ASP.NET Multiple File Upload With Drag & Drop and Progress Bar Using HTML5

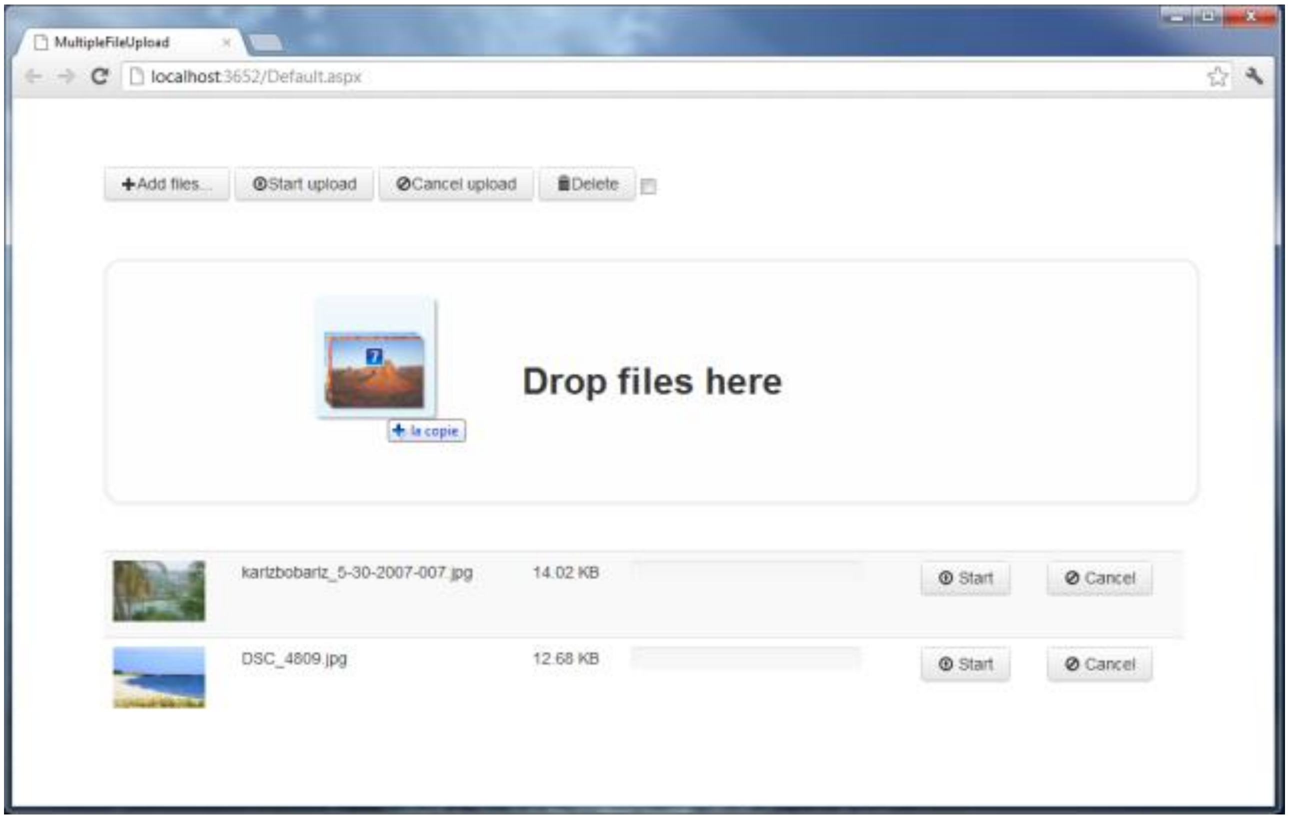
Akram El Assas, 29 Sep 2012 MIT

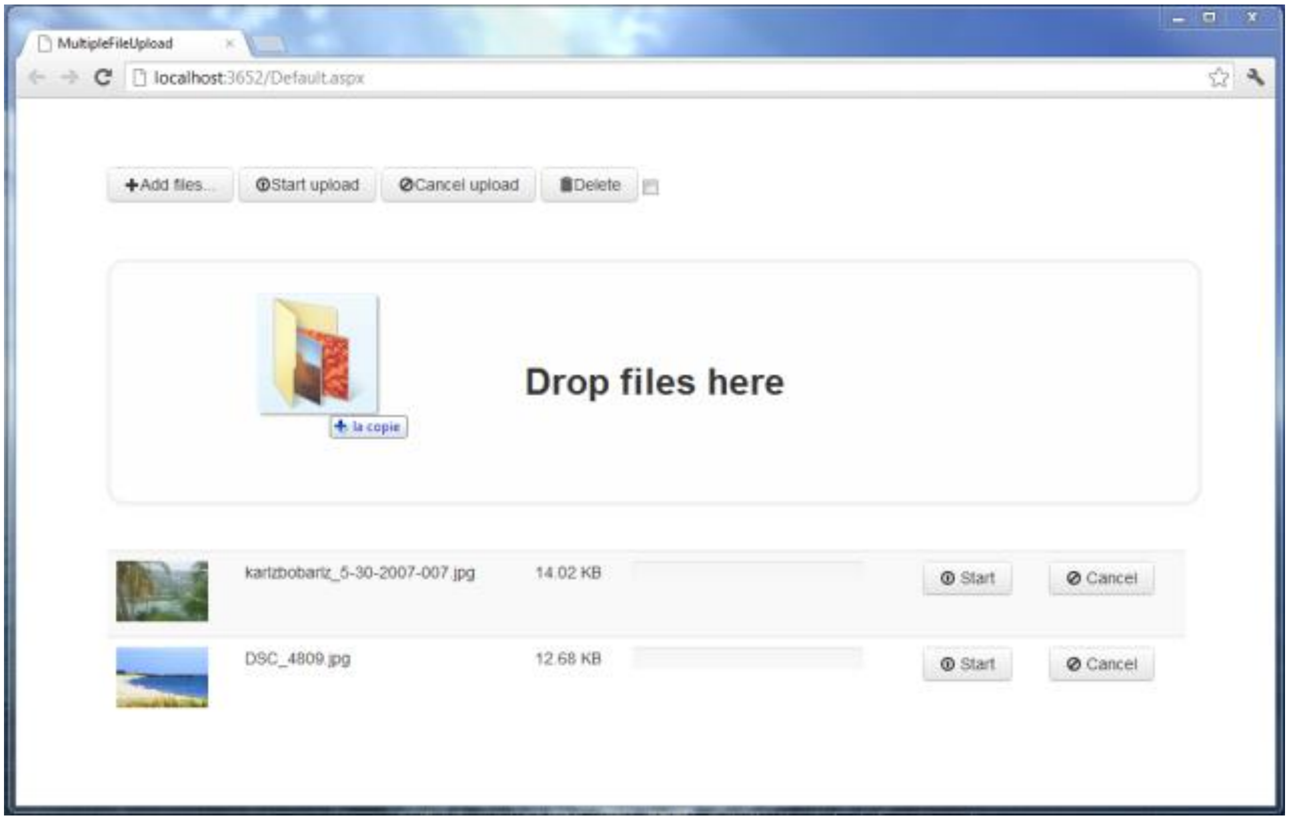
★★★★★ 4.97 (110 votes)

Rate this: ○ ○ ○ ○ ○

ASP .NET File upload widget with files and folders Drag & Drop, multiple file selection, progress, preview images, client-side image resizing and responsive layout. Supports cross-domain, chunked, resumable file uploads and automatic retries.
This is an old version of the currently published article.

- Download MultipleFileUpload.zip - 147.5 KB
- Download MultipleFileUpload.UserControl.VS2012.zip - 151.8 KB
- Download MultipleFileUpload.UserControl.VS2010.zip - 183.4 KB
- Download MultipleFileUpload.MVC4.zip - 8.9 MB




MultipleFileUpload
localhost:3652/Default.aspx
+Add files...
Start upload
Cancel upload
Delete
la copie
Drop files here
karlzbobarlz_5-30-2007-007.jpg
14.02 KB
Start
Cancel
DSC_4809.jpg
12.68 KB
Start
Cancel

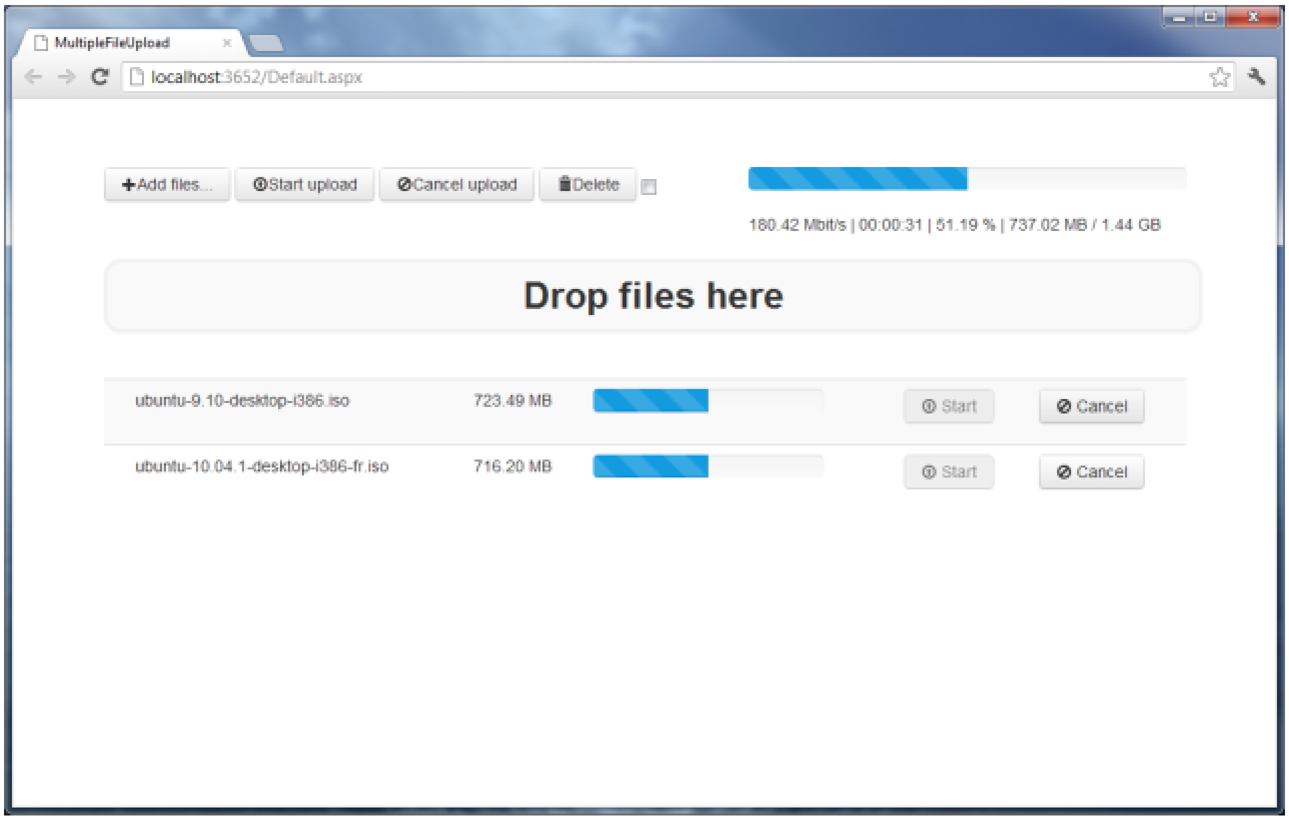

MultipleFileUpload
localhost:3652/Default.aspx
+Add files...
Start upload
Cancel upload
Delete
180.42 Mbit/s | 00:00:31 | 51.19 % | 737.02 MB / 1.44 GB
Drop files here
ubuntu-9.10-desktop-i386.iso
723.49 MB
Start
Cancel
ubuntu-10.04.1-desktop-i386-fr.iso
716.20 MB
Start
Cancel

