The Definitive Guide to Stellent Content Server Development

Brian “Bex” Huff

Apress®

The Definitive Guide to Stellent Content Server Development

ISBN-13: 978-1-59059-684-5

ISBN-10: 1-59059-684-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewer: Samuel White
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole LeClerc
Copy Editor: Nancy Sixsmith
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Gunther
Compositor: Molly Sharp
Proofreader: Dan Shaw
Indexer: Toma Mulligan
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 3

Using HCSTs

Hypertext Content Server Template (HCST) pages are a quick and easy way to create custom dynamic web pages in the Content Server. A developer can execute any service with a properly constructed HCST. Any of the hundreds of standard services can be executed from a form post, including check-in, workflow processing, subscriptions, or searches. A small subset of read-only services, such as search and content info, can be executed directly on the page with IdocScript.

The most common use of HCSTs is to create check-in screens, search pages, and portal pages. Pages that execute custom services or database queries are also possible, but they require a bit more work. This chapter covers the most common uses of HCST pages.

Simple HCSTs

Any IdocScript function or variable can be rendered on an HCST. For example, a simple page to say hello to the current user looks like this:

```
<html>
<head></head>
<body>
<h1>Hello <$UserName$></h1>
</body>
</html>
```

To render this page, simply save the preceding text into a file named `hcst-hello.hcst` and then check that file into the Content Server. Run a search to find it and then click on the `weblayout` URL for that file. You will see a page similar to Figure 3-1.

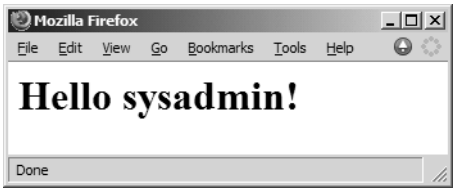

Figure 3-1. *Simple Hello User HCST*

In addition, any standard dynamichtml include can be referenced on this page. These includes are used throughout the core templates to establish a standard look and feel for all Content Server templates. To give the HCST the standard look and feel, four core includes are needed:

```
<html>
<head>
<title>Hello</title>
<$include std_html_head_declarations$>
</head>
<$include body_def$>
<$include std_page_begin$>

<h1>Hello <$UserName$></h1>

<$include std_page_end$>
</body>
</html>
```

When viewed in the web browser, the page looks like any other page in the Content Server (see Figure 3-2):

Figure 3-2. *Hello User HCST with standard layout*

This HCST is a good starting point for any page that a developer wants to present to the user. The std_page_begin and std_page_end includes are on almost all HTML templates used by the Content Server. The body_def include is found on most pages; if not, body_def_internal is used:

```
<html>
<head>
<title>Hello</title>
<$include std_html_head_declarations$>
</head>
<body <$include body_def_internal$> >
```

```
<$include std_page_begin$>

<h1>Hello <$UserName$></h1>

<$include std_page_end$>
</body>
</html>
```

The `std_html_head_declarations` include is almost always present, but it does not contain enough information for some pages. Search and contribution pages need additional JavaScript code to be defined in the HEAD of the HTML page. On these pages, specialized versions of these includes are used. They will be covered in the next few examples, which demonstrate how to run services from an HCST and how to make custom search and check-in pages.

For a list of the commonly used includes, see Appendix D.

Search Results Portal

There are several ways to execute the core Content Server features in an HCST. One simple and common method is to use the `executeService` IdocScript function. This function enables you to run a small subset of core Content Server services directly in IdocScript (for example, obtaining the content info for an item or running a search).

This example is a page that displays a list of search results. It can be used to obtain a list of the most recent items, all items by a specific author, or items that are from a specific department. To obtain a list of all items that have the word `test` in their titles, create a file with the following code:

```
<html>
<head>
<title>Search</title>
</head>
<body>

<h3>Test Documents</h3>

<$if not QueryText$>
<$QueryText="dDocTitle <substring> `test`"$>
<$endif$>

<$executeService("GET_SEARCH_RESULTS")$>
<ul>
<$loop SearchResults$>
<li><a href="<$URL$>"><$dDocTitle$></a></li>
<$endloop$>
</ul>

</body>
</html>
```

As in the previous example, this text needs to be saved to a file named `hcst-search.hcst` and checked-in to the Content Server. Be sure to give this HCST the content ID value of `hcst-search`. When viewed from the web, it looks like Figure 3-3.

