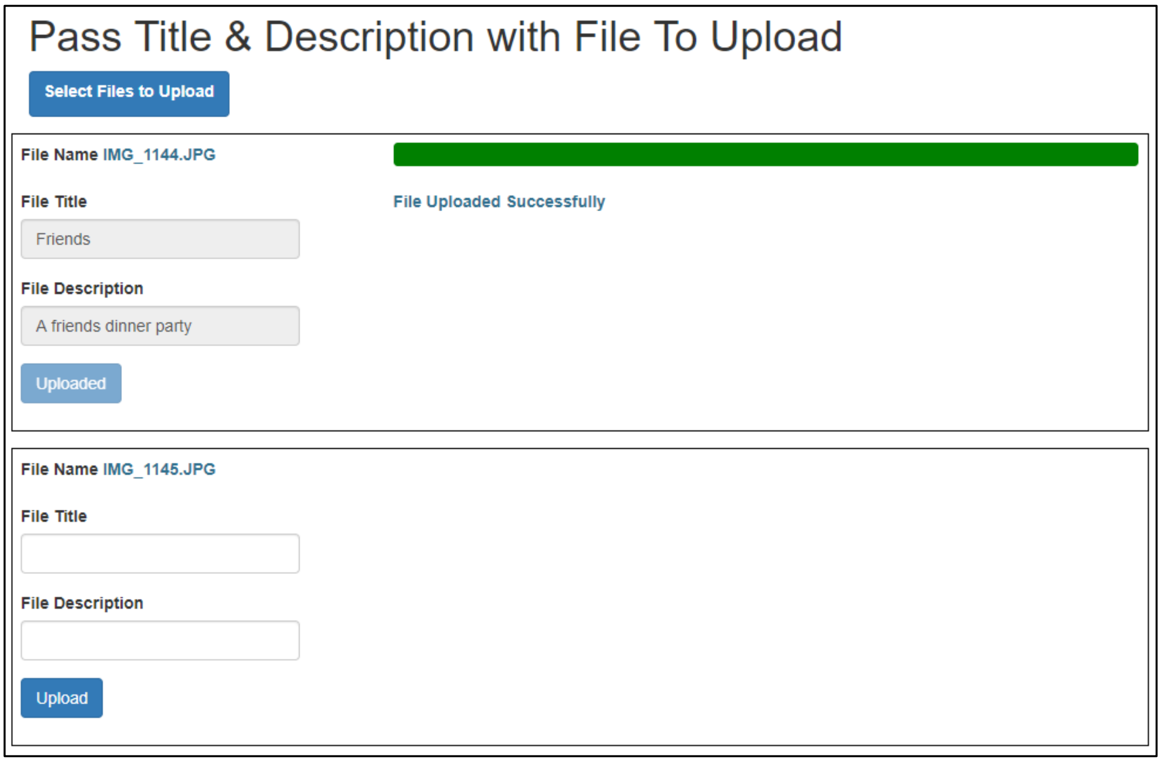
Upload Multiple Files Asynchronously - Part 2

In this blog post, you are going to code a web page that allows a user to select one or many files, and display a title and description input area for each file. Once the user adds a title and a description, they click on an Upload button to upload the file selected (see Figure 1), along with the title and the description entered about that file. Each file is uploaded asynchronously through a Web API call and the progress is reported in a progress bar that appears on the web page.



Pass Title & Description with File To Upload

Select Files to Upload

File Name IMG_1144.JPG

File Title

Friends

File Description

A friends dinner party

Uploaded

File Uploaded Successfully

File Name IMG_1145.JPG

File Title

File Description

Upload

Figure 1: Allow a user to add a title and a description for each file selected for uploading.

Create HTML Page for Uploading Files

To build this sample, I am going to use Visual Studio, jQuery, Bootstrap 3.x and the Microsoft ASP.NET Web API. However, feel free to use whatever tools you want. All the front-end code is generic and can be used with any JavaScript framework.

While I am using Microsoft's ASP.NET Web API, you should be able to adapt the code to any Web API framework with ease.

To start creating this sample, bring up Visual Studio and select the ASP.NET Web Application (.NET Framework) project. Choose the **Empty** template but select the **Web API** option. Using the **NuGet Package Manager**, add Bootstrap 3.4.x to your project. Make sure you are installing Bootstrap 3.4.x and not Bootstrap 4.x. Installing Bootstrap 3.4.x also installs jQuery. After installing these, click on the **Updates** tab in the NuGet Package Manager window and update any packages, except Bootstrap.

