Stellent™

Programmer's Reference Guide

SDK-001-500

Table of Contents

Chapter 1: The Development Kit

Overview 1-1
SDK Documentation 1-1
Creating Custom Conversion Engines 1-1
IdcCommand Reference Guide 1-2
Custom Scripting Reference Guide 1-2
Programmer's Reference Guide 1-2
Component Wizard 1-3

Chapter 2: Understanding Component Architecture

Overview 2-1
Examine or Modify Source Code 2-2
Create Customizations 2-2
Reinstall or Upgrade 2-2
Required Skills and Tools 2-3
Required Skills 2-3
Required Tools 2-4
Customizing the Interface 2-5
Customizing Product Functionality 2-6
Component Architecture and the Content Server 2-7
Server Behavior 2-7
Server Actions 2-8
Page Retrieval 2-8
Content Server Services 2-8
Search Services 2-9
Customizing Options 2-10
Customizing Graphics 2-10

Image Format 2-10
Image Referencing 2-11
Files Used for Customization 2-12
Bin Directory 2-12
Config Directory 2-13
Shared/Config Directory 2-13
Weblayout Directory 2-13
Development Recommendations 2-14
Development Instance 2-14
Component File Structure 2-14
Consistent File Structure 2-14
Naming Conventions 2-15
Use Unique File Names 2-16
Use Appropriate File Name Extensions 2-16
Use Consistent Naming Conventions 2-16
Observe Case 2-16
Change Form Methods 2-16
Read Server Errors 2-17

Chapter 3: Understanding Component Assembly

Overview 3-1
Page Assembly 3-1
Server Start Up Actions 3-3
Internal Initialization Occurs 3-3
Standard Resources, Templates, and Reports Load 3-4
Custom Components Load 3-4
Merge Rules 3-5
Component Architecture Process 3-5
Components File 3-5
Component Definition File 3-6
Modifying Resources 3-6
Modifying Standard Templates 3-7
Defining Custom Environment Resources 3-10
Defining Custom Queries 3-10
Defining Custom Services 3-13

Chapter 4: Understanding Resource Types

Overview 4-1
HTML Include 4-2
Dynamic Table 4-4
Query 4-5
Service 4-6

Template 4-7
Environment 4-8

Chapter 5: Understanding HDA and HTM File Types

Overview 5-1
HDA File Type 5-2
HDA File Structure 5-2
Section Types 5-2
Purpose 5-2
HDA Section Type: @Properties 5-3
HDA Section Type: @ResultSet 5-4
Data Binder 5-8
HTM File Type 5-9
Templates and Reports 5-9
Resources 5-9
HTM Tables 5-9
Structure 5-10
Dynamic Content Resources 5-11
Structure 5-11
Including Dynamic Content in a Template 5-15

Chapter 6: Understanding the Component Definition File

Overview 6-1
ResourceDefinition 6-2
ResourceDefinition Columns 6-3
type 6-3
filename 6-3
tables 6-4
loadOrder 6-4
Example ResourceDefinition 6-4
MergeRules 6-6
MergeRules Columns 6-6
fromTable 6-7
toTable 6-7
column 6-7
Example MergeRules 6-7

Chapter 7: Understanding the Components HDA File

Overview 7-1
Component Structure 7-1
Component Columns 7-2
name 7-2

location 7-2
Implementing a Component 7-2
Removing A Component 7-3
Configuration File 7-4
Defining a Variable 7-5
Referencing a Variable 7-6

Chapter 8: Understanding Templates

Overview 8-1
Content Server Loading 8-1
Templates File 8-2
IntradocTemplates 8-2
IntradocTemplates Columns 8-3
name 8-4
class 8-4
formtype 8-5
filename 8-5
description 8-5
VerityTemplates 8-6
SearchResultTemplates 8-6
SearchResultTemplates Columns 8-7
name 8-7
formtype 8-7
filename 8-8
outfilename 8-8
flexdata 8-8
description 8-9
Defining Custom Templates 8-10

Chapter 9: Understanding Content-Centered Template Metadata

Overview 9-1
Multi-Checkin Environment File 9-2
Multi-Checkin Menu Display 9-2
Multi-Checkin Content Types 9-2

Chapter 10: Understanding Query and Service Resources

Overview 10-1
Query Resource 10-1
Query Definition Tables 10-2
Query Definition Table Columns 10-2
name 10-2
queryStr 10-3

parameters 10-3
Database Tables 10-3
Example Query 10-7
Service Resource 10-8
Service Resource Structure 10-8
Service Name 10-9
Service Attributes 10-10
Service Class 10-10
Access Level 10-11
Template Page 10-11
Sub-Service 10-11
Subjects Notified 10-12
Error Message 10-12
Service Actions 10-13
Type of Action 10-13
Function Name 10-14
Function Parameters 10-14
Example Service 10-16

Chapter 11: Understanding the MultiCheckin Component

Overview 11-1
Component Description 11-4
MultiCheckinManifest.zip 11-4
manifest.hda 11-5
Example Manifest 11-6
components/doc_man.htm 11-7
components/multi_checkin_resource.htm 11-9
components/multi_checkin.hda 11-11
components/multi_checkin_environment.cfg 11-12
components/multi_checkin_templates.hda 11-14
readme.txt 11-14

Chapter 12: Understanding Workflows and Workflow Branching

Overview 12-1
Workflow Types 12-2
Basic Workflows 12-2
Criteria Workflows 12-2
Sub-Workflows 12-3
Workflow Steps 12-3
Jumps 12-4
Tokens 12-5
Workflow and Script Templates 12-6
Workflow Templates 12-6

Script Templates 12-6
Workflow Branching 12-7
Evaluating the Script 12-7
Actions Performed on the Last Step 12-8
Actions Performed on Restart 12-8
Actions Performed on Exit 12-8
Actions Performed on Error 12-8
Actions Performed on Reject 12-8
Executing the Script 12-9
Workflow Information Storage 12-10
Database Tables 12-10
Associated Files 12-10
Workflow Rules and Error Handling 12-11

Chapter

1

The Development Kit

Overview

The Development Kit for the Content Server consists of the SDK documentation and The Component Wizard.

SDK Documentation

The development kit documentation provides programmer level development information. This information is accessed as PDF files by selecting **Start—Programs—Stellent Content Server— Master_on_*server*—Utilities—SDK Documentation**.

Creating Custom Conversion Engines

This document provides information on creating custom conversion engines for the Refinery and Visual Basic module API specifications. This guide provides developers with the information they need to create and implement multiple custom conversion engines for the Refinery.

IdcCommand Reference Guide

This document provides information on the Java Command Utility and ActiveX Command Utility for the Content Server. The IdcCommand utility is a stand-alone Java application that enables users to execute services. The program reads a command file containing commands and parameters and calls the specified services. IdcCommandX is an ActiveX control that enables a program to execute a service and retrieve file path information.

Custom Scripting Reference Guide

This document provides information about Idoc Script application, functions, predefined variables and configuration settings; Web server variables; and HTML Forms scripting. The document contains syntax, code references, examples, and descriptions

Programmer's Reference Guide

This document provides a general description of how the system works and background information required for performing customizations. This guide supplies the pertinent information developers need to develop custom components for the Content Server. Information includes code references, technical tips, and examples.

Component Wizard

The Component Wizard is a development tool that automates the process of creating custom components. A developer can create custom components and modify existing components. Additionally, a developer can package any files associated with the custom component.

Launch the Component Wizard by selecting **Start—Programs—Stellent Content Server—Master_on_*server*—Utilities—Component Wizard**.

The Component Wizard can also be launched by navigating with Windows NT/2000 Explorer to the <*home*>\bin\ComponentWizard.exe file.

Follow these steps to launch from a command prompt:

1. Type `cd stellent\bin`
2. Press **Enter**.
3. Type `ComponentWizard`
4. Press **Enter**.

Chapter

2

Understanding Component Architecture

Overview

Components are program modules that are designed to interact with each other at runtime. Components can vary in size, can be authored by various programmers using different development environments and may or may not be platform independent. Components can be run on a single instance or across multiple instances such as a corporate intranet. Component architecture is derived from object-oriented technologies. Component software, such as the Content Server, implies the use of small modules that enables customization of the application.

There are several advantages to using Component Architecture with the Content Server.

- Examine or modify source code without compromising the integrity of the product.
- Create Customizations with copies of original code modules.
- Reinstall or upgrade without compromising customizations.

Examine or Modify Source Code

The Content Server loads many of its resources from external text files. Thus, it provides the ability to view the files to analyze how the system works.

Create Customizations

The Content Server was designed to provide the ability to make changes to copies of these resources and override the look and feel of the system. The primary file for implementing customizations is the <*home*>/config/components.hda file.

When the server loads, the final step is to load any defined components. The components.hda file provides the Content Server the required information on which component to load.

Note: Items with identical names override one another, with the last item loaded having its definition take precedence over all others.

Reinstall or Upgrade

Files such as std_page.htm can be copied and the definitions rewritten for some or all of the resources defined within the product. This leaves the original files intact. Once rewritten, the customized files simply need to be included with the use of the components.hda file.

Required Skills and Tools

To take advantage of the extensibility of the Content Server, it is important to understand the skills and tools needed for performing customizations using Component Architecture.

Required Skills

The Content Server brings together a wide variety of technologies to deliver advanced functionality. To modify the system, certain experience and skills with some or all of these technologies is required. The technical skills required will vary depending on the complexity of the customization. Many customizations can be accomplished with a knowledge of HTML, Component Architecture, and Idoc Script.

This list describes, in descending order of importance, the technologies you may need experience and skill when modifying the content Server.

- **HTML/CSS—**To make changes to the templates a good understanding of HTML and cascading style sheets (CSS) is required. The templates are not complex in their use of HTML, but they make constant use of HTML tables and frequent use of forms. The std_page.htm file includes cascading style sheets to control the look-and-feel of the default templates, including fonts and layout. Therefore, knowledge of these aspects of HTML is essential to creating customizations.
- **Component Architecture—**To understand how your changes will be implemented, a conceptual understanding of how the Content Server works is required.
- **Idoc Script—**Almost every page that is statically or dynamically assembled includes some Idoc Script. Idoc Script is a proprietary scripting language. It provides the method for processing various page elements after the browser has made a request, but before the requested page is returned.

  For additional information, refer to the *Custom Scripting Reference Guide*. This reference manual includes a categorically arranged, alphabetical listing of pre-defined variables, Idoc Script commands, and functions. This reference tool includes a description of each of the commands, as well as proper syntax and examples.

- **JavaScript—**Most Content Server pages do not use JavaScript. Notable exceptions are the Search and Check in pages. For changes to these pages you should have an understanding of JavaScript. In addition, it is important to understand how JavaScript works with HTML forms.
- **SQL—**Structured Query Language is used in the system to manage information related to the content items. The queries you build with SQL can relate relevant information about each content item on a web page.
- **Java Programming—**The server is implemented with Java classes. A thorough understanding of Java and the Content Server Java class files is required before any changes can be made to that part of the system. However, the product can be customized extensively without having to work with Java.

Required Tools

These are some of the tools that you may find useful in performing modifications:

- **Text Editor—**Most product customizing can be done with a normal text editor such as Microsoft WordPad.
- **HTML Editor** (non-graphical mode)**—**Use caution when using an HTML editor. Often, such programs change the source HTML. If you use a graphical editor such as the one provided with Microsoft Visual InterDev, make sure you edit in a non-graphical mode.

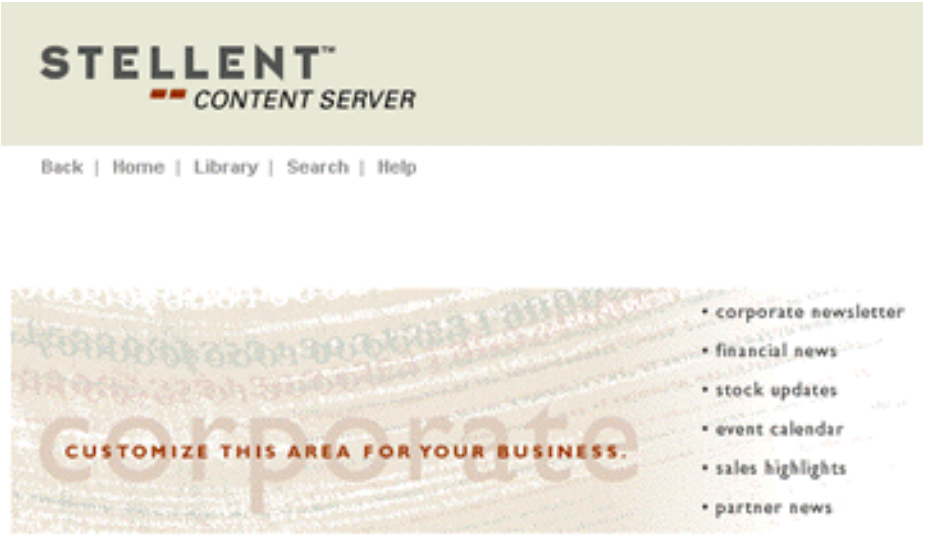
Important: Using an HTML editor in its graphical mode may cause Idoc Script tags to be converted into a string of characters that will no longer be recognized by the Content Server.

- **JavaScript Debugger—**A JavaScript debugger will ease the task of JavaScript development. Java development will require that you have an appropriate Java development environment.
- **Multiple Browsers—**We recommend that changes be tested in all versions of browsers that your clients will use. Internet Explorer and Netscape Navigator do not display content in the same manner. In addition, different versions of the same browser may exhibit different behavior.
- **Software Development Kit (SDK)—**The Software Development Kit is a collection of documentation and the Component Wizard. The documentation provides the conceptual knowledge for customization and reference information. The Component Wizard is designed to assist you in developing your custom components.

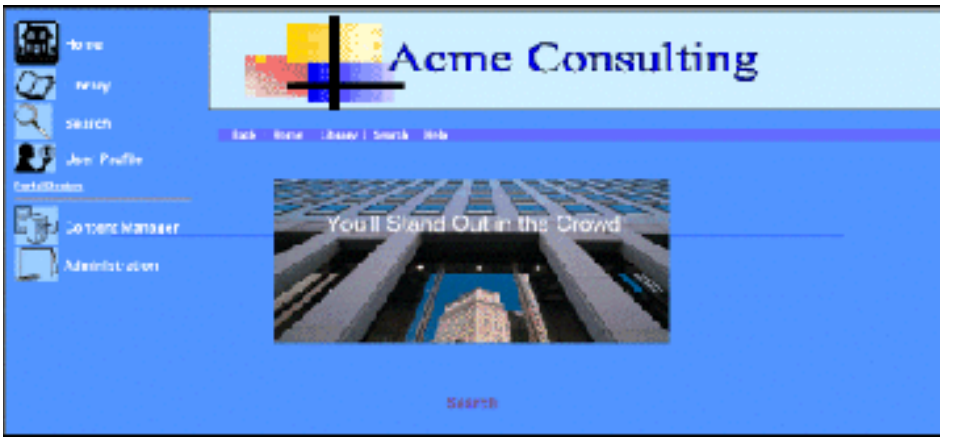
Customizing the Interface

By creating custom components, the interface can be customized to meet your business specifications. Some modifications can be as simple as replacing the graphic images that appear on displayed pages. In this example, a sample, customized interface is provided.

Before Customization:



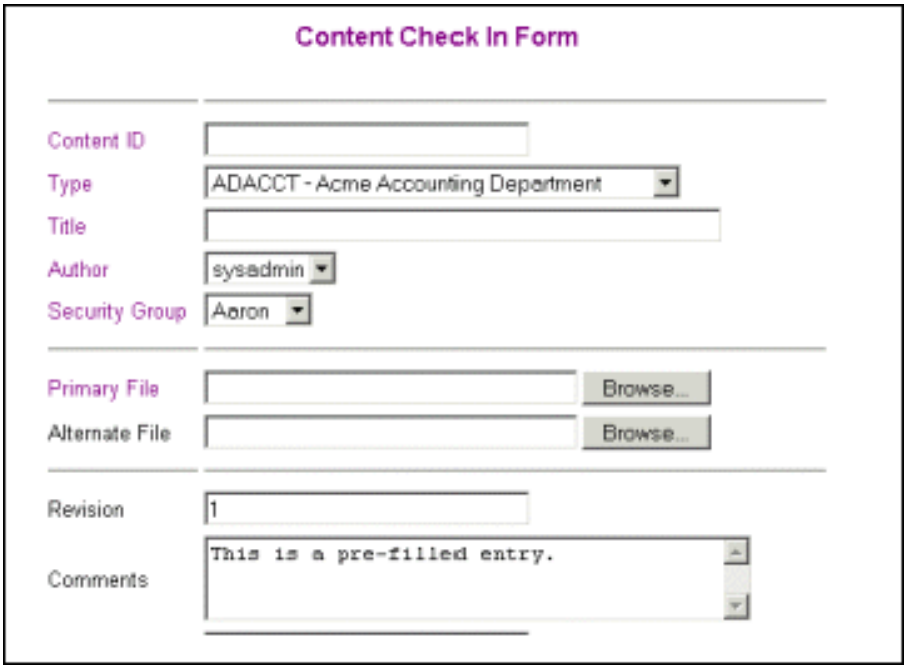
After Customization:



Customizing Product Functionality

Customizations can be performed to the Content Server to change the functionality of the software. For example, a custom component can be created that changes how Info Fields (metadata) are presented. For example, the *Comments* field on the Content Check In Form can be pre-filled with information, or the *Expiration Date* can be specified as a required field.

Sample Content Check In Form:



Component Architecture and the Content Server

Server Behavior

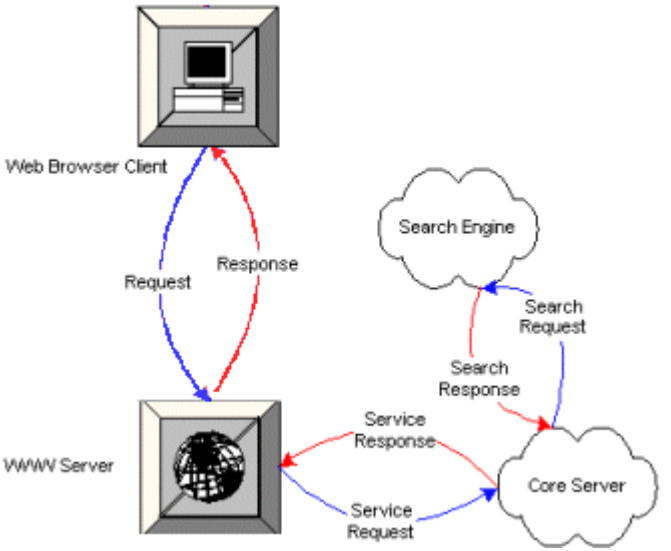
The Content Server enables you to use a web browser to check in and retrieve content items, control access to content items, and search for content items. This section gives a high-level description of how the Content Server works using the Verity Search API and Content Server search operations.

The web browser sends a request to the web server and the web server responds to the request. Three types of requests to the web server can be made:

- Retrieve pages
- Run a system server service
- Run a search engine service

When a search request is made, the web server routes the request through the Content Server. The system server then communicates with the search engine to perform the search.

Server Information Flow:



Server Actions

There are three main types of server actions that can be performed with the software:

- Page retrieval
- Content Server services
- Searching services

Page Retrieval

When a request for a page is made, one of two available page types is delivered. The two types of pages are: static and dynamic.

- **Static Page Retrieval**—Only one of the pages in the Content Server web site is considered *static*. This means that the content of the page is pre-formatted at the time of the request. The home page, *weblayout/portal.htm*, is a static page. When a browser request is made for this page the request is handled by the standard functionality of the web server.
- **Dynamic Page Retrieval**—A protected page is also referred to as *dynamic*. When a browser request is made for a protected page, a dynamic page such as the standard query page or a search request, the web server relies on the Content Server or the search engine to fulfill the request.

Content Server Services

When a request is made for a protected page, such as the Administration link on the Personal Navigation area, the browser is placing a request for a Content Server service. In the case of the Administration link, it is requesting the GET_ADMIN_PAGE service. The URL of the Administration link contains the following commands:

```
Idc_cgi_isapi.dll?IdcService=GET_ADMIN_PAGE&Action=GetTemplatePage&Page=ADMIN_LINKS
```

The web server recognizes this request as a Content Server function and sends the specific request to the Content Server. When the Content Server processes the request, it passes the result of the request back to the web server. The web server then delivers the results of the Content Server service to your web browser.