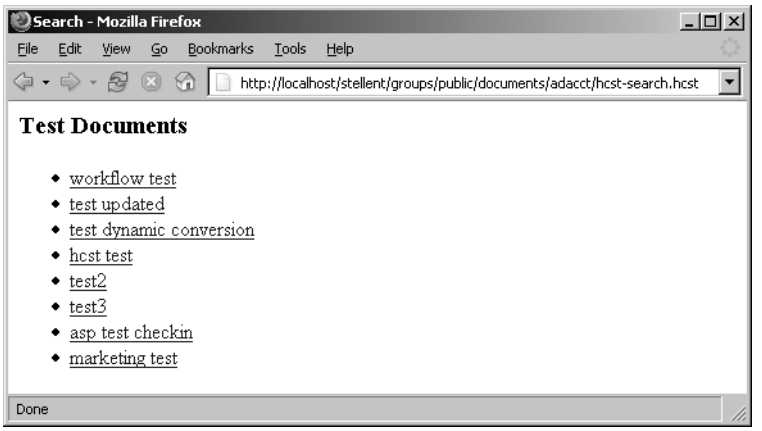

Figure 3-3. *Simple HCST search portal*

The content ID is important in this case because it is used in the URL to the item. Notice the URL shown in Figure 3-3:

```
http://localhost/stellent/groups/public/documents/adacct/hcst-search.hcst
```

This URL contains several pieces of metadata that you filled in on the check-in page. The first piece is `public`, which denotes that this content item is in the *Public* security group. The next piece is `adacct`, which means that this item has a content type of *ADACCT*. The final piece is the web filename, which corresponds to the content ID of the item, followed by a dot, and then the extension for this file's format. In this case, the web filename equals `hcst-search.hcst` because the content ID is `hcst-search` and the file extension is `hcst`. It is pure coincidence that this web filename is equal to the title of the checked-in file.

These URLs seem verbose, but they are useful for many purposes. One of them is fast security checks, as mentioned in Chapter 2. Another is to make easy references from one file to another. If two items have the same security group and content type, they reside in the same web folder, and links between them are simple (this becomes important later).

This page runs the service `GET_SEARCH_RESULTS` with all the variables currently defined on the page, including any variable explicitly set in IdocScript and any parameter from the URL of the request. For example, if the URL looks like this, the results from the search are sorted according to their titles:

```
http://localhost/stellent/.../hcst-search.hcst?SortField=dDocTitle
```

Likewise, the value for `SortField` can be set on the page itself, just as `QueryText` is. The following change in the code yields a similar result:

```
<$if not QueryText$>
<$QueryText="dDocTitle <substring> `test`"$>
```

```
<$endif$>
<$SortField="dDocTitle"$>
```

After executing this search, the response data is available on the page as IdocScript variables. For example, the variable `TotalRows` is now available, telling us how many results are present. Tables of data in IdocScript are called `ResultSets`. In this case, the `ResultSet` named `SearchResults` contains all the needed data. To display the results, use the following code:

```
<$loop SearchResults$>
<li><a href="<$URL$>"><$dDocTitle$></a></li>
<$endloop$>
```

For each row in the results, one HTML bulleted list item is output to the page. Each row in the `ResultSet` has multiple columns, or fields. There exists one field for each piece of custom metadata for this item, as well as standard document metadata. In the preceding example, you extracted the values for `URL` and `dDocTitle` and output them to the page as HTML. Other values are present, such as `dDocAuthor`, `dDocType`, and `xComments`.

More information about IdocScript syntax and looping is available in Appendix A. More information about the `GET_SEARCH_RESULTS` service is in Appendix E.

As mentioned, this search is run based on the variables currently in the local data, including any parameter in the URL to the `hcst-search.hcst` page itself. This line is used to set a default value for `QueryText` if one is not present in the URL:

```
<$if not QueryText$>
<$QueryText="dDocTitle <substring> `test`"$>
<$endif$>
```

This page can also be used as the result page for a search form. For example, the following HTML page can be used as a simple search form:

```
<html>
<head>
<title>Search</title>
</head>
<body>

<form action="hcst-search.hcst">
<input name="QueryText" value="dDocTitle <substring> `test`">
<input type="submit">
</form>

</body>
</html>
```

Save this HTML to a file named `hcst-search-simple.hcst`. Next, check it into the Content Server with the same security group and content type as `hcst-search.hcst`, after which the `action` parameter in the HTML form points to the URL of the previously created search portal. When the user clicks Submit, the form data is sent to the search portal. Because the value for `QueryText` is submitted as well, the search portal uses the submitted value to run the search.

