

Deployit Command Line Interface Manual

Version 3.6.4

Table of Content

Preface	3
Using the CLI	3
Environment variables	3
Starting the CLI	3
Starting up with different settings	4
Passing arguments to CLI commands or script	5
Logging in and Logging out	6
CLI extensions	6
Objects available on the CLI	6
The deployit object	6
The deployment object	6
The repository object	6
The factory object	7
The security object	7
Help with CLI objects when logged in	7
Setting up Security	8
Adding users to the repository	8
Granting, revoking and denying permissions to users	8
Permissions available in the security system	9
Managing Users	9
Changing a user's password	10
Login as a different user	10
Retrieving and listing user permissions	10
Example setting up initial security	11
Performing Common Tasks	13
Working with Configuration Items	13
Finding out types of available CIs and their properties	13
Discovering middleware Configuration Items	13
Create a Configuration Item starting point	14
Start discovery passing a Configuration Item	14
Complete discovered middleware CIs	15
Store the CIs in the repository	15
Adding CIs to Environments	15
Executing a Control Task	16
Retrieving archived tasks from the repository	16
Exporting archived tasks from the repository to a local XML file	17
Performing deployments	17
Performing a simple deployment	17

Preface

Sometimes the need arises to interact *Deployit* programmatically. Discovering your middleware topology, setting up environments, importing packages and performing deployments are just some examples of parts of your deployment life-cycle that you may want to control or execute programmatically.

Just to make it possible for users to use *Deployit*'s powerful capabilities in a programmatic way, *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.

The CLI has been designed to allow it to be programmed by using the standard *Python* programming language. This language has been specifically chosen as the default because most system administrators and developers are familiar with it. In the Appendix to this manual, all objects available for scripting are listed including an overview of the methods that can be invoked on them. Throughout this manual, examples will also 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.

Using the CLI

The CLI connects to the *Deployit* server using the standardized HTTP / HTTPS protocol. This makes it possible to easily use the CLI remotely without hindrance from, e.g., firewalls.

To start the CLI successfully, the *Deployit* server to connect to must be running. See the *Deployit System Administrator Manual* for more information on how to startup the *Deployit* server.

Environment variables

After installing the *Deployit* CLI, 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, named `DEPLOYIT_CLI_OPTS` can be set in the startup `cli.cmd` or `cli.sh` script. This variable allows to override default JVM options and to set, for example, memory parameters. 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 like:

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

Suggested options for the CLI are:

- Minimum heap size 256 megabytes (`-Xms256m`)
- Minimum permgen space 256 megabytes (`-XX:MaxPermSize=256m`)

Starting the CLI

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

Once a terminal window or command shell has been opened and the directory changed to where the *Deployit* binaries are, the CLI may be started by entering the command `./cli.sh` on *Unix* like platforms, or `./cli.cmd` on *Windows* platforms.

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, it will be possible to execute commands, within the limits of the granted permissions; see later section on security.

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:

<code>./cli.sh [options] [--] arguments</code>	
options:	
<code>-configuration VAL</code>	: Specify the location of the configuration file
<code>-context VAL</code>	: The context <i>Deployit</i> is running at
<code>-f (-source) VAL</code>	: Execute a specified python source file
<code>-host VAL</code>	: Connect to a specified host, defaults to 127.0.0.1
<code>-port N</code>	: Connect to a specified port, defaults to 4516
<code>-secure</code>	: Use https to connect to the <i>Deployit</i> server
<code>-username VAL</code>	: Connect as the specified user
<code>-password VAL</code>	: Connect with the specified password

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

- `-configuration /path/to/config/file`

This option is used to pass the location of the *Deployit* configuration file. The default location for the configuration file is `conf/deployit.conf` in the *Deployit* installation directory.

- `-context /url/path/part/you/want`

If connecting to the *Deployit* server is hindered because of intermediate proxy servers, URL rewrite rules, etc., it's possible by use of this option to change the base part of *Deployit*'s URL. So instead of `/deployit/rest/of/the/url`, the URL will become `/url/path/part/you/want/rest/of/the/url`.

- `-f /path/to/Python/script`

Starts the CLI in batch mode and instruct it to execute the script in the specified file. Once the script completes, the CLI will terminate.

- `-source /path/to/Python/script`

Alternative for the `-f` option.

