



---

# **Coveo Enterprise Search 5**

Search Interface API

## Contents

<b>Basic Concepts .....</b>	<b>4</b>
About SearchControl and QueryControl .....	5
About SearchInterfaces and Skins.....	5
About Search Settings .....	5
Search Setting Persistence.....	6
About Ajax and SearchUpdatePanel .....	6
How the Search Interface Determines which Panels to Update.....	6
Making Updates More Granular.....	7
Writing New Controls Managed by SearchUpdatePanels .....	7
Forcing the Update of a Control in Code .....	7
<b>Customizing the Search Interface .....</b>	<b>8</b>
Creating a New Skin .....	8
Modifying a Skin.....	8
Developing New Controls in Visual Studio .....	8
<b>Internals .....</b>	<b>9</b>
What Is the Structure of a Skin .....	9
How to Load the Search Interface .....	9
What Is the Lifetime of a Search Page Request.....	10
OnInit.....	10
LoadViewState (when performing a postback).....	10
LoadPostBackData (when performing a postback) .....	10
PreLoad.....	10
OnLoad .....	10
RaisePostDataChangedEvent (when performing a postback) .....	10
OnPreRender .....	11
SaveViewState.....	11
Render .....	11
How to Perform Searches.....	11
When Are Searches Executed.....	11
Events Fired When Performing a Search .....	12
Populating the SearchBuilder Object.....	12
Using the SearchBuilder to Get a QueryWrapper Object.....	12
Creating the ASP.NET Controls to Render Results .....	12
<b>Samples.....</b>	<b>13</b>

---

How to Download and Install the Search Interface Samples.....	13
Downloading and Installing the Samples.....	13
Installing the Samples.....	13
Sample: Calendar Date Search Control.....	13
Installing and Viewing the Sample.....	13
Creating the Calendar Search Control.....	13
Creating the CalendarDateSearchSetting Class .....	17
Displaying a Comment to the User When Search Results are Being Restricted .....	20
Sample: Creating a Skin from Scratch.....	21
About this Skin .....	21
<b>Upgrade Notes.....</b>	<b>24</b>

Coveo Search Interface is designed to enable extensive customization; therefore responding to the ever-growing needs of the customers. To customize the Search Interface, it is possible to either use the Interface Editor or create a new skin (for more information, refer to [About SearchInterfaces and Skins](#)) and modify it according to your specific needs.

The Interface Editor (**Search** or **Initial** page > **Edit this Interface**) is certainly the easiest way to customize the Search Interface; however, because it is a GUI tool, there are certain limitations. This approach is strongly recommended to enable or disable certain features, manage facets, change colors, etc.

If, on the other hand, more complex modifications, such as adding or removing controls, modifying the global layout or the behavior of the Search Interface using custom code are required, it is recommended to create a custom skin (for more information, refer to [About SearchInterfaces and Skins](#)) and author custom search controls in order to implement features, which are not available in Coveo *Enterprise Search* (CES) out-of-the-box.

This section of the CES documentation describes how to customize the Search Interface using the second approach. For more information concerning the Interface Editor, refer to the [Interface Editor Online Help](#).

**Important:** Even if it is possible to customize the Search Interface using the two approaches simultaneously, it is recommended to use only one of them at a time, as the two may conflict in some circumstances. Therefore, to perform advanced customizations, use the second approach from the start. Everything that can be done through the Interface Editor can also be done directly from within a custom skin.

## Basic Concepts

Coveo Search Interface is made of several modular *ASP.NET* controls that interact together in order to provide the Search UI. These controls are referenced in a set of user control files called skins (for more information, refer to [About SearchInterfaces and Skins](#)). Using skins makes it easy to add, extend or remove functionalities from the interface, as this can usually be done by adding or removing a control from the skin.

The default Coveo Search Interface is made of more than a hundred separate controls that implement the different functionalities available to the user. The framework is fully extensible, meaning that it is possible to author your own controls, that integrate in the same manner in a custom Search Interface.

The following lists examples of Search Interface controls:

Control	Description
Query	Allows the user to enter the search keywords.
SearchButton	Renders the buttons used to perform new searches.
ResultTitle, ResultExcerpt and ResultPrintableUri	Output the title, excerpt and URI for each result (those are typically put in a repeated template).
Pager	Allows users to browse the various result pages a query returns.
RefineByType	Allows users to refine the query to a specific result type.

All the interactions between the various controls making up the Search Interface are controlled by the SearchControl control. For more information, refer to [About SearchControl and QueryControl](#).

The complete Coveo Search Interface can be added to any *ASP.NET* page by using a single special control called SearchInterface. This control loads the appropriate skin and settings into the page. For more information, refer to [How to Load the Search Interface](#).

### About SearchControl and QueryControl

The Search Control control is responsible of synchronizing the interactions between all the controls of the Search Interface. It also keeps track of the current query and fires the various events that cause other controls to perform various tasks. There is always only one SearchControl per Search Interface; however, it is possible to have more than one Search Interface per page (this is useful when using search Web parts).

A single Search Interface can display results from several queries; therefore making it possible to search several kinds of items from the same UI (for example, combining a *People Search* feature with a more general document search). Every query is performed by the QueryControl control. When the user performs a search, each QueryControl uses the global query (provided by SearchControl), adds additional information (for example, restricting to people results) and sends the query to the server in order to retrieve the results, which are then displayed.

### About SearchInterfaces and Skins

*Skin* designates a set of user control files that define the layout of the Search Interface. By default, CES is shipped with the **Default** and **Email** skins. It can also be shipped with the **SharePoint** skin if the *Microsoft SharePoint* integration was installed.