In the case of the Administration link, the service:

- Provides a login prompt if not currently logged in

- Verifies that the login has administrator privileges
- Assembles the ADMIN_LINKS template page and returns the page

When a file is checked in/out or a report requested from the database, the Content Server performs the listed tasks. When a search request is submitted, the browser first sends a request for the STANDARD_QUERY_PAGE template. The Content Server also handles this request. The result of the request is that the web server delivers a search form to the web browser.

Search Services

When a search form is completed and a request made, the browser sends a request to the web server to perform a search. When using the search engine, the URL for the request contains the following syntax:

```
/intradoc-cgi/idc_cgi_isapi.dll? idcService=GET_SEARCH_RESULTS
```

The web server recognizes this request as a Content Server function and sends the specific request to the Content Server for processing. In doing so, the Content Server sends a request to the search engine using a search engine API.

The search engine sends the search results back to the Content Server, which sends the results to the web server. The web server then delivers the result of the search service to the web browser.

The Content Server has been placed between the web server and the search engine to enable the search and search results template to be processed based on information supplied by the requestor. If additional security is needed for your site, the Content Server can perform those functions. For example, limiting search result fields based on the role of the user requesting the search.

Customizing Options

Customizing Graphics

The easiest way to change the look of the Content Server web site is to change just the graphic images that are referenced on the corresponding template page. Component Architecture is not required for making these types of changes, however we do not recommend this method.

If you choose to change the image references without using Component Architecture, you should be aware that it may have the following limitations:

- **Awkward Geometry**—The image may appear skewed or misshapen unless image dimensions are identical to the original image replaced.
- **No Addition/Deletion**—You will only be able to replace images. You will not be able to add or delete existing images. Additionally, after replacing the images, you are still left with the same layout and functionality.
- **Lost Data**—Changes will be lost if the product has to be reinstalled or upgraded, since the files in the <*home*>/weblayout/images/ directory will be overwritten.

Image Format

The graphic images used by the software is located in the <*home*>/weblayout/images/ directory. These images are in a GIF format that can be opened, viewed, and edited in most any image editor. For best results, you should keep the image geometry (height and width) of the replacement image the same as that of the original image. If the height or width is changed, the web browser will scale the images and the images may be distorted.

Image Referencing

All images are defined in the following file: <*home*>/shared/config/resources/std_page.htm. To implement your images, either of these methods can be used:

Method 1

1. Give your new image the same name as the original image it will replace.
2. Copy the existing file (<*home*>/weblayout/images) to another location and rename it.
3. Copy your new image files to the <*home*>/weblayout/images/ directory.

Method 2

1. Locate the image reference in the std_page.htm file.
2. Change the path name to accommodate the location of your new images.
3. Copy your new images to the folder located beneath the <*home*>/weblayout/images/ directory.

Files Used for Customization

Most customizations made with Component Architecture are done with the files that are found primarily in four directories:

- *home*>/bin/
- <*home*>/shared/config/
- <*home*>/config/
- <*home*>/weblayout

Bin Directory

To use the command line features of the Content Server, access the executable files located in the <*home*>/bin/ directory.

Note: If you have the Content Server set up as an automatic service and attempt to start the Content Server using this method (IdcServer or IdcServerNT) at the command prompt, you will receive an error message that states: *The port could not be listened to and is already is use*.

This is the default structure of the <*home*>/bin/ directory:

Element	Description
bin	The bin directory stores a number of executable files including: • BatchLoader • Component Wizard • SystemProperties • IdcServer • IdcServerNT

Config Directory

The <*home*>/config/ directory acts as a location for storing global information. The two main files in the <*home*>/config/ directory that are utilized when performing customizations with the component architecture process are described in the following table.

Element	Description
components.hda	The file that describes custom components that have been added to the system.
config.cfg	The file that defines system configuration variables.

Shared/Config Directory

The <*home*>/shared/config/ directory contains files with HDA and HTM formats. This is the file structure with one Content Server instance installed. The top-level in the directory used when customizing the product is described in the following table.

Element	Description
reports	Holds templates for the Content Server reports.
resources	Holds resource definitions (queries, page resources, and services) for the Content Server, including the std_page.htm file.
templates	Holds templates for all Content Server pages.

Weblayout Directory

The weblayout directory contains images that are displayed on the various pages of the content Server web site. The structure for the <*home*>/weblayout/ directory is described in the following table.

Element	Description
weblayout	The file that stores any images or web viewable content items that are checked into the system.

Development Recommendations

This section contains some guidelines to assist you in developing custom components. This information includes recommendations about development instances, custom component file structures, naming conventions, case observance, form methods, and server errors.

Development Instance

Whenever you are performing development, you should isolate your development efforts from your production system. Remember to include the same custom information fields you will be using in both your production and development instances. Be sure to check in a few sample content items in your development instance.

Once you have successfully tested your modifications on the development instance, it is a simple matter of copying the required files to your production system, installing the components using the <*home*>*/config*/components.hda file and restarting your server.

If you are having problems with your server and you have installed custom components, you may need to disable (uninstall) the custom components and restart your server.

Component File Structure

Your custom components should be placed into their own directory. By default, the Component Wizard places all custom components into a folder named *custom*, which is located directly beneath the root Content Server installation. Custom components do not have to be stored on the same machine as the home installation, but must be accessible by the Content Server.

Images and other objects that must be referenced by HTML pages must reside somewhere in the <*home*>/weblayout/ directory (which is accessible by the web server).

Consistent File Structure

To keep your custom components organized, we recommend keeping a consistent file structure that emulates the Content Server <*home*>/shared/config/ directory. To accomplish this, create three sub-directories in the component directory:

- resources/ for holding resource files
- templates/ for holding template files
- reports/ for holding report files

Place the component resource definition HDA file, at the top-level of your component directory. When referencing files within these directories, use relative path names. This makes it easier for you to move your component to a different location without having to edit all of the files in the component.

For example, use templates/templates.hda to reference a templates.hda file in the my_component/templates/ directory, instead of c:/my_component/templates/templates.hda. This example shows that type of reference:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
template
templates/test_template.hda
null
1
@end
```

Note: The Content Server is a Java-based application. Forward slashes must be used in the pathnames.

Naming Conventions

In the event that you have multiple components installed, and the components share a common file name (for example, `my_resource.htm`) the definition for the component that is loaded last will take precedence.

There are certain naming conventions that are recommended for developing custom components. These recommendations extend to the directories, individual files and the contents of those files.

Use Unique File Names

It is recommended that you give all of your component directories and files unique and meaningful names. A common convention used with creating file names is to place the prefix *custom_* in front of the original file name. It is also a preventative step for avoiding conflict among multiple components.

Use Appropriate File Name Extensions

HTM files should have an HTM extension and HDA files should have an HDA extension. If you are creating the file with a text editor like WordPad, place the file name within quotation marks so the proper file extension will be assigned to it (for example,`"myfile.hda"`). Failure to use quotation marks to define the file may result in a file name such as `myfile.hda.txt`.

Use Consistent Naming Conventions

Be consistent with your naming conventions. For example, if you are modifying the standard query template (std_query.htm), it is recommended that you use a naming convention like custom_query.htm for your modifications. This practice is a two-fold solution: you do not overwrite any default templates and your customizations are easy to categorize and identify.

Observe Case

The Content Server is case sensitive even if your file system is not. For example, when a template name is defined as `My_Template`, the Content Server will not recognize case variations such as `my_template` or `MY_TEMPLATE`.

Change Form Methods

HTML forms have a method that is used to communicate the form data to the web server. Change the METHOD attribute of any FORM from a POST to a GET. This will enable you to see all of the parameters as they are passed from a web browser to the web server, filtered through the Content Server and then back to the web browser. To change the form method, you must make an entry in your form's HTML code with the `METHOD="GET"` command.

Read Server Errors

When developing components, there are a number of problems that can arise. For example, you may have made a mistake somewhere in your files or the Content Server detects something wrong with one or more of your files, such as an extra carriage return or character: this can cause the server to fail to load a file. If the server fails, it will report the error via the command prompt window (on Windows NT) or to a log file (on UNIX).

Getting to that information is important to helping you resolve the problem. How you get to the information depends on the operating system on which the server is running.

Using the Content Admin Server page, the server log file can be viewed by selecting the content server, and then clicking the **View Server Output** link.

Chapter

3

Understanding Component Assembly

Overview

Component Architecture enables customizations to be made to the product without modifying the original source files. To understand what happens when a custom component is loaded, we must take a high-level view of the Content Server's behavior and then determine the additional processes.

Page Assembly

When a request is received from a web browser client for a dynamic page, the server performs a specific set of actions to deliver that page. These actions assemble template pages into the final displayed page. Each page provides specific markup for the final displayed page and has a specific place in the final page.

Resource types can be any of the following: HTML markup, queries to gather information from the database, and special code to conditionally format the information. Each assembled page has three standard conventions and occasionally some dynamically generated data. As a rule, each page consists of three resources:

- A standard page header.
- A standard page beginning.
- A standard page ending.

All of these definitions are cached in memory. When the server gets a request for a page, it already has a definition for the pieces that appear on the page. The server combines many elements together into a template that is ready to be processed for a specific data request by the client. After the Content Server has been started and loaded all of the resource information into the memory, it waits for requests from clients.

Since this is the standard software behavior whenever you define new resources, templates, or reports, you must restart the server. If you have made a change, but the change does not appear to have taken affect, restart the server.

Server Start Up Actions

All the template pages in the Content Server are pre-parsed and cached. When the Content Server starts, it reads the main templates table file templates.hda. This table describes each template and points to the corresponding HTML template file. The HTML template file is read and some of the HTML server side scripts are resolved immediately. The resulting template page is then stored in memory to speed up page presentation.

The following general steps occur when the server starts:

- Internal initialization occurs.
- Configuration variables load.
- Standard *resources*, *templates*, and *reports* load.
- Custom components load.

Internal Initialization Occurs

When the server initializes internally, the Java class files from the Content Server are read and the Java Virtual machine is evoked.

`Configuration Variables Load`

After initializing, the Content Server locates the file name `<home>/config.cfg`. The config.cfg file stores the system properties and default configuration variables. The configuration file consists of a number of name/value pairs.

The value assigned to each variable can be displayed in any specified template, by using Idoc Script substitution. For example, if you want to display the variable *Master_on_secondserver*, you could place the Idoc Script command <$InstanceDescription$> within a template file.

The information contained within the configuration file was supplied during the Content Server installation process.

Standard Resources, Templates, and Reports Load

There are number of resources, templates, and reports that need to be loaded for the Content Server to function properly. A number of these files are located in the following directories:

- <*home*>/shared/config/templates/
- <*home*>/shared/config/resources/
- <*home*>/shared/config/reports/

For the server to know which files to load, it reads the entries made in a file located at: <*home*>/shared/config/templates/templates.hda. The templates.hda file notifies the Content Server to load specific default templates.All of these template files are stored in the directory <*home*>/shared/config/templates/ and are the pages that make up the Content Server web site.

Custom Components Load

The Content Server loads any custom components last. The Content Server locates the file named <*home*>/config/components.hda. The Content Server then searches for references to any components that might be enabled. This is an example of the components.hda file:

```
@ResultSet Components
2
name
location
My Component
C:/stellent/custom/my_component/my_component.hda
@end
```

In this example, the information contained within the components.hda file directs the Content Server to the component definition file named *my_component.hda*. The component definition file contains location references to any new resources that have been defined.

Merge Rules

When developing custom components, the custom template files are referenced by creating a component definition file named MergeRules. The MergeRules table forces the Content Server to perform a comparison check on the name of your table by the template page column table.

- If the name of your custom template page column matches the name of the default template page column, your custom template will overwrite the existing default template.
- If your custom template page name does not match any of the default template page column names, your file will be appended to the templates available in the <*home*>/shared/config/templates/ templates.hda file.

Component Architecture Process

Component architecture involves a variety of processes and include these steps:

1. Making copies of some of the standard templates.
2. Modifying those templates to meet your specifications.
3. Creating a ResourceDefinition table in the component definition HDA file (this may or may not contain MergeRules).
4. Making a reference in the components.hda file to the name and location of your component.

Components File

The components.hda file is located in the directory <*home*>/config/ and serves as the ultimate location where your custom component's name and location are referenced. The components.hda file contains a result set name *Components*. This is an example of the file structure:

```
@ResultSet Components
2
name
location
@end
```

Once you have defined a component, you will reference the component by making an entry into the Components ResultSet that contains information about the name and location of your custom component. An absolute path can be used when specifying the location of your component or a relative path relative to the Content Server home directory.

Component Definition File

The component definition HDA file is the portion of your component that points to any custom resources that you have defined and, if applicable, defines any accompanying MergeRules. This is an example of the general structure for the component definition HDA file:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
@end
```

Once you have modified copies of the standard templates reference these changes in the ResourceDefinition ResultSet.

Modifying Resources

After making changes to graphic images in your copy of the file <*home*>/ shared/config /std_page.htm, you must make an entry in the ResourceDefinition table.

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
resource
resources/my_std_page.htm
null
1
@end
```

After making an entry into the components.hda file, the file should be saved and the server stopped and restarted to implement the changes.

Modifying Standard Templates

Follow these steps to modify the standard templates:

1. Make a copy of the templates you intend to modify and a copy of the file templates.hda and place them into the component *templates/* directory.
2. Within the templates.hda file, rename the ResultSet IntradocTemplates to something descriptive, such as MyTemplates.
3. Delete all entries for the template names that you are not modifying, along with the ResultSets VerityTemplates and SearchResultsTemplates.
4. Update the reference to the template name that is implemented by default to the name of your custom template.

Note: Spaces can not be used in the table name.

```
@ResultSet MyTemplates
5
name
class
formtype
filename
description
HOME_PAGE
RootPage
HomePage
my_std_home_page.htm
Custom Home page for weblayout
CHECKIN_NEW_FORM
Document
CheckinForm
my_checkin_new.htm
```

```
Custom New Document Check in Form
CHECKIN_SEL_FORM
Document
CheckinForm
my_checkin_sel.htm
Custom Document Check in Form
DOC_INFO
Document
DocumentInfoForm
my_doc_info.htm
Custom Document Information Form
STANDARD_QUERY_PAGE
Search
QueryPage
my_std_query.htm
Custom Document Search Form
UPDATE_DOC_INFO
Document
UpdateDocInfoForm
my_update_docinfo.htm
Custom Document Update Doc Info Form
@end
```

In the ResultSet ResourceDefinition, make a reference to the templates.hda file that you modified and then create a ResultSet MergeRules. In this example, the templates.hda file has been renamed to mytemplates.hda and stored the file to the path *c:/MyComponent templates/*. Also, the ResultSet in the mytemplates.hda file has been renamed to MyTemplates.

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
template
templates/mytemplates.hda
MyTemplates
1
@end
@ResultSet MergeRules
3
fromTable
toTable
column
MyTemplates
IntradocTemplates
name
@end
```

It is not necessary to separately define the new resources that have been defined. By making reference to the mytemplates.hda file, the system has already been instructed which templates (HTM files) to merge into the ResultSet IntradocTemplates. The references made to templates such as my_std_home_page.htm will be automatically detected when the server starts up and the merge is performed.

After making an entry into the components.hda file, the file should be saved and the server stopped and restarted to implement the changes.

Defining Custom Environment Resources

Create a text file at the top level of your component that has a file extension *.cfg*. This file should be defined in the ResourceDefinition table and implemented by making a reference to the component that contains the environment resource in the components.hda file.

In this example, assume that we have opened the file my_environment.cfg and defined an environment variable.

```
Customer=wise@intranetsolutions.com
ThemeColor=rose
```

To reference your environmental variables in copies of the templates to be modified, you will use an Idoc Script tag, such as <$Customer$> or <$ThemeColor$>. A reference in the components.hda file must be made for your changes to be implemented. This is an example of an entry in the ResourceDefinition ResultSet:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
environment
resources/my_environment.cfg
null
1
@end
```

Defining Custom Queries

To create a custom query, the process is much the same as creating a custom template. However, you will have to make a copy of the query.htm resource file, place it into the component resources file and modify the table entry to suit your purposes. The structure of the Query Table is that it has three columns with the following names: labels, queryStr, and parameters.

The HTM format of the file looks similar to the following code:

```
<HTML>
<HEAD>
<META HTTP-EQUIV='Content-Type' content='text/html;
charset=iso-8859-1'>
<TITLE>Custom Query Definition Resources</TITLE>
</HEAD>
<BODY>
<@table MyQueries@>
<table border=1><caption><strong>Custom Query Definition
Table</strong></caption>
<tr>
 <td>name</td><td>queryStr</td><td>parameters</td>
</tr>
<tr>
 <td>Ireport</td>
 <td>insert into Reports (dReportName, dProject,
dDescription) values (?, ?, ?)</td>
 <td>dReportName varchar
    dProject varchar
    dDescription varchar
 </td>
</tr>
<tr>
 <td>Qreports</td>
 <td>select * from Reports</td>
 <td>
 </td>
</tr>
</table>
<@end@>
</BODY>
</HTML>
```

In this instance, you would need to make an entry in the Component definition HDA file and set a MergeRule. Once completed, the Component definition HDA file will look similar to the following:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
query
resources/MyQueries.htm
MyQueries
1
@end
@ResultSet MergeRules
3
fromTable
toTable
column
MyQueries
QueryTable
name
@end
```

After making an entry into the components.hda file to reference your file, the server should be stopped and restarted to implement the changes.

Defining Custom Services

The process of defining custom services is nearly identical to the process of creating a custom query. The main difference lies in the information that you must supply within the".htm" file itself. Make a copy of the file *<home>*/ shared/config/ resources/std_services.htm and place it into the component *resources/* directory. Make entries into the table definition columns: Name, Attributes, and Actions.

This is an example of the script used to define a custom service named MyServices:

```
<HTML>

<HEAD>

<META HTTP-EQUIV='Content-Type' content='text/html;
charset=iso-8859-1'>

<TITLE>Custom Scripted Services</TITLE>

</HEAD>

<BODY>


<@table MyServices@>

<table border=1><caption><strong>Scripts For Custom
Intra.<i>doc!</i> Services

 </strong></caption>

<tr>

 <td>Name</td><td>Attributes</td><td>Actions</td>

</tr>

<tr>

 <td>ADD_REPORT</td>

 <td>Service

   18

   ADD_REPORT_FORM

   null

   null<br>

   Unable to add report.</td>
```

```
 <td>2:Ireport::0:null</td>
</tr>
<tr>
 <td>REPORTS_LIST</td>
 <td>Service
   17
   REPORT_LIST_FORM
   null
   null<br>
   Unable to retrieve reports.</td>
 <td>5:Qreports:REPORT_LIST:0:null</td>
</tr>
</table>
<@end@>
<br><br>
</BODY>
</HTML>
```

The method of having the custom service recognized is by creating a reference to your custom file in the ResourceDefinition ResultSet and by creating a MergeRule that merges MyServices with the Services table. This is an example of the associated Component definition HDA file:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
service
resources/MyServices.htm
MyServices
1
@end
@ResultSet MergeRules
3
fromTable
toTable
column
MyServices
Services
name
@end
```

Chapter

4

Understanding Resource Types

Overview

Resources play many roles within the Content Server environment. Resources can be snippets of HTML code, dynamic page elements, HDA files within HTM table, queries that gather information data from the database, or special code to conditionally format specific information. Since resources are a critical part of the software, it is essential to be familiar with them. Each resource type has its own purpose, structure, and application.

Resources fall into seven distinct categories:

- HTML Include
- Static Table (HTML format)
- Dynamic Table (HDA format)
- Query
- Service
- Template
- Environment

HTML Include

This is a resource type and part of an HTM file that is used to defined the pieces of HTML markup that normally appear in more than one template or report file. The standard HTML includes are defined in the <*home*>/shared/config/resources/std_page.htm file

An example of one such resource is <*@dynamichtml std_page_begin*>. This particular convention is used during the page assembly process for dynamic pages. This resource is defined in the <*home*>/shared/config/resources/std_page.htm file and defines the layout for how any standard page will begin. This is a script sample from the std_page.htm file:

```
<@dynamichtml std_page_begin@>

<$if not coreContentOnly$>

<table border=0 cellpadding=0 cellspacing=0 width="100%"
height="100%">

<tr>

<!-- sidebar for nav links -->

<td width=<$pne_nav_width$> valign=top><$include
pne_nav_links$></td>

<!--Overall page table with logo and head banner -->

<$if widePage$>
<$StdPageWidth=550$><$else$><$StdPageWidth=500$><$endif$>

<td valign="top"><table border=0 cellspacing=0 cellpadding=0
width="100%">

<tr>

<!-- top banner -->

<td colspan=3 valign="top" align="left" height=1
width="100%" bgcolor="<$banner_top_color$>">

<img src= "<$HttpImagesRoot$> <$banner_top_image$>"

align="top" border="0"alt="Top banner logo."></td>

</tr>

…

<@end@>
```

Any dynamic Include is referenced in an appropriate template file by using Idoc Script:

```
<$include std_page_begin$>
```

Dynamic Table

The dynamic table is a resource type with the HDA file format. These resource types are used to define tables that will be used to communicate with the Content Server during the page assembly process. This is an example of a Dynamic Table resource:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
service
resources/preview_service.htm
preview_Services
1
query
resources/preview_query.htm
preview_Queries
1
template
templates/preview_template.hda
null
1
resource
resources/preview_resource.htm
null
1
@end
```

Query

The query resource is a table that defines the location of an HTM file containing the definition of database queries. The Content Server reads the default queries defined in the system from the file <*home*>/config/shared/resources/query.htm. Services that generate pages use queries to get data to merge into the template pages.