Table of contents

Features

- **General:**

- Allows to select multiple files at once and upload them simultaneously or sequentially (defaults to simultaneously).
    - Allows to limit the maximum file size for uploads (defaults to unlimited).
    - Allows to limit the minimum file size for uploads (defaults to 1 byte).
    - Allows to set the maximum number of files that are allowed to be uploaded.
    - Allows to trigger uploads automatically after file selection (the files are queued by default)
    - Allows to add restrictions on file names via regular expressions.
    - Allows to set the allowed file types (defaults to any file type).
    - Allows to set the allowed file types for preview images.
    - Allows to set the maximum file size for preview images.
    - Allows to set the maximum width and the maximum height for preview images.
    - Allows to limit the number of concurent file uploads.
    - Allows to upload large files in smaller chunks with browsers supporting the Blob API.
    - Provides callBack methods for various upload events.
- **Drag & Drop support:**
    - Allows to upload files by dragging them from your desktop or filemanager and dropping them on your browser window.
    - Allows to upload folders by dragging them from your desktop or filemanager and dropping them on your browser window. This is currently only supported by Google Chrome.
- **Upload progress bar:**
    - Shows a progress bar indicating the upload progress for individual files.
    - Shows a progress bar indicating the upload progress for all uploads combined.
- **Cancelable uploads:**
    - Individual file uploads can be canceled to stop the upload progress.
    - All file uploads can be cancelled at the same time.
- **Resumable uploads:**
    - Aborted uploads can be resumed with browsers supporting the Blob API.
- **Automatic retries:**
    - Failed uploads can be resumed automatically with browsers supporting the Blob API.
- **Client-side image resizing:**
    - Images can be automatically resized on client-side with browsers supporting the required JS APIs.
- **Preview images:**
    - A preview of image files can be displayed before uploading with browsers supporting the required JS APIs.
- **No browser plugins required:**
    - The implementation is based on open standards like HTML5 and JavaScript and requires no additional browser plugins.
- **Graceful fallback for legacy browsers:**
    - Uploads files via `XMLHttpRequest`s if supported and uses iframes as fallback for legacy browsers.
- **HTML file upload form fallback:**

    - Shows a standard HTML file upload form if JavaScript is disabled.
- **Cross-site file uploads:**
    - Supports uploading files to a different domain with Cross-site `XMLHttpRequest`s.
- **Multipart and file contents stream uploads:**
    - Files can be uploaded as standard `multipart/form-data` or file contents stream (HTTP PUT file upload).

Browser support

- Google Chrome - 7.0+
- Apple Safari - 4.0+
- Mozilla Firefox - 3.0+
- Opera - 10.0+
- Microsoft Internet Explorer 6.0+

Drag & Drop is supported on Google Chrome, Firefox 4.0+, Safari 5.0+ and Opera 12.0+. Microsoft Internet Explorer 9 and lower have no support for multiple file selection or upload progress but allows to add multiple files to the upload queue by selecting files multiple times.

Took me some time to make it work properly under IE 10, but now Microsoft Internet Explorer 10 is fully supported. Multiple Drag & Drop, multiple file selection, upload progress, chunked uploads, resumable file uploads, automatic retries, preview images and other options are supported.

More detailed information about browser support can be found here.

Software you need to install

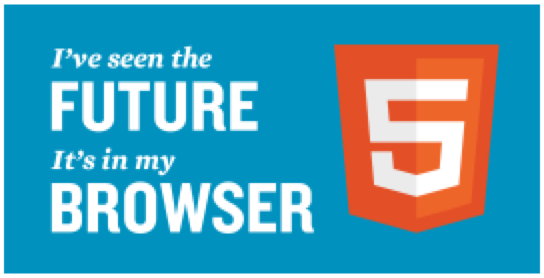
To run the applications, you'll need Visual Studio 2010/2012 or the free Visual Web Developer Express and and the ASP .NET MVC 4 Framework.

Architecture

The following section describes the architecture of the application.

Libraries

Bootstrap
Sleek, intuitive, and powerful front-end framework for faster and easier web development.



The applications relies on the following Open Source libraries:

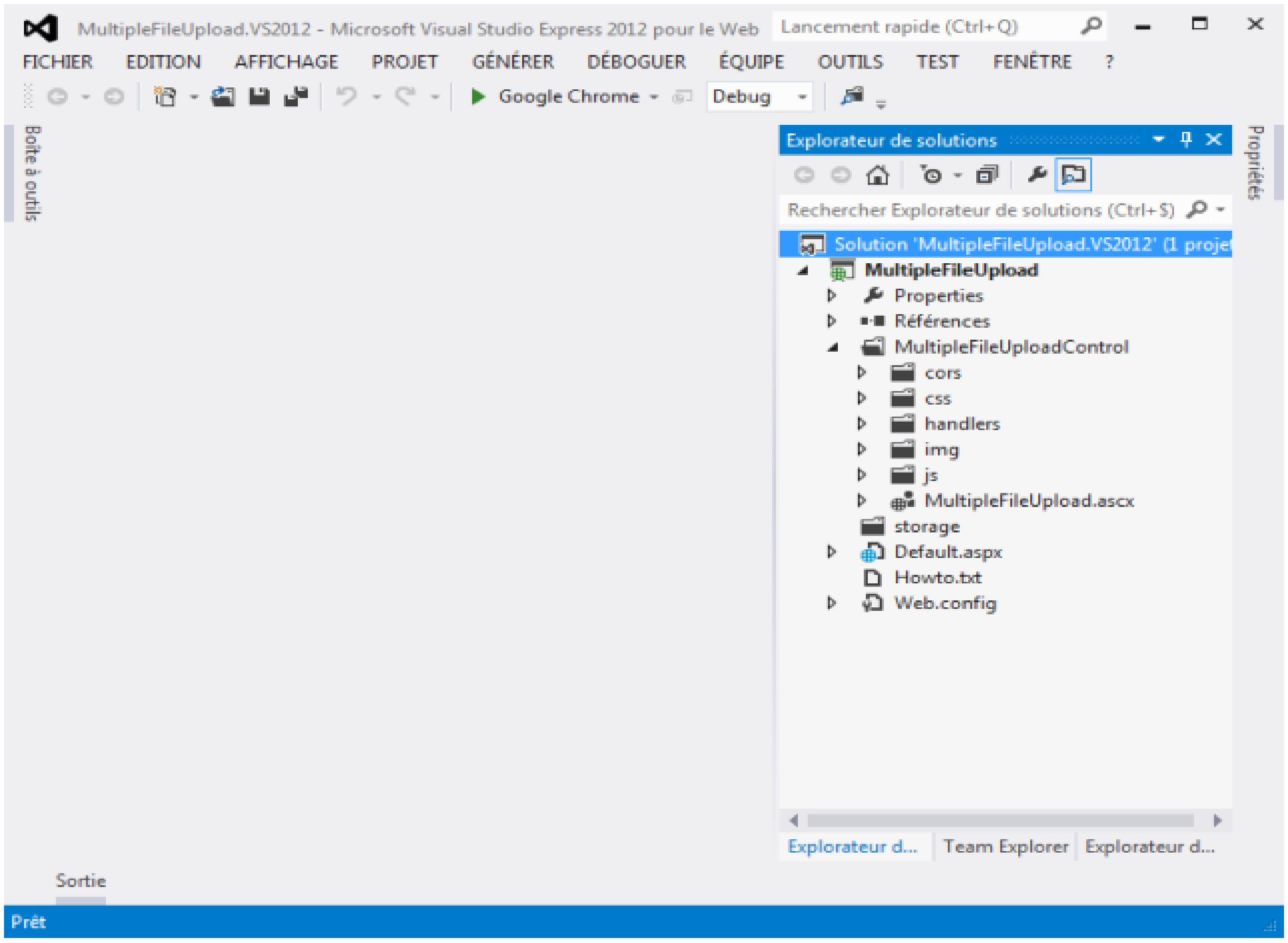
- jQuery 1.8.2
- jQuery File Upload 5.17.5
- Twitter's Bootstrap toolkit
- Twitter's Bootstrap responsive layout (supporting different screen sizes)
- Bootstrap Image Gallery
- jQuery Iframe Transport plugin
- JavaScript Templates engine
- JavaScript Load Image function
- JavaScript Canvas to Blob function
- html5shim
- ASP .NET Web API
- ASP .NET MVC 4 Framework

How does it work?

In this section is described the architecture of the MultipleFileUpload and MultipleFileUpload.UserControl. A detailed description of the MultipleFileUpload.UserControl is available in the following section and a detailed description of MultipleFileUpload.MVC4 is available in the following section.

In order to get in touch with the project, let's start explaining how the MultipleFileUpload application works. This will help a lot to understand the user control and the MVC 4 Web API versions.

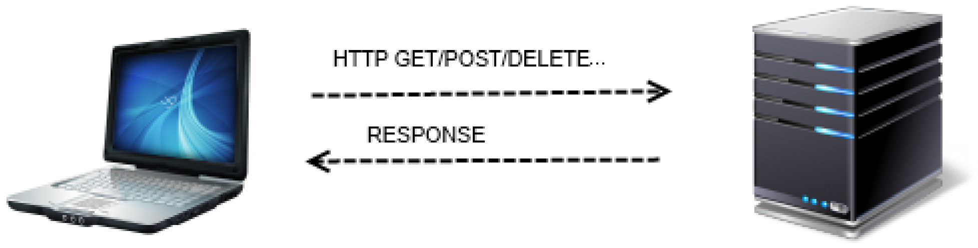
The figure below illustrates the architecture of the solution:


MultipleFileUpload.VS2012 - Microsoft Visual Studio Express 2012 pour le Web
Lancement rapide (Ctrl+Q)
FICHIER EDITION AFFICHAGE PROJET GÉNÉRER DÉBOGUER ÉQUIPE OUTILS TEST FENÊTRE ?
Google Chrome
Debug
Boîte à outils
Explorateur de solutions
Rechercher Explorateur de solutions (Ctrl+$)
Solution 'MultipleFileUpload.VS2012' (1 projet
MultipleFileUpload
Properties
Références
MultipleFileUploadControl
cors
css
handlers
img
js
MultipleFileUpload.ascx
storage
Default.aspx
Howto.txt
Web.config
Explorateur d...
Team Explorer
Explorateur d...
Propriétés
Sortie
Prêt

The `js` folder contains the JavaScript libraries listed in the previous section, the `css` folder contains the Twitter's Bootstrap toolkit and the stylesheets, the `img` folder contains the images used by the application, the `cors` folder contains HTML pages used for cross-domain access via redirect option, the `handlers` folder contains ASP .NET HTTP Handlers that are described further and finally the `storage` folder is the server folder where the uploaded files are stored (configurable).

The jQuery File Upload plugin uses jQuery.ajax() for the file upload requests. The options set for the File Upload plugin are passed to jQuery.ajax(). The jQuery file upload plugin sends HTTP requests to a server, the URL to which requests are sent can be defined via the `url` option of the plugin. If this option is not set, it defaults to the `action` property of the file upload form if available, otherwise the requests are sent to the URL of the current page. The HTTP request method for file uploads can be set to `POST` or `PUT` via the `type` option and it defaults to `POST`.

The figure below illustrates the communication between the client and the server:



The HTTP requests are handled on the server side through ASP .NET HTTP handlers. There are three HTTP handlers listed below:

- `Upload.ashx`: Main HTTP handler that handles File upload, File download (when the file is uploaded on the server, it can be downloaded), File deletion and files listing.
- `FileComplete.ashx`: Sends a thumbnail to the client saying that the file upload is completed with success.
- `FileError.ashx`: Sends a thumbnail to the client saying that the file upload is failed.

