Compass Server 3.0
Developer's Guide

Recycled and Recyclable Paper

Netscape Compass Server 3.0

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

Contents

About This Book

This document is a developer's guide for Compass Server 3.0, a cataloging and indexing server. This document is intended for developers who are customizing the Compass Server 3.0 programmatically.

This document does not attempt to explain how to administer the Compass Server interactively. For details on Compass Server administration, see the *Compass Server Administrator's Guide*, or press the **Manuals** button in the Compass Server Administrator environment.

Users use the Compass Server end user page to send queries to the Compass Server database. By editing template files, you can modify the appearance of the end user page, and also access it using query string URLS, as discussed in Part 1.

The standard procedure for interacting with the Compass Server robot is to use the **Robot** page in the Compass Server Administration Interface. However, if you want to modify the robot behavior in a way that is not accommodated by the interface, you can do it programmatically by modifying robot configuration files. If you want to extend the robot behavior even further, you can define your own robot application functions to be used in the configuration files. Part 2 discusses how to programmatically modify and extend robot behavior.

The standard procedure for accessing information in the Compass Server database is to use the Compass Server End User page, as discussed in the *Compass Server Administration Guide*. However, hooks are provided for you to access the Compass Server database directly, without using the Compass Server end user page. You can access the database using either C or Java. Netscape provides the Compass Server RDM API for accessing the database using C (discussed in Part 3), and the Compass Server Java SDK for accessing it from Java applets (discussed in Part 4).

This document has the following parts:

Part 1 - Customizing the End User Search Page

How to customize the Compass Server End User page, which is the page that users see when they use the Compass Server to search or browse. To customize the End User page, use HTML and JavaScript to modify the files that define the various sections of the page.

Part 2 - Customizing Robot Behavior

How to programmatically customize the way the Compass Server robot decides which resources to index, and how it indexes them. You do this by editing configuration files in a text editor, and by defining Robot Application Functions in C or C++.

Part 3 - Using C to Send Requests to the Compass Server

How to use the RDM C API to work with SOIF objects and to create resource description messages (RDMs). The resource descriptions in the database are stored in SOIF format. To send queries to the database, you need to construct RDMs, which also use SOIF format.

Part 4 - Using Java to Access the Compass Server Database

How to use Java applets to submit queries and to add new entries to the Compass Server database.

What You Should Already Know

This book assumes you have this basic background:

- A general understanding of the purpose and use of the Compass Server. For more information about the Compass Server, see the *"Compass Server 3.0 Administrator's Guide"* which is available through the Manuals button in the Compass Server Administrator interface.

- For modifying the end user page, you need good working knowledge of HyperText Markup Language (HTML). Experience with forms and the Common Gateway Interface (CGI) is useful. Experience with JavaScript is also useful.
- For modifying robot behavior using configuration files, you do not need to know any languages. However, for creating new robot application functions to extend the range of robot functionality, you need to know how to use and compile C.
- For using the RDM SDK to work with SOIF objects and create resource description messages, you need to know how to use the C language.
- For using Java to interact with the Compass Server database, you need to know how to use and compile Java.

Where to Find Information About HTML, JavaScript, and Java

To customize the Compass Server End User page, you need to use HTML and/or JavaScript. Here's where you can find information about these two languages:

The online HTML references include:

- *HTML Tag Reference* provides a reference to all the HTML tags supported in Netscape Communicator 4.0 and earlier.
- *Dynamic HTML in Netscape Communicator* provides information about dynamic HTML, which includes style sheets, positioned content, and downloadable fonts.

For information about JavaScript, see the *JavaScript Guide* or the *JavaScript Reference*.

A good place to start learning Java is Sun'*Java Tutorial* at:

`http://java.sun.com/docs/books/tutorial/index.html`

Document Conventions

This book uses the following font conventions:

- The `monospace font` is used for sample code and code listings, API and language elements (such as function names and class names), filenames, pathnames, directory names, HTML tags, and any text that must be typed on the screen.
- *`Monospace italic font`* is used for placeholders embedded in code.
- *Italic type* is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.
- **Bold** font is used for parameters in robot configuration files and for robot application function names.

1

Customizing the End User Search Page

This part of the document discusses how to customize the Compass Server End User page.

Compass Server 3.0 enables the creation of a searchable database of documents and other networked resources on one or more web sites. Users can browse through categories or perform searches on the database by using the Compass Server End User page. Usually users are not aware that they are searching a database; to all intents and purposes they are searching a web site. The following figure shows a sample Compass Server End User page.

Figure P1.1 The End User page

Each time the Compass Server End User page updates, its content changes depending on whether the user is browsing or searching. In general, if the user is browsing a category, the page shows the subcategories in the browsed category. If the user submitted a search query, the page shows the results of the search.

The exact appearance of the End User page is ultimately determined by the currently selected *user interface template (UI template)*.

The Compass Server has a variety of UI templates that you can use to determine the appearance of the Compass Server End User page. For example you could pick a template that presents a page that only allows the user to do simple searches, or you could pick a template that allows the user to browse through categories in addition to doing simple or advanced searches.

As well as using the pre-defined templates, you can create your own templates. The best way to do this is to copy existing templates and modify them to suit your needs. By modifying the templates, you can customize the Compass Server End User page to your exact requirements using HTML or JavaScript. You can make modifications as simple as changing the color of the page's background or as complex as using a completely different format for displaying search results.

Each template is made up of a *configuration* file and a set of *pattern* files. Each configuration file lists the pattern files that are assembled together to make up the Compass Server End User page. Each pattern file defines the content of a piece, or component, of the page. To customize the page, you need to edit the configuration file and the pattern files.

This part contains the following chapters:

Chapter 1. Composition of the End User Page

Overview of the sections and modes in the end user page.

Chapter 2. Templates for the End User Page

Overview of the templates used in the end user page.

Chapter 3. Configuration Files for UI Templates

Discussion of the configuration and pattern files used by the templates.

Chapter 4. Pattern Files for UI Templates

All about pattern files.

Chapter 5. Opening the End User Page With Query String URLs

How to invoke the end user page using query string URLs.

Chapter 6. Tutorial: Customizing the End User Page

A step-by-step tutorial to customizing the appearance of the end user page.

Chapter 7. Error Message Files

How to modify the error messages that appear when a request to connect to the Compass Server fails.

Chapter

1

Composition of the End User Page

Compass Server 3.0 enables the creation of a searchable database of documents and other networked resources on one or more web sites. Users can browse through categories or perform searches on the database by using the Compass Server End User page. Usually users are not aware that they are searching a database; to all intents and purposes they are searching a web site. The following figure shows a sample Compass Server End User page.

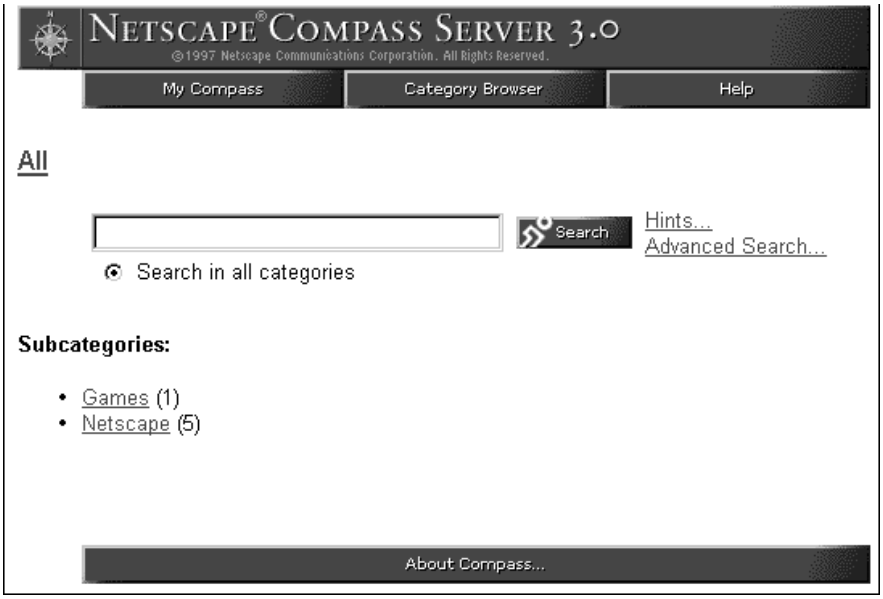

Figure 1.1 The End User page

The Compass Server End User page has two basic modes of appearance: browse mode and search-results mode.

- Browse Mode

  The Compass Server End User page uses browse mode when an end user first visits the page, and also whenever the end user clicks a category to browse through its subcategories and documents.

- Search-results Mode

  The Compass Server End User page uses the search-results mode to display the results after an end user submits a search.

The mode is determined by how the user reached the page, not what the user can do in the page. (This is a very important, if somewhat subtle, distinction.)

The browse mode and search-results mode often share aspects of their appearance. Often pages using either browse or search-results mode allow the user to both search and browse.

Each mode in turn is made up of several component parts. Each mode has a top and bottom component, which determine the appearance of the top and bottom of the pages. Each mode has additional components that provide the content for the rest of the sections in the page.

Components in Browse Mode Pages

The top (first) component in a browse mode page is the browse-top component, and the bottom (last) component is the browse-bottom component.

As mentioned already, the Compass Server End User page uses browse mode whenever the user clicks a category. A browse mode page usually lists the subcategories in the selected category, and also any documents in that category.

The list of subcategories has three components: the top of the list (such as a heading or introduction), the entries in the list, and the bottom of the list (such as a closing remark or a link to the next page.) These components are known as the category-browse-top, category-browse-hit, and category-browse-bottom components.

Similarly the list of documents in the category has three components: the top of the list, the entries in the list, and the bottom of the list. These are known as the document-browse-top, document-browse-hit, and document-browse-bottom components.

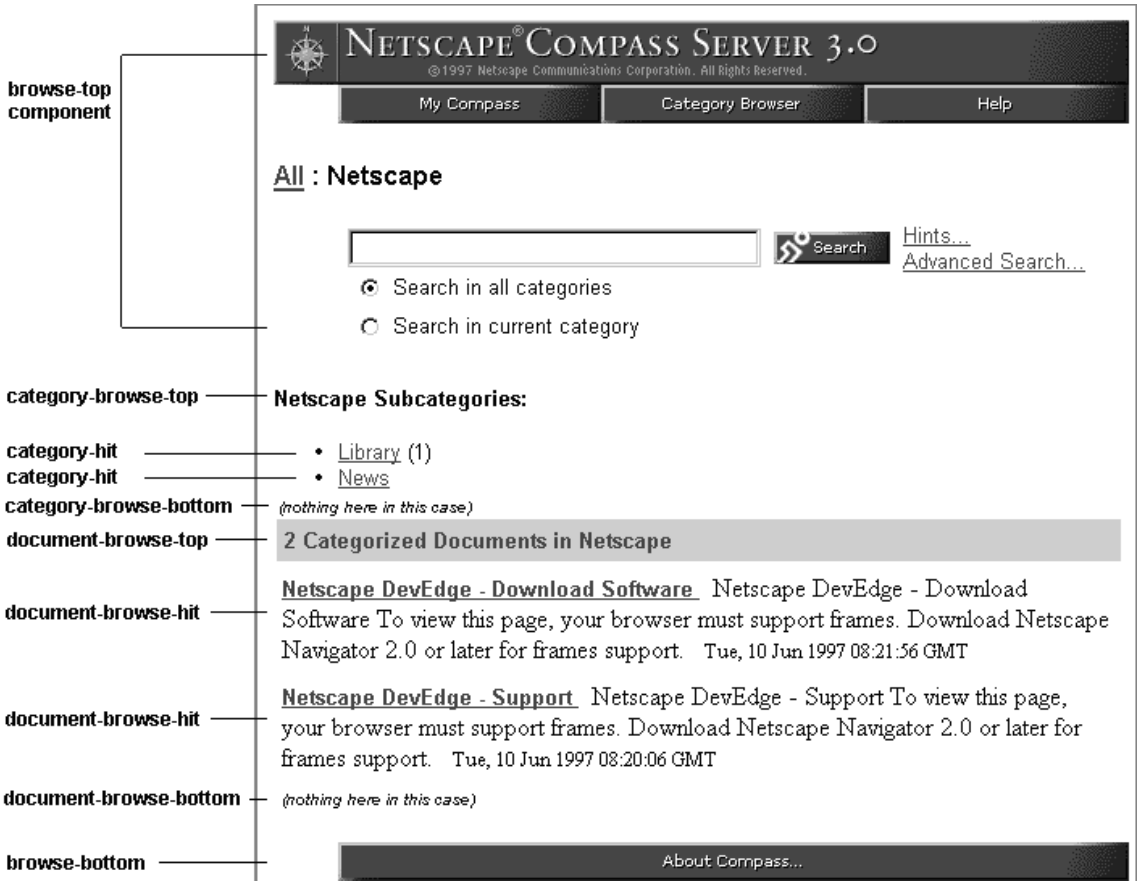

Figure 1.2 Components in a browse mode page

Components in Search-Results Pages

The top (first) component in a search-results mode page is the search-top component, and the bottom (last) component is the search-bottom component.

The Compass End User page uses search-results mode to display the results of a search. The search-results mode page usually displays a list of categories that matched the search criteria and a list of documents that matched the search criteria.

The list of categories has three components: the top of the list (such as a heading or introduction), the entries in the list, and the bottom of the list (such as a closing remark or a link to the next page.) These components are known as the category-match-top, category-match-hit, and category-match-bottom components.

Similarly the list of documents that match the search criteria has three components: the top of the list, the entries in the list, and the bottom of the list. These are known as the document-search-top, document-search-hit, and document-search-bottom components.

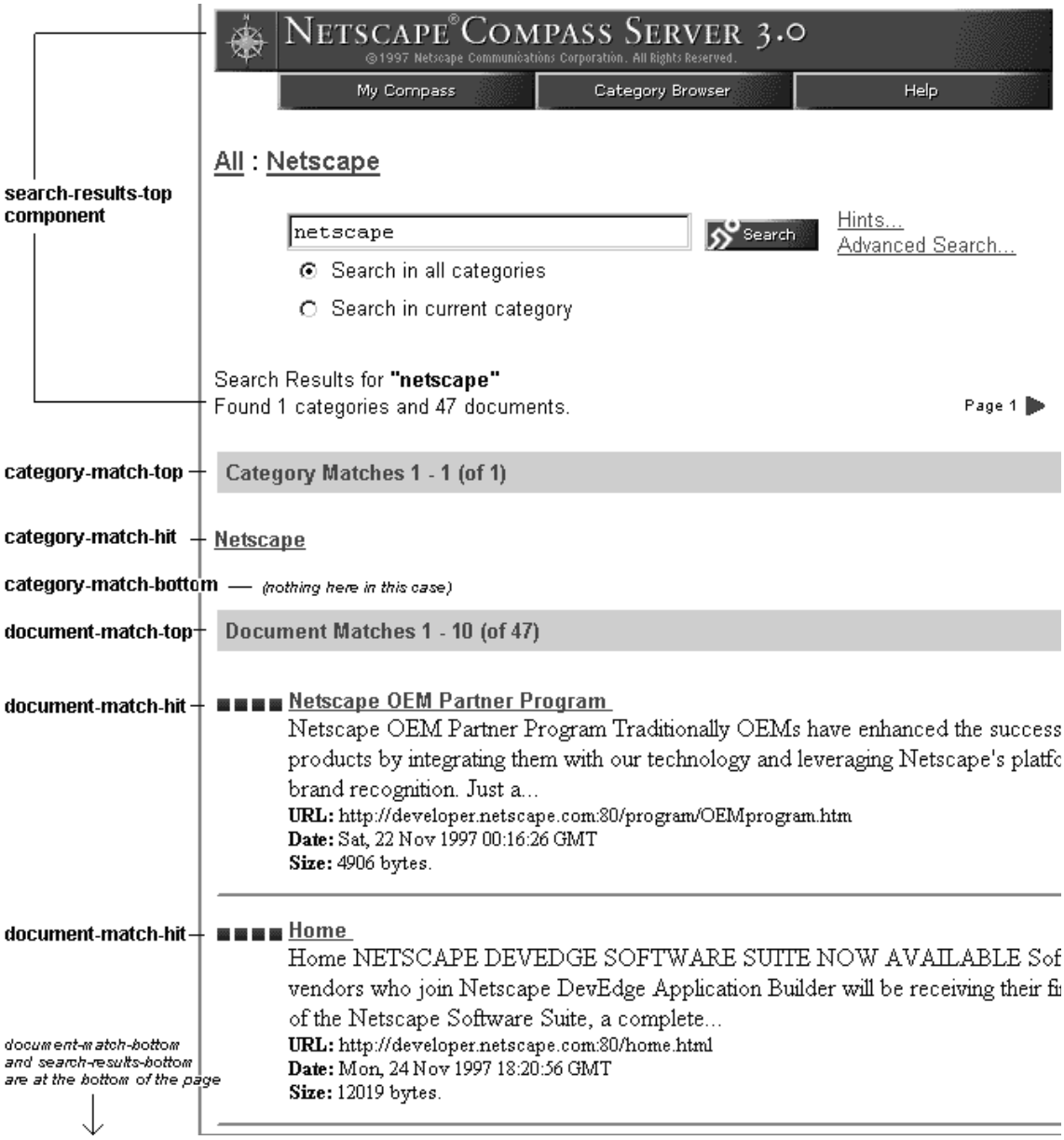

Figure 1.3 Components in a search-results mode page

Summary of the Components in Each Mode

Table 1.1 lists the components used by browse mode and search-results mode pages.

Table 1.1. Components in Browse and Search-results Mode

	Browse Mode	Search-Results Mode
When is this mode used?	When the user first visits the Compass Server End User page, and also whenever the user clicks a category.	To display the results of a search submitted by the user.
What does this mode usually show?	A list of subcategories in a category, and a list of documents in the category.	A list of categories that match the search criteria and a list of documents that match the search criteria.
What components does this mode use?	browse-top category-browse-top category-browse-hit (one for each category) category-browse-bottom document-browse-top document-browse-hit (one for each document) document-browse-bottom browse-bottom	search-results-top category-match-top category-match-hit (one for each category) category-match-bottom document-match-top document-match-hit (one for each document) document-match-bottom search-bottom

Chapter

2

Templates for the End User Page

Each time a user clicks a category name or submits a search in the Compass Server End User page, the system completely redraws the page, using either browse or search-results mode, depending on what action the user took.

The appearance of the Compass Server End User page is not only governed by which mode it uses, but also by the UI template that is currently selected for use.

The Compass Server comes with a set of pre-defined templates, and you can also create your own. You can specify which template is currently in use by choosing the desired template from the Template menu in the End User screen of the Compass Server Administration environment.

Each template consists of a configuration file and a set of pattern files. The configuration file defines which pattern files to use for which components in the browse mode and search-results modes. Each pattern file contains HTML and/or JavaScript code that defines the content of the component. Pattern files can use special variables to access data such as the number of matches in the current search query, which category was being searched, and so on.

For example, the configuration file for the "Normal" template is called `normal.conf`. This configuration file has code that specifies which pattern files define which components in pages that use this template. For example, the browse-top component of the End User page is defined by the `normal-bw-`

`top.pat` pattern file; the category-browse-top component is defined by the `normal-cb-top.pat` pattern file; the category-browse-hit component is defined by the `normal-cb-hit.pat` file; and so on.

To create a new template, you can copy and rename an existing configuration file. Edit the configuration file and specify which pattern files it uses. You can use existing pattern files, or you may want to create new ones, by copying and modifying existing ones. You can then modify the pattern files to suit your needs, using HTML or JavaScript.

The configuration file and all pattern files in a template must live in the `templates` directory for the Compass Server. If you create additional configuration and pattern files, you can make them available to the Compass Server simply by putting them in the `templates` directory.

Saving and Applying Changes to UI Templates

This section describes the mechanics of creating and modifying templates, and loading the changed files into the Compass Server. You will learn more about the details of editing configuration and pattern files in the following chapters.

- Creating a New Template
- Loading Updated Template Files
- Selecting a New UI Template

Creating a New Template

To create a new template, copy an existing configuration file and rename it to something that ends in `.conf`. Open the file in your favorite text editor and change the value of the `RDM-description` variable at the top of the file to a description of your new template. (This description will appear in the Template menu of the End User screen of the Compass Server Administration environment.)

Change the values of variables in the configuration file as appropriate. See the section "Configuration Files for UI Templates" for a discussion of these variables. Save the configuration file.

Copy and modify any pattern files as needed. See the section "Pattern Files for UI Templates" for a discussion of pattern files. Save the changes.

Make sure that the configuration file and all pattern files that it uses are saved in the `templates` directory of the server that will be using the template.

Selecting a New UI Template

In the Compass Server Administration environment, go to the End User screen, by selecting the End User button on the Administration page for the appropriate server.

In the "Search/Browse Preferences" section, select the desired template from the Template menu. Press the OK button. You will also need to press the Apply button at the top of the page, and press the "Apply Changes" button that subsequently appears.

If the new template uses files that were not previously loaded, or have been modified since they were last loaded, you will also need to load the configuration files.

Loading Updated Template Files

Before you create or modify configuration and pattern files, you need to change the value of the `template-refresh-rate` variable in the `csid.conf` file. This file lives in the `config` directory of the directory containing the specific Compass Server. For example, suppose you have installed Compass Server 3.0 in *`compassdir`*, and have created a server instance named `topper`. In this case, the `csid.conf` file will be in:

```
compassdir/compass-topper/config/csid.conf
```

The `template-refresh-rate` variable tells the Compass Server how often (in seconds) to update the templates used by the End User page.

Set this to a low value such as 1:

```
template-refresh-rate=1
```

This tells the Compass Server to refresh the UI template once every second. This ensures that the Compass Server End User page uses the latest versions of the configuration and pattern files as you make changes to them.

Save the change. Open the Compass Server Administration environment (if it is not already open), and go to the Administration page for the appropriate server.

From any page in the Administration page of the appropriate server, you can press the Apply button at the top of the page. The main frame updates to show the "Apply Changes" frame, which includes a button labelled "Load Configuration Files." Press this button to load the changed `csid.conf` file. The next time you use the Compass Server End User page, you should see that the changes have taken effect.

You only need to change and load the `csid.conf` file once. From then on, any changes you make in the configuration or pattern files used by the currently selected template should automatically appear the first time you search or browse after making the changes.

Hint: If your changes do not appear automatically, make sure you set the `template-refresh-rate` variable in the directory *`compassdir`*`/compass-`*`name`*`/config/`, not in the *`compassdir`*`/bin/compass/config/` directory.

When you have finished developing templates, you should set the `template-refresh-rate` variable to a value such as 3600 so that the Compass Server does not waste time updating its UI template each time a user uses the End User page.

Chapter

3

Configuration Files for UI Templates

Each UI template has one configuration file. A configuration file must have a `.conf` extension. The main purpose of a configuration file is to specify which pattern files are used to create the End User page. However, configuration files also do several other things, such as determining whether categories or documents are displayed first.

This chapter has the following sections:

- Variables in Configuration Files
- Assigning Pattern Files to Components. Which pattern files provide the content for which components?
- Determining the Order of Search Results. Are matching categories or matching documents displayed first?
- Determining What the Compass End User Page Lets the User Do. Can it browse categories? Browse for documents? Search for categories? Search for documents?
- Specifying Which Attributes Can Be Viewed in Hit Components -- this lets you choose which document and category attributes can be displayed.
- Specifying Defaults for Basic Parameters.

- Determining Which Images to Use to Indicate Rankings. Which images should be used to indicate how well categories and documents match a search query.
- Determining What Colors to Use for Documents and Categories. What is the list of alternating colors to use as the background to the descriptions of document hits?

Variables in Configuration Files

A configuration file contains variables that assign pattern files to components, as well as variables used for several other tasks. The names of all variables in a configuration file always start with `RDM-`. RDM standards for *resource description messages*. The RDM protocol is the protocol that the Compass Server uses for discovering and retrieving data about resources on a network.

The first thing a configuration file does is define the `RDM-description` variable, which indicates the name of the template. This name appears in the menu of templates in the End User screen of the Compass Server Administration environment. For example:

```
RDM-description="Corporate Search and Browse"
```

Assigning Pattern Files to Components

A configuration file uses special variables to indicate which pattern file to use for each component in a Compass Server End User page.

Each variable corresponds to a component of the page. The components are discussed in Composition of the End User Page.

The variable names are not case-sensitive.

The variables are:

Variables for components for browse-mode pages

Variable name	Description of value for this variable
`RDM-browse-top`	A pattern file that provides the content for the top of a browse-mode page.
`RDM-browse-bottom`	A pattern file that provides the content for the bottom of a browse-mode page.
`RDM-category-browse-top`	A pattern file that provides the content immediately preceding the list of subcategories in a browsed category.
`RDM-category-browse-hit`	A pattern file that provides the content for each subcategory of a browsed category.
`RDM-category-browse-bottom`	A pattern file that provides the content immediately following the list of subcategories.
`RDM-document-browse-top`	A pattern file that provides the content immediately preceding the list of documents in a browsed category.
`RDM-document-browse-hit`	A pattern file that provides the content for each document in a browsed category.
`RDM-document-browse-bottom`	A pattern file that provides the content following the list of documents in a browsed category.

Variables for components for search-results pages

Variable name	Description of value for this variable
`RDM-search-results-top`	A pattern file that provides the content for the top of a search-results page.

`RDM-search-results-bottom`	A pattern file that provides the content for the bottom of a search-results page.
`RDM-category-match-top`	A pattern file that provides the content immediately preceding the list of categories that matched the search criteria.
`RDM-category-match-hit`	A pattern file that provides the content for each matching category.
`RDM-category-match-bottom`	A pattern file that provides the content immediately following the list of matching categories.
`RDM-document-match-top`	A pattern file that provides the content immediately preceding the list of documents that matched the search criteria.
`RDM-document-match-hit`	A pattern file that provides the content for each matching document.
`RDM-document-match-bottom`	A pattern file that provides the content immediately following the list of matching documents.

The value of each of these variables must be an existing pattern file in the same `templates` directory as the configuration file. For example:

```
RDM-document-match-top=normal-dm-top.pat
RDM-category-browse-top=custom1-bw-top.pat
RDM-category-browse-hit=custom1-bw-hit.pat
RDM-category-browse-bottom=custom1-bw-bot.pat
```

If you want to omit a component, you can just omit the appropriate `RDM-`*variable*. Alternatively, you can set its value to `nothing.pat`, which is a file in the `templates` directory that has no content. For example, the following code causes the Compass Server End User page to omit the components that display subcategories in a category:

```
RDM-category-browse-top=nothing.pat
RDM-category-browse-hit=nothing.pat
RDM-category-browse-bottom=nothing.pat
```

For more information on pattern files, see Pattern Files for UI Templates.

Determining the Order of Search Results

The variable `RDM-search-results-order` determines whether matching categories or matching documents are listed first when the Compass Server End User page displays the results of a search. The value is either `cm,dm` (meaning categories are listed first) or `dm,cm` (meaning documents are listed first).

```
RDM-search-results-order=cm,dm
```

OR

```
RDM-search-results-order=dm,cm
```

Determining What the Compass End User Page Lets the User Do

You can determine what a Compass End User page allows the user to do by specifying values for the `RDM-perform-`*`task`* variables. These are:

- `RDM-perform-document-match`

  Determines whether or not users can search for documents.

- `RDM-perform-document-browse`

  Determines whether or not users can browse for documents.

- `RDM-perform-category-match`

  Determines whether or not users can search for categories.

- `RDM-perform-category-browse`

  Determines whether or not users can browse categories.

The value of each of these variables is `true` or `false`. The following example enables searching for categories and documents, enables category browsing, and disables browsing for documents.

```
RDM-perform-document-match=true
RDM-perform-category-match=true
RDM-perform-document-browse=false
RDM-perform-category-browse=true
```

If a task is enabled, the three components for that task are included in the page. If the task is disabled, the three components are not included. For example, if document browsing is disabled, then the document-browse-top, document-browse-hit, and document-browse-bottom components will not be included in the Compass Server End User page when it uses browse mode.

Note however that the browse-top and browse-bottom components are always included in browse-mode pages, regardless of what tasks are disabled. Similarly, search-top and search-bottom components are always included in search-results mode pages. Both the search-top and browse-top components often include a search query entry field. In the unlikely event that you want to disable searching for both categories and documents, you should make sure that the pattern files used for the browse-top and search-top components do not display a search query entry field. If the user sees a search query entry field, they will expect to be able to perform searches, and will be surprised if their search requests are ignored.

