same directory from which you will start the CLI - during startup the current directory will be searched for the existence of the `ext` directory.
2. **Copy Python scripts into the `ext` directory**
3. **(Re)Start the CLI** During startup, the CLI will search for, load and execute all scripts with the `py` or `cli` suffix found in the extension directory.

Please note that the order in which scripts from the ext directory are executed is not guaranteed.

Deployit Objects in the CLI

This chapter describes the various objects available on the CLI for scripting. The Deployit CLI provides five objects to interact with the Deployit Server: `deployit`, `deployment`, `repository`, `factory` and `security`.

It's also possible to define custom helper objects. A brief description of these four objects and how to define custom helper objects is given below. The next chapters will present examples on how to use these objects for scripting purposes.

The deployit object

The `deployit` object provides access to the main functions of Deployit itself. It allows the user to import a package, work with tasks and executing discovery.

The deployment object

The `deployment` object provides access to the deployment engine of Deployit. Using this object it is possible to create a task for an initial deployment or upgrade. The created tasks can be executed by using the `deployit` object.

The factory object

The `factory` object facilitates the creation of new Configuration Items (CI) and artifacts (files and packages), to be saved in Deployit's repository.

The repository object

The `repository` object allows the user to access Deployit's repository. It provides Create, Read, Update and Delete (CRUD) operations on Configuration Items (CI) in the repository. It can also export task overviews to a local XML file.

The security object

The `security` object facilitates the logging in or out of Deployit and the creation or deletion of users in the Deployit repository. Users of Deployit may also be administered using another credentials store like a LDAP directory, but creation and deletion of users on these specific stores is not within Deployit's scope.

Users with administrative permissions my also grant, deny or revoke security permissions to other users, even those users that are not administered in Deployit's repository but in some other credentials store. By default, users with administrative permissions own **all** permissions available in Deployit.

The tasks object

The `tasks` object makes it possible to interact with tasks in Deployit, such as deployments or discovery tasks. Tasks can be started, stopped or aborted or can be assigned to another

user.

Help with CLI objects when logged in

When you log in, a welcome message is shown. The message looks like this:

Welcome to the Deployit Jython CLI! Type 'help' to learn about the objects you can use to interact with Deployit.

```
Deployit Objects available on the CLI:

* deployit: The main gateway to interfacing with Deployit.
* deployment: Perform tasks related to setting up deployments
* factory: Helper that can construct Configuration Items (CI) and Artifacts
* repository: Gateway to doing CRUD operations on all types of CIs
* security: Access to the security settings of Deployit.
* tasks: Access to the task engine of Deployit.

To know more about a specific object, type <objectname>.help()
To get to know more about a specific method of an object, type <objectname>.help("<methodname>")
```

To have this message shown again, just type `help` on the CLI command prompt, followed by `<enter>`.

It's easy to obtain information about a specific Deployit object by invoking the `help()` function. This will list all methods on the object that are available for scripting:

```
deployit> factory.help()

factory: Helper that can construct Configuration Items (CI) and Artifacts

The methods available are:
* factory.artifact(String id, String ciType, Map values, byte[] data) : ArtifactAndData
* factory.configurationItem(String id, String ciType) : ConfigurationItemDto
* factory.configurationItem(String id, String ciType, Map values) : ConfigurationItemDto
* factory.types() : void
```

Extensive help about the usage of a specific method can be obtained by issuing a command like:

```
deployit> security.help('getPermissions')
```

on the object in question. Notice that the name of the method is given enclosed in quote marks and without parentheses.

Setting up security

See the *Deployit Security Manual* for more information about how to setup security.

Discovering middleware

In order to do effective deployments, you need to have an accurate model of your infrastructure in Deployit. This can be done automatically by Deployit by using the *discovery* feature. Using discovery, Deployit will populate your infrastructure repository by scanning your middleware environment and creating Configuration Items in the repository.

The CIs discovered during discovery will help you in setting up your infrastructure. However, they need not be complete: some CIs contain properties that can not be automatically discovered, like passwords. These kind of properties will still need to be entered manually.

Discovery is part of the Deployit plugin suite, and the exact discovery functionality available varies depending on the middleware platforms present in your environment. Please refer to the appropiate plugin manual for more detailed information about discovery on a certain middleware platform, including examples.

