5

Consuming 3rd party components

This chapter covers

* Learning the basic MvcContrib Grid
* Advanced MvcContrib Grid techniques
* Uploading files with SlickUpload

The ASP.NET MVC Framework provides a lot of control out of the box but that comes at a cost. The HTML helpers are basic and provide very simple user interface elements. This leaves you with the choice to hand craft nice user interfaces using HTML and CSS. While this is a great option for an experienced web designer, most developers find relying on a 3rd party component much more productive. This allows you to develop your application rather than spend lots of time on user interface infrastructure. This chapter will demonstrate two 3rd party components that will show you different styles of integrating with the MVC framework. The first is the Grid component available from the open source MvcContrib project that can be used to render an HTML table. The second component is the SlickUpload component for uploading large or multiple files. First let's look at the MvcContrib Grid.

5.1 The MvcContrib Grid

The MvcContrib Grid is a user interface component which creates a well formed HTML table. It uses a fluent interface, which allows you to define the configuration of the grid with a strongly typed and refactoring-friendly syntax. The refactoring support makes this style of component work very nicely with refactoring tools like JetBrains ReSharper and DevExpress Refactor Pro. This type of component generally requires a strongly typed view, which is used to drive the API of the grid. The scenario where you would want to use a grid like this would be to display a list of model objects. First let's look at the action method which would send a list or IEnumerable<Person> to a view for rendering.

Listing 5.1 The Action to render a list of Person objects

public ActionResult AutoColumns() { |#A

return View(\_peopleFactory.CreatePeople()); |#A

}

#A Passes IEnumerable<Person> to the view.

Listing 5.1 shows the action that will send a IEnumerable model to the view for rendering. This is a fairly common scenario. This example ignores more advanced features like paging. It would simply send every Person object in the application to the view for rendering. The next step is to use the MvcContrib Grid to see how easily we can get a table formatted view of our Person objects.

Listing 5.2 Rendering a table using the Grid AutoGenerateColumns

<%= Html.Grid(Model).AutoGenerateColumns() %> #A

#A Renders a table from the model.

The AutoGenerateColumns method will automatically generate columns in the table based on the public properties of the Person object, as shown in Figure 5.1. However, this is only useful in certain situations. You will see from Figure 5.1 that there are some columns for which the grid does not know how to render a value. The default behavior is to call ToString on each property value. However, this is not particularly useful for complex types as this just displays the type name, as illustrated in Figure 5.1. Because of this, AutoGenerateColumns is most useful if you're using a dedicated Presentation Model rather than a nested object hierarchy.

