Content Deployment Library

Table of contents

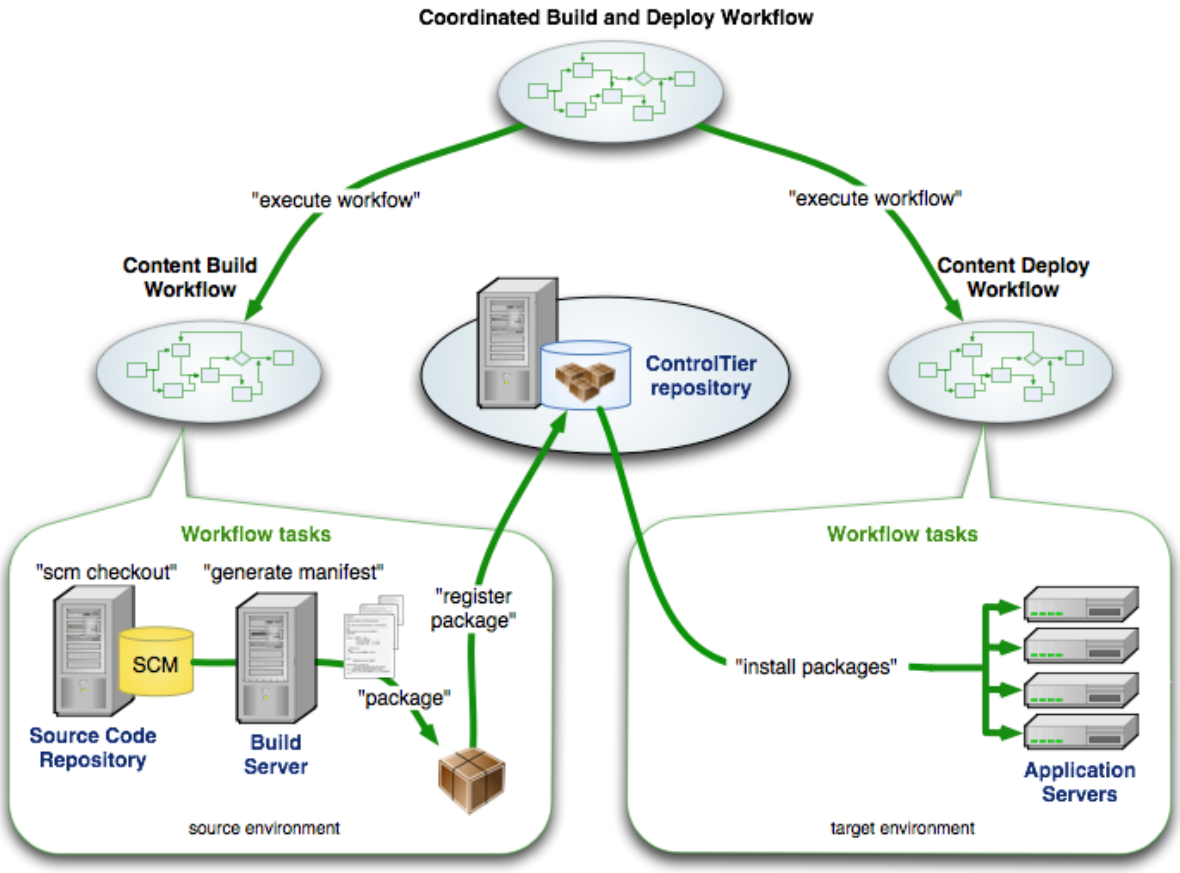
1. Welcome to "Content Deployment Library"

This library provides a simple framework that automates the promotion of content from one environment to another. This automation extracts content from a source code repository, packages it, and deploys it to target environments using one standardized end-to-end process.

What is it?

This is a library of command modules that build on the ControlTier automation base types and establish an end-to-end build and deployment process. This process is defined in terms of command workflows. The figure below describes there are three primary workflows:

1. Coordinated Build and Deploy: This workflow defines the end-to-end process and a single command to execute it called "BuildAndDeploy".
2. Content Build: This workflow is responsible for checking for changes in the SCM repository, checking out or updating files from the repository and creating a strucutred package to deploy the files.
3. Content Deploy: The deploy workflow manages the distribution and installation of the content packages.



Why use it?

This library builds on the ControlTier service provisioning platform to offer a number of notable features for organizations where content migration and promotion is an important part of the application lifecycle and is a frequent and critical process:

- *End-to-end automation:* This framework ties together many individual procedures that are often run independently. Building on the ControlTier base types, the Content Library offers a single command that: checks code out (both full checkout and incremental updates), packages the content, manages package dependencies and deploys packages to multiple hosts.
- *Rollback:* All file distributions are packaged and versioned. You can rollback to any previous version. Files that will be overwritten by new versions are automatically backed up in their own package and stored in the repository and can be used to restore a previous state.
- *Task delegation and self-service:* You end up with schedulable "jobs" letting a less

knowledable person run the checkout-deploy process on demand or at defined periods.
- *Standardized operational interface: End users of the Content Library can rely on simple and predictable commands no matter what environment, source code module, or application.*
- *Structured Packaging:* The content of each package is declared in a manifest file. This file is used during the installation process to ensure integrity, and facilitate rollback.
- *Reporting:* Every deployment done via the Content Library is logged. Reports show who deployed what where and when. The full output of the job is saved can can be used for auditing later.
- *Graphical interfaces:* All operational tasks can be done via a graphical interface. ControlTier's JobCenter is used to run commands while Workbench can be used to review the current deployment environment dependencies.
- *Security:* Using the ControlTier access control infrastructure, you can control who updates what when and where.
- *Extendability:* The library offers a workinng end to end process but it is not monolithic. You can override various parts of the process via sub-typing.

Why not use a shell script?

You might already have a scripted procedure to check out source files and use a utlity like scp, rsync or rdist to distribute the files. Why not continue to follow that approach? If you have a simple environment where the process does not change then you probably don't need this library. But, if you have to run these kinds of processes for multiple applications and would benefit from some of the features listed above, then it is worth the effort to give it a try.