The following steps comprise discovery:

1. Create a CI representing the starting point for discovery (often a *Host* CI).
2. Start discovery with this CI.
3. Store the discovered CIs in the repository.
4. Complete the discovered CIs by providing missing properties manually if needed.
5. Add the discovered CIs to an environment.

The last step of discovery is optional. The discovered CIs will be stored under the

`Infrastructure` root node in the repository and may be added to an environment at some later time.

This manual describes how to perform discovery via the CLI but discovery can also be done via the GUI. See the **GUI Manual** for more information.

Create a starting point

The first step taken in discovery is to create a starting point. The starting point is a Configuration Item that discovery process needs to get going. This is usually a server host. Depending on the middleware you are trying to discover, additional parameters may be needed.

Following is an example of how to perform discovery based on the demo-plugin, an example plugin that is part of XebiaLabs community-plugins repository. First a CI is created for localhost. Then we create a CI for the demo server we want to discover. This CI will be the starting point for discovery.

```
# Create the host CI
deployit> host = factory.configurationItem('Infrastructure/demoHost', 'overthere.LocalHost')
deployit> host.os = 'UNIX'
deployit> repository.create(host)

deployit> server = factory.configurationItem('Infrastructure/demoHost/demoServer', 'demo.Server')
deployit> server.host = host.id
```

Execute discovery task

Now discovery task can be created and executed.

```
deployit> taskId = deployit.createDiscoveryTask(server)
deployit> deployit.startTaskAndWait(taskId)
deployit> discoveredCIs = deployit.retrieveDiscoveryResults(taskId)
```

The result of these commands will be an object containing a list of discovered CIs.

Store the discovered CIs

Deployit returns a list of discovered middleware CIs. Note that these are not yet persisted. To store them in the repository, use the following code:

```
deployit> repository.create(discoveredCIs)
```

Complete discovered middleware CIs

The easiest way to find out which of the discovered CIs require additional information is by printing them. Any CI that contains passwords (displayed as '********') will need to be completed. To print the stored CIs, the following code can be used:

```
deployit> for ci in discoveredCIs: deployit.print(repository.read(ci.id));
```

Note: the created CIs can also be edited in the GUI using the Repository Browser if they have been stored in the repository.

Adding CIs to Environments

Middleware that is used as a deployment target must be grouped together in an environment. Environments are CIs of type `udm.Environment` and, like all CIs, can be created from the CLI by using the factory object. The following command can be used for this:

```
deployit> env = factory.configurationItem('Environments/DiscoveredEnv', 'udm.Environment')
```

Add the discovered CIs to the environment:

```
deployit> env.values['members'] = [ci.id for ci in discoveredCIs]
```

Note that not all of the discovered CIs should necessarily be stored in an environment. For example, in the case of WAS, some nested CIs may be discovered of which only the top-level one must be stored.

Don't forget to stopre the new environment in the repository:

```
deployit> repository.create(env)
```

The newly created environment can now be used as a deployment target.

Note: the user needs specific permission to store CIs in the database. See the *Deployit System Administration Manual*.

Performing deployments

You can also do a deployment from the Deployit CLI. Here is an example of how to perform a simple deployment.

```
# Import package
deployit> package = deployit.importPackage('demo-application/1.0')

# Load environment
deployit> environment = repository.read('Environments/DiscoveredEnv')

# Start deployment
deployit> deploymentRef = deployment.prepareInitial(package.id, environment.id)
deployit> deploymentRef = deployment.generateAllDeployeds(deploymentRef)
deployit> taskID = deployment.deploy(deploymentRef).id
deployit> deployit.startTaskAndWait(taskID)
```

Undeployment follows the same general flow:

```
deployit> taskID = deployment.undeploy('Environments/DiscoveredEnv/demo-application').id
deployit> deployit.startTaskAndWait(taskID)
```

Working with Tasks

Deployit can perform many deployments at the same time. Each of these deployments is called a *task*. Users can ask Deployit to start a task, stop a task or cancel a task.

Once a task is completed or canceled, it is moved to the *task archive*. This is where Deployit stores it's task history. You can query it for tasks and examine the tasks steps and logs or export the task archive to an XML file.

Active tasks are stored in the Deployit *task registry* which is periodically backed up to a file. If the Deployit Server is stopped abruptly, the tasks in the registry are persisted and can be continued when the server is restarted.