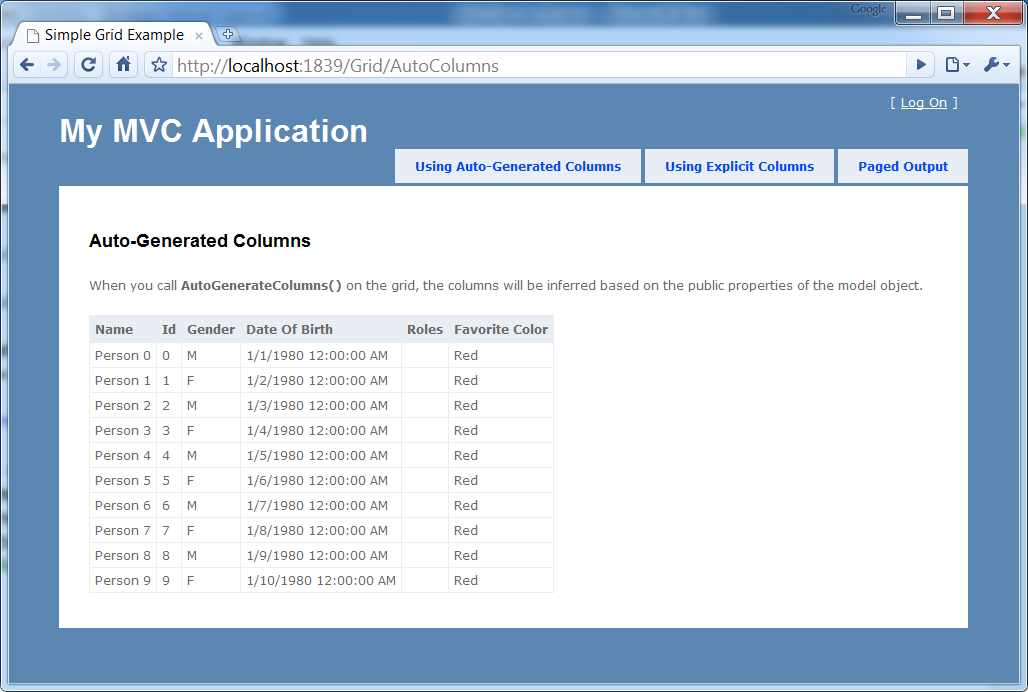


Figure 5.1 - The view of a Grid.AutoGenerateColumns

5.2 The MvcContrib Grid advanced usage

While the previous example of the MvcContrib Grid seemed to just work magically with a single line of view code, it has some pretty strong opinions about how it is going to render a model. For example, it assumes that all public properties should be rendered as columns (unless they are decorated with the ScaffoldColumn attribute). If you do not like this behavior then you do have some more options—this is where the power of the Grid really comes into play. Listing 5.3 shows how you can use the grid to customize the output for individual columns.

Listing 5.3 The MvcContrib Grid with more control

<%= Html.Grid(Model).Columns(column => { #A

column.For(x => x.Id).Named("Person ID"); #B

column.For(x => x.Name); |#C

column.For(x => x.Gender); |#C

column.For(x => x.DateOfBirth).Format("{0:d}"); #D

column.For(x => Html.ActionLink("View Person", "Show", |#E

new { id = x.Id})).DoNotEncode(); |#E

}) %>

#A Specifies individual columns

#B Overrides column name

#C Rendered using default conventions

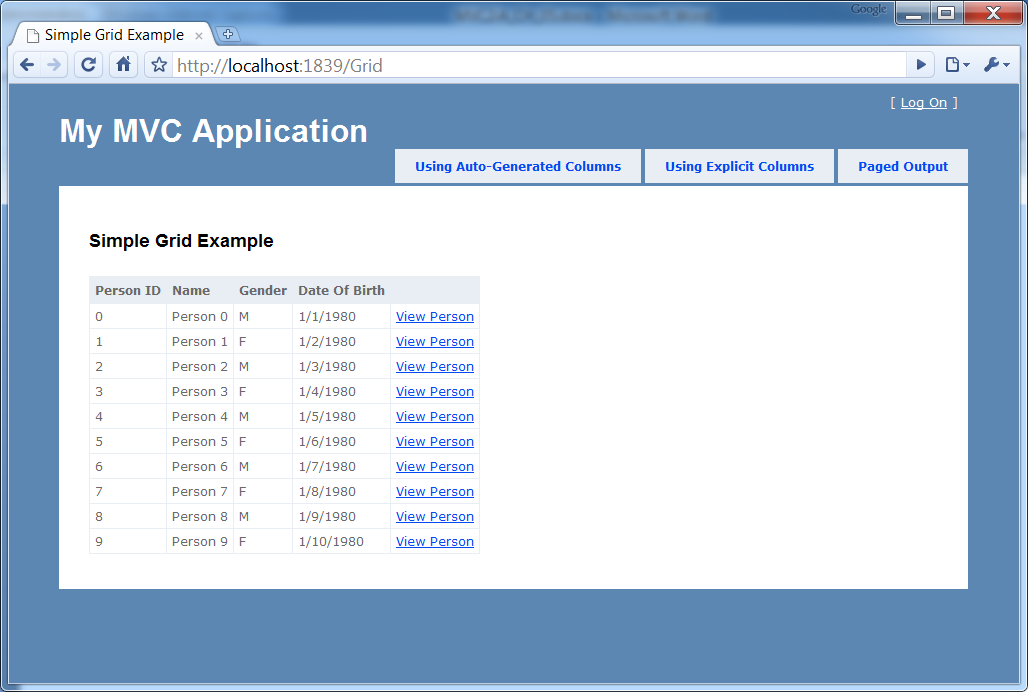
#D Specifies custom output format

#E Defines a custom column

The MvcContrib Grid created using the view code from Listing 5.3 will render nicely in a table as shown in Figure 5.2.