Specifying Which Attributes Can Be Viewed in Hit Components

You can use `RDM-`*`view`* variables to determine which SOIF attributes can be included in the description of categories and documents. A SOIF attribute is derived from a database field of the resource description for the category or document. For more about SOIF attributes, see SOIF Attributes.

The `RDM-`*`view`* variables are:

- RDM-category-browse-view-attributes
- RDM-category-browse-view-hits
- RDM-category-browse-view-order
- RDM-document-browse-view-attributes
- RDM-document-browse-view-hits
- RDM-document-browse-view-order
- RDM-category-match-view-attributes
- RDM-category-match-view-order

- RDM-document-match-view-attributes
- RDM-document-browse-view-order

RDM-category-browse-view-attributes

Specifies which `$$SOIF-`*`attribute`* variables can be used in the category-browse-hit components in browse-mode pages. A `$$SOIF-`*`attribute`* is an attribute that is derived from a database field for a resource description. For more about SOIF attributes, see SOIF Attributes.

For example, the following statement allows the `$$id`, `$$parent-id`, and `$$description` variables to be used in the `RDM-category-browse-hit` pattern file. (This basically means that the end user page can display the id, parent-id, and description for browsed categories.)

```
RDM-category-browse-view-attributes=id,parent-id,description
```

RDM-category-browse-view-hits

Specifies the first and last subcategory to be displayed in the list of subcategories in browse-mode pages.

For example, the following statement invokes the `RDM-category-browse-hit` pattern file to display the first subcategory, and continues invoking it until the 100th subcategory has been displayed.

```
RDM-category-browse-view-hits=1..100
```

RDM-category-browse-view-order

Specifies which attributes are used to order the subcategories. For example, the following statement specifies that the subcategories are ordered by increasing value of their id. Since a category id is basically the category name, the subcategories are listed in increasing alphabetical order.

```
RDM-category-browse-view-order=+id
```

RDM-document-browse-view-attributes

Specifies which `$$SOIF-`*`attribute`* variables can be used in the document-browse-hit components in browse-mode pages. A `$$SOIF-`*`attribute`* is an attribute that is derived from a database field for a resource description. For more about SOIF attributes, see SOIF Attributes.

For example, the following statement allows the `$$url`, `$$title`, `$$description`, and `$$author` variables to be used in the `RDM-document-browse-hit` pattern file. (This basically means that the End User page can display the URL, title, description, and author of documents.)

```
RDM-document-browse-view-attributes=url,title,description,author
```

RDM-document-browse-view-hits

Specifies the first and last documents in a category to be displayed in browse-mode pages.

For example, the following statement invokes the `RDM-document-browse-hit` pattern file to display the first document in the current category, and continues invoking it until the 20th document has been displayed. That is, for each subcategory, up to 20 documents in the category will be shown.

```
RDM-document-browse-view-hits=1..20
```

RDM-document-browse-view-order

Specifies which attributes are used to order the documents in a category in browse-mode pages. For example, the following statement specifies that documents are listed in increasing alphabetical order of their titles.

```
RDM-document-browse-view-order=+title
```

RDM-category-match-view-attributes

Specifies which `$$SOIF-`*`attribute`* variables can be used in the category-match-hit components in search-results mode pages. A `$$SOIF-`*`attribute`* is an attribute that is derived from a database field for a resource description. For more about SOIF attributes, see SOIF Attributes.

For example, the following statement allows the `$$id` variable to be used in the `$$RDM-category-match-hit` pattern file. (This basically means that the End User page can display a category's id.)

```
RDM-category-match-view-attributes=id
```

RDM-category-match-view-order

Specifies which attributes are used to order the matching categories in a search-results page. For example, the following statement specifies that matching categories are ordered by decreasing value of their match score. Categories that have the same score are ordered by increasing alphabetical order of their id.

```
RDM-category-match-view-order=-score,+id
```

RDM-document-match-view-attributes

Specifies which `$$SOIF-`*`attribute`* variables can be used in the document-match-hit components in search-results pages. A `$$SOIF-`*`attribute`* is an attribute that is derived from a database field for a resource description. For more about SOIF attributes, see SOIF Attributes.

For example, the following statement allows the `$$url`, `$$title`, `$$description`, and `$$last-modified` variables to be used in the `RDM-document-match-hit` pattern file. (This basically means that the End User page can display the URL, title, description, and last modified date of each document.)

```
RDM-document-match-view-attributes= url,title,description,last-modified
```

RDM-document-match-view-order

Specifies which attributes are used to order the matching documents in a search-results page. For example, the following statement specifies that the matching documents are ordered by decreasing value of their match score.

```
RDM-document-match-view-order=-score
```

Specifying Defaults for Basic Parameters

You can specify default values for some basic parameters by assigning values to the following variables:

- `RDM-default-browse-category`

  which category to browse by default. For the top-level category, you can give the value as `root`.

- `RDM-default-chunk-size`

  how many documents returned by a search query to display in one page. This variable has a maximum value of 2730.

- `RDM-default-page`

  Page number for the first page showing the results of a search.

- `RDM-default-scope`

  The default search string to display in search query entry fields.

- `RDM-default-search-category`

  which category to search by default.

For example:

```
RDM-default-browse-category=ROOT
RDM-default-chunk-size=10
RDM-default-page=1
RDM-default-scope=NULL
RDM-default-search-category=NULL
```

Determining Which Images to Use to Indicate Rankings

When a user submits a search, the page that shows the results can display "ranking" icons that indicate how well each document matches the query. (Usually the document-match-hit component displays these icons.)

You can specify whatever images you want to use as the ranking icons, by specifying values for the following variables:

- `RDM-score-icon1`

  indicates 0-20 percentage match.

- `RDM-score-icon2`

  indicates 21-40 percentage match.

- `RDM-score-icon3`

indicates 41 -60 percentage match.

- `RDM-score-icon4`

  indicates 61-80 percentage match.

- `RDM-score-icon5`

  indicates 81-100 percentage match.

The value of each of these variables must be a GIF or JPEG file. For example:

```
RDM-score-icon1=images/goldstar1.gif
RDM-score-icon2=images/goldstar2.gif
RDM-score-icon3=images/goldstar3.gif
RDM-score-icon4=images/goldstar4.gif
RDM-score-icon5=images/goldstar5.gif
```

Determining What Colors to Use for Documents and Categories

When the user browses a category, the documents in the category can be shown in a table, where each document appears in a separate row that has a different background color from its adjacent rows. Similarly, when the end user searches for documents, the matching documents can be displayed in table rows with different colored backgrounds. It is also possible to display categories in tables, where each row has a different colored background.

You can specify the list of colors to use as the backgrounds of the table rows by providing a value for the `RDM-rotated-colors` variable. The value must be a comma-separated list of colors, with no spaces surrounding the commas. For example:

```
RDM-rotated-colors="red,white,blue,green"
RDM-rotated-colors="#445566,#67FF22,cyan,#FFAABC,#708090"
```

The table rows are defined by the pattern files assigned to the `RDM-document-browse-hit`, `RDM-document-match-hit`, `RDM-category-browse-hit`, and `RDM-category-match-hit` variables. The rotated colors will be used if these files make use of the `RDM-rotated-colors` variable.

Whenever the `$$RDM-rotated-colors` variable is used in the pattern files for the category-match-hit, category-browse-hit, document-match-hit, or document-browse hit components, the color will be advanced to the next color in the list. (The list is cyclic.)

Chapter

4

Pattern Files for UI Templates

Each time the Compass Server End User page is displayed, it is assembled from several components. The content of each component is defined by a pattern file.

This chapter has the following topics related to pattern files:

What is a Pattern File?

A pattern file is an HTML file. It can contain HTML, JavaScript, or any other content that can be displayed by a web browser.

Each pattern file defines a piece of a Compass Server End User page. The piece at the very top of the page has the tags that start an HTML page, such as <HTML>, <HEAD>, and <BODY>. The piece at the bottom of the page has the tags that close an HTML page, such as </BODY> tag and </HTML> tags.

Pattern files can use special variables to access data such as the number of documents that match the current search query, which category was being searched, and so on. You can use these variable directly in HTML and also in JavaScript code. If you want to use these variables inside Java code, you need to use the Java Search SDK, which is described in the Chapter 14, “Using Java to Access and Modify the Compass Server Database.”

Using Variables

Each special variable starts with `$$`, for example, `$$RDM-Hits-Searched`. See Attributes derived from the Compass Database Schema for a full list of the variables.

You can use the pre-defined variables in three ways:

- `$$var`

  Use the variable name by itself to evaluate to the variable’s value. For example, the following code displays a table cell that shows the title of a document. `$$url` is the URL of the document, `$$title` is the title of the document, and `$$description` is the description.

  ```
  <td valign=top align=left width=90%>
  <a href="$$url"><b>$$title</b></a>
  <BR> $$description
  </td>
  ```

- `$$var[do this if $$var is defined]`

  If the variable name is followed by a single set of square brackets, the contents of the brackets are displayed or evaluated only if the named variable has a value. Basically this syntax is shorthand for "if $$var has a value, then do the code in square brackets."

  For example, the following code uses the variable `$$rdm-scope`, which evaluates to the text for which the user is searching. If the user did not submit a search, then `$rdm-scope` has no value. If `$$rdm-scope` has a value, (for example, "springer spaniel"), then this code prints a paragraph such as "Search results for springer spaniel:"

  If `$$rdm-scope` has no value, nothing is displayed.

  ```
  $$rdm-scope[<P>Search results for <I>$$rdm-scope:</I></P>]
  ```

- `$$var[do this if $$var is defined][do this if $$var is not defined]`

  If the variable name is followed by two sets of square brackets, the contents of the first set of brackets are displayed or evaluated only if the named variable has a value. The contents of the second set of brackets are displayed or evaluated if the named variable does not have a value. (Note that if the variable's value is 0 or false, the variable is considered to have a value.)

  Basically this syntax is shorthand for "if $$var has a value, then do the code in the first set of square brackets, otherwise do the code in the second set of square brackets."

  For example, the following code uses the variable `$$rdm-scope`, which evaluates to the text for which the user is searching. If the user did not submit a search, then `$rdm-scope` has no value. If `$$rdm-scope` has a value, (for example, "australian shepherd"), then this code prints a paragraph such as "Search results for australian shepherd:"

  If `$$rdm-scope` has no value, the message "No search submitted" is displayed.

  ```
  $$rdm-scope[<P>Search results for <I>$$rdm-scope:</I></P>]
  [<P>No search submitted</P>]
  ```

SOIF Attributes

Some of the variables you can use in the pattern files have their values set by variables in the configuration file. For example, the value of the `$$RDM-rotated-color` variable is drawn from a list that is defined in the configuration file. Other variables give you information about the current environment, or what the user is doing, for example `$$category` variable tells you what category the user is currently browsing, while the `$$RDM-scope` variable tells you what text the user was using as their search criteria.

Yet other variables reveal data that is stored in the Compass Server's database. Each entry in the database is known as a resource description (RD). Each resource description contains data about a resource, such as a web page, on the network. Each resource description has multiple attributes or fields. These attributes are known as SOIF attributes, where SOIF stands for Summary Object Interchange Format.

A SOIF attribute is an attribute whose value is derived from a field in the database. For each SOIF attribute, there is a `$$` variable that directly corresponds to that attribute. For example, each resource description has attributes Description and Title, consequently you can use the variables `$$description` and `$$title`.

Some of these SOIF attributes are fairly standard, such as description and title. However, developers can set up the schema of their database however they like.

When editing the schema of your database (as discussed in the *"Compass Server Administrator's Guide"*), you can add new attributes or delete existing attributes. When the robot finds a resource, such as an HTML file, to be indexed, it needs to know how to get the data to assign to each attribute for storage in the database. The most common way to map resource data to database attributes is through the use of the META tag. That is, the robot looks for a META tag whose name is the same as the attribute, and if it finds such a tag, the robot takes the content of the tag and uses it as the value of the attribute in the database.

Anyway, as you can see, the precise list of `$$`*`SOIF-attribute`* variables varies from database to database, depending on the schema of the database.

List of Variables

This section discusses the `$$` variables you can use in your pattern files. The variables are presented in tables categorized by use.

- Attributes derived from the Compass Database Schema
- Hit variables
- Variables for Use in Top Pattern Files only
- Variables that are constant for each browse or search submission
- Page-related variables
- Developer-editable variables defined in the configuration file

Table 4.1. Attributes derived from the Compass Database Schema

Variable Name	Description

Table 4.1. Attributes derived from the Compass Database Schema

`$$SOIF-Attribute`	A SOIF attribute is an attribute included in the resource description of a document in the Compass Server database. The precise list of $$SOIF-attributes depends on the schema of the database, which can be edited by the Compass Server administrator. Common $$SOIF-Attributes include `$$Title`, `$$URL`, `$$Description`. For more information about $$SOIF-attribute variables, see SOIF Attributes. Also note that you can only use $$SOIF-attributes in category-hit and document-hit components. Each $$SOIF-attribute used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$id`	The id of the category being processed, which is usually the name of the category. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$title`	The title of the document currently being processed. The title is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$author`	The author of the document currently being processed. The author is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.

Table 4.1. Attributes derived from the Compass Database Schema

`$$description`	The description of the document or category currently being processed. The description is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$URL`	The URL of the document currently being processed. The URL is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$last-modified`	The date the document was last modified. This value is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$content-length`	The length of the document in bytes. This value is taken from the resource description in the Compass Server database. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.

Table 4.1. Attributes derived from the Compass Database Schema

`$$classification`	If the current document or category has been classified as being in a particular category, this variable will return that classification category. If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$Preserve-`*`SOIF-Attribute`*	Use this variable to prevent values from being URL-encoded. Use the $$Preserve- version in place of the regular variable to prevent the value from being URL-encoded. For example, the following code displays the unencoded URL in a paragraph. `<P>$$Preserve-URL<P>`
`$$encode-`*`SOIF-Attribute`*	Use this variable to URL-encode the value of a variable. For example, `$$Encode-description` URL-encodes the description. For example, `\n` is converted to `%0A`. Note however that encoding happens by default, so you should not need to use this variable. However, you would use `$$Preserve-description` to prevent the description from being URL-encoded.
`$$schema-name`	The name of the schema for the resource description for this document or category. Usually for a document the value is DOCUMENT and for a category it is CLASSIFICATION.

The variables listed in the previous table and the next table change for each document or category, and should only be used in pattern files for hit components.

Table 4.2. Hit variables

Variable Name	Description
`$$category`	The category currently being processed.

`$$score`	The score indicates how well the current category or document matches the search criteria if a search was submitted.
`$$RDM-Hit`	The number of the hit currently being processed.

`$$RDM-Is-First-Hit`	This variable has the value `true` if the current hit is the first hit. If the hit is not the first hit, this variable has no value.
	You could use this variable with the special conditional syntax. The following example, draws a horizontal line before the title and description of the first hit:
	`$$RDM-Is-First-Hit[<HR>]`
	`$$Title $$Description`
	This example would be used in a document-match-hit or category-match-hit pattern file.

`$$subclassification-count`	This is the number of subcategories in the category being processed.
`$$categorized-count`	This is the number of documents in the category being processed.

Table 4.3. Variables for Use in Top Pattern Files only

`$$RDM-Hits-Available`	The number of available hits (documents or categories in the browsed category or that matched the search criteria)
`$$RDM-Hits-Searched`	The number of entries in the database that were checked during the search. This is usually the total number of entries in the database.
`$$RDM-has-hits`	This variable has the value `true` if the search or browse found categories or documents. For example: `$$RDM-has-hits[<H3>There are some hits</H3>]`

The variables listed in the next table can be used in any pattern file.

Table 4.4. Variables that are constant for each browse or search submission

Variable	Value
`$$UI`	Indicates the kind of End User page in use, that is, indicates whether it is a browse or search template. The value will be `bw` or `sr`.
`$$RDM-View-Template`	This is the name of the template in use. For example, if the configuration file for the template is `normal.conf`, then `$$RDM-View-Template` has the value `normal`.
`$$Browse-Category`	The category that the user is browsing.
`$$Search-Category`	The category that the user is searching.
`$$RDM-Scope`	The string that was submitted as the search criteria.
`$$taxonomy`	The name of the top level category in the taxonomy tree. For example, if the top level category is `netscape`, with subcategories `software` and `documentation`, which in turn each have their own subcategories, `$$taxonomy` is `netscape`.
`$$RDM-Version`	Version of the Compass Server serving the request.
`$$RDM-Server`	URL of the server where the Compass Server is installed.

Table 4.5. Page-related variables

`$$Page`	The current page. If there are more hits than can fit on one page (as determined by `$$Chunk-size`), multiple pages may be needed to display all the hits. `$$Page` evaluates to the page number for the page that is currently being viewed.

`$$Prev-Page`	The previous page. If there are more hits than can fit on one page (as determined by `$$Chunk-size`), multiple pages may be needed to display all the hits. `$$Prev-Page` evaluates to the page number for the page previous to the one that is currently being viewed.
`$$Next-Page`	The next page. If there are more hits than can fit on one page (as determined by `$$Chunk-size`), multiple pages may be needed to display all the hits. `$$Next-Page` evaluates to the page number for the page following the one that is currently being viewed.
`$$RDM-Hit-Max`	The number of the last hit to display in the current page.
`$$RDM-Hit-Min`	The number of the first hit to display in the current page.
`$$RDM-Is-First-Page`	This variable has the value `true` if the current page is the first page that displays a list of hits. If the current page is not the first page, the variable has no value. You could use this variable with the special conditional syntax, for example: `$$RDM-Is-First-Page[text for the first page only]`
`$$RDM-Hits-Returned`	The number of matching documents displayed in the current page. This is equal to or less than `$$Chunk-size`, which defines the maximum number of documents displayed in a page. For example, the last page may show only 5 hits, in which case `$$RDM-hits-returned` is 5 for the last page, although for all other pages it might be 10, which is the value of `$$Chunk-size`.
`$$RDM-has-hits-more`	This variable has the value `true` if there are more than one page worth of hits, otherwise it has no value.

`$$RDM-has-hits-zero`	This value returns `true` if there are no hits, otherwise it has no value. For example: `$$RDM-has-hits-zero[<H3>There are NO hits</H3>]`
`$$RDM-has-hits-this`	This variable has the value `true` if all the hits fit on one page, otherwise it has no value.

Table 4.6. Developer-editable variables defined in the configuration file

Variable name	Value
`$$Chunk-size`	The number of hits that fit in a page. The maximum allowable value is 2730.
`$$RDM-Rotated-Color`	A color in the list of RDM-rotated-colors. Each time you access this variable, the value moves on to the next color in the list. See Determining What Colors to Use for Documents and Categories for details. This variable can only be used in pattern files for hit components (category-browse-hit, category-match-hit, document-browse-hit, document-match-hit.)
`$$RDM-Score-Icon`	This variable evaluates to a URL for an image that represents the ranking of a hit. This image will show one, two, three, four, or five stars (or something like stars) that indicates how well a document matched a search query. For example: `"http/your.server.com/images/star1.gif"`
`$$RDM-Score-Image`	This variable evaluates to an <IMG> tag that points to the URL for an image that represents the ranking of a hit. For example: `<IMG SRC="http/your.server.com/images/star1.gif">`

Pattern File Summary

This section briefly discusses what the pattern files for each component should or could do. Although you can define pattern files to do whatever you want, there are some basic guidelines that cover typical uses of pattern files.

- browse-mode-top and search-results-mode-top
- category-browse-top and category-search-results-top
- category-browse-hit and category-search-results-hit
- category-browse-bottom and category-search-results-bottom
- document-browse-top and document-search-results-top
- document-browse-hit and document-search-results-hit
- document-browse-bottom and document-search-results-bottom
- browse-bottom and search-results-bottom

browse-mode-top and search-results-mode-top

The pattern files for the browse-mode-top and search-results-mode-top components define the content at the top of browse-mode pages or search-mode pages as appropriate. This section refers to these pattern files collectively as *top pattern files*.

Since top pattern files start a page, they must include the `<BODY>` tag for the whole page. You can set background color, link color, text color and so on for the whole page by specifying appropriate attributes for the `<BODY>` tag. For a list of all the attributes for the `<BODY>` tag see the entry for BODY in the *HTML Tag Reference*.

The top pattern files should also define the title for the Compass End User Page (that is, they should have a `<TITLE>` tag.) These files should also include any desired `META` tags or other tags that need to go between `<HEAD>` and `</HEAD>`.

Top pattern files often contain a text field for search criteria. If you want both browse-mode and search-results mode pages to show the text field, then you must include the text field in the pattern files for both the browse-mode-top and search-mode-top components. (For information about defining forms that have text entry field for search criteria, see Using Forms to Submit Searches.)

If you want the Compass Server End User page to display a logo for your company, instead of, or as well as, the Compass Server logo, you need to put the logo in the top pattern files.

category-browse-top and category-search-results-top

The pattern file for the category-browse-top component defines the introduction to the list of categories in the browsed category. The pattern file for the category-search-results-top component defines the introduction to the list of categories that match the search criteria when a search was submitted. These files are known collectively in this section as *category-top files*.

If you want the subcategories or matching categories to be displayed in a structure such as a table or list, you can define the opening tag, such as `<TABLE>` or `<UL>` in the category-top files.

Table 4.3. Variables for Use in Top Pattern Files only lists some variables that may be useful in category-top pattern files. The page-related variables, as discussed in Table 4.5 may also be useful in category-top files. Additionally, the following variables may be useful:

`$$Browse-Category`	The category that the user is browsing.
`$$Search-Category`	The category that the user is searching.
`$$RDM-Scope`	The string that was submitted as the search criteria.
`$$subclassification-count`	This is the number of subcategories in the category being processed.
`$$categorized-count`	This is the number of documents in the category being processed.

category-browse-hit and category-search-results-hit

The pattern file for the category-browse-hit component is invoked for each sub-category in the browsed category. The pattern file for the category-search-results-hit component is invoked for each category that matches the search criteria when a search was submitted. This section refers to these pattern files collectively as *category-hit files*.

You can define category-hit files to display the information about the category being processed. If the corresponding category-top file started a list, then you should display each category as a list item. If the corresponding category-top file started a table, then you should display each category as a table row. You can, if desired, set the background of each row to `$$RDM-rotated-color` so that the table uses alternating colors for the rows. For more details, see Determining What Colors to Use for Documents and Categories.

Variables that may be useful in category-hit files include:

`$$id`	The id of the category being processed, which is usually the name of the category. (This is derived from the Compass Server database.) If this variable is used, it used must be listed in the appropriate RDM-attribute-view variable in the configuration file, as discussed in Specifying Which Attributes Can Be Viewed in Hit Components.
`$$browse-category`	The category that the user is browsing.
`$$search-category`	The category in which the user is searching.
`$$category`	The category currently being processed.
`$$subclassification-count`	This is the number of subcategories in the category being processed.
`$$categorized-count`	This is the number of documents in the category being processed.

category-browse-bottom and category-search-results-bottom

The pattern file for the category-browse-bottom component defines the content that follows the list of subcategories in a browse-mode page. The pattern file for the category-search-results-bottom component defines the content that follows the list of categories that matched the search criteria when a search was submitted. This section refers to these two files collectively as *category-bottom files*.

If a category-top file start a structure such as a table or list, the corresponding category-bottom file must close that structure, for example by writing a `</TABLE>` or `</UL>` tag.

document-browse-top and document-search-results-top

The pattern file for a document-browse-top component defines the introduction to the list of documents in a browsed category. The pattern file for a document-search-results-top component defines the introduction to the list of documents returned by a search submission. These two pattern files are referred to collectively in this section as *document-top files*.

If you want to display the documents in a structure such as a table or a list, you can define the opening tag, for example `<TABLE>` or `<OL>` in the document-top files.

Table 4.3. Variables for Use in Top Pattern Files only lists some variables that can be useful in document-top pattern files. The page-related variables, as discussed in Table 4.5, can also be useful in document-top files. Additionally, the following variables may be useful:

`$$Browse-Category`	The category that the user is browsing.
`$$Search-Category`	The category that the user is searching.

`$$RDM-Scope`	The string that was submitted as the search criteria.

document-browse-hit and document-search-results-hit

The pattern file for the document-browse-hit component is executed once for each document in the browsed category. The pattern file for the document-search-hit component is executed once for each document that matches the search criteria when a search has been submitted. These pattern files are referred to collectively in this section as *document-hit files*.

Usually document-hit files write the data about the document currently being processed.

You can define the document-hit files to display the documents however you like. However, it is common practice to display them in a table, with each document displayed in its own row, where each row has one cell that shows the ranking icon (if applicable) and one cell that shows the document description. You can, if desired, set the background of each row to `$$RDM-rotated-color` so that the table uses alternating colors for the rows. For more details, see Determining What Colors to Use for Documents and Categories.

To retrieve information about a document, you can use the SOIF variables, as discussed in Table 4.1. Attributes derived from the Compass Database Schema.

Other variables that may be useful in document-hit files include:

`$$RDM-Hit`	The number of the hit currently being processed.

`$$RDM-Is-First-Hit`	This variable has the value `true` if the current hit is the first hit. If the hit is not the first hit, this variable has no value.
	You could use this variable with the special conditional syntax. The following example, draws a horizontal line before the title and description of the first hit:
	`$$RDM-Is-First-Hit[<HR>]`
	`$$Title $$Description`
	This example would be used in a document-match-hit or category-match-hit pattern file.

`$$RDM-Rotated-Color`	A color in the list of RDM-rotated-colors. Each time you access this variable, the value moves on to the next color in the list. See Determining What Colors to Use for Documents and Categories for details. This variable can only be used in pattern files for hit components (category-browse-hit, category-match-hit, document-browse-hit, document-match-hit.)
`$$score`	The score indicates how well the current category or document matches the search criteria if a search was submitted.
`$$RDM-Score-Icon`	This variable evaluates to a URL for an image that represents the ranking of a hit. This image will show one, two, three, four, or five stars (or something like stars) that indicates how well a document matched a search query. For example: `"http/your.server.com/images/star1.gif"`
`$$RDM-Score-Image`	This variable evaluates to an <IMG> tag that points to the URL for an image that represents the ranking of a hit. For example: `<IMG SRC="http/your.server.com/images/star1.gif">`

document-browse-bottom and document-search-results-bottom

The pattern file for the document-browse-bottom component writes concluding remarks at the bottom of the list of documents in a browsed category. The pattern file for the document-search-results-bottom component writes the concluding remarks at the bottom of the list of documents returned by a search submission. These files are known collectively as *document-bottom files* in this section.

Often these files do nothing, or they write closing tags such as `</TABLE>` or `</OL>` that were opened in the corresponding document-top components. If the document-top files open structures such as tables or lists, you must close these structures in the corresponding document-bottom files.

browse-bottom and search-results-bottom

The pattern files for the browse-bottom and search-results-bottom components define the content that appears at the bottom of browse-mode and search-results mode pages as appropriate.

Since these files end the page, they should include `</BODY>` and `</HTML>` tags.

Chapter

5

Opening the End User Page With Query String URLs

You can invoke the Compass Server End User Page by opening a URL that has a query string appended. This section describes how to do this, and discusses some of the uses.

Attributes for Query String URLs

This section lists the attributes you can use in the query string. The section Using Forms to Submit Searches discusses how to use forms to submit query strings URLs to the Compass Server.

ui

Selects the mode of the End User Page, which is either browse or search-results. The default is `bw`, which is the same as `browse`. The value can be:

```
bw
browse
sr
```

`search-results`

For example:

`http://www.yourcompass.com/compass?ui=browse`

view-template

The name of the template to use. The default is `normal`. The value can be the main part of the name of any configuration file in the `templates` directory. For example:

`http://www.yourcompass.com/compass?view-template=searchandbrowse`

taxonomy

The name of the taxonomy. This is the name of the top level category in the category tree.

The value can be any text string (which should specify an existing top level category if you want any results returned). There is no default.

browse-category

The category to browse. This is only applicable when `UI` is `bw` or `browse`. The default is the root category. For example:

`http://www.yourcompass.com/compass?ui=bw&browse-category=netscape`

scope

The search criteria to search for. This is only applicable when `UI` is `sr` or `search-results`. There is no default. The following query string URL returns the results of searching for the string "netcaster."