Skins must not be confused with *Search Interfaces*. A Search Interface acts as a container for settings such as styles and configuration settings—one of these settings is the name of the skin used with the Search Interface. There can be several Search Interfaces bound to a single skin, much in the same way that many search pages can use the same Search Interface. In other words:

- A **Search** page makes use of a specific Search Interface (represented by the InterfaceSettings class) that is loaded into the page using the SearchInterface control.
- A Search Interface provides settings such as styles, custom refine fields, preference defaults, etc. and is associated with the InterfaceSettings class.
- A skin provides the layout of the rendered search page and is associated at page initialization with a Search Interface depending on the value of the Skin property.

### About Search Settings

The SearchControl object keeps track of the current query's content using a collection of objects called **Search Settings**. Each of these objects is responsible for storing a particular aspect of the query, for example the keywords entered by the user or the type of document to restrict to. All these objects derive from the SearchSetting class.

Search settings usually have a strong relationship to a control that allows the user to control a specific aspect of the query. For example:

- The Query control (the textbox in which users enter keywords) is associated with the QuerySetting object.
- The Format control (the drop-down used to restrict to a specific format) is associated with the FormatSetting object.

When the user changes the content of these controls (called **parameter controls**), the associated search setting is updated to reflect the new value. Then, any instance of the same control located elsewhere on the page is notified in order to update the content.

### Search Setting Persistence

The SearchControl object is responsible of persisting the search settings across requests. Two sets of settings are persisted:

- The **Staging** settings hold the search settings currently in the various parameter controls.
- The **Effective** settings hold the search settings that make up the current query.

Having two distinct sets of settings allows the user to edit queries without it having an immediate effect on the results displayed. The changes made by the user are only effective when the **Search** button is clicked. At this point, the current effective settings are discarded and replaced by a copy of the staging settings. Most parameter controls thus only need to care about the staging settings (e.g. they never access the effective settings). However, this is not valid for all controls. For example, all the *Refine By* controls directly alter the effective settings.

Both sets of search settings are persisted by the SearchControl object in the page's view state. When a postback occurs, the settings are read from the saved state and an event is triggered to inform all parameter controls that they should initialize themselves using the content of the setting collections. The framework then proceeds through the normal page lifecycle and at the end of it, the settings are saved back to the view state.

### About Ajax and SearchUpdatePanel

CES uses advanced *Ajax* techniques to provide a more streamlined user experience. Most of the time, when customizing the Search Interface, the user does not need take care of it; however, there are some cases where additional knowledge is required.

Coveo's Search Interface uses a model that is very close to *Microsoft Ajax's* UpdatePanels in order to update parts of the **Search** page when a user performs actions without fully reloading the page (referred to as postback). Instead, it uses *JavaScript* code to perform a partial postback on the server, that executes the required actions and renders new HTML for the parts of the page to update.

The zones that can be updated are defined by the SearchUpdatePanel control. Whenever a control inside SearchUpdatePanel must be updated, the HTML for the whole content of the nearest SearchUpdatePanel is rendered, sent to the browser and then updated. A SearchUpdatePanel can contain other SearchUpdatePanel controls. When the *ASP.NET* request reaches the **Render** phase, the Search Interface uses different factors to build a list of panels in order to update and only the HTML for these specific panels is sent to the browser.

### How the Search Interface Determines which Panels to Update

During request processing, SearchControl keeps track of several events that can cause certain controls to require an update. The following lists these events:

- A new query has been performed
- The staging settings have been modified
- The effective settings have been modified
- The Mode of the Search Interface has been modified
- The current result page has been modified
- The currently selected result has been modified
- A saved query or saved filter has been modified

When request processing reaches the **Render** phase, each `SearchUpdatePanel` recursively scans its children, looking for controls implementing special marker interfaces associated to the events that the `SearchControl` keeps track of. The available marker interfaces are:

- `IUpdateOnNewSearch`
- `IUpdateOnStaging`
- `IUpdateOnEffectiveChange`
- `IUpdateOnModeChange`
- `IUpdateOnPageChange`
- `IUpdateOnSelectedChange`
- `IUpdateOnQueryOrFilterChange`

When a control implementing one of the marker interfaces is found and the associated event has been raised during processing, the `SearchUpdatePanel` automatically schedules itself for updating. The parent `SearchUpdatePanel` does not access the child `SearchUpdatePanel`, as it assumes that it updates itself automatically.

### **Making Updates More Granular**

In order for updates to be more granular, it is possible to add `SearchUpdatePanel` controls to the custom skins in order to wrap around the zones to update. No further logic is required, the Search Interface automatically uses these new panels when possible.

### **Writing New Controls Managed by SearchUpdatePanels**

The updates of the custom search control developed are managed by `SearchUpdatePanel`. In some cases, e.g. for controls used to render information about a result (typically in result templates), the user does not manage this, as it is automatically updated when the result changes. However, if a user develops his own search controls, he must determine which event causes its rendered HTML to be updated, and implement the associated marker interfaces. Then, the control is automatically updated whenever required.

### **Forcing the Update of a Control in Code**

It is possible to force a specific control to be updated (through the nearest `SearchUpdatePanel`) by calling `UpdatePanel.UpdateContainingPanel` and passing a reference to the control to update.

## Customizing the Search Interface

### Creating a New Skin

A [skin](#) is a set of user control files (.ascx) that define the appearance and behavior of a Search Interface. All the user control files for a given skin are located in the `Web\Coveo\Skins\SkinName` folder located in the CES installation.

By default, CES comes with several skins; however, it is possible to create one in order to perform more advanced customizations. The easiest way to create a new skin is to make a copy of an existing one and rename the resulting folder. The skin files can then be modified to achieve the desired result.

If the search interface being built differs radically from the ones provided by Coveo, it is also possible to create a new skin from scratch. This approach should only be considered if changing an existing skin requires more work than creating a new one. If a skin is created from scratch, the user must include all the controls required in order to implement the appropriate functionality. For more information, refer to [Sample: Creating a Skin from Scratch](#).