The main reason to explicitly specify the columns for the grid is to be able to customize the output of various columns (for example, by using a custom string format or to add additional columns to the table). The syntax for defining the grid may look odd at first—it uses some of the newer features of the C# language. For example, lambda expressions are used to specify which properties should be rendered as columns in the table. By using this syntax, when you change the name of a property using a refactoring tool, the property gets changed in your view code. This eliminates the runtime errors that you would see when using magic strings and late binding to configure how to pull property values out of your model and render them into a table.

While the MvcContrib was one of the first components to use this method of configuration this style has caught on. The Grid was created and is currently maintained by Jeremy Skinner, a committer on the MvcContrib project. For more information about the Grid, go to the MvcContrib project at http://www.MvcContrib.org and see more information and blog posts from the creator of the grid at http://www.jeremyskinner.co.uk. There are a large number of additional features built in to the grid that we cannot cover in this chapter. The MvcContrib project has a number of samples which walk through the extensive options for using the Grid.

Figure 5.2 The MvcContrib Grid rendered using Column configuration

5.3 The SlickUpload file uploading component

For small files, the default HTML file input element works quite well. However, its usefulness tends to wane as the desire for better feedback about file uploads grows. For example, we do not see file progress, upload speed, and failures tend to be difficult to detect. On the server side, large files pose a particularly difficult problem. If we want to display a file upload progress bar to the user or stream the file directly to disk instead of loading it to memory first, we start developing more and more complex extensions.

However, many 3rd-party libraries exist solely to tackle the difficult issue of allowing users to upload files to the server. One such library is the SlickUpload product from KrystalWare (<http://krystalware.com/Products/SlickUpload/>). While many free options exist, SlickUpload offers many benefits over its competitors, including:

* Multiple file uploads
* Detailed progress information, including percent complete, upload speed and more
* Handling large files (up to 4GB) without crashing the server
* Streaming directly to file or to a database
* Extensive documentation
* Support available

So how does SlickUpload work? Like many high-performance uploading components, SlickUpload processes uploads through an IHttpModule, bypassing much of the ASP.NET pipeline. By using an IHttpModule, files can be streamed directly to disk instead of loaded into memory. If large files are loaded into memory as is the case with the default file-uploading processing in ASP.NET, a large file can take down the entire server by consuming all available memory.

To use the SlickUpload component, we'll first need to add a reference to the KrystalWare.SlickUpload assembly. There is no need to install anything, as the SlickUpload component is only a single deployed .NET assembly. Next, we will need to modify our web.config file to configure and enable SlickUpload in our application. In listing 5.4 below, we add the configuration sections to the configSections group.

Listing 5.4 Adding the SlickUpload configuration sections

<configSections>

<sectionGroup name="slickUpload"

type="Krystalware.SlickUpload.Configuration.NameValueConfigurationSectionHandler, Krystalware.SlickUpload">

<section name="uploadParser"

type="Krystalware.SlickUpload.Configuration.NameValueConfigurationSectionHandler, Krystalware.SlickUpload"/>

<section name="uploadStreamProvider"

type="Krystalware.SlickUpload.Configuration.NameValueConfigurationSectionHandler, Krystalware.SlickUpload"/>

<section name="statusManager"

type="Krystalware.SlickUpload.Configuration.StatusManagerConfigurationSectionHandler, Krystalware.SlickUpload"/>

</sectionGroup>

These sections enable the custom component-specific SlickUpload configuration sections. Next, in the slickUpload section in listing 5.5, we turn off the handleRequests feature, as we will later configure a specific path for handling requests.