`http://www.yourcompass.com/compass?ui=sr&scope=netcaster`

chunk-size

The maximum number of hits to display per page. This value sets the limit for both the number of categories and number of documents listed per page. The default is 10, and the maximum allowable value is 2730. For example:

`http://www.yourcompass.com/compass?ui=sr&scope=netcaster&chunk-size=8`

page

This is the number of the page to display. This is only applicable when `UI` is `sr` or `search-results`. For example, if a search yields 300 hits, and the chunk-size is 10, there will be 30 pages, so you could display the 15th page as follows:

```
http://www.yourcompass.com/compass?ui=sr&scope=netcaster&page=15
```

search-category

This is the category to search. This is only applicable when `UI` is `sr` or `search-results`. The possible value can be any classification id, such as `arts:music` or `netscape:netcaster`. For example:

```
http://www.yourcompass.com/compass?ui=sr&scope=netcaster&search-
category=netscape
```

Using Forms to Submit Searches

As already mentioned, you can invoke the Compass Server End User Page by opening a URL that has a query string appended. Thus you can use forms to submit searches. These forms can appear on any page on your web site, so long as you give the form elements names that the Compass Server can parse.

To use a form to initiate a search, the form must have a METHOD attribute of GET, and its ACTION must be a URL to your Compass Server. The form must contain at least the following elements:

- A text entry field whose name is `scope` since the attribute `scope` specifies the search string for the Compass Server to search for). This field is where the user enters the string to search for.
- An element, usually a hidden element, whose name is `UI` and whose value is `sr` or `search-results`.
- A submit button, or some means of submitting the form.

The form can contain any additional elements you like whose names correspond to attributes that the Compass Server can understand. For example, you could have a SELECT menu whose name is `search-category` that lists the categories that a user can search.

The following code shows a simple form that allows users to submit a search. This form can be placed on any web page. It will return the Compass Server UI page listing the results of the search.

```
<FORM METHOD=GET ACTION=http://your.host.com/compass>
 <INPUT TYPE=text NAME=scope>
 <INPUT TYPE=submit VALUE=Search>
 <INPUT TYPE=hidden NAME=ui VALUE=sr>
</FORM>
```

The following code creates a form that uses the "Search button" image that the Compass Server End User page uses. This form uses JavaScript so that it can specify an image as the button.

```
<NOSCRIPT>This page requires JavaScript</NOSCRIPT>

<FORM METHOD=GET
  ACTION=http://your.host.com/compass NAME=searchform>
  <NOBR>
  <INPUT TYPE=text NAME=scope>
  <A HREF="javascript:document.searchform.submit()">
    <IMG SRC=http://your.host.com//images/search.gif BORDER=0
      ALIGN=ABSMIDDLE ALT=Search></A>
  </NOBR>
  <INPUT TYPE=hidden NAME=ui VALUE=sr>
</FORM>
```

The following form also uses an image for its Submit button. It uses an INPUT tag whose type is `image` as the Submit button. This form does not require JavaScript. However, the `<INPUT TYPE=image>` tag does not take an `ALT` attribute, so it cannot specify alternative text for the image if the image does not appear.

```
<FORM METHOD=GET ACTION=http://your.host.com/compass>
 <INPUT TYPE=text NAME=scope>
 <INPUT TYPE=image VALUE=Search
     SRC="http://your.host.com//images/search.gif">
 <INPUT TYPE=hidden NAME=ui VALUE=sr>
</FORM>
```

If you want to include the Hints link, you can just add the link, as shown below.

```
<NOSCRIPT>This page requires JavaScript</NOSCRIPT>

<FORM METHOD=GET
    ACTION=http://your.host.com/compass NAME=searchform>
  <NOBR>
  <INPUT TYPE=text NAME=scope>
  <A HREF="javascript:document.searchform.submit()">
  <IMG SRC=http://your.host.com//images/search.gif
     BORDER=0 ALIGN=ABSMIDDLE ALT="Search"></A>
  <A HREF=http://your.host.com/ug/hints.htm>Hints...</A>
  <INPUT TYPE=hidden NAME=ui VALUE=sr>
```

```
  </NOBR>
</FORM>
```

The file mysearch.htm shows examples of these forms in a separate window. The forms in this file use dummy names for the Compass Server, thus they do not return real results. However, if you copy the code and substitute the name of your Compass Server, the forms should work.

Optional Advanced Search Interfaces

Several of the pre-defined UI templates allow the user to choose a standard search view or an advanced search view. The standard view offers a single text entry field in which the user enters the string for which to search. The advanced view offers multiple fields that allow the user to express search criteria, and the user can also add more fields dynamically.

This functionality makes use of the ability to use forms to pass name/value pairs to CGI programs. In this case, the page contains links that re-display the page by submitting an invisible form that sends a query string URL specifying a `view-template` attribute to the Compass Server.

You can use this approach to allow users to choose different views of the page, such as a "standard" search view or an "advanced" search view.

You can create End User pages that allow the user to choose different views of the page by using a form that contains several hidden elements. The form should have one hidden element that indicates a UI template (also known as a view template), and another hidden element that indicates the category being browsed, as shown here:

```
<!-- The Search Switch Form --
the view-template MUST be the 0th element -->

<FORM METHOD="GET" ACTION="/compass" NAME="searchswitch">
<INPUT TYPE=hidden NAME=view-template VALUE="$$view-template">
<INPUT TYPE=hidden NAME=browse-category VALUE="$$browse-category">
</FORM>
```

You should put this form in the search-results-top and browse-results-top pattern files.

The form doesn't really do anything, instead, it provides a mechanism for passing name/value pairs to a CGI program that redisplays the page. Each component of the page that includes this form needs to have a link, that, when pressed, invokes the form with the appropriate values of `view-template` and `browse-category`.

The value of the `view-template` attribute should be the UI template to use when the page is redisplayed. The value of the `browse-category` attribute should be the category to be browsed.

To invoke a form within a link, use `javascript:` to specify that the link is JavaScript code. Within JavaScript, you can use the `submit()` function to submit a specified form. However, in this case you need to change the value of the form' s 0th element (that is, the element named `view-template`) to be the new template rather than the current template.

The following link says "get the document's `searchswitch` element (which in this case is a form). Then set the `searchswitch` element's 0th element (which is the hidden element named `view-template`). to "`advanced`". Then call the `submit()` function on the `searchswitch` form. The text in the link says "Advanced Search."

```
<a href =
"javascript:document.searchswitch.elements[0].value=&quot;advanced&quot;;
document.searchswitch.submit();">

Advanced Search...</a>
```

Note that there should be no line breaks in the `javascript:` URL.

There is no need to pass a value for the `browse-category` attribute, since it defaults to the existing value in the form, `$$browse-category`, which evaluates to the current category.

In a nutshell, when this link is invoked, it causes the page to be displayed again with the template named `advanced`.

When the page redisplays, it uses the pattern files defined in the configuration file for the `advanced` template. The entire page (rather than just the component containing the search form) is redisplayed using the pattern files specified in the configuration file for the `advanced` template.

The following paragraphs summarize the steps involved in creating two alternative templates, one for a standard search, and one for an advanced search.

First, create two new configuration files, one for the standard template and one for the advanced template.

- Standard Template

  In the search-results-top and browse-results-top pattern files in the standard template, include the standard search form.

  Include the invisible searchswitch form.

  Include a link that says something like "Advanced Search" that uses a `javascript:` URL that opens the page again (by modifying and submitting the searchswitch form) using the advanced template.

  Modify the other pattern files in the template as required.

- Advanced Template

  In the browse-top and search-results-top pattern files in your advanced template, include the advanced search form. Also include the invisible searchswitch form and a link that says something like "Standard Search" that redisplays the page using the standard template.

  The easiest way to do this is to copy the `adv-searched-browse-top` and `adv-browse-top` files and modify them to suit your needs. These files already contain advanced search forms as well as the search switch form and the link to show the standard search box. Just make sure that the link uses the name of your standard template.

  Modify the other pattern files in the template as required.

Chapter

6

Tutorial: Customizing the End User Page

This section provides step-by-step instructions for creating a new user interface template.

To test the results of the new template, you must use a Compass Server whose database already contains documents, categories, and categorized documents.

As you work through the tutorial, you will modify some files, and create others from scratch. The more you understand how to use HTML and JavaScript, the easier will be the task of modifying pattern files. Some pattern files, such as the one that creates the list of subcategories in a category, are fairly complex, and it is best to make only minor modifications to them. Other files, such as the pattern file that defines the appearance of a document that was returned by a search submission, are fairly simple, and you can modify them as much as you want.

Part of the skill involved in modifying pattern files is identifying which code is best left alone, and which is suitable for modification. Of course, the more you understand JavaScript, the more complex the changes you can make. However, even inexperienced JavaScript programmers can make substantial changes to the appearance of the Compass Server End User page by modifying the pattern files if they constrain themselves to modifying code that they understand.

To get the most from this tutorial, it is best to work through it from beginning to end.

The main tasks covered in the tutorial are:

Getting Ready for Development

First, you need to set up the development environment so that any changes you make in the configuration and pattern files will take effect immediately. To do this, you need to make a small change in the `csid.conf` file. This file lives in the `config` directory of the directory containing the specific Compass Server. For example, suppose you have installed Compass Server 3.0 in *`compassdir`*, and have created a server instance named `topper`. In this case, the `csid.conf` file will be in:

```
compassdir/compass-topper/config/csid.conf
```

Open the file `csid.conf` in a plain-text editor. Search for `template-refresh-rate`, (it will be close to the end of the file) and set its value to 1 as follows:

```
template-refresh-rate=1
```

Save the file.

Now you need to apply the change to the Compass Server. In the Compass Server Administration interface, select the Apply button, and then select the Load Configuration Files button.

Copying a Configuration File

The next thing to do is to copy the `normal.conf` configuration pattern and save it to a new name. This file lives in the `templates` directory of the directory containing the specific Compass Server. For example, suppose you have installed Compass Server 3.0 in *`compassdir`*, and have created a server instance named `topper`. In this case, the `normal.conf` file will be in:

`compassdir``/compass-topper/templates/normal.conf`

Make a copy of the `normal.conf` file and name it `yourcompany.conf`.

Open the file `yourcompany.conf`. Near the top of the file, you see a statement that defines the variable `RDM-description`. The `RDM-description` value determines the name of the template as it appears in the menu of templates in the Compass Server End User administration page. Change the value of `RDM-description` as follows.

```
RDM-description="Your Company Search and Browse "
```

Changing the Background and Adding a Heading to Browse Pages

In this section, you'll learn how to modify the content that appears at the top of browse-mode pages, and also how to modify the BODY tag that sets the background characteristics for browse-mode pages.

Create the file `yourcompany-bw-top.pat` as a copy of `normal-bw-top.pat`.

Open the file `yourcompany-bw-top.pat` in a plain-text editor. You'll see a `<SCRIPT>` tag near the top of the file that defines a series of functions. Don't worry about those for now. Scroll down the file or search for the `BODY` tag, which starts the actual page. Change the background color to pale yellow:

```
BODY BGCOLOR="#FFFFEF"
```

Immediately after the `<BODY>` tag, add a heading that mentions your company's name, and mention that this is a browse-mode-page, for example:

```
<H1><FONT COLOR="#FFFF00">Welcome to the Search Page for My Company
(Browse-Mode-Page)</FONT></H1>
```

Save the file.

Now you need to update the configuration file to use the new pattern file.

Edit the file `yourcompany.conf` and search for `RDM-browse-top`. Change the value of `RDM-browse-top` to `yourcompany-bw-top.pat`, as follows:

```
RDM-browse-top=yourcompany-bw-top.pat
```

Save the file.

Testing the Change

Before you can test the changes you have made, you need to tell the Compass Server to use your new configuration file. In the Compass Server Administration interface, open the End User page.

In the Search/Browse Preferences section, select **Your Company Search and Browse** in the Templates field, then press the **OK** button.

Stop the server then start it again so that the template change can take effect.

Access your server as a client to see the new "YourCompany" browse page.

You see that the browse page uses the new pattern file. However, if you submit a search, you'll see that the search page still has a white background. This is because in the file `yourcompany.conf`, the value of `RDM-search-results-top` is still `normal-sr-top.pat`. The file assigned to `RDM-search-results-top` defines the `BODY` tag and the top section of the document for pages generated by submitting searches.

Changing the Background and Adding a Heading to Search Results Pages

In this section, you'll learn how to modify the content that appears at the top of search-results pages, and how to modify the `BODY` tag that sets the background characteristics for search-results pages.

To make the search results page use a similar heading and background as the browse page, you need to repeat the process you have just gone through.

Create the file `yourcompany-sr-top.pat` as a copy of `normal-sr-top.pat`.

Open the file `yourcompany-sr-top.pat` in a plain text editor. Change the value of the `BGCOLOR` attribute of the `BODY` tag to `#FFFFCF`, which is a slightly darker yellow than the background color used in the browse mode page. Also add a heading immediately below the `BODY` tag, as follows:

```
<BODY BGCOLOR="#FFFFCF" TEXT="#000000" LINK="#006666" VLINK="#999999">
<H1><FONT COLOR="#22DDCC">Welcome to the Search Page for My Company
(Search-Results-Page)</FONT</H1>
<BR>
<BR>
```

Save your changes. Edit the file `yourcompany.conf`, and set the value of the `RDM-search-results-top` variable to `yourcompany-sr-top.pat`, as shown:

```
RDM-search-results-top=yourcompany-sr-top.pat
```

Save the file and test your change by submitting a search in the Compass Server End User page. The page that shows the search results should have a yellow background and an introductory greeting.

Modifying the List of Subcategories in a Browsed Category

This section shows how to modify the way that categories are displayed during browsing. (To browse a category, click it in the list of categories in the End User page.) The changes will be simple; you will change the color of the subcategories, and display them as level 4 headings instead of bulleted list items.

First, create the pattern files for displaying the subcategories:

- Copy `normal-cb-top.pat` to `yourcompany-cb-top.pat`.
- Copy `normal-cb-hit.pat` to `yourcompany-cb-hit.pat`.
- Copy `normal-cb-bot.pat` to `yourcompany-cb-bot.pat`.

Modifying the Category-Browse-Top Pattern File

First, open the file `yourcompany-cb-top.pat` in a plain-text editor. The contents of this file will be a script that looks like:

```
<!-- normal-cb-top.pat -->
<SCRIPT LANGUAGE=JavaScript>

var subCategoryCount = 0;
var subCategoryNameArray = new Object();
var subCategoryIdArray = new Object();
var subCategoryCountArray = new Object();
if ($$rdm-hits-returned > 1) { // 1 just means itself, 2 or more is ok
  document.writeln('<FONT SIZE=-1 FACE="arial, helvetica">
  <B>$$category Subcategories:</B></FONT>');
}
</SCRIPT>
```

This script defines and initializes some variables, such as `subCategoryCount`. These variables are not used again in this script. They are used by other files in the normal template, but they will not be used again in the new template that you are developing. To keep your file tidy, delete the variable declaration statements.

The special variable `$$rdm-hits-returned` evaluates to the number of hits returned, which is essentially the number of sub-categories in the category being browsed. (This file is used only during browsing.) If the value is 1, it means that the only hit is the browsed category itself, so a value of 2 or more means there are subcategories.

The special variable `$$category` evaluates to the category that is currently being processed. (There is also another special variable, `$$browse-category`, which evaluates to the category being browsed. When you browse categories, `$$category` and `$$browse-category` both have the same value, since the category being processed is the category being browsed. However, the first time the End User page is opened, no categories are being processed, so `$$category` has no value.)

If there are any sub-categories, the script writes a line stating the name of the browsed category, followed by the word "Subcategories." For example, if the browsed category is Netscape, the script writes the line: "Netscape Subcategories."

Just to see the effect of making a change, edit the argument to `document.writeln` so that it prints its line in italics, and uses a more wordy introduction to the list of subcategories. For example, the introduction might say "Netscape: These are the subcategories: ".

There is one tricky issue here, and that is that the `$$category` variable does not have a value if the root category is being browsed. Thus before using `$$category` to write the name of the category, you need to check that `$$category` has a value. You can use the following special syntax:

`$$variable[`*`do this if $$var has value`*`][`*`else do this if $$var has no value`*`]`

Regardless of whether the root category or another category is being browsed, you can show a message that says "Here are the subcategories" if there are subcategories.

To increase the impact of the change, you can change the color of the message. Replace the entire `if` clause in the script by the following code:

```
if ($$rdm-hits-returned > 1) { // 1 just means itself, 2 or more is ok
  document.writeln("<FONT COLOR='#0055FF'><B>");
  // write category name if $$category has a value
  $$category[document.writeln("$$category: ");]
  document.writeln("These are the subcategories:</B></FONT>");
}
```

The following listing shows the full script with the changes:

```
<SCRIPT LANGUAGE=JavaScript>
  if ($$rdm-hits-returned > 1) { // 1 just means itself, 2 or more is ok
    document.writeln("<FONT COLOR='#0055FF'><B>");
    // write category name if $$category has a value
    $$category[document.writeln("$$category: ");]
    document.writeln("These are the subcategories:</B></FONT>");
}
</SCRIPT>
```

Hint. Don't forget that Javascript does not handle punctuations such as " or '. If you wish to use one of these punctuation characters, you need to put a "\" in front of it to escape it.

Save your changes.

Modifying the Category-Browse-Hit Pattern File

Open the file `yourcompany-cb-hit.pat` in the text editor. (If it doesn't exist, create it as a copy of `normal-cb-hit.pat`.) This file is called once for each sub-category.

The contents of the file is a script that looks something like:

```
<!-- Category Hit: $$rdm-hit -->
<SCRIPT LANGUAGE=JavaScript>
```

```
  // compute the amount of stuff under category and write in parens
  var cnt = 0;
  $$subclassification-count[cnt += $$subclassification-count;]
  $$categorized-count[cnt += $$categorized-count;]
  subCategoryNameArray[subCategoryCount] = "$$category";
  subCategoryCountArray[subCategoryCount] = cnt;
  subCategoryIdArray[subCategoryCount] = "$$preserve-id";
  subCategoryCount++;
</SCRIPT>
```

This pattern file accumulates data about each subcategory into arrays rather than writing it out directly. This is because the format used by the normal template to display subcategories depends on the number of sub-categories. If there are more than four subcategories, then they are displayed in a table, and if there are four or less, they are displayed in a bulleted list. We don't know how many subcategories there are until they have all been processed. Thus the `normal-cb-hit.pat` file does not write data, it simply accumulates it. The task of writing the data falls to `normal-cb-bot.pat`, which decides how to write the results based on how many subcategories were found.

However, to keep things simple in this tutorial, you will do away with the conditional criteria for displaying subcategories, and instead display them all as red level four headings, no matter how many there are.

The first thing to do is to delete the entire contents of the file `yourcompany-cb-hit.pat`. Before doing this, make absolutely sure you are editing `yourcompany-cb-hit.pat`, and not `normal-cb-hit.pat`.

After deleting the entire contents of the file, define an `<H4>` tag, followed by a `<FONT>` tag that sets the text color to red. Then display the category name (`$$category`) as a link that, when clicked, invokes the `browse()` helper function (which is written in JavaScript and defined in the Compass Server) to browse the category. The argument to `browse()` is the category's id, which is usually equivalent to its name.

```
<H4><FONT COLOR="red">
<A HREF="javascript:browse('$$id');">
  $$category
  </A>
```

Then define a script that checks whether the number of subcategories in this category (`$$subclassification-count`) is equal to zero. If it is not zero, then write the number of sub-categories in italics.

```
<SCRIPT LANGUAGE=JavaScript>
  if ($$subclassification-count != 0)
```

```
    document.writeln("(<I>" + count + "</I>)");
</SCRIPT>
```

Finally, close the `FONT` and `H4` tags.

```
</FONT></H4>
```

The following code shows the entire contents of the file after you have made the changes.

```
<H4><FONT COLOR="red">
<A HREF="javascript:browse('$$id');">
  $$category
</A>
<SCRIPT LANGUAGE=JavaScript>
  if ($$subclassification-count != 0)
    document.writeln("(<I>" + count + "</I>)");
</SCRIPT>
</FONT></H4>
```

Save the file.

Modifying the Category-Browse-Bottom Pattern File

Open the file `yourcompany-cb-bot.pat` in the text editor. (If the file doesn't exist, create it as a copy of `normal-cb-bot.pat`.)

This file contains a complex script. The upshot of the outcome of this script is that if there are more than 4 subcategories, they are displayed in a table. If there are 4 or less subcategories, they are displayed as bulleted lists. However, since we have changed the entire approach to displaying subcategories, you can delete the entire contents of this file.

Just for the sake of it, add a simple remark that will appear at the end of the list of subcategories, as shown:

```
<P><I>-- end of browsed categories -- </I></P>
```

Save your changes.

Updating the Configuration File

Edit the file `yourcompany.conf.` Set the value of the `RDM-category-browse-top` variable to `yourcompany-cb-top.pat`; set the value of the `RDM-category-browse-hit` variable to `yourcompany-cb-hit.pat`; and set the value of the `RDM-category-browse-bottom` variable to `yourcompany-cb-bot.pat` as shown here:

```
RDM-category-browse-top=yourcompany-cb-top.pat
RDM-category-browse-hit=yourcompany-cb-hit.pat
RDM-category-browse-bottom=yourcompany-cb-bot.pat
```

Save the configuration file and test your changes by browsing a category in the Compass Server End User page. You should see a message introducing the subcategories, and each subcategory should be listed as a level-four heading instead of a bullet. The number of subcategories in a category should appear in red. You probably expected the subcategory names to appear in red too. And in fact they would have appeared in red, except that they are active links, so they appear in the active link color.

Check the difference between browsing the root category and any other category. You should see that when you browse the root category, the category name is not displayed in the introductory message "The subcategories are:"

Now submit a search and see what happens. The resultant page does not use the new pattern files. So far you have defined pattern files that are used only by browse-mode pages (except for the top of the page). You need to change the pattern files used by search-results mode pages to change the appearance of pages displayed by submitting a search.

Modifying the List of Documents in a Browsed Category

You can modify the appearance of the list of documents that belong in a browsed category in much the same fashion that you modified the appearance of the list of subcategories. However, since documents do not have sub-documents, it is easier to modify document listings than it is to modify category listings.

In summary, the document-browse-top pattern file introduces the top of the browsed document list. The document-browse-hit pattern file is called to display each document in turn. The document-browse-bottom pattern file displays closing remarks at the end of the list. Thus to modify the introduction to the document list, you need to edit the document-browse-top pattern file. To modify the way each document is displayed, you need to make changes to the document-browse-hit pattern file. To modify the conclusion of the list, you need to modify the document-browse-bottom pattern file.

This section does not give step by step instructions for modifying the list of documents in a browsed category. The section "Modifying the List of Matching Documents" gives details for modifying the appearance of a document list.

Modifying the List of Matching Categories

This section shows how to modify the appearance of the list of documents that is returned by a search submission.

So far in this tutorial, you have modified the top section of both browse-mode and search-results mode pages. You have also modified the appearance of the list of subcategories in a browsed category.

When a user submits a search, the search-results list all categories and documents that matched the search criteria. The categories that matched the search criteria are known as *matching categories*.

You can modify the appearance of the list of matching categories in much the same way that you modified the appearance of the list of subcategories.

In summary, the category-match-top pattern file introduces the top of the list of matching categories. The category-match-hit pattern file is called to display each matching category in turn. The category-match-bottom pattern file displays closing remarks at the end of the list of categories. Thus to modify the introduction to the category list, you need to edit the category-match-top pattern file. To modify the way each category is displayed, make changes to the category-match-hit pattern file. To modify the conclusion of the list, edit the category-match-bottom pattern file.

This section does not give step by step instructions for modifying the list of matching categories, since the steps involved are very similar to those needed for modifying the list of subcategories, as discussed in Modifying the List of Subcategories in a Browsed Category.

Modifying the List of Matching Documents

This section gives instructions for modifying the list of documents that is displayed when you submit a search. These documents are known as *matching documents* since they match the search criteria.

The document-match-top pattern file introduces the list of matching documents. The document-match-hit pattern file defines the appearance of each document in the list. This file is invoked for each matching document in turn. The document-match-bottom pattern file defines the appearance of the bottom of the matching documents list.

First, create the pattern files for displaying the matching documents:

- Copy `normal-dm-top.pat` to `yourcompany-dm-top.pat`.
- Copy `normal-dm-hit.pat` to `yourcompany-dm-hit.pat`.
- Copy `normal-dm-bot.pat` to `yourcompany-dm-bot.pat`.

Modifying the Document-Match-Top Pattern File

In this tutorial, the list of documents will be displayed in a table to make the results look neat and tidy. Each document will be displayed in its own row. Since only one table is needed to contain all the document results, you can start the table in the document-browse-top pattern file, which defines the content that precedes the list of matching documents.

Open the file `yourcompany-dm-top.pat` in a plain text editor. (If the file does not exist, create it by copying the file `normal-dm-top.pat`.)

This file checks how many hits were returned. If there are more hits than will fit in a page, it adds the **more** link. You can modify this file so that it starts the table that displays the matching documents.

At the very end of the file, add a <TABLE> tag, and give the table a border and a white background. The following listing shows the entire contents of the file `yourcompany-dm-top.pat` after you make these changes.

```
<FONT COLOR="006666" SIZE=-1 FACE="arial,helvetica">
<SCRIPT LANGUAGE=JavaScript>
if ($$rdm-hit-min > 0 && $$rdm-hits-returned > 0) {// some hits
document.writeln("Document Matches $$rdm-hit-min - $$rdm-hit-max
(of $$rdm-hits-available)");
} else { // no hits
  if ($$page > 1) {// nth page
   document.writeln("No Further Document Matches");
  } else {// first page
   document.writeln("No Document Matches");
  }
}
</SCRIPT>
```

```
<BR>
<TABLE WIDTH="100%" CELLPADDING=2 CELLSPACING=0
  BGCOLOR=white BORDER=1>
```

Defining a List of Alternating Colors

You can use the special variable `$$RDM-rotated-color` inside hit pattern files to rotate through colors in a pre-defined list. You will use this functionality when displaying the list of matching documents.

First, you need to define the list of colors. Open the file `yourcompany.conf`, and search for `RDM-rotated-colors`. Comment out the existing variable binding, and create your own list of colors. You can have as many colors as you like in the list. The following statement sets the list to be a set of two colors including pale green and pale blue.

```
RDM-rotated-colors="#DDDDFF,#DDFFDD"
```

It is important that there are no spaces in the list, not even after the comma that separates the list items.

Save the changes in the configuration file.

Modifying the Document-Match-Hit Pattern File

Open the file `yourcompany-dm-hit.pat` in a plain text editor. (If the file does not exist, create it by copying the file `normal-dm-hit.pat`.)

Inside a document-match-hit pattern file, you can use several special variables to refer to information about the document. These variables include the following (this is not a complete list):

$$title The title of the document.

$$description The description of the document currently being processed.

$$URL The URL of the document being processed.

$$score The score for how well the current document matched the search criteria.

$$last-modified The date the document was last modified.

$$classification Which category the document is in.

$$content-length The length in bytes of the document.

$$author The document's author, if known.

$$RDM--is-first-hit True if this is the first document in the list, otherwise this variable has no value.

$$RDM-hit The number of the current document in the list of matching documents.

$$RDM-rotated-color The next color in the list of colors defined by the variable `RDM-rotated-colors` in the configuration file. You can use this variable to rotate through a list of colors.

Instead of modifying the current content of this file, you will write new content from scratch. Each document will be displayed in a table row, and the rows will use alternating colors.

The goal is to display each document in a row in the table. Each row contains two cells. The first cell shows an image indicating the score ranking of the document. (The images are defined in the configuration file.) The second cell displays information about the document, such as its title, its description, its URL, its author, its length, and when it was last modified. The following figure shows what the document list might look like:

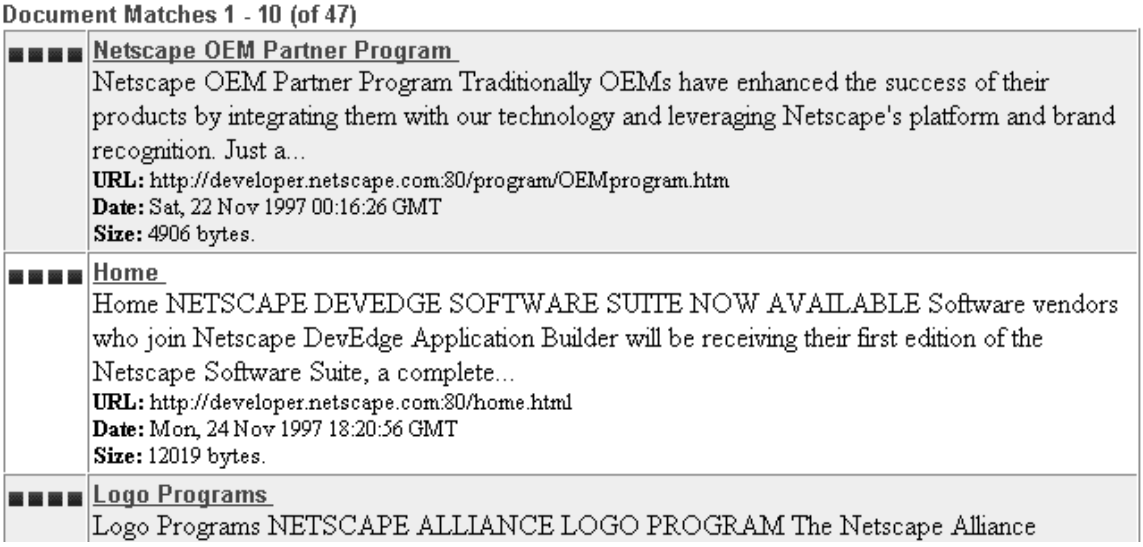

Figure 6.1 A table showing a list of matching documents

Delete the entire content of the file `yourcompany-dm-hit.pat.` (Before you do this, make sure you are really editing `yourcompany-dm-hit.pat`, and not the original `normal-dm-hit.pat`.)