Add an index.html page in the root of your project. Locate the <head> element and modify it to look like the following code.

```
<head>
  <title>Upload Multiple Files Sample</title>
  <meta charset="utf-8" />
  <meta name="viewport"
        content="width=device-width, initial-scale=1" />

  <link href="/Content/bootstrap.min.css" rel="stylesheet" />
  <link href="/Content/site.css" rel="stylesheet" />
</head>
```

Right mouse-click on the **Content** folder and add a new style sheet named **site.css**. Add the following rules in the site.css file.

```
.body-content {
   margin-top: 2em;
}

.file-upload-info-area {
  border: solid black .1em;
  margin-top: 1em;
  margin-bottom: .5em;
  padding: .5em;
}
```

Back in the index.html page, just below the <body> element, add a <div class="container"> element. This is the main wrapper for Bootstrap to contain all other Bootstrap CSS classes.

```
<div class="container body-content">
  <h1>Pass Title & Description with File To Upload</h1>

</div>
```

Add File Input Element

Just below the <h1> element, add another <div> with the *class* attribute set to "form-group btn btn-primary". These Bootstrap classes style the file input element to look like a Bootstrap button instead of a normal HTML button. Add a label and an input type within the div. Notice the input element has the *multiple* attribute set. In addition, it has the *style* attribute set to "display:none;". This allows the Bootstrap CSS classes to style the file input element. The *onchange* event calls a method in a closure named "uploadController.uploadFiles()". You are going to add this closure and method a little later in this blog post.

```
<div class="form-group btn btn-primary">
  <label for="fileUploadControl">Select Files to Upload</label>
  <input type="file"
         id="fileUploadControl"
         onchange="uploadController.addFiles();"
         multiple="multiple"
         style="display:none;" />
</div>
```

Add Area to Gather User Input

Below the div that contains the file input element, add another div with an *id* attribute set to "fileUploadProgressArea". It is within this div you display the user inputs for title and description. You also display a label with the file name and a progress bar for each file the user selects to upload.

```
<!-- BEGIN: File Upload Progress Area -->
<div id="fileUploadProgressArea">
  <!-- This area is where each file to be uploaded is displayed -->
</div>
<!-- END: File Upload Progress Area -->
```

Add Template for Each Progress Bar

Below the file upload progress area, add a <script> tag. Set the *id* attribute to "fileUploadProgressTemplate" for this script tag and the *type* attribute to "text/html". This lets the browser know that this script tag does not contain any executable JavaScript, but plain text that should be ignored. The HTML contained within this script tag is used to display the title and description input fields, an Upload button, the file name being uploaded, and the progress bar as shown in Figure 1.

```
<!-- BEGIN: Display File Upload Progress Template -->
<script id="fileUploadProgressTemplate" type="text/html">
  <div class="row file-upload-info-area">
    <div class="col-md-3">
      <div class="row">
        <div class="form-group">
          <label for="fileUploadLabel">File Name</label>
          <label id="fileUploadLabel" class="text-info"></label>
        </div>
        <div class="form-group">
          <label for="fileUploadTitle">File Title</label>
          <input type="text"
                 id="fileUploadTitle"
                 class="form-control" />
        </div>
        <div class="form-group">
          <label for="fileUploadDescription">
            File Description
          </label>
          <input type="text"
                 id="fileUploadDescription"
                 class="form-control" />
        </div>
        <div class="form-group">
          <button id="fileUploadButton"
                  class="btn btn-primary">
            Upload
          </button>
        </div>
      </div>
    </div>
    <div class="col-md-offset-1 col-md-8">
      <div class="row">
        <div class="form-group">
          <div class="progress hidden">
            <progress id="fileUploadProgressBar"
                      class="progress-bar progress-bar-success"
                      value="0"
                      max="100"
                      style="width: 100%">
            </progress>
          </div>
        </div>
        <div class="form-group">
          <label id="fileUploadProgressComplete"
                 class="text-info hidden">
            File Uploaded Successfully
          </label>
        </div>
      </div>
    </div>
  </div>
</script>
<!-- END: Display File Upload Progress Template -->
```

You are going to write some JavaScript/jQuery code to clone the HTML within this script tag. Once cloned, you are going to insert the cloned HTML into the div tag with the *id* of "fileUploadProgressArea".

Add jQuery and Bootstrap

Add two more script tags to reference jQuery and the Bootstrap JavaScript library from within the Scripts folder where they are installed by NuGet.

```
<script src="/Scripts/jquery-3.4.1.min.js"></script>
<script src="/Scripts/bootstrap.min.js"></script>
```

Make sure you check to see what version of jQuery is loaded after you update all the packages using the NuGet Package Manager. You may need to change the version of jQuery listed above. Also, strictly speaking, Bootstrap does not need to be included on this page for this sample, but I like to include it just in case I use some features of Bootstrap later.