The `Upload.ashx` handler is based on IHttpHandler example from Iain Ballard. Some enhancements and modifications have been made on the `Upload.ashx` handler:

- Error handling added
- Error messages sent in JSON format
- Failure thumbnail shown on the client in case of errors
- IE file name retrieval fixed
- Chunked uploads added
- Resumable file uploads added
- Automatic retries added

When a request is sent to the `Upload.ashx` handler, It is handled through the `DoWork` method:

```
public void ProcessRequest(HttpContext context)
{
    context.Response.AddHeader("Pragma", "no-cache");
    context.Response.AddHeader("Cache-Control", "private, no-cache");
    DoWork(context);
}
```

The `DoWork` method does the main job relying on the HTTP method:

```
private void DoWork(HttpContext context)
{
    switch (context.Request.HttpMethod)
    {
        case "HEAD":
        case "GET":
            if (GivenFilename(context)) DeliverFile(context);
            else ListCurrentFiles(context);
            break;
        case "POST":
        case "PUT":
            UploadFile(context);
            break;
        case "DELETE":
            DeleteFile(context);
            break;
        case "OPTIONS":
            ReturnOptions(context);
            break;
        default:
            context.Response.ClearHeaders();
            context.Response.StatusCode = 405;
            break;
    }
}
```

File upload

In case of a file upload, the HTTP method is either `POST` or `PUT`. When the files or folders are selected the jQuery file upload library loads the file names and preview images if necessary on the client side. The display is done through the following template:

```
<!-- The template to display files available for upload -->
<script id="template-upload" type="text/x-tmpl">
{% for (var i=0, file; file=o.files[i]; i++) { %}
    <tr class="template-upload fade">
        <td class="preview"><span class="fade"></span></td>
        <td class="name"><span>{%=file.name%}</span></td>
        <td class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
        {% if (file.error) { %}
            <td class="error" colspan="2"><span class="label label-
important">{%=locale.fileupload.error%}</span>
{%=locale.fileupload.errors[file.error] || file.error%}</td>
        {% } else if (o.files.valid && !i) { %}
            <td>
                <div class="progress progress-striped active"
role="progressbar" aria-valuemin="0" aria-valuemax="100" aria-
valuenow="0"><div class="bar" style="width:0%;"></div></div>
            </td>
            <td class="start">{% if (!o.options.autoUpload) { %}
                <button class="btn btn-action">
                    <i class="icon-upload"></i>
                    <span>{%=locale.fileupload.start%}</span>
                </button>
            {% } %}</td>
        {% } else { %}
            <td colspan="2"></td>
        {% } %}
        <td class="cancel">{% if (!i) { %}
            <button class="btn btn-action">
                <i class="icon-ban-circle"></i>
                <span>{%=locale.fileupload.cancel%}</span>
            </button>
        {% } %}</td>
    </tr>
{% } %}
</script>
```

The jQuery file upload library relies on the Javascript templates Library which is a lightweight, fast and powerful JavaScript templating engine with zero dependencies. This library is developed and maintained by Sebastian Tschan, the author of the jQuery file upload library.

Of course this template can be customized.

When the upload request is received by the server, the server checks whether it's an upload of an entire file whether it's an upload of a partial file. Once the upload is finished the status is sent in JSON format to the client in order to display information of the upload.

The file status exposes these informations to the client:

```
public string group { get; set; }
public string name { get; set; }
public string type { get; set; }
public int size { get; set; }
public string progress { get; set; }
public string url { get; set; }
public string complete_url { get; set; }
public string error_url { get; set; }
public string delete_url { get; set; }
public string delete_type { get; set; }
public string error { get; set; }
```

The upload is handled throught the `UploadFile` method as follows:

```
private void UploadFile(HttpContext context)
{
    List<FileStatus> statuses = new List<FileStatus>();

    try
    {
        var headers = context.Request.Headers;

        if (string.IsNullOrEmpty(headers["X-File-Name"]))
        {
            UploadWholeFile(context, statuses);
        }
        else
        {
            UploadPartialFile(headers["X-File-Name"], context, statuses);
        }
    }
    finally
    {
        WriteJsonIframeSafe(context, statuses);
    }
}
```

The `WriteJsonIframeSafe` method sends the status to the client in JSON format. The JSON serialization is done through JavaScriptSerializer. The JSON object sent is then used in the template that displays files available for download:

```
<!-- The template to display files available for download -->
<script id="template-download" type="text/x-tmpl">
{% for (var i=0, file; file=o.files[i]; i++) { %}
    <tr class="template-download fade">
        {% if (file.error) { %}
            <td class="preview">
                <img src="{%=file.error_url%}">
            </td>
            <td class="name"><span>{%=file.name%}</span></td>
            <td
class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
            <td class="error" colspan="2"><span class="label label-
important">{%=locale.fileupload.error%}</span>
{%=locale.fileupload.errors[file.error] || file.error%}</td>
        {% } else { %}
            <td class="preview">{% if (file.complete_url) { %}
                <a href="{%=file.url%}" title="{%=file.name%}" rel="gallery"
download="{%=file.name%}"><img src="{%=file.complete_url%}"></a>
            {% } %}</td>
            <td class="name">
                <a href="{%=file.url%}" title="{%=file.name%}"
rel="{%=file.thumbnail_url&&'gallery'%}"
download="{%=file.name%}">{%=file.name%}</a>
            </td>
            <td
class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
            <td colspan="2"></td>
        {% } %}
        <td class="delete">
            <button class="btn btn-action" data-type="{%=file.delete_type%}"
data-url="{%=file.delete_url%}">
                <i class="icon-trash"></i>
                <span>{%=locale.fileupload.destroy%}</span>
            </button>
            <input type="checkbox" name="delete" value="1">
        </td>
    </tr>
{% } %}
</script>
```

Of course this template can be customized.

File download

Once the file uploaded, It is possible to download it by clicking on it. You can of course disable this option if you want to. This is handled through the DeliverFile method:

```
private void DeliverFile(HttpContext context)
{
    string filename = context.Request["f"];
    string filePath = Path.Combine(StorageFolder, filename);

    if (File.Exists(filePath))
    {
        context.Response.AddHeader("Content-Disposition", "attachment;
filename=\"" + filename + "\"");
        context.Response.ContentType = "application/octet-stream";
        context.Response.ClearContent();
        context.Response.WriteFile(filePath);
    }
    else
    {
        context.Response.StatusCode = 404;
    }
}
```

File deletion

Once the file is uploaded, It can be deleted from the server. When the end-user clicks on the delete button, an HTTP DELETE request is sent to the server and the `DeleteFile` method runs:

```
private void DeleteFile(HttpContext context)
{
    var filePath = Path.Combine(StorageFolder, context.Request["f"]);
    if (File.Exists(filePath))
    {
        File.Delete(filePath);
    }
}
```

Files list

The files uploaded are listed on the client. This is done when the page is loaded:

```
//
// Load existing files
//
$('#fileupload').each(function () {
    var that = this;
    $.getJSON(this.action, function (result) {
        if (result && result.length) {
            $(that).fileupload('option', 'done')
                   .call(that, null, { result: result });
        }
    });
});
```

When the page is loaded, an HTTP GET request is sent to the server in order to retreive the files. This is handled through the `ListCurrentFiles` method:

```
private void ListCurrentFiles(HttpContext context)
{
    FileStatus[] statuses =
        (from file in new DirectoryInfo(StorageFolder).GetFiles("*",
SearchOption.TopDirectoryOnly)
            select new FileStatus(file)).ToArray();
    string jsonObj = _javaScriptSerializer.Serialize(statuses);
    context.Response.AddHeader("Content-Disposition", "inline;
filename=\"files.json\"");
    context.Response.ContentType = "application/json";
    context.Response.Write(jsonObj);
}
```

Callbacks

The jQuery File Upload library provides many options and callbacks. These options are described in the following page. Below the list of callbacks provided:

```
//
// Callbacks (if you want to use them refer to the documentation)
//
$('#fileupload')
.bind('fileuploadadd', function (e, data) { /* ... */ })
.bind('fileuploadsubmit', function (e, data) { /* ... */ })
.bind('fileuploadsend', function (e, data) { /* ... */ })
.bind('fileuploaddone', function (e, data) { /* ... */ })
.bind('fileuploadfail', function (e, data) { /* ... */ })
.bind('fileuploadalways', function (e, data) { /* ... */ })
.bind('fileuploadprogress', function (e, data) { /* ... */ })
.bind('fileuploadprogressall', function (e, data) { /* ... */ })
.bind('fileuploadstart', function (e) { /* ... */ })
.bind('fileuploadstop', function (e) { /* ... */ })
.bind('fileuploadchange', function (e, data) { /* ... */ })
.bind('fileuploadpaste', function (e, data) { /* ... */ })
.bind('fileuploaddrop', function (e, data) { /* ... */ })
.bind('fileuploaddragover', function (e) { /* ... */ });
```

User control

I had a comment in the forum talking about making an ASP .NET custom control (redistributable assembly that is, a DLL) which can be referenced and used in ASP .NET applications. In ASP .NET, there are two ways of making server controls. The first way called "Custom control" consists of writing a control from scratch using code (C#/VB). In this case, the control inherits directly or indirectly from the `Control` class and the logic of the control including its child controls, pre-rendering, redering, ect... is implemented on top of it. Once the control written, it can be buit in a dedicated redistributable assembly and used in any other ASP .NET application just by referencing the assembly in the ASP .NET application and using it in the application. This approach requires a lot of development and design especially in the case of the MultipleFileUpload.

On the other hand, the second way called "User control" consists of writing a control in an `*.ascx` file which looks like an `*.aspx`. With this approach you can simply drag and drop your controls in the design view and you don't need to implement the control from scratch or implement the pre-rendering, rendering, etc.. methods. And of course writing a "User control" takes less time and energy than writing a "Custom control".

Thus, I created a "User Control" `MultipleFileUpload`. I'll maybe write a "Custom control" with options, server side and client side events in a future update (really depends on my free time).

In the following section, the set up of the "User control" is described in details.

A "User control" named `MultipleFileUpload.ascx` has been added in the solution and the `StorageFolder` setting has been added as a property of the "User control". Below the code-behind of the "User control":

```
namespace MultipleFileUpload
{
    public partial class MultipleFileUpload : System.Web.UI.UserControl
    {
        public string StorageFolder
        {
            get
            {
                return (string)(ViewState["StorageFolder"] ?? string.Empty);
            }
            set
            {
                ViewState["StorageFolder"] = value;
            }
        }

    }
}
```

The use of this option in the jQuery File Upload plugin will be described later.

On the client side, all the view, scripts and templates have been moved to `MultipleFileUpload.ascx` file. A change has been made on the view, the requests are no longer sent through the action of the `multipart/form-data` form. The form has been replaced by a`div` :

```
<div id="fileupload"><!-- view --></div>
```

and the `url` of the `Upload.ashx` HTTP handler is passed to the jQuery file Upload plugin in order to send the HTTP requests to the server. The `StorageFolder` is also passed as a query string to the `Upload.ashx` HTTP handler as follows:

```
var handler = 'handlers/Upload.ashx?storageFolder=' + '<%=
Server.UrlEncode(StorageFolder)  %>';

//
// Initialize the jQuery File Upload widget
//
$('#fileupload').fileupload({
    url: handler,
    dropZone: $('#dropzone'),
    sequentialUploads: true
});

//
// Load existing files
//
$('#fileupload').each(function () {
    var that = this;
    $.getJSON(handler, function (result) {
        if (result && result.length) {
            $(that).fileupload('option', 'done')
                .call(that, null, { result: result });
        }
    });
});
```

As you can see the `StorageFolder` is encoded and sent as a query string to the `Upload.ashx` handler. Thus, the `Upload.ashx` handler has been changed in order to retrieve the `StorageFolder` from the query strings:

```
public string StorageFolder
{
    get
    {
        return HttpContext.Current.Request.QueryString["storageFolder"];
    }
}
```

Note that there is no need to decode the query string. It's done by default when it's retrieved.

And the `StorageFolder` has been added in the `FileStatus` class:

```
public string storageFolder { get; set; }
```

In order to send it as a query string to the `url` parameter and the `delete` action:

```
private void SetValues(string fileName, int fileLength, string storageFolder)
{
    name = fileName;
    type = "image/png";
    size = fileLength;
    progress = "1.0";
    url = HandlerPath + "Upload.ashx?f=" + fileName + "&storageFolder=" +
storageFolder;
    complete_url = HandlerPath + "FileComplete.ashx?f=" + fileName;
    error_url = HandlerPath + "FileError.ashx?f=" + fileName;
    delete_url = HandlerPath + "Upload.ashx?f=" + fileName +
"&storageFolder=" + storageFolder;
    delete_type = "DELETE";
}
```

At this step our "User control" is finished.

I added the `StorageFolder` in order to illustrate how to add parameters to the "User Control" and pass them to the jQuery File Upload plugin. The same approach can be used to add new options.