Congratulations! You have just created your first web application with the Content Server!

The default search on the portal page uses the Verity Full Text search engine to find all content with `test` in their titles. Using this engine, you can query the metadata in the document along with the text in the document. This is extremely powerful, especially if you have XML or keywords embedded in your documents that are not present in the metadata.

However, this example does not need to query the contents of the documents, just the title metadata. If you have version 7.0 or later of the Content Server, you can bypass the search engine and query the database directly. This procedure is faster when it comes to metadata queries that do exact matches, but is sometimes slower when doing wildcard searches. Querying the database directly requires slightly different parameters:

```
<html>
<head>
<title>Search</title>
</head>
<body>

<h3>Test Documents</h3>

<$if not QueryText$>
<$QueryText="dDocTitle LIKE '%test%'"$>
<$endif$>
<$SearchEngineName="DATABASE"$>
<$executeService("GET_SEARCH_RESULTS")$>
<$loop SearchResults$>
<li><a href="<$URL$>"><$dDocTitle$></a>
<$endloop$>

</body>
</html>
```

Notice that the value of `SearchEngineName` is `DATABASE`, which instructs the Content Server to execute the query using the database. It requires slightly different query syntax than the previous example. In version 7.0, the `QueryText` must be pure SQL. In version 7.5, this is not required, but still recommended. In either case, the result page will still look like Figure 3-3.

Not all Content Server services can be executed with the `executeService` IdocScript function. The Content Server supports hundreds of different service requests, but most are used by the administrative applets and are not generally useful on an HCST. Only a small handful of service requests are relevant to developers. See Appendix E for a list of these services.

Advanced Search Page

As mentioned in the previous example, it is fairly simple to create an HTML form that can submit to an HCST and run a search. However, doing so requires the HCST developer to write some custom code to properly assemble the `QueryText`. This is not complicated; executing a substring search on a metadata field requires fairly simple syntax. However, it becomes more complicated when you add more fields and more options. Integer field, date field, and precise matches all add complexity to assembling the `QueryText`.

For complex search pages with multiple fields, it makes sense to take advantage of the resources that the Content Server uses. In particular, it makes sense to use the same JavaScript functions and create forms with similar fields. This example presents one possible way to create an advanced search page similar to the Content Server's search pages.

This example uses these Content Server resources to create an advanced search page. It also demonstrates how to use the `urlTemplate` parameter to redirect a service to a custom template. It finally displays the service response in a way that handles multiple result pages.

To begin, create the file `hcst-advanced-search.hcst` with the following code:

```
<html>
<head>
<$PageTitle="Guest Search Page"$>
<$if isTrue(IsLoggedIn)$>
    <$PageTitle="Search Page For " & UserName$>
<$endif$>
<$include std_query_html_head_declarations$>
<script language="JavaScript">
<$include query_form_submit_script$>
</script>
</head>

<$include body_def$>
<$include std_page_begin$>

<h3 class=pageTitle>HCST Search Page</h3>
<form name="QUERYTEXTCOMPONENTS">
<input type=hidden name="QueryText" value="">
<table>
<tr>
    <td align=right><span class=searchLabel>Title</span></td>
    <td>
        <input type="hidden" name="opSelected"
            value="hasAsSubstring">
        <input type="text" size=30 name="dDocTitle"
            value="">
    </td>
</tr>
<tr>
    <td align=right><span class=searchLabel>Content ID</span></td>
    <td>
        <input type="hidden" name="opSelected"
            value="hasAsSubstring">
        <input type="text" size=30 name="dDocName" value="">
    </td>
</tr>
<tr>
    <td align=right><span class=searchLabel>
        Release Date From</span></td>
```

```
    <td align=left>
        <input type="hidden" name="opSelected" Value="dateGE">
        <input type="text" size=12 maxlength=20
            name="dInDate" value="">
        <span class=searchLabel>To</span>
        <input type="hidden" name="opSelected" Value="dateLess">
        <input type="text" size=12 maxlength=20
            name="dInDate" value="">
    </td>
</tr>
<tr>
    <td align=right><span class=searchLabel>Full Text</span></td>
    <td><input type="text" name="FullTextSearch" size=30></td>
</tr>
</table>
</form>
<$c="This page is hcst-advanced-search.hcst, and we wish to render the results
with the template hcst-advanced-search-results.hcst"$>
<$urlTemplate = strSubstring(fileUrl, 0, strLength(fileUrl)-5)
    & "-results.hcst"$>
<form name="SEARCHFORM" method="GET"  action="<$HttpCgiPath$>">
<input type=hidden name="IdcService" value="GET_SEARCH_RESULTS">
<input type=hidden name="urlTemplate" value="<$urlTemplate$>">
<input type=hidden name="QueryText" value="">
<input type=hidden name="ResultCount" value="5">
<table>
<tr>
    <td><span class=searchLabel>Sort By:</span></td>
    <td><select name="SortField">
        <option selected  value="dInDate">Release Date
        <option  value="dDocTitle">Title
        <option  value="Score">Score
        </select>
    </td>
    <td><select name="SortOrder">
        <option value="Asc">Ascending
        <option selected value="Desc">Descending
        </select>
    </td>
    <td>
    <input type=submit value ="Search" onClick="submitFrm(false)">
    </td>
</tr>
</table>
</form>

<$include std_page_end$>
```