The uploadController Closure

A best practice in JavaScript is to create closures around code. The *uploadController* closure you attached earlier to the onchange event of the file input element is what you are going to write now. To build this closure, add a new <script> tag and add the closure and the stubs of the various methods you are going to write to upload files, display the file name, and show the progress bar.

```
<script>
  var uploadController = (function () {
    /************************
    * Private Variables
    ************************/
    const UPLOAD_URL = "/api/FileUpload/UploadFile";
    var fileObjects = [];

    /************************
    * Private Functions
    ************************/
    function addFiles() {
    }

    function uploadFileInfo(index) {
    }

    function cloneProgressBarTemplate(fileToUpload, index) {
    }

    function postFile(data, index) {
    }

    function updateProgressBar(e, index) {
    }

    /************************
    * Public Functions
    ************************/
    return {
      uploadFiles: uploadFiles ,
      uploadFileInfo: uploadFileInfo
    }
  })();
</script>
```

addFiles() Method

Once the user selects one or more files using the file input element, the *onchange* event is fired. That event calls the addFiles() method. The first thing to do is to grab the collection of files by accessing the files method from the file input element that has the id "fileUploadControl".

```
function addFiles() {
  // Get collection of files selected by user
  var files = $("#fileUploadControl")[0].files;

  // Loop through collection of files selected by user
  for (var i = 0; i < files.length; i++) {
    // Add to files array
    fileObjects.push({
      "index": i,
      "uploadObject": files[i]
    });

    // Create new <div> for displaying file,
    // title, description and progress
    cloneProgressBarTemplate(files[i], i);
  }
}
```

Once you have the collection of files the user selected, loop through this collection and, each time through the loop, store the index number and the file object to upload into the *fileObjects* array. Storing this data into a local array makes your code simpler for retrieving each file object later in the code.

After adding the file information to the *fileObjects* array, pass the file object and index number to the cloneProgressBarTemplate() method. It is in this method where the title and description input fields are created and added to the HTML page.

cloneProgressBarTemplate() Method

As you can see in Figure 1, the file name, a title, and description input areas are displayed on the HTML page. The cloneProgressBarTemplate() method is responsible for creating all of this new HTML for each file passed in. Instead of writing JavaScript or jQuery code to create each of element required for the file name label, the label and inputs for the tile and description and a progress bar element, you create the HTML within the <script> tag with the *id* attribute set to "fileUploadProgressTemplate". It is much easier to layout the HTML within an HTML editor than it is to write the same code using JavaScript.

To create a new DOM element from the HTML within the <script> tag, use the following line of code.

```
var elem = $($("#fileUploadProgressTemplate").html()).clone();
```

The variable *elem* is now a new DOM object and you can append that wherever you wish using the append() method. For example, you created an empty <div> tag earlier in this blog post with the *id* attribute set to "fileUploadProgressArea". Use the following line of code to insert this newly cloned DOM object into that <div> tag.

```
$("#fileUploadProgressArea").append(elem);
```

If you only needed a single DOM object for a single file, this is all the code you would need. However, you can have multiple file objects selected by the user, so you need to build multiple DOM objects. As you know, you are not allowed to have multiple HTML elements with the same *id* attribute value. Therefore, you pass in the index value of each file along with the file object. You use this index number to create a unique *id* attribute value after you have cloned the HTML.

There are a few HTML elements within the <script> tag you are cloning that have *id* attributes; "fileUploadLabel" and "fileUploadProgressBar" to name just a couple. You are going to use the find() method on the new DOM element to locate any elements with an *id* attribute. Set the *id* attribute to the name of the element plus an underscore (_) plus the index number. Thus, the first file passed to this method from the addFiles() method passes a zero (0) in the index, so the *id* attributes are "fileUploadLabel_0" and "fileUploadProgressBar_0" respectively. The second time through the id attributes are "fileUploadLabel_1" and "fileUploadProgressBar_1", and so on.

```
function cloneProgressBarTemplate(fileToUpload, index) {
  // Clone the HTML from the template
  var elem = $($("#fileUploadProgressTemplate").html()).clone();

  // Replace the 'id' attributes with current index number
  $(elem).find("#fileUploadLabel").attr("id",
    "fileUploadLabel_" + index);
  $(elem).find("#fileUploadTitle").attr("id",
    "fileUploadTitle_" + index);
  $(elem).find("#fileUploadDescription").attr("id",
    "fileUploadDescription_" + index);
  $(elem).find("#fileUploadProgressBar").attr("id",
    "fileUploadProgressBar_" + index);
  $(elem).find("#fileUploadButton").attr("id",
    "fileUploadButton_" + index);
  $(elem).find("#fileUploadProgressComplete").attr("id",
    "fileUploadProgressComplete_" + index);

  // Attach a click event
  $(elem).find("#fileUploadButton_" + index).on("click",
     function () {
       uploadController.uploadFileInfo(index);
     }
  );

  // Append this new HTML to the file upload area
  $("#fileUploadProgressArea").append(elem);

  // Display the file name
  $("#fileUploadLabel" + "_" + index).text(fileToUpload.name);
}
```