To use the new skin, access the Search page (*Windows Start* menu > All Programs > Coveo Enterprise Search > Search Interface) and click **Edit this Interface**. In the **Advanced** section of the **Features** tab, select the skin to use for the interface being edited and click **Apply**.

### Modifying a Skin

A skin is a set of user control files (.ascx) that define the appearance and behavior of a Search Interface. These files can be modified freely to achieve the appropriate result.

To modify a skin file, open it in a text editor, modify it, save the modifications and reload the Search page in the Web browser. The modifications are immediately taken into consideration.

For more information concerning the files that compose a skin, refer to [What Is the Structure of a Skin](#).

### Developing New Controls in Visual Studio

CES comes pre-installed with a *Visual Studio* solution and project that can be used to author custom search controls. The solution file is named `CustomControls.sln` and is located in the `Web` folder of the CES installation.

The solution contains a single project (`CustomControls.csproj`) that properly references to Coveo's assemblies. When this project is built, it outputs a DLL named `CustomControls.dll` in the `Web/Bin` folder of the CES installation. It is possible to reference the controls and other entities from this DLL in the user controls making up a skin (for more information, refer to [About SearchInterfaces and Skins](#)).

## Internals

### What Is the Structure of a Skin

A skin is a set of user control files (.ascx) that define the appearance and behavior of a Search Interface. The following lists the most important files in a skin:

- **CoveoSearch.ascx:** This is the root file of the skin. It defines the overall HTML layout and includes the other skin files in the appropriate places. This file is a good place to add custom code that handles events triggered by the SearchControl object (for more information, refer to [About SearchControl and QueryControl](#)).
- **InitialPanel.ascx:** This file defines the content of the panel displayed when the user loads the Search page without passing a query through the query string.
- **SearchPanel.ascx:** This file defines the content of the panel above the header bar when a query is performed. It usually contains the Query control and Search button.
- **ResultsPanel.ascx:** This file defines the content of the panel containing the header bar and the results of the current query.
- **ResultTemplate.ascx:** This file defines the content that is repeated for each query result. Modify this file in order to alter the result template.
- **Toolbar.ascx:** This file defines the content of the panel on the right of the results that usually contains search facets and other misc controls.
- **AdvancedSearchPanel.ascx:** This file defines the content of the full-page **Advanced Search** page. It includes several other user control files for each section of the advanced search. The same sections are also displayed in the tabbed inline advanced search. The related files are:
  - AdvancedCollections.ascx
  - AdvancedKeywords.ascx
  - AdvancedDate.ascx
  - AdvancedProperties.ascx
- **PreferencesPanel.ascx:** This file defines the content of the **Preferences** page.

Obviously, there are several other files inside a typical skin; feel free to explore to perform modifications that cannot be done using the files listed above.

### How to Load the Search Interface

The Coveo Search Interface is usually added to an *ASP.NET* page through the SearchInterface control. It is responsible of loading the rest of the controls that make up the Search Interface:

1. In the **OnInit** stage, the SearchInterface control first determines which Search Interface to load on the page. The easiest way to configure this is to set the Name property to the name of the appropriate Search Interface. Otherwise, the SearchInterface control looks for a name specified through the query string (using the sk=somename parameter) and attempts to use **Per Uri Settings** to define a name. If none is found, the **Default** Search Interface is used.

2. The SearchInterface then loads the InterfaceSettings object for the selected Search Interface. This object holds all the Search Interface settings, such as styles, custom refine fields, etc. and can be edited using the Interface Editor.
3. Using the Skin property of the InterfaceSettings object, the SearchInterface then loads the proper version of *CoveoSearch.ascx* in the page. This user control is the root control that includes all other components required by the Search Interface and is located in the sub folders of the *Coveo/Skins* folder.

### What Is the Lifetime of a Search Page Request

The processing of a request on a **Search** page involves several steps:

#### OnInit

- The SearchInterface control establishes the connection to the CES server and loads the InterfaceSettings as well as the associated skin in the page. If the connection attempt fails, no search control is loaded and an error page is displayed.
- The SearchControl object initializes itself, loading preferences, saved queries and filters, etc.

#### LoadViewState (when performing a postback)

The SearchControl object loads the persisted SearchState object containing, among other things, the staging and the effective settings. It then triggers the LoadSettings event so that all parameter controls can initialize themselves with the value they had on the last request (normally controls load their previous value directly from the viewstate, but parameter controls are using the search settings instead).

#### LoadPostBackData (when performing a postback)

The parameter controls are updated with the values posted by the browser (this is done automatically by the underlying *ASP.NET* framework). If the posted value is different from the one loaded using the search settings, the framework raises a flag to remember to trigger some kind of **OnChange** event on the control after the **OnLoad** phase.

#### PreLoad

If a query was executed in the previous postback, it is performed again in order to re-create the result controls from the previous query (this enables them to use the viewstate and handle postback events).

#### OnLoad

If this is the first load of the page:

- The query string is selected and any search parameter that is present is loaded.
- The InitializeSettings event is raised to give the opportunity to initialize the search settings based on any custom rule.

#### RaisePostDataChangedEvent (when performing a postback)

In this stage, the framework triggers the **OnChange** event (or the equivalent) on all controls flagged as *Modified* since the last postback, before the **OnLoad** event. When a parameter control receives an **OnChange** event, it updates the staging search settings accordingly.