Listing 5.5 Turning off global SlickUpload request handling

<slickUpload>

<uploadParser handleRequests="false" />

</slickUpload>

With the global handling turned off, we now want to configure a specific path for handling uploads. The SlickUpload AJAX client component will send requests to this path, instead of the normal form target for processing the file. Listing 5.6 below includes the complete path-specific SlickUpload configuration.

Listing 5.6 Configuring location-specific SlickUpload information

<location path="SlickUpload.axd"> #1

<slickUpload>

<uploadParser handleRequests="true" /> #2

<uploadStreamProvider

provider="File" #3

location="~/Files/" #4

existingAction="Overwrite" /> #5

</slickUpload>

<system.web>

<httpRuntime maxRequestLength="1048576" executionTimeout="300"/> #6

</system.web>

<system.webServer>

<security>

<requestFiltering>

<requestLimits maxAllowedContentLength="2072576000"/> #7

</requestFiltering>

</security>

</system.webServer>

</location>

For the specific path to the SlickUpload handler, "SlickUpload.axd" (1), we first turn upload parsing back on (2). Next, we configure the upload stream provider to use files (3). We upload files to a Files folder (4) and existing files will be overwritten (5). Next, we need to configure ASP.NET to handle larger files. First, we'll set the maximum request length to a much larger value (6), as well as configure the maximum allowed content length to something on the order of 2 gigabytes (7). Each of these configuration settings is in place to ensure that ASP.NET does not detect large files and abort the file upload. These settings can depend on the available disk space, so we may need to adjust these values depending on the production environment. In our example, we save files to the local disk, but we could also save to the database or to a network share.

The final piece of web.config modifications we need to include are the custom IHttpModule and IHttpHandler, shown in listing 5.7 below.

Listing 5.7 Adding the SlickUpload HttpHandler and HttpModule

<httpHandlers>

<remove verb="\*" path="\*.asmx"/>

<add verb="\*" path="\*.asmx" validate="false"

type="System.Web.Script.Services.ScriptHandlerFactory, System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>

<add verb="\*" path="\*\_AppService.axd" validate="false"

type="System.Web.Script.Services.ScriptHandlerFactory, System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>

<add verb="GET,HEAD" path="ScriptResource.axd"

type="System.Web.Handlers.ScriptResourceHandler, System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" validate="false"/>

<add path="SlickUpload.axd" verb="GET,HEAD,POST,DEBUG" #1

type="Krystalware.SlickUpload.SlickUploadHandler, Krystalware.SlickUpload" />

</httpHandlers>

<httpModules>

<add name="ScriptModule"

type="System.Web.Handlers.ScriptModule, System.Web.Extensions, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>

<add name="UrlRoutingModule"

type="System.Web.Routing.UrlRoutingModule, System.Web.Routing, Version=3.5.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />

<add name="HttpUploadModule" #2

type="Krystalware.SlickUpload.HttpUploadModule, Krystalware.SlickUpload"/>

</httpModules>

We may have more or less existing handlers and modules, but we just need to add the custom IHttpHandler (1) and IHttpModule (2) to the end of the list. With SlickUpload referenced and configured in our web.config file, we can now create a controller and view to allow the user to upload files. We'll create a very simple screen to upload files, with the Index action displaying a simple form. The controller in listing 5.8 merely returns a ViewResult.

Listing 5.8 The UploadController's Index action

public class UploadController : Controller

{

public ActionResult Index()

{

return View();

}

SlickUpload uses a traditional Web Control to process file uploads. However, because we can still use Web Controls in an MVC application, the SlickUpload control will not pose a problem for us. We also have additional configuration options that enable MVC scenarios, such as hosting in a non-server control form tag. In listing 5.9, we see the Index view including the form tag and SlickUpload control.