Starting, stopping and canceling

Whenever you start a deployment or undeployment in Deployit, the Deployit CLI returns a *TaskInfo* object. This object describes the steps Deployit will take to execute your request, but it doesn't yet start execution. Instead, Deployit creates a task for the request and returns its id as the *id* field in the *TaskInfo* object. Using the task id, you can start, stop or cancel the task.

There are two ways to start a task in the Deployit CLI: *startTaskAndWait* and *startTask*. In the deployment example above we used *startTaskAndWait*. This method starts the tasks and waits for it to complete. The *startTask* method starts the task in Deployit and returns immediately. The task is run in the background in ths case. Both methods can also be used to restart a failed task.

If a task is running and you want to to stop it, use the *stopTask* method. This attempts to interrupt the currently running task.

The *cancelTask* method is used to cancel a task. That is, abandon execution of the task and move it to the archive.

When a task has finished running, you can use the *archive* method to archive.

Listing tasks

Before you can work with any of the tasks, you'll need to list them:

```
deployit> depl = deployment.prepareInitial(package.id, environment.id)
deployit> depl = deployment.generateAllDeployeds(depl)
deployit> taskId = deployment.deploy(depl).id
deployit> print deployit.listUnfinishedTasks()
```

This retrieves and shows a list of unfinished tasks that are assigned to the current user. The method returns an array of *TaskInfo* objects which you can actually print:

```
deployit> for t in deployit.listUnfinishedTasks(): print "Task id " + t.id + " is assigned to user " + t.user
```

If you have *admin* permission, you can also list all tasks in Deployit:

```
deployit> print deployit.listAllUnfinishedTasks()
deployit> deployit.cancelTask(taskId)
```

Assigning tasks

Tasks in Deployit are assigned to the user that started them. This means that they will appear in the Deployit GUI whenever this user logs out and back in again.

Deployit also supports reassigning tasks. If you have *task#assign* permission, you are allowed to assign a task currently assigned to you to another principal. If you have the *admin* permission, you can assign any task in the system to another principal.

This is how you assign a task in the CLI:

```
# Import package
deployit> package = repository.read('Applications/demo-application/1.0')

# Load environment
deployit> environment = repository.read('Environments/DiscoveredEnv')

# Start deployment
deployit> deploymentRef = deployment.prepareInitial(package.id, environment.id)
deployit> deploymentRef = deployment.generateAllDeployeds(deploymentRef)
deployit> taskID = deployment.deploy(deploymentRef).id

deployit> deployit.assignTask(taskID, 'admin')

# perform some operations on the task

deployit> deployit.cancelTask(taskID)
```

Note: Deployit does not validate the principal you enter as the recipient of the task.

Retrieving archived tasks from the repository

The `repository` object has facilities to retrieve an overview of all archived tasks, or a number of tasks within a specified date range.

The command to export all tasks is:

```
deployit> archivedTasks = repository.getArchivedTasks()
```

This command will return an object that contains all tasks, and tasks in turn contain all of their steps. For instance, to get the number of tasks retrieved, execute:

```
deployit> print repository.getArchivedTasks().size()
```

or, when you've assigned the object to a variable named `archivedTasks`:

```
deployit> print archivedTasks.size()
```

To obtain the first retrieved Task from the object, yield:

```
deployit> firstTask = archivedTasks.getTasks().get(0)
```

The task count starts at 0 to size() exclusive. This call will give you a `TaskInfo` object on which you may call all normally available methods. To obtain the first step from the acquired

task, execute:

```
deployit> firstStep = firstTask.getSteps().get(0)
```

Again, the count of steps starts at 0 to getSteps().size() exclusive. This will give you a `StepInfo` object on which all regular methods may be called.

Once you've obtained a `TaskInfo` or `StepInfo` object, you can query it for all relevant information, like, for instance in the case of a step, it's state:

```
deployit> firstStep.getState()
```

or it's step number:

```
deployit> firstStep.getNr()
```

Next to all tasks, one may also just export all tasks within a given date range executing the following command:

```
deployit> repository.getArchivedTasks('01/01/2010', '01/01/2011')
```

Both date parameters in the method signature should be specified in the following format **mm/dd/yyyy**, with **m** a month digit, **d** a day digit and **y** a year digit. The above method call will return an object that simply wraps the requested tasks, analogous to the getArchivedTasks() method call.