Besides modifying the *id* attributes, you also need to connect a click event to the upload button within the template. Below is the code snippet from the cloneProgressBarTempate() method that connects the click event on the newly created button to the uploadFileInfo() method in the uploadController closure.

```
$(elem).find("#fileUploadButton_" + index).on("click",
   function () {
     uploadController.uploadFileInfo(index);
   }
);
```

uploadFileInfo Method

After the user fills in the title and description information, they click on the Upload button. That button calls the uploadFileInfo() method and passes in the index number associated with that button. Create a *FormData* object and append the index number, the file upload object, the title, and description into this object. Once this *FormData* object is created, pass that object and the current index number to the postFile() method to be uploaded to the Web API method.

```
function uploadFileInfo(index) {
  // Create FormData to post to Web API
  var data = new FormData();
  data.append("fileIndex", fileObjects[index].index);
  data.append("fileUploadObject", fileObjects[index].uploadObject);
  data.append("fileTitle", $("#fileUploadTitle_" + index).val());
  data.append("fileDescription",
    $("#fileUploadDescription_" + index).val());

  // Post the form data
  postFile(data, index);
}
```

postFile() Method

The first thing the postFile() method does is to unhide the progress bar. It then disables the title and description input controls, and the upload button. An Ajax call is then made passing in the *FormData* object.

In the xhr function of the Ajax call you add an event listener so you can keep track of the file upload progress. When the Web API sends a status of how much of the file has been uploaded, this data is passed as argument *e* in the function of the event listener. Pass this *e* argument and the index to the updateProgressBar() method to update the correct progress element on the web page.

```
function postFile(data, index) {
  // Display progress bar
  $("#fileUploadProgressBar_" + index)
    .parent().removeClass("hidden");

  // Disable controls
  $("#fileUploadTitle_" + index).prop("disabled", true);
  $("#fileUploadDescription_" + index).prop("disabled", true);
  $("#fileUploadButton_" + index).prop("disabled", true);

  // Change text on button while uploading
  $("#fileUploadButton_" + index).text("Uploading...");

  // Make the Ajax call
  $.ajax({
    url: UPLOAD_URL,
    type: 'POST',
    data: data,
    contentType: false,
    processData: false,
    xhr: function () {
      var req = $.ajaxSettings.xhr();
      if (req.upload) {
        // Setup event listener for progress returned from Web API
        req.upload.addEventListener('progress', function (e) {
          if (e.lengthComputable) {
            // Update the progress bar
            updateProgressBar(e, index);
          }
        }, false);
      }
      return req;
    },
  }).done(function (response) {
    console.log(response);
  }).fail(function (error) {
    console.log(error);
  });
}
```

updateProgressBar() Method

The updateProgressBar() method calculates the percentage of the data that has been uploaded to the server. It uses this percentage value to set the value of the progress bar element. This forces the browser to update the progress bar. Once the percentage is greater than or equal to 100, set the progress bar value to 100 to ensure it shows the file has been completely uploaded. Another label created in the HTML template is made visible to display the text "File Uploaded Successfully".

```
function updateProgressBar(e, index) {
  // Calculate current percentage
  var percentage = (e.loaded * 100) / e.total;

  // Modify the progress bar
  $("#fileUploadProgressBar_" + index).val(percentage);

  // Are we done?
  if (percentage >= 100) {
    // Display completion
    $("#fileUploadProgressBar_" + index).val(100);
    $("#fileUploadProgressComplete_" + index).removeClass("hidden");
    $("#fileUploadButton_" + index).text("Uploaded");
  }
}
```

The Web API Controller

All the client-side code is in place, so you can create your Web API method to upload your file now. If you are using Microsoft MVC, right mouse-click on the **Controllers** folder and choose **Add** | **Web API Controller Class (v2.1)** from the context-sensitive menu. Set the name to **FileUploadController** and click the OK button. At the top of the file, remove all the using statements, and just make sure you have the following four at the top.

```
Using System;
using System.IO;
using System.Web;
using System.Web.Http;
```