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

Next, check this file into the Content Server with the same security group and content type as the previous two HCSTs. Be sure to give the file the content ID `hcst-advanced-search`. When viewed from the web, this page looks like Figure 3-4.

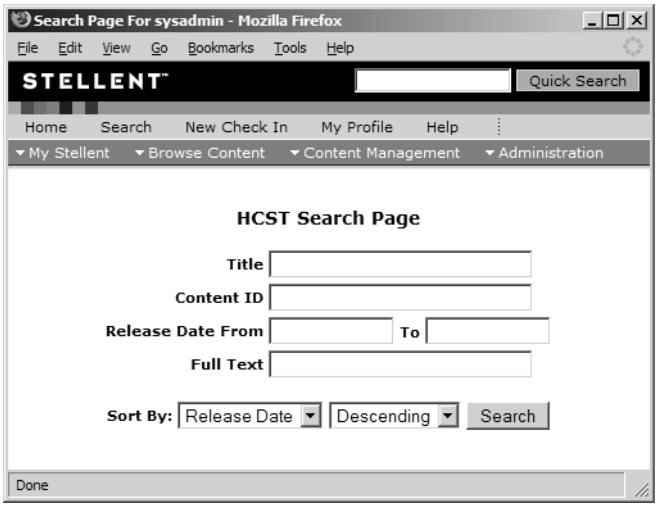

Figure 3-4. *Advanced HCST search form*

The first thing to notice about this page is that there are two HTML forms. The primary purpose in using two forms is to simplify the URL to the search results page. It is only necessary to pass in the value for `QueryText` in the request. Passing any other data only clutters the URL to the results page, making it less portable and harder to reuse. To this end, the page is split into two forms: one with the search terms and another to help create the search terms.

The first form, named `QUERYTEXTCOMPONENTS`, contains metadata fields such as title (`dDocTitle`) and release date (`dInDate`). This data is used in JavaScript functions to generate the `QueryText` parameter for the search. This form is simply a placeholder for these fields, so it does not have a Submit button.

The second form (named `SEARCHFORM`) is the actual form submitted to run the search, which contains optional search terms such as `SortOrder` and `SortField`. When the Search button is clicked, it executes the JavaScript function `submitFrm`. This function takes one parameter. If `true` is passed, the search terms will be saved after the search is submitted. This is not necessary at this point, so `false` is passed.

The `submitFrm` function is defined in the resource `query_form_submit_script` or `query_form_submit_form_function`, depending on the Content Server version. Both resources are defined in the `std_page.htm` resource file mentioned in Chapter 2.

The `submitFrm` function builds the value for `QueryText` by analyzing the fields in the `QUERYTEXTCOMPONENTS` form. If a field in that form contains a value, it is appended as a token in the `QueryText` string.

This process works automatically because of the special structure of the fields in the form. For example, to allow a user to search the title field (dDocTitle), place this code in the form:

```
<td align=right><span class=searchLabel>Title</span></td>
<td>
    <input type="hidden" name="opSelected"
        value="hasAsSubstring">
    <input type="text" size=30 name="dDocTitle"
        value="">
</td>
```

Notice that the preceding code contains two input fields. The hidden one is named opSelected; the other is the name of the metadata field to query.

The value for opSelected determines how this metadata field is queried. In the preceding case, it is set to hasAsSubstring, which is used in version 7.0 to do a substring query of a text field. In older versions, it is simply Substring. Other useful values for opSelected for strings include beginsWith, hasAsWord, equals, and endsWith. For systems older than version 7.0, they correspond to Starts, Contains, Matches, and Ends, respectively.