- `-host myhost.domain.com`

Specifies the host where the *Deployit* server is running to connect to. 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 try to connect to the *Deployit* server on the secure port 4517. In order to be able to connect,

the *Deployit* server must have been started using this secured port, which is the default. It is possible to startup using a different port using the `-port` option. Of course the *Deployit* server should also have started up using this port instead of the default one for a secure connection.

- `-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.

One example of using options might be:

```
./cli.sh -username User -password UserPassword -context /deployit-proxy/deployit
```

This will log the user with username 'User' and password 'UserPassword' in on the CLI when connecting to the *Deployit* server running on the local machine ('localhost') and listening on port 4516.

Passing arguments to CLI commands or script

As can be seen from the message shown in the previous section, next to options it's also possible to pass arguments from the command line to the CLI while starting up the CLI. It's not mandatory to specify any options in order to pass arguments. Following examples demonstrate two ways of passing arguments, one without options:

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

and one 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, with the first argument having 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[2] + " " + sys.argv[3] + " " + sys.argv[5]
```

would yield as a result:

```
there are arguments
```

Logging in and Logging out

Once the CLI has obtained the username and password of the user that wants to login, either by using the command line options or interactively, it will attempt to connect to the *Deployit* server at the specified (or default) address and port. If a successful connection can not be established, a stacktrace will be printed 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.

CLI extensions

It is possible to install CLI extensions - 'extensions' for short - which are loaded during CLI startup. Extensions are nothing more but *Python* scripts, for example with *Python* class definitions, that will be available after CLI startup to be used in commands or scripts. This feature may be nicely 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 loading and execution of scripts that are present in the extension directory will take place, is not guaranteed!

Objects available on the CLI

This chapter describes the various objects available on the CLI for scripting. At present there are five objects available, to wit: `deployit`, `deployment`, `repository`, `factory` and `security`.

Next to the four available objects, it's also possible to define custom helper objects. A brief description of these four objects and how to define custom helper objects will be given in the next sections. 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 - stopping, starting, canceling or aborting 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 or upgrade deployment. It's also possible to generate a task for just a single deployment. The task thus created can be executed by using the `deployit` object and hence actually executing the deployment itself.

The `repository` object

The `repository` object allows the user to access *Deployit's* repository. This includes searching the repository and performing Create, Read, Update and Delete (CRUD) operations on Configuration Items (CI) in the repository. It's also possible to export the complete task overview , or just the ones within a specified date range, to a local XML file.

The factory object

The `factory` object facilitates the creation of new Configuration Items (CI) and artifacts. These will be saved in *Deployit's* repository. Artifacts are all sorts of files and packages.

The security object

The `security` object facilitates the login or logout of *Deployit* and the creation or deletion of users in *Deployit's* own 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 may 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*.

Help with CLI objects when logged in

As stated in the previous section, a welcome message is shown once the user has been successfully logged in. This message looks like:

```
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: Access to the deployment engine of Deployit.
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.

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>`.

Looking at the last paragraph of this message, one can see that it is easy to obtain information about a specific object by, for example, issuing the command:

```
security.help()
```

This will list all methods on the specific object like `security` available for scripting. Extensive help about the exact usage of a specific method can be obtained by issuing a command like:

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

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

Also, the Appendix of this manual lists all objects available on the CLI for scripting as well as a description of the methods available on those objects with their required parameters.

Setting up Security

After a fresh installation of *Deployit* no permissions are granted to any user. The only users that have any permissions granted to them, are the `administrator` users and they will, by default, have **all** permissions granted to them.

Currently, *Deployit* ships with one predefined administrator user called `admin`, with default password `admin`.

So the first task an `admin` user should do, is change the default password to something hard to guess and keep this password private. The next tasks should consist of adding users to *Deployit's* repository - if no other credentials store is in use, or possibly in addition to another credentials store that will be used - and granting permissions to users that will work with *Deployit*, starting with the (global) permission to login.

Adding users to the repository

In a typical medium to large size company, there are several different groups of people that perform tasks related to deployments. For example, there are administrators that install, test and maintain hardware and there are deployers that deploy applications to development, test, acceptance as well as production environments. And obviously there are developers who build the applications waiting to be deployed to their respective target environments.