Now, we can register our brand new "User control" in the `Default.aspx` page:

```
<%@ Register TagPrefix="jQuery" TagName="MultipleFileUpload"
Src="~/MultipleFileUpload.ascx" %>
```

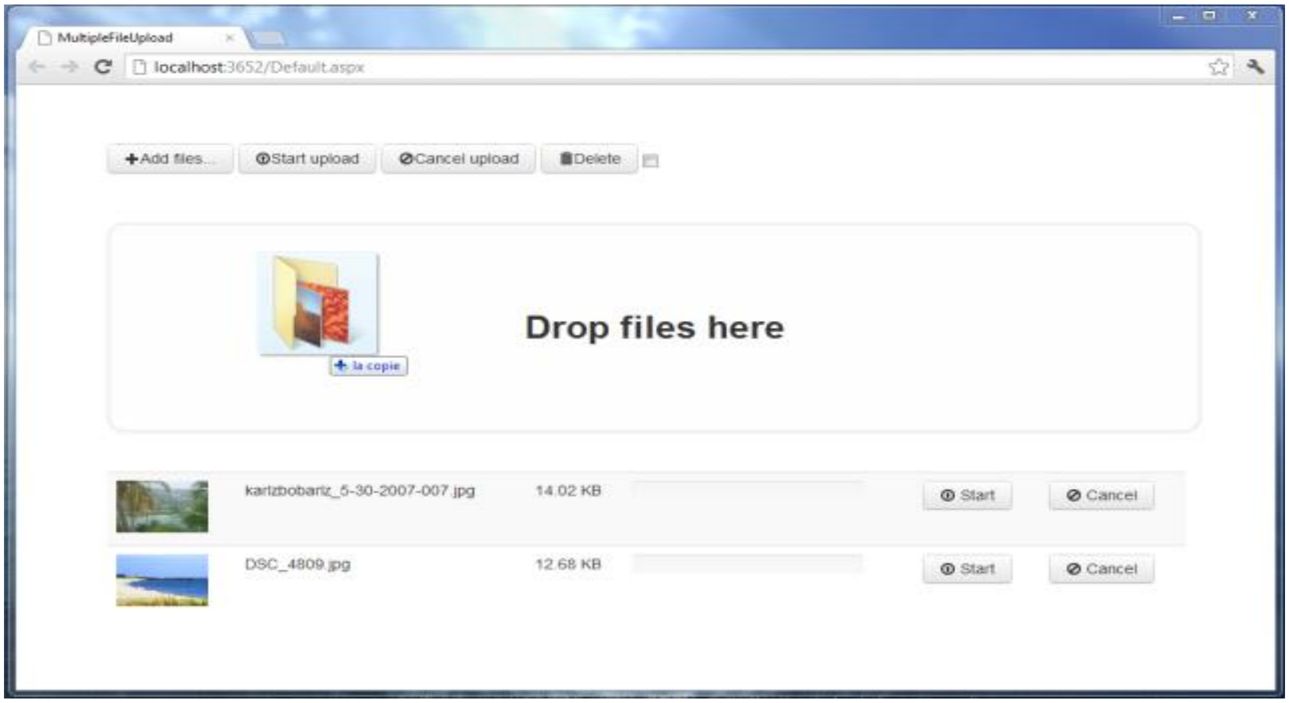
Install it in our form:

```
<form id="Form1" runat="server">
    <jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server" />
</form>
```

And set the `StorageFolder` option in the code-behind of the `Default.aspx` page:

```
protected void Page_Load(object sender, EventArgs e)
{
    //
    // Set the storage folder of the control
    // Other options and events could be added for sure
    //
    if (!IsPostBack)
    {
        MultipleFileUpload.StorageFolder =
MapPath(ConfigurationManager.AppSettings["StorageFolder"]);
    }
}
```

Now, we're done. We can use our brand new "User control":



The advantage of the "User control" approach is that we can use it on different pages of the application and only one `*.ascx` file needs to be maintained.

Options

The user control brings the following options:

- `StorageFolder`: The full path of the storage folder (ex. C:\Storage). Make sure that the pool identity has the permissions to read, write and delete on this folder.
- `AcceptFileTypes`: Allowed file types. It defaults to any file type and accepts a list of pipe-separated file extensions (ex. jpg|gif|png).
- `EnableChunkedUploads`: Enables Chunked Uploads for browsers supporting the Blob API. Defaults to False. Even if the option is True and the client Browser is not supporting Chunked Uploads, Uploads works normally.
- `MaxChunkSize`: Preferred maximum chunk size in Bytes. Defaults to 10 MB.
- `SequentialUploads`: Enables Sequential uploads. Defaults to simultaneous uploads.
- `Resume`: Enables resuming Uploads for browsers supporting the Blob API. Defaults to False. Chunked uploads needs to be enabled to enable this option.
- `AutoRetry`: Enables automatic resuming Uploads for browsers supporting the Blob API in case of upload failures. Defaults to False. Resumable file uploads needs to be enabled to enable this option.
- `MaxRetries`: Maximum retries. Defaults to 100 retries.

- `RetryTimeout`: Retry time out in milliseconds, before the file upload is resumed. Defaults to 500 milliseconds.
- `LimitConcurrentUploads`: To limit the number of concurrent uploads, set this option to an integer value greater than 0. Defaults to undefined.
- `ForceIframeTransport`: Set this option to True to force iframe transport uploads, even if the browser is capable of XmlHttpRequest file uploads. This can be useful for cross-site file uploads, if the Access-Control-Allow-Origin header cannot be set for the server-side upload handler which is required for cross-site XmlHttpRequest file uploads. Defaults to False.
- `AutoUpload`: By default, files added to the UI are uploaded as soon as the end user clicks on the start buttons. To enable automatic uploads, set this option to True. Defauls to False. This option is currently not supported on Microsoft Internet Explorer 10.
- `MaxNumberOfFiles`: This option limits the number of files that are allowed to be uploaded. By default, unlimited file uploads are allowed.
- `MaxFileSize`: The maximum allowed file size in bytes, by default unlimited.
- `MinFileSize`: The minimum allowed file size, by default 1 byte.
- `PreviewAsCanvas`: By default, preview images are displayed as canvas elements if supported by the browser. Set this option to false to always display preview images as img elements. Defaults to True. This option is currently not supported on Microsoft Internet Explorer 10.
-

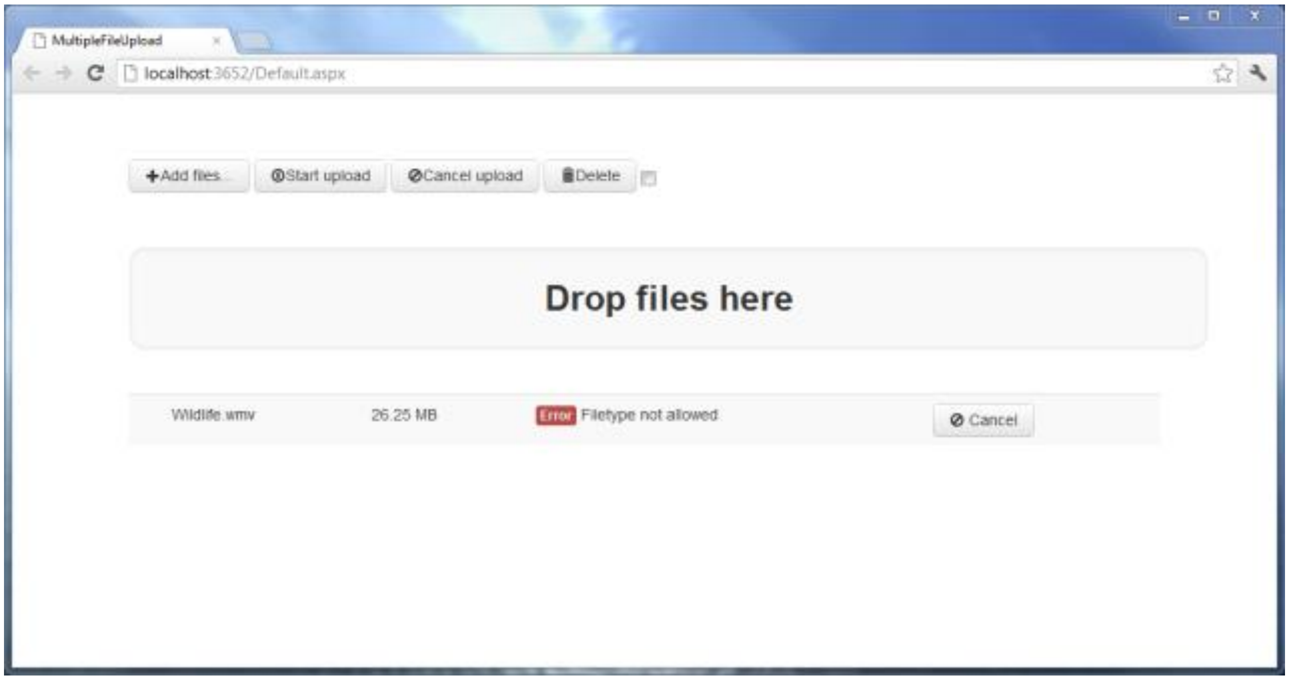
In the section below is described how the `AcceptFileTypes` option has been added. The other options were added almost the same way.

AcceptFileTypes

This option lets you set the allowed file types for file uploads. It defaults to any file type and accepts a list of pipe-separated file extensions as follows:

```
"jpg|gif|png"
```

If the file selected by the end user doesn't have the allowed file type the file is rejected by the user control. For example, If the allowed file types are the ones listed above, the user control will reject his file as follows:



To add this option to the user control, a new property `AcceptFileTypes` has been added as follows:

```
public string AcceptFileTypes
{
    get
    {
        return (string)(ViewState["AcceptFileTypes"] ?? ".*");
    }
    set
    {
        ViewState["AcceptFileTypes"] = value;
    }
}
```

The property defaults to any file type.

Then, the property is retrieved in the jQuery code as a Regular Expression as follows:

```
var acceptFileTypes = new RegExp('(\.|\/)(' + '<%= AcceptFileTypes %>' + ')',
'i');
```

And passed to the jQuery File Upload plugin as follows:

```
 $('#fileupload').fileupload({
    url: handler,
    dropZone: $('#dropzone'),
    sequentialUploads: true,
    acceptFileTypes: acceptFileTypes
});
```

That's it!

The option can be set up in the ASP code as follows:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
AcceptFileTypes="jpg|gif|png" />
```

Or in the code behind as follows:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        MultipleFileUpload.StorageFolder =
MapPath(ConfigurationManager.AppSettings["StorageFolder"]);
        MultipleFileUpload.AcceptFileTypes = "jpg|gif|png";
    }
}
```

This option is set to its default value (any file type) in the application available in downloads.

Events

The user control brings the following client side events:

- `OnFileUploadAdd`: This event fires when the file is added to the upload queue.
- `OnFileUploadSubmit`: This event fires when the file is submitted. If this event returns False, the file upload request is not started.
- `OnFileUploadSend`: This event fires at the start of file upload request. If this event returns False, the file upload request is aborted.
- `OnFileUploadDone`: This event fires when the file upload is completed successfuly.
- `OnFileUploadFail`: This event fires in case of failed (abort or error) uploads.
- `OnFileUploadAlways`: This event fires in case of completed (success, abort or error) uploads.
- `OnFileUploadProgress`: This event fires for upload progress of a file.
- `OnFileUploadProgressAll`: This event fires for global upload progress.
- `OnFileUploadStart`: This event fires when upload starts.
- `OnFileUploadStop`: This event fires when upload stops.
- `OnFileUploadChange`: This event fires when the file input collection changes.
- `OnFileUploadPaste`: This event fires for paste events to the dropZone collection.
- `OnFileUploadDrop`: This event fires for drop events of the dropZone collection.

- ⚡`OnFileUploadDragOver`: This event fires for dragover events of the dropZone collection.
- ⚡`OnFileUploadDestroy`: This event fires before the file is deleted.
- ⚡`OnFileUploadDestroyed`: This event fires after the file is deleted, the transition effects have completed and the download template has been removed.
- ⚡`OnFileUploadAdded`: This event fires after the file has been added to the list, the upload template has been rendered and the transition effects have completed.
- ⚡`OnFileUploadSent`: This event fires after the send event and the file is about to be sent.
- ⚡`OnFileUploadCompleted`: This event fires after successful uploads after the download template has been rendered and the transition effects have completed.
- ⚡`OnFileUploadFailed`: This event fires after failed uploads after the download template has been rendered and the transition effects have completed.
- ⚡`OnFileUploadStarted`: This event fires after the start event has run and the transition effects called in the start event have completed.
- ⚡`OnFileUploadStopped`: This event fires after the stop event has run and the transition effects called in the stop event have completed.