We have noticed several disadvantages of the typical scripted approach:

- Hard to delegate: It may be difficult to delegate running the script to someone not technical or familiar with the shell environment.
- Hard to generalize: The orginal script might be sufficient for the initial use case, but uses a lot of hard coded values that make it hard to use for new use cases.
- Unweildly: It might be a single script that is hard to customize with special behavior as needs evolve.
- Lacks enterprise features: Simple shell scripts are typically minimal bare bones implementations. It is difficult to imagine investing a great deal of time and effort in them so they include essential enterprise features (eg, reporting, logging, security) offered by a specialized and advanced framework.

Getting started

You can start using the this library by following the steps of the documentation pages listed below:

1. Install: Describes how to download and install the library
2. Configure: Describes how to configure the library to define new "BuildAndDeploy" job
3. Run: Explains how to run the job either via the graphical JobCenter application or by command line.

Next: Install #

2. Install

2.1. Install

Overview

This document describes the installation steps necessary to use the library.

2.1.1. Step #1: Install the ControlTier platform software

This library assumes you have installed the latest stable release of the ControlTier platform software on a a designated server host and one or more client hosts. Refer to the general installation procedures for more info.

You will need to know the URL to the Workbench and JobCenter applications.

2.1.2. Step #2: Download the library archive

Binary distributions of the library can be found in the "File Releases" section of the ModuleForge Download page on Sourceforge. The package is called "Content Module Library" and will be named something like: content-seed-*x.y*.jar where "x.y" denote the version.

It is always suggested to download the latest release.

2.1.3. Step #3: Choose or create a project

All work is done within the context of a "project". You may already have a project in mind, or you may wish to create a new one just for the use of this library.

Note:

If you are new to ControlTier and are not sure about projects see the Projects section in the ProjectBuilder tutorial.

1. Navigate to the Admin page. (eg., go to the URL: http://localhost:8080/itnav/do/menu/Admin
2. Press the "Create Project" button. (eg., go to the URL:

http://localhost:8080/itnav/do/projects/Input
3. Fill out the form and press "Create" button. It takes a few minutes for the new project to be created.

2.1.4. Step #4: Load the library archive

Once you have chosen the desired project, you can load the library into that project.

Be sure you have already logged into Workbench and selected the desired project where you want the library loaded. If you just created a new project, you are all ready.

1. Navigate to the Admin page. (eg., go to the URL: http://localhost:8080/itnav/do/menu/Admin
2. Press the "Import Seed" button. (eg., go to the URL: http://localhost:8080/itnav/do/projects/ImportSeedInput
3. Locate and select the content-seed-x.y.jar file in the file chooser. This is the same file you downloaded in step #2.
4. Press the "Import" button. It takes a few minutes for Workbench to load the library.

Once the library has been installed into your chosen project the next step is to configure these modules for use.

Next: Configure #

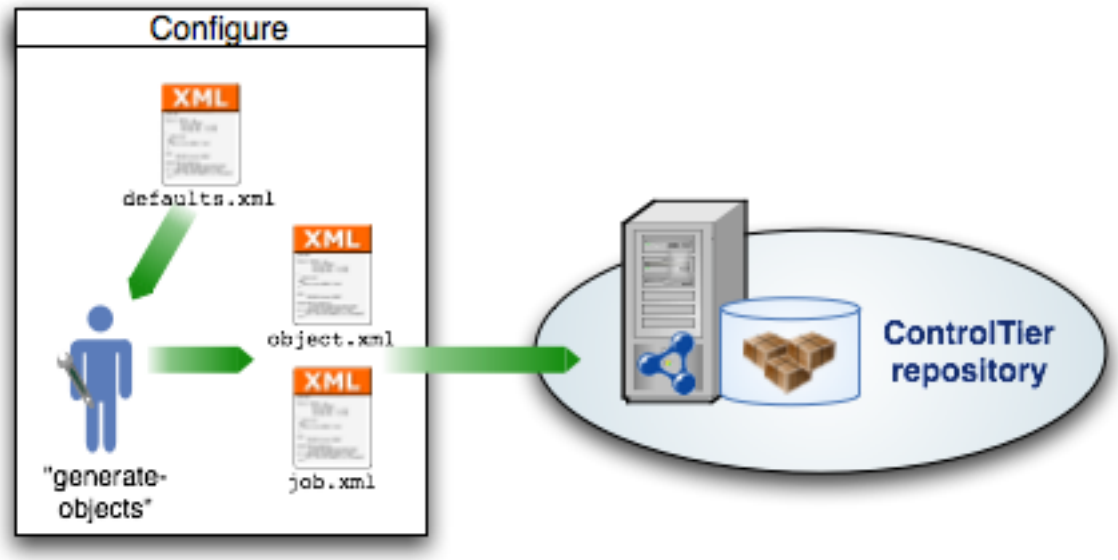
3. Configure

3.1. Configure

Overview

This document describes how to configure a project to use the content library.

The diagram below illustrates the configuration is driven by a `defaults.xml` file. This file is used as input by a command `generate-objects` which in conjunction with template files, produces two output files: `object.xml` and `job.xml`.



Note:

Be sure you have already installed the ControlTier software, chosen a project and loaded the library archive. See the Install page for instructions.

3.1.1. Step #1: Edit defaults.xml

The defaults.xml file contains all the essential environment-specific information needed by the library. It answers questions like: What host will the files be checked out, and to which directory? What host will the files be distributed to and to which directory?

Open a text editor or better yet an XML editor. Cut and paste the contents of the XML shown below and save it to disk.

```
<defaults>
  <default>
    <node>${framework.node}</node>
    <name>${opts.name}</name><!--this is passed in as the -name arg-->
  </default>
  <updater>
    <node>${defaults.default.node}</node>
    <builder>
      <!--hostname where build and packaging to run-->
     <node>buildhost</node>
      <!--directory where files will be checked out on buildhost-->
     <basedir>/path/to/workspace</basedir>
      <!--directory where package archive will be written on buildhost-->
     <targetdir>/path/to/targetdir</targetdir>
```

```
      <package>
        <!--directory where package archive will be extracted on
deployhost-->
        <installroot>/path/to/installation/dir</installroot>
      </package>
      <scm>
        <!--scm checkout string. eg, if svn may be a url-->
        <connection>uri</connection>
        <!-- module name relative to connection string-->
        <module>modulename</module>
      </scm>
    </builder>
    <site>
      <node>${defaults.updater.node}</node>
      <deployment>
        <!--hostname where packages will be extracted-->
        <node>deployhost</node>
        <name>${opts.name}Target</name>
        <basedir>/path/to/installation/dir</basedir>
        <!--directory where package content will be extracted-->
<installroot>${defaults.updater.builder.package.installroot}</installroot>
      </deployment>
    </site>
  </updater>
</defaults>
```

Some of the defaults can be taken as-is but the tags that are bold are ones that you must change, supplying the required local information.

