

Deployment

An interface to an application deployment

Table of contents

1 Overview.....	2
1.1 Problem.....	2
1.2 Discussion.....	2
1.3 Example.....	3
1.4 Structure.....	3
1.5 Check List.....	4
2 Design.....	4
3 Constraints.....	4
3.1 Allowed Child Dependencies.....	4
3.2 Allowed Parent Dependencies.....	4
3.3 Allowed Property Values.....	4
4 Commands.....	5
4.1 Configure.....	5
4.2 Docs-Generate.....	5
4.3 Docs-Verify.....	6
4.4 Get-Properties.....	6
4.5 Install.....	7
4.6 Packages-Install.....	7
4.7 Properties.....	8
4.8 Register.....	8
4.9 Register-Dependency.....	9
4.10 Update.....	10

1. Overview

Deployment: *An interface to an application deployment*

- Define an object that encapsulates the life-cycle of software deployment.
- Separate the concerns of software installation from software configuration.
- Encourage abstracting software deployments so that a single definition can be used in multiple environment contexts.

1.1. Problem

We want to design a reusable software deployment procedures, but dependencies between software packages and environment-specific configuration demonstrate that an inflexible "hard-coded" result often ensues.

1.2. Discussion

The post development application life-cycle is dominated by migrating software from one environment to another. Primarily, the migration tasks fall into two categories:

- software installation: locating and distributing the needed files to the designated host
- configuration setup: customizing the software installation using the needed application and environment specific parameters

Because a software deployment can differ depending on context, the installer must often be aware of both the constant and variable details and then mentally translate the steps of the deployment process accordingly. The figure below illustrates the different dimensions of information a person deploying an application must consider.

The differences between deployments within their specific application environment context undermine a desirable goal to have a single procedure that can be used anywhere in any environment and any software version. This jumble of configuration and procedural information seem intertwined and difficult to separate.

An alternate approach would be to introduce an abstraction of deployment that breaks the configuration and installation problem into separate concerns and encapsulate the contextual application environment differences.

This offers several advantages: Configuration and installation are decoupled from one another, many mappings between environment contexts and configuration and installation can be maintained and the entire deployment procedure can be extended by defining derived classes.

Further, the deployment abstraction defines a standard structure for software deployment as well as, a standard deployment life-cycle.

As a standard structure, the deployment abstraction accounts for key parameters like installation root and deployment base directory, two properties which may be the same in some cases, but should be separable to distinguish directories where static executable files reside from those where runtime files supporting configuration and logging exist.

Along similar lines, the deployment structure organizes dependency, configuration and environment information into a mapped context that facilitates abstracting configuration files and procedures. The abstracted configuration and procedures can then take form as template files increasing their reuse and decreasing the maintenance of environment specific variations.

Finally, the deployment abstraction defines a standard update life-cycle for the deployment processes promoting the configuration and installation steps into formalized and standard operations.

1.3. Example

The Deployment type defines an object that maintains environment and dependency context, as well as, one that coordinates the installation and configuration processes. A deployment of an Apache web server is a simple yet representative scenario wherein a set of package dependencies, configuration files and environment specific parameters must be combined to produce a working httpd deployment.

The figure below illustrates these parts. A Deployment named "apache" is defined to depend on two packages - httpd-2.0.1.tgz and mod_jk.zip, and corresponding configuration files, httpd.conf and jk.conf. The environment specific context is provided to the Deployment object which in turn uses it when parameterizing the configuration and package installation steps.

1.4. Structure

The deployment abstraction breaks down into a common design pattern comprised of several types

- Deployment: provides parameters and procedures that govern the deployment life-cycle.
- Package: encapsulates the package format specific installation procedures (e.g., extraction and verification).
- Document: encapsulate the generation of configuration files from templates.

1.5. Check List

1. Identify the package dependencies and configuration files required for the deployment
2. Discern the parameters that differ across environments that affect the configuration and installation process.
3. Encapsulate those parameters as attributes of a new class of Deployment.
4. Delegate the package installation and doc generation to objects of those types.

2. Design

Super Type
Managed-Entity

Role	Concrete. (Objects can be created.)
Instance Names	Unique
Notification	false
Template Directory	
Data View	Children, proximity: 1
Logger Name	Deployment

3. Constraints

3.1. Allowed Child Dependencies

- DeploymentSetting
- FilePath

3.2. Allowed Parent Dependencies

- [Deployment](#)
- Node

3.3. Allowed Property Values

Property	Allowed Values	Default	Enforced	Description
manages-deployme	<ul style="list-style-type: none"> • false • true 		true	<i>Deprecated</i>

4. Commands

Note:

Commandline options displayed in square brackets "[]" are optional. If an option expects arguments, then angle brackets are shown after the option "<>". Any default value is shown within the brackets.

4.1. Configure

runs the configuration cycle

The Configure command manages the configuration phase of the deployment lifecycle and invokes the [Docs-Generate](#) to generate any configured template-based documents. The typical usage of Configure, is to customize a Deployment's configuration according to its environmental context after the package installation step has occurred. Subtypes can override the Configure workflow to include additional configuration related commands.

Usage

Configure

4.1.1. Workflow

1. [Docs-Generate](#)

4.2. Docs-Generate

generates all defined docs

The role of the Docs-Generate command is to generate any template based documents. Typically, these are configuration file templates that reference property names provided by the object context. The implementation provided by Deployment is to iterate over the set of properties of the pattern: `^document.(.*)template$` and to call the command, Doc-Generate for each one.