In the section below is described how the `OnFileUploadDone` event has been added. The other events were added pretty much the same way.

OnFileUploadDone

In order to add the `OnFileUploadDone` event, the following property has been added to the user control in order to encapsulate the name of the JavaScript function that is fired for event as follows:

```
public string OnFileUploadDone
{
    get
    {
        return (string)(ViewState["OnFileUploadDone"] ?? string.Empty);
    }
    set
    {
        ViewState["OnFileUploadDone"] = value;
    }
}
```

Then the JavaScript function name is retrieved in the jQuery code as follows:

```
fileuploaddone = '<%= OnFileUploadDone %>'
```

Then, the event is registerd in the jQuery code as follows:

```
$('#fileupload')
    .bind('fileuploaddone', function (e, data) {
        typeof window[fileuploaddone] == 'function' &&
window[fileuploaddone](e, data);
});
```

First, we check whether the JavaScript function exists in the current context by using the `typeof` operator, then if it exists, it is fired.

That's it!

To register the event, It can be done as follows in the ASP page:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
OnFileUploadDone="fileuploaddone" />
<script type="text/javascript">
    function fileuploaddone(e, data) {
        var file = data.files[0];
        alert(file.name + ' uploaded with success (' + file.type + ' ' +
file.size + ' bytes).');
        return false;
    }
</script>
```

An in depth look in IIS

In this section is described how IIS handles the file uploads, how to configure IIS to upload large files with less memory usage and finally a detailed use case is described through screen-shots.

Before to start talking about how IIS is handling the file uploads, I'll talk about some enhancements I've made in the application.

If you take the application provided in the MultipleFileUpload.zip and deploy it in IIS7, you'll notice that the uploads work but when you'll try to delete a file, you'll get an error saying that the "DELETE" HTTP method is not allowed. You can see this error through the Chrome Dev toolbar or Fiddler.

Well, to fix this issue I installed all the HTTP handlers in a clean way in the Web.config of the `MultipleFileUpload.UserControl` application by referencing them in the `system.web` and `system.webServer` sections as follows:

```
<system.web>
  <httpHandlers>
    <!-- MultipleFileUpload handlers -->
    <add verb="*" path="Upload.axd" type="MultipleFileUpload.handlers.Upload,
MultipleFileUpload" />
    <add verb="*" path="FileComplete.axd"
type="MultipleFileUpload.handlers.FileComplete, MultipleFileUpload" />
    <add verb="*" path="FileError.axd"
type="MultipleFileUpload.handlers.FileError, MultipleFileUpload" />
```

```
  </httpHandlers>
</system.web>
<system.webServer>
  <validation validateIntegratedModeConfiguration="false"/>
  <handlers>
    <!-- MultipleFileUpload handlers -->
    <add name="Upload" verb="*" path="Upload.axd"
type="MultipleFileUpload.handlers.Upload, MultipleFileUpload" />
    <add name="FileComplete" verb="*" path="FileComplete.axd"
type="MultipleFileUpload.handlers.FileComplete, MultipleFileUpload" />
    <add name="FileError" verb="*" path="FileError.axd"
type="MultipleFileUpload.handlers.FileError, MultipleFileUpload" />
  </handlers>
</system.webServer>
```

Thus, I modified the URL of the `Upload.axd` HTTP handler in the jQuery code of the user control as follows:

```
var handler = 'Upload.axd?storageFolder=' + '<%=
Server.UrlEncode(StorageFolder)  %>';
```

And in the `FileStatus` class:

```
private void SetValues(string fileName, int fileLength, string storageFolder)
{
    name = fileName;
    type = "image/png";
    size = fileLength;
    progress = "1.0";
    url = "Upload.axd?f=" + fileName + "&storageFolder=" + storageFolder;
    complete_url = "FileComplete.axd?f=" + fileName;
    error_url = "FileError.axd?f=" + fileName;
    delete_url = "Upload.axd?f=" + fileName + "&storageFolder=" +
storageFolder;
    delete_type = "DELETE";
}
```

You'll notice that I removed the `HandlerPath` field. Indeed, now it's not necessary anymore. The `Upload.axd` acts as a shortcut.

That's the only things that changed to make this work properly under IIS7. I also added two new setup projects. One for deploying the application in 32 bits and another in 64 bits.

Theory

The file upload is handled through the HttpPostedFile class. When uploads are launched by the client, a collection of `HttpPostedFile` becomes available in the HTTP request. Each file is retrieved and saved on the storage folder. This is done through the `HttpPostedFile.SaveAs` method for entire uploads an through `FileStream` in case of partial uploads.

In the Web.config or the Machine.config files there is a section that controls the behavior of the HTTP requests. This section is `httpRuntime`. In this section, there are three important parameters that control the behavior of the uploads. These parameters are listed below:

- `requestLengthDiskThreshold`: represents at what point the threshold that the request will start to be buffered in the disk. Defaults to 80 KB for .NET 4.0.
- `maxRequestLength`: represents the overall total length the HTTP request cannot exceed. Defaults to 4096 KB for .NET 4.0.
- `executionTimeout`: The maximum number of seconds that a request is allowed to execute before being automatically shut down by ASP.NET. This time-out applies only if the debug attribute in the compilation element is `False`. Defaults to 110 seconds for .NET 4.0.

`requestLengthDiskThreshold` should always be lower than `maxRequestLength`. Let's take the default values (.NET 4.0) and explain how this is working in IIS:

```
<httpRuntime maxRequestLength="4096" requestLengthDiskThreshold="80" />
```

In the example obove, the request cannot exceed `4096 KB`. Once the request crosses `80 KB`, the request will start to buffer to disk and thus the request will not stay in memory. If you increase the `requestLengthDiskThreshold`, the w3wp.exe processus will use a lot of memory.

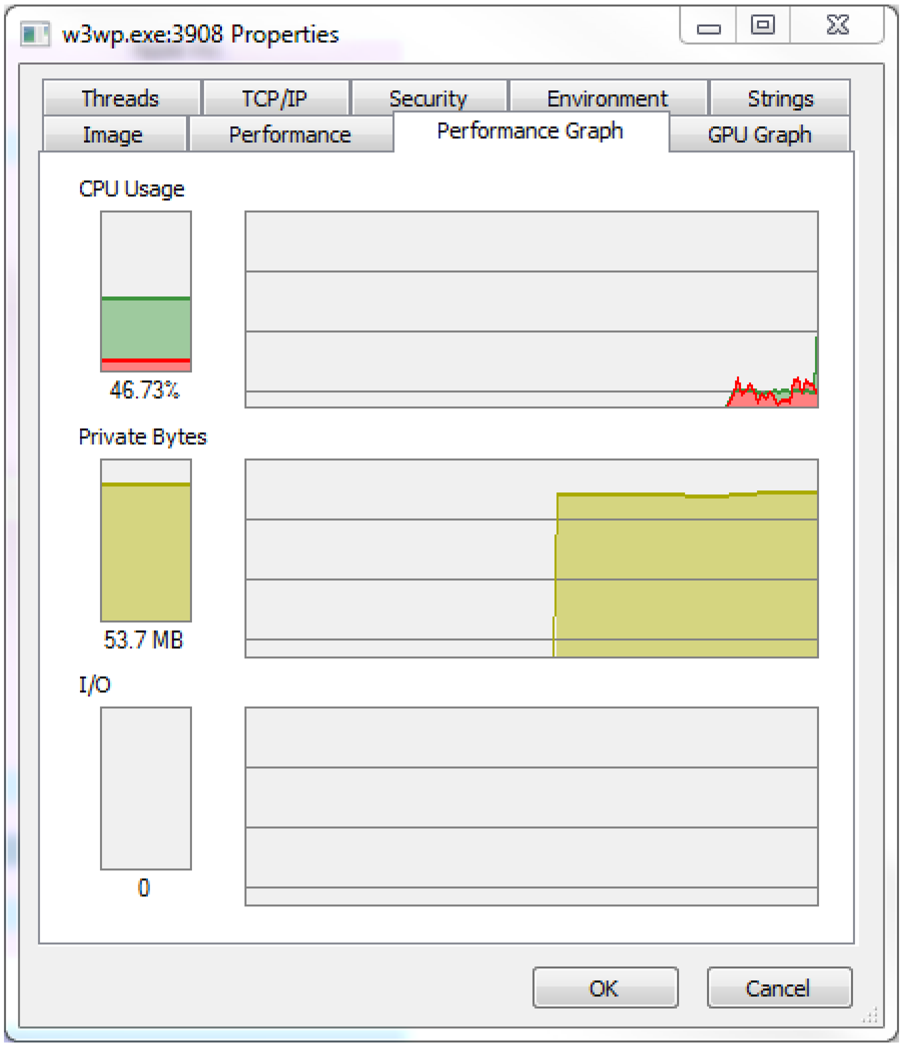
In conclusion, IIS buffers to disk in order to handle and keep the memory usage under control. We'll see this more in details in the next section.

Real world

I deployed the application on IIS and played with a file of 683 MB.

Working with the default value of requestLengthDiskThreshold

In this case, I monitored the execution of the IIS Worker process (w3wp.exe) through Process Explorer and got the following result when the upload was finishing:



In the monitoring above, we conclude that the IIS Worker Process was finishing the file upload (the increase in CPU usage from ~10% to 46,73%). We can also notice that during the process of buffering the memory usage is about ~50 MB even if the file being uploaded is 683 MB of size and that the CPU usage is about ~10%. But how does it work?

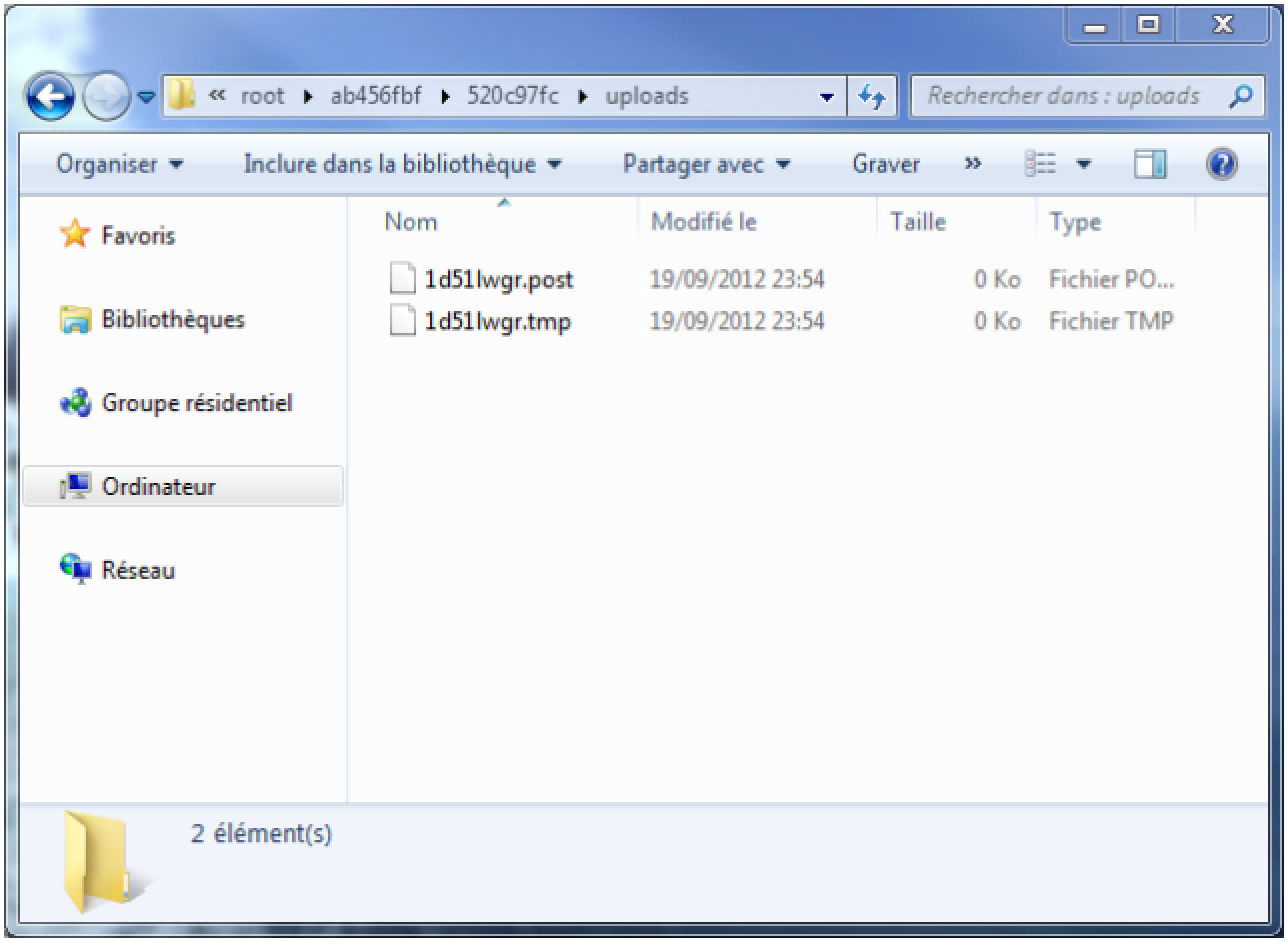
Well, when the upload is fired IIS Worker process writes temporary files in the following folder:

```
%windir%\Microsoft.NET\Framework64\v4.0.30319\Temporary ASP.NET
Files\root\RandomName\RandomName\uploads
```

in case of 64 bits operating systems. And, the one below in case of 32 bits operating systems:

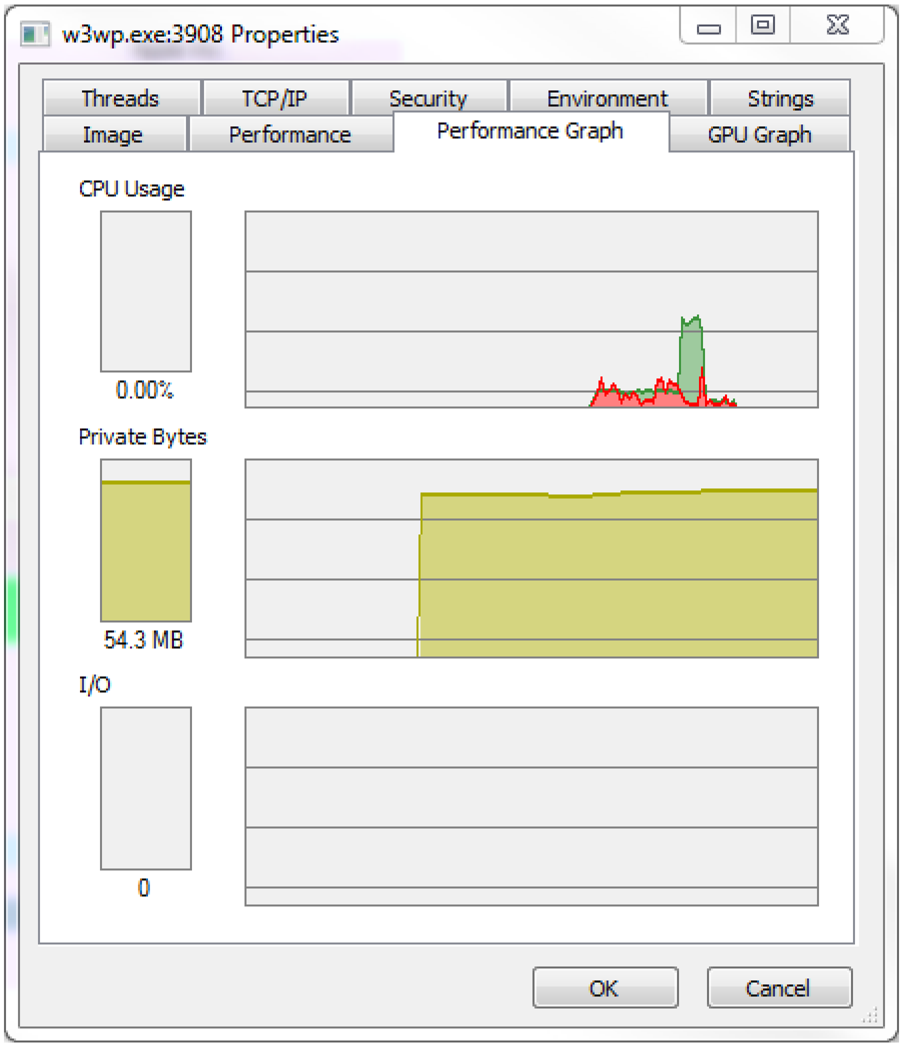
```
%windir%\Microsoft.NET\Framework\v4.0.30319\Temporary ASP.NET
Files\root\RandomName\RandomName\uploads
```

During the upload process, I was watching this folder and I've been able to see the w3wp.exe creating the temporary files:

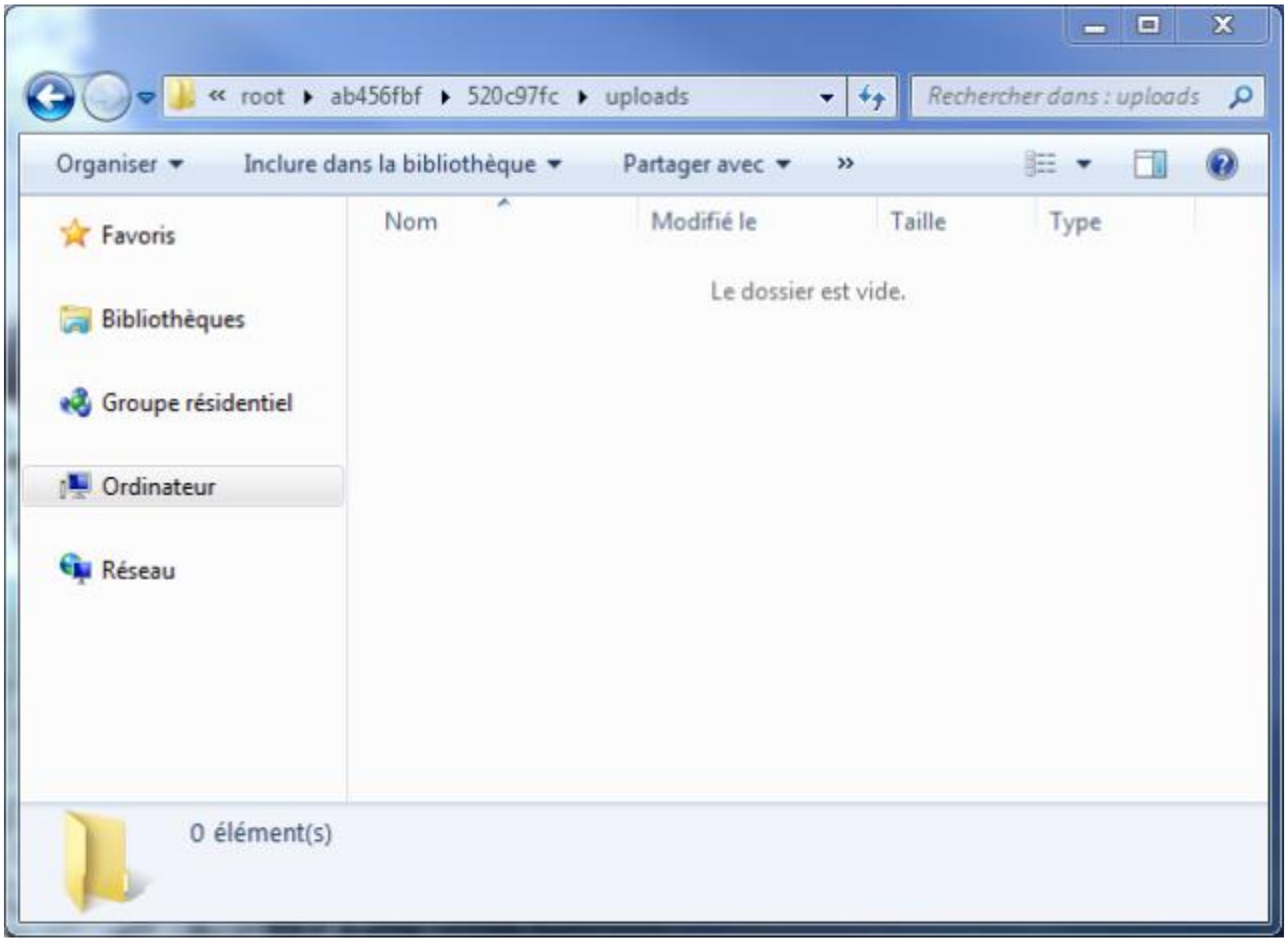


Nom	Modifié le	Taille	Type
1d51lwgr.post	19/09/2012 23:54	0 Ko	Fichier PO...
1d51lwgr.tmp	19/09/2012 23:54	0 Ko	Fichier TMP

Once the upload was finished, The entire file was created in the storage folder and the w3wp.exe returned to its normal activity:

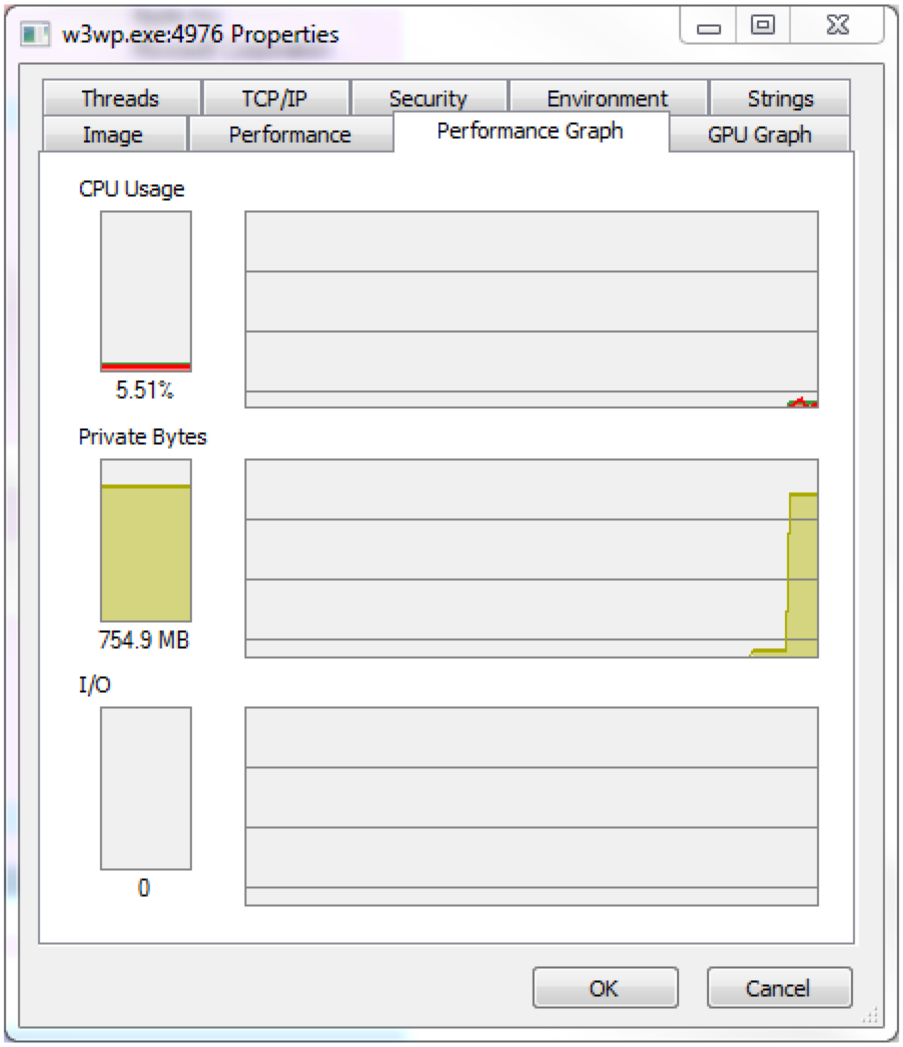


And I've also been able to see the w3wp.exe deleting the temporary files after that the file has been uploaded:



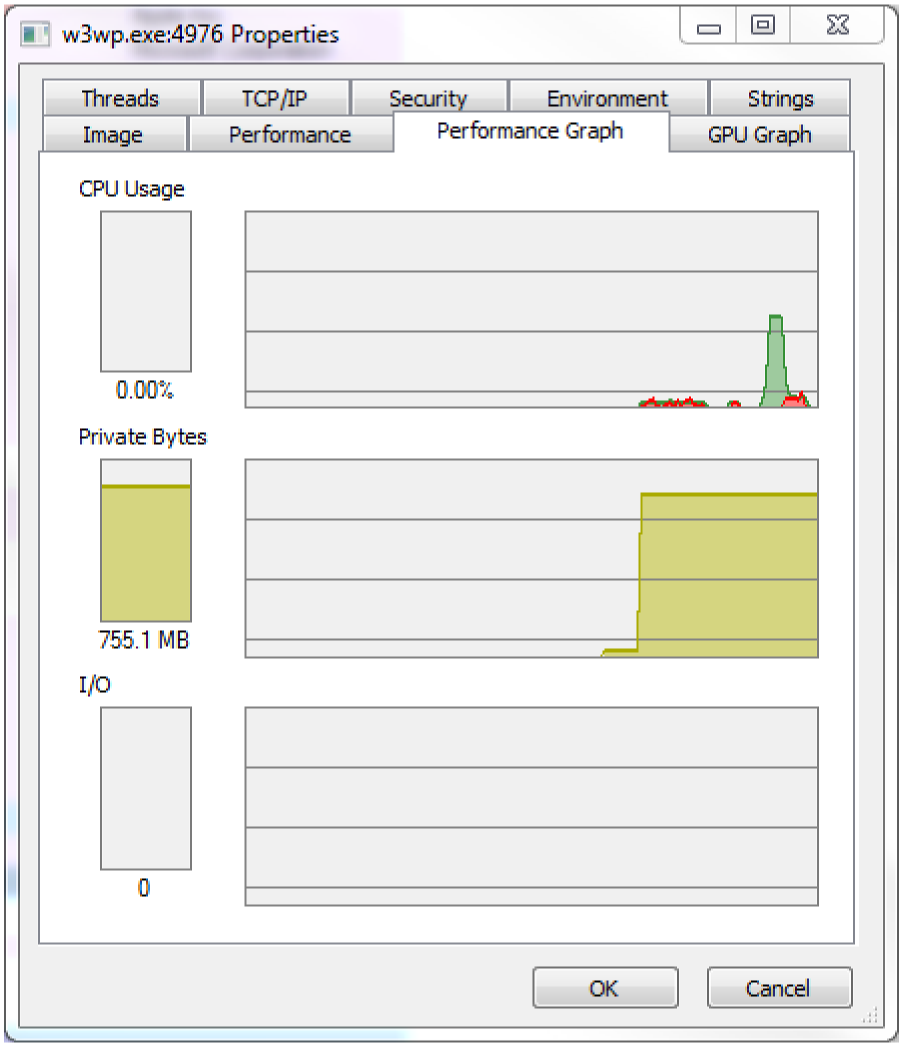
Working with a big value of requestLengthDiskThreshold

In this case, I set up a value bigger than the file size (683 MB) for `requestLengthDiskThreshold` and monitored the execution of the IIS Worker process through Process Explorer and got the following result when the upload was just starting:



Wow! The increase in memory usage was incredible. From ~50MB to 754.0 MB in few seconds. This increase is the result of the cration of a buffer of ~683 MB in memory.

When the upload was finished I got the following monitoring:



The file has been created in the storage folder and the w3wp.exe was consuming 755.1 MB in memory.

We can notice that in the first case where the `requestLengthDiskThreshold` was set to its default value the w3wp.exe was using about ~50 MB in memory and in the following case the w3wp.exe consumed ~x15 in memory comparing to the first.

Conclusion

We can notice that best configuration is to let the `requestLengthDiskThreshold` to its default value. In that case, the w3wp.exe will manage memory usage very well through the buffering technique.

An in depth look in the HTML5 Blob Interface

In this section we will see what's going on behind the scene between the server and the client when chunked uploads are ongoing.

Before to start the discussion about that, I'll talk about chunked uploads and we'll see how I added chunked uploads support on the client side and on the server side.

What's a chunked upload?

Chunked upload is a feature introduced through the Blob Interface of the HTML5 File API. A Blob represents an immutable row data. It provides a method to slice data between ranges of bytes into chunks of raw data and an attribute representing the chunk size.

If the maximum chunk size is set for uploads, the client side code splits up the file into multiple Blobs and sends each blob to the server. Thus, the file is saved on the server chunk by chunk. This is very useful for uploading large files and allows to implement resuming for instance.

What changed on the client side?

The jQuery File Upload plugin provides an option to enable chunked uploads. This option is called `maxChunkSize`. It allows to set the maximum chunk size in bytes. This option was added as an option `MaxChunkSize` to the user control, it defaults to 10 MB. `EnableChunkedUploads` option was added too in order to enable or disable the chunked uploads, it defaults to False. Chunked uploads are supported by Google Chrome and FireFox 4+. If the chunked uploads are enabled, all other browsers work through a fallback.

Chunked uploads set the following headers:

```
{
    'X-File-Name': file.name,
    'X-File-Type': file.type,
    'X-File-Size': file.size
}
```

These headers allow the server to combine the uploaded Blobs into one file.

The jQuery File Upload plugin provides two important setting options to retrieve the current chunk index and the total number of chunks. These two settings allow to upload the chunks into one file. Thus, these two settings (`settings.chunkIndex` and `settings.chunksNumber`) are sent to the server through HTTP headers as follows:

```
 $('#fileupload').fileupload({
    // In order for chunked uploads to work in Mozilla Firefox,
    // the multipart option has to be set to false. This is due to
    // the Gecko 2.0 browser engine - used by Firefox 4+ - adding blobs
    // with an empty filename when building a multipart upload request
    // using the FormData interface
    multipart: false,
    maxChunkSize: maxChunkSize, // Defaults to 10 MB
    beforeSend: function (jqXHR, settings) {
        if (settings.chunksNumber) {
            jqXHR.setRequestHeader('X-Chunk-Index', settings.chunkIndex);
            jqXHR.setRequestHeader('X-Chunks-Number', settings.chunksNumber);
        }
    }
});
```

To still get the "cancel" operation running, a DELETE request is sent to the server to delete the partial file (collection of chunks) being uploaded:

```
// Delete the chunks written on the server when canceling the upload
$('#fileupload').bind('fileuploadfail', function (e, data) {
    if (data.errorThrown === 'abort') {
        $.ajax({
            url: 'Upload.axd?storageFolder='
               + '<%= Server.UrlEncode(StorageFolder)  %>'
               + '&'
               + $.param({ f: data.files[0].name }),
            type: 'DELETE'
        });
    }
});
```

That's it for the client side.

What changed on the server side?

Browsers don't allow custom headers in cross-site file uploads. Thus, the custom headers described in the previous section needs to be declared as allowed. Therefore, the `ProcessRequest` method changed as follows:

```
public void ProcessRequest(HttpContext context)
{
    context.Response.AddHeader("Pragma", "no-cache");
    context.Response.AddHeader("Cache-Control", "private, no-cache");
    // Cross-site chunked uploads
    context.Response.AddHeader("Access-Control-Allow-Headers", "X-File-
Name,X-File-Type,X-File-Size");
    DoWork(context);
}
```

In order to be able to calculate the exact file size being uploaded through the chunk index and the maximum chunk size, I added the following property:

```
public long MaxChunkSize
{
    get
    {
        return
long.Parse(HttpContext.Current.Request.QueryString["maxChunkSize"]);
    }
}
```

Then, I changed the `UploadPartialFile` method in order to handle the Blobs being uploaded as follows:

```
private void UploadPartialFile(string fileName, HttpContext context,
List<FileStatus> statuses)
{
    NameValueCollection headers = context.Request.Headers;

    //
    // Retrieve chunks information from the request
    //
    int chunkSize = int.Parse(headers["Content-Length"]);
    int chunksNumber = int.Parse(headers["X-Chunks-Number"] ?? "0");
    int chunkIndex = int.Parse(headers["X-Chunk-Index"] ?? "0");

    long fileLength = chunkIndex * MaxChunkSize + chunkSize; // in Bytes!
    FileStatus status = new FileStatus(fileName, fileLength, StorageFolder);

    try
    {
        string fullName = Path.Combine(StorageFolder,
Path.GetFileName(fileName));
        const int bufferSize = 1024;
        Stream inputStream = context.Request.InputStream;

        // FileShare.Delete for fileuploadfail event
        using (FileStream fileStream =
            new FileStream(fullName, FileMode.Append, FileAccess.Write,
FileShare.Delete))
        {
            byte[] buffer = new byte[bufferSize];

            int l;
            while ((l = inputStream.Read(buffer, 0, bufferSize)) > 0)
            {
                fileStream.Write(buffer, 0, l);
            }

            fileStream.Flush();
        }
    }
    catch (Exception e)
    {
        status.error = e.Message;
    }

    statuses.Add(status);
}
```

That's it for the server side.

An in depth look

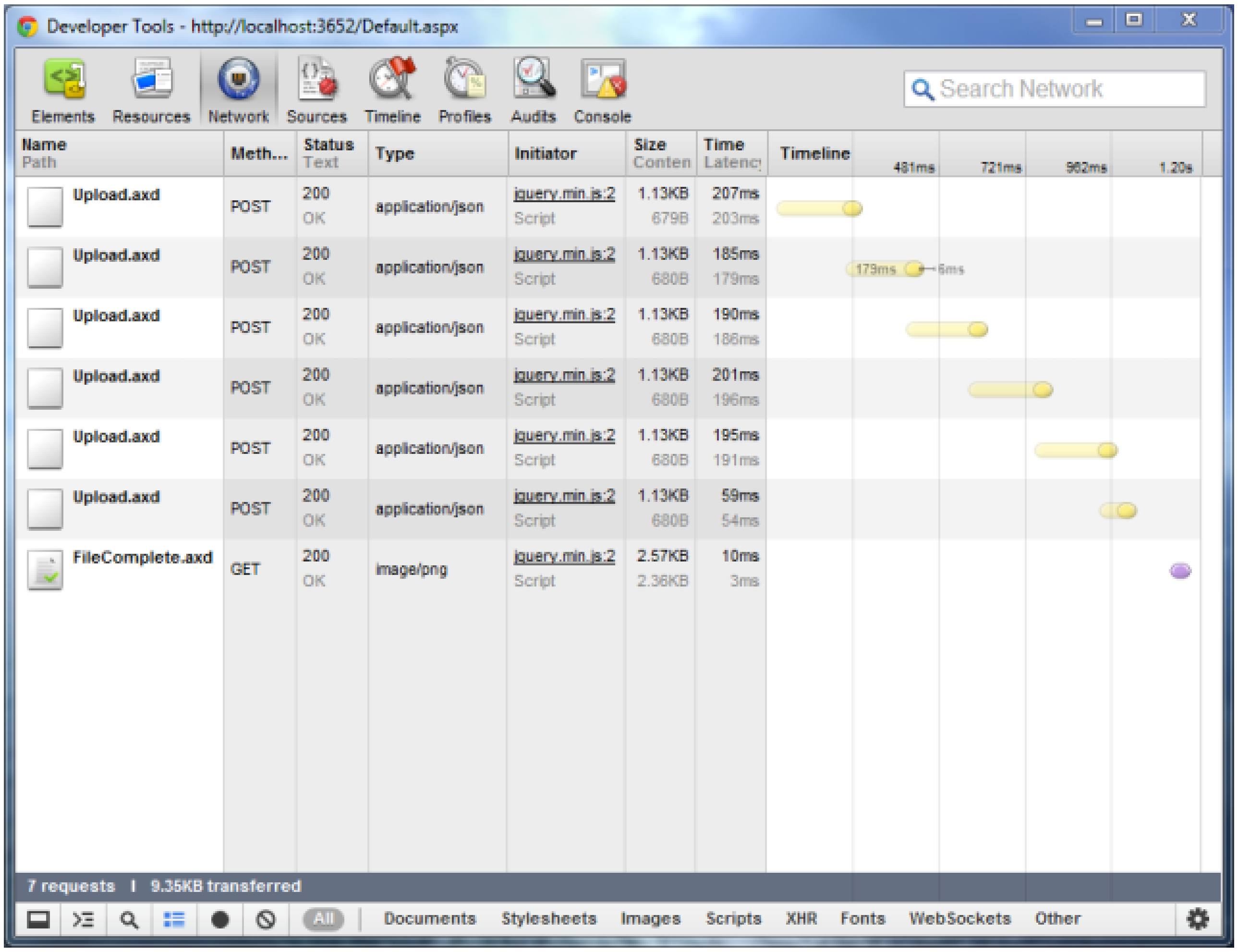
In this section we'll see what's going on behind the scene between the client and the server while Blobs are transiting in the network.

Ok, to make this work I enabled chunked uploads by setting `EnableChunkedUploads` to True, set up `MaxChunkSize` to 5 MB, and played with the famous video file `Wildlife.wmv` HD (26.25 MB) in Google Chrome.

Below the options configuration:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
AcceptFileTypes=".*" EnableChunkedUploads="true" MaxChunkSize="5000000"
SequentialUploads="true" />
```