Listing 5.9 The Index view using the SlickUpload web control

<% using (Html.BeginForm("UploadResult", "Upload", FormMethod.Post, #1

new { id = "uploadForm", enctype = "multipart/form-data" })) { %> #2

<kw:SlickUpload ID="SlickUpload1" runat="server"

UploadFormId="uploadForm" MaxFiles="1" #3

ShowDuringUploadElements="cancelButton" #4

HideDuringUploadElements="uploadButton"> #5

<DownlevelSelectorTemplate>

<input type="file" /> #6

</DownlevelSelectorTemplate>

<UplevelSelectorTemplate>

<input type="button" value="Add File" /> #7

</UplevelSelectorTemplate>

<FileTemplate>

<kw:FileListRemoveLink runat="server">

[x] remove</kw:FileListRemoveLink> #8

<kw:FileListFileName runat="server" />

<kw:FileListValidationMessage runat="server" ForeColor="Red" />

</FileTemplate>

<ProgressTemplate>

<table width="99%"><tr><td>

<p>Upload Progress:</p>

<div class="progressBorder">

<kw:UploadProgressBarElement runat="server" CssClass="progressBar"/>

<div class="progressValue">

<kw:UploadProgressElement runat="server" #9 Element="PercentCompleteText">

(calculating)

</kw:UploadProgressElement>

</div>

</div>

</td></tr></table>

</ProgressTemplate>

</kw:SlickUpload>

<hr />

<p>

<input type="submit" value="Upload" id="uploadButton" /> #10

</p>

<% } %>

To build our file upload form, we first need to build the outermost form HTML tags with the Html.BeginForm method (1). The target will be the UploadResult action of the Upload controller, which will be the action redirected to after the upload is complete. To ensure our form works correctly with the browser and the SlickUpload control, we give the form a unique ID and set the encoding to "multipart/form-data" (2). Next, we add the SlickUpload control (3), matching the upload form ID to the form tag's ID and setting the maximum number of uploaded files to one. The control allows adding multiple files, but we will restrict the number of files to one for this example.

The next two configuration properties match up to a cancel button (which we leave off) (4) and the upload button (5). The values match up to HTML element identifiers, as our input button for initiating the upload has an ID of "uploadButton" also (10).

With the control configured, we supply a set of templates for the file selector, file template and progress template. To support older browsers, we configure both the down-level (6) and up-level (7) file templates. Older browsers are presented with the normal file input, while newer browsers are presented with a simple button. In figure 5.3 below we see the site showing the file selector template.

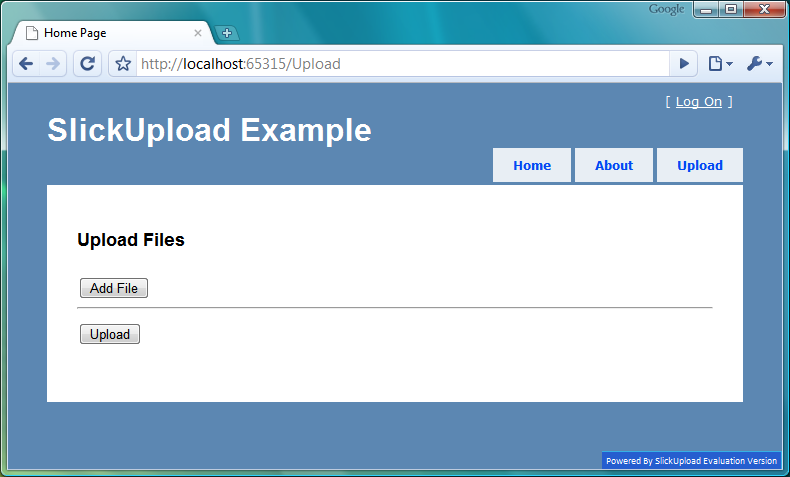


Figure 5.3 The upload file screen showing the file selector template

Once the user chooses a file, the file template is shown for each file. In this example, we include the name of the file and a remove link (8). If we allowed multiple file uploads, the user could select to remove a file from the list before uploading the entire group of files. Because we only allow one file, the Add File button is hidden once a file is chosen, as shown in figure 5.4.