On this form the content ID field is nearly the same as the title field, which is possible because submitFrm automatically configured itself based on the names of the fields. The only two differences are the name of the field and the label.

Date fields, such as the release date (dInDate) and the expiration date (dOutDate), are more complex. Typically, a user is more interested in a range instead of equality or substring tests. Therefore, the submitFrm function uses different values for opSelected for date fields. The release date field is formatted in this way:

```
<tr>
    <td align=right><span class=searchLabel>
        Release Date From</span></td>
    <td align=left>
        <input type="hidden" name="opSelected" Value="dateGE">
        <input type="text" size=12 maxlength=20
            name="dInDate" value="">
        <span class=searchLabel>To</span>
        <input type="hidden" name="opSelected" Value="dateLess">
        <input type="text" size=12 maxlength=20
            name="dInDate" value="">
    </td>
</tr>
```

In this case, the user has two form fields. These fields are labeled From and To, but both have the field name dInDate. If a value for the From field is specified, the generated query finds all items with a release date after the specified date. If the To field is specified, the query finds all items released before the specified date. If they are both specified, the query returns all items in a specific date range.

Number fields are used in a similar fashion to date fields. Useful values for opSelected for numbers include numberEquals, numberGE, numberGreater, numberLE, and numberLess. For

systems older than version 7.0, they correspond to Equal, GreaterEqual, Before, LessEqual, and After, respectively.

The third thing to notice is the structure of the form SEARCHFORM. Unlike the previous example, the action parameter does not point to an HCST page. Instead, it points directly to the content server. It sets the parameter IdcService to GET_SEARCH_RESULTS and a specific value for urlTemplate, as shown following:

```
<$urlTemplate = strSubstring(fileUrl, 0, strLength(fileUrl)-5)
    & "-results.hcst"$>
<form name="SEARCHFORM" method="GET"  action="<$HttpCgiPath$>">
<input type=hidden name="IdcService" value="GET_SEARCH_RESULTS">
<input type=hidden name="urlTemplate" value="<$urlTemplate$>">
```

As mentioned in Chapter 2, when you execute a service request it formats the response with the standard template for the service. Parameters such as urlTemplate and docTemplateName can be used to redirect the service to a different template. This example redirects the response to the page hcst-advanced-search-results.hcst, which is shown later.

The URL to the result page is determined from the URL of the current page. As in the previous examples, the result page is checked-in with the same security group and content type so they appear in the same folder in the weblayout directory. This is important for determining the path to the result template.

To generate the path to the result template, you also need the fileUrl parameter. This parameter is set for all Dynamic Service Pages (DSPs) and contains the URL to the current file. The result template is in the same directory as the current page and it has a similar name. Therefore, it is a simple matter to determine the result URL: strip off the last five letters of the URL and append the letters -result.hcst. The first line in the previous code does precisely this.

When the Search button is clicked, the function submitFrm generates a value for QueryText. This value is then inserted into the form SEARCHFORM. Finally, the form SEARCHFORM is submitted. The Content Server runs the query and generates the response based on the template hcst-advanced-search-results.hcst.

Before testing, you need to create the result page, which is similar to the previous simple portal pages, with a handful of differences. First, because it is purely a result template, it does not run the executeService IdocScript function. Second, it can display multiple pages of results.

Now create the file hcst-advanced-search-results.hcst with the following code:

```
<html>
<head>
<$PageTitle="Guest Search Page"$>
<$if isTrue(IsLoggedIn)$>
    <$PageTitle="Search Page For " & UserName$>
<$endif$>
<$include searchapi_result_html_head_declarations$>
</head>

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

```
<h3 class=pageTitle>HCST Search Results Page</h3>

<p>Found <$TotalRows$> results.</p>

<$c="You can also use the include searchapi_navigation_list"$>
<p align="center">
<$if PreviousPage$>
[ <a href="<$strRemoveWs(inc("searchapi_navigation_previous"))$>">
        PREV</a> ]
<$endif$>
<$if NextPage$>
[ <a href="<$strRemoveWs(inc("searchapi_navigation_next_page"))$>">
        NEXT</a> ]
<$endif$>
</p>

<br><br>

<table border='1' cellpadding='0' cellspacing='3' width='540'>
<$loop SearchResults$>
    <tr>
    <td align='left'>
        <a href='<$URL$>'><$dDocName$></a>
    </td>
    <td>
        <b><$dDocTitle$></b><br>
        Release Date: <$dInDate$><br>
        Author: <$dDocAuthor$>
    </td>
    <td align='center' width='50'>
        <a href='<$HttpCgiPath$>?IdcService=DOC_INFO&dID=<$dID$>'>
        <img src='<$HttpWebRoot$>/images/stellent/InfoIcon_sm.gif'
            border='0'>
        </a>
    </td>
</tr>
<$endloop$>
</table>