## OnPreRender

- The SearchControl object triggers the LoadSettings event again in order for all the parameter controls to update themselves using the settings that may have been modified by an **OnChange** event.
- The SearchControl object then triggers the BeforeQueries, PerformQueries and AfterQueries events, which causes the QueryControl objects on the page to execute their respective queries. The following details what occurs for each QueryControl:
  - The QueryControl calls the CreateSearchBuilder method to retrieve a SearchBuilder object configured with all the settings managed by the SearchControl (e.g. global to all queries on the page). This includes the query expressions for the effective search settings.
  - The SearchBuilder is then further updated with the various settings proper to each QueryControl. This includes settings such as whether duplicate documents are to be filtered, the current result page, etc.
  - The QueryControl then fires the SetupSearchBuilder event to give external code an opportunity to tweak the effective search that will be performed.
  - The final SearchBuilder is then passed to the QueryWrapperFactory to retrieve a QueryWrapper that encapsulates the query.
  - The CreateResultControls event is raised. When this occurs, the ResultList control instantiates the result template for each result that is to be displayed.

## SaveViewState

The SearchControl and QueryControl objects both persist their state in the viewstate.

## Render

When this stage of page processing is reached, all controls are rendered to produce the HTML for the page, which includes the result controls instantiated by ResultList after the query is executed.

## How to Perform Searches

Whenever the Search Interface needs to execute a query on the CES server, the SearchControl and QueryControl objects run several steps in order to build the query, send it to the server and transform the results in *ASP.NET* controls, which are inserted in the page and rendered to the browser.

## When Are Searches Executed

The Coveo Search Interface can execute a query at two moments during the lifetime of the *ASP.NET* request:

- **PreLoad:** When a query is executed at the previous postback, it is automatically re-executed before the **Load** stage in order to allow result controls to load previous values from the viewstate and allow them to process postback events. This makes it possible to use controls such as buttons in the result templates along with event handlers that perform processing when the user clicks the button. Because this is done only when the query is previously executed, the results are most often retrieved directly from the cache; therefore the cost is negligible.
- **PreRender:** When a new search is performed, following an action from the user such as clicking the **Search** button, a query is executed during the **PreRender** stage.

## Events Fired When Performing a Search

Event	Description
BeforeQueries	The BeforeQueries event is fired by the SearchControl object; therefore other components can execute certain tasks before the other query events are fired.
PerformQueries	The PerformQueries event is fired by the SearchControl object. It tells all its children QueryControl to execute their query on the server (for more information, refer to <a href="#">About SearchControl and QueryControl</a> ). When it receives this event, the QueryControl executes all the steps required to run the query.

## Populating the SearchBuilder Object

The SearchBuilder object accumulates all the information concerning the query to execute before sending it to the server. When the query is being prepared, various components of the Search Interface add their own bits of information (such as query parts, sorting settings, etc.) to the SearchBuilder until all the information has been gathered.

To prepare the SearchBuilder object, the QueryControl first calls CreateSearchBuilder on the SearchControl—which is responsible for creating a SearchBuilder—and populates it with all the information global to all the QueryControl objects. This includes all the search settings, along with various other settings.

The CreateSearchBuilder also fires the SetupSearchBuilder event on the SearchControl in order to allow other components to tweak the SearchBuilder before it is handed back to the QueryControl.

When the QueryControl receives the SearchBuilder, it adds more information related only to the current query and fires a separate event that is called SetupSearchBuilder on the QueryControl; therefore other components can tweak the SearchBuilder for only this particular query. Afterwards, the SearchBuilder is completely populated.

## Using the SearchBuilder to Get a QueryWrapper Object

The QueryWrapper object provides a unified view on the results of a query that can be cached across several postbacks in order to avoid fetching results from the server that have already been retrieved. A QueryWrapper is obtained by passing the SearchBuilder object to the QueryWrapperFactory, which looks in a cache for an existing QueryWrapper built from an identical SearchBuilder. If none are found, it creates a new one.

Creating a QueryWrapper sends a query to the server in order to retrieve the first page of results (additional pages can be fetched using the same QueryWrapper). This is done using the COM APIs provided by CES.

## Creating the ASP.NET Controls to Render Results

After retrieving the QueryWrapper, each QueryControl fires the CreateResultControls event in order to instruct the ResultList controls to create the proper templates for each returned result. The resulting *ASP.NET* controls are inserted in the page's control tree in order to process events and render them following the request.

## Samples

### How to Download and Install the Search Interface Samples

#### Downloading and Installing the Samples

The Search Interface samples are packaged in a *Zip* archive that must be extracted in the CES installation folder.

#### Installing the Samples

The content of the *Zip* archive must be extracted from the Web folder located directly in the CES installation path. After extracting, the following folder is displayed `COVEO_INSTALLATION_FOLDER\Web\Samples`, containing all the samples (one by subfolder).

#### Sample: Calendar Date Search Control

This topic describes the steps to perform to create the **Calendar Search Control** that comes as a sample with CES.

<input checked="" type="checkbox"/> Specify Minimum Date							<input checked="" type="checkbox"/> Specify Maximum Date						
≤ November 2007 ≥							≤ November 2007 ≥						
Sun	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Mon	Tue	Wed	Thu	Fri	Sat
<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>31</u>	<u>1</u>	<u>2</u>	<u>3</u>
<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>9</u>	<u>10</u>
<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>	<u>17</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>15</u>	<u>16</u>	<u>17</u>
<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>	<u>18</u>	<u>19</u>	<u>20</u>	<u>21</u>	<u>22</u>	<u>23</u>	<u>24</u>
<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>1</u>	<u>25</u>	<u>26</u>	<u>27</u>	<u>28</u>	<u>29</u>	<u>30</u>	<u>1</u>
<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>8</u>

#### Installing and Viewing the Sample

For information on how to obtain and install the samples, refer to [How to Download and Install the Search Interface Samples](#). When the samples are installed, a search page containing this sample control should be accessible at `Samples/CalendarDateParams/CalendarDateParams.aspx`, starting from the URI where CES is installed.