In order to create a new user, the following command must be issued while logged in as the `admin` user:

```
security.create('username', 'userPassword')
```

To create a user with username 'John' and password 'Doe', this command would become:

```
security.create('John', 'Doe')
```

Users created this way will however only end up in *Deployit's* own repository. In order to create users in, for example, a LDAP credentials store, use your favorite LDAP administration tool.

The concept of groups is also supported by *Deployit* when using LDAP as the credentials store. All groups defined in LDAP can be assigned permissions in *Deployit* and users belonging to these groups will be assigned the group permissions when they use the system.

Granting, revoking and denying permissions to users

Permissions in the *Deployit* security system are either global or local. Global permissions that are granted to the user are in effect for **all** Configuration Items (CI) in the repository, meaning the permission is granted on **all** Configuration Items (CI), even the CI's that will be created after the permission has been granted. On the other hand, local permissions are set on specific CI's in the repository and need not be automatically in effect for newly created CI's.

Setting global and/or local permissions will result in a so-called `Access Control Entry` (ACE) being added to the user's `Access Control List` (ACL). Permissions set on CI's are **transitive**, meaning that a permission set on a CI will also be in effect on all of its children, their children and so forth, unless explicitly denied.

A permission can be either granted, denied or revoked:

- `granted`
a user will be explicitly allowed to execute a specific task
- `denied`
a user will be explicitly denied to execute a specific task
- `revoked`
a user's previously `granted` permission will be revoked, whereby the user is no longer allowed to execute this

specific task, but it will not be explicitly denied.

Permissions available in the security system

Following is a list of all permissions available in *Deployit's* security system. Some permissions are termed `global`, meaning that they are in effect on **all** CI's. Permissions not specifically termed `global` can be either allowed on all CI's, by not specifying any CI's on which the permission should be in effect, or on a set of CI's - and their children! - by specifying a list of the specific CI's the permission should be in effect upon.

Permissions available in the *Deployit* security system are:

- `login`
This global permission allows a user to login to *Deployit*. Without this permission a user will not be able to work with *Deployit*.
- `read`
This permission is needed to read a CI, or any of its children, from the repository.
- `repo#edit`
This permission allows a user to edit CI's in the repository.
- `discovery`
This global permission allows a user to be able to do discovery of infrastructure topology. Although it is termed as being global, it is actually only in effect on the CI's under the `Infrastructure` node because only infrastructure can be discovered.
- `import#initial`
This permission allows a user to import a package that is not already in *Deployit's* repository.
- `import#upgrade`
This permission allows a user to import a package that already exists in *Deployit's* repository and effectively upgrades it to a newer version.
- `deploy#initial`
This permission allows a user to perform a deployment from a package to an environment for the first time, thereby allowing to specify what specific package to deploy to what environment.
- `deploy#upgrade`
This permission allows a user to perform a deployment from a package to an environment that has already been performed before, thereby effectively upgrading the previous deployment. Note that no deployment of a package, application or resource is allowed to a member of an environment that has not been specified in the initial deployment, or a different environment altogether. Also, no other package may be specified.
- `deploy#undeploy`
This permission allows a user to undo a previously performed deployment.
- `task#move_step`
This global permission allows a user to move a step present in the sequence of steps to another position in this sequence before the execution of the deployment process, being either initial, upgrade or undeploy, thereby effectively changing the order in which the steps comprising the deployment process are executed.
- `task#skip_step`
This global permission allows a user to skip a step during the deployment process, being either initial, upgrade or undeploy, thereby offering the possibility of not executing specific actions that would otherwise be executed as being comprised in the deployment process.

Managing Users

Sometimes it proves necessary to change a user's password, to delete a user or to login as a different user in order to perform a specific task with a different set of permissions. An example of this latter case might be the user `admin` that normally would work using another username with less permissions to perform regular deployment tasks, but needs to switch to `admin` in order to, for example, create a new user.

And sometimes it proves useful - in case of troubleshooting for instance - to just be able to see what permissions exactly have been granted to what user on which CI's.

Changing a user's password

If for some reason a user's password gets compromised or the user has forgotten it, it will be possible to set a new password for the user. Note that it will not be possible to retrieve the user's password since *Deployit* does not store it as plain text. When requesting and viewing a user CI, passwords will always be shown as eight asterisks (*).