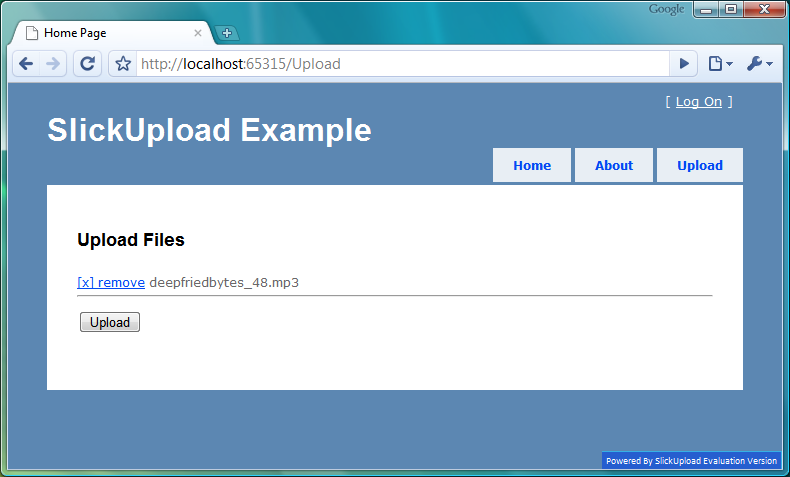


Figure 5.4 The upload files screen showing the file listing template

Finally, we configure the progress template. The upload progress template is shown to the user after they click the "Upload" button. We just show a simple progress bar (10) using the supplied SlickUpload controls. However, we could show much more information including the file count, current file being uploaded, upload speed and time remaining. Our site in figure 5.5 below shows the progress bar and percentage complete.

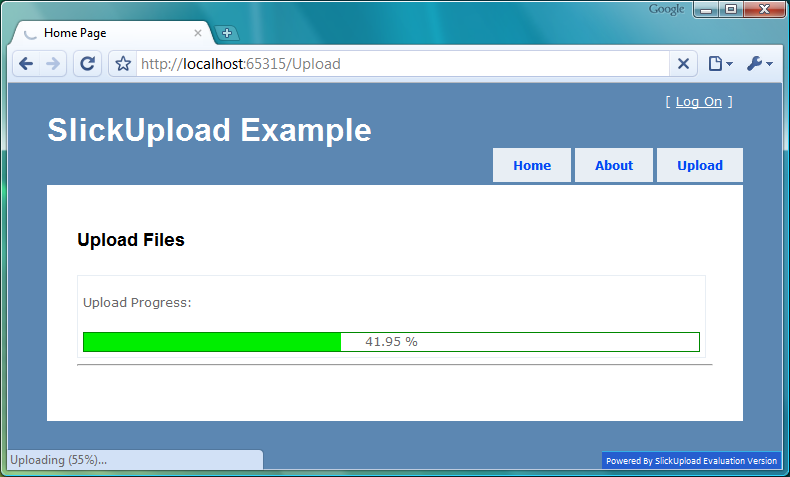


Figure 5.5 The file upload screen with the progress bar shown

The file we choose was quite large, around 64 megabytes. A progress indicator is a great mechanism to provide the user feedback that their file is being uploaded. Without a progress bar, the user receives no information on the progress of their upload. In practice, many users believe that the upload is not being processed, and will refresh the form or stop the upload.

The final piece we need to implement is the action that we redirect to after the file upload is successful. We will very likely need to retrieve some sort of information about the file uploaded, especially if we want to store file metadata somewhere else for processing and viewing. In listing 5.10, our UploadResult action uses the SlickUpload UploadConnector class to retrieve the UploadStatus for the uploaded files. The UploadStatus class contains file metadata which we can then process as needed.