#### Creating the Calendar Search Control

The **Calendar Search Control** is built using *Microsoft Visual Studio* and *C#* languages. For more information concerning this topic, refer to [Developing New Controls in Visual Studio](#).

The **Calendar Search Control** is made up of two files, one for the control itself and one for the associated `SearchSetting`(for more information concerning search settings, refer to [About Search Settings](#)).

#### To Create the Class for the Control

Create a new *C#* class named **CalendarDateSearch**, which inherits from `BoundToSearch`. This base class makes it easier to interact with `SearchControl`. The class should also implement the

IUpdateOnStagingChange marker interface, so that its rendered HTML is properly updated when required (for more information, refer to [About Ajax and SearchUpdatePanel](#)).

### Class Declaration

```
public class CalendarDateSearch : BoundToSearch,
                                IUpdateOnStagingChange,
                                INamingContainer
```

### To Create Child Controls

The control uses two checkboxes, as well as two **Calendar** controls to accept its input. Those are declared as private members and added as children of the **CalendarDateSearch** in the **CreateChildControls** method (inherited from Control). There is one calendar for the minimum modified date and another for the maximum date. Each calendar is associated to a checkbox that enables or disables it.

### Control Declarations

```
// The child ASP.NET controls that we use
private CheckBox CHKUseMinimumDate;
private Calendar CALMinimumDate;
private CheckBox CHKUseMaximumDate;
private Calendar CALMaximumDate;
```

### The CreateChildControls Method

```
protected override void CreateChildControls()
{
    CHKUseMinimumDate = new CheckBox();
    CHKUseMinimumDate.ID = "CHKUseMinimumDate";
    CHKUseMinimumDate.Text = "Specify Minimum Date";
    CHKUseMinimumDate.AutoPostBack = true;
    CHKUseMinimumDate.CheckedChanged += this.CHKUseMinimumDate_CheckedChanged;
    Controls.Add(CHKUseMinimumDate);

    CALMinimumDate = new Calendar();
    CALMinimumDate.ID = "CALMinimumDate";
    CALMinimumDate.SelectedDate = DateTime.Today;
    CALMinimumDate.SelectionChanged += this.CALMinimumDate_SelectionChanged;
    Controls.Add(CALMinimumDate);

    CHKUseMaximumDate = new CheckBox();
    CHKUseMaximumDate.ID = "CHKUseMaximumDate";
    CHKUseMaximumDate.Text = "Specify Maximum Date";
    CHKUseMaximumDate.AutoPostBack = true;
    CHKUseMaximumDate.CheckedChanged += this.CHKUseMaximumDate_CheckedChanged;
    Controls.Add(CHKUseMaximumDate);

    CALMaximumDate = new Calendar();
    CALMaximumDate.ID = "CALMaximumDate";
    CALMaximumDate.SelectedDate = DateTime.Today;
    CALMaximumDate.SelectionChanged += this.CALMaximumDate_SelectionChanged;
    Controls.Add(CALMaximumDate);
}
```

The **CreateChildControls** method is automatically called by the *ASP.NET* framework when a control should create and initialize its child controls. In this case, two checkboxes and the two calendars that the controls use are setup. On each control the **AutoPostBack** property is set to *True* so that *ASP.NET* automatically performs a postback each time the user changes something.

## The Render Method

The control overrides the **Render** method to render additional markup between each child controls. Here is the code for this method:

```
protected override void Render(HtmlTextWriter p_Writer)
{
    RenderBeginTag(p_Writer);
    p_Writer.RenderBeginTag("tr");
    p_Writer.RenderBeginTag("td");
    CHKUseMinimumDate.RenderControl(p_Writer);
    p_Writer.Write("<br/>");
    CALMinimumDate.RenderControl(p_Writer);
    p_Writer.RenderEndTag();
    p_Writer.RenderBeginTag("td");
    CHKUseMaximumDate.RenderControl(p_Writer);
    p_Writer.Write("<br/>");
    CALMaximumDate.RenderControl(p_Writer);
    p_Writer.RenderEndTag();
    p_Writer.RenderEndTag();
    RenderEndTag(p_Writer);
}
```

### To Load the *CalendarDateSearchSetting*

Like most search controls, the **CalendarDateSearch** controls uses a class derived from **SearchSetting** to store its current value (for more information, refer to [About Search Settings](#)). Whenever the control should load its value from its associated **CalendarDateSearchSetting** object, the **SearchControl** fires the **LoadSettings** event. To handle it, the control must hook up to some events fired by the **SearchControl** to properly manage the setting. This is done in the **OnInit** stage using the following code:

#### Hooking up to the **LoadSettings** event

```
SearchObject.LoadSettings += this.Search_LoadSettings;
```

The event handler for **LoadSettings** should try to grab the setting from the Staging settings collection. If a setting is found, the control must update its value to represent the state stored in the setting. If the setting is not present, it must reset its state to the default one.

#### LoadSettings Event Handler

```
private void Search_LoadSettings(object p_Sender,
                                EventArgs p_Args)
{
    CalendarDateSearchSetting setting = (CalendarDateSearchSetting)
    SearchObject.State.Staging
    [CalendarDateSearchSetting.NAME];

    // Update the controls depending of the current setting, if any
    if (setting != null) {
        if (setting.MinimumDate != DateTime.MinValue) {
            CHKUseMinimumDate.Checked = true;
            CALMinimumDate.Enabled = true;
            CALMinimumDate.SelectedDate = setting.MinimumDate;
        } else {
            CHKUseMinimumDate.Checked = false;
            CALMinimumDate.Enabled = false;
        }

        if (setting.MaximumDate != DateTime.MinValue) {
            CHKUseMaximumDate.Checked = true;
        }
    }
}
```

```

        CALMaximumDate.Enabled = true;
        CALMaximumDate.SelectedDate = setting.MaximumDate;
    } else {
        CHKUseMaximumDate.Checked = false;
        CALMaximumDate.Enabled = false;
    }
} else {
    CHKUseMinimumDate.Checked = false;
    CALMinimumDate.Enabled = false;
    CHKUseMaximumDate.Checked = false;
    CALMaximumDate.Enabled = false;
}
}
}

```

### To Update the CalendarDateSearchSetting

Whenever the user modifies the value of one of the child controls (changing the date on a calendar, etc.), the **CalendarDateSearchSetting** object that is persisted across postbacks by the SearchControl has to be updated. To do so, hook up to the proper child controls events that signals when the values changes. This is done in the **CreateChildControls** method presented above in this topic.