In order to change the password for user `developer`, issue following command:

```
devUser = security.readUser('developer')
devUser.password = 'newpassword_1'
security.modifyUser(devUser)
```

Please note the difference between the name of the user in the first command, `developer`, and the object representing this user that was returned after executing the first command; this `user` object need also be used in the next two commands following the first one.

Login as a different user

It is possible to login as a different user once the CLI has started up. In order to log in as another user, you must first logout the user that is currently logged in. This will show the `deployit>` prompt from which it is then possible to login again as a different user.

In following example an administrator is logged in as user `senior-deployer` and needs to login as `admin` to delete user `ex-developer`.

```
security.logout()
security.login('admin', 'admin')

# Delete the user ex-developer
security.deleteUser('ex-developer')

#switch back to the account with less privileges
security.logout()
security.login('senior-deployer', 'password_2')
```

Retrieving and listing user permissions

In some cases, for instance trouble shooting, it's necessary to get an overview of the permissions granted to a specific user. A user logged in to *Deployit*'s CLI may also want to retrieve and list the granted or denied permissions. *Deployit*'s CLI comes with some auxiliary methods on its `security` object to retrieve and list granted or denied permissions.

To retrieve and show ones own permissions, assuming user `'deployer'` is logged in, issue `security.getPermissions()` or, alternatively, `print security.getPermissions('deployer')`

The latter form of the command will normally only be issued by an administrator user. This is because there exists a slight difference between both commands. The first command will just simple list all permissions of the user currently logged in on standard output. The second command will return a `PrincipalPermissions` object (see the appendix for a complete description of this object) on which the `getPermissions('deployer')` method is executed. Without the `print` statement, nothing would have shown up on standard output, because the method has a `PrincipalPermissions` object as its return value. In combination with the `print` statement, as shown in the above command, the object will print its string representation which, for this object, is a list of granted permissions and the CIs on which these have been granted.

The `PrincipalPermissions` object makes it possible to retrieve and inspect user permissions, to find out if a given user has any permissions at all or has a specific permission and on what CIs. This will, for example, allow an administrator to automate granting and/or revoking of permissions by use of a script, depending on the retrieved

permissions and their state or CIs.

Take note that only an administrator user, i.e. `admin`, is allowed to retrieve permissions for another user and to grant, deny or revoke permissions. Normally it is only possible to retrieve and list ones own permissions. Trying to grant permissions using the second form of the command while not being logged in as user `admin`, will lead to an exception.

A somewhat more extensive example of using this object, to be executed as `admin`, is listed in following snippet:

```
permissions = security.getPermissions('deployer')
if not permissions.hasPermission('login'):
    security.grant('login', 'deployer')

if permissions.hasPermission('repo#edit', 'Applications/PetClinic/1.0'):
    security.revoke('repo#edit', 'deployer' ['Applications/PetClinic/1.0'])
```

Example setting up initial security

In a hypothetical company following users will work with *Deployit*:

- **administrator**

This user will have the overall authority to administer deployments in *Deployit*. He will have permissions to create, update and delete infrastructure as well as permissions to create, update and delete environments. Beware not to confuse this user with the `admin` user, i.e. a user that has administrative privileges over the whole of *Deployit*.

- **senior-deployer**

A user with permission to import new applications and to perform deployments to the development (DEV), test (TEST) and production (PROD) environments.

- **deployer**

A user with the permission to import new applications, deploy applications to the DEV and TEST environments and to view the PROD environment.

- **developer**

A user with permission to import new versions of existing applications and to upgrade existing deployments.

The first task for a user with administrator privileges, *Deployit* ships with a predefined `admin` user, will be to create the users in *Deployit* and allowing them to be able to login, executing the following code (the passwords in the examples are chosen for clarity. When creating users it's advisable to choose strong and difficult to guess passwords):

```
#
# Sample security setup.
#
security.createUser('administrator', 'password_1')
security.createUser('senior-deployer', 'password_2')
security.createUser('deployer', 'password_3')
security.createUser('developer', 'password_4')

security.grant('login', 'administrator')
security.grant('login', 'senior-deployer')
security.grant('login', 'deployer')
security.grant('login', 'developer')
```

Since the `login` permission is a global permission, there's no need to specify a list of CI's it will be in effect upon.

Next is to grant the individual users their needed aggregated permissions in order for them to perform their tasks. The following snippet of code is a continuation of the above snippet; together with the snippets to come, it will constitute the complete script for this example.