A query .htm file can be opened in a Web browser or in a text editor. The information is presented in tabular form when opened in a browser, and as script with html tags and Idoc Script when opened in a text editor.

Important: Using an HTML editor in its graphical mode may cause Idoc Script tags to be converted into a string of characters that will no longer be recognized by the Content Server.

Service

The service type resource defines the location of an HTM file containing the definitions of service scripts. The standard service file is located at <*home*>/shared/config/resources/std_services.htm.

An IdcService is a defined function or procedure that can be performed within the Content Server. Since a service is a mechanism for interacting with the Content Server and consequently the database, any program or HTML page that requests information from the server or performs a function must use these services.

The service type resource is discrete and may require parameters. Services are also the only way a client can talk to the server or access the database. Services are, in fact, the only way user-initiated functionality is implemented. This is because the service is a call that could happen from either the client or server side.

By having both server and client execute the same service, we ensure integrity in the system. Everyone eventually does the same thing, even if they start from completely different places. So if a browser requests all the users in the system, it will perform the same service as the applet that requests all users. Services are of primary importance when creating custom components to change server behavior.

A .htm file can be opened in a Web browser or in a text editor. The information is presented in tabular form when opened in a browser, and as script with html tags and Idoc Script when opened in a text editor.

Important: Using an HTML editor in its graphical mode may cause Idoc Script tags to be converted into a string of characters that will no longer be recognized by the Content Server.

Template

A template type defines the location of an HDA file. This HDA file contains a table describing the names, types, and locations of template files that should be loaded as part of this component.

An example of such a file can be found at the following location: <*home*>/ shared/config/ templates/templates.hda. This file holds the default templates loaded by the system.

The templates.hda file defines three tables:

- The IntradocTemplates table contains the full list of cached template files.
- The VerityTemplates table contains the results templates used by the Verity search script engine.
- The SearchResultTemplates table contains the search results pages implemented by the Content Server.

Environment

The environment resource type defines the location of a file with a .cfg extension that enables a component to define its own configuration. An environment resource file contains name/value pairs (using the same format as the config.cfg file) and is loaded after the config.cfg file is loaded.

This is an example of the entries found in an environment resource file located in the directory <*home*>/admin/config.cfg. The information contained in this file will be different with each installation

```
HttpRelativeWebRoot=/stellent/
CgiFileName=idc_cgi_isapi.dll
HttpServerAddress=techpubs
IDC+Admin_Name+techpubs
#Internet Variables
```

Chapter

5

Understanding HDA and HTM File Types

Overview

The HDA and HTM file types are used extensively when performing custom component development with the Content Server. Both the HDA and HTM files types present tabular information. HDA files present tabular data in a simple structured ASCII file format. The HDA file format is very useful for dynamic data. The compact size and simple format of HDA files make data communication faster and easier for the Content Server.

HTM tables are useful for storing information as tabular data that does not change often. HTM tables allow resource information files to be displayed properly in a web browser.

Because there are a variety of external files that are gathered to deliver information to the user, a number of resources types are used. The resource types queries and services use the HTM file format to communicate with the Content Server. These resource types use the HDA format to get information to and from the server: environment, template, resource, and dynamic tables.

HDA File Type

An HDA (hyper data) file is a structured ASCII text file. This file format is designed to be compact to improve network communication. In addition, HDA files allow for persistent storage. This provides the ability to maintain consistency after the application reads in data and writes out any changes. The system creates several files specifically for this purpose. HDA files are used to define custom components that are added to Content Server. The types of resources that use HDA files are: HTML includes, environment, dynamic resource tables, and templates. There are two section types of an HDA file used during the customization process: Properties and ResultSets.

HDA File Structure

An HDA file contains sections that begin with *@SectionType* and end with *@end*. The two main section types in an HDA files created by the system are: *@Properties* and *@ResultSet*. When creating custom components, the ResultSet section type is primarily used.

Section Types

An HDA file is divided into two *tagged* sections of the form:

```
@SectionType sectionname
... Section data
@end
```

There are only two section types that are relevant to Component Architecture development: *@Properties* and *@ResultSet*. All other section tags are for internal application use only.

Note: None of the section types are mandatory and can be deleted if they are not being used.

Purpose

The purpose of the HDA file is to store data and communicate with the Content Server when a request for a Content Server service is made. Service request data is comprised of name/value pairs that are defined in the properties section of the HDA file named *LocalData*. When using applets to make a service request, the data exists in the form of a ResultSet.

HDA Section Type: @Properties

The *@Properties* section of an HDA file consists of a set of name/value pairs (for example, IsJava=1) separated by carriage return line feeds. This section type begins with *@Properties* name and ends with the syntax *@end*.

For custom component creation, the only valid name for a Properties section is *LocalData*. This is because the name/value pairs are only valid for the current HDA file. The LocalData section refers to data specific to this particular file.

Structure

A Properties section has the following structure:

```
@Properties LocalData
property1_name=property1_value
property2_name=property2_value
...
propertyn_name=properlyn_value
@end
```

There is no comment escape character for the Properties section of an HDA file. However, you can place comments in the file either before the start of the Properties section (@Properties) or after the end of the Properties section (@end).

An example of a Properties section is the index.hda file, located at <*home*>/documentation/data/pages/index.hda. This is a sample of that file:

```
@Properties LocalData
PageLastChanged=952094472723
LocationInfo=Directory,Public,
IsJava=1
refreshSubMonikers=
PageUrl=/intradoc/groups/public/pages/index.htm
LastChanged=-1
TemplatePage=DIRECTORY_PAGE
IdcService=PAGE_HANDLER
LinkSelectedIndex=0
```

```
PageName=index

HeaderText=This is a sample page.  The Page Name must remain
index.  The Page Properties for this index page should be
customized.

PageFunction=SavePage

dSecurityGroup=Public

restrictByGroup=1

PageType=Directory

PageTitle=Stellent Content Server Index Page

@end
```

The LocalData consists of name/value pairs. This information is only maintained during the lifetime of the request and response. Unlike information about the server environment, which rarely changes, the information for each request is dynamic. From the point of view of an HTTP request, the initial LocalData is collected from the REQUEST_METHOD, CONTENT_LENGTH, and QUERY_STRING HTTP environment variables. As the service request is processed, the values in the LocalData section will be added and changed.

HDA Section Type: @ResultSet

The *@ResultSet* section of an HDA file consists of a data representation of the results of a database query. ResultSets include serialized HDA tables.

These steps describe the page assembly process:

- Information is retrieved from the std_page_begin, std_page_end, and std_header_sections.
- The database is queried and the results are returned.
- The returned information is merged to complete the final page.
- A ResultSet becomes active during a loop of a page merge. The active ResultSet take precedence over any other ResultSets during a value search.

The @ResultSet section holds a definition of a table with the number of columns on the first line, the names of the columns on the next lines and the actual row values in the same order as the columns on the last lines.

A ResultSet section begins with @ResultSet name and ends with the *@end* tag. This section enables you to define columns and rows of data (a table) when creating components. Unlike a Properties section, a ResultSet name is not limited to a single value. The ResultSet can be given any name. However, certain names used by the Content Server are reserved.

This table lists some of the standard ResultSet names that have significance to the Content Server:

ResultSet Name	Significance
Components	This file contains references to the name and location of any components you may have created.
IntradocReports	This file contains information about any reports that have been defined in the system.
IntradocTemplates	This file holds all of the default templates for the system. Do not overwrite this file.
ResourceDefinition	This file contains information about any components that you might create.
SearchResultsTemplates	This file holds information about any custom templates created for returning SearchResults to the browser.

There is no comment character for a ResultSet section of an HDA file. Blank lines must not be left between the start of a section (@ResultSet) and the corresponding end of the section (@end). Blank lines and text can only be used between sections.

Note: An HDA file is not web viewable.

Structure

A ResultSet provides the ability to define columns and rows of data. After the @ResultSet name, the number of columns that the serialized table will contain is listed. The names of each of the columns with one column name per line are then listed. Each row of the table is then defined, one column at a time, with each column value appearing on a separate line. This is an example of the file structure for a ResultSet that has *n* columns and *m* rows:

```
@ResultSet name
n
column1-name
column2-name
…
columnn-name
row1-column1-value
row1-column2-value
…
row1-columnn-value
row2-column1-value
row2-column2-value
…
row2-columnn-value
rowm-column1-value
rowm-column2-value
…
rowm-columnn-value
@end
```

Sample ResultSet

This sample depicts a ResultSet named *scores*. It contains four columns: name, game1, game2, and game3. There are four sets of data for this ResultSet:

name	game1	game2	game3
Jim	187	145	154
Joe	125	167	121
John	134	134	123
Sam	125	114	133

```
@ResultSet scores
4
name
game1
game2
game3
Jim,
187
145
154
Joe
125
167
121
John
134
134
123
Sam
125
114
133
@end
```

Data Binder

The Content Server stores a service request internally in a Data Binder. The Data Binder manages information and organizes it into these distinct categories:

- LocalData
- ResultSets
- Environment

The Data Binder differentiates between active and non-active ResultSets during the creation of an HTML page. The Data Binder categories are used to group data to determine where the data came from and how it was created. This enables the system to determine such things as search precedence when looking up a value.

By default, when trying to evaluate the substitution of a lookup key, the data in the request is evaluated in the following order:

1. LocalData
2. Active ResultSets
3. All other ResultSets
4. Environment

Note: This precedence can be changed using Idoc Script functions.

An HDA file is a serialized Data Binder and is used for both communication and data representation. The @Properties LocalData category maps to the LocalData of the Data Binder and the @ResultSet category maps to a named result in the Data Binder.

HTM File Type

An HTM file is an HTML file type, but is not an HTML document. The difference is that an HTML file is ready for viewing in a web browser, but an HTM file is not. A number of HTM files are found in these directories:

- <*home*>/shared/config/templates/
- <*home*>/shared/config/reports/
- <*home*>/shared/config/resources/

There are three types of HTM files within the Content Server:

- templates
- reports
- resources

Templates and Reports

Templates and reports deliver a web page during the page assembly process. However, an HTM file contains a large amount of script that has not been resolved by the Content Server and will remain unresolved until the final page is assembled. These HTM files are template files, not displayable HTML files.

Resources

Resources play a variety of roles within the system. Generally, they are used to present information displayed as a web page in a browser.

HTM Tables

The HTM format is another type of table used by the Content Server. An HTM table is very similar to the HDA format, except that it uses HTML table tags to layout the format. This enables the resource files to be displayed properly in a web browser.

Structure

A table, or ResultSet, in an HTM file begins with <*@table name@*> and ends with <@end@>. Between the start and end markup tags is an HTML table. Unlike a ResultSet in an HDA file, the number of columns do not need to be specified. This is implied by the table markup.

Like an HDA file ResultSet, the column names in the first table row are listed first. The data for each row of the table follows. HTML comments are allowed within the table. The HTML style attribute can be used to format the contents to improve the presentation of the data in a web browser.

This is an example of the structure of a ResultSet in an HTM file. The ResultSet has *n* columns and *m* rows.

```
<@table TableName@>
<table border=1>
<caption><strong>Table Description</strong></caption>
<tr>
  <td>ColumnName1</td>
  <td>ColumnName2</td>
  …
  <td>ColumnNamen</td>
</tr>
<tr>
  <td>Row1ColumnValue1</td>
  <td>Row1ColumnValue2</td>
  …
  <td>Row1ColumnValuen</td>
</tr>
<tr>
  <td>Row2ColumnValue1</td>
  <td>Row2ColumnValue2</td>
  …
  <td>Row2ColumnValuen</td>
</tr>
```

```
…
<tr>
  <td>RowmColumnValue1</td>
  <td>RowmColumnValue2</td>
  …
  <td>RowmColumnValuen</td>
</tr>
</table>
<@end@>
```

Note: Any HTML syntax that does not define the data structure is ignored when the table is loaded. For example, all the <td> tags can use any of their options (such as alignment or spacing) and the title can be formatted to taste. The HTM format is useful for resources that are read in and parsed by an application but are never changed except through manual editing.

Dynamic Content Resources

Dynamic content resources are HTML markup that is used in more than one template or report file. This dynamic content consists of the resources that assemble the HTML page. These resources are defined in the <*home*>/shared/config/resources/ std_page.htm file.

Structure

Dynamic resources begin with the tag <*@dynamichtml name@*> and end with the tag <*@end@*>. The name of the resource is how the HTML markup is referenced in template and report HTM files. To reference a template or report, the HTM file contains an include statement. For example: <$include *name*$>. The variable name is the information to be included in the file. There are three pieces of dynamic content that are a part of almost every page in the Content Server web site. These are defined in the std_page.htm file:

- body_def
- std_page_begin
- std_page_end

These items are included in page templates by using the following markup <$include body_def$> <$include std_page_begin$>, and <$include std_page_end$>, respectively.

Body Definition

The body definition (BODY element) appears on almost every page in a Content Server web site. The body element definition sets the page background color, the color of hyperlinks, and the background image.

For example:

```
<@dynamichtml body_def@>

<!--Background image defined as part of body tag--->

<body

<$if background_image$>

 background="<$HttpImagesRoot$><$background_image$>"

<$elseif colorBackground$>

       bgcolor="<$colorBackground$>"

<$endif$>

     link="#663399" vlink="#CC9900"

<$if noBackgroundIndent$>marginwidth="0" marginheight="0"
topmargin="0" leftmargin="0"<$else$>topmargin="10"
leftmargin="10"

<$endif$>

>

<@end@>
```

Page Begin

This example demonstrates how most pages begin in a Content Server web site. By examining the source script, it can be determined that most of the page content is inserted into a table. This table includes several rows and columns that allow space for the sidebar and its links, space for the logo, and any additional content. This is the code for the std_page_begin resource:

```
<@dynamichtml std_page_begin@>

<$if not coreContentOnly$>
```

```
<table border=0 cellpadding=0 cellspacing=0 width="100%"
height="100%">

<tr>

<!-- sidebar for nav links -->

<td width=<$pne_nav_width$> valign=top><$include
pne_nav_links$></td>

<!--Overall page table with logo and head banner -->

    <$if widePage$>
<$StdPageWidth=550$><$else$><$StdPageWidth=500$><$endif$>

    <td valign="top"><table border=0 cellspacing=0 cellpadding=0
width="100%">

<tr>

<!-- top banner -->

<td colspan=3 valign="top" align="left" height=1  width="100%"
bgcolor="<$banner_top_color$>">

<img src= "<$HttpImagesRoot$> <$banner_top_image$>"

align="top" border="0"alt="Top banner logo."></td>

</tr>

<tr>

<td colspan=3 height=20> <!--vertical spacer--></td>

</tr>

<tr>

<!-- horizontal spacer -->

<td width=10><img src="<$HttpImagesRoot$>space.gif" alt=""
width=10></td>

<!-- purple nav bar -->

<td valign="top" align="right" valign="middle">

<img src="<$HttpImagesRoot$>header_curve.gif" width=12 height=24
border=0>

</td>

<td bgcolor="#993399" valign="middle" align="left" width="100%"
nowrap>

 <span class=headerNav>    
```

```
<a class=headerNav href="javascript:history.back()"> Back</
a>  |  <$if isTrue(#env.IsProxiedServer)$>

<a class=headerNav href="<$HttpEnterpriseCgiPath$>
?IdcService=GET_DOC_PAGE&Action=GetTemplatePage&Page=HOME_PAGE”>
Home</a>  |  

<a class=headerNav href="<$HttpCgiPath$>
IdcService=GET_DOC_PAGE&Action= GetTemplatePage&Page=HOME_PAGE">
<$#env.InstanceMenuLabel$></a>  |  

<$else$>

<a class=headerNavhref="<$HttpCgiPath$>
?IdcService=GET_DOC_PAGE&Action= GetTemplatePage&Page=HOME_PAGE">
Home</a>  |  

<$endif$>

<a class=headerNav
href="<$HttpCgiPath$>?IdcService=GET_DYNAMIC_PAGE&PageName=index"
>Library</a>  |  

<a class=headerNav
href="<$HttpCgiPath$>?<$strTrimWs(inc(’std_query_page_link_args’)
)$>"

>Search</a>  |  

<a class=headerNav href="<$HttpHelpRoot$>default.htm"
target="IntradocHelp">Help</a>

</span>

</td>

</tr>

</table>

<$endif$>

<!--Overall table row which contains sidebar and dynamic listing
of folders and documents (main display area) -->

<table border=0 width=<$StdPageWidth$> border=0 cellspacing=0
cellpadding=0>

<tr>

    <td rowspan=1000 width=15><img
src="<$HttpImagesRoot$>space.gif" alt="" width=15></td>

</tr>
```

```
<tr>
<td height=15><!-- vertical spacer --></td>
</tr>
<tr>
<td width="<$StdPageWidth$>" valign="top" align="center"
colspan=3>
<@end@>
```

Page End

This example of dynamic content shows how the script in most Content Server web pages ends. In this definition the table cell (TD element) is closed, the table row (TR element) is closed, and, the table (TABLE element) is closed.

```
<@dynamichtml std_page_end@>
<!-- new page end -->
<!--Main display area column end-->


<!--End content table -->
</td>
</tr>
</table>
```

Including Dynamic Content in a Template

This is an excerpt from the <*home*>shared/config/templates/admin.htm template file includes dynamic content in a template. This information is defined in the <*home*>shared/config/resources/std_page.htm file as <*@dynamichtml name@*> and is included in individual template files with the convention <*$include name$*>.

This example shows the admin.htm template file:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>


<head>
```

```
        <$defaultPageTitle="Administration"$>
        <$include std_html_head_declarations$>
    </head>

    <$include body_def$>

    <$include std_page_begin$>

    <$include std_header$>
    …

    <$include std_page_end$>

    </div>
    </body>
    </html>
```

Chapter

6

Understanding the Component Definition File

Overview

The component definition file is an HDA file that defines specific ResultSets. These ResultSets define the location of resources and merge information for a custom component.

There are two types of ResultSets in a component definition file:

- ResourceDefinition
- MergeRules

ResourceDefinition

The ResourceDefinition ResultSet is defined in an HDA file and is used by the Content Server to define the location of the resources that make up a custom component.

This is the structure of the ResourceDefinition ResultSet:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
resource1-type
resource1-filename
resource1-tables
resource1-loadOrder
resource2-type
resource2-filename
resource2-tables
resource2-loadOrder
...
resourcen-type
resourcen-filename
resourcen-tables
resourcen-loadOrder
@end
```

ResourceDefinition Columns

A ResourceDefinition ResultSet consists of four columns. Each column has an associated function. These are the ResourceDefinition columns:

- type
- filename
- tables
- loadOrder

type

The *type* column can be one of six resource types. These are the resource types and the associated functions:

Type	Function
Environment	Used to define global variables, as well as hiding and displaying certain metadata fields.
Dynamic Resource Table	Used to define dynamic content for HTML pages (HDA tables).
Static Resource Table	Used to define content for HTML pages (HTM tables).
Template	Used to define page and report templates.
Query	Used to define database queries.
Services	Used to define Content Server services.

filename

The *filename* column is the name of the file that defines a specific resource. This entry can be an absolute or relative path. To use a relative path the resource should be located in the appropriate custom component directory:

- resources/ directory for a resource type.
- templates/ directory for a page template.
- reports/ directory for a report template.