Whenever one of these events is fired, it is important to create a new **CalendarDateSearchSetting** that will reflect the updated state, and add it to the SearchControl's Staging setting collection, overwriting any previous similar setting. This is done in a private method called **UpdateSearchSetting** that is called in each event handler.

#### CHKUseMinimumDate\_CheckedChanged

```

private void CHKUseMinimumDate_CheckedChanged(object p_Sender,
                                             EventArgs p_Args)
{
    UpdateSearchSetting();
}

```

#### CHKUseMaximumDate\_CheckedChanged

```

private void CHKUseMaximumDate_CheckedChanged(object p_Sender,
                                             EventArgs p_Args)
{
    UpdateSearchSetting();
}

```

#### CALMinimumDate\_SelectionChanged

```

private void CALMinimumDate_SelectionChanged(object p_Sender,
                                             EventArgs p_Args)
{
    UpdateSearchSetting();
}

```

#### CALMaximumDate\_SelectionChange

```

private void CALMaximumDate_SelectionChanged(object p_Sender,
                                             EventArgs p_Args)
{
    UpdateSearchSetting();
}

```

#### The UpdateSearchSetting method

```

private void UpdateSearchSetting()

```

```

{
    // Build a CalendarDateSearchSetting from the content of our child controls
    CalendarDateSearchSetting setting = new CalendarDateSearchSetting();
    if (CHKUseMinimumDate.Checked) {
        setting.MinimumDate = CALMinimumDate.SelectedDate;
    }
    if (CHKUseMaximumDate.Checked) {
        setting.MaximumDate = CALMaximumDate.SelectedDate;
    }

    // If the setting is non-empty, add, otherwise clear any existing one
    if (!setting.Empty) {
        SearchObject.State.Staging.Add(setting);
    } else {
        SearchObject.State.Staging.Remove(CalendarDateSearchSetting.NAME);
    }
}
}

```

### Creating the CalendarDateSearchSetting Class

The **CalendarDateSearchSetting** class is responsible of holding the state of the **CalendarDateSearch** control across postbacks and adding the proper expressions to the query when it is executed by theSearchControl.

#### To Provide a Name for the Setting

Each search setting class needs to have a unique name that will be used when serializing and deserializing.

##### The NAME constant

```

/// <summary>
/// The name of the <see cref="CalendarDateSearchSetting"/> setting.
/// </summary>
public const string NAME = "CalendarDateSearch";

```

#### To Define the Properties to Store the State

The **CalendarDateSearchSetting** uses two properties to store the state of the control. One holds the minimum restricted date and the other the maximum restricted date. Either of these dates is set to DateTime.MinValue whenever the associated checkbox is not selected.

##### Declaring the Private Fields

```

// The minimum date
private DateTime m_MinimumDate;
// The maximum date
private DateTime m_MaximumDate;

```

##### Properties for Minimum and Maximum Date

```

/// <summary>
/// The minimum date on which the results may have been modified.
/// </summary>
public DateTime MinimumDate
{
    get {
        return m_MinimumDate;
    }
}

```

```

    set {
        m_MinimumDate = value;
    }
}

/// <summary>
/// The maximum date on which the results may have been modified.
/// </summary>
public DateTime MaximumDate
{
    get {
        return m_MaximumDate;
    }
    set {
        m_MaximumDate = value;
    }
}

```

### To Implement the Empty Property

Each search setting class must implement the Empty property. This property should return *True* when the setting class does not contain any significant value.

#### The Empty Property

```

public override bool Empty
{
    get {
        return m_MinimumDate == DateTime.MinValue && m_MaximumDate ==
DateTime.MinValue;
    }
}

```

### To Implement the Constructors

A search setting typically implements two constructors. One of them is used when creating a new setting, while the other is used when deserializing the setting when a new postback is performed.

#### The Default Constructor

```

public CalendarDateSearchSetting()
: base(NAME)
{
}

```

#### The Constructor Used for Deserialization

```

public CalendarDateSearchSetting(StringDeserializer p_Deserializer)
: base(NAME, p_Deserializer)
{
    m_MinimumDate = p_Deserializer.GetDateTime();
    m_MaximumDate = p_Deserializer.GetDateTime();
}

```

### To Implement the Code for Serializing the Setting

The SearchSetting class defines a method called Serialize that must be overridden to properly serialize the content of the setting.

### The Serialize Method

```
public override void Serialize(StringSerializer p_Serializer)
{
    base.Serialize(p_Serializer);

    p_Serializer.Append(m_MinimumDate);
    p_Serializer.Append(m_MaximumDate);
}
```

### To Implement the Code for Applying the Setting to the Query

When the SearchControl performs a search, it uses an object called SearchBuilder in order to accumulate all the information about the query to execute. As part of this process, all the current search settings are given an opportunity to contribute their own part of the query. This is done in the Apply method (for more information, refer to [How to Perform Searches](#)).