The first user we will grant permissions to, is user `administrator`. At minimum he needs `read` permission on the `Infrastructure` and `Environments` root nodes defined in *Deployit* and `repo#edit` permission in order to perform any creation, deletion and updating of CI's under these root nodes.

```
security.grant('read', 'administrator', ['Infrastructure', 'Environments'])
security.grant('repo#edit', 'administrator')
```

Notice the use of a list of target CI's in the first command. The permissions will only be set on these two specified CI's - which happen to be the root nodes for `Infrastructure` and `Environments` in this case. If for instance there would exist a user that had limited administrator privileges, it would be possible to just grant this user rights to specific environments by using the command:

```
security.grant('repo#edit', 'partial-administrator', ['Environments/PROD'])
```

Going back to the example, the senior-deployer needs the permissions to import and deploy applications to the `DEV`, `TEST` and `PROD` environments. Therefore, the permissions can be set "globally" and need not be set on specific CI's:

```
security.grant("import#initial", 'senior-deployer')
security.grant("import#upgrade", 'senior-deployer')
security.grant("deploy#initial", 'senior-deployer')
security.grant("deploy#upgrade", 'senior-deployer')
```

Setting up permissions for user `deployer` will be a bit more complicated because of the restrictions imposed. The first set of permissions are similar to those of the user `senior-deployer`:

```
security.grant("import#initial", 'deployer')
security.grant("import#upgrade", 'deployer')
security.grant("deploy#initial", 'deployer', ['Environments/DEV', 'Environments/TEST'])
security.grant("deploy#upgrade", 'deployer', ['Environments/DEV', 'Environments/TEST'])
```

As can be seen from the commands in the snippet above, `deployer` does not have permission to perform a deployment to the `Environment/PROD` environment. In order for the user `deployer` to see deployments in the `PROD` environment, read permission should be granted on both the `Environment/PROD` and the corresponding `Infrastructure` in this environment.

```
prodEnv = repository.read('Environments/PROD')
security.grant('read', 'deployer', [prodEnv.id] + prodEnv.values['members'])
```

Finally permissions need to be granted to user `deployer`. This user is permitted to import new versions of applications into *Deployit* and to perform the deployment upgrade that goes hand in hand with this new version.

```
security.grant("import#upgrade", 'developer')
security.grant("deploy#upgrade", 'developer')
```

In this scenario the user `deployer` has the ability to import new versions of **all** applications known to *Deployit*. To restrict his import permission to new versions of some specific applications, like for example the `PetClinic` application, the following command can be issued:

```
security.grant('import#upgrade', 'developer', ['Applications/PetClinic'])
```

After the commands in the above snippets have been executed, the initial security setup of *Deployit* will match the intended security setup as described in the beginning of this section.

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

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

The output of this command will show something like:

```
ConfigurationItem "udm.Dictionary" (udm.Dictionary):  
Description: A Dictionary contains key-value pairs that can be replaced  
  
Properties  
  * environment(): The environment this dictionary belongs to  
  * dict(MAP_STRING_STRING): The dictionary values  
  
Properties marked with a '*' are required.
```

Discovering middleware Configuration Items

It is possible to let *Deployit* help with setting up your environment by attempting to discover the available middleware in your environment. In order to do this, *Deployit* will scan your environment for as far as possible and create Configuration Items in its repository based on the configurations it encounters during the scan. This process is known as *discovery*.

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

Because *discovery* is part of the *Deployit* plugin suite, the exact discovery functionality available varies depending on

the middleware platforms present in your environment.

The following steps comprehend *discovery*:

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

Note however that 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.

Create a Configuration Item starting point

The first step taken in *discovery* is to create a starting point to kick off the process from. This starting point consists of a Configuration Item specifying at least the host that *discovery* should start at. Depending on the middleware you are trying to discover, additional parameters may be needed.