As the first thing in the file, start the table row for this document. Set the row's background to `$$rdm-rotated-color` so that it automatically uses the next color in the color list.

```
<!-- yourcompany-dm-hit.pat -->
<!-- use table rows for nice clean layout -->
<TR BGCOLOR=$$RDM-ROTATED-COLOR>
```

Write the table cell that shows the score icon. When the user moves the mouse over the icon, the status message displays the numerical score value for this document.

```
  <!-- present the score here - use image with nice mouseover -->
  <TD VALIGN=top ALIGN=right WIDTH="7%">
<A HREF="javascript:void(0);"
  <!-- no line break allowed in onMouseOver value -->
  onMouseOver="self.status='$$score% Relevant';return true;">
  $$RDM-Score-Image</A>
  </TD>
```

Write the cell that displays the document details. Create a link. Inside the link, display the document's title if it has one, (that is, if `$$title` has a value) otherwise display the URL. When the user clicks the link, the URL will open.

```
  <!-- present the document details here -->
  <TD VALIGN=top ALIGN=left WIDTH="93%">
  <FONT SIZE="-1" FACE="arial, helvetica">
  <A HREF="$$url"><B>$$title[$$title][$$url]</B>
  </A></FONT><BR>
```

Display the description, if known.

```
  $$description[$$description<BR>]
```

Display the URL.

```
  <FONT SIZE=-1>
  <B>URL:</B> $$url<BR>
```

Display the date when the file was last modified, if known. Also display the length of the document and the author, if known.

```
  $$Last-Modified[<B>Date:</B> $$Last-Modified<BR>]
  $$Content-Length[<B>Size:</B> $$Content-Length bytes<BR>]
  $$Author[<B>Author:</B> $$Author]
```

If the document has a classification (that is, if it is in a category), display a link that displays all the documents in that category. Notice that this code uses the helper function `browse()`.

```
  $$classification[
  <A HREF="javascript:browse(unescape('$$encode-classification'));">
  Find similarly categorized documents.</A>.]
  <!-- End the cell and end the row. -->
  </FONT>
  </TD>
</TR>
```

Save your changes. The following listing shows the entire contents of the file `yourcompany-dm-hit.pat` with the changes.

```
<!-- yourcompany-dm-hit.pat -->
<!-- use table rows for nice clean layout -->
<TR BGCOLOR=$$RDM-ROTATED-COLOR>
  <!-- present the score here - use image with nice mouseover -->
  <TD VALIGN=top ALIGN=right WIDTH="7%">
    <A HREF="javascript:void(0);"
      onMouseOver="self.status='$$score% Relevant';return true;">
    $$RDM-Score-Image</A>
  </TD>
  <!-- present the document details here -->
  <TD VALIGN=top ALIGN=left WIDTH="93%">
  <FONT SIZE="-1" FACE="arial, helvetica">
  <A HREF="$$url"><B>$$title[$$title][$$url]</B></A></FONT><BR>
  $$description[$$description<BR>]
  <FONT SIZE=-1>
  <B>URL:</B> $$url<BR>
  $$Last-Modified[<B>Date:</B> $$Last-Modified<BR>]
  $$Content-Length[<B>Size:</B> $$Content-Length bytes<BR>]
  $$Author[<B>Author:</B> $$Author]
  $$classification[
  <A HREF="javascript:browse(unescape('$$encode-classification'));">
  Find similarly categorized documents.</A>.]
  </FONT>
  </TD>
</TR>
```

Modifying the Document-Match-Bottom Pattern File

You may have noticed that the `TABLE` tag defined in the file `yourcompany-dm-top.pat` has not yet been closed. But don't worry, it will be closed in the file `yourcompany-dm-bot.pat`.

Open the file `yourcompany-dm-bot.pat` in a plain text editor. (If the file does not exist, create it by copying the file `normal-dm-bot.pat`.)

This file is very simple, all it does is close the table and add a break, as shown here:

```
</TABLE>
<BR>
```

Save your changes

Updating the Configuration File

Edit the file `yourcompany.conf`. Set the value of the `RDM-document-match-top` variable to `yourcompany-dm-top.pat`; set the value of the `RDM-document-match-hit` variable to `yourcompany-dm-hit.pat`; and set the value of the `RDM-document-match-bottom` variable to `yourcompany-dm-bot.pat` as shown here:

```
RDM-document-match-top=yourcompany-dm-top.pat
RDM-document-match-hit=yourcompany-dm-hit.pat
RDM-document-match-bottom=yourcompany-dm-bot.pat
```

Save the configuration file and test your changes by submitting a search in the Compass Server End User page. The documents returned by the search process should be displayed in a table. Each row should display the details of one document, and the rows should have alternating background colors.

Hint: If the results do not appear, check that the file `yourcompany-sr-bot.pat` contains the closing table tag `</TABLE>`. Also check in the configuration file that `RDM-document-match-bottom` is set to `yourcompany-dm-bot.pat`. If the table is not closed, none of the rows will be displayed.

Changing the End of The Page

The final section at the bottom of a browse mode page is defined by the browse-bottom pattern file. The final section at the bottom of a search-results mode page is defined by the search-results-bottom pattern file.

If you would like to add a final section to the browse mode page, edit the file `yourcompany-bw-bot.pat` as you see fit. If you would like to add a final section to the bottom of a search-results page, edit the file `yourcompany-sr-bot.pat` as you see fit. Be sure to edit the file `yourcompany.conf` and set the following variables:

```
RDM-browse-bottom=yourcompany-bw-bot.pat
RDM-search-results-bottom=yourcomapny-sr-bot.pat
```

Finishing Up

Hopefully working through this tutorial has helped you understand the process involved in modifying configuration and pattern files, and you are now ready to modify them in whatever way you like.

When you have completely finished customizing the configuration and pattern files, be sure to set the `template-refresh-rate` variable in the file `csid.conf` back to a high value, such as 3600. This file lives in the `config` directory of the directory containing the specific Compass Server.

The `template-refresh-rate` variable determines the interval in seconds between each time the Compass Server gets the latest versions of the configuration and pattern files it needs. If the value has a low value, such as 1, then the Compass Server will spend a lot of its time unnecessarily refreshing the template files in its cache.

Chapter

7

Error Message Files

You can edit the error messages that the Compass Server generates when an attempt to access the server is unsuccesful.server generates.

The original files are in:

compass-installdir/bin/compass/docs/

You can put your images in:

compass-installdir/bin/compass/images

and reference them with

```
src="/images/etc.gif"
```

Here is a list of the kinds of errors, and the files to edit to modify the error message.

Cause of error:

No access to the Compass Server

File to edit:

compass-installdir/bin/compass/docs/nologin

Cause of error:

File not found (404 not found):

File to edit:

compass-installdir/compass-*name*/docs/notfound.html

Cause of error:

Server error

File to edit:

compass-installdir/compass-*name*/docs/server_error.html

Cause of error:

Access is forbidden

File to edit:

compass-installdir/compass-*name*/docs/forbidden.html

Cause of error:

Unauthorized access

File to edit:

compass-installdir/compass-*name*/docs/unauthorized.html

2

Customizing Robot Behavior

This part of the document describes how the Netscape Compass Server robots, and how to modify their behavior.

The Compass Server robot is responsible for identifying resources on a network to store in the Compass Server database as resource descriptions. You can run the robot and customize most aspects of its behavior interactively through the **Robot** page of the Compass Server Administration Interface. The *Netscape Compass Server Administrator's Guide* (available through the **Manuals** button in the Administrator Interface) describes how to use the Compass Server Administration Interface.

For more complex customizations, however, you need to program the robot directly, by either modifying its configuration files, writing robot application functions, or perhaps both.

Many aspects of customizing Compass Server robots are similar to programming Netscape web servers through the Netscape Server API (NSAPI). If you are not already acquainted with Server Application Functions (SAFs), parameter blocks, and other basic aspects of NSAPI, you should consult the *Netscape NSAPI Programmer's Guide* before continuing with this guide.

You can find the *NSAPI Programmer's Guide* at:

`http://developer.netscape.com/library/documentation/enterprise/nsapi/index.htm`

This part includes chapters that provide reference information on filtering functions and the API for writing your own robot application functions. The chapters are:

Chapter 8. How Netscape Compass Server Robots Work

Overview of how the Compass Server robots operate.

Chapter 9. Defining Parameters in Process.conf

This chapter lists the process parameters you can configure to modify robot behavior.

Chapter 10. The Pre-defined Robot Application Functions

This chapter lists the pre-defined robot application functions you can use to modify robot filtering behavior.

Chapter 11. Creating New Robot Application Functions

This chapter discusses how to define new robot application functions to extend robot filtering behavior.

Chapter

8

How Netscape Compass Server Robots Work

This chapter describes how the Compass Server robot works, and discusses the configuration files that it uses.

What is a Robot?

A robot is an agent, a piece of software, that can traverse segments of a network generating resource descriptions as it locates and identifies appropriate resources.

The Netscape Compass Server robot uses two kinds of filters: an enumerator filter and a generator filter.

- The enumerator filter searches out and locates, or discovers, resources using network protocols and mechanisms. The enumerator tests each resource to see if it should be enumerated, and if so, goes ahead and enumerates it. For example, the enumerator filter can extract hypertext links from an HTML file, and use the links to find additional resources.
- The generator filter tests each resource to see whether or not to create an resource description (RD) for it to save in the Compass Server database. If the resource passes the test, the generator goes ahead and generates an RD for it.

How The Robot Works

Figure 8.1 illustrates how the Netscape Compass Server robot works.

Starting with the seed URLs, the robot examines URLs and their associated network resources. Each resource is tested by both the enumerator and the generator. If the resource passes the enumeration test, the robot enumerates it, that is checks it for more URLs. If the resource passes the generator test, the robot generates a resource description for it to put in the Compass Server database.

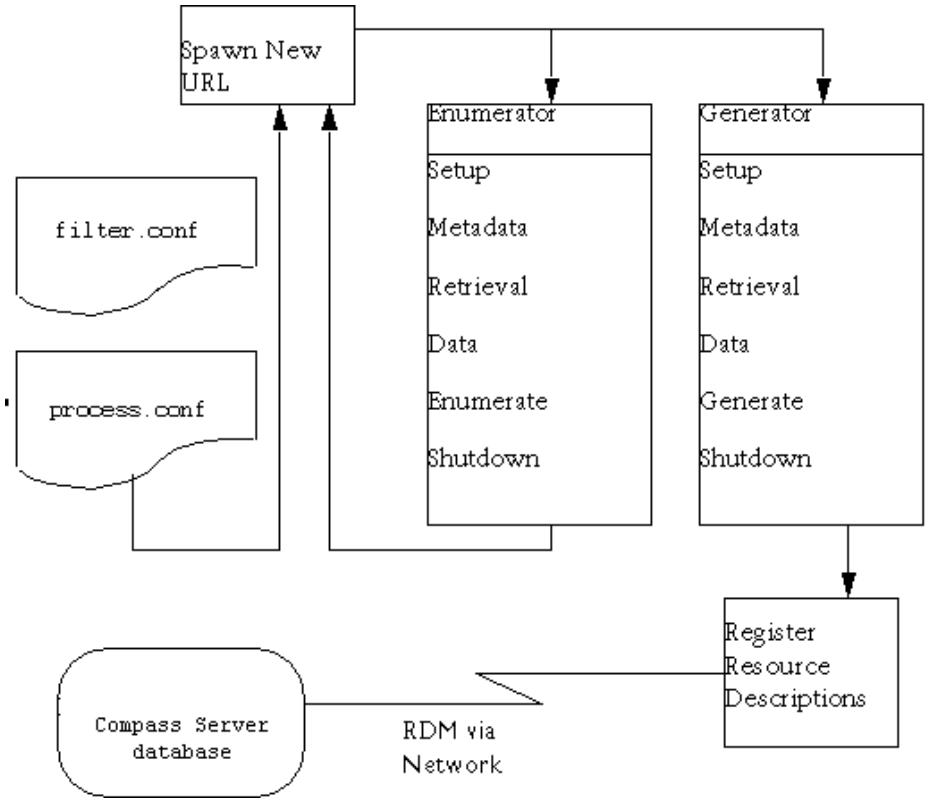

Figure 8.1 How the Robot Works

Files that Define Robot Behavior

Three robot configuration files, `process.conf`, `filter.conf`, and `filterrules.conf` control the behavior of the Compass Server robots. These files live in the directory *`compass-installdir/compass-name/`*`config`. For example, if you installed Compass Server in `suitespot`, and you created a Compass Server instance named `nikki`, the directory would be `suitespot/compass-nikki/config`

- `process.conf` defines most of the operating parameters for the robot, including telling it which of the filters from `filter.conf` to use.
- `filter.conf` contains all the filters available to the Compass Server robot for both enumeration and generation. It also includes by reference the filtering rules stored in `filterrules.conf`. Including the same filtering rules for both the enumeration filters and the generation filters ensures that a single rule change affects both filters.
- `filterrules.conf` contains the starting points, or seed URLS, as well as the rules to be used for filtering.

Each of these configuration files is a plain text file that you can edit with any standard text editor.

Setting Robot Process Parameters

The file `process.conf` defines many options for the robot, including telling it which of the filters from `filter.conf` to use.(For backwards-compatibility with the Catalog Server, `process.conf` can also contain the starting points.)

In general, you do not need to edit the file `process.conf`. It is written by the Compass Server when you make changes in the Robot page of the Compass Server Administration Interface. However, there are a few parameters that you might want to manually edit. These parameters are discussed in Chapter 9, “Defining Parameters in Process.conf”.

The two most important process parameters you can set by editing `process.conf` directly are `enumeration-filter` and `generation-filter`. These parameters determine which filters the robot uses for

enumeration and generation. The default values for these are `enumeration-default` and `generation-default`, which are the names of the filters provided by default in `filter.conf` file.

All filters must be defined in the file `filter.conf.` If you define your own filters in `filter.conf`, you must add the corresponding parameters to `process.conf`.

For example, if you define a new enumeration filter named `my-enumerator`, you would add the following parameter to `process.conf`:

```
enumeration-filter=my-enumerator
```

The Filtering Process

Robots use filters to control which resources to process and how to process them. When the robot discovers references to resources (and resources themselves), it applies filters to each one to enumerate it (that is, examine it for more resources) and to determine whether or not to generate a resource description for it to put in the Compass Server database.

The robot starts by examining one or more starting points or seed URLs and applying the filters to each one, and then applying the filters to the URLS spawned by enumerating the seed URLS, and so on. (The seed URLS are defined in the `filterrules.conf` file.)

A filter starts by performing any required initialization operations. Then it applies comparison tests to the current resource. The goal of each test is to either allow or deny the resource. A filter also has a shutdown phase during which it performs any required cleanup operations.

If a resource is allowed, that means that it is allowed to continue passage through the filter. If a resource is denied, then the resource is rejected. No further action is taken by the filter for resources that are denied. If a resource is not denied, the robot will eventually enumerate it, attempting to discover further resources. The generator might also create a resource description for it.

Note that these operations are not necessarily linked. Some resources result in enumeration; others result in RD generation. Many result in both. For example, if the resource is an FTP directory, that resource will probably not have an RD generated for it. However, the robot might well enumerate the individual files

in the FTP directory. An HTML document that contains links to other documents will probably get an RD for itself and lead to enumeration of the linked documents as well.

Stages in the Filter Process

Both enumerator and generator filters each have five phases in the filtering process. They both have four common phases, Setup, Metadata, Data, and Shutdown. If the resource makes it past the Data phase, there is either an Enumerate or Generate phase, depending on whether the filter is an enumerator or a generator.

Setup

Performs initialization operations. Occurs only once in the life of the robot.

Metadata

Filters the resource based on metadata that is available *about* the resource. Metadata filtering occurs once per resource before the resource is retrieved over the network.

Examples of metadata include the following:

Metadata	Meaning	Example
Complete URL	The location of a resource	http://home.netscape.com/
Protocol	The access portion of the URL	http, ftp, file
Host	The address portion of the URL	www.netscape.com:1080
IP address	Numeric version of the host	198.95.249.6:8000
URI	The path portion of the URL	/index.html
Depth	Number of links from the seed URL	5

Data

Filters the resource based on data contained by the resource. Data filtering is done once per resource after it is retrieved over the network. Some data that can be used for filtering follow:

- content-type
- content-length
- content-encoding
- content-charset
- last-modified
- expires

Enumerate

Enumerates the current resource to find if it points to other resources to be examined.

Generate

Generates a resource description (RD) for the resource, and saves the RD in the Compass Server database.

Shutdown

Performs any needed termination operations. Occurs only once in the life of the robot.

Filter Syntax

The `filter.conf` file contains definitions for Enumeration and Generation filters. This file can contain multiple filters for both enumeration and generation. (The robot knows which ones to use because they are specified by the `enumeration-filter` and `generation-filter` parameters in the file `process.conf.`)

Filter definitions have a well-defined structure: a header, a body, and an end. The header identifies the beginning of the filter and declares its name, for example:

```
<Filter name="myFilter">
```

The body consists of a series of *filter directives* that define the filter's behavior during setup, testing, enumeration or generation, and shutdown. Each directive specifies a function, and if applicable, parameters for the function.

The end is marked by `</Filter>`. The following code shows an example filter named `enumeration1`.

```
<Filter name="enumeration1>
   Setup fn=filterrules-setup config=./config/filterrules.conf

   #  Process the rules
   MetaData fn=filterrules-process

   #  Filter by type and process rules again
   Data fn=assign-source dst=type src=content-type
   Data fn=filterrules-process


   #  Perform the enumeration on HTML only
   Enumerate enable=true fn=enumerate-urls max=1024 type=text/html

   #  Cleanup
   Shutdown fn=filterrules-shutdown
</Filter>
```

Filter Directives

Filter directives use Robot Application Functions (RAFs) to perform their operations. Their usage and flow of execution is similar to that of NSAPI directives and Server Application Functions (SAFs) in the file `obj.conf`. Data is stored and transferred using parameter blocks, also called *pblocks*. If you are not already acquainted with SAFs and pblocks and other basic aspects of NSAPI, you should consult the NSAPI Programmer's Guide before continuing with this material.

You can find the *NSAPI Programmer's Guide* at:

`http://developer.netscape.com/library/documentation/enterprise/nsapi/index.htm`

There are six robot directives, or RAF classes, corresponding to the filtering phases and operations listed in "The Filtering Process" on page 88:

- Setup
- Metadata
- Data
- Enumerate
- Generate
- Shutdown

Each directive has its own particular robot application functions. For example, you use filtering functions with the Metadata and Data directives, enumeration functions with the Enumerate directive, generation functions with the Generate directive, and so on.

The built-in robot application functions are listed and explained in Chapter 10, Chapter 2, "The Pre-defined Robot Application Functions." You can also write your own robot application functions, as described in Chapter 11, Chapter 3, "The Pre-defined Robot Application Functions."

Writing or Modifying a Filter

In most cases, you should not need to write filters from scratch. You can create most of your filters using the **Robot** page in the Compass Server Administration Interface. You can then modify the `filter.conf` and `filterrules.conf` files to make any desired changes. These files live in the directory *`compass-installdir/compass-name/`*`config`.

However, if you want to create a more complex set of parameters than is supported by the Robot page in the interface, you will need to edit the configuration files used by the robot.

In most cases, you can probably see what you need to do by looking at the existing configuration files and the examples in Chapter 11, Chapter 3, "The Pre-defined Robot Application Functions. You will need to keep the following points in mind when writing or modifying a filter:

- The order of execution of directives (especially what information is available at each phase) is important.
- The order that rules appear in the file is significant.

See the Chapter 9, Defining Parameters in Process.conf, for a discussion of the parameters you can modify in the file `process.conf`, and see the Chapter 10, The Pre-defined Robot Application Functions for a discussion of the robot application functions that you can use in the file `filter.conf`. See Chapter 11, Chapter 3, "The Pre-defined Robot Application Functions for a discussion of how to create your own robot application functions.

Chapter

9

Defining Parameters in Process.conf

The file `process.conf` defines many options for the robot, including telling it which of the filters from `filter.conf` to use.(For backwards-compatibility with the Catalog Server, `process.conf` can also contain the starting points.)

In general, you do not need to edit the `process.conf` file directly. You can set most parameters by using the interactive options in the **Robot** page in the Compass Server Administration Interface.

However, advanced users may want to edit this file manually to set parameters that cannot be set through the interface.

This chapter lists the parameters that you can set in `process.conf`.

User-Modifiable Parameters in Process.conf

auto-proxy

```
auto-proxy="http://punk.mcom.com:80/"
```

This is the proxy setting for the robot. It can be a Netscape proxy server or a Javascript file for automatically configuring the proxy. For more information see:

```
http://home.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html
```

bindir

`bindir=path`

If you specify `bindir` then the robot will add it to PATH environment. This is an extra PATH for users to run external program in a robot, such as those specified by **cmd-hook** parameter.

cmd-hook

`cmd-hook="command-string"`

There is no default.

This specifies an external completion script to run after the robot completes one run. This must be a full path to the command name. The robot will execute this script from the `compass-`*`name/`* `directory`.

There must be at least one RD registered for the command to run.

See How To Write Completion Scripts for information about writing completion scripts.

command-port

`command-port=port_number`

The socket that the robot listens at for accepting command from other programs, such as the Administration Interface or robot control panels.

For security reasons, the robot can accept commands only from the local host unless `remote-access` is set to `yes`.

connect-timeout

`connect-timeout=seconds`

The default is 120.

This is the maximum time to allow for a network to respond to a connection request.

convert-timeout

`convert-timeout=seconds`

The default is 600 seconds.

This is the maximum time to allow for document conversion.

depth

```
depth=integer
```

The default is 20. This is the number of links from the starting point (seed URLs) that the robot should examine. This parameter sets the default value for seed URLs that do not specify a depth.

A value of negative one (`depth=-1`) means the link depth is infinite.

email

```
email=user@hostname
```

The default is `user@domain`

This is the email address of the person who runs the robot.

This is sent out along with user-agent in the http request header, so that web managers can contact the people who run robots at their sites.

enable-ip

```
enable-ip=[true | yes | false | no]
```

The default is `true`.

Generates an IP address for the url in each RD that is created.

enable-rdm-probe

```
enable-rdm-probe=[true | false | yes | no]
```

The default is yes.

This determines whether or not the robot queries each server it encounters to find out if the server supports RDM. If the server supports RDM, the robot will not attempt to enumerate the server's resources, as that server can act as its own resource description server.

enable-robots-txt

`enable-robots-txt=[true | false | yes | no]`

The default is yes.

This variable determines whether or not the robot checks the `robots.txt` file (if available) at each site it visits.

engine-concurrent

`engine-concurrent=[1..100]`

The default is 10.

The number of pre-created threads for the robot to use.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

enumeration-filter

`enumeration-filter=enumfiltername`

The default is `enumeration-default`.

This specifies the enumeration filter that the robot uses to determine whether or not to enumerate a resource. The value must be the name of a filter defined in the file `filter.conf`.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

generation-filter

`generation-filter=genfiltername`

The default is `generation-default`.

This specifies the generation filter that the robot uses to determine whether or not to generate a resource description for a resource. The value must be the name of a filter defined in the file `filter.conf`.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

index-after-ngenerated

```
index-after-ngenerated=30
```

This tells the robot how many minutes to collect RDs for before batching them to the Compass Server.

If you do not specify this parameter, it is set to 256 minutes.

loglevel

```
loglevel=[0...100]
```

The default value is 2.

The values mean:

Level 0: log nothing but serious errors
Level 1: also log generate and enumerate activity
Level 2: also log retrieval activity (DEFAULT)
Level 3: also log filtering activity
Level 4: also log spawning activity
Level 5: also log retrieval progress

max-concurrent

```
max-concurrent=[1..100]
```

The default is 8.

The maximum number of concurrent retrievals that a robot can make.

max-filesize-kb

```
max-filesize-kb=1024
```

The maximum file size in kilobytes for files retrieved by the robot.

max-memory-per-url / max-memory

```
max-memory-per-url=n_bytes
```

The default is 1.

Maximum memory in bytes used by each url. If the URL needs more memory, the RD is saved to disk.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

max-working

```
max-working=1024
```

The size of the robot working set, which is the maximum number of URLS the robot can work on at one time.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

onCompletion

```
OnCompletion=[idle | loop | quit]
```

The default is `idle`.

This determines what the robot does after it has completed a run. The robot can either go into idle mode, loop back and start again, or quit.

This parameter works with the **cmd-hook** parameter. When the robot is `done`, it will do the action of **onCompletion** and then run the **cmd-hook** program

password

```
password=string
```

The default is `netscape@.`

This is used for httpd authentication and ftp connection.

referer

```
referer=string
```

This is sent in the http-request if it is set to identify the robot as the referer when accessing web pages.

register-user and register-password

```
register-user=string
```

```
register-password=string
```

These are the username and password used for registering RDs to the Compass Server database.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

remote-access

```
remote-access=[true | false | yes | no]
```

The default is no.

This determines whether or not the robot can accept commands from remote hosts.

robot-statedir

```
robot-statedir="/newport/robot/state"
```

This specifies the directory where the robot saves its state. The robot uses this as a working directory for saving its internal state, including recording how many RDs have been collected, and so on.

robots-txt-refresh-rate

```
robots-txt-refresh-rate=seconds
```

The default is 3600*24.

This is the number of seconds the robot must wait before reading a `robots.txt` file again.

This parameter is cannot be set interactively through the Compass Server Administration Interface.

schema-name

```
schema-name=schema
```

The default value is DOCUMENT.

This is the name for a schema

This parameter is cannot be set interactively through the Compass Server Administration Interface.

server-delay

```
server-delay=delay_in_seconds
```

The time period between two visits to the same web site. This prevents the robot from hitting the same site too frequently.

smart-host-heuristics

```
smart-host-heuristics=[true | false]
```

The default is `true`.

This enables the robot to work out those sites rotating their DNS canonical hostnames. for example, it turns www123.netscape.com to www.netscape.com.

tmpdir

```
tmpdir=path
```

Specifies a place for the robot to create temporary files.

Use this value to set the environment variable TMPDIR (and TMP for NT).

user-agent

```
user-agent=Netscape-Compass-Robot/3.0
```

This is sent with the email address in the http-request to the server.

username

```
username=string
```

The default is `anonymous`.

This is the username of the user who runs the robot. This is used for httpd authentication and ftp connection.

Sample process.conf File

Note that this sample file includes some parameters used by the Compass Server Administration Interface that you should not modify.

Here is a sample `process.conf file`. The parameters that are commented out in this case use the default values shown. The first parameter, `csid`, indicates the Compass Server instance that uses this file. Do not change the value of the `csid` parameter.

```
<Process csid="x-catalog://boots.nikki.com:80/jack" \
   auto-proxy="http://punk.mcom.com:80/"
   auto_serv="http://punk.mcom.com:80/"
   command-port=21445
   convert-timeout=600
   depth="-1"
   # email="user@domain"
   enable-ip=true
   enumeration-filter="enumeration-default"
   generation-filter="generation-default"
   index-after-ngenerated=30
   loglevel=2
   max-concurrent=8
   onCompletion=idle
   password=boots
   proxy-loc=server
   proxy-type=auto
   robot-state-dir="/export/home/suitespot/compass-newport/robot/state"
   server-delay=1
   smart-host-heuristics=true
   tmpdir="/export/home/suitespot/compass-newport/tmp"
   user-agent="Netscape-Compass-Robot/3.0"
   username=nikki
</Process>
```

How To Write Completion Scripts

The **cmdHook** parameter specifies a program to execute after the robot completes one run.