### The Apply Method

```
public override void Apply(SearchBuilder p_Builder,
                          SearchControl p_Search)
{
    if (m_MinimumDate != DateTime.MinValue) {
        string expr = CustomFields.SYSDATE + ">=" +
SearchUtilities.DateToString(m_MinimumDate);
        p_Builder.AddAdvancedExpression(expr);
    }
    if (m_MaximumDate != DateTime.MinValue) {
        string expr = CustomFields.SYSDATE + "<=" +
SearchUtilities.DateToString(m_MaximumDate);
        p_Builder.AddAdvancedExpression(expr);
    }
}
```

If either a minimum or maximum date is specified, the code will append a query expression to restrict the search to documents modified in the appropriate time range. Dates need to be converted to string using SearchUtilities.DateToString in order to ensure that the correct format is used.

### To Register the CalendarDateSearchSetting in the Search Setting Factory

Each search setting must be registered with the SearchSettingFactory when the Web application initializes, for a mapping to be established between the search setting name and method responsible of deserializing it. In this case, registration of the **CalendarDateSearchSetting** is done in the static constructor of **CalendarDateSearch**.

### Registration of CalendarDateSearchSetting

```
static CalendarDateSearch()
{
    // Register our setting with the factory
    SearchSettingFactory.RegisterSetting(CalendarDateSearchSetting.NAME, new
SearchSettingFactory.DeserializeDelegate(CalendarDateSearchSetting.Deserialize));
}
```

The code registers the **Deserialize** static method of the **CalendarDateSearchSetting** class as the handler for deserializing those search settings. Here is the implementation of the method. Note that it only calls the proper constructor to deserialize the search setting, described earlier.

### The Deserialize Method

```
public static SearchSetting Deserialize(StringDeserializer p_Deserializer,
```

```

        string p_Custom)
    {
        return new CalendarDateSearchSetting(p_Deserializer);
    }

```

## Displaying a Comment to the User When Search Results are Being Restricted

It is often useful to display a message to the user to mention that search results are restricted for some criteria (modified dates, in this particular case). This can be done by hooking on the `GenerateComments` event fired by the `SearchControl` and adding a comment to the list of those to be displayed.

Comments are represented using the `CommentInfo` class, that stores the message to display along with the list of search settings that should be cleared whenever the user clicks the `Clear` link that is displayed next to each comment. Here is the code for the **CalendarDateSearch** control:

### GenerateComments Event Handler

```

private void Search_GenerateComments(object p_Sender,
                                     GenerateCommentsEventArgs p_Args)
{
    // If there is an effective CalendarDateSearchSetting setting, we want
    // to generate a comment that will be displayed to the user.
    CalendarDateSearchSetting setting = (CalendarDateSearchSetting)
SearchObject.State.Staging
[CalendarDateSearchSetting.NAME];
    if (setting != null && !setting.Empty) {
        // Create a new comment that will remove the CalendarDateSearchSetting
        when cleared
        CommentInfo comment = new CommentInfo();
        comment.Settings.Add(setting);
        // Depending on what we're restricting on the comment will be phrased
        differently
        if (setting.MinimumDate != DateTime.MinValue && setting.MaximumDate !=
DateTime.MinValue) {comment.Comment = String.Format("Only results modified
between {0} and {1} will be displayed.", setting.MinimumDate.ToString("D",
System.Globalization.CultureInfo.CurrentCulture),
setting.MaximumDate.ToString("D",
System.Globalization.CultureInfo.CurrentCulture));
        } else if (setting.MinimumDate != DateTime.MinValue) {comment.Comment =
String.Format("Only results modified after {0} will be displayed."
setting.MinimumDate.ToString("D",
System.Globalization.CultureInfo.CurrentCulture));
        } else if (setting.MaximumDate != DateTime.MinValue) {comment.Comment =
String.Format("Only results modified before {0} will be displayed."
setting.MaximumDate.ToString("D",
System.Globalization.CultureInfo.CurrentCulture ));
        } else {
            // Should never happen!
        }
        p_Args.Comments.Add(comment);
    }
}

```

The `GenerateComments` event is wired up in the `OnInit` phase using the following code:

### Hooking up to the GenerateComments Event

```

SearchObject.GenerateComments += this.Search_GenerateComments;

```

### Sample: Creating a Skin from Scratch

This topic describes the steps required to create a skin from scratch, instead of using a copy of an existing one (for more information concerning skin creation, refer to [Creating a New Skin](#)). The skin that this sample describes implements a simple search interface that displays results in a table, along with facets displayed on top.

Search For:  Format:

Type: <any>	Author: <any>	Cluster: <any>
Web Page (1240) x	David Nguyen (306) x	Online Privacy Policy x
Text (2) x	Loretta Jones (2) x	Newsletter Press Releases x
Excel (1) x	Chip Brush (1) x	Blizzard Insider x
		Blizzard Classic Arcade x
		Visit Blizzard x

Type	Title	Author
	<b>Blizzard Entertainment</b> GameSpot has posted an interview with Blizzard that discusses some of the new features in...	
	<b>Happy Holidays from Blizzard Entertainment!</b> Adorn your desktop with holiday images created by Blizzard Entertainment. ... Representing...	
	<b>Blizzard Entertainment - Blizzard Insider</b> Blizzard Newsletter Press Releases ... Links BLIZZARD INSIDER NEWSLETTER The Blizzard Insider...	
	<b>Blizzard Entertainment - Employment Opportunities</b> At Blizzard, you will work with some of the most creative and talented people in the industry,...	
	<b>Blizzard Entertainment - Inside Blizzard: Copyright and Trademark Information</b> Battle.net® ©1996 - 2002 Blizzard Entertainment. All rights reserved. ... All rights reserved....	
	<b>Blizzard Entertainment - Blizzard Insider</b> You can also get a copy online at the Blizzard Online Store. ... support@blizzard.com http://www....	
	<b>Blizzard Entertainment - Press Releases</b> Reviews ... Blizzard Entertainment® Announces "Blizzard Classic Arcade" Label And The Return Of...	
	<b>Blizzard Entertainment - Inside Blizzard: Legal FAQ</b> Therefore, Blizzard Entertainment® has adopted the unalterable policy of refusing to accept or...	
	<b>Blizzard Entertainment - Common Questions</b> Common Questions about Blizzard (FAQ) Answers to commonly asked questions about Blizzard...	
	<b>Blizzard Entertainment: Technical Support Site</b> - Never test hacks, send them to Blizzard (hacks@blizzard.com) to test ... Please note that...	David Nguyen