Note:

A copy of the defaults.xml template file can be found in the WebDAV under your project:
http://localhost:8080/webdav/*project*/modules/ContentProjectBuilder/templates/defaults.xml

3.1.2. Step #2: Configure library objects

Register and install a ContentProjectBuilder:

```
ad -p project -m Deployment -c Register -- \
      -name name -type  ContentProjectBuilder \
      -basedir $CTIER_ROOT/src/project -installroot
$CTIER_ROOT/target/project \
      -install
```

Copy the defaults.xml you created in Step #1 to $CTIER_ROOT/src/*project*/defaults.xml

Run the generate-objects command:

```
ad -p project -t ContentProjectBuilder -o name -c generate-objects -- \
      -name aName \
      -defaults $CTIER_ROOT/src/project/defaults.xml -upload
```

Before you can run the job, it is necessary to deploy the objects. This is done via the AntDepo command, `depot-setup`. On the adminstrative node, run:

```
depot-setup -p project -a install
```

After this command successfully completes, a new set of objects will be loaded into the ControlTier repository. You can view them via ContentProjectBuilder's find-objects command:

```
ad -p project -t ContentProjectBuilder -o name -c find-objects -- \
      -name aName
```

3.1.3. Step #3: Upload job definition

The `generate-objects` command run in Step #2 will have produced a job.xml file with a filename *aName*-job.xml. This file can be used to define a new job in the JobCenter application.

1. Login to JobCenter (e.g, go to URL: http://localhost:9090/jobcenter/menu/index
2. Press the "Create a new Job..." button
3. Press the "Upload job.xml" button
4. Locate and select the file, *aName*-job.xml, produced by `generate-objects` in the file chooser
5. Press "Save" button

The new job will be listed on the home page of JobCenter.

3.1.4. Optional Step: Check-in generated files

It is considered best practice to maintain the files generated by `generate-objects` in a source code repository.

Next: Run #

4. Run

4.1. Run

Overview

This document describes how to run a content `BuildAndDeploy` workflow job or command and is pertinent to users responsible for releasing content changes from the source code repository to targeted deployment environments.

By running one of these workflows, pending changes in the SCM repository will be

packaged up and deployed to the configured target node and directory.

For a new job to be established, the steps covered in the Configure section will already have been completed yielding a new content `BuildAndDeploy` workflow.

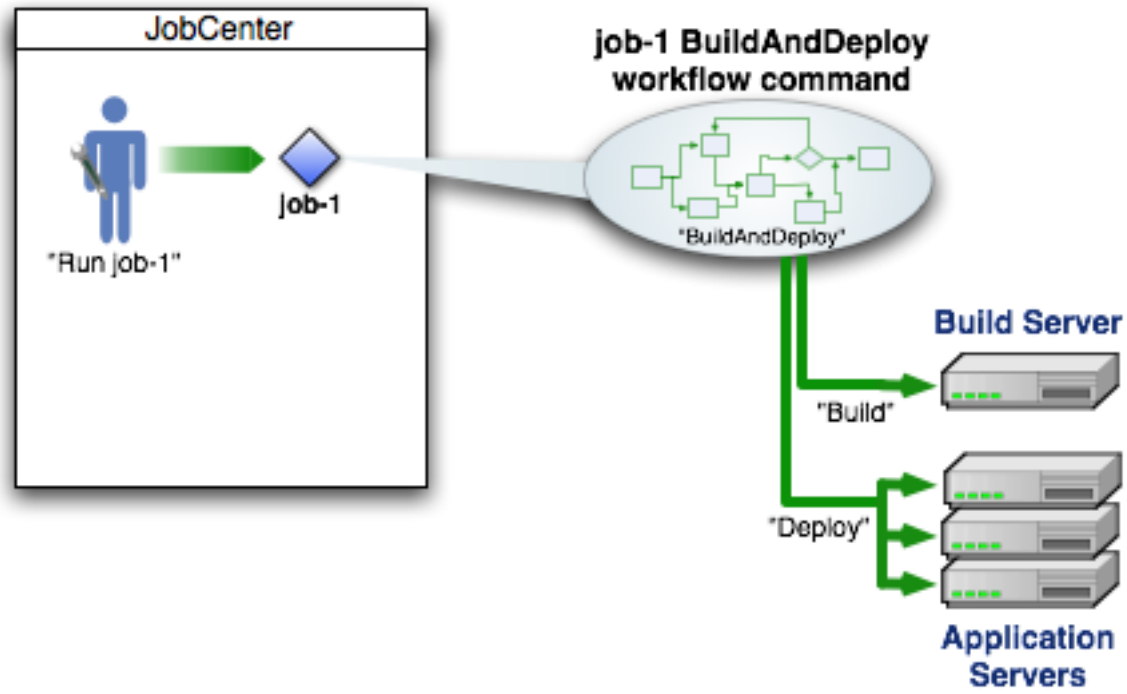
Their are two interfaces available to execute the content `BuildAndDeploy`:

- Using JobCenter, the web-based graphical application,
- Using AntDepo's `ad` shell command

Instructions for running the BuildAndDeploy command using either interface are explained below.

4.1.1. Execute via JobCenter

In Configure Step #3, a new job will have been defined for the BuildAndDeploy process. After logging into JobCenter a list of the defined jobs will be displayed. Choosing and running a job will execute the underlying `BuildAndDeploy` command.



