

Deployit Command Line Interface (CLI) Manual

January, 2011

Contents

Preface	1
Starting and Stopping	1
Logging in and Logging out	2
Help	2
Objects	3
Deployit	3
Repository	3
Factory	3
Security	3
Custom Helper Objects	3
Performing Common Tasks	3
Deploying an Application	4
Working with Archetypes	4
Discovering CIs	4
Creating the discovery starting point	5
Start Discovery	5
Store the CIs in the repository	5
Complete discovered middleware CIs	6
Adding CIs to Environments	6
Storing the CIs	6
Setting up Security	7
Managing Users	8
Performing a Simple Deployment	9

Preface

This manual describes how to use the Deployit CLI. Using the CLI, it is possible to programmatically control Deployit. Discovering middleware, setting up environments, importing packages and performing deployments can all be done via the CLI. The CLI can be programmed using the standard Python programming language.

The Deployit server must be running before accessing the CLI. See the **Deployit System Administrator Manual** for more information.

See **Deployit Reference Manual** for background information on Deployit and deployment concepts.

Starting and Stopping

To start the CLI, open a terminal window, change directory to `DEPLOYIT_HOME/cli/bin` and enter the command:

```
./cli.sh
```

on Unix or

```
./cli.cmd
```

on Windows.

The CLI will start, prompting the user for a username and password and attempting to connect to the Deployit server on `localhost` running on its standard port of `4516`.

By starting the CLI with the `-h` flag, a message is printed that shows the possible options it supports:

```
java -jar deployit-cli-<version>.jar [options...] arguments...
  -configuration VAL : Specify the location of the configuration file (deployit.conf)
  -f (-source) VAL   : Execute a specified python source file (Optional)
  -host VAL          : Connect to a specified host, defaults to 127.0.0.1 (Optional)
  -password VAL      : Connect with the specified password (Optional)
  -port N            : Connect to a specified port, defaults to 4516 (Optional)
  -username VAL      : Connect as the specified user (Optional)
```

The options are:

- `-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.
- `-f /path/to/Python/script` — starts the CLI in batch mode and instruct it to execute the specified script. 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 to connect to. The default host is `127.0.0.1`.
- `-password mypassword` — specifies the password on the command line. If the password is not specified, the CLI will enter interactive mode and prompt the user.

- `-port 1234` — specifies the port at which to connect to the server. If the port is not specified, the CLI will use default port 4516.
- `-username myusername` — specifies the username on the command line. If the username is not specified, the CLI will enter interactive mode and prompt the user.

Logging in and Logging out

When the CLI has obtained the username and password to use (either from the command line or interactively), it will attempt to contact the server at the specified (or default) address and port. If this is not successful, a stacktrace will be printed and the CLI will terminate. If the CLI is run in interactive mode and the login is successful, the CLI will display a welcome message.

Help

Help is available in the CLI by issuing CLI commands. The following message is shown when first logging in and describes the available options:

```
Welcome to the Deployit Jython CLI!
Use the 'deployit' object to interact with the Deployit server.

Deployit Objects available on the CLI
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>")

repository: Gateway to doing CRUD operations on all types of CIs
factory: Helper that can construct archetypes, CIs and artifacts
deployit: The main gateway to interfacing with Deployit.
security: Access to the security settings of Deployit.
```

Extensive help about the exact usage of the helper objects is available using the built-in CLI help.

Objects

This section describes the purpose of the various helper objects available in the CLI.

Deployit

The `deployit` helper object provides access to the main functions of the Deployit application. It allows the user to work with deployments (generate mappings, prepare and execute deployments), work with tasks (stopping, starting, canceling or aborting tasks) and some miscellaneous functionality.

Repository

The `repository` helper object allows the user to access Deployit's repository. This includes searching and performing CRUD operations on CIs in the repository.

Factory

The **factory** helper object facilitates the creation of new CIs, archetypes and artifacts.