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 [Next]

#### To Install and View the Sample

For information on how to obtain and install the samples, refer to [How to Download and Install the Search Interface Samples](#). When this is done, the skin for this sample should be found under *Web/Samples/SkinFromScratch* in the path where CES is installed. To use it, copy the folder under *Web/Coveo/Skins* and setup a search interface to use it (for more information, refer to [Creating a New Skin](#)).

#### About this Skin

A skin consists of one or more user control (.ascx) files, located in a folder under *Web/Coveo/Skins* in the CES installation. This folder should at least contain a user control named *CoveoSearch.ascx* included in an *ASP.NET* page by the SearchInterface control. This file may include other user controls, but is not

strictly required and is up to the skin developer. In this sample, all the code required is included within *CoveoSearch.ascx*.

### Control Directives

Typically, a user control file begins with a list of directives that will be processed by the *ASP.NET* runtime to determine the base class for the user control, import namespaces and map libraries of controls to prefixes. Here is the code for the *CoveoSearch.ascx* used in this sample:

#### ASP.NET Directives

```
<%@ Control Language="c#" AutoEventWireup="false"
Inherits="Coveo.CES.Web.Search.Controls.SearchControl, Coveo.CES.Web.Search,
Version=5.0.0.0,
Culture=neutral, PublicKeyToken=44110d16825221f2" %>
<%@ Register TagPrefix="ces" Namespace="Coveo.CES.Web.Search.Controls"
Assembly="Coveo.CES.Web.Search,
Version=5.0.0.0, Culture=neutral, PublicKeyToken=44110d16825221f2" %>
<%@ Assembly Name="Coveo.CNL, Version=5.0.0.0, Culture=neutral,
PublicKeyToken=44110d16825221f2"
<%@ Assembly Name="Coveo.CNL.Web, Version=5.0.0.0, Culture=neutral,
PublicKeyToken=44110d16825221f2"
>
<%@ Import Namespace="Coveo.CES.Web.Search" %>
```

#### Search Controls

The sample skin provides a text box to enter the query, a drop-down list that allows to select among common file formats, as well as a search button. All these functionality are implemented using the stock controls provided by Coveo. Here is the code that includes these controls in the skin:

```
<table cellpadding="5" cellspacing="0">
<tr>
<td>
Search For:
</td>
<td>
<ces:Query id="qry" runat="server"/>
</td>
<td>
Format:
</td>
<td>
<ces:Format runat="server"/>
</td>
<td>
<ces:SearchButton id="sbt" runat="server"/>
</td>
</tr>
</table>
```

#### Horizontal Facets

In this sample, the facets are placed in an horizontal fashion above the results. Here is the code to do so:

```
<table width="100%" cellpadding="5" cellspacing="0" style="border: 1px solid
silver; margin-bottom:
5px">
<tr>
```

```

    <td valign="top">
      <ces:RefineByType ControlStyle-BorderWidth="0" runat="server"/>
    </td>
    <td valign="top">
      <ces:RefineByAuthor ControlStyle-BorderWidth="0" runat="server"/>
    </td>
    <td valign="top">
      <ces:RefineByCluster ControlStyle-BorderWidth="0" runat="server"/>
    </td>
  </tr>
</table>

```

## The Results

The results are, as usual, rendered through the use of the ResultList control, using a header and footer to render the opening and closing tags for the table. Each result is rendered in a separate table

row (TR). Here is the code:

```

<ces:ResultList runat="server">
  <Header>
    <table width="100%" cellspacing="0" cellpadding="2" border="1">
      <tr>
        <th>
          Type
        </th>
        <th>
          Title
        </th>
        <th>
          Author
        </th>
      </tr>
    </Header>
    <ResultTemplate>
      <tr>
        <td align="center">
          <ces:ResultFormatIcon runat="server"/>
        </td>
        <td valign="top">
          <div>
            <ces:ResultOpenLink runat="server">
              <ces:ResultTitle id="ttl" runat="server"/>
            </ces:ResultOpenLink>
          </div>
          <div style="font-size: 8pt; color: gray">
            <ces:ResultExcerpt Length="100" runat="server"/>
          </div>
        </td>
        <td>
          <ces:ResultAuthor runat="server"/>
        </td>
      </tr>
    </ResultTemplate>
  <Footer>
    </table>
  </Footer>
</ces:ResultList>

```

## The Pager

The skin also includes a Pager control to allow the user to jump to other result pages.

```
<center>  
<ces:Pager runat="server"/>  
</center>
```

## Upgrade Notes

Version 5.2 introduces breaking changes to the Search Interface API that may require a recompilation if custom Search Interface controls have been authored. These changes were required in order to support some of the new features that 5.2 presents, such as email conversation folding and result merging.

In previous versions, the Search Interface API directly exposed some of the COM objects used to communicate with the CES server. These objects have been replaced by *.NET* interfaces providing similar functionality. Most members are still available as they were before; however, some have been renamed in order to improve consistency. When this was done, an equivalent member using the previous name was also kept, but flagged obsolete.

Customizations that are built in a DLL with Visual Studio should be recompiled after upgrading to version 5.2. Most of the time, no changes to the source code is required. Customizations that are using inline code directly within *aspx* or *ascx* files do not require a recompilation, as this is automatically done by the ASP.NET runtime.