The general steps to using JobCenter to operate content `BuildAndDeploy` are listed:

1. Login to JobCenter

2. Identify the desired job
3. Run the job
4. Customize a report

4.1.2. Execute via 'ad'

An alternative to executing the BuildAndDeploy command via JobCenter is to execute it directly via the `ad` shell command. The general usage is shown below:

```
ad -p project -t ContentUpdater -o name -c BuildAndDeploy -- \
      -buildstamp buildstamp
```

A typical convention is to use the date and time as the `-buildstamp` argument. For example:

```
ad -p project -t ContentUpdater -o name -c BuildAndDeploy -- \
      -buildstamp 200711071500
```

If you are an experienced AntDepo user, you may also know how to run individual parts of the build and update workflow by running the appropriate command from one of the subordinate commands. For example, to run just the Deploy:

```
adminhost$ ad -p project -t ContentUpdater -o name -c Deploy
```

Or to run just the build, you can execute the Builder directly:

```
buildhost$ ad -p project -t ContentBuilder -o name -c Build -- \
      -buildstamp buildstamp
```

4.1.3. Rolling back

A nice feature of the Content library is its support for rollback. Before a ContentPackage is deployed, the library first checks if existing local files will be overwritten by new ones in the package. If any are detected, a "rollback" package is generated and stored on the repository. This package can be used to later restore the original files.

General Usage

Rollback is accomplished by running two commands on the admin host:

1. runChangeDependencies: This configures the deployments to depend on the rollback package:
   ```
   ad -p project -t ContentUpdater -o name -c runChangedepencies -- \
         -buildstamp rollback-buildstamp
   ```
2. Deploy: This deploys the rollback package to the ContentDeployments:
   ```
   ad -p project -t ContentUpdater -o name -c Deploy
   ```

In depth descriptions of the Content library commands can be found in the Reference section of the documentation.

Next: Reference #

5. Reference

5.1. Reference

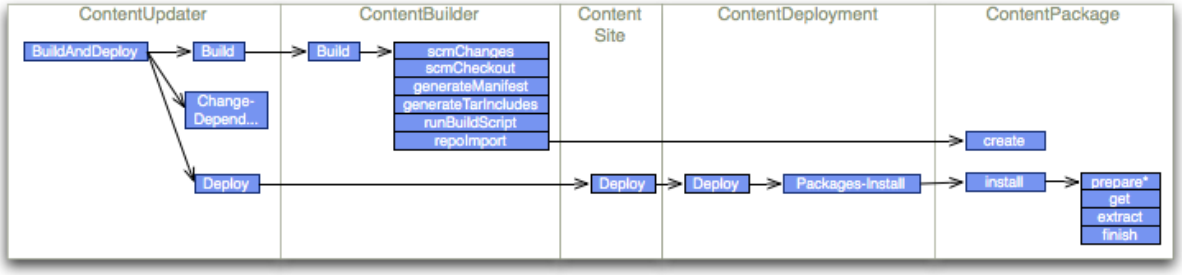
Overview

This section is useful to developers interested how the library works and users interested in knowing all the command syntax offered by the modules in this library.

This library builds on the standard ControlTier "process building blocks" - Package, Builder, Updater, Site and Deployment - each a separate workflow that compose into the single end-to-end workflow command, `BuildAndDeploy`.

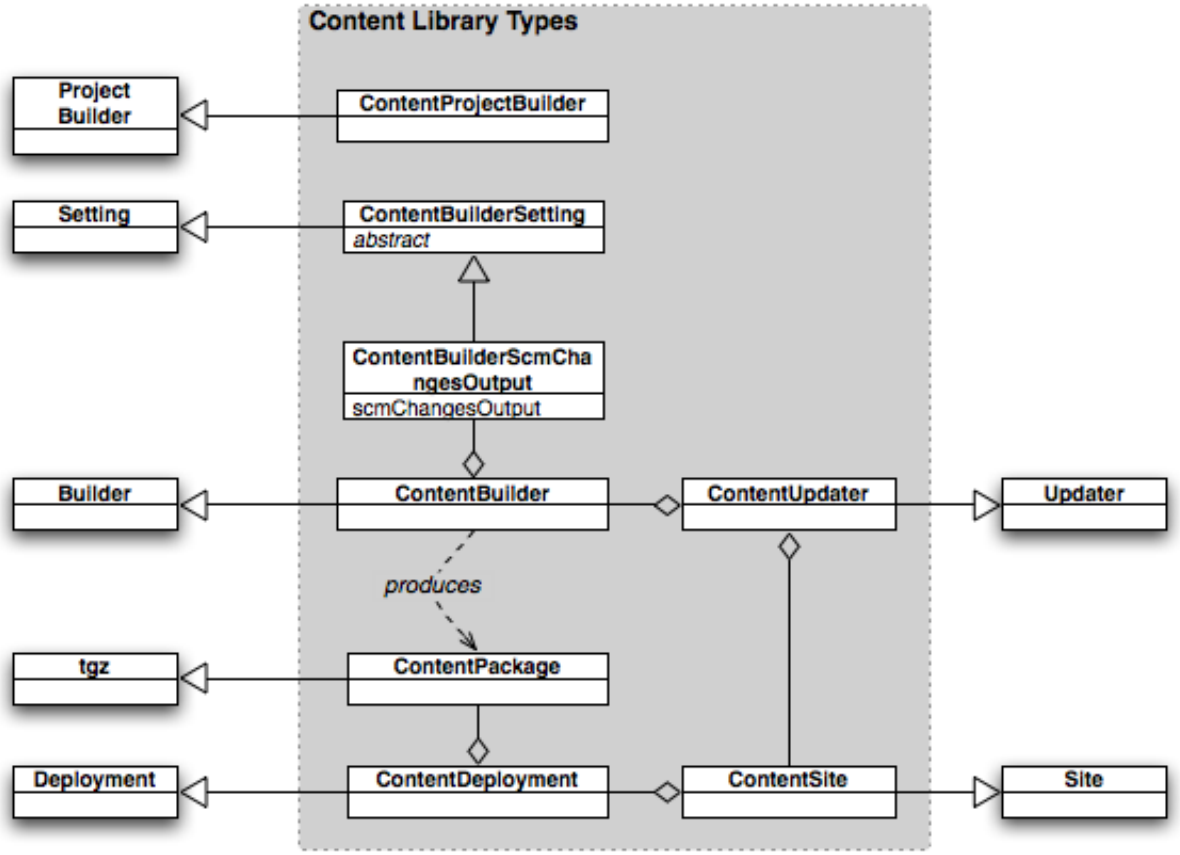
General Command Workflow

The sequence diagram below describes the workflow structure defined by the modules in the library.