For example, this allows the use of the relative path templates/mytemplates.hda instead of the entire file path, c:/<*home*>/mycomponentshared /templates/ mytemplates.hda.

tables

The *tables* column includes all of the ResultSets (tables) that should be loaded from the resource file. Table names are separated with a comma. If the resource file does not include ResultSets, this value will be null. Dynamic content resources do not include table definitions, so a reference to a dynamic content file will always use null in the *tables* column.

loadOrder

The *loadOrder* column is used to determine the order in which this resource is loaded. If you have more than one resource with the same name, the last resource loaded is the one used by the system. Normally, set this to a value of one (1).

When the Content Server reads a resource definition, only the environment and dynamic content resources are actually available for use by the system. To direct the system to load resources other than environment or dynamic content, MergeRules must be defined. The MergeRules specify which resources will be loaded and which specific internal tables they will be loaded into.

Example ResourceDefinition

This is an example of a ResourceDefinition. The name, number of columns, and column names are fixed because this is a ResourceDefinition ResultSet. This ResourceDefinition defines four resources, one of each type:

Type	Filename	Tables	loadOrder
resource	resources/ mypageresources. htm	null	1
template	templates/ mytemplates.hda	MyTemplates	1

Type	Filename	Tables	loadOrder
query	resources/ myqueries.htm	MyQueries	1
service	resources/ myservices.htm	MyServices	1

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
resource
resources/mypageresources.htm
null
1
template
templates/mytemplate.hda
MyTemplates
1
query
resources/myqueries.htm
MyQueries
1
service
resources/myservices.htm
MyServices
1
@end
```

MergeRules

Environment and dynamic content resources are available as soon as they are loaded. However, for all other resources, the system needs to know where to merge the resource information. This is accomplished by creating *merge rules*. Merge rules are defined using a MergeRules ResultSet. Since this ResultSet has the name MergeRules, it comes with a predefined number of columns and predefined column names. A MergeRules ResultSet has the following structure:

```
@ResultSet MergeRules
3
fromTable
toTable
column
mergerule1-fromTable
mergerule1-toTable
mergerule1-column
mergerule2-fromTable
mergerule2-toTable
mergerule2-column
...
mergerulen-fromTable
mergerulen-toTable
mergerulen-column
@end
```

MergeRules Columns

A MergeRules ResultSet consists of three columns. Each column has an associated function. These are the MergeRules columns:

- fromTable
- toTable
- column

fromTable

The column *fromTable* represents a new table that your component has defined and loaded as part of the ResultSet ResourceDefinition. To properly perform a merge, the *fromTable* must have the identical format as the *toTabl*e.

In the previous ResourceDefinition example, three tables were loaded: MyTemplates, MyQueries, and MyServices. These three tables are now available for use as a f*romTable*.

toTable

The column *toTable* is the name of an existing table. Usually, this is one of the Content Server internal tables, such as the IntradocTemplates table or QueryTable table.

column

The column *column* is the name of the column that Content Server performs a comparison on for the merge. Usually this value will be *name*. In some cases, you may set it to null. Setting the value to null will default to the first column, which is generally a name column.

For each row of the *fromTable*, if the content of *column* is not identical to a row already in the *toTable*, a new row is added to the *toTable* and populated with the data from the row of *fromTable*. However, if the content of *column* is identical to an entry already in the *toTable*, the row in the *toTable* is replaced by the row in the *fromTable*.

Example MergeRules

In this example, two ResultSets, *scores* and *newscores* are defined. An explanation of merging newscores into scores and of merging scores into newscores is also provided.

scores

The scores ResultSet has four columns labeled: name, game1, game2, and game3. There are four rows of information in the scores ResultSet. The following figure shows the HDA file representation of scores, as well as a tabular representation. Think of the tabular representation as the ResultSet scores after it has been loaded into memory by the system.

name	game1	game2	game3
Jim	187	145	154
Joe	125	167	121
John	134	134	123
Sam	125	114	133

```
@ResultSet scores
4
name
game1
game2
game3
Jim
187
145
154
Joe
125
167
121
John
134
134
123
Sam
125
114
133
@end
```

newscores

The newscores ResultSet has the same structure as the scores ResultSet. There are four columns labeled: game1, game2, and game3. There are three rows of data in ResultSet newscores. This example shows the HDA file representation of the newscores ResultSet, as well as a tabular representation of the data. Think of the tabular representation as the ResultSet newscores after it has been loaded into memory by the system.

name	game1	game2	game3
Andy	238	220	237
Ken	165	148	145
JIm	178	183	162

```
@ResultSet scores
4
name
game1
game2
game3
Amdy
238
220
237
Ken
165
148
145
Jim
178
183
162
@end
```

Merging newscores into scores

The first merge to occur is defined in the ResultSet MergeRules of the following figure. The fromTable is defined as *newscores*, and the toTable is defined as scores. The column on which to merge is *name*.

- This figure shows that the three rows of newscores are merged into the four rows of scores. The result is the scores ResultSet is given six total rows. Row-by-row, the merge happens as follows:
- The *newscore*s row named Andy is not present in *scores*, therefore, the entire row is appended to *scores.*
- The *newscores* row named Ken is not present in *scores*, therefore, the entire row is appended to *scores.*
- The *newscores* row named Jim is present in *scores*, the old row named Jim is replaced with the contents of the *newscores* row named Jim.

name	game1	game2	game3
Jim	178	183	162
Joe	125	167	121
John	134	134	123
Sam	125	114	133
Andy	238	220	237
Ken	165	148	145

```
@ResultSet MergeRules
3
fromTable
toColumn
newscores
scores
name
@end
```

Merging scores into newscores

The second merge to occur is defined in the ResultSet MergeRules of the following figure. The fromTable is defined as *scores*, and the toTable is defined as *newscores*. The column on which to merge is *name*.

This figure shows that the four rows of scores are merged into the three rows of newscores. The result is the newscores ResultSet is given six total rows. Row-by-row, the merge happens as follows:

- The *scores* row named Jim is present in *newscores*, therefore old row named Jim is replaced with the contents of the scores row named Jim.
- The scores row named Joe is not present in *newscores*, therefore the entire row is appended to *newscores*.
- The scores row named John is not present in *newscores*, therefore the entire row is appended to *newscores*.
- The scores row named Sam is not present in *newscores*, therefore the entire row is appended to *newscores.*

name	game1	game2	game3
Andy	238	220	237
Ken	165	148	145
Jim	187	145	154
Joe	125	167	121
John	134	134	123
Sam	125	114	133

```
@ResultSet MergeRules
3
fromTable
toTable
column
scores
newscores
name
@end
```

Chapter

7

Understanding the Components HDA File

Overview

The components.hda file enables software to access your component and is located in the <*home*>/config/ directory. This file contains a ResultSet named *Components*.

Component Structure

This is the structure of the C*omponents* ResultSet:

```
@ResultSet Components
2
name
location
component1-name
component1-location
component2-name
component2-location
```

```
…
componentn-name
componentn-location
@end
```

Component Columns

A C*omponents* ResultSet consists of two columns. Each column has an associated function. These are the C*omponents* columns:

- name
- location
- column

name

The *name* column is used to identify a component in case the Content Server has problems loading the component files. If there are major problems, the server may not start. Server errors can be checked using the Content Admin Server.

location

The *location* column references a location. Any location supplied can be an absolute or relative path to the Component definition HDA file. Since the recommendation is to place any new component into its own directory outside of <*home*>, it is easiest to use an absolute path. Always use forward slashes in the path name.

You may have multiple components referenced in the ResultSet. The order that they are listed is significant. If your first component in the ResultSet has a resource with the same name as the second component, the entry in the second component will take precedence.

Implementing a Component

To implement your component, simply make a two-line entry into the <*home*>/config/components.hda file that supplies the name and location of your custom component. Any name can be used for your component, but it is recommended that the name be related to the function of your component.

This example references a component named glue.hda:

```
@ResultSet Components
2
name
location
This is my component
c:/stellent/MyComponent/glue.hda
@end
```

The Component Wizard is used to enable a custom component as does the Component Manager functionality of the Content Admin Server. When a custom component is enabled a two-line entry is made in the components.hda file.

Removing A Component

To remove your component, simply remove the two-line entry from the components.hda file. An alternative is to move the two line entry so that it appears after the *@end* tag, as in the following example:

```
@ResultSet Components
2
name
location
@end


This is my component
c:/stellent/MyComponent/glue.hda
```

After installing or removing a component the Content Server must be restarted.

The Component Wizard is also used to disable a custom component as does the Component Manager functionality of the Content Admin Server. When a custom component is disabled an entry is removed from the components.hda file.

Configuration File

The configuration file is located at <*home*>/config/config.cfg and enables you to define global variables for the system. This allows you to access global variables within your component. This example illustrates a typical configuration file:

```
#Intradoc system properties
IDC_Name=Master_on_secondserver
InstanceMenuLabel=Master_on_secondserver
InstanceDescription=Master_on_secondserver

#Database Variables
IsJdbc=false
JdbcDriver=com.ms.jdbc.odbc.JdbcOdbcDriver
JdbcConnectionString=JDBC:ODBC:intradoc
JdbcUser=sa
JdbcPassword=

#Internet Variables
HttpServerAddress=secondserver
MailServer=mail.company.com
SysAdminAddress=sysadmin@company.com
SmtpPort=25
HttpRelativeWebRoot=/stellent/
CgiFileName=idc_cgi_isapi.dll
WebProxyAdminServer=true

#General Option Variables
EnterpriseSearchAsDefault=true
```

```
#Additional Variables
IsFormsPresent=true
IntradocServerPort=4444
NtlmSecurityEnabled=standard security
HttpRelativeCgiRoot=/intradoc-cgi/
```

Important: Modifying the default variables defined in config.cfg can cause your software to malfunction.

Global variables can be defined in a separate file that has the same structure as the <*home*>/config/config.cfg file. This separate file is normally maintained with the rest of the files that define a component and is loaded by placing the following entry into the ResultSet ResourceDefinition:

```
@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
environment
component_variables.cfg
null
1
@end
```

Defining a Variable

Within the config.cfg file, a global variable can be defined by entering the variable name and the value on the same line of the file separated by an equal sign. For example, to add a variable for the e-mail address of an individual in the Complaint Department, you would add a single line to the file, similar to the following:

```
Complaints=bill@mycompany.com
```

Referencing a Variable

After creating a variable in the config.cfg file, it can be included in your templates and resources with the following syntax: <*$variablename$*>. To reference the *Complaints* variable used in the “Defining a Variable” section, you would use <*$Complaints$*>.

Chapter

8

Understanding Templates

Overview

Templates can be classified into two distinct categories: presentation templates and resource templates.

- Presentation templates are those that contain Idoc Script and HTML and will ultimately become the actual pages that the Content Server web site delivers.
- Resource templates are those that define the information that is used by the presentation templates to deliver a web page. The resource templates define pieces of dynamic content that are incorporated into presentation templates using *<$include name$>* statements.

Content Server Loading

All resources in the application are cached at start up. The Content Server supports active loading of the templates and HTML resource include files. For example, the revision history template page can be edited and its changes become instantly available. However, this is only true for templates and resource includes. The Content Server does not actively load the list of custom components, services, queries, or environment if the list of components, services, or queries has changed. If a change has occurred, the Content Server or any stand-alone applications must be restarted before the changes will be reflected in the application.

Templates File

The templates.hda file is located in the <*home*>/shared/config/templates/ directory and contains information about which presentation templates will be used to help the Content Server deliver the various default web pages.

The templates.hda file contains three ResultSets:

- IntradocTemplates
- VerityTemplates
- SearchResultsTemplates

IntradocTemplates

IntradocTemplates is a ResultSet that defines the templates used with the system. IntradocTemplates is a ResultSet that defines the templates used with the system. The ResultSet has the structure shown in the following example. A description of each column follows the ResultSet structure.

```
@ResultSet IntradocTemplates
5
name
class
formtype
filename
description
template1-name
template1-class
template1-formtype
template1-filename
template1-description
template2-name
template2-class
template2-formtype
template2-filename
template2-description
```

```
…
templaten-name
templaten-class
templaten-formtype
templaten-filename
templaten-description
@end
```

In tabular format, the information contained in the example file would have the following structure:

name	class	formtype	filename
description	HOME_PAGE	RootPage	HomePage
pne_home_page.htm	Home Page for weblayout	ADMIN_LINKS	Administration
AdministrationLinks	admin.htm	Page containing links to administration applets and forms	

IntradocTemplates Columns

An IntradocTemplates ResultSet consists of five columns. Each column has an associated function. These are the IntradocTemplates columns:

- name
- class
- formtype
- filename
- description

name

The *name* column represents the unique name of the template page. This is how the template is referenced in the Content Server CGI URLs and in code. When merging custom template file entries into the IntradocTemplates table, it is used as the merge key.

For example, the URL for the standard search page, references the name of the page, STANDARD_QUERY_PAGE. If you find the STANDARD_QUERY_PAGE entry in the IntradocTemplates table, you will see that the name of the file that implements this template is called std_query.htm.

```
IdcService=GET_DOC_PAGE&Action=GetTemplatePage&
Page=STANDARD_QUERY_PAGE
```

This is the templates.hda file entry for the STANDARD_QUERY_PAGE:

```
STANDARD_QUERY_PAGE
Search
DocQueryPage
std_query.htm
Document Search Form
```

class

The *class* column represents the general category of the template. For example, many of the template pages are part of the *document* class. For examples see the <*home*>/shared/config/templates/ templates.hda file and look at CHECKIN_LIST, CHECKIN_NEW_FORM.

Note: Currently *class* is not used by the system, but may be used in future product releases to trigger extra functionality that would be specific to a particular *class* of templates. It is good coding practice to always categorize application elements when there are a large number of them.

formtype

The *formtype* column represents the specific type of functionality the page is trying to achieve. There are almost as many form types as there are templates within the ResultSet IntradocTemplates. In some cases, the form type determines if the template needs to be updated. For example, when we add a new search results page, it is referenced by the Web Layout Editor Query Result Pages menu option.

filename

The *filename* column represents the path to the template file. This can be either a relative path or an absolute path. A relative path is relative to this templates.hda file. The relative path is relative to the file holding the reference to the file name.

description

The *description* column contains a user-friendly description of the template. It may be used by the Content Server to display a description of a selected template in the Administration Tools.

VerityTemplates

As of version 3.5.3, the software no longer uses the VerityTemplates ResultSet. However, the VerityTemplates ResultSet remains a part of the templates.hda file as legacy code.

SearchResultTemplates

The SearchResultTemplates are used to build the search result pages of the Content Server web site. SearchResultTemplates contain Idoc Script, which is processed at the time a search is actually requested by a web browser.

Note: This ResultSet was known as VeritySearchAPITemplates prior to version 3.6 of our software.

The ResultSet has the structure shown in the following example. A description of each column follows the ResultSet structure.

```
@ResultSet SearchResultsTemplates
6
name
formtype
filename
outfilename
flexdata
description
template1-name
template1-formtype
template1-filename
template1-outfilename
template1-flexdata
template1-description
template2-name
template2-formtype
template2-filename
```

```
template2-outfilename
template2-flexdata
template2-description
...
templaten-name
templaten-formtype
templaten-filename
templaten-outfilename
templaten-flexdata
templaten-description
@end
```

SearchResultTemplates Columns

A SearchResultTemplates ResultSet consists of six columns. Each column has an associated function. These are the SearchResultTemplates columns:

- name
- formtype
- filename
- outfilename
- flexdata
- description

name

The *name* column is the unique name of the template. This is how the template is referenced within the Web Layout Editor applet. When a result template is referenced on a search form or query page, this is the name that is used.

formtype

The *formtype* column is the specific type of functionality the page is trying to achieve. Only ResultsPage is currently supported. This form type identifies the template as one that can be used to create query result pages using the Web Layout Editor, Query Result Pages menu.

filename

The *filename* column represents the path to the template file. This can be either a relative path or an absolute path. A relative path is relative to the templates.hda file.

If this template file is used to create a new search results template, the Web Layout Editor will create a new template with this name in the <*home*>/shared/config/templates/results/ directory and also create an entry in the ResultSet CurrentVerityTemplates.

outfilename

The *outfilename* column value is always null. Since search is a function of the Content Server, there is no search result template file that requires access. The results of a search are communicated from search server to the Content Server for final formatting and presentation to the web browser.

flexdata

The *flexdata* column contains information that is placed into the areas, Text1 and Text2, of a SearchResultTemplates file. The contents of Text1 and Text2 can be edited by accessing the Query Result Pages from the Web Layout Editor.

As items are place in the Text1 and Text2 areas, the Content Server converts the entries into Idoc Script that can be understood by the Content Server. This script, along with any additional markup you provide, is entered into the flexdata column.

The format of text entered in the *flexdata* column is:

```
Text2 “text 2 contents”%<Tab>Text1 “text 1 contents”%
```

In this instance, <Tab> is a literal tab character. The default value for *flexdata* for the only SearchResultTemplates template (search_results.htm) is:

```
Text2 <$dDocTitle$>%Text1 <$dDocName$>%
```

For any new SearchResultTemplates templates you define, the entries you provide for *flexdata* in the definition of a new template will appear as the default entries when a user adds a new Query Results page.

description

The description column contains a description of the template. The software may use this information to display the description of a selected template when using the Administration Tools. This is an example of what the ResultSet SearchResults looks like in the templates.hda file:

```
@ResultSet SearchResultTemplates
6
name
formtype
filename
outfilename
flexdata
description
StandardResults
SearchResultsPage
search_results.htm
null
Text2 <$dDocTitle$>%Text1 <$dDocName$>%
Page presenting results of a search using Verity Search API
@end
```

Defining Custom Templates

When new templates are created, they are made available by creating a ResultSet that describes them. The ResultSet name you create should be assigned a unique name, such as *MyTemplates*. The structure of the ResultSet must be identical to the IntradocTemplates ResultSet so that you can define a MergeRule from the custom templates file, MyTemplates to IntradocTemplates.

The name you assign to your templates page depends on whether you are trying to replace an existing template, or just augmenting the templates that come with the product.

- To replace an existing template page, use the same name for your template.
- To add a template page that you will create a reference to in the templates.hda file, use a new, unique name for your template.

The Content Server loads the page templates in a series of steps where each following step may redefine a template loaded earlier or add a new one. A template is an entry in a table that describes which HTML template file should be loaded for the particular template.

Chapter

9

Understanding Content-Centered Template Metadata

Overview

The manipulation of metadata is handled by the process of creating HTML resource includes using the *super* tag to override default behavior. Within specified parameters metadata manipulation can be performed for any of these content-centered templates.

- checkin_new.htm
- checkin_sel.htm
- doc_info.htm
- update_docinfo.htm
- std_query.htm

Multi-Checkin Environment File

The *multi_checkin_environment.cfg* configuration file is part of the MultiCheckIn component. This file is used to manipulate metadata fields on content-centered template pages for certain content types. This configuration file is an environment-type resource that provides information to the Content Server concerning the interaction with various content-centered pages. The configuration file, along with the HTML include resources, uses name/value pairs to suppress, display, pre-fill, or make metadata fields read-only based on the chosen content type.

Note: The std_page.htm file provides a list of universal resource includes that can be used by any Content Server page and a list of resource includes for pages that have flex areas (the two check in pages, the doc info page, and the search page). This file is located in the <*home*>/shared/config/resources/ directory.

Multi-Checkin Menu Display

The *UseMultiCheckinOnSidebar* environment setting enables or disables a pull-down menu on the portal. If the environment setting is disabled, the multi-checkin menu will only be accessible on the Content Management form. The multi-checkin menu display is defined in the multi_checkin_environment.cfg file using this format:

```
UseMultiCheckinOnSidebar=true
```

- Setting to TRUE enables the pull-down menu on the portal.
- Setting to FALSE disables the pull-down menu on the portal.

Multi-Checkin Content Types

The *MultiCheckinTypes* setting defines the list of content types that have special check in pages. Each content type must have a set of hidden and read-only fields. The multi-check in content types are defined in the multi_checkin_environment.cfg file using this format:

```
MultiCheckinTypes=ADACCT,ADCORP,ADENG,ADHR
```

The Configuration Manager applet enables you to create custom New Check In pages for custom content types. To create custom New Check In pages you must add the custom content type to the list and make an associated configuration entry for that content type.

Custom metadata fields are prefixed with an *x* on Content Server HTML pages. Each content type should define fields using this convention:

```
ContentTypeName_hide=xCustomMeta1, xCustomMeta2

ContentTypeName_checkin_readOnly=xCustomMeta3,xCustomMeta4

ContentTypeName_update_readOnly=xCustomMeta1, xCustomMeta2,
xCustomMeta3

ContentTypeName_xComments=This is the default comment for an
ADACCT field on the checkin page.