The **cmdHook** is provided as a way for you to extend the shutdown phase of the robot. Perhaps you want the robot to send email when it's done, start another process, analyze it's own log files and write a small report, and so on.

When the robot is 'done' with one run (that is: it has no more entries in the enumeration-pool and has finished all outstanding processing) it will call the executable specified in the **cmdHook** setting. If the **onCompletion** parameter is set to `idle` or `quit`, the script is called once before the robot shuts down or goes idle. If the parameter is set to `loop`, the script is called each time the robot restarts. See the log samples in "Monitoring the cmdHook Execution".

The **cmdHook** script can be written in any language (as a Perl or shell script, a C program, and so on). If you choose to use a C program, you'll have to insert the **cmdHook** parameter in the `process.conf` file manually, as the Compass Server Administration Interface does not scan binary executables. See the notes in "Preparing Your Completion Script to Appear in the Administration Interface."

The **cmdHook** script is run from the robot's execution environment. This means your script will inherit any environment variables set by the robot and will not have access to environment variables that might be set by your compass server or the admin server. The **cmdHook** script will be executed from *`compass-installdir`*`/compass-`*`name/`* instead of `bin/compass/admin/bin`. This is a common source of errors and is important to keep in mind if you are using any relative directory references, (for example, perl is located in "`../install/perl`" instead of "`../../../../install/perl`" and so on.

There are two UNIX-only examples provided in `bin/compass/admin/bin`. They are a simple 'touch' test (cmdHook0) and an email upon completion script (cmdHook1).

Monitoring the cmdHook Execution

At the default logging level, a log message is written in robot.log when the **cmdHook** is run.

For example, if the **onCompletion** parameter is set to `idle`, the robot.log output would look like:

```
[12:45:57]  Run cmd: /usr/suitespot/compass-test/bin/compass/admin/bin/
cmdHook1
[12:45:58]  Complete cmd: /usr/suitespot/compass-test/bin/compass/
admin/bin/cmdHook1
[12:45:58]  Robot is idle....
```

...if the **onCompletion** parameter is set to `shutdown`, the robot.log output

would look like:

```
[12:45:57]  Run cmd: /usr/suitespot/compass-test/bin/compass/admin/bin/
cmdHook1
[12:45:58]  Complete cmd: /usr/suitespot/compass-test/bin/compass/
admin/bin/cmdHook1
[12:46:33]  Workload complete.
```

...if the **onCompetion** parameter is set to `loop`, the robot.log output would look like:

```
[12:52:04]  Run cmd: /usr/suitespot/compass-test/bin/compass/admin/bin/
cmdHook1
[12:52:05] Complete cmd: /usr/suitespot/compass-test/bin/compass/admin/
bin/cmdHook1
[12:52:05] Restart Robot.
```

...if the **onCompetion** parameter is set to `loop`, there will be an additional entry in `filter.log` like this:

```
[12:54:41] Filter log started - loop 15
```

Preparing Your Completion Script to Appear in the Administration Interface

If you want your **cmdHook** script to show up as an option on the Compass Server Administration in the "**Robot** -> Crawling Settings" page, you should do the following:

1. Write your script in a run-time evaluated language.That is, don't use C or another compiled language. The Compass Server Administration Interface does not scan binary files when it looks for **cmdHook** scripts.

2. Place the file in `bin/compass/admin/bin.`

3. Name the file **cmdHook**, followed by an alphanumeric character-string, for example, **cmdHook0**, **cmdHookAlpha**, **cmdHook12a**

4. Place a description string in the file using the following format:

   ```
   #description="My menu choice string"
   ```

   The description assignment should be placed in a comment so it doesn't effect the execution of the script.

Since many scripts are platform specific, if the description contains the string "(non NT)", the Compass Server Administration Interface will not list that script as an option on NT platforms. This is not particularly useful for a specific instance of a Compass Server. However, if your script will be redistributed across a wide variety of platforms it is something to keep in mind.

Chapter

10

The Pre-defined Robot Application Functions

This chapter describes the pre-defined Netscape Robot Application Functions (RAFs), providing descriptions, parameter specifications, and examples of each one. You can use these functions in the `filter.conf` file to create and modify filter definitions. The file `filter.conf` lives in the directory *`compass-installdir`*`/compass-`*`name`*`/config`.

The file `filter.conf` contains definitions for the enumeration filter and the generation filter. Each of these filters can invoke a set of filter rules, which are stored in the file `filterrules.conf`. The general idea is that the filter definitions contain instructions that are specific to each filter, while the filter rules contain the rules used by both filters.

If you are interested, you can examine the file `filterrules.conf` to see how filter rules are defined. You should not need to manually edit this file, however, since you can create filter rules interactively by using the **Robot** page in the Compass Server Administration Interface.

You can examine the file `filter.conf` to see an example of filter definitions. You only need to edit the `filter.conf` file manually if you want to modify the filters in a way that is not accommodated in the interface, such as instructing the robot to enumerate some resources without generating resources for them.

This chapter describes each of the basic categories of functions:

- Setup Functions - these are used during set up.

- Filtering Functions - these can be used by both enumeration and generation filters.
- Filtering Support Functions - these can be used by both enumeration and generation filters.
- Enumeration Functions - these can be used by the enumeration filter.
- Generation Functions - these can be used by the generation filter.
- Shutdown Functions - these are used after the filtering and processing has finished.

Sources and Destinations

Most of the Robot Application Functions (RAFs) require *sources* of information and generate data that goes to *destinations*. The sources are defined within the robot itself, and are not necessarily related to the fields in the resource description it ultimately generates. Destinations, on the other hand, are generally the names of fields in the resource description, as defined by the resource description server's schema.

For details of using the Compass Server Administration Interface to determine the database schema, see the *Compass Server Administrator's Guide* (which you can get by pressing the **Manuals** button in the Compass Server Administration Interface).

Sources Available at the Setup Stage

At the Setup stage, the filter is busy setting up, and cannot yet get information about the resource's URL or content.

Sources Available at the MetaData Filtering Stage

At the MetaData stage, the robot has encountered the URL for a resource, but has not downloaded the resource's content, thus information is available about the URL itself, as well as data that is derived from other sources such as the `filter.conf` file. At this stage however, information is not available about the content of the resource.

During the MetaData phase, the following sources are available to RAFs:

Source	Meaning	Example
csid	Catalog Server ID	x-catalog//compass.mydomain.com:8086/alexandria
depth	Number of links traversed from starting point	5
enumeration-filter	Name of Enumeration filter	
generation filter	Name of Generation filter	
host	host portion of URL	home.netscape.com
ip	Numeric version of host	198.95.249.6
protocol	Access portion of the URL	http, ftp, gopher
uri	Path portion of the URL	/, /index.html, /documents/listing.html
url	Complete URL	http://developer.netscape.com/library/documentation/

At the Data stage, the robot has already downloaded the content of the resource at the URL, thus it can access data about the content, such as the description, the author, and so on.

If the resource is an HTML file, the Robot parses the <META> tags in the HTML headers. Thus any data contained in <META> tags is available at the Data stage.

During the data phase, the following sources are available to RAFs, in addition to those available during the MetaData phase:

Source	Meaning	Example
content-charset	Character set used by the resource	
content-encoding	Any form of encoding	

Source	Meaning	Example
content-length	Size of the resource in bytes	
content-type	MIME type of the resource	text/html, image/jpeg
expires	Date the resource itself expires	
last-modified	Date the resource was last modified	
data in <META> tags	Any data that is provided in <META> tags in the header of HTML resources	Author Description Keywords

All these sources (except for the data in `<META>` tags) are derived from the HTTP response header returned when retrieving the resource.

Sources Available at the Enumeration, Generation, and Shutdown Stages

At the Enumeration and Generation stages, the same data sources are available as for the Data stage.

At the Shutdown stage, the filter has done its filtering and is shutting down. Although functions written for this stage can use the same data sources as those available at the Data stage, usually shutdown functions restrict their operations to shutdown and clean up activities.

Enable Parameter

Each function can have an `enable` parameter. The values can be `true`, `false`, `on`, or `off`. The Compass Server Administration Interface uses these parameters to turn certain directives on or off.

The following example enables enumeration for text/html and disables enumeratiaon for text/plain.

```
#  Perform the enumeration on HTML only

Enumerate enable=true fn=enumerate-urls max=1024 type=text/html

Enumerate enable=false fn=enumerate-urls-from-text max=1024 type=text/
```

`plain`

Adding an `enable=false` or `enable=off` has the same effect as commenting out the line. (The Compass Server Admistration Interface does not know about writing comments, so it writes an `enable` parameter instead.)

Setup Functions

These functions are used during the setup phase by both enumeration and generation filters.

filterrules-setup

Parameters `config` is the pathname to the file containing the filter rules to be used by this filter.

`logtype` is the type of log file to use. The value can be `verbose`, `normal`, or `terse`.

Example

```
Setup fn=filterrules-setup config=./config/filterrules.conf
logtype=normal
```

setup-regex-cache

This function initializes the cache size for by the filter-by-regex and generate-by-regex functions. Use this function to specify a number other than the default of 32.

Parameters `cache-size` is the maximum number of compiled regular expressions to be kept in the regex cache.

Example

```
Setup fn=setup-regex-cache cache-size=28
```

setup-type-by-extension

This function configures the filter to recognize file name extensions. It must be called before the assign-type-by-extension function can be used. The file specified as a parameter must contain mappings between standard MIME content types and file extension strings.

Parameters `file` is the name of the MIME types configuration file.

Example

```
Setup fn=setup-type-by-extension file=./config/mime.types
```

Filtering Functions

These five functions operate at the Metadata and Data stages to allow or deny resources based on specific criteria specified by the function and its parameters.

These functions can be used in both Enumeration and Generation filters in the file `filter.conf`.

Each of these "filter-by" functions performs a comparison, then either *allows* or *denies* the resource.

- Allowing the resource means that processing continues to the next filtering step.
- Denying the resource means that processing should stop, that for some reason the resource does not meet the criteria for further enumeration or generation.

filter-by-exact

This function allows or denies the resource if the allow/deny string matches the source of information exactly. The keyword "all" matches any string.

Parameters `src` is the source of information.

`allow/deny` contains a string.

Example This example filters out all resources whose content-type is `text/plain`. It allows all other resources to proceed.

```
Data fn=filter-by-exact src=type deny=text/plain
```

filter-by-max

This function allows the resource if the specified information source is less than or equal to the given value. It denies the resource if the information source is greater than the specified value.

This function can be called no more than once per filter.

Parameters `src` is the source of information. It must be one of the following: `hosts`, `objects`, or `depth`.

`value` contains a value for comparison.

Example This example allows resources whose content-length is less than 1024 K.

```
MetaData fn-filter-by-max src=content-length value=1024
```

filter-by-md5

This function allows only the first resource with a given MD5 checksum value. If the current resource's MD5 has been seen in an earlier resource by this robot, the current resource is denied. This prevents duplication of identical resources or single resources that happen to have multiple URLs associated with them.

You can only call this function at the Data stage or later. It can be called no more than once per filter. The filter must invoke the generate-md5 function to generate an MD5 checksum before invoking **filter-by-md5**.

Parameters *none*

Example This example shows the typical way to handle MD5 checksums, first generating the checksum and then filtering based on it.

```
Data fn=generate-md5
Data fn=filter-by-md5
```

filter-by-prefix

This function allows or denies the resource if the given information source begins with the specified prefix string. The resource doesn't have to match completely. The keyword "all" matches any string.

Parameters `src` is the source of information.

`allow/deny` contains a string for prefix comparison.

Example This example allows resources whose content-type is any kind of text, including `text/html` and `text/plain`.

```
MetaData fn=filter-by-prefix src=type allow=text
```

filter-by-regex

This function supports regular expression pattern matching. It allows resources that match the given regular expression. The supported regular expression syntax is defined by the POSIX.1 specification. The regular expression `\\*` matches anything.

Parameters `src` is the source of information.

`allow/deny` contains a regular expression string.

Example This example denies all resources from sites in the government domain.

```
MetaData fn=filter-by-regex src=host deny=\\*.gov
```

filterrules-process

This function rules the rules in the `filterrules.conf` file.

Parameters `none`

Example

```
MetaData fn=filterrules-process
```

Filtering Support Functions

These functions can be called during filtering to manipulate or generate information on the resource that the robot can then process by calling filtering functions. These functions can be used in Enumeration and Generation filters in the file `filter.conf`.

assign-source

This function assigns a new value to a given information source. This permits a form of editing during the filtering process. The function can assign an explicit new value or it can copy a value from another information source.

Parameters `dst` is the name of the source whose value is to be changed.

`value` specifies an explicit value.

`src` specifies an information source to copy to `dst`.

Note You must specify either a `value` parameter or a `src` parameter, but not both.

Example

```
Data fn=assign-source dst=type src=content-type
```

assign-type-by-extension

This function uses the resource's file name to determine its type and assigns this type to the resource for further processing.

Note The setup-type-by-extension function must be called during setup before **assign-type-by-extension** can be used.

Parameters `src` is the source of the file name to compare. If you do not specify a source, the default is the resource's URI.

Example

```
MetaData fn=assign-type-by-extension
```

clear-source

This function deletes the specified data source. You should rarely need to do this. You can create or replace a source using assign-source.

Parameters src is the name of the source to delete

Example The following example deletes the `uri` source:

```
MetaData fn=clear-source src=uri
```

convert-to-html

This function converts the current resource into an HTML file for further processing if its type matches a specified MIME type. The conversion filter itself automatically detects the type of the file it is converting, only converting those it can actually convert.

Parameters `type` is the MIME type to convert from

Example The following sequence of function calls causes the filter to convert all Adobe Acrobat PDF files, Microsoft RTF files, and FrameMaker MIF files to HTML, plus any files whose type was not specified by the server that delivered it.

```
Data fn=convert-to-html type=application/pdf
Data fn=convert-to-html type=application/rtf
Data fn=convert-to-html type=application/x-mif
Data fn=convert-to-html type=unknown
```

copy-attribute

This function copies the value from one field in the resource description into another.

Parameters `src` is the field in the resource description to copy from

`dst` is the item in the resource description to copy the source into

`truncate` is the maximum length of the source to copy

`clean` is a Boolean parameter indicating whether to clean up truncated text (such as not leaving partial words), which is false by default

Example The following example copies up to 200 bytes from the partial text of the resource into the `description` field of the resource description, cleaning up the truncated text:

```
Generate fn=copy-attribute \
   src=partial-text dst=description truncate=200 clean=true
```

generate-by-exact

This function generates a source with a specified value, but only if an existing source exactly matches another value.

Parameters `dst` is the name of the source to generate.

`value` is the value to assign to `dst`.

`src` is the source to match against.

`match` is the value to compare to `src`.

Example This example sets the classification to Netscape if the host is `www.netscape.com`.

```
Generate fn="generate-by-exact" match="www.netscape.com:80" src="host"
value="Netscape" dst="classification"
```

generate-by-prefix

This function generates a source with a specified value, but only if the prefix of an existing source matches another value.

Parameters `dst` is the name of the source to generate

`value` is the value to assign to `dst`

`src` is the source to match against

`match` is the value to compare to `src`.

Example This example sets the classification to `Compass` if the protocol prefix is http.

```
Generate fn="generate-by-prefix" match="http" src="protocol"
value="Compass" dst="classification"
```

generate-by-regex

This function generates a source with a specified value, but only if an existing source matches a regular expression.

Parameters `dst` is the name of the source to generate

`value` is the value to assign to `dst`

`src` is the source to match against

`match` is the regular expression string to compare to `src`.

Example This example sets the classification to `Netscape` if the host name matches the regular expression `*.netscape.com`. For example, resources at both `developer.netscape.com` and `home.netscape.com` will be classified as `Netscape`.

```
Generate fn="generate-by-regex" match="\\*.netscape.com" src="host"
value="Netscape" dst="classification"
```

generate-md5

This function generates an MD5 checksum and adds it to the resource. You can then use the filter-by-md5 function to deny resources with duplicate MD5 checksums.

Parameters *none*

Example `Data fn=generate-md5`

generate-rd-expires

This function generates an expiration date and adds it to the specified source. The function uses metadata such as the HTTP header and HTML META tags to obtain any expiration data from the resource. If none exists, it generates an expiration date three months from the current date.

Parameters `dst` is the name of the source. If you omit it, it defaults to `rd-expires`.

Example `Generate fn=generate-rd-expires`

generate-rd-last-modified

This function adds the current time to the specified source.

Parameters `dst` is the name of the source. If you omit it, it defaults to `rd-last-modified`.

Example `Generate fn=generate-last-modified`

rename-attribute

This function changes the name of a field in the resource description. It is most useful in cases where, for example, extract-html-meta copies information from a META tag into a field, and you want to change the name of the field.

Parameters `src` is a string containing a mapping from one name into another.

Example The following example renames an attribute from `author` to `author-name`:

```
Generate fn=rename-attribute src="author->author-name"
```

Enumeration Functions

These functions operate at the Enumerate stage. Use them in the definition for Enumeration filters. They control whether and how a robot gathers links from a given resource to use as starting points for further resource discovery.

enumerate-urls

This function scans the resource and enumerates all URLs found in hypertext links. The results are used to spawn further resource discovery. You can specify a content-type to restrict the kind of URLs enumerated.

Parameters `max` is the maximum number of URLs to spawn from a given resource. The default, if `max` is omitted, is 1024.

`type` specifies a content-type that restricts enumeration to those URLs that have the specified content-type. `type` is optional. If omitted, it will enumerate all URLs.

Example This example enumerates HTML URLs only, up to a maximum of 1024.

```
Enumerate fn=enumerate-urls type=text/html
```

enumerate-urls-from-text

This function scans text resources, looking for strings matching this regular expression: `URL:.*`. It spawns robots to enumerate the URLs from these strings and generate further resource descriptions.

Parameters `max` is the maximum number of URLs to spawn from a given resource. The default, if `max` is omitted, is 1024.

Example

```
Enumerate fn=enumerate-urls-from-text
```

Generation Functions

These functions can be used in the Generate stage of filtering. Use them in Generation filters. They generate information that goes into a resource description. In general, they either extract information from the body of the resource itself or copy information from the resource's metadata.

extract-full-text

This function extracts the complete text of the resource and adds it to the resource description. This function should be used with caution. It can significantly increase the size of the resource description, thus causing database bloat and overall negative impact on network bandwidth.

Parameters `truncate` is the maximum number of characters to extract from the resource.

`dst` is the name of the schema item that will receive the full text.

Example

```
Generate fn=extract-full-text
```

extract-html-meta

This function extracts any META or TITLE information from an HTML file and adds it to the resource description. A content-type may be specified to restrict the kind of URLs that are generated.

Parameters `truncate` is the maximum number of bytes to extract.

`type` is optional. If omitted, it will generate all URLs

Example

```
Generate fn=extract-html-meta truncate=255 type=text/html
```

extract-html-text

This function extracts the first few characters of text from an HTML file, excluding the HTML tags, and adds the text to the resource description. This permits the first part of a document's text to be included in the RD. A content-type may be specified to restrict the kind of URLs that are generated.

Parameters `truncate` is the maximum number of bytes to extract.

`skip-headings` is `true` to ignore any HTML headers that occur in the document.

`type` is optional. If omitted, it will generate all URLs.

Example

```
Generate fn=extract-html-text truncate=255 type=text/html
skip-headings=true
```

extract-html-toc

This function extracts the table-of-contents from the HTML headers and add it to the resource description.

Parameters

`truncate` is the maximum number of bytes to extract.

`level` is the maximum HTML header level to extract. This controls the depth of the table of contents.

Example

```
Generate fn=extract-html-toc truncate=255 level=3
```

extract-source

This function extracts the specified values from the given sources and adds them to the resource description.

Parameters

`src` is a list of source names; you can use the `->` operator to define a new name for the RD attribute, for example, `type->content-type` would take the value of the source named `type` and save it in the RD under the attribute named `content-type`.

Example

```
Generate fn=extract-source src="md5,depth,rd-expires,rd-
last-modified"
```

harvest-summarizer

This function runs a Harvest summarizer on the resource and adds the result to the resource description.

Note

To run Harvest summarizers, you must have `$HARVEST_HOME/lib/gatherer` in your PATH before you run the robot.

Parameters

`summarizer` is the name of the summarizer program.

Example `Generate fn-harvest-summarizer summarizer=HTML.sum`

Shutdown Functions

These functions can be used during the shutdown phase by both enumeration and generation functions.

filterrules-shutdown

This function does some clean up after the rules have been run.

Parameters *none*

Example `Shutdown fn=filterrules-shutdown`

Chapter

11

Creating New Robot Application Functions

This chapter describes how to create and compile your own plug-in robot application functions using the Netscape Compass robot plug-in application programming interface (API).

You would need to create your own robot application functions (RAFs) when you want to modify the behavior of the Compass robot filters in a way that is not accommodated either by the Compass Server Administration Interface or by the predefined robot application functions. Robot filters are defined in the file `filter.conf`. Filter definitions consist of filter directives, which each specify a robot application function.

Of the systems the Netscape Compass Server supports, the following systems can load functions into the server at run time and can therefore use plug-in functions:

- SunOS
- IRIX
- Solaris
- AIX
- HP/UX
- OSF/1
- Windows NT

What is the Robot Plug-in API?

The robot plug-in API is a set of functions and header files that help you create your own robot application functions to use with the directives in robot configuration files. The Netscape Compass Server uses this API to create the built-in functions for the directives used in `filter.conf` (the robot filter configuration file).

The robot uses this API, so by becoming familiar with the API, you can learn how the robot works. This means you can override the robot functionality, add to it, or customize your own functions. For example, you can create functions that use a custom database for access control or you can create functions that create custom log files with special entries.

We expect that most people will write RAF functions in C. However, you can define the functions in any language as long as it can build a shared library. If you use C++, you will need to modify the provided C header files to be used by C++ files.

The following steps are a brief overview of the process for creating your own plug-in functions:

1. Compile your code to create a shared object (`.so`) file.

   For Windows NT, you'll produce a dynamic-link library (DLL) containing your plug-in functions. When this chapter refers to a shared object file, it refers also to Windows NT DLLs.

2. In the **Setup** directives at the top of `filter.conf`, you tell the robot to load your shared object file or dynamic-link library.

3. Write directives that use your plug-in functions in the robot configuration file (`filter.conf`).

The *`compassdir`*`/bin/compass/sdk/robot/include/` directory contains all the header files you need to include when writing your plug-in functions.

The *`compassdir`*`/bin/compass/sdk/robot/examples/` directory contains sample code, the header files, and a makefile. You should familiarize yourself with the code and samples.

The Robot Application Function Header Files

This section discusses the header files needed for creating robot application functions.

Header File Hierarchy

The hierarchy of robot plug-in API header files is (directories are shown in **`code bold`**):

```
robot
  include
    csinfo.h
    csmem.h
    filterrules.h
    robotapi.h
    base
      systems.h
    libc
      adt.h
      cs.h
      csidcf.h
      getopt.h
      log.h
      pblock.h
```

The robot and its header files are written in ANSI C.

Header File Contents

This section describes the header files you can include when writing your plug-in functions. This section is intended as a starting point for learning the functions included in the header files.

Most of the header files are stored in three directories:

- `robot/include` contains header files that define general purpose data structures and function prototypes.
- `robot/include/base` contains the `systems.h` header files which deal with low-level, platform independent functions such as memory, file, and network access.

- `robotapi/include/libcs` contains header files of functions that deal with robot and HTTP-specific functions such as handling access to configuration files and dealing with HTTP.

Table 11.1 Header files in the `include` directory

Header File	Description
`csinfo.h`	Contains functions for object typing, specifically for mapping files to MIME types
`csmem.h`	Contains memory-related definitions.
`robotapi.h`	Contains the type definitions for structures and the return-code definitions for robot API functions.
`filterrules.h`	Contains type definitions for structures needed by filter rules.

Table 11.2 Header files in the `base` directory

Header File	Description
`systems.h`	Contains functions that handle systems information.

Table 11.3 Header files in the `libcs` directory

Header File	Description
`adt.h`	Contains type definitions and function prototypes for utilities needed by the robot, such as linked lists, queues, and hash tables.
`cs.h`	Contains a library of common functions used by the Compass Server.
`csidcf.h`	Contains configuration definitions for Compass Server ID configurations.
`getopt.h`	Contains routines to get options from the command line, for example, command line "prog -n arg1 -p arg2", to get arg1 and arg2. This is provided because NT doesn't support the library "getopt" which is common in the Unix world.

Table 11.3 Header files in the `libcs` directory

Header File	Description
`log.h`	Contains routines for writing information to log files.
`pblock.h`	Contains functions that manage parameter passing and robot internal variables. It also contains functions to get values from a user via the server.

Writing Robot Application Functions

The file that defines your robot application functions must include `robotapi.h`. You will also find many useful functions in `csinfo.h`.

All Robot Application Functions use parameter blocks, (pblocks) to receive and set parameter values. A parameter block stores parameters as name-value pairs. A parameter block is a hash table that is keyed on the name portion of each parameter it contains.

RAF Prototype

All robot application functions have the following prototype:

```
int (*RobotAPIFn)(pblock *pb, CSFilter *csf, CSResource *csr);
```

`pb` is the parameter block containing the parameters for this function invocation.

`csf` is the pointer to an enumeration or generation filter.

Caution! The *pb* parameter should be considered read-only, and any data modification should be performed on copies of the data. Doing otherwise is unsafe in threaded server architectures, and will yield unpredictable results in multiprocess server architectures.

Writing Functions for Specific Directives

You should write each function for a particular stage in the filtering process, (setup, metadata, data, enumeration, generation, and shutdown.) The function should only use the data sources that are available at the relevant stage. See the section Sources and Destinations in the previous chapter for a list of the data sources available at each stage.

At the Setup stage, the filter is busy setting up, and cannot yet get information about the resource's URL or content.

At the MetaData stage, the robot has encountered the URL for a resource, but has not downloaded the resource's content, thus information is available about the URL itself, as well as data that is derived from other sources such as the `filter.conf` file. At this stage however, information is not available about the content of the resource.

At the Data stage, the robot has downloaded the content of the URL, so information is available about the content, such as the description, the author, and so on.

At the Enumeration and Generation stages, the same data sources are available as for the Data stage.

At the Shutdown stage, the filter has done its filtering and is shutting down. Although functions written for this stage can use the same data sources as those available at the Data stage, usually shutdown functions restrict their operations to shutdown and clean up activities.

Passing Parameters to Robot Application Functions

You must use parameter blocks (pblocks) to pass arguments into Robot Application Functions, and to get data out of them. For example, the following directive (which would be in the `filter.conf` file) invokes the `filter-by-exact` function.

```
Data fn=filter-by-exact src=type deny=text/plain
```

The `fn` parameter indicates the function to invoke, which in this case is `filter-by-exact`. The `src` and `deny` arguments are parameters for the function. They will be passed to the function in a parameter block, and the function must be defined to dis-assemble the parameter block and extract the parameters and their values from it.

The three structures that are used to hold parameters are `libcs_pb_param`, `libcs_pb_entry`, and `libcs_pblock`. These structures are defined in the header file `bin/compass/sdk/robot/include/pblock.h`.

- `libcs_pb_param`

  This holds a single parameter. It records the name and value of the parameter.

  ```
  typedef struct {
    char *name,*value;
  } libcs_pb_param;
  ```

- `libcs_pb_entry`

  This creates linked lists of `libcs_parameter` structures.

  ```
  struct libcs_pb_entry {
    libcs_pb_param *param;
    struct libcs_pb_entry *next;
  };
  ```

- `libcs_pblock`

  This is a hash-table containing an array of `libcs_pb_entry` structures.

  ```
  typedef struct {
    int hsize;
    struct libcs_pb_entry **ht;
  } libcs_pblock;
  ```

Working with Parameter Blocks

A parameter block stores parameters and values as name/value pairs. There are many pre-defined functions you can use to work with parameter blocks, to extract parameter values, change parameter values, and so on. For example, `libcs_pblock_findval(paramname, returnPblock)` uses the given return pblock to return the value of the named parameter in the RAF's input pblock. (For a fully-fledged example, see RAF Definition Example.)

When adding, removing, editing, and creating name-value pairs for parameters, your robot application functions can use the functions in the `pblock.h` header file (in `bin/compass/sdk/robot/include/libcs/`).