<$include std_page_end$>

</body>
</html>
```

Check-in the file with the same security group and content type as the previous HCST and give it the content name `hcst-advanced-search-results`. Now if you execute a search, the results look like Figure 3-5.

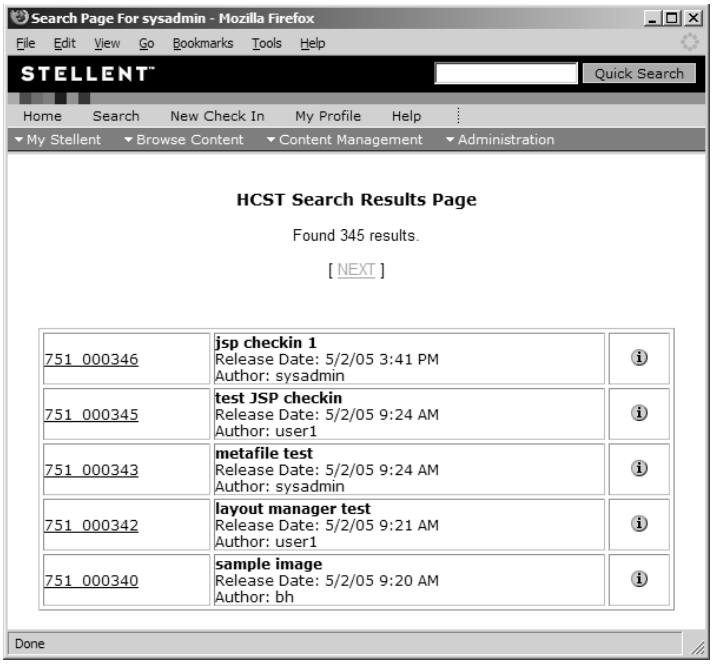

Figure 3-5. *Advanced HCST search results*

This page is slightly more complex than the previous search results page. It outputs the title of the item inside a link to the web-viewable rendition of it. It also has an additional link that takes the user to the Content Information page for this item. From that page, the user can check-out the item or subscribe to it.

In addition, if the flags `PreviousPage` or `NextPage` are present, some simple navigation also appears. The parameter `ResultCount` is hard-coded to `5` on the search page, which means that only five results will appear at a time. If more than five are present, the `NEXT` link is visible.

Clicking the `NEXT` link takes the user to the next five results, which display the `PREV` link. That link takes the user to the previous five results. If more than 10 results exist, another `NEXT` link is present.

These links to other pages are generated with core resource includes. The include `searchapi_navigation_next_page` creates the URL to the next page, whereas the include `searchapi_navigation_previous` creates the URL for the previous page. These resources are included with the `inc` function and then post-processed with the `strRemoveWs` function. Together they extract all the white space from the includes, which ensures a URL on a single line. See Appendixes A and B for more information on these functions.

The value for `urlTemplate` is automatically placed in these URLs to make sure that the pages are all generated in the same manner.

In conclusion, the search functionality is easy to implement on an HCST. A search can be run from IdocScript or by using the `urlTemplate` parameter. Most service requests can be altered in a similar manner (one important exception is the check-in service).

Custom check-in pages are more complex because they require special HTML to support a file upload and a larger list of metadata fields. The next example covers this very subject.

Custom Check-In Pages

The contribution and update pages display metadata based on complex rules. These rules are based on internal metadata definitions for proper HTML display. Simple core includes were covered earlier, such as `std_page_begin` and `std_page_end`. To display a check-in page, you need `std_checkin_html_head_declarations` and `std_meta_field_display` as well. The first loads up the metadata definitions, whereas the second renders them as HTML. The `std_meta_field_display` include also contains the logic needed to display option lists and render content profiles.

To create a custom check-in page, create the file `hcst-checkin.hcst` with the following code:

```
<html>
<head>
    <$PageTitle="Check In", isNew=1$>
    <$include std_checkin_html_head_declarations$>
</head>

<$include body_def$>
<$include std_page_begin$>

<h3>HCST Check In</h3>

<p>This check-in page gives the HCST developer total control over which
metadata fields to display, and how to format the HTML.</p>

<form name="Checkin" method="post" enctype="multipart/form-data"
    action="<$HttpCgiPath$>">