Security

The **security** helper object allows the user to log into or out of Deployit and to create or delete users in Deployit's own repository. Administrative users can also grant or deny security permissions.

Custom Helper Objects

It is possible to install custom helper objects in the CLI which are loaded on CLI startup.

To install CLI extensions follow these steps:

1. **Create a directory called ext.** This directory in the same directory from which you start the CLI.
2. **Copy Python scripts into the ext directory.**
3. **Start the CLI.** The CLI will load and execute all scripts with the **py** or **cli** suffix found in the extension directory.

Performing Common Tasks

This section describes common tasks you can perform with the CLI.

Deploying an Application

Importing and deploying a package using the default mappings is easy using the CLI. The following commands perform a simple version of this scenario:

```
importedPackage = deployit.importPackage("PetClinic-ear/1.0")
server = repository.create("Infrastructure/dummy1", factory.configurationItem("DummyJeeServer", {"name":
env = repository.create("Environments/env1", factory.configurationItem("Environment", {"members": [serv

taskId = deployit.deploy(importedPackage.id, env.id)
```

Note that the last command returns immediately, returning the **taskId** of the started task. The Deployit server will perform the deployment asynchronously in the background.

Working with Archetypes

Archetypes are a powerful feature of Deployit that allows easy creation and maintenance of lots of similar data. For a complete explanation of the archetype functionality, see the **Deployit Reference Manual**.

The following script demos a common way to use archetypes:

```

from com.xebialabs.deployit.core.api.dto import Archetype
from com.xebialabs.deployit.core.api.dto import ConfigurationItem

# create an archetype
archetype = Archetype()
archetype.configurationItemTypeName = "com.xebialabs.deployit.ci.Host"
archetype.values = {"address":"localhost","username":"deployit","password":"admin"}

# add the archetype to repository
createdArchetype = repository.create("Archetypes/hostArchetype", archetype)

# create a host using 'hostArchetype'
host = ConfigurationItem.fromArchetype(createdArchetype.id)
host.values = {"operatingSystemFamily":"UNIX","accessMethod":"SSH_SUDO"}
createdHost = repository.create("Infrastructure/hostDefinedWithArchetype", host)

# modify the archetype. The CIs used in deployments will have the new value.
archetype.values = {"address":"127.0.0.1"}
repository.update("Archetypes/hostArchetype", archetype)

```

Discovering CIs

To help in setting up Deployit, it is possible to discover middleware in your environment. Deployit will scan your environment and create CIs based on the configuration it encounters. The discovered CIs help you in setting up your infrastructure, but they may not be complete. Some CIs contain values that can not be discovered, like passwords. These will have to be entered manually. Because discovery is part of the Deployit plugins, the exact discovery functionality available varies depending on your middleware platform.

The discovery process consists of the following steps:

1. Create a CI representing the starting point for discovery (often a *Host* CI).
2. Start discovery passing in the starting CI.
3. Store the CIs in the repository.
4. Complete the discovered CIs by providing missing information.

Optional steps:

5. Add the discovered CIs to an environment.

Creating the discovery starting point

The discovery starting point is a CI that indicates where the discovery process should start. This starting point specifies at least the host that the discovery should start at. Depending on the middleware you are trying to discover, additional parameters may be needed.

Example:

```
# create ci with required discover parameters filled in
wasHost = repository.create("Infrastructure/rs94asob.k94.corp.nl (DEV99)",
    factory.configurationItem("Host",{ "address": "was-61", "username": "root",
    "password": "root", "operatingSystemFamily": "UNIX", "accessMethod": "SSH_SFTP" }))

aDmManager = factory.configurationItem("WasDeploymentManager",
    {'host': 'Infrastructure/rs94asob.k94.corp.nl (DEV99)',
    "wasHome": "/opt/ws/6.1/profiles/dmgr", "username": "wsadmin", "password": "wsadmin"})
aDmManager.id="c-ws-dev99"
```

Start Discovery

The Deployit CLI discovery functionality is synchronous, that means that the CLI will wait until the discovery process finishes.