The names of these functions are all prefixed by **libcs_**. These functions are similar to the non-prefixed functions documented in Chapter 4, "NSAPI Function Reference" of the *NSAPI Programmer's Guide*.

You can find the NSAPI Programmer'sGguide at:

`http://developer.netscape.com/library/documentation/enterprise/nsapi/index.htm`

For example, the **function pb_create** described in the *NSAPI Programmer's Guide*, has the same behavior as the function **libcs_pb_create** in the `bin/compass/sdk/robot/include/libcs/pblock.h` header file.

The parameter manipulation functions are listed here. Please view the `bin/compass/sdk/robot/include/libcs/pblock.h` header file for full function signatures, with return type and arguments.

libcs_param_create

creates a parameter with the given name and value. If the name and value aren't null, they are copied and placed into a new **pb_param** structure.

libcs_param_free

frees a given parameter if it's non-NULL. It returns 1 if the parameter was non-NULL, and 0 if it was NULL. This function is also useful for error checking before using the **libcs_pblock_remove** function.

libcs_pblock_create

creates a new parameter block with a hash table of a chosen size. Returns the newly allocated parameter block

libcs_pblock_free

frees a given parameter block and any entries inside it.

libcs_pblock_find

finds the entry with the given name in a pblock, and returns its value, otherwise returns NULL.

libcs_pblock_findval

also finds the entry with the given name in a pblock, and returns its value, otherwise returns NULL.

libcs_pblock_remove

behaves like the **libcs_pblock_find** function, but it also removes the entry from the pblock.

libcs_pblock_nninsert and **libcs_pblock_nvinsert**

these both create a new parameter with a given name and value, and insert it into a given parameter block. The **libcs_pblock_nninsert** function requires that the value be an integer, but the **libcs_pblock_nvinsert** function accepts a string.

libcs_pblock_pinsert

inserts a parameter into a parameter block.

libcs_pblock_str2pblock

scans the given string for parameter pairs in the format name=value or name="value", adds them to a pblock, and returns the number of parameters added.

libcs_pblock_pblock2str

places all of the parameters in the given parameter block into the given string. Each parameter is of the form `name=`"value" and is separated by a space from any adjacent parameter.

Getting Information About the Resource Being Processed

As mentioned earlier, the prototype for all robot application functions is:

```
int (*RobotAPIFn)(pblock *pb, CSFilter *csf, CSResource *csr);
```

where *`csr`* is a data structure that contains information about the resource being processed.

The `CSResource` structure is defined in the header file `robotapi.h`. This structure contains information about the resource being processed. Each resource is in SOIF syntax.

Objects in SOIF syntax have a schema name, an associated URL, and a set of attribute-value pairs. In the following SOIF example, the schema name is `@document`, the URL is `http://developer.netscape.com/library/documentation/htmlguid/index.htm`, and the SOIF contains attribute-value pairs for title, author, and description.

```
@DOCUMENT { http://developer.netscape.com/library/documentation/htmlguid/index.htm
  title{35}: HTML Tag Reference
  author{37}: Nikki Writer
  description{39}: Reference to HTML tags and attributes
}
```

A `CSResource` structure has a `url` field, which contains the URL for the SOIF. It also has an `rd` field, whose value is the SOIF for the resource. Once you get the SOIF for the resource, you can use the functions for working with SOIF that are defined in s`dk/rdm/include/soif.h` to get more information about the resource. (The file `robotapi.h` includes `soif.h`.)

For example, the macro `SOIF_Findval(soif, attribute)` gets the value of the given attribute in the given SOIF. The following sample code uses this macro to print the value of the META attribute if it exists for the resource being processed:

```
int my_new_raf(libcs_pblock *pb, CSFilter *csf, CSResource *csr)
  char *metavalue;
  if (metavalue = (char *)SOIF_Findval(csr->rd, "meta"))
  printf("The value of the META tag in the resource is %s" metavalue);
   /* rest of function ... */
}
```

We encourage you to examine the `CSResource` structure in the file `robotapi.h` to see what other fields it has and to see the macros, such as `CSResource_GetSource()`, you can use to get information from the resource and to set information in the resource description.

For more information about the routines for working directly with SOIF objects, see Chapter 11, "Using the SOIF API to Work with SOIF Objects."

Returning a Response Status Code

When your robot application function has finished whatever it needs to do, it must return a code that tells the server how to proceed with the request.

These codes are defined in the header file `bin/compass/sdk/robot/include/robotoapi.h`. The codes are:

`REQ_PROCEED`

The function performed its task, so proceed with the request.

`REQ_ABORTED`

The entire request should be aborted because an error occurred.

`REQ_NOACTION`

The function performed no task, but proceed anyway.

`REQ_EXIT`

End the session and exit.

`REQ_RESTART`

Restart the entire request-response process.

Reporting Errors to the Robot Log File

When problems occur, robot application functions should return an appropriator HTTP response status code (such as **`REQ_ABORTED`**) and they should also log an error in the error log file.

To use the error-logging functionality, you must include the files `log.h` and `objlog.h` in the `sdk/robot/include/libcs` directory. You should check that this directory does in fact contain these files (some of the early releases of Compass Server 3.0 omitted them).

If these files are not there, you can open them here, and save them to your `bin/compass/sdk/robot/include/libcs` directory:

```
log.h
```

```
objlog.h
```

After you have ensured that `log.h` and `objlog.h` exist in the correct place, you can use the `cslog_error` macro to report errors. The prototype is:

```
cslog_error(int n, int loglevel, char* errorMessage)
```

The first parameter is not currently used (it may be used in the future.) You can pass this as any integer.

The second parameter is the log level. When the log level is less than or equal to the log level setting in the file `process.conf`, the error message is written in the robot.log.

The third parameter is the error message to print, and it has the same form as the argument to the standard `printf()` function.

For example:

```
cslog_error(1, 1, ("fn=extract-html-text: Out of memory!\n"));
```

This invocation of `cslog_error` would generate the following error message in the robot log file:

```
[22/Jan/1998:15:57:31]  8270@0: ERROR: fn=extract-html-text: Out of
memory!
```

For another example:

```
cslog_error(1, 1,
  ("<URL:%s>: Error %d (%d): %s\n",
  ep->eo->key,
  urls->server_status,
  status,
  (s = cslog_linestr(urls->error_msg)))
```

This invocation of `cslog_error` would generate the following error message in the robot log file:

```
[22/Jan/1998:15:57:31]  8270@0: ERROR: <URL:http://nikki.boots.com:80/
>: Error 0 (-240): Can't connect to server
```

RAF Definition Example

This section shows an example definition for a robot application function.

This function copies a specified source data to a multi-valued field in an RD. For example, the Compass Server stores category or classification information in the `classification` field of an RD. The **`copy_mv`** function allows the robot to get the value of an HTML `<META>` tag of any name and store the value in the `classification` field in the database. For example, using this function, you could instruct the robot to get the content of the `<META NAME="topic">` tag, and store it as the classification of the resource.

You would invoke this function with a directive such as:

```
Generate fn=copy_mv src=topic dst=classification
```

Here is the sample function definition:

```
/******** example robot application function ********/

#include robotapi.h
#include pblock.h
#include log.h
#include objlog.h

NSAPI Public int copy_mv(libcs_pblock *pb, CSFilter *csf,
CSResource *csr)
{
  char *s, *mv, *mvp;
/* Use the libcs_pblock_findval function to get the values of the
 * "src" and "dst" parameters, which were specified by the
 * directive that invoked this function */

  char *src = libcs_pblock_findval("src", pb);
  char *dst = libcs_pblock_findval("dst", pb);

/* if either the src or dst has not been supplied,
 * log an error and return REQ_PROCEED */

  if(!src || !dst) {
    cslog_error(1, 1,
      ("<URL:%s>: Error: No source or destination available."
      csr->url,)
    return REQ_PROCEED;
  }

/* If the current document does not have a META tag whose name
 * matches the src parameter, just return, otherwise put the
 * src value in the string s */

/* The function SOIF_Findval(soif, attribute) is defined
 * in sdk/rdm/include/soif.h. It gets the value of the
 * given attribute from the given resource.
 * The rd in the CSResource is a soif that describes the resource.
*/

  if(!(s = (char *)SOIF_Findval(csr->rd, src)))
```

```
  return REQ_PROCEED;

/* Now insert the string s into the
 * Classification field of the RD */

/* Deal with possibility that the classification field
 * already has one or more values */
  if((mv = libcs_pblock_findval(dst, csr->sources)) != NULL) {
    mvp = malloc((strlen(mv)+strlen(s)+2));
    sprintf(mvp, "%s;%s", mv, s);
    /* append the new value to the existing values in the
     * classification field, separated by ';' */
    libcs_pblock_nvinsert(dst, mvp, csr->sources);
    /* do some clean up */
    free(mvp);
  }
  /* if no values already exist, do a simple value insert */
  else
  {
    libcs_pblock_nvinsert(dst, s, csr->sources);
  }

  /* We're all done. Return a status code */
  return REQ_PROCEED;
}
```

Compiling and Linking your Code

You can compile your code with any ANSI C compiler. See the makefile in the `bin/compass/sdk/robot/include` directory for an example. The UNIX makefile assumes the use of `gmake`.

For Windows NT

- Use Microsoft Visual C++ 4.x to compile a DLL.
- You also need the file `librobot30.lib` for NT only, (Unix does not need it.). The `librobot30.lib` is in the `sdk/robot/lib` directory

For UNIX

This section lists the linking options you need to use to create a UNIX shared object. The server can be instructed to load by commands in the `magnus.conf` configuration file.

The following table describes the commands used to link object files into a shared object under the various UNIX platforms. In these examples, the compiled object files **`t.o`** and **`u.o`** are linked to form a shared object called **`test.so`**.

Table 11.4 Options for linking

System	Compile options
IRIX	`ld -shared t.o u.o -o test.so`
SunOS	`ld -assert pure-text t.o u.o -o test.so`
Solaris	`ld -G t.o u.o -o test.so`
OSF/1	`ld -all -shared -expect_unresolved "*" t.o u.o -o test.so`
HP-UX	`ld -b t.o u.o -o test.so` When compiling your code, you must also use the `+z` flag to the HP C compiler.
AIX	`cc -bM:SRE -berok t.o u.o -o test.so -bE:ext.exp -lc` The `ext.exp` file must be a text file with the name of a function that is externally accessible for each line.

Loading Your Shared Object

The robot uses the filters defined in `filter.conf` to filter resources that it encounters. If the file `filter.conf` use your customized robot application functions, it must load the shared object (or DLL) that contains the functions.

To load the shared object, add a line to `filter.conf`:

For UNIX

```
Init fn=load-modules shlib=[path]filename.so funcs="function1,function2,...,functionN"
```

For Windows NT

```
Init fn=load-modules shlib=[path]filename.dll funcs="function1,function2,...,functionN"
```

This initialization function opens the given shared object file (or DLL) and loads the functions *function1*, *function2*, and so on. You can then use the functions *function1* and *function2* in the robot configuration file (filter.conf). Remember to use the functions only with the directives you wrote them for, as described in the following section.

Using your New Robot Application Functions

When you have compiled and arranged for the loading of your functions, you need to provide for their execution. All functions are called as follows:

Directive fn=*function* [*name1*=*value1*] ... [*nameN*=*valueN*]

- *Directive* identifies the class of function that is being called. Functions should not be called from the wrong directive!
- fn=*function* identifies the function to be called using the function's unique character-string name.

These two parameters are mandatory. After this, there may be an arbitrary number of function-specific parameters, each of which is a name-value pair.

You specify your function in the directive it was written for. For example, the following line uses a plug-in function called wordcount that can be used in the Data stage. This function counts the words in a resource and assigns the count to a destination specified by a parameter called dst.

```
Data fn=wordcount dst=word-count
```

3

Using C to Send Requests to the Compass Server

This part of the document discusses how to use the C functions in the RDM SDK to do the following tasks:

- create, read, and modify SOIF objects. SOIF is the syntax that is used to store resource descriptions in the Compass Server database.
- create instructions in the form of *request description messages* (RDMS) to send to the Compass Server database. RDMs also use SOIF syntax. After creating an RDM, you can send it to the database using the `sendrdm` program.

The standard procedure for accessing information in the Compass Server database is to use the Compass Server End User page, as discussed in the *Compass Server Administration Guide*. However, you can use the Compass Server RDM SDK to access the database using C. First you create an RDM using the functions defined in `rdm.h`, then send it to the database by using the `sendrdm` program, and finally process the results using the functions defined in `soif.h`.

The Compass Server RDM SDK is shipped with the Compass Server. You can find it in *`compassdir`*`/bin/compass/sdk/rdm`.

What's In the RDM SDK?

The RDM SDK contains `examples`, `bin`, `lib`, and `include` directories.

The `include` directory contains the following header files:

- `soif.h`

  This contains routines for manipulating RDs in SOIF format. For more information on SOIF format, see http://www.w3.org/TR/NOTE-rdm.html#soif.

- `rdm.h`

  This header file contains routines for talking to the Compass Server database. You can use can use them, for example, for making your own routines to query the Compass Server database.

The `examples` directory contains examples of the use of the routines in `soif.h`:

- `example1.c`: Parsing the RD input and printing out its URL string.
- `example2.c`: Parsing the RD input and printing out specified attributes.
- `example3.c`: This is the same as example2 but prints the specified attributes in SOIF format.

Chapter Summary

The chapters in this part of the document are:

Chapter 12. Using the SOIF API to Work with SOIF Objects

Overview of SOIF syntax, and a discussion of the structures and functions in `soif.h` that you can use to create, read, write, and modify SOIF objects.

Chapter 13. Using the RDM API To Accesss the Compass Server Database in C

Overview of resource description messages (RDMs), and a discussion of the structures and functions in `rdm.h` to create and process resource description messages to send to the Compass Server database.

Chapter

12

Using the SOIF API to Work with SOIF Objects

This chapter discusses the use of the Compass Server SOIF API to work with SOIF objects in C. Resource descriptions in the Compass Server database are described in SOIF, and so are resource description messages (RDMs) that processes can use to exchange resource descriptions across a network.

The SOIF API provides routines for creating and modifying SOIF objects in C.

The SOIF API is defined in the `soif.h` file in the following directory in your Compass Server installation directory:

```
bin/compass/sdk/rdm/include
```

This chapter does not teach you how to use C. This chapter is restricted to discussing the use of C functions that come with the Compass Server SOIF API.

What is SOIF?

SOIF stands for Summary Object Interchange Format. It is a syntax that can be used for many things. One of the things it is used for is to describe resource descriptions (RDs) in the Compass Server database.

The SOIF format is a basic attribute-value format. SOIF files look like text but should be treated as binary data and edited with care, especially on the PC platform. SOIF files contain tabs, and many editors will convert tabs to spaces

and corrupt the file. Many PC editors will add CR/LF line delimiters again corrupting the file. You can use SOIF-manipulation functions to create and modify SOIF objects so you do not have to write and edit them manually.

The following sample SOIF describes a document, whose title is "Rescuing English Springer Spaniels", whose author is "Jocelyn Becker" and whose URL is http://www.best.com/~jocelyn/resdogs/index.html."

```
@DOCUMENT { http://www.best.com/~jocelyn/resdogs/index.html
  title{20}: Rescuing English Springer Spaniels
  author{29}: Jocelyn Becker
}
```

Each SOIF object has a schema-name (or template type) and an associated URL, and it contains a list of attribute-value pairs. In this case, the schema name is `@DOCUMENT`, which indicates this resource is a document. Title and author are both attribute names, and you can see that each attribute has a value.

Using the SOIF API

The SOIF API is defined in the `soif.h` header file in *`compassdir/bin/`*`compass/sdk/rdm/include`.

The SOIF API defines structures and functions for working with SOIF objects. For example, the following code uses the functions `SOIF_Create()` and `SOIF_InsertStr()` to create a SOIF and add some attribute-value pairs to it:

```
SOIF mysoif=SOIF_Create("DOCUMENT", "http://someurl/doc.htm");
SOIF_InsertStr(mysoif, "title", "All About Style Sheets");
SOIF_InsertStr(mysoif, "author", "Robin Styles");
SOIF_InsertStr(mysoif, "description", "All you need to know about style sheets");
```

These command create a SOIF that looks like:

```
@document {http://someurl/doc.htm
  title{22}: All About Style Sheets
  author{12}: Robin Styles
  description{38}: All you need to know about style sheets
}
```

Each `SOIF` object contains attribute-value pairs, which are each represented as `SoifAVPair` objects. Using the SOIF API, you can get and set values of attributes, you can create and delete attribute-value pairs, you can change the values of attributes, and you can add values to existing attributes. (Some attributes can have multiple values.)

Multiple SOIF objects can be grouped together into SOIF streams, which are represented by `SOIFStream` objects. A `SOIFStream` object provides functionality for handling a stream of `SOIF` objects. For example, you can use the stream to filter attributes, and print the desired attributes for every `SOIF` in the stream.

The relevant data structures are:

- `SOIF` -- a SOIF object.
- `SOIFAVPair` - an attribute-value pair.
- `SOIFStream` -- a stream of SOIF objects.

An Introductory Example

You will find several examples of the use of the SOIF API in *compassdir*/bin/compass/sdk/rdm/examples.

This section discusses an example that is similar to (but not necessarily identical to) `example1.c`. It shows how to iterate through a SOIF stream and print the URL and number of attributes of each SOIF in the stream.

This example assumes that you have already created a file containing a SOIF stream which is available on stdin. For example, you could have created a SOIF stream containing one or more RDs from the Compass Server database, which you would do by using the routines in `RDM.h` (as is discussed in Chapter 12, "Using the RDM API To Accesss the Compass Server Database in C").

This example uses `SOIF_ParseInitFile()` to create a SOIF stream from the standard input.

```
/* Example 1 - Simple SOIF Stream Parsing */

#include <stdio.h>
#include <stdlib.h>
#include "soif.h"

int main(int argc, char *argv[])
{
  /* Define a SOIFStream and SOIF */
  SOIFStream *ss;
  SOIF *s;
  char *titleptr;
```

```
  /* Open a SOIF stream that gets its SOIF from stdin */
  ss = SOIF_ParseInitFile(stdin);
  /* SOIFStream_IsEOS() checks if this is the end of the stream */
  /* ss->parse gets the next SOIF in the stream */

  while (!SOIFStream_IsEOS(ss)) {
    if (!(s = SOIFStream_Parse(ss)))
      /* Exit the loop if the SOIF is invalid */
      break;

    /* Print the URL for each SOIF (will be "-" if there is no URL)*/
    printf("URL = %s\n", s->url);

    /* Print the title if it exists. */
    titleptr = SOIF_Findval(s, "title");
    printf("Title = %s\n", titleptr ? titleptr : "(none)")

    /* Print the number of attributes in the SOIF*/
    printf("# of Attributes = %d\n", SOIF_GetAttributeCount(s));

    /* release the memory used by the SOIF */
    SOIF_Free(s);
    }

  /* Close the SOIFStream and exit */
  SOIFStream_Finish(ss);
  exit(0);
}
```

Getting the Compass Server Database Contents as a SOIFStream

You can retrieve the entire contents of the Compass Server database as a SOIF stream by using the `minidba` utility.

From the *`compass-installdir/`*`compass-`*`name/`* directory, run the following command:

```
../bin/compass/bin/minidba dump
```

This command prints the database contents as a `SOIFStream`. You can pipe the output to a program that uses `SOIFStream` routines to parse the SOIFs in the stream.

SOIF API

The functions and objects defined in the `soif.h` header file are listed here by category. The categories are:

- SOIF Structure
- Attribute-Value Pair Routines
- Multi-valued Attribute Routines
- SOIF Stream Routines for Parsing and Printing SOIFs
- Filtering SOIF Objects
- Memory Buffer Management

The functions are not sorted alphabetically. However, here is an alphabetical list in case you need it:

Alphabetical Function List

SOIF Structure

A SOIF has a schema-name and it associates a URL with a collection of attribute- value pairs. The schema-name identifies how to interpret the attribute-value pairs. SOIF supports text and binary data, and attributes can have multiple values.

An example SOIF is:

```
@DOCUMENT { http://www.netscape.com/
title{20}:Welcome to Netscape!
author{15}:Marc Andreessen
}
```

A SOIF object has `url` and `schema-name` fields to store its url and schema name:

```
char *url;          /* The URL */
char *schema_name; /* The Schema-Name, such as @document or @RDMHeader*/
```

A SOIF object contains a collection of `SoifAVPair` objects, which each contain an attribute and one or more values. To access attribute values in a SOIF, use SOIF_`find()` to retrieve the `AVPair` for the given attribute, or use SOIF_`findval()` to retrieve the value string for a given attribute. You must use all lowercase for attribute names for `find*()`, since only exact attribute name lookups are supported.

You can create SOIF objects by using the `SOIF_create()` function. You can also read SOIF objects from a SOIF stream.

SOIF_Create

```
NSAPI_PUBLIC SOIF *SOIF_Create(char *schema_name, char *url)
```

Creates a SOIF structure with the given schema name and URL.

SOIF_Free

```
NSAPI_PUBLIC void SOIF_Free(SOIF *)
```

Frees the given SOIF structure.

SOIF_GetTotalSize

```
NSAPI_PUBLIC int SOIF_GetTotalSize(SOIF *s)
```

Gets the estimated total size of the SOIF in bytes.

SOIF_GetAttributeCount

```
NSAPI_PUBLIC int SOIF_GetAttributeCount(SOIF *s)
```

Gets the number of attributes in the SOIF.

SOIF_GetAttributeSize

```
NSAPI_PUBLIC int SOIF_GetAttributeSize(SOIF *s)
```

Gets the size of the attributes only.

SOIF_GetValueSize

```
NSAPI_PUBLIC int SOIF_GetValueSize(SOIF *s)
```

Gets the size of the values only.

SOIF_GetValueCount

```
NSAPI_PUBLIC int SOIF_GetValueCount(SOIF *s)
```

Gets the number of values only.

SOIF_Merge

```
NSAPI_PUBLIC int SOIF_Merge(SOIF *dst, SOIF *src);
```

Use this function to merge two SOIF objects (perform a Union of their attribute-values). It returns non-zero on error; otherwise, returns zero and the 'dst' SOIF object contains all the attribute-value pairs from the 'src' SOIF object.

If the 'dst' object contains the same attribute as 'src', then the attribute becomes a multi-valued attribute and all of the values are copied over to 'dst'. Only multi-valued attributes are copied over. For single-value attributes, discard the value in 'dst'. Currently only "classification" is a multi-valued attribute.

SOIF_Find

```
#define SOIF_Find(soif, attribute-name)
```

retrieves the `AVPair` for the given attribute in the given soif. For example, the following statement gets the `AVPair` for the title attribute in the soif *s*:

```
SOIFAVpair avp=SOIF_Find(s, "title");
```

SOIF_Findval

```
#define SOIF_Findval(soif, attribute-name)
```

retrieves the value string for the given attribute in the given soif. For example, the following statement prints the value of the title attribute of the soif *s*:

```
printf("Title = %s\n", SOIF_Findval(s, "title"));
```

SOIF_Remove

```
#define SOIF_Remove(soif, attribute-name)
```

Removes the given attribute from the given soif.

SOIF_Insert

```
#define SOIF_Insert(soif, attribute-name, value, value-size)
```

Inserts the given attribute and the value of the given size as an `AVPair` into the soif.

SOIF_InsertAVP

```
#define SOIF_InsertAVP(soif, avpair)
```

Inserts the given `AVPair` into the given soif.

SOIF_Apply

```
#define SOIF_Apply(soif, function, user-date)
```

Applies the given function with the given argument (*`user-data`*) to each `AVPair` in the given soif. For example:

```
void print_av(SOIF *s, SOIFAVPair *avp, void *unused)
{printf("%s = %s\n", avp->attribute, avp->value);}

/* print every attribute and value in the soif s */
SOIF_Apply(s, print_av, NULL);
```

Attribute-Value Pair Routines

Attribute-value pairs contain an attribute and an associated value. The value often is a simple null-terminated string; however, the value may be binary data. Attribute-value pairs are stored as `SOIFAVPair` structures.

The important fields in a `SOIFAVPair` structure are:

```
char *attribute;  /* Attribute string; '\0' terminated */
char *value;      /* primary value; may be '\0' terminated */
size_t vsize;     /* # of bytes (8 bits) for primary value */
char **values;    /* Multiple values for multivalued attributes */
size_t *vsizes;   /* the sizes for the values */
int nvalues;      /* number of values associated with attribute */
int last_slot;    /* last valid slot - array may contain holes */
```

SOIFAVPair_Create

```
NSAPI_PUBLIC SOIFAVPair * SOIFAVPair_Create(char *a, char *v, int vsz);
```

Creates an `AVPair` structure with the given attribute *a* and value *v*. The value *v* is a buffer of *vsz* bytes.

SOIFAVPair_Free

```
NSAPI_PUBLIC void SOIFAVPair_Free(SOIFAVPair *avp);
```

Frees the memory used by the given `SOIFAVPair` structure.

SOIF_Replace

```
NSAPI_PUBLIC int SOIF_Replace(SOIF *s, char *att, char *val, int valsz);
```

Replaces the value of an existing attribute *att* with a new value *val* of size *valsz* in the SOIF *s*.

SOIF_InsertStr

```
#define SOIF_InsertStr(soif, attribute, value)
```

Inserts the given attribute with the given value into the soif.

SOIF_ReplaceStr

```
#define SOIF_ReplaceStr(soif, attribute, value)
```

Replaces the existing value of the given attribute in the soif with the given value.

SOIF_Rename

```
NSAPI_PUBLIC int SOIF_Rename(SOIF *s, char *old_attr, char *new_attr);
```

Renames the given attribute to the given new name.

SOIF_AttributeCompare

```
NSAPI_PUBLIC int SOIF_AttributeCompare(const char *a1, const char *a2);
```

Compares two attribute names. Returns 0 (zero) if they are equal, or non-zero if they are different. Case (upper and lower) and trailing -s are ignored when comparing attribute names. The following table illustrates the results of comparing some attribute names.

AttributeA	AttributeB	Does SOIF_AttributeCompare() consider them to be the same?
title	Title	yes
title	TITLE	yes
title	title-	yes
title	title-page	no
titl	title	no
author	title	no

Multi-valued Attribute Routines

A SOIF attribute can have multiple values. SOIF supports the convention of using -### to indicate a multivalued attribute. For example, Title-1, Title-2, Title-3, etc. The -### do not need to be sequential positive integers (e.g., 1, 2, 3, ...); the sequence can have "holes" in it.

Compass Server currently supports only "classification" as a multi-valued attribute. In SOIF representation, it is represented using `classification-1`, `classification-2`, and so on. For example:

```
classification-1{5}: robot
classification-2{8}: netscape
classification-3{10}: web crawler
```

SOIF_AttributeCompareMV

```
NSAPI_PUBLIC int SOIF_AttributeCompareMV(const char *a1, const char
*a2);
```

Compares two attribute names. Returns 0 (zero) if they are equal, or non-zero if they are different. If neither of the attributes is multi-valued then use above routine SOIF_AttributeCompare(). If one or both of the attributes are multi-value, use the base name of the multi-valued attribute for comparison. The "base name" of a multi-valued attribute is the name portion before "-". For example, the base name of "classification-3" is "classification".

SOIF_MVAttributeParse

```
NSAPI_PUBLIC int SOIF_MVAttributeParse(char *a)
```

Returns the multivalued number of the given attribute, and strips the attribute string of its -### indicator; otherwise, returns zero in the case of a normal attribute name. For example, "`classification-3`" returns the number 3

SOIF_IsMVAttribute

```
NSAPI_PUBLIC char *SOIF_IsMVAttribute(const char *a);
```

Returns NULL if the given attribute is not a multivalued attribute; otherwise returns a pointer to where the multivalued number occurs in the attribute string. For example, for the multi-valued attribute "classification-3", it will return the pointer to "3".

SOIF_InsertMV

```
NSAPI_PUBLIC int SOIF_InsertMV(SOIF *s, char *a, int slot, char *v,
int vsz, int useval)
```

Inserts a new value *v* at index *slot* for the given attribute *a* (in non-multivalue form). If set, the *useval* flag tells the function to use the given value buffer rather than creating its own.

For example:

```
SOIF_InsertMV(s, "classification", 3, "web crawler", strlen("web
crawler");
```

inserts

```
classification-3{10}: web crawler
```

SOIF_ReplaceMV

```
NSAPI_PUBLIC int SOIF_ReplaceMV(SOIF *s, char *a, int slot, char *v,
int vsz, int useval);
```

SOIF_DeleteMV

```
NSAPI_PUBLIC int SOIF_DeleteMV(SOIF *s, char *a, int slot)
```

Deletes the value at the index *slot* in the attribute *a*. For example:

```
SOIF_DeleteMV(s, "classification", 3)
```

deletes "classification-3".