<!-- prefilled default values... you can place other default
     values here for metadata such as 'dDocType' and 'dSecurityGroup'
     to make specialized contribution paes -->
<input type=hidden name="IdcService" value="CHECKIN_NEW">
<input type=hidden name="dDocAuthor" value="<$UserName$>">
<input type=hidden name="dRevLabel" value="1">

<$c="This page is hcst-checkin.hcst, redirect to the page
    hcst-checkin-results.hcst after the checkin"$>
<$redirectUrl = strSubstring(fileUrl, 0, strLength(fileUrl)-5) &
    "-confirm.hcst?dDocName=<$dDocName$>&dID=<$dID$>"$>
<input type=hidden name="RedirectUrl" value="<$redirectUrl$>">

<table>

<!-- This field is optional if auto numbering is turned on for the
     Stellent Content Server.  -->
<tr>
    <td>Content ID</td>
```

```
    <td><input type="text" name="dDocName" size=35 maxlength=30
        value="">
    </td>
</tr>
<tr>
    <td>Type</td>
    <td>
        <select name="dDocType">
        <$loop DocTypes$>
            <option value="<$dDocType$>"><$dDescription$>
        <$endloop$>
        </select>
    </td>
</tr>
<tr>
    <td>Title</td>
    <td><input type="text" name="dDocTitle" size=35    maxlength=30>
    </td>
</tr>
<tr>
    <td>Security Group</td>
    <td>
        <$rsMakeFromList("SecurityGroupSet", "securityGroups")$>
        <select name="dSecurityGroup">
        <$loop SecurityGroupSet$>
            <option><$row$>
        <$endloop$>
        </select>
    </td>
</tr>
<tr>
    <td><hr></td><td><hr></td>
</tr>
<tr>
    <td>File</td>
    <td><input type="file" name="primaryFile" maxlength=250>
    </td>
</tr>

<!-- custom metadata section -->
<$loop DocMetaDefinition$>
    <$showSimple=""$>
    <$if showSimple$>
        <tr>
            <td><$lc(dCaption)$></td>
            <td><input name="<$dName$>"></td>
        </tr>
    <$else$>
```

```
        <$strTrimWs(inc("std_meta_field_display"))$>
    <$endif$>
<$endloop$>

<tr>
    <td><hr></td><td><hr></td>
</tr>
<tr>
    <td align=center colspan=2>
    <input type=button name=javaSubmit value=" Check In "
        onClick="postCheckInStandard(this.form)">
    <input type=reset name=reset value=" Reset ">
    </td>
</tr>
</table>
</form>

<$include std_page_end$>
</body>
</html>
```

Check this file into the Content Server with a Content ID of `hcst-checkin`. It looks like Figure 3-6.

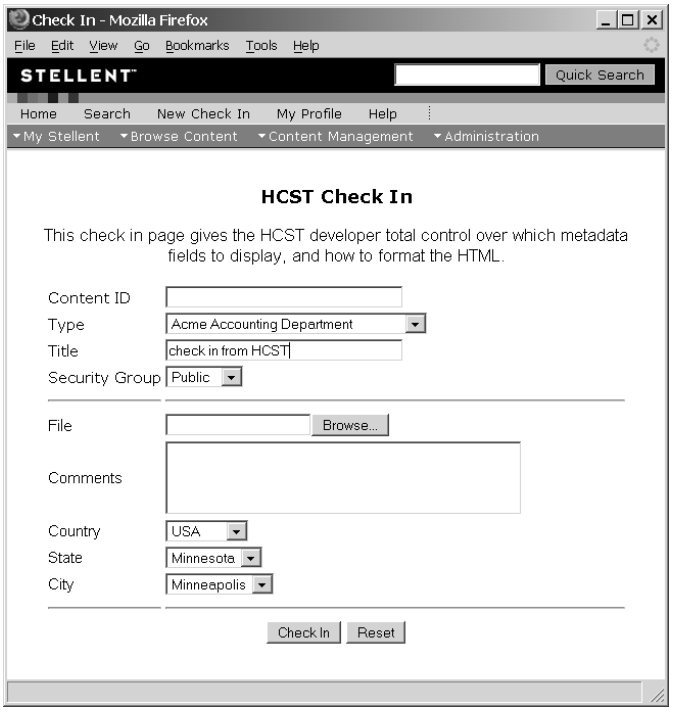

Figure 3-6. *HCST check-in form*

The core includes do most of the work to render this page. This page has a form with the value of IdcService set to CHECKIN_NEW. This service requires several mandatory parameters, which are also input fields: dDocTitle, dDocType, dSecurityGroup, dDocAuthor, dRevisionLabel, and primaryFile.

Beneath the standard fields are the custom metadata fields. These fields are displayed by looping over the ResultSet named DocMetaDefinition, which contains information about the custom metadata fields. There are two options on the page to display the field. One is simple, which displays only the field name with a text input field. The other way uses the resource std_meta_field_display, which renders the field exactly as it would appear on the default check-in page. The former is useful when you need total control over the HTML; the latter is useful when the field contains option lists.

Notice that the HTML form on this page has an encoding type of multipart/form-data, which is required to tell the browser to upload a file. Without it, it is impossible to upload a file from a client's browser.

Another important thing to note is the value for RedirectUrl in the form. This parameter is used by many services to redirect a POST request to a GET request after the POST request is successful. This URL typically contains IdocScript to generate a URL based on the response data from the POST service. After a check-in, we want to redirect to the page hcst-checkin-confirm.hcst, but the RedirectUrl needs the full URL, which includes the content type and group. We know that the fileUrl for the current page contains the security group and content type, and ends with hcst-checkin.hcst. So, instead of hard-coding the group and type information, we can parse out the last five characters of fileUrl, and append -confirm.hcst to make a full URL to the confirmation page:

```
<$redirectUrl = strSubstring(fileUrl, 0, strLength(fileUrl)-5) &
    "-confirm.hcst?dDocName=<$dDocName$>&dID=<$dID$>"$>
<input type=hidden name="RedirectUrl" value="<$redirectUrl$>">
```

Notice that the parameters dID and dDocName are appended to this URL, which is needed because unlike when urlTemplate is used, the response data is not forwarded to the redirected page. The IdocScript inside RedirectUrl is evaluated with the response data, but other than that no response data is available. So if you want the results of the service to be present on the confirmation page, each piece of data must be explicitly mentioned in the RedirectUrl parameter and encoded as IdocScript.

In addition, this page uses the IdocScript functions lc, strTrimWs, and rsMakeFromList. The lc function localizes a string into the user's language. The strTrimWs function is similar to strRemoveWs, except it only removes white space at the beginning or end of a string. The function rsMakeFromList turns an OptionList object into a ResultSet object so it can be more easily rendered on the page. See Appendix B for more information about these functions.

Finally, this example needs a custom confirmation page. This page does not need to be complicated; it need only display some information about the item and mention that it was successfully processed. To do this, create the file hcst-checkin-confirm.hcst with the following contents:

```
<html>
<head>
    <$PageTitle = "Check In Confirmation"$>
    <$include std_html_head_declarations$>
</head>
```

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

<h3>Check In Confirmation</h3>

<p>Successfully checked-in content item:
<a href="<$HttpCgiPath$>?IdcService=DOC_INFO&dID=<$dID$>">
    <$dDocName$></a></p>

<$include std_page_end$>

</body>
</html>
```

This page should be checked-in with the same security group and content type as the `hcst-checkin.hcst` page.

After a successful check-in with the custom page, the result page looks like Figure 3-7.

Figure 3-7. *HCST check-in confirmation page*

The check-in page for this example looks very similar to the default check-in page. The HTML has a two-column presentation style, with the label in one column and the field in another. But what if you want the display to be different? What if you want a four-column display or a check-in form that isn't in a table at all? The `std_meta_field_display` include uses flags to display a field differently, but it cannot do everything you might want. These flags are discussed in Appendix D.

One option is to set the `showSimple` flag in the HCST, which gives the developer complete control over how the metadata fields are displayed. This option has its limitations because it's more difficult to maintain. Option lists need to be hard-coded in HTML, and neither dependent choice lists nor content profiles will work.

The other option is to modify the includes that are referenced by `std_meta_field_display`. In this way, a developer can override the include that forces a user into the two-column display and make it display different HTML. This is the preferred approach, but it requires IDOC files or a component. This method is discussed further in Chapter 6 and Chapter 9.

Summary

For version 7.0 and older systems, HCSTs are commonly used to customize search, content info, and check-in pages. However, in version 7.5, the Content Profiles functionality eliminates the need for many of these custom pages because it is possible to customize the look and feel of metadata pages with the Configuration Manager applet. However, HCSTs are still used for custom portal pages, custom workflow pages, or other pages that are too complex to be created with profiles. But most developers generally move away from HCSTs in favor of custom components, content profiles, or HCSPs.