If `-archivedir <directory>` option is used then any files that were created from the previous run of Docs-Generate are copied to that that directory.

Usage

Docs-Generate [-archivedir <\${entity.instance.dir}/var>]

4.2.1. Options

Option	Description
archivedir	<i>dir to store current output docs</i>

4.3. Docs-Verify

verifies all generated docs

The Docs-Verify command generates all the defined template based documents to a temporary location and then compares them to the ones last generated by [Docs-Generate](#). If any differences are found an error is raised. This command can be useful to determine if modifications occurred outside the framework.

Usage

```
Docs-Verify
```

4.4. Get-Properties

Gets the properties for the object

The Get-Properties command requests information about the object from the server and saves its local instance directory in the CTL project depot.

The `-direction` and `-proximity` arguments use command settings defined with the module for defaults.

Internally, Get-Properties uses an Ant task to communicate with the server to retrieve object model information which is transformed into Java properties key/value pair format. To reduce load on the server, the Get-Properties command only requires the server to generate data if the object revision has changed since the last invocation. This check can be defeated through the use of the `-force` flag which will first call Remove-Properties.

Usage

```
Get-Properties [-destfile <${entity.properties.file}>]
[-direction <>] [-force] [-print] [-proximity <>]
```

4.4.1. Options

Option	Description
destfile	<i>file to save property data</i>
direction	<i>dependency view direction (internal, external, bidirectional)</i>
force	<i>Remove old properties file first</i>
print	<i>flag specifying Properties command should run after properties are saved</i>
proximity	<i>dependency view proximity</i>

4.5. Install

Installs the object, and its modules

Installs the object into the local CTL project depot. The Install command is essentially a command workflow that calls the Install-Module and [Get-Properties](#) command (though there are boolean flags to skip those commands).

Usage

```
Install [-basedir <${entity.instance.dir}>] [-depot
<${context.depot}>] [-name <${context.name}>] [-nodir]
[-nomodule] [-noproperties] [-subdirs <>] [-type
<${context.type}>]
```

4.5.1. Options

Option	Description
basedir	<i>instance directory</i>
depot	<i>project depot</i>
name	<i>object name</i>
nodir	<i>do not create directories</i>
nomodule	<i>do not update module</i>
noproperties	<i>do not update properties</i>
subdirs	<i>directory list</i>
type	<i>object type</i>

4.6. Packages-Install

installs all the package dependencies

The Packages-Install command iterates over all the package object dependencies declared in the object context, and for each one, calls [Package](#)'s install command. Packages-Install parses the object context for properties matching the pattern:

```
^package\.(${opts.package_type})\.(.*)\.package-install-rank.
```

It is possible to only install packages of a certain type using the `-package_type` option.

Usage

```
Packages-Install [-package_type <[^\.]*>]
```

4.6.1. Options

Option	Description
packagetype	<i>type of packages to install</i>

4.7. Properties

prints object properties.

The Properties command prints out the contents of the object properties file generated by the [Get-Properties](#) command.

By default, the command uses the `-format pretty` option to present the property data in a human readable form. The `-format plain` option will print the property data in the normal Java property key/value form.

The `-detail` flag will include detail about child object data.

Usage

```
Properties [-detail] [-format <pretty>]
```

4.7.1. Options

Option	Description
detail	<i>print more detail</i>
format	<i>output format. pretty or plain</i>

4.8. Register

Registers the object

The Register command registers an object in the project model maintained in the server. The registration information will include the values specified by the `depot`, `type`, `name` as well as `-basedir`, `-installroot` and `-node`. The `-node` argument specifies what node this object should be a child dependency to.

If the `-install` flag is set, the [Install](#) command is also run.

Usage

```
Register [-basedir <>] [-depot <>] [-description <>]
[-install] [-installrank <>] [-installroot <>] [-name <>]
[-node <${framweork.node}>] [-type <>]
```

4.8.1. Options

Option	Description
basedir	<i>base directory</i>
depot	<i>project depot name</i>
description	<i>object description</i>
install	<i>after registration, run Install command</i>
installrank	<i>install rank</i>
installroot	<i>install directory</i>
name	<i>object name</i>
node	<i>after registration, bind this object to the specified node</i>
type	<i>type name</i>

4.9. Register-Dependency

Registers an object-to-object dependency

The Register-Dependency command defines a dependency relationship between two objects. The `-relation` option specifies what direction the dependency should be defined. When `child` is set, then the object is made a child of the `-type` and `-object`. When `parent` is set, then the object is made a parent of the `-type` and `-object`.

The Register-Dependency command will check to see that both objects exists before attempting to define the dependency and will fail if either does not exist.

To make a dependency like: (Deployment) deployment -> (Setting) aSetting, run:

```
ctl -p project -t Deployment -o deployment -c Register-Dependency -- \
  -type Setting -o aSetting -relation child
```

To make a dependency like: (Node) aNode -> (Deployment) deployment, run:

```
ctl -p project -t Deployment -o deployment -c Register-Dependency -- \
  -type Node -o aNode -relation parent
```

Usage

```
Register-Dependency -object <> [-relation <child>] -type <>
```

4.9.1. Options

Option	Description
object	<i>object name</i>
relation	<i>parent or child (default)</i>
type	<i>type name</i>

4.10. Update

executes the deployment cycle

Update is a workflow that executes the commands that comprise deployment lifecycle: Packages-Install and Configure.

Usage

Update

4.10.1. Workflow

1. [Packages-Install](#)
2. [Configure](#)