Exporting archived tasks from the repository to a local XML file

It's also possible to store the contents of the task repository to a local XML file. In order to store the complete task repository to a local XML file, use the following command:

```
deployit> repository.exportArchivedTasks('/tmp/task-export.xml')
```

Note that you can use forward slashes in the path, even on Windows systems.

It is also possible to export a number of tasks in a certain date range from the task repository to a local XML file using the following command:

```
deployit> repository.exportArchivedTasks('/tmp/task-export.xml', '01/01/2010', '01/01/2011')
```

Performing Common Tasks

This section describes common tasks that may be performed using the CLI. Its main purpose is to present examples of how to combine commands to perform the desired tasks.

Working with Configuration Items

This section shows some examples of how to work with CIs. The two main objects involved are the `factory` object and the `repository` object. The `factory` object is used to actually create the CI itself, while with the `repository` object it is possible to store the CI in the repository.

Finding out types of available CIs and their properties

The available CIs and their respective type need to be known before being able to create one. Using the command

```
deployit> factory.types()
```

an overview will be shown on standard output of all the available types that are shipped with Deployit. If at some point more plugins are added to Deployit, types defined therein will be added to Deployit's type registry and will then also be available in addition to the types initially shipped with Deployit. The new types should also show up in the output of this command.

In order to obtain some more details of a specific type, for instance its required properties, execute the `describe` method on the `deployit` object with the fully qualified type name as its parameter:

```
deployit> deployit.describe('udm.Dictionary')
```

The output of this command will show something like:

```
ConfigurationItem udm.Dictionary:
Description: A Dictionary contains key-value pairs that can be replaced
Control tasks:
Properties:
    - entries(MAP_STRING_STRING): The dictionary entries

Properties marked with a '!' are required for discovery.
Properties marked with a '*' are required.
```

Creating common UDM CIs

The following snippet shows examples of creating common UDM CIs.

```
# Create a host
deployit> sampleHost = factory.configurationItem('Infrastructure/sampleHost', 'overthere.SshHost',
    { 'os': 'UNIX', 'address': 'localhost', 'username': 'scott' })
deployit> repository.create(sampleHost)
deployit> deployit.print(sampleHost)

# Create a dictionary
deployit> sampleDict = factory.configurationItem('Environments/myDict', 'udm.Dictionary')
deployit> sampleDict.entries = { 'a': '1', 'b': '2' }
deployit> repository.create(sampleDict)
deployit> deployit.print(sampleDict)

# Create an environment
deployit> sampleEnv = factory.configurationItem('Environments/sampleEnv', 'udm.Environment')
deployit> sampleEnv.dictionaries = [ sampleDict.id ]
deployit> sampleEnv.members = [ sampleHost.id ]
deployit> repository.create(sampleEnv)
deployit> deployit.print(sampleEnv)
```

Moving and Renaming CIs

The repository allows you to move or rename CIs as well. Note that a CI can only be moved within the root node it was created in. That is, a CI under the *Application* root node can only be moved to another place in this tree.

The following snippet shows examples of moving and renaming CIs:

```
# Create a directory to store environments
deployit> directory = factory.configurationItem('Environments/ciGroup', 'core.Directory')
deployit> repository.create(directory)

# Move the sample environment in the new directory
deployit> repository.move(sampleEnv, directory.id + '/sampleEnv')
deployit> sampleEnv = repository.read('Environments/ciGroup/sampleEnv')

# Rename the directory
deployit> repository.rename(directory, 'renamedCiGroup')
deployit> sampleEnv = repository.read('Environments/renamedCiGroup/sampleEnv')

# References to renamed or moved CIs are kept up-to-date
deployit> repository.rename(sampleHost, 'renamedSampleHost')
deployit> sampleEnv = repository.read('Environments/renamedCiGroup/sampleEnv')
deployit> sampleHost = repository.read(sampleEnv.members[0])
```

Note: moving or renaming CIs when deployments are in progress or when the CIs concerned are used by Deployit clients (GUI or CLI) is discouraged.

Executing a Control Task

Control tasks can be executed from the CLI. Take, for example, the *start* control task on a Glassfish server in the glassfish-plugin (available as a community plugin, see the introduction of this manual). It can be executed as follows:

```
deployit> server = repository.read('Infrastructure/demoHost/demoServer')
deployit> deployit.executeControlTask('start', server)
```

Shutting down the Server

The CLI can also be used to shutdown the Deployit Server as follows:

```
deployit.shutdown()
```

Note that you must have administrative permissions for this.