Remove all the methods within the FileUploadController class and add the following method signature.

```
public class FileUploadController : ApiController
{
  [HttpPost]
  public int UploadFile()
  {
  }
}
```

Get the File Object

To retrieve the fileUploadObject from the FormData object you created in the client-side code, use the Files[] property on the Request object as shown in the code below. This returns to you an object of the type HttpPostedFile. This object is the .NET equivalent of the file object created by the browser.

```
HttpPostedFile fileToUpload =
  HttpContext.Current.Request.Files["fileUploadObject"];
```

You can retrieve the rest of the FormData elements you created in JavaScript using the Form[] array.

```
string index = HttpContext.Current.Request.Form["fileIndex"];
string title = HttpContext.Current.Request.Form["fileTitle"];
string description =
  HttpContext.Current.Request.Form["fileDescription"];
```

Convert Input Stream into a Byte Array

The posted file contents are sent via an input stream, so you need to read that stream and store the data into an array of bytes. The easiest method to get the bytes of data is to copy into a MemoryStream object. Use the code shown below to request the posted file object and copy the data into a byte array.

```
byte[] contents;

if (fileToUpload != null && fileToUpload.ContentLength > 0)
{
  // Get the uploaded file contents
  using (MemoryStream ms = new MemoryStream())
  {
    fileToUpload.InputStream.CopyTo(ms);
    ms.Position = 0;
    contents = ms.ToArray();
  }
}
```

File Object Properties

Just as the file object on the client had properties such as length and file name, so does the HttpPostedFile object. It is useful to gather that data into an object. Instead of creating a real class, for now, just build an anonymous type and store the data from the HttpPostFile object into that anonymous type.

```
// Gather file information into anonymous type
var fileInfo = new
{
  ContentLength = fileToUpload.ContentLength,
  ContentType = fileToUpload.ContentType,
  FilePath = Path.GetDirectoryName(fileToUpload.FileName),
  FileName = Path.GetFileName(fileToUpload.FileName),
  Contents = contents
};
```

Once you have the contents of the file and the file name, store the file on your file server. You can also store the index number, the title, and the description in another file on your server. Ultimately, you would store this information in a table in a SQL Server, but I will leave that for you to accomplish. The complete UploadFile() method is shown in the code below.

```
[HttpPost]
public int UploadFile()
{
  byte[] contents;

  // Retrieve file to upload and the rest of the FormData object
  HttpPostedFile fileToUpload =
    HttpContext.Current.Request.Files["fileUploadObject"];
  string index = HttpContext.Current.Request.Form["fileIndex"];
  string title = HttpContext.Current.Request.Form["fileTitle"];
  string description =
    HttpContext.Current.Request.Form["fileDescription"];

  if (fileToUpload != null && fileToUpload.ContentLength > 0)
  {
    // Get the uploaded file contents
    using (MemoryStream ms = new MemoryStream())
    {
      fileToUpload.InputStream.CopyTo(ms);
      ms.Position = 0;
      contents = ms.ToArray();
    }

    // Gather file information into anonymous type
    var fileInfo = new
    {
      ContentLength = fileToUpload.ContentLength,
      ContentType = fileToUpload.ContentType,
      FilePath = Path.GetDirectoryName(fileToUpload.FileName),
      FileName = Path.GetFileName(fileToUpload.FileName),
      Contents = contents
    };

    // Write File to Server File System
    var file = HttpContext.Current.Server.MapPath("/UploadedFiles/"
      + fileInfo.FileName);
    File.WriteAllBytes(file, contents);

    // Write a description file
    File.WriteAllText(file + ".txt", "Index: " + index.ToString()
      + Environment.NewLine
      + "Title: '" + title + "'"
      + Environment.NewLine
      + "Description: '" + description + "'");
  }

  return 0;
}
```

Summary

In this blog post, you learned to upload multiple files using JavaScript, jQuery and Microsoft ASP.NET Web API. There is not a lot of code required to perform the actual upload, but adding extra input fields such as a title, description, and a progress bar for each file selected requires you to get a little creative. A template of HTML in a <script> tag is an ideal method to clone HTML for each file the user selects. Once cloned, rename the *id* attributes so each has a unique value. Each cloned HTML can be appended below the file input area so the user can add a title and a description to each file. Use the *FormData* object in JavaScript to build an object with the title and description input to send to your Web API call.

Source Code

NOTE: You can download the sample code for this article by visiting my website at http://www.pdsa.com/downloads. Select “Fairway/PDSA Blog”, then select “Upload Multiple Files Asynchronously - Part 2” from the dropdown list.