Listing 5.10 The UploadResult action

public ActionResult UploadResult()

{

UploadStatus status = UploadConnector.GetUploadStatus();

return View(status);

}

Typically, we store file metadata in a database as we will likely want to be able to show the user a list of uploaded files to download. With the file metadata in a database, we can show this information much more easily than trying to read the file information from the disk. In our sample, we will only display the metadata information in the view, shown in listing 5.11.

Listing 5.11 Displaying the file metadata

<p>Result: <%=ViewData.Model.State%></p> #1

<table class="results" width="99%" cellpadding="4" cellspacing="0">

<thead>

<tr>

<th align="left">Name</th>

<th align="left">Mime Type</th>

<th align="left">Length (bytes)</th>

</tr>

</thead>

<tbody>

<% foreach (UploadedFile file in ViewData.Model.GetUploadedFiles()) {%>#2

<tr>

<td><%=file.ClientName %></td> #3

<td><%=file.ContentType %></td> #4

<td><%=file.ContentLength %></td> #5

</tr>

<% } %>

</tbody>

</table>

Our view is passed an UploadStatus object, which contains upload status as well as file information. We first show the result of the upload (1), which indicates success or failure. Next, we display a table of the uploaded files (2). We show the name (3), content type (4) and content length (5). In figure 5.6 we see the final upload results screen.

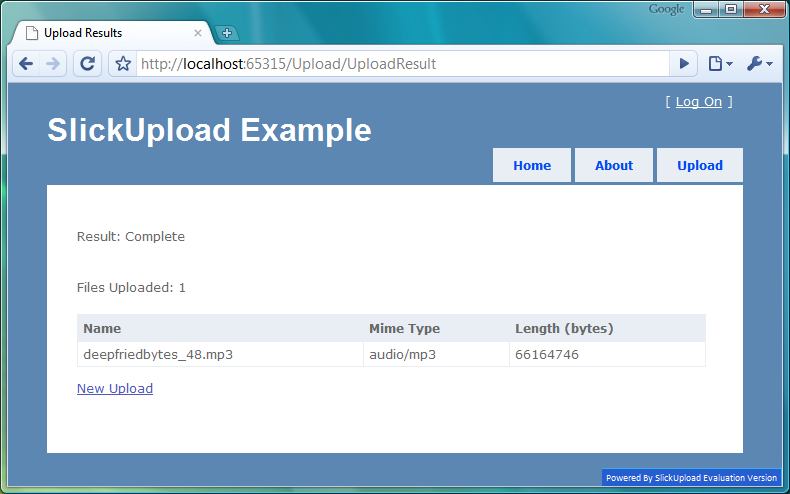


Figure 5.6 The upload results screen

Since we have the file size, mime type and file name, we can provide a good file downloading solution. Supplying file downloads have a much better end-user experience if we supply the file size, mime type and name. The browser's download file dialog box uses this information to provide a download progress bar.

SlickUpload isn't the only file upload component, but it works well with ASP.NET MVC. We used a web control, but it is fully supported in an ASP.NET MVC environment. Using a file streaming component prevents end users from crashing our web server by uploading large files. With the configuration abilities of SlickUpload, we have a lot of flexibility in processing uploads without resorting to a lot of custom code.

5.5 Summary

This chapter covered using 3rd party components in an MVC application. We covered using a page level component, the MvcContrib Grid. We walked through using the Grid's AutoGenerateColumns feature for simple cases. We also demonstrated a more advanced usage of the Grid, using its powerful strongly typed API. We also looked at integrating a 3rd party upload component product, SlickUpload. With its configuration options, performance benefits and progress bar, we were able to provide a good user experience for uploading files. These two different types of components show that there are different ways to look at how much functionality a component could provide. The Grid provides a single control like experience, while the SlickUpload component demonstrates integrating existing web control products. By using these components you can provide more functionality quickly. The next chapter will cover hosting a MVC application in Internet Information Server.