ContentTypeName_xCustomMeta3=This value will show on the checkin
page, but its uneditable.
```

These content types have associated special check-in pages:

- ADACCT
- ADCORP
- ADENG
- ADHR

For ADACCT, no metadata is hidden, no fields are read only, and the comment field is pre-filled on check in.

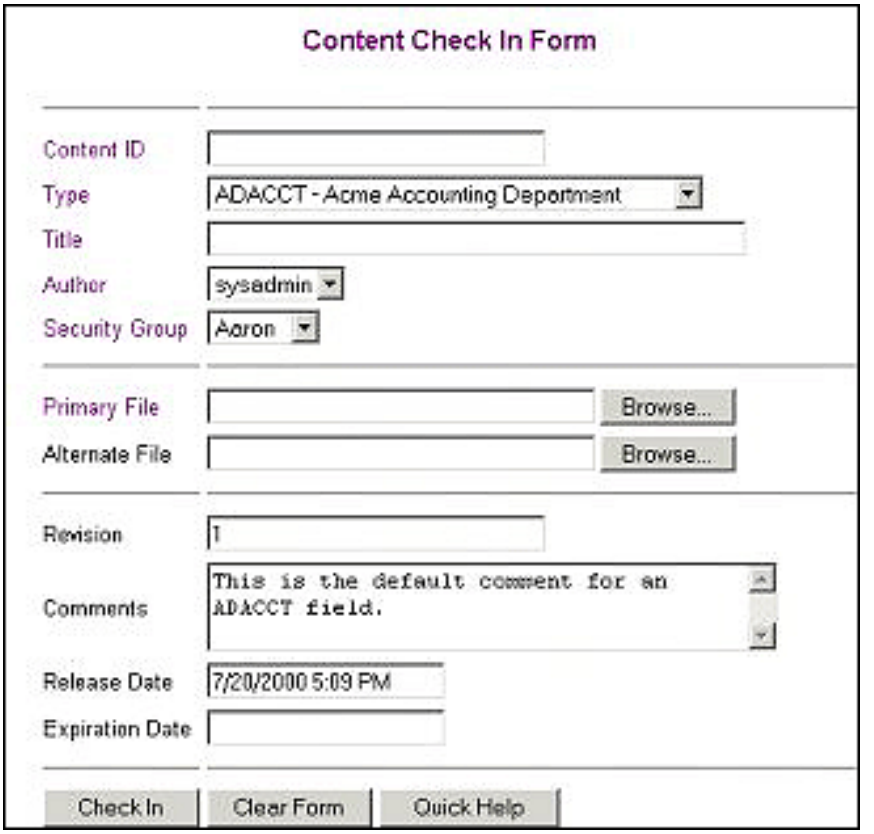
```
ADACCT_hide=

ADACCT_checkin_readOnly=

ADACCT_update_readOnly=

ADACCT_xComments=This is the default comment for an ADACCT field.
```

Example Content Type ADACCT:



For ADCORP, no metadata is hidden, no fields are read only on check in, but on the update page the Comments field will be read only. Also, the comment field is pre-filled on check in with a default value.

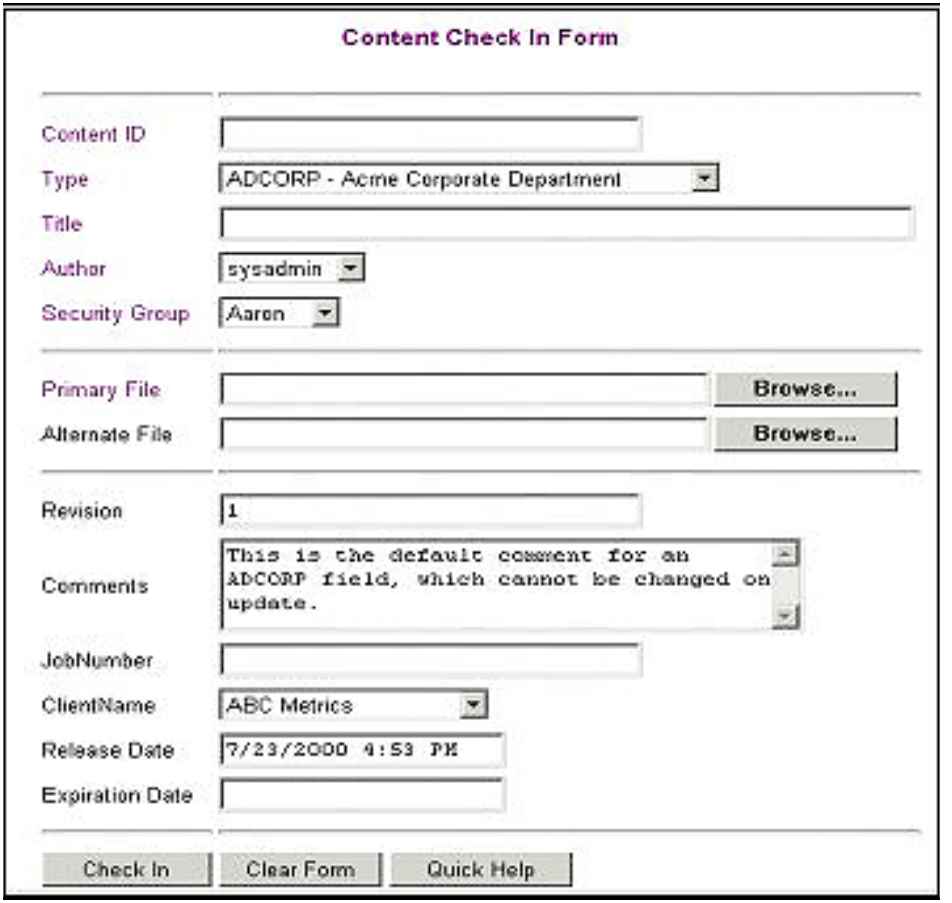
```
ADCORP_hide=

ADCORP_checkin_readOnly=

ADCORP_update_readOnly=xComments

ADCORP_xComments=This is the default comment for an ADCORP field,
which cannot be changed on update.
```

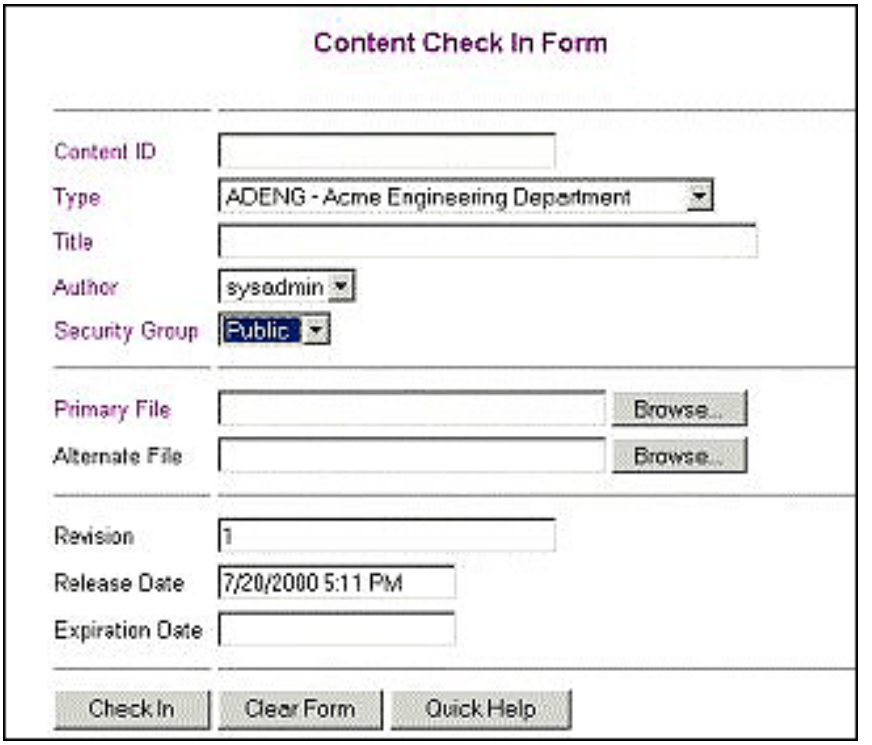
Example Content Type ADCORP:



For ADENG, the Comments field is hidden entirely

```
ADENG_hide=xComments
ADENG_checkin_readOnly=
ADENG_update_readOnly=
```

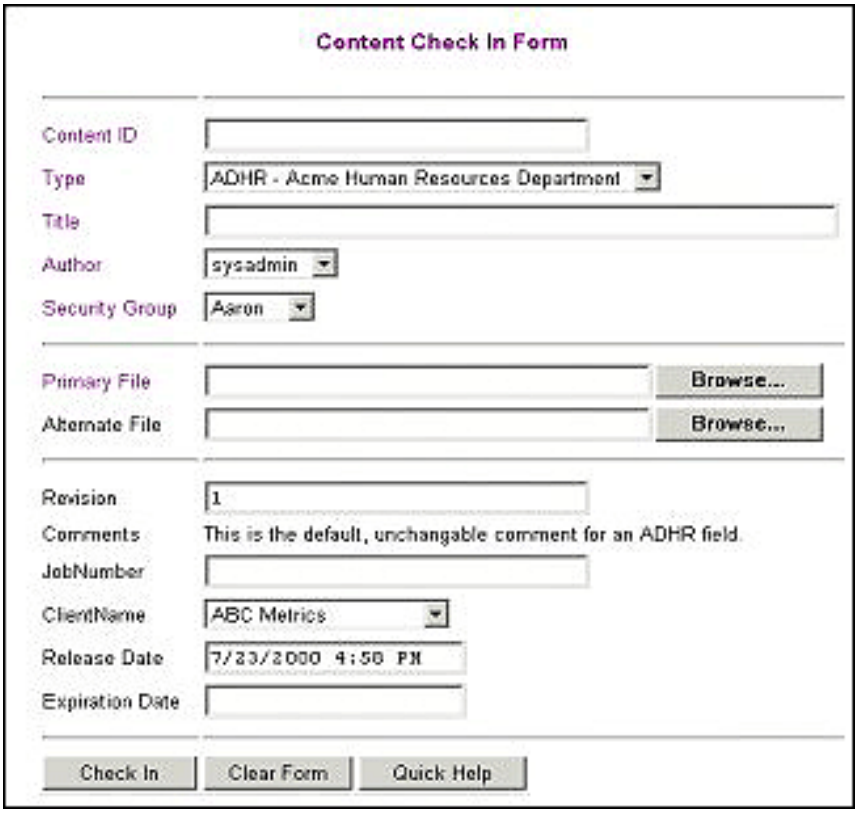
Example Content Type ADENG:



For ADHR, the Comments field is read only always, and set to a default value.

```
ADHR_hide=
ADHR_checkin_readOnly=xComments
ADHR_update_readOnly=xComments
ADHR_xComments=This is the default, unchangable comment for an
ADHR field.
```

Content Type ADHR:



Chapter

10

Understanding Query and Service Resources

Overview

There are two types of resources: query and service. These resources comprise some of the main coding mechanisms that drive the software.

Query Resource

Queries are used with the product to manage information in the system database. Queries are used in conjunction with service scripts to perform such tasks as adding to, deleting and retrieving data in the Content Server database.

These are general guidelines for developing your own query:

- Define a new query in an HTM file. The file must include a table that is identical in structure to the QueryTable table.
- Load the query by defining it in a ResourceDefinition ResultSet.
- Merge the table defining your query with the QueryTable table.

Query Definition Tables

A Query resource definition points to an HTM file. The HTM file defines a table with a specific format for query definitions. To better understand the definition, look at the Query resource definition that comes with the system <*home*>/shared/config/resources/query.htm.

This HTM file contains two query tables:

- QueryTable
- QueryWebChangesTable

These HTM tables are delimited with a start tag <@table *tablename*@> and an end tag <@end@>. The content of the table is held in an HTML table element. The QueryWebChangesTable contains queries that are used to maintain the HTML pages on a Content Server web site.

Query Definition Table Columns

Both QueryTable and QueryWebChangesTable consist of three columns. Each column has an associated function. These are the three columns:

- name
- queryStr
- parameters

name

The *name* column contains a unique name for the query. To override a query, you would use the same name for a query that you define. To add a new query, use any other unique name. Normally, the first character of the query name defines the query type:

Query Type	Description
D	delete query
I	insert query
Q	select query
U	update query

queryStr

The *queryStr* column defines the query. This query is defined using SQL. If there are any parameters, their place is held with a question mark (?) as an escape character.

parameters

The *parameters* column describes the parameters that are passed to the query. A query is called from a service and a service is called by a web browser. It is the responsibility of the web browser to provide the values for each of the parameters for the query. This can be done with a FORM element in the web page. In the case of the DOC_INFO service, the parameter is provided in a directory listing or query result page, as show in the following figure. The URL for DOC_INFO is created with the dID parameter specified as part of the URL.

Database Tables

This table lists the database tables along with a brief description of each.

Table Name	Description
Alias	Provides a list of workflow aliases and their descriptions.
AliasUser	Provides a list that associates aliases with users.
Config	Provides a record of database changes. This feature references the database list to determine whether the database is configured properly. If a change is needed, the feature updates the database and records the change in the Config table for future reference.
Counters	Provides centralized storage of sequence numbers used by the application.
DocFormats	Provides a list of formats and their associated conversion methods and descriptions.

Table Name	Description
DocMeta	Provides a table containing the custom DocInfo field values for each document. This is updated by the system server when content items are checked in, deleted, or updated.
DocMetaDefinition	Provides a list of the custom DocInfo fields and their attributes.
DocTypes	The service returns a list of the content item types (.doc, .gif, etc.), their descriptions, and their file name
DocumentAccounts	Provides a list of accounts.
DocumentHistory	Provides a journal of content item transactions such as *checkin*, *checkout*, *delete*, or *update*.
Documents	Provides a list of content item files in the system. Each file normally has two records: one for the native file and one for the web file.
ExtensionFormatMap	Provides a list of extensions defined in the system and the format each is mapped to.
OptionsList	Provides a table of all option lists. Each list has a common key value, option value, and order.
ProjectDocuments	Provides a table that stores information about all content items associated with a Content Publisher project.
ProblemReports	Provides a table that contains problem report information that is generated through the workflow process.

Table Name	Description
RegisteredProjects	Provides a table that stores information about any projects registered through Content Publisher.
Revisions	Provides a list of all content items in the system. One record for each revision of each document is provided including the status of that revision and any required metadata values.
RoleDefinition	Provides a list of roles and their permissions to each security group: one row for each role of each security group.
SecurityGroups	Provides a list of security groups and their descriptions.
Subscription	Provides a list of currently subscribed content items.
Users	Provides a list of all users registered in the system with their primary attributes: username, full name, password, e-mail address, directory, old style role (when only one role was given each user), type, and password encoding.
UserSecurityAttributes	Provides a list of users and their security attributes. This is where the new account and multiple role data for each user identity are stored.
WorkflowAliases	Provides a table used to associate user aliases to workflow steps.
WorkflowCriteria	Provides a list of workflow criteria used to build the where *clause* in the query that determines if a content item should follow a particular workflow.

Table Name	Description
WorkflowDocAttributes	Provides an internal status table that stores information about content items in active workflows.
WorkflowDocuments	Provides a list of all content items in workflows. This is updated by the system server to keep track of the status of content items (state and step) that are in workflows.
Workflows	Provides a list of workflows including their description, security group, status, and type.
WorkflowStates	Provides an internal status table that stores information about content items in active workflows.
WorkflowSteps	Provides a list of workflow steps, including step description, type, and number of reviewers required to pass step.

Example Query

This script is the QdocInfo query as it is defined in the file <*home*>/shared/config/resources/query.htm. The queryStr is a SQL select statement that obtains the necessary information to display about a file in the DOC_INFO template page. This is the page that will be displayed when a user requests the information page (the *i* icon) from the search results page.

```
<tr>
<td>QdocInfo</td>
<td>SELECT DocMeta.*, Documents.*, Revisions.*
     FROM DocMeta, Documents, Revisions
     WHERE DocMeta.dID = Revisions.dID AND
Revisions.dID=Documents.dID
     AND Revisions.dID=? AND Revisions.dStatus<>'DELETED' AND
     Documents.dIsPrimary<>0
</td>
<td>dID int</td>
</tr>
```

Notice that this query joins the three tables (DocMeta, Revisions, and Documents) on the dID field (content ID), which is also the parameter for this query. This query also takes one argument, the dID (content ID). The dID parameter is provided by the URL that requests the DOC_INFO service.

Service Resource

A service is a function performed by the system server on behalf of the web browser (the client). For example, the standard query page is delivered to your web browser as a service when a request is made to get the search form by clicking Search link on the portal page. The URL for the page includes the following information:

```
IdcService=GET_DOC_PAGE&Action=GetTemplatePage&Page=STANDARD_QUERY_PAGE
```

An IdcService placed in a URL indicates that a service is being requested from the system server. A service provides a function for a web browser. However services are functions that can be performed by the server on behalf of the entire system and the system server is written so that it will use services when it needs to perform a task.

A service is defined by a script. The script defines the name, attributes and actions of the service. A service script is defined in an HTM file, but the service is also dependent upon other resource definitions to perform its job. A service needs a template, and most likely a query. The HTM file defines a table with a specific format for a service definition.

The file <*home*>/config/shared/resources/std_services.htm provides a sample of scripted services.

Important: Do not edit this file in a graphical browser in its graphical mode. Use a text editor

These are the general steps needed to define a new service:

1. Define a service in an HTM file. The file must include a table that is identical in structure to the StandardServices table.
2. Load the service by defining it in a ResourceDefinition ResultSet.
3. Merge the table defining the service with the StandardServices table.

Service Resource Structure

The structure of a service-type resource is defined by a three column table. The table is delimited with a start tag <@table “tablename”@> and an end tag <@end@>. The first column contains the service’s unique name. The second column describes the attributes of the service. The third column describes the actions that are performed by the service.

This example shows the HTML markup for a service entry in this table. This describes a service with *n* actions:

```
<tr>
<td>service name</td>
<td>service type
     access level
     template page
     sub-service
     subjects notified<br>
     error message</td>
<td>type of action1:function name1:function parameters1:action
control mask1:error message1[<br>]
     type of action2:function name2:function parameters2:action
control mask2:error message2[<br>]
     ...
     type of actionn:function namen:function parametersn:action
control maskn:error messagen</td>
</tr>
```

The
 tag at the end of each action line is strictly for display purposes only and is optional. However, the </td> must occur on the same line as the last action.

Service Name

This column contains information about the unique name of the service.

```
<td>GET_DYNAMIC_PAGE</td>
```

The reference to a service called in a URL is the service name.For example, this URL is calling the service named GET_DYNAMIC_PAGE:

```
/intradoc-cgi/idc_cgi_isapi.dll?IdcService=GET_DYNAMIC_PAGE&
PageName=index
```

Service Attributes

The *service attribute* column is composed of six distinct items. This example shows the syntax of these items. Following the syntax is a description of each attribute.