The command to start discovery is:

```
discoveredRepositoryObjectsContainer = deployit.discover(aDmManager)
discoveredCIs = discoveredRepositoryObjectsContainer.objects;
```

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

Store the CIs in the repository

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:

```
repository.create(discoveredRepositoryObjectsContainer)
```

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.

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('Environment')
```

Add the discovered CIs to the environment:

```
devEnv.values['members'] = [ci.id for ci in discoveredCIs]
```

Note: 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('Environments/Dev', 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**.

Storing the CIs

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.

This is the command to do this:

```
repository.create(discoveredRepositoryObjectsContainer)
```

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

Setting up Security

This section continues the example from the **Deployit System Administration Manual**.

In a typical medium to large size company, there several different groups of people that perform tasks related to deployments. There are administrators that install, test and maintain hardware, there are deployers that deploy applications to development, test, acceptance and production environments. And finally there are the developer who build these applications.

Translating this into Deployit terms:

- **administrators:** permission to create, update and delete infrastructure as well as permission to create, update and delete environments. (all handled by `_infrastructure#management` permission)
- **deployers:** permission to import new applications (*import#initial*), deploy to DEV, TEST and view PROD (*deploy#initial* and *deploy#upgrade* on environments DEV and TEST, *read* rights on environment PROD).
- **senior deployers:** permission to import new applications (*import#initial*), deploy to DEV, TEST and PROD (*deploy#initial* and *deploy#upgrade* on environments DEV, TEST and PROD).
- **developers:** permission to import new versions of existing applications (*import#upgrade*) and to upgrade existing deployments (*deploy#upgrade*).

NOTE: you have to be logged in as user 'admin' to perform these commands.

This script performs the setup:

```
#
# Sample security setup.
#

# Create test data: environments DEV, TEST, PROD
# ...

# Create users: administrator, deployer, srdeployer, developer
# ...

# Grant login rights
security.grant("login", 'administrator')
security.grant("login", 'deployer')
security.grant("login", 'srdeployer')
security.grant("login", 'developer')

# Configure administrator privileges
security.grant("repo#edit", 'administrator')
security.grant("read", 'administrator', ['Infrastructure'])
security.grant("read", 'administrator', ['Environments'])
# To grant rights to specific environments, use:
#security.grant('repo#edit', 'administrator', ['Environments/PROD'])

# Configure deployer privileges
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'])

# In order to allow deployer to see the deployments to PROD, he needs rights on both the
# environment and the infrastructure in the environment.
prodEnv = repository.read('Environments/PROD')
security.grant("read", 'deployer', [prodEnv.id] + prodEnv.values['members'])

# Configure sr deployer privileges
security.grant("import#initial", 'srdeployer')
security.grant("import#upgrade", 'srdeployer')
security.grant("deploy#initial", 'srdeployer')
security.grant("deploy#upgrade", 'srdeployer')

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

# The developer user has permission to import new versions of all applications.
# To restrict to new versions of a specific application, use:
#security.grant('import#upgrade', 'developer', ['Applications/PetClinic'])
```


After these commands, the security setup will match the intended setup described above.

Managing Users

Note: Deployit is only able to create, modify or delete users in it's own repository.

Here is an example of how to deal with users in Deployit:

```
# Create user
devUser = security.createUser("developer", "developer")

# Modify user (for instance, change password)
# Note that the password will always be shown as 8 *'s in the user CI
devUser.password = 'newpassword'
security.modifyUser(devUser)

# Login as the new user
security.logout()
security.login('developer', 'newpassword')

# Login as admin
security.logout()
security.login('admin', 'admin')

# Delete user
security.deleteUser("developer")
```

Performing a Simple Deployment

Here is an example of how to perform a simple deployment (one where the default mappings suffice):

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

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

# Start deployment
deployit.deployAndWait(package.id, env.id)
print "Done."
```