Type Model

The diagram below describes the inheritance and composition hierarchies defined by the types in the library.



All the attributes and commands are defined in the Type Reference section of the documentation. There you will find a document page for each type.

Next: Type Reference #

5.2. Types

5.2.1. Type Reference

- ContentBuilder: Builds a ContentPackage archive
  - ContentBuilderScmChangesOutput: The results of the scmChanges command is stored in this file.
  - ContentBuilderSetting: A ContentBuilder setting.
- ContentDeployment: manages content deployment
- ContentPackage: A content package
- ContentProjectBuilder: Builds and manages projects that use the content library

  - ContentProjectBuilderDefaults: file containing project defaults properties
  - ContentProjectBuilderSetting: A ContentProjectBuilder setting.
  - ContentProjectBuilderTemplateDir: file containing project template files
- ContentSite: A content deployment site
- ContentUpdater: Establishes the end-to-end build and deployment of ContentPackage archives.

5.2.2. ContentProjectBuilder

5.2.2.1. Overview

ContentProjectBuilder: *Builds and manages projects that use the content library*

ContentProjectBuilder is derived from ProjectBuilder providing the ability to manage a ControlTier project. If you are new to ProjectBuilder see the tutorial starting with: Getting started using ProjectBuilder

The following sections describe how to obtain, install and use the Content Module Library.

Obtaining the content library

Follow the steps below to obtain and install the content library.

1) Download the desired library jar file from Sourceforge:

Content Module Library file release

2) Run the `Register` command specifying the following arguments:

```
ad -p project -t ProjectBuilder -o content -c Register -- \
        -basedir /path/to/basedir -installroot /path/to/targetdir
```

3) Run the `load-library` command specifying the following arguments:

```
ad -p project -t ProjectBuilder -o content -c load-library -- \
       -jar /path/to/downloaded/content-version.jar
```

The destination project should now have the content library types loaded.

Using the content library

If you do not have the content types loaded, see the preceding section.

After the content library is installed, begin using the types in the library beginning with ContentProjectBuilder:

1) Run the Register command specifying the following arguments:

```
 ad -p project -t ContentProjectBuilder -o name -c Register -- \
        -basedir /path/to/module/srcdir -installroot
/path/to/build/targetdir -install
```

2) Run the generate-objects command specifying the following arguments:

```
 ad -p project -t ContentProjectBuilder -o name -c generate-objects -- \
        -name aName -defaults /path/to/your/defaults.properties -load
```

The `-name` argument is used as the name for each generated object. The `-load` flag specifies to have loaded the object definitions loaded on the server.

3) Run the depot-setup command to deploy the objects in AntDepo:

```
depot-setup -p project -a deploy
```

The objects are now ready for use either via the AntDepo CLI command, `ad`, or via JobCenter webapp GUI.

The objects are now ready for use either via the AntDepo CLI command, `ad`, or via JobCenter webapp GUI.

4) Run the `BuildAndUpdate` command that will use the newly generated set of objects:

```
ad -p project -t ContentUpdater -o aName -c BuildAndUpdate -- \
        -buildstamp buildstamp
```

Note:

Optionally, you can upload the job definition to Job Center. The generate-objects command will have also generated a *name*-job.xml. Login to JobCenter and push the "Create a new Job" button and upload the *name*-job.xml file.

The generate-objects command reads two template files and a defaults.properties file to generate two working files that can be used to load in the server. Without specific arguments, generate-objects will use templates from its templates directory as defined by the attribute `templateDir`.

Example:

```
     ad -p project -t ContentProjectBuilder -o object -c generate-objects
--\
        -name aName -defaults /path/to/defaults.xml -load
```

Yields an object model like so in the *project*:

```
       +aName [ContentUpdater]
         |
         + aName [ConentBuilder]
         |  |
         |  |-aName [BuilderPackageInstallroot]
         |  |-aName [BuilderScmConnection]
         |  `-aName [BuilderScmModule]
```

```
        |
        `- aName [ContentSite]
           |
           `-aName [ContentDeployment]
```

The `-defaults` option specifies the `defaults.properties` file to use in conjunction with a object.template.xml file and provides a set of properties used during the generation process.

Example: defaults.properties

```
#
# Common name given to generated objects
common.name=${opts.name}
#
# ContentBuilder settings
#
scmConnection=http://svn/repos/example-content
scmModule=${opts.name}.war
packageInstallroot=/apps/${opts.name}
#
# deployment locations
#
common.node=${framework.node}
ContentBuilder.node=${framework.node}
ContentDeployment.node=${framework.node}
```

5.2.2.2. Design

Super Type
ProjectBuilder

Extends ProjectBuilder

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

5.2.2.3. Constraints

Allowed Child Dependencies

- BuilderBuildFile1
- BuilderBuildTarget1
- BuilderScmBinding1
- BuilderScmConnection1
- BuilderScmLabel1
- BuilderScmModule1
- BuilderStageExtension1
- BuilderStageFilebase1
- ProjectBuilderDefaults1
- ProjectBuilderDocBase1
- ProjectBuilderForrestHome1
- ProjectBuilderOrganizationDescription1
- ProjectBuilderOrganizationName1
- ProjectBuilderOrganizationURL1
- ProjectBuilderProjectDescription1
- ProjectBuilderProjectName1
- ProjectBuilderProjectURL1
- ProjectBuilderTemplateDir1

1: These types have a *Singleton* constraint. Only one instance may be added as a resource.