Type reference

Here's an overview of types used in the CLI

Artifact		A CI with an associated file
extends ConfigurationItem		
file	OverthereFile	An abstract representation of a file that can be access through an OverthereConnection

ArtifactAndData		Contains an Artifact and the actual file contents
artifact	Artifact	The artifact
data	byte[]	The file contents
filename	String	The name of the file

ConfigurationItem		A new CI, or a CI from the repository
id	String	The ID of the Configuration Item
type	String	The UDM type of the Configuration Item
values	Map	The properties of the CI
validations	List	The validation errors on the CI

ConfigurationItemId		A CI reference, with type information
id	String	The ID of the Configuration Item
type	String	The UDM type of the Configuration Item

Deployment		Settings for doing a deployment
deployedApplication	ConfigurationItem	The application being deployed
deployeds	List	The configured deployeds
deployables	List	The available deployables in the package
containers	List	The containers being deployed to
deploymentType	String	The type of deployment. Possible values: `INITIAL, UPDATE, UNDEPLOYMENT`

StepState		Information about a task step
description	String	The human-readable description of the step
state	String	The current state of the step. Possible values: `PENDING, SKIP, EXECUTING, PAUSED, FAILED, DONE, SKIPPED`
startDate	Calendar	The date the step was started
completionDate	Calendar	The date the step was completed
log	String	The log output of the step
failureCount	String	The times the step has failed
metadata	map	Step metadata, containing the ```order``` and ```deployed```

TaskState		Deployit task
id	String	The ID of the task
state	String	The state the task was in when queried. Possible values: `PENDING, QUEUED, EXECUTING, STOPPED, EXECUTED, DONE, CANCELLED`
startDate	Calendar	The date the task was started
completionDate	Calendar	The date the task was completed
nrSteps	int	The number of steps in the task
currentStepNr	int	The number of the step the task is currently at.
metadata	map	Task metadata, including the ```application```, ```environment```, ```version```
failureCount	int	The number of times the task has stopped because of a failed step.
owner	String	The current owner of the task.

TaskWithSteps		Deployit task including it's steps
id	String	The ID of the task
state	String	The state the task was in when queried. Possible values: `PENDING, QUEUED, EXECUTING, STOPPED, EXECUTED, DONE, CANCELLED`
startDate	Calendar	The date the task was started
completionDate	Calendar	The date the task was completed

nrSteps	int	The number of steps in the task
currentStepNr	int	The number of the step the task is currently at.
metadata	map	Task metadata, containing the ```application```, ```environment```, ```version```
failureCount	int	The number of times the task has stopped because of a failed step.
owner	String	The current owner of the task.
steps	List	All steps in the task represented as StepState objects

FullTaskInfo		Task that includes step information
extends TaskInfo		
steps	List	The steps in the task.

FullTaskInfos		List of tasks with step information
tasks	List	All retrieved tasks as a list.

ValidationMessage		Indicates a validation error and provides a message
ciId	String	The ID of the Configuration Item this validation message refers to
propertyName	String	The name of the property in the CI this validation message refers to
message	String	The message itself

StepInfo *Deprecated, use StepState*		Information about a task step
description	String	The human-readable description of the step
state	String	The current state of the step. Possible values: `PENDING, SKIP, EXECUTING, PAUSED, FAILED, DONE, SKIPPED`
startDate	Calendar	The date the step was started
completionDate	Calendar	The date the step was completed
log	String	The log output of the step
failureCount	String	The times the step has failed
nr	int	The position of the step in the task

TaskInfo *Deprecated, use TaskState*		Deployit task
id	String	The ID of the task
label	String	The label describing the task
state	String	The state the task was in when queried. Possible values: `PENDING, QUEUED, EXECUTING, STOPPED, EXECUTED, DONE, CANCELLED`
startDate	Calendar	The date the task was started
completionDate	Calendar	The date the task was completed
nrOfSteps	int	The number of steps in the task
currentStepNr	int	The number of the step the task is currently at.
application	String	*For deployment tasks* - The udm.Application the task is for
version	String	*For deployment tasks* - The version of the udm.Application the task is for. This refers to the actual package being deployed, viz. the deployment package or composite package
environment	String	*For deployment tasks* - The environment the package is being deployed to
failureCount	int	The number of times the task has stopped because of a failed step.