```
<td>service type
     access level
     template page
     sub-service
     subjects notified<br>
     error message</td>
```

Service Class

There are several types or class of services, and the class of service determines, in part, what actions can be performed by the service. There are actions that all services share, and there are actions that are quite specific to the service type. These are the types of services currently available:

Service Class	Description
Service	The default service.
DocService	Used for performing actions on content items, for example: check in/out, content item information, resubmit, etc.
FileService	Used to retrieve files from the system, for example: get copy.
MetaService	Used to manage doc info fields.
PageHandlerService	Used by Web Layout Editor to edit pages.
UserService	Used to manage users, for example: add/edit/ delete users.
WorkflowService	Used to manage workflows.

Access Level

Each service calls a global security check to determine if the logged in user has permission to execute the service. The global security check is only relevant if the service requires global privilege. The check validates if the user needs to be part of the administration role or if only a given privilege is required (less than ADMIN_PRIVILEGE) on at least one group.

The bit flags are combined with a logical AND to create an access level:

READ_PRIVILEGE = 1
WRITE_PRIVILEGE = 2
DELETE_PRIVILEGE = 4
ADMIN_PRIVILEGE = 8
GLOBAL_PRIVILEGE = 16

For example, to access the Administration page, the service requires the user to be part of the administration role. Consequently, users need to have global administration privileges and the service has the access level set to 24. If the user wants to access the check in page, the user needs write privileges on at least one group, and the access level of the security group is set to 18.

If no user is logged in and the service has access level with the GLOBAL_PRIVILEGE flag set, a log on prompt is returned. This log on prompt forces the user to log into the system before the product will perform the service.

Template Page

The template page is used to communicate a successful request back to the web browser. Information that the service gathers is merged with the template page. Not all types of services require or even use a template page. For example, the PageHandlerService, which is called from an applet, does not specify a template page. The template page name is mapped to an HTML file using the templates.hda file.

Sub-Service

The service may define a sub-service to execute, otherwise, the value null is used. For example, the service ADD_WORKFLOWDOCUMENT executes the sub-service ADD_WORKFLOWDOCUMENT_SUB. This sub-service is a workflow related service that adds a revised content item to the workflow and consists of these actions:

- Queries whether the content item workflow is locked.

- Inserts the workflow content item information in the database.
- Retrieves the workflow content item name from the database.
- Evaluates the revision status of the content item.
- Creates a new revision.

Subjects Notified

If a service changes one or more subjects, it must notify the affected subjects of the changes. The subjects notified string is a comma-separated list of changed subjects. For example, the ADD_USER service adds a new user to the system and subsequently informs the system that the ‘users’ subject has changed. Possible subjects are: accounts, aliases, collections, docformats, doctypes, documents, dynamicqueries, metadata, metaoptlists, templates, and users. You can think of subjects as subsystems within the product.

Each service by default will inform its requestor of changes to subjects. Consequently, the PING_SERVER service, which has no action, is used by the Administration applets to detect changes in the state of the server.

Error Message

The error message is returned by the service, if no action overrides it. Each action can have an error message associated with it that would override the error message provided as an attribute. If the action error message is not null, it becomes the error message for the remainder of the actions in the service. If it is null, the error message remains unchanged from the previous action. For example, the error message defined as an attribute of CHECKIN_NEW_FORM is “Unable to build check in form,” but on executing the second action it becomes “Error retrieving option lists for custom fields.”

```
<tr>
     <td>CHECKIN_NEW_FORM</td>
     <td>DocService
       18
       CHECKIN_NEW_FORM
       null
       null<br>
       Unable to build check in form.</td>
     <td>3:setLocalValues:isNew,1:0:null
```

```
      3:loadMetaOptionsLists::0:Error retrieving option lists
for custom fields.
      3:loadDocDefaults::0:null
      3:loadDefaultInfo::0:null
      3:loadMetaDefaults::0:null</td>
</tr>
```

Service Actions

The third column of a defined service are the service actions. Each service may contain one or more actions, which determine what happens on execution. An action is defined by the following syntax:

```
type of action:function name:function parameters:action control
mask:error message
```

An action consists of five parts, each part separated from the previous part by a colon. If there is no entry for a part, then the part is left empty. In such a case, you will find successive colons.

Type of Action

An action can be used to execute an SQL statement, perform a query, run code, cache the results of a query, and load an option list. These are the possible types of actions:

Action Type	Action Function
QUERY_TYPE = 1	For QUERY_TYPE, the function must be a “select” query.
EXECUTE_TYPE = 2	For EXECUTE_TYPE, the function specifies a query that performs an action on the database.
CODE_TYPE = 3	For CODE_TYPE, the function specifies a code module that is a part of the Java class implementing the service.

Action Type	Action Function
OPTION_TYPE = 4	For OPTION_TYPE, the function refers to an option list stored in the system.
CACHE_RESULT_TYPE = 5	For CACHE_RESULT_TYPE, the function is as in QUERY_TYPE, but here the results returned by the query are stored for later use.

Note: The difference between QUERY_TYPE and CACHE_RESULT_TYPE is that in the first case the query is immediately discarded.

Function Name

The function name determines which query or Java function is used to perform the action. The function name is restricted by the type of service and the type of action.

Function Parameters

The parameters that are used by the functions are comma-separated. In the case of QUERY_TYPE and CACHE_RESULT_TYPE, the first parameter will be the name the action assigns to the ResultSet returned from the query. This ResultSet can then be referenced in the template page. For OPTION_TYPE, the parameters are optional. However, if they are given, they are used as follows: the first parameter is the key under which the option list is loaded; the second parameter is the selected value for display on an HTML page.

The control mask is especially useful in controlling the results from queries to the database. Possible bit values and their meanings are shown in the following table. These values can be logically combined using AND. For example, a database query that checks to make sure that a content item does not exist, and also starts a database transaction to add a new content item would have a control mask value of 20 (16 + 4).

Control Mask	Description
CONTROL_IGNORE_ERROR = 1	Do not abort the service on error.
CONTROL_MUST_EXIST = 2	At least one record must be returned by the query.
CONTROL_BEGIN_TRAN = 4	Starts a database transaction.

Control Mask	Description
CONTROL_COMMIT_TRAN = 8	Concludes a database transaction.
CONTROL_MUST_NOT_EXIST = 16	Query must not return any rows.

Note: CONTROL_MUST_EXIST and CONTROL_MUST_NOT_EXIST are used only for QUERY_TYPE and CACHE_RESULT_TYPE.

Example Service

The DOC_INFO service provides a good example of how queries and services are related. The DOC_INFO service definition from the <*home*>/config/resources/std_services.htm file is shown:

```
<tr>
<td>DOC_INFO</td>
<td>DocService
    1
    DOC_INFO
    null
    null<br>
    Unable to retrieve information about the revision.</td>
<td>5:QdocInfo:DOC_INFO:2: This document no longer exists.
    3:checkSecurity:DOC_INFO:0:Unable to retrieve information
for ''{dDocName}''.
    3:getDocFormats:QdocFormats:0:null
    3:getURLAbsolute::0:null
    3:getUserMailAddress:dDocAuthor,AuthorAddress:0:null

3:getUserMailAddress:dCheckoutUser,CheckoutUserAddress:0:null
    3:getWorkflowInfo:WF_INFO:0:null
    3:getDocSubscriptionInfo:QisSubscribed:0:null
    5:QrevHistory:REVISION_HISTORY:0: Unable to retrieve
revision history for ''{dDocName}''.<br></td>
</tr>
```

This table summarizes the attributes of the DOC_INFO service.

Attribute	Value
Service Type	DocService This service is providing information about a content item.
Access Level	1 The user requesting the service must have read privilege on the content item.
Template Page	DOC_INFO This service uses the DOC_INFO template (doc_info.htm file).
Sub-Service	null This service does not define a sub-service to execute.
Subjects Notified	null No subjects are affected by this service.
Error Message	Unable to retrieve information about the revision.

The template page for the DOC_INFO service is the DOC_INFO template. It is important to know what is happening between the files so that you can understand the interactions between the template page and the actions performed in a service.

The definition for the content that the doc_info.htm template contains is located in the <*home*>/shared/config/resources/std_page.htm file. Code from both files appear in the following markup section:

Markup from the <*home*>/shared/config/templates/doc_info.htm file:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
 <$include std_info_html_head_declarations$>
</head>
<$include info_body_def$>
<$include info_page_content$>
</body>
</html>
```

Markup from the <*home*>/shared/config/resources/std_page.htm file that defines what will appear in the doc_info.htm template:

```
<@dynamichtml info_page_content@>
<$include std_page_begin$>
<$include std_header$>
…
 <!-- Do a loop on DOC_INFO so that all substitution tags
will use DOC_INFO as their first place to find their values.
Otherwise their is confusion between this result set and the
REVISION_HISTORY table that comes later.  For example
'dStatus' is a value in both tables-->
 <$loop DOC_INFO$>
 <$if AllowPrimaryMetaFile and isTrue(AllowPrimaryMetaFile)
and
    isTrue(dFormat like "*idcmeta*")$>
    <$showPrimaryMetaFileFields = "1"$>
 <$endif$>
 <$include doc_info_notify_data$>


 <table border=0 cellpadding=2 cellspacing=0
width=<$docInfoWidth-30$>>
 <caption align=top><h4 class=pageTitle><$pageTitle$></
caption>
```

```
 <$include special_checkin_fields1$>
 <$include std_revision_label_field$>
 <$include std_document_type_field$>
 <$include std_document_title_field$>
 <$include author_checkin_field$>
 <$include std_meta_fields$>
 <$include security_checkin_fields$>
 <$include checkout_author_info_field$>
 <$if IsStagingDoc$>
 <$include doc_date_fields$>
 <$endif$>
 <$fieldName = "dStatus", fieldCaption = "Status"$><$include
std_displayonly_field$>
 <$if HasOriginal$>
 <$fieldName = "dDocFormats", fieldCaption =
"Formats"$><$include std_display_field$>
 <$endif$>
 <$include workflow_list_for_doc$>
 <$if HasUrl$>
   <$include doc_url_field$>
 <$endif$>
 <$if HasOriginal and not ClientControlled and not
showPrimaryMetaFileFields$>
   <$fieldName = "dOriginalName", fieldCaption = "Get Native
File"$>
   <$if DownloadApplet$>
     <$valueStyle="xxsmall", fieldValue =
strTrimWs(inc("download_file_by_applet_form_content"))$>
   <$else$>
     <$fieldValue = strTrimWs(inc("doc_file_get_copy"))$>
   <$endif$>
   <$if DownloadApplet$><form name=downloadForm><$endif$>
```

```
   <$include std_displayonly_field$>
   <$if DownloadApplet$></form><$endif$>
 <$endif$>
 <$if IsFailedConversion or IsFailedIndex or
IsDocRefinePassthru$>
 <$if IsFailedConversion$><$include
std_namevalue_separator$><$endif$>
 <tr>
   <td align=right><span class=errorHighlight>
   <$if IsFailedIndex$>Index Error:
   <$else$>Conversion Error:
   <$endif$></span></td>
   <td>
     <table>
     <tr>
       <td><span class=tableEntry>
       <$dMessage$>
       <$if IsFailedIndex$>
       <br>Content has been indexed with Info only.
       Resubmit should only be performed if the problem has
been resolved.
       <$elseif IsDocRefinePassthru$>
       <br>Content Refinery failed to convert the content
item but released it to the
       web by copying the native file.
       <$endif$></span></td>
       <td><form action="<$HttpCgiPath$>" method="POST">
         <input type=hidden name=dID value="<$dID$>">
         <input type=hidden name=dDocName
value="<$dDocName$>">
         <input type=hidden name=IdcService
value="RESUBMIT_FOR_CONVERSION">
         <input type=submit value=" Resubmit ">
```

```
          <$if ClientControlled$>
            <input type=hidden name=ClientControlled
value="DocMan">
          <$endif$>
          </form></td>
      </tr>
      </table>
   </td>
 </tr>
 <$if IsFailedConversion$><$include
std_namevalue_separator$><$endif$>
 <$endif$>
 </table>
 <$if IsNotSyncRev$>
 <table width="100%">
   <tr>
     <td align=center><span class=errorHighlight>The local
copy of this content item has
       not been updated to the latest revision.  Use <i>Get
Native File</i> or <i>Check out</i>
       to update your local copy of <i><$dDocName$></i>.</
span></td>
   </tr>
 </table>
 <$endif$>


 <$if IsStagingDoc$>
 <br>
 <table width="90%">
 <tr>
   <td width="20%" align=center><$include
doc_problem_reports$></td>
```

```
   <td width="20%" align=center><$include
project_problem_reports$></td>
 </tr>
 </table>
 <$include doc_provider_info$>
 <$else$>
 <table width="90%">
 <tr>
   <$if ClientControlled$>
     <td width="20%" align=center><$include
doc_select_actions$></td>
   <$else$>
     <td width="20%" align=center><$include
doc_file_undo_checkout$></td>
     <td width="20%" align=center><$include
doc_file_checkout$></td>
     <td width="20%" align=center><$if
showPrimaryMetaFileFields$><$include meta_file_update$>
       <$else$><$include doc_file_update$><$endif$></td>
   <$endif$>
   <td width="20%" align=left><$include
doc_subscription_unsubscription$></td>
   <$if ClientControlled$>
   <td width="20%"></td>
   <td width="20%"></td>
   <$endif$>
 </tr>
 </table>
 <$endif$>
 <$if HasOriginal and DownloadApplet$>
 <$include download_native_applet$>
 <$endif$>
```

```
 <!-- end loop on DOC_INFO-->
 <$endloop$>
 <$if IsStagingDoc$>
 <!-- present a problem report form -->
 <$include doc_add_problem_report$>
 <$else$>
 <!-- Table holding information about all revisions of this
document-->
 <$include doc_rev_table$>
 <$endif$>
 </td>
</tr>
</table>
<$include std_page_end$>
<@end@>
```

A service can have one or more service actions associated with it. In the case of the DOC_INFO service, the service consists of ten actions:

Attribute	Value
1	Cached query action that retrieves information from the database using a query. This action retrieves content item information. The result of this query is assigned to the parameter DOC_INFO and stored for later use. The control mask setting specifies that the query must return a record or the action fails with the given error message. The action throws a data exception if the content item no longer exists and returns an error message. This content item no longer exists
2	Code action specifying a code module that is a part of the Java class implementing the service. This action retrieves the data assigned to the parameter DOC_INFO and maps the result set values for *dStatus* and *dDocTitle*.
3	Code action specifying a code module that is a part of the Java class implementing the service. This action retrieves the data assigned to the parameter DOC_INFO and evaluates the assigned security level to verify that the user is authorized to perform this action. If the user fails the security check a message is returned. *Unable to retrieve information for "{dDocName}."*

Attribute	Value
4	Code action specifying a code module that is a part of the Java class implementing the service. This action retrieves the file formats for the content item. The action passes *QdocFormats* as a parameter (defined in <home>/config/resources/query.htm). The file formats are passed to the *Formats:* entry of the DOC_INFO template.
5	Code action specifying a code module that is a part of the Java class implementing the service. This action resolves the URL of the content item. The URL is passed to the *Web Location:* entry of the DOC_INFO template.
6	Code action specifying a code module that is a part of the Java class implementing the service. This action resolves the e-mail address of the content item author and the user who has checked out the content item. The action passes *dDocAuthor* and *AuthorAddress* as parameters.
7	Code action specifying a code module that is a part of the Java class implementing the service. This action resolves the email address of the content item author and the user who has checked out the content item. The action passes *dCheckoutUser* and *CheckoutUserAddress* as parameters.

Attribute	Value
8	Code action specifying a code module that is a part of the Java class implementing the service. This action evaluates whether the content item is part of a workflow. The action passes WF_INFO as a parameter. The DOC_INFO template is referenced and if WF_INFO exists then workflow information is included in the DOC_INFO template.
8	Code action specifying a code module that is a part of the Java class implementing the service. This action evaluates whether the current user has subscribed to the content item and modifies the DOC_INFO page. If the current user is subscribed, an **Unsubscribe** button is displayed. If the user is not subscribed, a **Subscribe** button is displayed. The action passes *QisSubscribed* as a parameter (defined in <home>/config/resources/query.htm).
10	Cached query action that retrieves information from the database using a query. This action retrieves revision history information. The result of this query is assigned to the parameter REVISION_HISTORY. The DOC_INFO template uses REVISION_HISTORY in a loop to present information about each revision in the DOC_INFO page. If the action fails, this error message is displayed: *Unable to retrieve revision history for "{dDocName}."*

Chapter

11

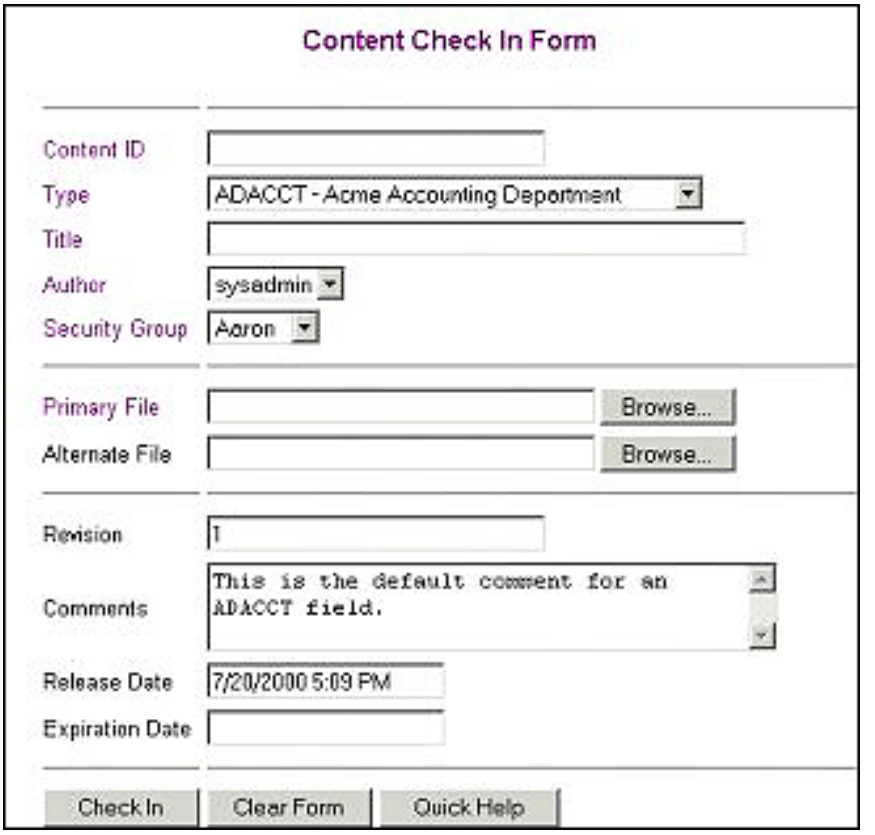
Understanding the MultiCheckin Component

Overview

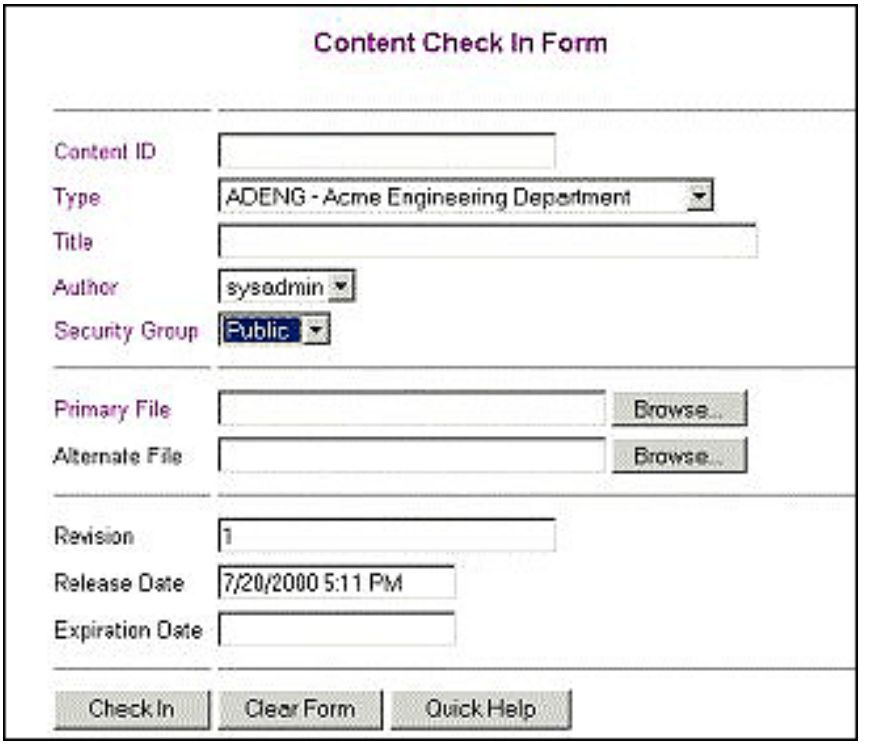
This section discusses the MultiCheckin component and analyzes the functionality of each file within the component. After implementing the MultiCheckin component you must log into the Content Server and click the Configuration Manager link.

After implementation, the Configuration Manager screen displays a drop-down list. Also, other changes will be noticed after selecting a specific file type. For example, the difference between choosing the content type ADACCT versus the content type ADENG is compared:

Example Content Type ADACCT:



Example Content Type ADENG:



Component Description

One of the more popular customizations performed is to have metadata displayed or suppressed on the check in screen, depending upon the content type selected. This particular component focuses on metadata fields that appear on the Content Server by default.

Accordingly, you may have different content metadata and you will need to modify the environment file, multi_checkin_environment.cfg to achieve your desired results.