Allowed Parent Dependencies

- Node

5.2.2.4. Attributes

Exported Attributes

Name	Property
`basedir`	deployment-basedir
`targetdir`	deployment-install-root

Defaults for Imported Attributes

Name	Default
`buildfile`	${modules.dir}/ContentProjectBuilder/lib/build.xml
`buildtarget`	all
`defaults`	${modules.dir}/ContentProjectBuilder/templates/defaults.properties

organizationDescription	Maker of the ControlTier software
organizationName	ControlTier
organizationURL	http://www.controltier.com
projectDescription	The content deployment types
projectName	content
projectURL	http://open.controltier.com
scmConnection	https://moduleforge.svn.sourceforge.net/svnroot/moduleforge/trunk
scmModule	content
stageextension	jar
stagefilebase	.*
templateDir	${modules.dir}/ContentProjectBuilder/templates

5.2.2.5. 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.

5.2.2.6. Related Types

The following types are defined for use with ContentProjectBuilder.

ContentProjectBuilderSetting

Overview

ContentProjectBuilderSetting: *A ContentProjectBuilder setting.*

Design

Super Type
Setting

Role	**Abstract**. (Objects cannot be created.)
Instance Names	**Unique**

Constraints

Allowed Parent Dependencies

- Builder

ContentProjectBuilderDefaults

Overview

ContentProjectBuilderDefaults: *file containing project defaults properties*

Design

Super Type
ContentProjectBuilderSetting

Role	**Concrete**. (Objects can be created.)
Instance Names	**Unique**

Attributes

Exported Attributes

Name	Property
`defaults`	settingValue

ContentProjectBuilderTemplateDir

Overview

ContentProjectBuilderTemplateDir: *file containing project template files*

Design

Super Type
ContentProjectBuilderSetting

Role	**Concrete**. (Objects can be created.)
Instance Names	**Unique**

Attributes

Exported Attributes

Name	Property
`templateDir`	settingValue

5.2.3. ContentBuilder

5.2.3.1. Overview

ContentBuilder: *Builds a ContentPackage archive*

A ContentBuilder is responsible for building ContentPackage packages. Normally, users of this type will run two commands: scmChanges to check if the files in the local checkout tree are different from those on the repository, and Build to package those changes and store the resulting package archive in the repository.

5.2.3.2. Design

Super Type
Builder

Role	**Concrete**. (Objects can be created.)
Instance Names	**Unique**
Notification	**false**
Template Directory	`${modules.dir}/ContentBuilder/templates`
Data View	Children, proximity: **1**
Logger Name	ContentBuilder

5.2.3.3. Constraints

Allowed Child Dependencies

- BuilderBuildFile1
- BuilderBuildTarget1
- BuilderImportMax1
- BuilderImportMin1
- BuilderPackageExtension1
- BuilderPackageFilebase1
- BuilderPackageInstallroot1

- BuilderPackageType1
- BuilderPackageVersion1
- BuilderScmBinding1
- BuilderScmConnection1
- BuilderScmLabel1
- BuilderScmModule1
- ContentBuilderScmChangesOutput 1

1: These types have a *Singleton* constraint. Only one instance may be added as a resource.

Allowed Parent Dependencies

- ContentUpdater
- Node

5.2.3.4. Attributes

Exported Attributes

Name	Property
`basedir`	deployment-basedir
`targetdir`	deployment-install-root

Defaults for Imported Attributes

Name	Default	Description
`buildFile`	${modules.dir}/ContentBuilder/lib	The purpose of the ContentBuilder build.xml is to generate a new ContentPackage tgz archive.
`buildTarget`	package	target to call from runBuildscript command
`importMax`	1	maximum number of expected ContentPackage files
`importMin`	1	minimum number of expected ContentPackage files
`packageExtension`	tgz	
`packageFilebase`	.*?	

packageInstallroot		directory where package archive should be extracted
packageType	ContentPackage	Name of type to use to store and registrate new packages. Do not change unless subtyping ContentPackage.
packageVersion	${opts.buildstamp}	
scmBinding	svn	
scmChangesOutput	${entity.instance.dir}/var/scmCha	location of scmChanges output
scmConnection		connection string to SCM repository.
scmLabel		
scmModule		Name of module to checkout from SCM repository.

5.2.3.5. 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.

Build

Run the build cycle.

The purpose of the ContentBuilder build life cycle is to package any incremental changes between the SCM repository and the local workspace.

Usage

```
Build [-buildstamp <>]
```

Workflow

1. scmChanges
2. scmCheckout
3. generateManifest
4. generateTarIncludes
5. runBuildScript
6. repoImport

Success Handler

Email	
Subject	[${context.type}:${context.name} @ ${framework.node}] ${command.name} - SUCCESS
File	`${modules.dir}/Deployment/templates/notice.html`

Options

Option	Description
`buildstamp`	*build identifier*

generateManifest

Generate a ContentPackage manifest.

Creates a manfiest.xml file from the results of the scmChanges command

Usage

```
generateManifest [-buildstamp <>] [-manifest
<${entity.instance.dir}/var/manifest.xml>] [-module <>]
[-print] [-scmChangesOutput <>]
```

Options

Option	Description
`buildstamp`	*build identifier*
`manifest`	*output file*
`module`	*module name*
`print`	*print results to console*
`scmChangesOutput`	*file list*

generateTarIncludes

Generate a file for tarset includes.

The format of the tar includes file lists one file per line in plain text.

Usage

```
generateTarIncludes [-buildstamp <>] [-module <>] [-print]
[-scmChangesOutput <>] [-tarincludes
<${entity.instance.dir}/var/tarincludes.txt>]
```

Options

Option	Description
`buildstamp`	*build identifier*
`module`	*module name*
`print`	*print result to console*
`scmChangesOutput`	*file list*
`tarincludes`	*file to write tar includes*

runBuildScript

Build a new ContentPackage archive.

Usage

```
runBuildScript [-basedir <>] [-buildfile <>] -buildstamp <>
[-manifest <${entity.instance.dir}/var/manifest.xml>]
[-target <package>] [-targetdir <>] [-tarincludes
<${entity.instance.dir}/var/tarincludes.txt>]
```

Execution	bash
Arguments	${ant.home}/bin/ant -f ${opts.buildfile} ${opts.target} -Dopts.basedir=${opts.basedir}/${entity.attribute.scmModule} -Dopts.tarincludes=${opts.tarincludes} -Dopts.targetdir=${opts.targetdir} -Dopts.tarfilename=${context.name}-${opts.buildstamp}.tgz -Dopts.manifest=${opts.manifest};

Options

Option	Description
`basedir`	*directory where build resources reside*
`buildfile`	*build file to execute*
`buildstamp`	*build identifier*
`manifest`	*package manifest file*

target	*build target to evaluate*
targetdir	*directory build artifacts will be written*
tarincludes	*tar file includes*

scmChanges

Report the SCM status.