SOIF_FindvalMV

```
NSAPI_PUBLIC const char *SOIF_FindvalMV(SOIF *s, const char *a,
int slot)
```

Finds the value at the index *slot* in the attribute *a.* For example:

```
SOIF_FindvalMV(s, "classification", 3)
```

returns "**web crawler**" (using the previous example).

SOIF_SqueezeMV

```
NSAPI_PUBLIC void SOIF_SqueezeMV(SOIF *s)
```

Forces a re-numbering to ensure that the multivalue indexes are sequentially increasing (e.g., 1, 2, 3,...). This function can be used to fill in any holes that might have occurred during `SOIF_InsertMV()` invocations. For example: to insert values explicitly for the multivalue attribute `author-*`:

```
SOIF_InsertMV(s, "author", 1, "John", 4, 0);
SOIF_InsertMV(s, "author", 2, "Kevin", 5, 0);
SOIF_InsertMV(s, "author", 6, "Darren", 6, 0);
SOIF_InsertMV(s, "author", 9, "Tommy", 5, 0);
SOIF_FindvalMV(s, "author", 9); /* == "Tommy" */
SOIF_SqueezeMV(s);
SOIF_FindvalMV(s, "author", 9); /* == NULL */
SOIF_FindvalMV(s, "author", 4); /* == "Tommy" */
```

SOIFAVPair_IsMV

```
#define SOIFAVPair_IsMV(avp)
```

Use this to determine if the AVPair has multiple values or not.

SOIFAVPair_NthValid

```
#define SOIFAVPair_NthValid(avp,n)
```

Use this to determine if the Nth value is valid or not.

SOIFAVPair_NthValue

```
#define SOIFAVPair_NthValue(avp,n)((avp)->values[n])
```

Use this to access the Nth value. For example:

```
for (i = 0; i <= avp->last_slot; i++)
  if (SOIFAVPair_NthValid(avp, i))
    printf("%s = %s\n", avp->attribute,
      SOIFAVPair_NthValue(avp, i));
```

SOIFAVPair_NthVsize

```
#define SOIFAVPair_NthVsize(avp,n)((avp)->vsizes[n])
```

Use this to get the size of the Nth value.

SOIF_Contains

```
NSAPI_PUBLIC boolean_t SOIF_Contains(SOIF *s, char *a, char *v,
int vsz);
```

Indicates if the given attribute contains the given value. It returns `B_TRUE` if the value matches one or more of the values of the attribute *a* in the given SOIF *s*.

SOIF Stream Routines for Parsing and Printing SOIFs

A `SOIFStream` contains one or more `SOIF` objects.

The general approach is that you use SOIF streams to get streams of SOIF objects. Given a SOIF stream, you can parse it to get the SOIF objects from it. Use the `parse()` routine to get the next SOIF object in a SOIF stream. You can use `SOIFStream_IsEOS`() to check whether the last object has been parsed.

You can use filtering functions for a SOIF stream to specify that certain SOIF attributes are *allowed* or *denied*. If an attribute is allowed, you can parse and print that attribute for SOIF objects in the stream. If it is denied, you cannot parse or print that attribute of SOIF objects in the stream.

SOIF streams can be disk-based or memory based.

When you create a `SOIFStream`, you need to specify if you will be printing or parsing the SOIF stream and if you will be using a memory- or disk-based stream. There are several SOIF stream creation functions, and which one you need to use depends on what you will be doing with the SOIF stream.

For creating a SOIF streams that you will be printing SOIFS into, the functions are:

- `SOIF_PrintInitFile()` - creates a disk-based stream ready for printing.
- `SOIF_PrintInitStr()` - creates a memory-based stream ready for printing.
- `SOIF_PrintInitFn()` - creates a generic application-defined stream ready for printing. The given 'write_fn' is used to print the stream.

To create SOIF stream from a file or a string containing SOIF, use the following functions:

- `SOIF_ParseInitFile()` - creates a disk-based stream ready for parsing. The stream is created from an input containing SOIF syntax.
- `SOIF_ParseInitStr()` - creates a memory-based stream ready for parsing. The stream is created from an input containing SOIF syntax.

`SOIFStream` objects have a `caller-data` field, which you can use as you like:

```
void *caller_data;/* hook to be used by caller */
```

Use `SOIFStream_Parse()` to get the SOIF objects from the SOIF stream, and use `SOIFStream_Print()` to write SOIF objects to the SOIF stream.

When you've finished with the stream, close it using `SOIFStream_Finish()`. Use `SOIFStream_SetFinishFn()` to trigger the given finish_fn function.

The following example code takes a SOIF stream in stdin and prints each SOIF in the stream to stdout. Notice that this code uses `SOIF_ParseInitFile()` to create the `SOIFStream` to parse the input file, and uses `SOIF_PrintInitFile()` to create the stream to print the SOIFs to stdout.

```
SOIFStream *soifin = SOIF_ParseInitFile(stdin);
SOIFStream *soifout = SOIF_PrintInitFile(stdout);
SOIF *s;
while (!SOIFStream_IsEOS(soifin)) {
    if ((s = SOIFStream_Parse(soifin)) {
        SOIFStream_print(soifout, s);
        SOIF_Free(s);
    }
}
```

SOIF_PrintInitFile

```
NSAPI_PUBLIC SOIFStream *SOIF_PrintInitFile(FILE *file)
```

Creates a disk-based stream ready for printing.

SOIF_PrintInitStr

```
NSAPI_PUBLIC SOIFStream *SOIF_PrintInitStr(SOIFBuffer *memory)
```

Creates a memory-based stream ready for printing.

SOIF_PrintInitFn

```
NSAPI_PUBLIC SOIFStream *SOIF_PrintInitFn(int (*write_fn)(void *data,
char *buf, int bufsz), void *data)
```

Creates a generic application-defined stream ready for printing. The given `write_fn` is used to print the stream.

This function allows you to hook up your own routine for printing.

SOIF_ParseInitFile

```
NSAPI_PUBLIC SOIFStream *SOIF_ParseInitFile(FILE *fp)
```

Creates a disk-based stream ready for parsing. The file must contain SOIF-formatted data. The function reads SOIF data from the file `fp`.

SOIF_ParseInitStr

```
NSAPI_PUBLIC SOIFStream *SOIF_ParseInitStr(char *buf, int bufsz)
```

Creates a memory-based stream ready for parsing. The character buffer must contain SOIF-formatted data.

SOIFStream_Finish

```
NSAPI_PUBLIC int SOIFStream_Finish(SOIFStream *)
```

Call this to close the stream when you have finished with it.

SOIFStream_SetFinishFn

```
NSAPI_PUBLIC int SOIFStream_SetFinishFn(SOIFStream *,
int (*finish_fn)(SOIFStream *))
```

This allows you to hook up a function for cleaning up after the SOIF stream finishes its business. The `finish_fn` will be called when `SOIFStream_Finish()` has finished executing".

SOIFStream_Print

```
#define SOIFStream_Print(ss, s)
```

Prints another SOIF object to the SOIF stream `ss`. Returns 0 on success, or non-zero on error.

SOIFStream_Parse

```
#define SOIFStream_Parse(ss)
```

Parses and returns the next SOIF object in the SOIF stream.

SOIFStream_IsEOS

```
#define SOIFStream_IsEOS(s)
```

Returns 1 if the soif is the last one in its stream.

SOIFStream_IsPrinting

```
#define SOIFStream_IsPrinting(s)
```

Returns 1 (true) if the soif has been set up in a stream by `SOIF_PrintInitFile()` or `SOIF_PrintInitStr()`.

SOIFStream_IsParsing

```
#define SOIFStream_IsParsing(s)
```

Returns 1 (true) if the soif has been setup in a stream by `SOIF_ParseInitFile()` or `SOIF_ParseInitStr()`.

Filtering SOIF Objects

To support targeted parsing and printing, you can use the attribute filtering mechanisms in the SOIF stream. For each SOIF stream object, you can associate a list of "allowed" attributes. When printing a SOIF stream, only the attributes that match the "allowed" attributes will be printed. When parsing a SOIF stream, only the attributes that match the "allowed" attributes will be parsed.

`SOIFStream_IsAllowed()` and `SOIFStream_SetAllowed()` allow attributes, while `SOIFStream_IsDenied()` and `SOIFStream_SetDenied()` deny attributes. You can allow or deny an attribute, but not both.

SOIFStream_IsAllowed

```
NSAPI_PUBLIC boolean_t SOIFStream_IsAllowed(SOIFStream *ss,
char *attribute);
```

Indicates that the given attribute is allowed (that is, it can be printed or parsed).

SOIFStream_SetAllowed

```
NSAPI_PUBLIC int SOIFStream_SetAllowed(SOIFStream *ss,
char *allowed_attrs[])
```

Sets all the attributes in the *allowed_attrs* array to allowed.

SOIFStream_SetDenied

```
NSAPI_PUBLIC int SOIFStream_SetDenied(SOIFStream *ss,
char *denied_attrs[]);
```

Sets all the attributes in the *allowed_attrs* array to be denied (that is, they cannot be parsed or printed).

SOIFStream_GetAllowed

```
NSAPI_PUBLIC char **SOIFStream_GetAllowed(SOIFStream *ss)
```

Returns an array of all the attributes that are allowed.

SOIFStream_GetDenied

```
NSAPI_PUBLIC char **SOIFStream_GetDenied(SOIFStream *ss);
```

Returns an array of all the attributes that are denied.

Memory Buffer Management

You can use SOIF buffers in parsing or printing routines. They take care of memory allocation for inserting and appending. They are basically memory blocks that are easy for SOIF routines to use.

A SOIF Buffer is represented in a `SOIFBuffer` structure, that is created with the `SOIFBuffer_Create()` function and freed with the `SOIFBuffer-Free()` function. The `SOIFBuffer` structure provides the `append()`, `increase()`, and `reset()` functions for manipulating the data in the buffer.

SOIFBuffer_Create

```
NSAPI_PUBLIC SOIFBuffer *SOIFBuffer_Create(int default_sz);
```

The SOIFBuffer is used in `SOIF_PrintInitStr(SOIFBuffer *memory)`. Before you can print SOIF to memory, you need to create a buffer for output.

SOIFBuffer_Free

```
NSAPI_PUBLIC void SOIFBuffer_Free(SOIFBuffer *sb);
```

This is to release the memory buffer created by `SOIFBuffer_Create()`.

append

```
void (*append)(SOIFBuffer *sb, char *data, int n)
```

copies *n* bytes of data into the buffer.

increase

```
void (*increase)(SOIFBuffer *sb, int add_n)
```

increase the size of the data buffer by *addn* bytes.

reset

```
void (*reset)(SOIFBuffer *sb)
```

Resets the size of the data buffer and invalidates all currently valid data

Chapter

13

Using the RDM API To Accesss the Compass Server Database in C

This chapter discusses the use of the Compass Server RDM API to access the Compass Server and its database using C. The RDM API provides routines to parse, modify, or create resource description messages (RDMs) using C.

The robot generates RDs and saves them to the Compass Server database. You can get the RDs in SOIF format, use the RDM API to modify them, then have Compass import them back.

The RDM API is defined in the `rdm.h` file in the following directory in your Compass Server installation directory:

```
bin/compass/sdk/rdm/include
```

This chapter does not teach you how to use C. This chapter is restricted to discussing the use of C functions that come with the Compass Server RDM API .

What is RDM?

Resource Description Messages (RDM) is a messaging format which two processes can use to exchange resource descriptions across a network. In RDM, one process (a client or agent) sends a request RDM message to another process (a server) which processes the request, then sends a response RDM message, similiar to the HTTP/1.0 request/response model.

For more information about RDM, see:

`http://www.w3.org/TR/NOTE-rdm.html`

For example, you can send an RDM to a Compass Server database to request RDs that match a certain criteria (or scope). The Compass Server will send back a response RDM that contains the requested RDs (such as all the documents containing the string "style sheets.")

RDM Format Syntax

Each RDM message contains a header and a body. The header identifies the nature of the RDM message, while the body contains the data required to carry out the needed request (for example, scoping critiera). Both the header and body of the RDM message is encoded using SOIF. (Chapter 11, Using the SOIF API to Work with SOIF Objects gives more information about SOIF format.)

RDM Header

The RDM header section begins with a SOIF object whose schema-name is `RDMHEADER` which must contain as least the following attributes:

- rdm-version

  A string identifying the version of the message specification. (e.g., 1.0).

- rdm-type

  A string identifying the nature of this message.

- rdm-query-language: (required only for Request messages)

  A string which identifies which query language is used in the given request (for example, `"compass"`).

The following RDM header attribute is optional:

- catalog-service-id

  A CSID identifying the catalog to which the request/response applies (for example, `x-catalog://nikki.boots.com:80/techpubs`). If not present, the RDM server will use its default.

For more optional header parameters, see http://www.w3.org/TR/NOTE-rdm.html.

Some example RDM headers include:

```
@RDMHEADER { -
  rdm-version{3}: 1.0
  rdm-type{14}: status-request
}
```

```
@RDMHEADER { -
  rdm-version{3}: 1.0
  rdm-type{9}: RDM_RD_REQ
  rdm-query-language{8}:  compass
  catalog-service-ID{39}: x-catalog://nikki.boots.com:80/techpubs

}
```

RDM Body

The RDM message body contains the data needed to carry out the request. For example, if the header is `RDM_RD_REQ`, which indicates a query request, the body must contain the query criteria, or the scope. For example, if you are sending a request to find all documents that contain the string "netscape," then the body must be SOIF object whose schema-name is `RDMQuery` and whose `Scope` attribute is `netscape`:

```
@RDMQUERY { -
  scope{3}:netscape
}
```

If the RDM is a query request, the body of the message can also indicate which attributes of the resulting RDS are wanted, how many results you want, and how you want them sorted. For example, the following RDM body specifies that we're interested in the URL, title, author, and last-modified attributes. The result set contains at most 10 hits and the result set is ordered by the title attribute:

```
@RDMQUERY { -
  scope{3}:netscape
  view-attributes{x}: url,title,author,last-modified
  view-hits{x}: 10
  view-order{x}: title
}
```

If the RDM message is some other kind of request, such as a Schema-Description-Request or a Server-description-request or a Status request, the body of the message must contain appropriate data. See the following document for more details of RDM bodies:

`http://www.w3.org/TR/NOTE-rdm.html`

About the RDM API

The file `rdm.h` defines structures and functions for creating RDMs. The intent of these functions is to construct queries and instructions in C, which can be output as RDM and sent via the `sendrdm` program to a Compass Server for processing .

The basic structures are:

- `RDMHeader`
- `RDMQuery`

To support each of these main structures, there are additional structures, for example `RDMViewAttributes`, `RDMViewOrder`, `RDMViewHit`, which would each be used to represent attributes in an `RDMQuery`.

The RDM API defined in `rdm.h` provides functions for creating and modifying these structures. For example, the following statement:

```
RDMQuery *myquery = RDMQuery_Create("netscape");
```

creates an `RDMQuery` that corresonds to the following SOIF:

```
@RDMQUERY { -
  scope{3}:netscape
}
```

The following statement:

```
RDMQuery_SetViewAttr(myquery, "URL,Title");
```

modifies the `RDMQuery` so that it corresponds to the following SOIF:

```
@RDMQUERY { -
  scope{3}:netscape
view-attributes{x}: URL,Title,Author,Last-Modified
}
```

Both the RDMHeader and RDMQuery structures have soif fields, which contain the SOIF data for the header or the query. To get the SOIF data out of the RDMHeader or RDMQuery, you can access the soif field.

OK, so you can use the RDM API to create and modify RDMHeader and RDMQuery objects. But at some point you'll want to send them to the Compass Server to actually perform a query request. How do you do that?

The answer is that you can use the pre-built sendrm program, which you can find in the *compassdir*/bin/compass/bin directory. This program takes an RDM request (which is a message in SOIF format), sends it to the Compass Server, and outputs the results as a SOIF stream.

Example of Submitting a Query

Here is a sample program that generates an RDM for querying a Compass Server database. (You can get the source code as a separate file in example4.c.)

You must pipe the output to a temporary file then use the sendrdm program to post it to the Compass Server.The sendrdm program is at compassdir/bin/compass/bin/sendrdm.

You must change the definitions for MY_CSID, MY_SCOPE and MY_ATTR_VIEW in the code to suit your needs.

MY_CSID is the id of your Compass Server instance. This is the specific Compass Server that you created in the Compass Server Administration Interface. You can find the exact value in *compassServerDir*/config/csid.conf.

MY_SCOPE is the query string to search for. The default is to search for documents containing the string "netscape". For example, if you want to search for all documents containing the string "style sheets", then set MY_SCOPE to "style sheets."

MY_ATTR_VIEW is the attributes to be printed out, such as URL, title and description.

```
/******* Example Use of the RDM API to submit a query *******/

#include <stdio.h>
#include "soif.h"
#include "rdm.h"
```

```
#define MY_SCOPE "netscape"
#define MY_CSID "x-catalog://nikki.boots.com:6714/compass1"
#define MY_ATTR_VIEW "title,content-type"

int main(int argc, char *argv)
{
  RDMQuery *myquery = NULL;
  CSID *csid = NULL;
  RDMHeader *myheader = NULL;
  SOIFStream *out = SOIF_PrintInitFile(stdout);

  /* Create the RDMQuery and specify its scope */
  if(!(myquery = RDMQuery_Create(MY_SCOPE))) {
    perror("RDMQuery_Create\n");
    exit(-1);
  }

  /* Set the view attributes of the RDMQuery */
  RDMQuery_SetViewAttr(myquery, MY_ATTR_VIEW);

  /* Create the CSID that points to your Compass Server instance */
  if(!(csid = CSID_Parse(MY_CSID))) {
    perror("CSID_Parse\n");
    exit(-1);
  }

  /* Create the RDMHeader */
  if(!(myheader = RDMHeader_CreateRequest(RDM_RD_REQ, "compass", csid)))
  {
    perror("RDMHeader_CreateRequest\n");
    exit(-1);
  }

  /* print the RDMHeader to the output SOIF stream */
  /* print the RDMQuery to the output SOIF stream */
  (*out->print)(out, myheader->soif);
  (*out->print)(out, myquery->soif);

  /* free the structures and exit */
  RDMHeader_Free(myheader);
  RDMQuery_Free(myquery);
  SOIFStream_Finish(out);
  exit(0);

}

/*********** EOF ****************/
```

Running the Example

You can find the complete source code in example4.c.

1.) Edit the definitions for `MY_SCOPE`, `MY_CSID`, and `MY_ATTR_VIEW` as appropriate for your situation.

2.) Save the file in *compassdir*/bin/compass/sdk/rdm/examples.

3.) Create a makefile. You can find sample makefiles at *compassdir*/bin/compass/sdk/rdm/examples. Edit the makefiles to include example4. The following code shows the makefile with the changes needed for example4 in bold:

```
#  Makefile for SOIF/RDM SDK examples
#  Use make and cc.
CC              = cc
SDKDIR          = ..
SDKLIB          = $(SDKDIR)/lib/librdm.a
SDKINC          = $(SDKDIR)/include/
CFLAGS          = -I$(SDKINC) -DXP_UNIX
CFLAGS          += -DSOLARIS
#CFLAGS         += -DHPUX
#CFLAGS         += -DAIX
#CFLAGS         += -DIRIX
EXAMPLES        = example1 example2 example3 example4
all:    $(EXAMPLES)
example1:       example1.o
        $(CC) -o $@ $@.o $(SDKLIB)
example2:       example2.o
        $(CC) -o $@ $@.o $(SDKLIB)
example3:       example3.o
        $(CC) -o $@ $@.o $(SDKLIB)
example4:       example4.o
        $(CC) -o $@ $@.o $(SDKLIB)
```

4.). From the directory *compassdir*/bin/compass/sdk/rdm/examples, build the example as follows.:

```
Solaris:    gmake
```

```
NT: nmake -f makefile.win
```

5.). From the directory *compassdir*/bin/compass/sdk/rdm/examples, run the example4 program to generate the RDM file.

```
example4 > rdm.soif
```

The file rdm.soif will look something like:

```
RDMHEADER { -
  catalog-service-id{41}: x-catalog://newport.mcom.com:6714/newport
  rdm-version{3}: 1.0
  rdm-type{10}: rd-request
  rdm-query-language{7}: compass

}

@RDMQUERY { -
  view-attributes{18}: title,content-type
  scope{8}: netscape
}
```

6.). Send the SOIF contained in rdm.soif to the program sendrdm.

```
../../../bin/sendrdm  rdm.soif
```

The results of the sendrdm program will be a SOIF stream containing the results of the query, such as:

```
@RDMHEADER { -
  catalog-service-id{41}: x
  catalog://newport.mcom.com:6714/newport
  rdm-version{3}: 1.0
  rdm-type{11}:   rd-response
  rdm-response-interpret{51}:20 results out of 36281 hits across 88985
documents
}

@DOCUMENT {
  http://fury.netscape.com:999/it/newsref/pr/newsrelease417.html
  content-type{9}: text/html
  score{3}: 100
  title{17}: Comunicato stampa
}

@DOCUMENT {
  http://fury.netscape.com:999/it/newsref/pr/newsrelease374.html
  content-type{9}:        text/html
  score{3}:       100
  title{17}: Comunicato stampa
}
```

You can pipe the output of `sendrdm` (which is a `SOIFStream`) to another program to print the results of the query, or do whatever else with the results that you want.

See the section An Introductory Example in Chapter 11, Using the SOIF API to Work with SOIF Objects for an example of how to print information about SOIFs contained in a SOIF stream.

API Reference

The rest of this chapter discusses the RDM API, which you can find in *compassdir*/bin/compass/sdk/rdm/include/rdm.h.

Finding the RDM Version

RDM_Version

```
NSAPI_PUBLIC const char *RDM_Version(void);
```

Returns the version of the RDM library.

Creating and Freeing RDM Structures

For most `RDM`*X* structures, such as `RDMRequest`, `RDMQuery`, and so on, there are two or more creation functions. There is usually an `RDM`*X*`_Parse()` function, which takes SOIF arguments, and an `RDM`*X*`_Create()` function, which takes non-SOIF arguments.

There is also an `RDM`*X*`_Free()` function, which releases the object.

Several RDM structures also have an `RDM`*X*`_merge()` function, which merges data from a SOIF object into an existing `RDM`*X* structure.

RDMHeader

AN `RDMHeader` represents the header of an RDM. An `RDMHeader` structure has one public field, `soif`, which is a SOIF containing a collection of attribute-value pairs. The allowable attribute names of the attribute-value pairs in the SOIF are:

- `rdm-version` (required) -- This is set automatically if you use `RDMHeader_CreateRequest()` or `RDMHeader_CreateResponse()` to create the `RDMHeader`.
- `rdm-type` (required)
- `catalog-service-id` (recommended, this indicates which Compass Server instance to send the RDM to.)
- `rdm-query-language` (required for a request) -- For talking to the Compass Server, you can give this as `"compass"`.

Response RDMs also have several attributes, and you can find them in `rdm.h` if you are interested.

The following statements create an `RDMHeader` whose `RDMType` is `RDM_RD_REQ`, and whose query language is "compass". This RDM will be sent to the Compass Server instance `compass1` of the Compass Server at `http://nikki.boots.com:6714`.

```
CSID *csid = CSID_Parse("x-catalog://nikki.boots.com:6714/compass1");
RDMHeader *myheader = RDMHeader_CreateRequest(RDM_RD_REQ, "compass", csid);
```

The allowable values for `RDM-Type` are:

- `RDM_MSGTYPE_UNKNOWN` undefined or unknown message type
- `RDM_RD_UPD_REQ` -- requesting an RD update
- `RDM_RD_REQ` -- requesting an RD update (same as `RDM_RD_UPD_REQ`)
- `RDM_RD_DEL_REQ` -- requesting an RD delete
- `RDM_RD_UPD_RES` -- responding to an RD update request
- `RDM_RD_RES` -- responding to an RD update request (same as `RDM_RD_UPD_RES`)
- `RDM_RD_DEL_RES` -- responding to an RD delete request
- `RDM_SD_REQ` -- requesting a server description
- `RDM_SD_RES` -- responding to a server description request
- `RDM_SCH_REQ` -- requesting a schema description
- `RDM_SCH_RES` -- responding to a schema description request
- `RDM_TAX_REQ` -- requesting a taxonomy description

- RDM_TAX_RES -- responding to a taxonomy description request
- RDM_STAT_REQ -- requesting a status
- RDM_STAT_RES -- responding to a status description request

You can use the following macros to get and set the string values of the attributes:

```
RDMHeader_GetType(RDMHeader) /* returns RDMType */
RDMHeader_GetVersion(RDMHeader)
RDMHeader_GetQueryLanguage(RDMHeader)
RDMHeader_GetCSID(RDMHeader)
RDMHeader_GetResponseInterpret(RDMHeader)
RDMHeader_GetErrorMessage(RDMHeader)
RDMHeader_GetErrorNumber(RDMHeader)
```

```
RDMHeader_SetType(RDMHeader,RDMType)
RDMHeader_SetVersion(RDMHeader,stringvalue)
RDMHeader_SetQueryLanguage(RDMHeader,stringvalue)
RDMHeader_SetCSID(RDMHeader,stringvalue)
RDMHeader_SetResponseInterpret(RDMHeader,stringvalue)
RDMHeader_SetErrorMessage(RDMHeader,stringvalue)
RDMHeader_SetErrorNumber(RDMHeader,stringvalue)
```

RDMHeader_Parse

```
NSAPI_PUBLIC RDMHeader *RDMHeader_Parse(SOIFStream *ss);
```

RDMHeader_Create

```
NSAPI_PUBLIC RDMHeader *RDMHeader_Create(RDMType t);
```

RDMHeader_CreateRequest

```
NSAPI_PUBLIC RDMHeader *RDMHeader_CreateRequest(RDMType, char *ql,
CSID *)
```

RDMHeader_CreateResponse

```
NSAPI_PUBLIC RDMHeader *RDMHeader_CreateResponse(RDMType, char *ri,
CSID *);
```

RDMHeader_Free

```
NSAPI_PUBLIC int RDMHeader_Free(RDMHeader *r);
```

RDMHeader_Merge

```
NSAPI_PUBLIC int RDMHeader_Merge(RDMHeader *, SOIF *)
```

Merges data from a SOIF object into the `RDMHeader` object.

RDMQuery

AN `RDMQuery` represents the body of an RDM. An `RDMQuery` structure has one public field, `soif`, which is a SOIF containing a collection of attribute-value pairs. The allowable attribute names of the attribute-value pairs in the SOIF are:

- `scope` (required)
- `view-attributes` (optional)
- `view-hit` (optional)
- `view-order` (optional)
- `view-template` (optional)

You can use the following macros to get and set the string values of these attributes:

```
RDMQuery_GetScope(query)
RDMQuery_GetViewAttr(query)
RDMQuery_GetViewHits(query)
RDMQuery_GetViewOrder(query)
RDMQuery_GetViewTemplate(query)
```

```
RDMQuery_SetScope(query, value)
RDMQuery_SetViewAttr(query,value-list)
RDMQuery_SetViewHits(query,value)
RDMQuery_SetViewOrder(query,value-list)
RDMQuery_SetViewTemplate(query,value)
```

RDMQuery_Parse

```
NSAPI_PUBLIC RDMQuery *RDMQuery_Parse(SOIFStream *ss);
```

RDMQuery_Create

```
NSAPI_PUBLIC RDMQuery *RDMQuery_Create(const char *scope);
```

RDMQuery_Free

```
NSAPI_PUBLIC int RDMQuery_Free(RDMQuery *);
```

RDMQuery_Merge

```
NSAPI_PUBLIC int RDMQuery_Merge(RDMQuery *, SOIF *);
```

Other RDM Structures

In addition to `RDMHeader` and `RDMQuery`, the file `rdm.h` provides definitions and functions for the following auxilliary objects. See the file `rdm.h` in `compass/bin/sdk/rdm/include` for details.

- `RDMRequest`
- `RDMResponse`
- `RDMServer`
- `RDMView`
- `RDMViewHits`
- `RDMViewOrder`
- `RDMTaxonomy`
- `RDMClassification`
- `RDMSchema`

4

Using Java to Access the Compass Server Database

This part of the document discusses how to submit queries and add entries to the Compass Server database using Java applets.