The MultiCheckin component contains the following files:

- MultiCheckinManifest.zip
- manifest.hda
- components/doc_man.htm
- components/multi_checkin.hda
- components/multi_checkin_environment.cfg
- components/multi_checkin_resource.htm
- components/multi_checkin_templates.hda
- readme.txt

MultiCheckinManifest.zip

The *MultiCheckinManifest.zip* is the zipped file containing all of the files that are part of the component. The Component Manager portion of the Content Admin Server adds the ability for a properly zipped component file to be automatically uploaded and installed with the Content Admin Server application.

manifest.hda

The *manifest.hda* file is at the heart of the Component Manager. This file is used by the Component Manager feature of the Content Admin Server to easily upload and enable custom components without complicated installation procedures, customized installation CDs or installation logic of third party products. The purpose of the manifest.hda is to move individual files that are located within a properly zipped component file into the correct Content Server directories. For example, image files will be moved into the <*home*>/ weblayout/images directory/. However, this does not update the database.

For a component to be installed, removed, or unpackaged, the user must have a properly formatted manifest.hda file. Although simple to create, if the file is improperly formatted, the Component Manager will not execute.

Typically, the manifest.hda file is encapsulated in the zip file along with all files to be installed. The only valid name for this file is *manifest.hda*. It must be on a top level of the zip file directory structure and must contain at least one result set using this format:

```
@ResultSet Manifest
2
entryType
location
@end
```

This must be on a top level of the zip file directory structure and must contain at least one ResultSet with *entryType* and *location* entries.

The *entryType* entry must be one of the following:

Entry	Description
common	Files to be placed in <weblayout>/common/
images	Files to be placed in <weblayout>/images/
help	Files to be placed in <weblayout>/images/
component	A component to be placed by default into <*home*>/custom/…

Entry	Description
classes	A class file to be placed in <*home*>/classes/ with certain restrictions
componentextra	A file associated with a component, such as a readme.txt file, or other documentation.

Note: There are certain restrictions on installing a new class file. This installer is not intended as a patch utility for the Content Server, therefore it will not allow placement of Java class files into the <*home*>/classes/intradoc/ directory, nor will it place single files onto the <*home*>/classes/ directory. Class files must first be packaged into directories and then can be placed into the <*home*>/classes directory.

The *location* entry indicates both the file's location in the zip file, and its installed location. For example, directing the manifest.hda file to the location c:/<*home*>/custom would not allow a component to be installed on a server where the Content Server resides in the d:/ directory. Accordingly, a relative path should be used. It is the responsibility of the individual creating the component to ensure that full path names are used as rarely as possible. This will help ensure that many different Content Server users can share the packaged component.

Example Manifest

This is an example of a manifest.hda file for a component:

```
@ResultSet Manifest
2
entryType
location
component
MyComponent/MyComponent.hda
componentExtra
MyComponent/readme.txt
images
MyComponent/
@end
```

This is an example of the accompanying .zip file structure:

```
manifest.hda
component/MyComponent/MyComponent.hda
component/MyComponent/my_component_std_page.htm
component/MyComponent/my_component_resources.htm
component/MyComponent/readme.txt
images/MyComponent/image1.jpg
images/MyComponent/image2.jpg
```

The example defines these actions:

- The component MyComponent.hda and all files referenced by that component are installed into the directory: <*home*>/custom/MyComponent/.
- The readme.txt will also be placed in this directory.
- The images in the folder MyComponent/ are installed into the directory <weblayout>/images/MyComponent/.
- An entryType of *common*, *help*, or *class* works in a similar fashion to *images*.

components/doc_man.htm

This file serves as one of the template files that will be implemented by the Content Server and is the template file referenced in the multi_checkin_template.hda. This will be implemented through the MergeRules set in the multi_checkin.hda file.

These are some of the contents of the components/doc_man.htm file:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
    <meta name="GENERATOR" content="Intra.doc! 4.0">
    <title>Content Management</title>
    <$include std_html_head_declarations$>
</head>
```

```
<$include body_def$>
<$include std_page_begin$>
<$include std_header$>

<table border="0" cellpadding="2" cellspacing="2" width="450">
    <tr>
      <td colspan=4 <$if not
isNavOnSideBar$>align="center"<$endif$>>
        <h3 class=pageTitle>Content Management</h3>
      </td>
    </tr>
      <td width="30"> </td>
      <td>
        <a href="<$HttpCgiPath$>?IdcService=CHECKIN_NEW_FORM">
        <img src="<$HttpImagesRoot$><$docman_checkinnew_image$>"
        align="middle" border="0" <$if conmgr_btn_size$>
        width=<$conmgr_btn_size$>
height=<$conmgr_btn_size$><$endif$>></a>
      </td>
      <td>
        <a href="<$HttpCgiPath$>?IdcService=CHECKIN_NEW_FORM"
          class=largeTableEntry>New Check In</a>
      </td>

      <$include content_type_checkin_form_table_cell$>

    </tr>
```

components/multi_checkin_resource.htm

The components/multi_checkin_resource.htm file is a template file that contains the code that allows the user to choose the content type at check in time. The type of content chosen determines the metadata fields that will be displayed on the page.

This is an example of the script from the multi_checkin_resource.htm file:

```
<html>

<head>

<title>Resources for multi checkin component</title>

</head>

<body>

<br>

<p align=center><strong>MultiCheckin resources</strong></p>

<br>


<!-- altered to upload the document type environment data any
time the document types are shown on a page-->

<@dynamichtml std_document_type_field@>

<$include super.std_document_type_field$>

<$hiddenFields = getValue("#active", #active.dDocType &
"_hide")$>

<$checkinReadOnly = getValue("#active", #active.dDocType &
"_checkin_readOnly")$>

<$updateReadOnly = getValue("#active", #active.dDocType &
"_update_readOnly")$>

<@end@>


<!-- set the hidden and read only flags, and the default value --
>

<@dynamichtml compute_std_field_overrides@>

<$include super.compute_std_field_overrides$>
```

```
<$if strIndexOf(#active.hiddenFields, #active.fieldName) >= 0$>
<$isFieldHidden=1$> <$endif$>

<$if #active.isCheckin and strIndexOf(#active.checkinReadOnly,
#active.fieldName) >= 0$> <$isFieldInfoOnly=1$> <$endif$>

<$if #active.isUpdate and strIndexOf(#active.updateReadOnly,
#active.fieldName) >= 0$> <$isFieldInfoOnly=1$> <$endif$>

<$dynamicFieldValue = getValue("#active", #active.dDocType & "_"
& fieldName)$>

<$if dynamicFieldValue and isCheckin$> <$fieldValue =
dynamicFieldValue$> <$endif$>

<@end@>


<!-- this form will allow the user to obtain a check in page for
content

     with the specified type -->

<@dynamichtml content_type_checkin_form_table_cell@>

<form name=checkinNewGoForm method=get action="<$HttpCgiPath$>">

<td colspan=2>

    <input type=hidden name=IdcService value="CHECKIN_NEW_FORM">

    <select name=dDocType>

    <$docTypesList = #active.MultiCheckinTypes & ","$>

    <$index = strIndexOf(docTypesList, ",")$>

    <$loopwhile not strEquals(index, "-1")$>

    <$currentDocType = strSubstring(docTypesList, 0, index)$>

    <$docTypesList = strSubstring(docTypesList, index + 1)$>

    <$index = strIndexOf(docTypesList, ",")$>

      <option value="<$currentDocType$>" <$if
strEquals(#active.dDocType,
currentDocType)$>SELECTED<$endif$>><$currentDocType$>

    <$endloop$>

    </select>

    <input type=submit value=" GO ">

</td>
```

```
</form>
<@end@>


<!-- this include is overridden to enable the checkin pull-down
menu on the  side navigation bar, along with the "Check in new"
link.  Look for the    tag "MultiCheckin component changes" below
-->
```

…

components/multi_checkin.hda

The components/multi_checkin.hda file is the file that references your components. The purpose of a file of this type is to direct the Content Server to any custom defined resources.

This is the contents of the multi_checkin.hda file:

```
@Properties LocalData
ComponentName=MultiCheckin
@end


@ResultSet ResourceDefinition
4
type
filename
tables
loadOrder
resource
multi_checkin_resource.htm
null
1
template
multi_checkin_templates.hda
null
1
```

```
environment
multi_checkin_environment.cfg
Null
1
@end

@ResultSet MergeRules
3
fromTable
toTable
column
MultiCheckinTemplates
IntradocTemplates
name
@end
```

components/multi_checkin_environment.cfg

This file is used to display, hide, or manipulate metadata for the new content checkin pages of the Content Server. This script determines how the metadata fields for each content type selected is presented. This type of metadata manipulation can be performed for any of the content-centered templates (checkin_new.htm, checkin_sel.htm, doc_info.htm, update_docinfo.htm, and std_query.htm) using different parameters.

All configurations of this nature can be handled in an environment-type resource file. Each content type has a list of hidden fields, read-only fields and default checking values for any of the fields. Changes require you to restart the Content Server.

This is a portion of the script from the multi_checkin_environment.cfg file:

```
UseMultiCheckinOnSidebar=true
MultiCheckinTypes=ADACCT,ADCORP,ADENG,ADHR

ADACCT_hide=
ADACCT_checkin_readOnly=
ADACCT_update_readOnly=
ADACCT_xComments=This is the default comment for an ADACCT field.

ADCORP_hide=
ADCORP_checkin_readOnly=
ADCORP_update_readOnly=xComments
ADCORP_xComments=This is the default comment for an ADCORP field,
which cannot be changed on update.

ADENG_hide=xComments
ADENG_checkin_readOnly=
ADENG_update_readOnly=

ADHR_hide=
ADHR_checkin_readOnly=xComments
ADHR_update_readOnly=xComments
ADHR_xComments=This is the default, unchangable comment for an
ADHR field.
```

components/multi_checkin_templates.hda

To implement a template change, a MergeRule must be posted in the multicheckin.hda file. This will be in the form of merging from the ResultSet MultiCheckinTemplates into the default ResultSet IntradocTemplates using the column *name*. The name will refer to the template page name entry, DOC_MANAGEMENT_LINKS.

This is the script from the multi_checkin_templates.hda file:

```
@ResultSet MultiCheckinTemplates
5
name
class
formtype
filename
description
DOC_MANAGEMENT_LINKS
DocManagement
DocManagementLinks
doc_man.htm
Page containing links to various document management functions
@end
```

readme.txt

The readme.txt file documents the purpose of this component and directions for installing it both manually and with the Component Wizard and Component Manager tools.

Chapter

12

Understanding Workflows and Workflow Branching

Overview

Workflows are useful in the process of reviewing and approving content before it is released and published to the website. They specify how content is routed and who needs to review and approve it.

Workflows are defined and managed using Workflow Admin, which is one of the tools accessed from the Administration page. Only persons with administrator or sub-administrator privileges can create workflows. Defined workflows can be turned on and off. This means that workflows can be temporarily disabled, if required.

Note: The Content Management page contain a link called Active Workflows, which displays all workflows that are currently enabled.

Note: The User Profile page contains a link called Workflows in Queue for *[user]*, which displays a list of content items that the user needs to review.

A branching workflow allows a content item, or revision, to move from workflow step to another workflow step based on a set of criteria and evaluated Idoc Script. This can be used to allow revisions to share a common workflow entry point, but then diverge depending on who the original author is, who is currently working on the revision, and other revision metadata.

A workflow branch is initiated through the occurrence of an event and the evaluation of Idoc Script. Idoc Script has been enhanced with some very particular workflow functions. These functions allow the designer of the workflow to maintain extra revision state information and perform activities such as extra notifications. As a consequence, the system now maintains a state file for each workflow revision.

Note: Refer to the *Custom Scripting Reference Guid*e for information on Workflow Script Functions, Workflow Step Event Scripts, and Workflow Script Variables.

Workflow Types

There are three types of workflows: basic, criteria, and sub-workflow. A workflow becomes active in a system once is it enabled. All workflows are bound to a security group. This means that any content item that belongs to a workflow must be in that workflow security group on entry.

Basic Workflows

Basic workflows are workflows in which content is specifically assigned to the workflow. This type of workflow requires someone to initiate the process. The administrator or a sub-administrator selects one or more specific files for entry into the workflow (using Content IDs), and defines the workflow steps and the reviewers for each step.

Basic workflows consist of at least one named content item and an initial contribution step with defined users. Optionally, it consists of multiple reviewer and reviewer/contribution steps.

- When a basic workflow is enabled, initial revisions are created for the related content items and the contributors are notified that the workflow is active.
- When a basic workflow is disabled, the revisions for these content items are deleted from the system.

Criteria Workflows

Criteria workflows are workflows in which any content matching predefined criteria enters the workflow automatically upon check-in. The administrator or a sub-administrator selects the entry criteria for the workflow, consisting of a security group and a value for one content information (metadata) field, and defines the workflow steps and the reviewers for each step.

For example, if strategic reports must always be reviewed and approved by key individuals before being released, a criteria workflow could be set up for this content type and security group. If a strategic report is then checked into Stellent, a workflow is automatically initiated to start the approval process.

Criteria workflows consist of an auto-contribution step and at least one reviewer or reviewer/contributor step. A workflow enters a criteria workflow by satisfying a metadata criteria during check in.

- When a criteria workflow is enabled, it becomes available to the system during an initial check in. At that time, the metadata for the revision is evaluated against all active criteria workflow (for workflows in the same security group as the revision). A revision could match several criteria workflows, but it may only enter into one workflow.
- When a criteria workflow is disabled, all revisions in the workflow are moved out of the workflow state. Unlike the basic workflow, the revisions are not deleted.

Sub-Workflows

A sub-workflow is a workflow that does not have an initial contribution step. Sub-workflows are useful for splitting large, complex workflows into manageable pieces. A file can enter a sub-workflow only through a jump from a criteria workflow.

The Sub-workflow type is related to the criteria workflow. However, the sub-workflow does not have the initial contribution step. A revision can only enter a sub-workflow through a jump. A sub-workflow can become a criteria workflow by defining a criteria and vice-versa.

Workflow Steps

A workflow consists of one or more steps, and multiple users can be assigned to review the content at each step. There are four types of steps:

- A **contribution** step is the initial step of a basic workflow. Contributors are defined when the workflow is created.
- An **auto-contribution** step is the initial step of a criteria workflow. There are no predefined users involved in this step.
- In a **reviewer** step, the assigned users can only approve or reject the file. Editing is not allowed.
- In a **reviewer/contributor** step, users can edit the file, if necessary, and then approve or reject it.

If there is more than one user assigned to a step, it is possible to specify how many of them need to approve the content before it moves to the next step.

All persons involved in a workflow are notified about any actions they need to perform for each step. This is done entirely through e-mail. E-mail messages can also be sent to content authors and other users to inform them of the status of the workflow.

Each step has three events: entry, update and exit. Each event consists of a script that is evaluated at a well-defined time. The events have an effect on the workflow only if a script has been defined for it.

On entering a step, the entry script is evaluated. This event script consists of a standard default script plus potentially a user defined script. The default script computes the number of times this step has been entered and the last time the step has been entered.

The update event initiates:

- During a timed update cycle.
- Upon update of the revision's metadata.
- After an approval or check in.

The exit event script is evaluated when a revision has completed the step requirements.

Jumps

Jumps enable you to customize workflows for your system, your content, and your users. Jumps are created using Idoc Script, which is Stellent's proprietary scripting language.

Typical uses of jumps include:

- Specifying more than one metadata field as the criteria for entering a workflow.
- Taking action on content automatically after a certain amount of time has passed.
- Defining different paths for files to move through the same workflow depending on metadata and user criteria.

Note: Refer to "Workflow Step Event Scripts" in the *Custom Scripting Reference Guid*e for additional information.

Tokens

A token is a piece of Idoc Script that defines variable users in a workflow. Tokens can be used for any of the following:

- Specify a variable user, such as the original author or the author's supervisor.
- Include users and aliases in workflow jumps.
- Define users through conditional statements.

Note: Refer to "Workflow Script Functions" in the *Custom Scripting Reference Guide* for additional information.

Workflow and Script Templates

Workflow Templates

Workflow templates are a quick way to reuse workflows that you have already created. Each workflow template is an outline for a basic workflow, criteria workflow, or sub-workflow that is stored in the Workflow Admin tool. A workflow template is not tied to a security group, and it cannot include jumps.

For example, if the first and last step of several workflows need to be the same, you could save these steps as a workflow template, and then use the template as the starting point for creating the individual workflows.

Script Templates

Script templates are a quick way to reuse jumps that you have already created. Each script template is a piece of Idoc Script stored in the Workflow Admin tool.

For example, if you have several workflow steps that require approval within one week, you could save the jump script for this as a template, and then reuse it.

Workflow Branching

A step event may move a revision from one workflow step to another workflow step. Depending on the type of workflow, a revision may jump backwards and forwards in the same workflow or into the step of a completely different workflow. The system keeps careful track of the history of where a revision has been, what jumps have been performed, entry counts and entry times and any custom information that the designer of the workflow has chosen to maintain.

The flow consists of these basic steps:

- Evaluating the script.
- Actions performed on the Last Step.
- Actions performed on Restart.
- Actions performed on Exit.
- Actions performed on Error.
- Actions Performed on Reject.
- Executing the script.

Evaluating the Script

1. Execute the update script.
2. Determine if the step has been completed.
3. If the step is finished, evaluate the exit script.
4. If the exit script moves us to another step:
    a. Inform users of step that revision has entered step.
    b. Evaluate the entry script for this step.
    c. If this takes us to a new step, keep track of where we have been and repeat.
    d. If this specifies an exit, determine the exit step and repeat.
    e. Determine if the step is finished. This could be a notification step, which is automatically finished, or it could be one requiring one or more reviewers.
    f. If the step is finished, go to *Actions Performed On Restart*.

Actions Performed on the Last Step

1. Determine if the step has been completed.
2. Unwind the stack of parents looking for jump steps.
3. For each jump step, determine if there is a return point. Stop once you have found a return point.
4. If there is a return point, go to the return point and evaluate the entry script and perform the actions in *Evaluating the Script, step 4*.
5. If there is no return point, exit the workflow.

Actions Performed on Restart

1. After the execution of a script, determine if this is a restart.
2. If this is a restart of a step, evaluate the entry script and perform the actions in *Evaluating the Script, step 4*.

Actions Performed on Exit

If the script specifies that the revision is to exit the workflow, go to *Actions Performed on the Last Step, step 2*.

Actions Performed on Error

If for any reason an error occurs in evaluating the script, ignore the script. However, if the script evaluated correctly and for example, the target step is invalid, fall into the exit scenario. See *Executing the Script*.

Note: A jump can specify its return point as a side effect. It however is not required to define a return point. Consequently, on error you may go back to a return point defined by another jump and not the jump you originally came from. If there are no return points, exit the workflow.

Actions Performed on Reject

Search through the stack of parents for a step that allows contribution. The first contribution step that is found is the target for the reject.

Executing the Script

1. Evaluate the script.
2. On error, go to the closest return point.
3. Evaluate the entry script of the return point and go to *Evaluating the Script, step 4*.
4. On error, repeat the previous step until there are no more return points and exit the workflow.

Note: Be aware of loops. If we have already entered a step once before, then skip the entry script execution. The stack has no repeats. If the revision is moved to a step that has already been referred to in the stack, unwind the stack to the referenced step.

Workflow Information Storage

Database Tables

These tables have existed since version 4.0 and only a few columns have been added for maintenance of the workflow design:

WorkflowAliases	WorkflowHistory
WorkflowCriteria	Workflows
WorkflowDocAttributes	WorkflowStates
WorkflowDocuments	WorkflowSteps

Associated Files

These associated files store workflow information:

File	Description
Workflow Design	Each workflow has design that maintains the event script information. Located in the ~/data/workflow/design directory.
Script Design	The system allows for the creation of event scripts outside of the context of a particular workflow. The script templates can be used as starter examples and allow for ease of sharing of complicated scripts. Maintained in the ~/data/workflow/script directory.
Revision State Information	These files are also known as the revision's companion file. This file maintains the current state of the revision in a workflow. Located in ~/data/workflow/states directory.
Saved companion files	Saved companion files are maintained in the ~/data/workflow/saved directory. This directory maintains the latest state information for a revision that has completed its workflow.
Tokens	The list and definition of tokens is located in ~/data/workflow/tokens/tokens.hda.

Workflow Rules and Error Handling

- A basic workflow may not jump to another workflow. The jump may only take a revision to steps within the workflow.
- A criteria workflow may only jump to a criteria or sub workflow belonging to the same security group.
- A jump to a step in an inactive workflow is an error. However, when initially defining a target step for a workflow step, the step is not validated. The target step is validated when it is actually used.
- If a jump takes you to an inactive workflow, the jump will be treated as an error and the revision falls into the exit scenario.
- An event script that has been badly defined and causes an error in execution is treated as if it had never executed. However, if this is an entry script then the default entry script, which keeps track of entry time and number of times entered, is still evaluated.
- When jumping to a step that is already in the parent list, the parent list is unwound. For example, if the progression has been step_1, step_2, step_3 and the revision is jumped to step_2, the parent list becomes step_1, step_2 not step_1, step_2, step_3, step_2. This is an attempt to avoid recursion.
- The system does its best to avoid fast loops. These are loops that are executed within the workflow engine without user interaction. If a jump takes you to a step that has already been visited in the current cycle, the workflow ignores the script, thereby refusing to calculate the jump. For example, a user approves a revision in step_a. On evaluation of the update step, the revision is moved to step_b. The entry script for step_b is evaluated, it causes a jump to step_c. For step_c's entry script, the target step is step_b and now we are in a fast loop, since without user interaction or a break in the processing, we have returned to step_b. Consequently, the entry script for step_b is ignored. If it were not ignored, we would be in an infinite loop.
- Slow loops are allowed. For example, loops that happen due to user interaction or a break in workflow processing.
- All script evaluation occurs inside a database transaction. This means any serious errors or aborts that are encountered cause no change to either database or companion file. This also means that no Idoc Script function should take more than a negligible amount of time. Consequently, to trigger and outside process, an Idoc Script function should be written to execute in a separate thread.
- A reject causes the parent list to be unwound in search of a contribution step.

- An exit of a workflow takes the revision to the most recently specified return step. If none is defined, the revision exits the workflow process. The parent list is unwound accordingly.

Index

A

access level, 10-11
action types, 10-13
    CACHE_RESULT_TYPE = 5, 10-14
    CODE_TYPE = 3, 10-13
    EXECUTE_TYPE = 2, 10-13
    OPTION_TYPE = 4, 10-14
    QUERY_TYPE = 1, 10-13
ADD_USER service, 10-12
Administration link, 2-8
Alias *(database table)*, 10-3
AliasUse *(database table)*, 10-3
assembling the ADMIN_LINKS template page and returns the page, 2-9
AuthorAddress, 10-25
awkward geometry, 2-10

B

bin directory, 2-12
body definition, 5-12
BODY element, 5-12

C

CACHE_RESULT_TYPE = 5 *(action type)*, 10-14
change form methods, 2-16
CHECKIN_LIST, 8-4
CHECKIN_NEW_FORM, 8-4
CheckoutUserAddress, 10-25
class *(IntradocTemplates column)*, 8-4
CODE_TYPE = 3 *(action type)*, 10-13
column, 6-7
columns
    IntradocTemplates, 8-3
    MergeRules, 6-6
    SearchResultTemplates, 8-7
component
    columns, 7-2
        location, 7-2
        name, 7-2
    file structure, 2-14
    implementing, 7-2
    removing, 7-3
    structure, 7-1
        columns, 7-2
        implementing a component, 7-2
        removing a component, 7-3
component architecture, 2-3
    process, 3-5
        component definition file, 3-6
        components file, 3-5
        defining custom environment, 3-10
        defining custom queries, 3-10
        defining custom services, 3-13

modifying resources, 3-6
modifying standard templates, 3-7
component architecture and the Content Server, 2-7
server actions, 2-8
server behavior, 2-7
component definition
file, 3-6
HDA file, 3-15
component description, 11-4
doc_man.htm, 11-7
manifest.hda, 11-5
multi_checkin.hda, 11-11
multi_checkin_environment.cfg, 11-12
multi_checkin_resource.htm, 11-9
multi_checkin_templates.hda, 11-14
MultiCheckinManifest.zip, 11-4
readme.txt, 11-14
component file structure
consistent file structure, 2-14
component wizard, 1-3
components, 5-5
doc_man.htm, 11-7
file, 3-5
multi_checkin.hda, 11-11
multi_checkin_environment.cfg, 11-12
multi_checkin_resource.htm, 11-9
multi_checkin_templates.hda, 11-14
components.hda, 2-13
Config *(database table)*, 10-3
config directory, 2-13
component.hda, 2-13
config.cfg, 2-13
config.cfg, 2-13
configuration file, 7-4
defining a variable, 7-5
referencing a variable, 7-6
Configuration Variables Load, 3-3
consistent
file structure, 2-14
Content Server loading, 8-1
Content Server services, 2-8
use Administration link to..., 2-8
assemble the ADMIN_LINKS template page and return the page, 2-9
provide a login prompt if not currently logged in, 2-8
verify that the login has administrator privileges, 2-9
content types, multi-checkin, 9-2
CONTROL_BEGIN_TRAN = 4 *(function parameters)*, 10-14
CONTROL_COMMIT_TRAN = 8 *(function parameters)*, 10-15
CONTROL_IGNORE_ERROR = 1 *(function parameters)*, 10-14
CONTROL_MUST_EXIST = 2 *(function parameters)*, 10-14
CONTROL_MUST_NOT_EXIST = 16 *(function parameters)*, 10-15
Counters *(database table)*, 10-3
create customizations, 2-2
Creating Custom Conversion Engines, 1-1
custom
components load, 3-4
environment resources, defining, 3-10
queries, defining, 3-10
services, defining, 3-13
templates, defining, 8-10
Custom Scripting Reference Guide, 1-2
customizing
graphics, 2-10
awkward geometry, 2-10
lost data, 2-10

no addition/deletion, 2-10
options, 2-10
customizing graphics, 2-10
image format, 2-10
image referencing, 2-11
product functionality, 2-6
the interface, 2-5

D

data binder, 5-8
database tables, 10-3
Alias, 10-3
AliasUse, 10-3
Config, 10-3
Counters, 10-3
DocFormats, 10-3
DocMeta, 10-4
DocMetaDefinition, 10-4
DocTypes, 10-4
DocumentAccounts, 10-4
DocumentHistory, 10-4
Documents, 10-4
ExtensionFormatMap, 10-4
OptionsList, 10-4
ProblemReports, 10-4
ProjectDocuments, 10-4
RegisteredProjects, 10-5
Revisions, 10-5
RoleDefinition, 10-5
SecurityGroups, 10-5
Subscription, 10-5
UserSecurityAttributes, 10-5
Uses, 10-5
WorkflowAliases, 10-5
WorkflowCriteria, 10-5
WorkflowDocAttributes, 10-6
Workflows, 10-6
WorkflowStates, 10-6
WorkflowSteps, 10-6
dCheckoutUser, 10-25
dDocAuthor, 10-25
dDocTitle, 10-24
defining
a variable, 7-5
custom environment Resources, 3-10
custom queries, 3-10
HDA file, 3-12
HTM format, 3-11
custom services, 3-13
component definition HDA file, 3-15
MyServices, 3-13
custom templates, 8-10
description
IntradocTemplates column, 8-5
SearchResultTemplates column, 8-9
development
instance, 2-14
recommendations, 2-14
change form methods, 2-16
component file structure, 2-14
development instance, 2-14
naming conventions, 2-15
read server errors, 2-17
Development Kit, 1-1
component wizard, 1-3
SDK documentation, 1-1
displaying the multi-checkin menu, 9-2
DOC_INFO service, example, 10-16
DOC_INFO template, 10-17
doc_man.htm, 11-7
DocFormats *(database table)*, 10-3
DocMeta *(database table)*, 10-4
DocMetaDefinition *(database table)*, 10-4
DocService *(service type)*, 10-10

DocTypes *(database table)*, 10-4
document class *(template pages)*, 8-4
DocumentAccounts *(database table)*, 10-4
DocumentHistory *(database table)*, 10-4
Documents *(database table)*, 10-4
dStatus, 10-24
dynamic content
    including in a template, 5-15
dynamic content resources, 5-11
    body definition, 5-12
    page begin, 5-12
    page end, 5-15
    structure, 5-11
dynamic include, 4-3
dynamic page retrieval, 2-8
dynamic resource table, column type, 6-3
dynamic table, 4-4

E

environment, 4-8
    column type, 6-3
environment file, multi-checkin, 9-2
    content types, 9-2
    displaying the menu, 9-2
error message, 10-12
examine source code, 2-2
    create customizations, 2-2
    reinstall, 2-2
EXECUTE_TYPE = 2 *(action type)*, 10-13
ExtensionFormatMap *(database table)*, 10-4

F

filename
    IntradocTemplates column, 8-5
    ResourceDefiniton column, 6-3
    SearchResultTemplates column, 8-8
files used for customization, 2-12
    bin directory, 2-12
    config directory, 2-13
    shared/config directory, 2-13
    weblayout directory, 2-13
FileService *(service type)*, 10-10
flexdata,*(SearchResultTemplates column)*, 8-8
formtype
    IntradocTemplates column, 8-5
    SearchResultTemplates column, 8-7
fromTable, 6-7
function name, 10-14
function parameters, 10-14
    CONTROL_BEGIN_TRAN = 4, 10-14
    CONTROL_COMMIT_TRAN = 8, 10-15
    CONTROL_IGNORE_ERROR = 1, 10-14
    CONTROL_MUST_EXIST = 2, 10-14
    CONTROL_MUST_NOT_EXIST = 16, 10-15

H

HDA file
    structure, 5-2
    type, 5-2
        data binder, 5-8
        HDA section type-@ResultSet, 5-4
        HDA section-type@Properties, 5-3
        purpose, 5-2
        section types, 5-2
        structure, 5-2
HDA section type
    @Properties, 5-3
        structure, 5-3
    @ResultSet, 5-4
        sample ResultSet, 5-7
        structure, 5-6
HTM file type, 5-9

dynamic content resources, 5-11
including dynamic content in a template, 5-15
reports, 5-9
template, 5-9
HTM tables, 5-9
structure, 5-10
HTML editor, 2-4
HTML include, 4-2
HTML/CSS, 2-3

I

IdcCommand Reference Guide, 1-2
Idoc Script, 2-3
image
format, 2-10
referencing, 2-11
implementing a component, 7-2
including dynamic content in a template, 5-15
internal initialization occurs, 3-3
Configuration Variables Load, 3-3
IntradocReports, 5-5
IntradocTemplates, 5-5, 8-2
columns, 8-3
class, 8-4
description, 8-5
filename, 8-5
formtype, 8-5
name, 8-4
STANDARD_QUERY_PAGE, 8-4
table, 4-7

J

Java programming, 2-4
JavaScript, 2-4
debugger, 2-4

L

loading, Content Server, 8-1
loadOrder, ResourceDefinition column, 6-4
location *(component column)*, 7-2
lost data, 2-10

M

manifest.hda, 11-5
example, 11-6
menu, displaying multi-checkin, 9-2
merge rules, 3-5
MergeRules, 6-6
columns, 6-6
example, 6-7
merging newscores into scores, 6-10
merging scores into newscores, 6-11
newscores, 6-9
scores, 6-8
MergeRules columns, 6-6
column, 6-7
fromTable, 6-7
toTable, 6-7
merging
newscores into scores, 6-10
scores into newscores, 6-11
MetaService *(service type)*, 10-10
modify source code, 2-2
create customizations, 2-2
reinstall, 2-2
upgrade, 2-2
modifying
resources, 3-6
standard templates, 3-7
multi_checkin.hda, 11-11
multi_checkin_environment.cfg, 11-12
multi_checkin_resource.htm, 11-9
multi_checkin_templates.hda, 11-14

multi-checkin
content types, 9-2
environment file, 9-2
multi-checkin content types, 9-2
multi-checkin menu display, 9-2
menu display, 9-2
MultiCheckinManifest.zip, 11-4
multiple browsers, 2-4
MyServices, 3-13

N

name
component column, 7-2
IntradocTemplates column, 8-4
query definition table column, 10-2
SearchResultTemplates column, 8-7
service resources, 10-9
naming conventions, 2-15
observe case, 2-16
use appropriate file name extensions, 2-16
use consistent naming conventions, 2-16
use unique file names, 2-16
newscores, 6-9
merging into scores, 6-10
no addition/deletion, 2-10

O

observe case, 2-16
OPTION_TYPE = 4 *(action type)*, 10-14
OptionsList *(database table)*, 10-4
outfilename *(SearchResultTemplates column)*, 8-8

P

page assembly, 3-1
standard page beginning, 3-2
standard page ending, 3-2
standard page header, 3-2
page begin, 5-12
std_page_begin resource, 5-12
page end, 5-15
std_page_end, 5-15
page retrieval, 2-8
dynamic page retrieval, 2-8
static page retrieval, 2-8
PageHandlerService *(service type)*, 10-10
parameters
query definition table column, 10-3
PING_SERVER service, 10-12
ProblemReports *(database table)*, 10-4
Programmer's Reference Guide, 1-2
ProjectDocuments *(database table)*, 10-4
providing a login prompt if not currently logged in, 2-8

Q

QisSubscribed, 10-26
query, 4-5
column type, 6-3
query definition table columns, 10-2
name, 10-2
parameters, 10-3
queryStr, 10-3
query definition tables, 10-2
query resource, 10-1
database tables, 10-3
example, 10-7
query definition
table columns, 10-2
tables, 10-2
QUERY_TYPE = 1 *(action type)*, 10-13
queryStr *(query definition table column)*, 10-3

R

read server errors, 2-17
readme.txt, 11-14
referencing a variable, 7-6
RegisteredProjects *(database table)*, 10-5
reinstall, 2-2
removing a component, 7-3
reports *(shared/config directory)*, 2-13
reports load, 3-4
required skills, 2-3
    component architecture, 2-3
    HTML/CSS, 2-3
    Idoc Script, 2-3, 2-4
    Java programming, 2-4
    JavaScript, 2-4
required tools, 2-3
    HTML editor, 2-4
    JavaScript debugger, 2-4
    multiple browsers, 2-4
    Software development kit, 2-4
    text editor, 2-4
ResourceDefinition, 5-5, 6-2, 6-3
    columns, 6-3
    example, 6-4
    loadOrder, 6-4
    tables, 6-4
ResourceDefinition columns, 6-3
    dynamic resource table, 6-3
    environment, 6-3
    filename, 6-3
    query, 6-3
    services, 6-3
    static resource table, 6-3
    template, 6-3
    type, 6-3
resources, 5-9
resources *(shared/config directory)*, 2-13
ResultSet name
    components, 5-5
    Intradoc Templates, 5-5
    IntradocReports, 5-5
    ResouceDefinition, 5-5
    SearchResultsTemplates, 5-5
ResultSet, sample, 5-7
retrieve pages, 2-7
Revisions *(database table)*, 10-5
RoleDefinition *(database table)*, 10-5
run
    a search engine service, 2-7
    a system server service, 2-7

S

sample ResultSet, 5-7
scores, 6-8
    merging into newscores, 6-11
SDK *See Software development kit*, 2-4
SDK documentation, 1-1
    Creating Custom Conversion Engines, 1-1
    Custom Scripting Reference Guide, 1-2
    IdcCommand Reference Guide, 1-2
    Programmer's Reference Guide, 1-2
searching services, 2-9
SearchResultsTemplates, 5-5
SearchResultTemplates, 8-6
    columns, 8-7
        description, 8-9
        filename, 8-8
        flexdata, 8-8
        formtype, 8-7
        name, 8-7
        outfilename, 8-8
    table, 4-7
section types *(HDA file structure)*, 5-2
SecurityGroups *(database table)*, 10-5

server actions, 2-8
Content Server services, 2-8
page retrieval, 2-8
searching services, 2-9
server behavior, 2-7
server information flow, 2-7
web browser requests, 2-7
server information flow, 2-7
server start up actions, 3-3
custom components load, 3-4
internal initialization occurs, 3-3
standard resources, templates, and reports, 3-4
service, 4-6
attributes, 10-10
type, 10-10
name, 10-9
resource, 10-8
resource structure, 10-8
service *(service type)*, 10-10
service actions, 10-13
function name, 10-14
function parameters, 10-14
CONTROL_BEGIN_TRAN = 4, 10-14
CONTROL_COMMIT_TRAN = 8, 10-15
CONTROL_IGNORE_ERROR = 1, 10-14
CONTROL_MUST_EXIST = 2, 10-14
CONTROL_MUST_NOT_EXIST = 16, 10-15
types, 10-13
CACHE_RESULT_TYPE = 5, 10-14
CODE_TYPE = 3, 10-13
EXECUTE_TYPE = 2, 10-13
OPTION_TYPE = 4, 10-14
QUERY_TYPE = 1, 10-13
service resources
access level, 10-11
actions, 10-13
DOC_INFO service example, 10-16
error message, 10-12
example, 10-16
service attributes, 10-10
service name, 10-9
subjects notified, 10-12
sub-service, 10-11
template page, 10-11
service type, 10-10
DocService, 10-10
FileService, 10-10
MetaService, 10-10
PageHandlerService, 10-10
service, 10-10
UserService, 10-10
WorkflowService, 10-10
services
column type, 6-3
shared/config directory, 2-13
reports, 2-13
resources, 2-13
templates, 2-13
Software development kit, 2-4
SQL, 2-4
standard page
beginning, 3-2
ending, 3-2
header, 3-2
standard resources, templates, and reports load, 3-4
STANDARD_QUERY_PAGE *(IntradocTemplate entry)*, 8-4

static page retrieval, 2-8
static resource table, column type, 6-3
std_page_begin, 5-12
std_page_end, 5-15
structure *See component structure*, 7-1
subjects notified, 10-12
Subscription *(database table)*, 10-5
sub-service, 10-11

T

tables *(ResourceDefinition column)*, 6-4
template, 4-7
    column type, 6-3
    page, 10-11
templates, 3-4
    defining (custom), 8-10
    file, 8-2
    IntradocTemplates, 8-2
    SearchResultTemplates, 8-6
    VerityTemplates, 8-6
templates *(shared/config directory)*, 2-13
templates and reports, 5-9
    HTM tables, 5-9
    resources, 5-9
templates.hda file
    IntradocTemplates table, 4-7
    SearchResultTemplates table, 4-7
    VerityTemplates table, 4-7
text editor, 2-4
toTable, 6-7
type, ResourceDefinition coluumn, 6-3

U

Unable to retrieve information for {dDocName}, 10-24
Unable to retrieve revision history for {dDocName}, 10-26
Understanding Component Architecture, 2-1, 2-2
    component architecture and the Content Server, 2-7
    customizing options, 2-10
    customizing product functionality, 2-6
    customizing the interface, 2-5
    development recommendations, 2-14
    files used for customization, 2-12
    required skills, 2-3
    required tools, 2-3
Understanding Component Assembly, 3-1
    component architecture process, 3-5
    merge rules, 3-5
    page assembly, 3-1
    server start up actions, 3-3
Understanding Content-Centered Template Metadata, 9-1
    multi-checkin environment file, 9-2
Understanding HDA and HTM File Types, 5-1
    HDA file type, 5-2
    HTM file type, 5-9
Understanding Query and Service Resources, 10-1
    query resource, 10-1
    service resource, 10-8
Understanding Resource Types, 4-1
    dynamic table, 4-4
    environment, 4-8
    HTML include, 4-2
    query, 4-5
    service, 4-6
    template, 4-7
Understanding Templates, 8-1
    Content Server loading, 8-1
    defining custom templates, 8-10
    IntradocTemplates, 8-2

SearchResultTemplates, 8-6
template file, 8-2
VerityTemplates, 8-6
Understanding the Component Definition File, 6-1
MergeRules, 6-6
ResourceDefinition, 6-2
Understanding the Components HDA File, 7-1
component structure, 7-1
configuration file, 7-4
Understanding the MultiCheckin Component, 11-1
component description, 11-4
Understanding Workflow Brnching, 12-1
upgrade, 2-2
use appropriate file name extensions, 2-16
use consistent naming conventions, 2-16
use unique file names, 2-16
Users *(database table)*, 10-5
UserSecurityAttributes *(database table)*, 10-5
UserService *(service type)*, 10-10

V

variable
defining, 7-5
referencing, 7-6
verifying that the login has administrator privileges, 2-9
VerityTemplates, 8-6
VerityTemplates table, 4-7

W

web browser requests, 2-7
retrieve pages, 2-7
run a search engine, 2-7
run a system server service, 2-7
weblayout directory, 2-13
WorkflowAliases *(database table)*, 10-5
WorkflowCriteria *(database table)*, 10-5
WorkflowDocAttributes *(database table)*, 10-6
WorkflowDocuments, 10-6
Workflows *(database table)*, 10-6
WorkflowService *(service type)*, 10-10
WorkflowStates *(database table)*, 10-6
WorkflowSteps *(database table)*, 10-6