Usage

```
scmChanges [-basedir <>] [-binding <svn>] -buildstamp <>
[-connection <>] [-format <xml>] [-label <HEAD>] [-module
<>] [-quiet] [-scmChangesOutput <>]
```

Options

Option	Description
basedir	*base directory*
binding	*scm binding*
buildstamp	*build identifier*
connection	*connections string*
format	*output format. plain, markdown, xml, xmlproperty* • plain: lists one file per line w/o any adornment • markdown: Uses simple text formatting rules. See: Daring Fireball: Markdown for reference. • xml: Uses xml format intrinsic to SCM tool. • xmlproperty: Transformation of xml format to one suitable for loading by XmlProperty task.
label	*revision identifier*
module	*scm module*
quiet	*do not print output*
scmChangesOutput	*file list*

5.2.3.6. Related Types

The following types are defined for use with ContentBuilder.

ContentBuilderSetting

Overview

ContentBuilderSetting: *A ContentBuilder setting.*

Design

Super Type
Setting

Role	**Abstract**. (Objects cannot be created.)
Instance Names	**Unique**

Constraints

Allowed Parent Dependencies

- Builder

ContentBuilderScmChangesOutput

Overview

ContentBuilderScmChangesOutput: *The results of the scmChanges command is stored in this file.*

The results file should contain a list of changes between the local basedir and the SCM repository

Design

Super Type
ContentBuilderSetting

Role	**Concrete**. (Objects can be created.)
Instance Names	**Unique**

Attributes

Exported Attributes

Name	Property
`scmChangesOutput`	settingValue

5.2.4. ContentUpdater

5.2.4.1. Overview

ContentUpdater: *Establishes the end-to-end build and deployment of ContentPackage archives.*

The ContentUpdater is a simple subtype of the base type, Updater, that overrides its constraints to confine it to coordinate ContentBuider and ContentSite.

See ContentProjectBuilder for information on automating new object models.

5.2.4.2. Design

Super Type
Updater

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

5.2.4.3. Constraints

Allowed Child Dependencies

- ContentBuilder
- ContentSite
- DefaultAllowMultiplePackageMatches1
- DefaultFailIfPackageNotReplaced1
- DefaultPackageName1
- DefaultPackageType1
- DispatchBaseType1
- DispatchChangeDependencies1
- DispatchExecutionStrategy1
- DispatchOptions1
- DispatchResourceName1
- DispatchResourceType1

- DispatchSortOrder1
- DispatchThreadCount1

1: These types have a *Singleton* constraint. Only one instance may be added as a resource.

Allowed Parent Dependencies

- Node

5.2.4.4. Attributes

Defaults for Imported Attributes

Name	Default	Description
`defaultAllowMultiplePack`	false	Used by runChangeDependencies to specify if more than one matching Package can be assigned as a dependency
`defaultDeploymentType`	Deployment	Used by runChangeDependencies to match any Deployment sub-type
`defaultFailIfPackageNotR`	true	Used by runChangeDependencies to fail if no matching package could be found
`defaultPackageName`	^$	Used by runChangeDependencies to match any Package object name.
`defaultPackageType`	Package	Used by runChangeDependencies to match any Package sub-type
`dispatchBaseType`	(?:deployment|mediator)	Used by dispatchCmd to match any object that is a Deployment or Service
`dispatchChangeDependenci`	false	Specifies if Change-Dependencies should dispatch the command to the subordiante objects

dispatchExecutionStrateg	nodedispatch	Used by dispatchCmd option: -strategy
dispatchOptions	buildstamp	Used by dispatchCmd option: -dispatchOptions
dispatchResourceName	.*	Used by dispatchCmd to match any object name
dispatchResourceType	[^\.]*	Used by dispatchCmd to match any object type
dispatchSortOrder	ascending	Used by dispatchCmd option: -sortorder
threadCount	1	Defaults to sequential execution.

5.2.4.5. 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.

5.2.5. ContentSite

5.2.5.1. Overview

ContentSite: *A content deployment site*

The ContentSite type is a simple subtype of the base type, Site that participates in the ContentUpdater build and deployment workflow.

5.2.5.2. Design

Super Type
Site

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

Logger Name	ContentSite

5.2.5.3. Constraints

Allowed Child Dependencies

- ContentDeployment
- DispatchBaseType1
- DispatchChangeDependencies1
- DispatchExecutionStrategy1
- DispatchOptions1
- DispatchResourceName1
- DispatchResourceType1
- DispatchSortOrder1
- DispatchThreadCount1

1: These types have a *Singleton* constraint. Only one instance may be added as a resource.

Allowed Parent Dependencies

- ContentUpdater
- Node

5.2.5.4. Attributes

Exported Attributes

Name	Property
`basedir`	deployment-basedir
`install-root`	deployment-install-root

Defaults for Imported Attributes

Name	Default	Description
`defaultAllowMultiplePack`	false	Used by runChangeDependencies to specify if more than one matching Package can be assigned as a dependency
`defaultDeploymentType`	Deployment	Used by runChangeDependencies to

		match any Deployment sub-type
defaultFailIfPackageNotR	true	Used by runChangeDependencies to fail if no matching package could be found
defaultPackageName	^$	Used by runChangeDependencies to match any Package object name.
defaultPackageType	Package	Used by runChangeDependencies to match any Package sub-type
dispatchBaseType	(?:deployment|service)	Used by dispatchCmd to match any object that is a Deployment or Service
dispatchChangeDependenci	false	Specifies if Change-Dependencies should dispatch the command to the subordiante objects
dispatchExecutionStrateg	nodedispatch	Used by dispatchCmd option: -strategy
dispatchOptions	buildstamp	Used by dispatchCmd option: -dispatchOptions
dispatchResourceName	.*	Used by dispatchCmd to match any object name
dispatchResourceType	[^\.]*	Used by dispatchCmd to match any object type
dispatchSortOrder	ascending	Used by dispatchCmd option: -sortorder
threadCount	1	Defaults to sequential execution.