To do these things, you need to use the Compass Server Java SDK, which is available at:

`http://developer.netscape.com/products/sdks/compass/download.html`

The Java SDK includes the Java classes needed for interacting with the Compass Server database, Java API documentation, and sample Java files for search and update applications.

This document does not teach you how to use Java. This document is restricted to discussing the use of Java functions that come with the Compass Server Java SDK for interacting with the Compass Server database.

For help with learning Java, check out Sun's Java Tutorial at:

`http://java.sun.com/docs/books/tutorial/index.html`

This document does not include the Javadoc for the Java API since it is included in the Java SDK. You can find it in the `docs` directory of the directory where you installed the Compass Server Java SDK. The first file to open is `packages.html`.

This part of the document contains the following chapter:

Chapter

14

Using Java to Access and Modify the Compass Server Database

This chapter discusses how to submit queries and add entries to the Compass Server database using Java.

To do these things, you need to use the Compass Server Java SDK, which is available at:

`http://developer.netscape.com/products/sdks/compass/download.html`

The Java SDK includes the Java classes needed for interacting with the Compass Server database, Java API documentation, and sample Java files for search and update applications.

The Compass Server Java SDK

The classes in the Compass Server Java SDK provide a Java interface for interacting with the CGI programs for accessing and updating the Compass Server database.

Using the Java SDK, you can build Java applets that submit searches to the Compass Server or add entries to the Compass Server database.

The Compass Server database contains resource descriptions for the indexed resources (such as documents). Each resource description is described in SOIF format, where SOIF stands for Summary Object Interchange.

For a discussion of SOIF format, see:

`http://www.w3.org/TR/NOTE-rdm.html#soif.`

Downloading and Installing the Compass Server Java SDK

To download the Compass Server Java SDK, point your browser to

`http://developer.netscape.com/products/sdks/compass/download.html`

Download the appropriate file for your platform and follow the installation instructions on the download page.

Running the Demo Applications

The Compass Server Java SDK comes with a `demo` directory that contains sample Java files and HTML files.

The `SearchPanel.java` and the `SubmitDemo.java` files are ready for compiling and running. You will need to edit the `SearchPanel.html` and `SubmitDemo.html` files to pass the information about your Compass Server to the SearchPanel and SubmitDemo applications.

Installing and Running the Search Panel Demo

1. Compile the `SearchPanel.java` file. Make sure the class path includes the directory where you installed the Compass Server Java SDK.
2. Copy or move the resultant `SearchPanel.class` file to the `bin/compass/java/demo` directory of your Compass Server installation directory. (If this directory does not already exist, create it.)
3. Edit the `SearchPanel.html` file in the `demo` directory of the Compass Server Java SDK.

The `codebase` parameter indicates the directory containing the main class and the ancillary classes for the applet. You can specify this in the form:

```
"http://host_name:compass_server_port/java"
```

The `code` parameter indicates the name of the main class file for the applet. In this case, the class file will be in the `demo` package (subdirectory) of the directory specified by `codebase`.

For example:

```
<applet codebase="http://CorporateCompass:80/java"
   code="demo.SearchPanel" width=480 height=75>
```

The `CGILocation param` parameter indicates the URL for the Compass Server.

```
<param name="CGILocation"
   value="http://host.domain.com:compass_server_port/">
```

For example:

```
<param name="CGILocation"
    value="http://www.CorporateCompass.com:80/">
```

The `host` param indicates the domain name of the host.

```
<param name="host" value="host.domain.com">
```

For example:

```
<param name="host" value="www.CorporateCompass.com">
```

The `port` param indicates the port number for the Compass Server.

```
<param name="port" value="compass_server_port">
```

For example:

```
<param name="port" value="80">
```

The `name` param indicates the name of the individual Compass Server instance. (This is a server that you created after starting the Compass Server administration interface.)

```
<param name="name" value="compass_server_name">
```

For example

```
<param name="name" value="guru">
```

Make these changes, save the file, and move it to wherever you like. When you open it in a web browser, the SearchPanel application should start. Be sure to open the Java console to see the results of submitting a search.

Installing and Running the Submit Demo

1. Compile the `SubmitDemo.java` file. Make sure the class path includes the directory where you installed the Compass Server Java SDK.
2. Copy or move the resultant `SubmitDemo.class` file to the `bin/compass/java/demo` directory of your Compass Server installation directory. (If this directory does not already exist, create it.)
3. Edit the `SubmitDemo.html` file in the `demo` directory of the Compass Server Java. Change the param values as appropriate.

This demo adds three resource descriptions to the Compass Database.

Using Java To Access the Compass Server Database

You can use the Compass Server Java SDK to write Java programs that interface with the `sendrdm` program retrieve information from the Compass Server database.

The main steps are:

1. Creating a Search Object
2. Executing A Query
3. Getting the Results of a Query

Creating a Search Object

The entry point for submitting searches is the `Search` class. You need to create a new `Search` object, then call `doQuery()` or `doIncrementalQuery()` on it to submit a search query.

The first thing you need to do is create a new `Search` object. The constructor syntax is:

```
public Search(
   String scope,
   String viewAttributes,
   String viewOrder,
   int viewHits,
   String queryLanguage,
   CSID csid,
   String CGILocation,
   String CGIName
)
```

You can omit `CGIName`, and it defaults to `"rdm/incoming"`.

The arguments for the constructor are:

`String scope`

The query string or scope, that is, the string being searched for.

`String viewAttributes`

A comma-delimited list of the SOIF attributes to be retrieved from the database, such as URL, Author, Description. For example:

`"score,url,title,description,classification"`

`String viewOrder`

The order by which to sort the attributes. This is a comma-delimited list of attributes. Use the minus sign to indicate descending order, and a plus sign to indicate ascending order. For example:

`"-score,+title"`

`int viewHits`

Maximum number of results to return. This value cannot be greater than 1024.

`String queryLanguage`

The Compass Server query language. You should use `"compass"` for this value.

`CSID csid`

This is the Compass Server ID. To get this argument, you can create a new CSID instance, passing it the page's codeBase, the Compass Server domain name, the Compass Server Port, and the Compass Server name.

For example, if your Compass Server host name is `http://www.yourcomany.com/`, it is installed on port 80, and it has a server instance called `guru`, you could create a CSID for it as follows:

```
CSID CS=new CSID(codebase, "http://www.yourcompany.com/", 80,
"guru")
```

A common way to provide the arguments to the constructor for CSID is to pass them as parameters in the `<APPLET>` tag that invokes the applet.

`String CGILocation`

The path for the location of the CGI directory of the hosting server, which is needed to access the `sendrdm` program. This argument has the form:

"http:*//host.domain.com:compass_server_port/*"

For example, if the Compass Server is installed on `www.yourcompany.com` on port 80, the value would be:

```
"http://www.yourcompany.com:80/"
```

A common way to provide this argument is to pass it as a parameter in the `<APPLET>` tag that invokes the applet.

`String CGIName`

This is the name of the CGI program that is used to access the Compass Server database. You can usually omit this argument. It defaults to `"/rdm/incoming"`.

Executing A Query

You can submit a query in two ways. The `doQuery()` function concatenates the query results into one big string buffer. If the string buffer is too big, the query fails. Only use this method when you know that the search results will not be too big. For example, you could use it to get the RD for a specific URL.

You can use the `getStringResult()` to get the results returned by `doQuery()` as a string of SOIF items

For larger queries use the `doIncrementalQuery()` method. This method returns `true` if there is more data available, and `false` if all data has been read.

```
public void doQuery()
```

```
public boolean doIncrementalQuery(
   int Increment,
   int ByteLimit)
```

The average resource descriptor size should be 5K-6K, (assuming you are using the default robot indexing setting of partial text and 4K length). So for 5 results 30K should be a big enough buffer.

Checking The Processing Status

You can use the method `Search.getStatus()` to get the status of the query process. You can use the method `Search.finished()` to check if the query process has finished or not.

```
public final int getStatus()
```

```
public final boolean finished()
```

The following tables shows the possible values returned by `getStatus()`.

Table 1

Processing Status	class variable name	value
Processing completed successfully.	COMPLETE	0;
Prepared for processing.	PREPARED	1;
Currently processing.	PROCESSING	2;
Abnormal end -- an error occurred.	ABEND	3;
.	EMPTY	4;

Getting the Results of a Query

The results from `Search.doQuery()` and `Search.doIncrementalQuery()` can be obtained using `Search.getSOIFResult()` or `Search.getStringResult()`. The next search will overwrite the previous results so you must process them after each query.

```
public SOIF getSOIFResult()
```

```
public String getStringResult()
```

`Search.getSOIFResult()` returns -1 if the status is not COMPLETE.

The function `Search.resultCount()` returns the number of results. The result count is based on being able to count the number of SOIF objects, so if `doSOIFParse` is `false`, -1 is returned. (The default for `doSOIFParse` is true.)

```
public int resultCount()
```

Formatting the Results as HTML

You can use the `Results` class to format the results from `Search.doQuery()` and `Search.doIncrementalQuery()` as HTML. The `Results` class has methods that wrap the search results inside HTML tags.

The method `setDisplay()` allows you to specify which attributes to display inside HTML tags. These attributes can be a subset of the attributes returned by the search query.

```
public void setDisplay(String s)
```

The attributes must be specified as a comma-delimited list, for example:

```
setDisplay("score", "url", "title", "description", "author");
```

The method `getDisplay()` returns a comma-delimited list of the attributes to display inside HTML tags.

```
public String getDisplay()
```

The method `setDisplayAsLink()` specifies which attributes should be displayed as links to the document's URL when the results are formatted as HTML (by invoking the `resultsToHTML ()` method.)

```
public void setDisplayAsLink(String s, boolean b)
```

For example, if you set both title and url to be displayed as links, then when you HTML-ize the results, both the url and the title for each document will be displayed as links to the document's URL.

```
setDisplayAsLink("title, true");

setDisplayAsLink("url, true");
```

The method `setScoreGif()` associates an image with a range of score values.

```
public void setScoreGif(int low,int high,String gif)
```

For example, the following statement associates the image "/images/fourstar.gif" with the score range 80 to 100.

```
setScoreGif(80, 100, "/images/fourstar.gif")
```

The function `toScoreGifHTML()` formats a score returning both percent and image, if any.

```
public String toScoreGifHTML(String sc)
```

```
public String toScoreGifHTML(float sc)
```

The `resultsToHTML()` function wraps HTML tags around the results. You can specify the start and end tag for each document and for each SOIF attribute. Alternatively you can specify just the start and end tags for each attribute.

```
public String resultsToHTML(SOIF soif,String preResult,String
postResult, String preField, String postField)
```

```
public String resultsToHTML(SOIF soif, String preField, String
postField)
```

For example, the following statement specifies that each document is in a table row, and each attribute is in a table cell:

```
resultsToHTML(search.getSOIFResult(),"<TR>", "</TR>", "<TD>", "</TD>");
```

The `Results` class is provided more as an example of a formatting class than as a finished class. You are encouraged to copy the `Results` class, and modify it to create a report format that suit your needs.

Working Through An Example Search Application

This section discusses the creation of an example application that uses the Java SDK to do a search query. The application that this example builds is similar, but not identical to, the demo search application that is shipped in the `demo` directory of the Java SDK.

The purpose of this example is to show how to use the Compass Server to submit a query to the database and how to extract the results from the Search object. The example application is very simple, and limits is use of Java to achieving the goals of the example. It creates a Java applet that presents the user with a text field in which to enter a search query, and a "Search" button to

initiate the search. The results of the query are saved in a Search object. The query results are displayed in the Java console as straight text and also as HTML-formatted text.

Import the Necessary Files

In your favorite editor or Java development environment, open a file called `QueryExample.java`. At the top of the file declare this class to be in package `demo`, and import the necessary classes. The class will use the `CSID`, `Results`, `SOIF`, and `Search` classes in the `soif` package. It will use the `Applet` class in the `applet` package, and it will use several classes in the `awt` package. The `soif` package is provided as part of the Compass Server Java SDK, while `applet` and `awt` are standard Java packages.

```
package demo;
import soif.CSID;
import soif.Results;
import soif.SOIF;
import soif.Search;

import java.applet.Applet;
import java.awt.Button;
import java.awt.Event;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.Color;
```

Define the QueryExample Class

The class `QueryExample` is an applet, so it must extend the class Applet.

```
public class QueryExample extends Applet
```

Define some variables for use in the class. The variable `RETURN` represents the ASCII code for the return key on the keyboard. The `searchTF` variable represents the text field in which the user enters their search query, and `searchBtn` represents the button that they press to submit the search query. The variable CGIlocation represents the cgi location of the `sendrdm.cgi` program (which executes queries in the database), and `csid` represents the specific Compass Server instance ID.

```
public class QueryExample extends Applet
{
    public final static int RETURN = 10;
    TextField   search
```

```
    TFButton      searchBtn;

    String      CGILocation;
    CSID        csid;
```

Initialize the Applet to Create the Search Form

Define the `init()` method. The first thing it does is print a brief greeting in the Java console window. (When you run the applet, be sure to open the Java Console to see all the output.)

```
public void init()
{
    System.out.println("Welcome to the Example Search Application");
```

The next task is to define the appearance of the applet. Its background will be light blue (or change the color to suit your liking.) The applet needs to have a text entry field where the user enters the search query, and a search button that initiates the search.

```
/* Set the background to light blue*/
setBackground(new Color(180, 220, 250));
```

```
/* Add the text entry field and the search button */
searchBtn   = new Button("Search");
searchTF    = new TextField("Netscape", 60 );
setLayout(new FlowLayout(FlowLayout.CENTER ) );
add(searchTF);
add(searchBtn);
```

Before submitting the search, you need to create a Search object. The constructor for the Search class takes several arguments. Two of these arguments are CGILocation and CSID (Compass Server ID.) The `init()` method of `QueryExample` is a good place to initialize the CGILocation and CSID arguments, since they do not change for the duration of the applet session.

This applet will be invoked by an `<APPLET>` tag that has `PARAMETER` attributes for CGILocation, host, port, and name. You can use these parameters to generate the values for CGILocation and CSID (which have already been declared as instance variables.)

```
    CGILocation = getParameter("CGILocation");
```

```
    csid = new CSID(
                getCodeBase().toString(),
                getParameter("host"),
```

```
                getParameter("port"),
                getParameter("name")
    );
/* end of init method */
}
```

Define the handleEvent() Method

When the user presses the Search button in the form or presses the Return key on the keyboard, the search process is initiated. Thus you need to define a `handleEvent()` method that checks if the search button was pressed or if the Return key was released while the text field was being edited. If any other event happened, do nothing. If these criteria are met, continue. (The example `handleEvent` shown here is for Java 1.02.)

```
public synchronized boolean handleEvent(Event e)
{
    if ((( e.id == Event.KEY_RELEASE)
        && (e.target == searchTF)
        && (e.key == RETURN))
    || (( e.id == Event.ACTION_EVENT)
       && (e.target == searchBtn))
    )
```

While this method is in progress, make sure the user cannot type in the text field and that nothing else happens if the search button is pressed.

```
searchTF.setEditable(false);
searchBtn.disable();
```

Next, we need to construct the query to submit to the database. This involves creating a Search object. The arguments for the constructor for the Search class are:

1. the search string. In this case, we can get the value of the `searchTF` text field to use as the search criteria.

2. a comma delimited list of the attributes you want returned (such as description, title, and so on). In this case, we want to get the title, score, rule, and description of all documents that match the query.

3. a comma delimited list of the attributes to be used to sort the results. A minus sign indicates descending order, a plus sign indicates ascending order. In this case, sort the results by decreasing numerical score value, and use alphabetical order of the title as the secondary sort order.

4. the maximum number of results to return.(This value cannot be more than 1024.)

5. the Compass Server query language, which you can specify as `"compass"`.

6. CSID (Compass Server ID) instance. In this case, we have already defined the variable `csid` to represent the Compass Server ID.

7. path to the cgi directory for sendrdm. In this case, we have already defined the variable `CGILocation` for this purpose.

```
Search search = new Search(
   searchTF.getText(),
   "title,description,score,url",
   "-score,+title",
   10,
   "compass",
   csid,
   CGILocation
);
```

Having created the `Search` object, use it to submit the query.

```
search.doQuery();
```

The results are stored in the Search object. Now do something with the results. The functions `doSomethingWithResults()` and `displayHTMLResults()` will be defined in this file. They each show a different way of extracting the results from the `Search` object.

```
   doSomethingWithResults(search);
   displayHTMLResults (search);
```

Switch the text field and the search button back on. Return true and end the conditional case for handling the search submission.

```
   searchTF.setEditable(true);
   searchBtn.enable();
   return true;
}
```

Be sure to return `super.handleEvent()` if the user did not submit a query to ensure that when the user types in the text field, the text entry events are handled correctly.

```
return super.handleEvent (e);
/* end of handleEvent */

}
```

Show the Results in the Java Console

The example application includes a simple function that displays the query results in the Java console window. In reality, you would do more with the results than just print them to the console window, but once you know how to get the results out of the Search object, it is up to you what you do with them. You can use standard Java functionality to process the results in any way you like.

The class `Search` has a method `getSOIFResult()` that returns a `SOIF` object representing a single result. Each `SOIF` object has a `next` instance variable that points to the next result, if there is another one. You can use the `getValue()` method of a `SOIF` object to get the value of a particular field, for example, `getValue("title")` gets the title of a `SOIF` object.

The example function `doSomethingWithResults()` first prints the total number of results, then gets each result in turn as a `SOIF` object, then prints its url, title, description, and score in the Java console window.

```
public void doSomethingWithResults(Search search)
{

/* Get a result count. -1 indicates an error,
** which can also be discovered by using search.getStatus().
*/
System.out.println(
   "number of hits: ["
   + search.resultCount()
   + "]"
);
/* Examine the results of the search. The following
** block loops through a list of SOIF instances.
*/

for (SOIF soif = search.getSOIFResult();
   soif != null;
   soif = soif.next
)
{
/* use the getValue() method to get a value associated
** with the specified attribute.
*/
String u = soif.getValue("url");
String t = soif.getValue("title");
String d = soif.getValue("description");
String sc = soif.getValue("score");
/* Print the results. */
System.out.println(
```

```
   "TITLE:       " + t +
   "\nURL:         " + u +
   "\nSCORE:       " + sc +
   "\nDESCRIPTION: " + d +
   "\n---------------------------------\n\n"
   );
}
}
```

Displaying the Results in HTML Format

The example `displayHTMLResults()` displays the results as HTML-formatted text. This function uses the pre-defined `resultsToHTML()` function of the `Results` class. In this case, each resulting document is wrapped in the HTML tags for a table row, while each field within the document is wrapped in the HTML tags for a table cell.

This function sends the HTML-formatted results to the Java console. In practice, you would most likely want to extend this function (using standard Java code) to send the results to a web page.

```
public void displayHTMLResults (Search search) {
   /* Use a Results object to format the results in HTML */
   Results results = new Results();
   results.setDisplay("title,description,score,url");
   results.setDisplayAsLink("title", true);
   results.setScoreGif( 0, 50, "okmatch.gif");
   results.setScoreGif(51, 100, "goodmatch.gif");
   System.out.println("Here comes the HTML-formatted text...");
   System.out.println("");
   System.out.println(
   results.resultsToHTML(
   search.getSOIFResult(),
      "<tr>\n",
      "</tr>\n\n",
      "<td>\n",
      "\n</td>\n"
      )
   );
}
/* End the class */
}
```

For the complete code for the class file, see `QueryExample.java`.

Creating the HTML File to Display the Applet

After you've compiled the QueryExample.java file, copy the resulting class file to the `bin/compass/java/demo` directory of your Compass Server installation directory.

Next, you need to define the HTML file to display the applet, as shown below. (You can use whatever colors you like for the `text`, `bgcolor`, `vlink`, `link`, and `alink` parameters of the `body` tag.)

```
<html>
<head>
<title>Search Query Example</title>
</head>

<body text="#0000FF" bgcolor="#FFFFFF"
   vlink="#B8860B" link="#F8C64B" alink="#D3D3D3">
<h1>Search Panel Example</h1>

<p><center>
<applet codebase="http://yourCompassServerName:Port#/java"
   code="demo.QueryExample"
   width=480 height=75 align=left>

<param name="CGILocation" value="http://CompassServerName:Port#/">
<param name="host" value="CompassServerName">
<param name="port" value="port#">
<param name="name" value="CompassServerInstanceName">
This interface requires a Java aware browser.
</applet>
</center>
<BR CLEAR=ALL>
<p><BR>
<P>Note: Search results are sent to the Java console.</p>
</body>
</html>
```

The `<APPLET>` tag has parameters for `CGILocation`, `host`, `port`, and `name`, since these values need to be passed to the Java class. The `host` is the Compass Server name, whereas `name` is the name of the Compass Server instance (which you created after starting the Compass Server administration interface.). For example, if you installed the Compass Server at `www.CompassName.com` on port 80, and created a server instance called `guru`, then the `<APPLET>` tag would be:

```
<applet codebase="http://CompassName:80/java"
   code="demo.QueryExample" width=480 height=75 align=left>
   <param name="CGILocation"
      value="http://www.CompassName.com:80/">
   <param name="host" value="www.CompassName.com">
   <param name="port" value="80">
```

```
   <param name="name" value="guru">
This interface requires a Java aware browser.
</applet>
```

Using Java To Add Entries to the Compass Server Database

The `communications` package in the Compass Server Java SDK contains classes for communicating with the Compass Server database. These classes connect to the database through a CGI program, `rdsubmit`. You can use these classes to import data in SOIF format into the Compass Server database. You can import new RDs into the database, or modify existing ones.

The basic process for adding entries to the database is to convert the input to SOIF and then add it to the database using `rdsubmit`.

To add entries to the database, convert the input to SOIF format, then pass the input to the `doWrite()` method of the `PostConnection` class. The `dowrite()` method invokes the `rdsubmit` program to submit the data.

PostConnection.doWrite()

Use the `doWrite()` method of the `PostConnection` class in the `communications` package to write data to the Compass Server database.

The signature is:

```
public static String doWrite(
   String CGIlocation,
   String rdm-message,
   boolean b
)
```

The arguments are:

`String CGIlocation`

The URL to the `rdsubmit` program on the Compass Server instance. For example:

```
"http://guynt:11000/compass-compassjan/bin/rdsubmit"
```

`String rdm-message`

This argument is an RDM in SOIF format to be submitted to the Compass Server database. See section More Details on the rdm-message Argument for more information about this argument.

`Boolean b`

This argument specifies whether or not to return the string result. If the value is `false`, then `doWrite()` returns a `null` string. If the value is `true`, the result string is returned. The result string is in the form of a SOIF string and is the result the server generated in response to the submission.

More Details on the rdm-message Argument

The `PostConnection.doWrite()` method submits a resource description message (RDM). RDM messages must be in SOIF format. Each RDM must have a header and a body. The header contains information about the kind of message, where it needs to go, and what it needs to do. The body contains the data for the message. For more information about RDM and SOIF formats, see:

```
http://www.w3.org/TR/NOTE-rdm.html
```

The second argument for `PostConnection.doWrite()` must be an RDM. In this specific case, we need an RDM whose header is of type `@request`, (since we are making a request), and whose body is of type document (since we are sending a document to be submitted to the Compass Server database.)

A SOIF object consists of a schema name (such as `@request` or `@document`), a URL, and a list of attribute-value pairs. The `soif` package in the Compass Server Java SDK contains a class `AVPairs` that has a class method `toSoif()`. This method takes an attribute name and value and generates an attribute-value pair in SOIF format. For example:

```
AVPairs.toSoif("title", "All about SOIF");
```

generates a result something like:

```
title{20}: All about SOIF
```

You can use the `AVPairs.toSOIF()` method to help create an RDM in SOIF format.

Here is an example of constructing an RDM that can be used as the second argument to `PostConnection.doWrite()`.

First, create a string buffer:

```
StringBuffer sb = new StringBuffer();
```

Write the header part of the RDM to send to the database. SOIF objects of type `@Request` do not have an associated URL.

```
sb.append("@REQUEST { -\n");
```

An RDM that is requesting an update to a Compass Server has the following attribute-value pairs:

- `submit-csid`
- `submit-type`
- `submit-operation`
- `submit-view`

Write attribute-value pairs for each of these attributes to the string buffer:

```
sb.append(AVPairs.toSOIF("submit-csid", x-catalog://nikki.boots.com:80/compass1
+ AVPairs.toSOIF("submit-type", "persistent")
+ AVPairs.toSOIF("submit-operation", "merge")
+ AVPairs.toSOIF("submit-view", "title,author,description")
+ "}\n");
```

Write the body part of the RDM. We'll be sending a resource description for a document, whose URL is "http://www.best.com/~jocelyn/resdogs.index.htm", whose title is "Saving English Springer Spaniels," whose author is Jocelyn Becker, and whose description is "English Springer Spaniels in need of homes."

```
sb.append("\n@DOCUMENT{\n"
+ "http://www.best.com/~jocelyn/resdogs.index.htm\n"
+ AVPairs.toSOIF("title", "Saving English Springer Spaniels")
+ AVPairs.toSOIF("author", "Jocelyn Becker")
+ AVPairs.toSOIF("description", "English Springer Spaniels in need of
homes")
+ "}"
```

The string buffer now contains the following RDM in SOIF format:

```
@REQUEST { -/
  submit-csid{20}: x-catalog://nikki.boots.com:80/compass1
  submit-type{23}: persistent
  submit-operation{29}: merge
  submit-view{30}: title,author,description
}
@DOCUMENT { http://www.best.com/~jocelyn/resdogs/index.html
  title{35}: Saving English Springer Spaniels
  author{37}: Jocelyn Becker
  description{39}: English Springer Spaniels in need of homes
}
```

You can convert the string buffer to a string, and use it as the second argument for `PostConnection.doWrite()`. For example, the following statement writes the new resource description to the Compass Server database:

```
result = PostConnection.doWrite(CGILocation, sb.toString(),true);
```

Example

See the `SubmitDemo.java` class file and `SubmitDemo.html` HTML file in the `demo` directory of the Java SDK for an example.

Appendix

A

Glossary

Attribute-Value Pair An attribute and a value, such as `title` and `John Brown`. Attribute values are used in SOIF objects and in parameter blocks (or pblocks).

Browse Page A page that is displayed when an end user browses a category using the Compass Server. or goes to the Compass Server End User Page for the first time.

Compass Server End User Page The page that users see when they use the Compass Server to search or browse. This is also the page that is presented when a developer accesses a Compass Server as a client.

Configuration File A file that lists the pattern files needed to make up a Compass Server End User Page.

Directive A statement that uses a particular format to invoke a function (such as a robot application function) and pass parameters to the function in a parameter block. For example, the following directive invokes the `enumerate-urls` function, and passes parameters for `max` and `type`.

```
Enumerate fn=enumerate-urls max=1024 type=text/html
```

Matching Category A category that matches a search query. It is returned as a result of a search submission.

Matching Document A document that matches a search query. It is returned as the result of a search submission.

Parameter Block/Pblock	A C data structure that holds pairs of attributes and values.
Pattern File	A file that defines the content for a component of a Compass Server End User Page.
Resource Description/RD	A resource description is an entry in the Compass Server database that describes a resource, such as a document, on a network. Resource descriptions are stored in SOIF format.
Resource Description Message/RDM	RDM standards for *resource description messages*. An RDM has a header and a body. The header contains information about the message, and the body contains the data for the message. An RDM uses SOIF syntax. Processes can communicate using RDM, for example, queries to the Compass Server database must use RDM format.
Robot Application Function/RAF	A function that can be used in robot filter configuration files. User-defined robot application functions are also called plug-in functions. These functions are invoked by directives.
Schema-name	The schema, or type, of a SOIF. For example, a SOIF for a document has the schema-name `@DOCUMENT`, while a SOIF for an RDM header has the schema-name `@RDMHeader`.
Search Results Page	The page that is displayed when an end user submits a search query to the Compass Server.
SOIF	SOIF stands for Summary Object Interchange format. Each SOIF object has a schema-name that identifies the kind of SOIF. It has a URL and a set of attribute-value pairs. Resource descriptions in the Compass Server database are stored in SOIF. For example, a document resource has the schema name `@DOCUMENT`, a URL for the document, and a set of attribute value pairs that describe the document, such as title, description, author, and so on..
SOIF Attribute	Each resource description in the Compass Server database has multiple attributes or fields. These attributes are known as SOIF attributes, where SOIF stands for Summary Object Interchange Format.
View Template or UI Template	A configuration file and its associated set of pattern files, that together determine the appearance of the Compass End User Page.