Following is an example of how to start *discovery* in case of a *WebSphere Application Server (WAS)*. First a CI is created for the host itself and next a Configuration Item is created for the deployment manager running on that host. The deployment manager CI will be the starting point for *discovery*.

```
#Create a CI with the required discovery parameters filled in
wasHost = factory.configurationItem('Infrastructure/rs94asob.k94.corp.com', 'overthere.SshHost', {
    'address':'was-61',
    'username':'root',
    'password':'rootpwd',
    'os':'UNIX',
    'accessMethod':'SSH_SFTP'
})

repository.create(wasHost)

#Now create a WAS deployment manager
dmManager = factory.configurationItem(wasHost.id + '/wasDM', 'was.DeploymentManager', {
    'host':'Infrastructure/rs94asob.k94.corp.com',
    'version':'WAS_61',
    'wasHome':'/opt/ws/6.1/profiles/dmgr',
    'username':'wsadmin',
    'password':'wsadmin'
})
```

Start discovery passing a Configuration Item

After the CI starting point has been created, it can be used to perform *discovery*. The *Deployit* CLI *discovery* functionality is synchronous, which means that the CLI will wait until the *discovery* process finishes.

The process of *discovery* works exactly like a regular task in that it executes a number of steps behind the scenes. Whenever one of these steps fails, the entire *discovery* fails and aborts. It is not possible to continue an interrupted *discovery* process.

The command to start discovery is:

```
discoveredObjects = deployit.discover(dmManager)
```

Note there are no single- or double quotes around *dmManager*, because it's an object and not a string. The result of this command will be an object containing a list of discovered CIs. These CI objects can be obtained as follows:

```
discoveredCIs = discoveredObjects.objects;
```

Complete discovered middleware CIs

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

```
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.

Store the CIs in the repository

The *repository* CLI object makes it possible to store all of the discovered CIs in one go. This is also possible when the discovered CIs reference each other.

The command to do this is:

```
repository.create(discoveredObjects)
```

In order to store CIs in the database, the user needs specific permission to be able to do so.

Adding CIs to Environments

Middleware that is used as a deployment target must be grouped together in an environment. Environments are CIs and like all CIs, they can be created from the CLI. The following command can be used for this:

```
devEnv = factory.configurationItem('Environments/Dev', 'udm.Environment')
```

Add the discovered CIs to the environment:

```
devEnv.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.

Store the new environment:

```
devEnv = repository.create(devEnv)
```

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*.

Executing a Control Task

Control tasks can be executed from the CLI as well. Take, for example, the *start* control task on a webserver in the webserver-plugin. It can be executed as follows:

```
webserver = repository.read('Infrastructure/MyApache')
deployit.executeControlTask('start', webserver)
```

Retrieving archived tasks from the repository

The *repository* CLI 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:

```
repository.getArchivedTasks()
```

This command will return an object on which you can interact with tasks and their interned steps. The returned object contains all tasks, and tasks in turn contain all of their steps. For instance, to get the number of tasks retrieved, execute:

```
repository.getArchivedTasks().size()
```

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

```
archivedTasks.size()
```

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

```
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:

```
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:

```
firstStep.getState()
```


or it's step number:

```
firstStep.getNr()
```

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

```
repository.getArchivedTasks(String beginDate, String endDate)
```

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:

```
repository.exportArchivedTasks(String your-file-path)
```

The variable *your-file-path* contains the complete path to the file, e.g.: `/tmp/my-xml.xml` on Unix/Linux systems or `d:/temp/my-xml.xml` on Windows systems. The last path may alternatively be typed as `d:\\temp\\my-xml.xml` - notice the escaped backslash in this latter variant.

Of course 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:

```
repository.exportArchivedTasks(String your-file-path, String beginDate, String endDate)
```

For the format of the parameters used in the method, see above. Some examples of these commands are:

```
repository.getArchivedTasks()
repository.getArchivedTasks("01/01/1900", "12/31/2100")
repository.exportArchivedTasks("/exports/xml/task-repository.xml")
repository.exportArchivedTasks("d:\\exports\\task-repository.xml", "11/11/2000", "03/07/2200")
```

Performing deployments

From the CLI it is possible to perform deployments. Following will be two examples showing how to do this. Here is an example of how to perform a simple deployment (one where the default mappings suffice):

Performing a simple deployment

A simple deployment using a package may be performed as follows:

```
# Import package
package = deployit.importPackage('petclinic-1.0.dar')
```

```
# Load environment TEST
environment = repository.read('Environments/TEST')

# Start deployment
deploymentRef = deployment.prepareInitial(package.Id, environment.Id)
deploymentRef = deployment.generateAllDeployeds(deploymentRef)
taskID = deployment.deploy(deploymentRef).taskId
deployit.startTaskAndWait(taskID)
print "Done."
```