5.2.6. ContentDeployment

5.2.6.1. Overview

ContentDeployment: *manages content deployment*

The ContentDeployment type is a simple subtype of the base type, Deployment that participates in the ContentPackage build and deployment workflow.

5.2.6.2. Design

Super Type
Deployment

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

5.2.6.3. Constraints

Allowed Child Dependencies

- ContentPackage

Allowed Parent Dependencies

- ContentSite
- Node

Allowed Property Values

Property	Allowed Values	Default	Enforced
manages-deployments	• `false` • `true`		true

5.2.7. ContentPackage

5.2.7.1. Overview

ContentPackage: *A content package*

A ContentPackage contains a set of application content files. Typically built by ContentBuilder and consumed by ContentDeployment, ContentPackages contain incremental updates to a directory of content.

Being based on the tgz package type, a ContentPackage takes a directory of files and creates a gzip compressed tar file archive of those files. ContentPackage enhances the `tgz` type by including a package manifest file that declares the contents of the package. Additiionally, ContentPackage makes provisions to make backups of files that would be overridden when the archive is extracted to its installation root directory during deployment. These backups are stored in a new ContentPackage archive, tagged for rollback, which is uploaded and registered to the repository before the archive is extracted to its assigned installation root directory.

manifest.dtd

A valid ContentPackage is one that contains a manifest.xml in the root directory of the archive. Archives that do not contain a proper `manifest.xml` will be rejected by the ContentPackage extract command. The manifest.xml file is described by the following DTD:

```
<!ELEMENT manifest (package,builder,files*)>

<!ATTLIST package
installroot CDATA #REQUIRED
version CDATA #IMPLIED
type CDATA #IMPLIED
buildstamp CDATA #IMPLIED
>

<!ELEMENT builder (logentry*)>

<!ATTLIST builder
depot CDATA #IMPLIED
type CDATA #IMPLIED
object CDATA #IMPLIED
basedir CDATA #IMPLIED
scmConnection CDATA #IMPLIED
scmModule CDATA #IMPLIED
>

<!ATTLIST logentry
date CDATA #IMPLIED
author CDATA #IMPLIED
revision CDATA #IMPLIED
>

<!ELEMENT files (file*)>

<!ATTLIST file
```

```
        pathname CDATA #REQUIRED
        ftype (file|directory) #IMPLIED
        installpath CDATA #IMPLIED
        checksum CDATA #IMPLIED
        mtime CDATA #IMPLIED
        mode CDATA #IMPLIED
        owner CDATA #IMPLIED
        group CDATA #IMPLIED
        >
```

Example: manifest.xml

```
<manifest>
  <package installroot='/tmp/about.war'
           buildstamp='123'
           type='ContentPackage'
           version='${opts.buildstamp}' />
  <builder
      depot='test1'
      type='ContentBuilder'
      object='about'
      basedir='/app02/daily_content_builds/trunk/about/j2ee-apps'
      scmConnection='http://svn/repos/trunk/about/example'
      scmModule='about.war'>
    <logentry date='2007-10-13T20:36:52.339629Z' author='alexh'
revision='833' />
  </builder>
  <files>
    <file
        pathname='jsps/AssemblyView.jsp'
        installpath='/tmp/about.war/jsps/AssemblyView.jsp'
        ftype='file'
        checksum='b21453a03c9da9fd2865d9a49ce154e2' />
    <file
        pathname='jsps/DeploymentView.jsp'
        installpath='/tmp/about.war/jsps/DeploymentView.jsp'
        ftype='file'
        checksum='b21453a03c9da9fd2865d9a49ce154e2' />
  </files>
</manifest>
```

Rollbacks

During the extraction stage of package installation ContentPackage will read the `manifest.xml` and determine if any of the files contained in the archive would overwrite an existing local file. If so, a new ContentPackage is created and marked as having rollback files. These rollback packages can be used to restore the previous state, if the results of the new package are no longer desired.

Naming

ContentPackage rollback package objects are initialized using data from the incoming ContentPackage object. Their `name, filename` and `version` values however, are different and follow a convention. The table below describes the convention applied to rollback package objects.

Property	Naming convention
Name:	rollback-${package.base}-${package.version}.tgz
Filename:	rollback-${package.base}-${package.version}.tgz
Version:	rollback-${package.version}.tgz
Release tag:	rollback

If after a deployment occurs a rollback is desired use the ContentUpdater `runChangeDependencies` command:

```
          ad -p project -t ContentUpdater -o object -c
runChangeDependencies --\
             -buildstamp rollback-${buildstamp}
```

This will assign the ContentDeployment package dependencies to use the rollback packages for that version.

5.2.7.2. Design

Super Type
tgz

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

5.2.7.3. Constraints

Allowed Child Dependencies

- ContentPackage

Allowed Parent Dependencies

- ContentDeployment

Allowed Property Values

Property	Allowed Values	Default	Enforced
package-arch	• noarch	• noarch	
package-filetype	• tgz	• tgz	true
package-repo-url	• ${framework.pkg	• ${framework.pkg	false
package-vendor	•	•	false

5.2.7.4. 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.

extract

extracts the package

Extracts the content of the package archive file into the installroot directory.

Creates a rollback file containing any local files that would be overwritten by the content of the package.

The extract command utilizes the `create` command when making rollback archives. Additonally, it makes use of the `-upload -register` flags to load and register the new rollback packge.

Usage

```
extract [-filename <>] [-installroot <>]
```

Options

Option	Description
filename	*file to extract*
installroot	*destination directory*

create

creates the package archive

Reads data from context to register the package as a new object. See the manifest DTD section above.

Usage

```
create [-filename <>] [-installroot <>] [-register]
[-upload] [-version <>]
```

Options

Option	Description
`filename`	*file to extract*
`installroot`	*destination directory*
`register`	*optional flag specifying package file should be registered*
`upload`	*optional flag specifying package should be uploaded to the repository*
`version`	*package version*

6. All