Let's have a look at this network capture:



Name Path	Meth...	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 679B	207ms 203ms	
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 680B	185ms 179ms	179ms 6ms
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 680B	190ms 186ms	
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 680B	201ms 196ms	
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 680B	195ms 191ms	
Upload.axd	POST	200 OK	application/json	jquery.min.js:2 Script	1.13KB 680B	59ms 54ms	
FileComplete.axd	GET	200 OK	image/png	jquery.min.js:2 Script	2.57KB 2.36KB	10ms 3ms	

Timeline: 481ms 721ms 962ms 1.20s

7 requests | 9.35KB transferred

The file size is 26.25 MB and the maximum chunk size is 5 MB. So in theory, the file will be devided in 6 chunks, 5 chunks of 5 MB and 1 chunk of 1.25 MB. In practice, this is exactly what the network capture is showing. We can see that 6 chunks were sent from the client to the server and that the server handled each chunk through the `Upload.axd` handler. The `Upload.axd` handler executed the `UploadPartialFile` method for each chunk and thus I've been able to see the construction of the `Wildlife.wmv` file on the server storage folder chunk per chunk.

We can also notice that the request length of the last `Upload.axd` call is lower than the others. This is simply due to the fact that this chunk is 1.25 MB size and all the others 5 MB size.

And finally when all chunks were combined into one file, we can see that the `FileComplete.axd` fired in order to give the complete thumbnail to the client.

Conclusion

The Blob Interface is a great feature introduced by the HTML5 File API. This feature is currently not supported by all the browsers. The user control provides fallbacks but one day all the browsers will support it. Chunked uploads are very useful for large files uploads, and can be used for resuming uploads.

An in depth look in resumable file uploads

In this section we'll see how resumable file uploads work.

Before to start talking about that, I'll start by explaining how I added the resumable file upload functionality.

What's a resumable file upload?

A resumable file upload is an upload that takes into consideration the uploaded bytes on the server of an aborted upload. That is to say, If the end-user starts uploading a file of 1 GB to the server and If the user cancels his upload at 0.5 GB, when the end-user uploads the same file again, the upload will start from 0.5 GB (only the last 0.5 GB are uploaded).

A resumable file upload works with the HTML5 Blob interface an thus needs the chunked uploads to be enabled.

What changed on the client side?

The only thing that changed on the client side in order to make the resumable file uploads working is the `add` callback. Indeed, the `add` callback has been overriden in order to send a JSON request with the file name to the server. If the file exists on the storage folder, the server sends a response in JSON to the client containing file information including file size which is set in the `uploadedBytes` of the jQuery file upload plugin. If `uploadedBytes` option exits, the plugin uploads the remaining parts of the file as a chunked upload. The override has been implemented as follows:

```
if (resume) {
    $('#fileupload').fileupload({
        add: function (e, data) {
            var that = this;
            $.getJSON(handler, { f: data.files[0].name }, function (file) {
                data.uploadedBytes = file && file.size;
                $.blueimpUI.fileupload.prototype.options.add.call(that, e,
data);
            });
        }
    });
}
```

In this override, the file size is retrieved from the server and set to the `uploadedBytes` option.

`resume` is calculated as follows:

```
resume = enableChunkedUploads && '<%= Resume %>'.toLowerCase() === 'true'
```

That's it for the client side.

What changed on the server side?

On the server side a `Resume` property was added as follows:

```
public bool Resume
{
    get
    {
        return bool.Parse(HttpContext.Current.Request.QueryString["resume"]);
    }
}
```

And the `DoWork` method has changed as below in order to take the resume request into consideration:

```
private void DoWork(HttpContext context)
{
    switch (context.Request.HttpMethod)
    {
        case "HEAD":
        case "GET":
            if (GivenFilename(context))
            {
                if (Resume)
                {
                    SendFileInfo(context);
                }
                else
                {
                    DeliverFile(context);
                }
            }
            else
            {
                ListCurrentFiles(context);
            }
            break;
        case "POST":
```

```
        case "PUT":
            UploadFile(context);
            break;
        case "DELETE":
            DeleteFile(context);
            break;
        case "OPTIONS":
            ReturnOptions(context);
            break;
        default:
            context.Response.ClearHeaders();
            context.Response.StatusCode = 405;
            break;
    }
}
```

A new method `SendFileInfo` was added in order to send to the client, information on the file and to set the `uploadedBytes` option as follows:

```
private void SendFileInfo(HttpContext context)
{
    string filename = context.Request["f"];
    string filePath = Path.Combine(StorageFolder, filename);
    FileStatus status = File.Exists(filePath)
                        ? new FileStatus(new FileInfo(filePath),
StorageFolder)
                        : new FileStatus(filename, 0, StorageFolder);
    string jsonObj = _javaScriptSerializer.Serialize(status);
    context.Response.AddHeader("Content-Disposition", "inline;
filename=\"files.json\"");
    context.Response.ContentType = "application/json";
    context.Response.Write(jsonObj);
}
```

Then the `UploadPartialFile` method was updated in order to calculate the exact file size for resumable file uploads as follows:

```
private void UploadPartialFile(string fileName, HttpContext context,
List<FileStatus> statuses)
{
    NameValueCollection headers = context.Request.Headers;

    //
    // Retrieve chunks information from the request
    //
    int chunkSize = int.Parse(headers["Content-Length"]);
    int chunksNumber = int.Parse(headers["X-Chunks-Number"] ?? "0");
    int chunkIndex = int.Parse(headers["X-Chunk-Index"] ?? "0");

    string fullName = Path.Combine(StorageFolder,
Path.GetFileName(fileName));
    long fileLength;// in Bytes!
    if (Resume)
    {
        long previousLength = GetFileSize(fullName);
```

```
            fileLength = previousLength + chunkSize;
        }
        else
        {
            // If not resuming, no need to use GetFileSize
            fileLength = chunkIndex * MaxChunkSize + chunkSize;
        }

        FileStatus status = new FileStatus(fileName, fileLength, StorageFolder);

        try
        {
            const int bufferSize = 1024;
            Stream inputStream = context.Request.InputStream;

            // FileShare.Delete for fileuploadfail event
            using (FileStream fileStream =
                new FileStream(fullName, FileMode.Append, FileAccess.Write,
FileShare.Delete))
            {
                byte[] buffer = new byte[bufferSize];

                int l;
                while ((l = inputStream.Read(buffer, 0, bufferSize)) > 0)
                {
                    fileStream.Write(buffer, 0, l);
                }

                fileStream.Flush();
            }
        }
        catch (Exception e)
        {
            status.error = e.Message;
        }

        statuses.Add(status);
}
```

A new method `GetFileSize` has been added in order to get the file size of aborted file uploads for resuming them:

```
private long GetFileSize(string fullName)
{
    long fileSize = File.Exists(fullName)
                        ? new FileInfo(fullName).Length
                        : 0;
    return fileSize;
}
```

That's it for the server side.

An in depth look

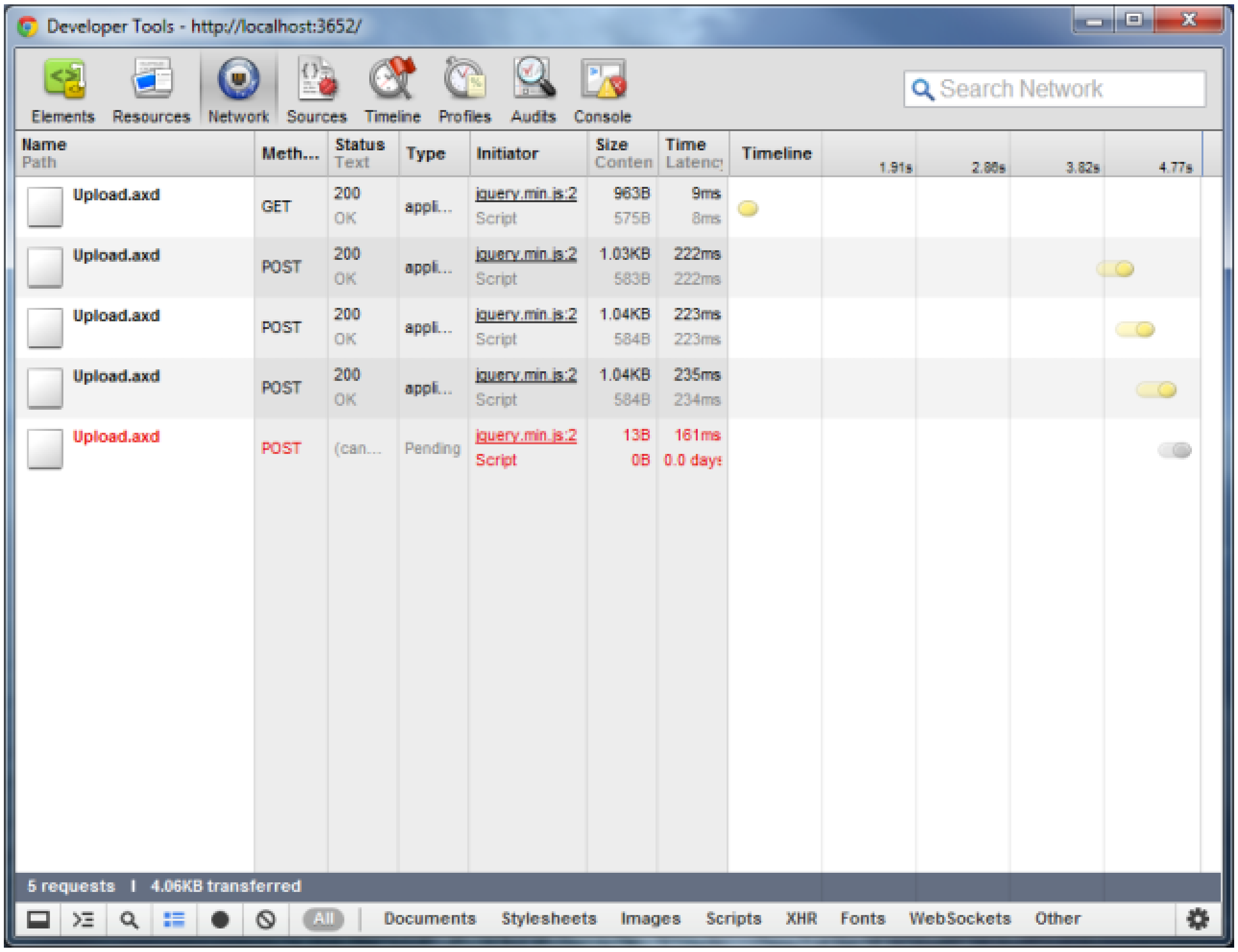
In this section we'll see how resumable file uploads work.

Ok, to make this work I enabled resumable uploads by setting `EnableChunkedUploads` and `Resume` to True and, `MaxChunkSize` to 5 MB, and played with the famous video file `Wildlife.wmv` HD (26.25 MB) in Google Chrome.

Below the options configuration:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
AcceptFileTypes=".*" SequentialUploads="true" EnableChunkedUploads="true"
MaxChunkSize="5000000" Resume="true" />
```

Let's have a look at this network capture:

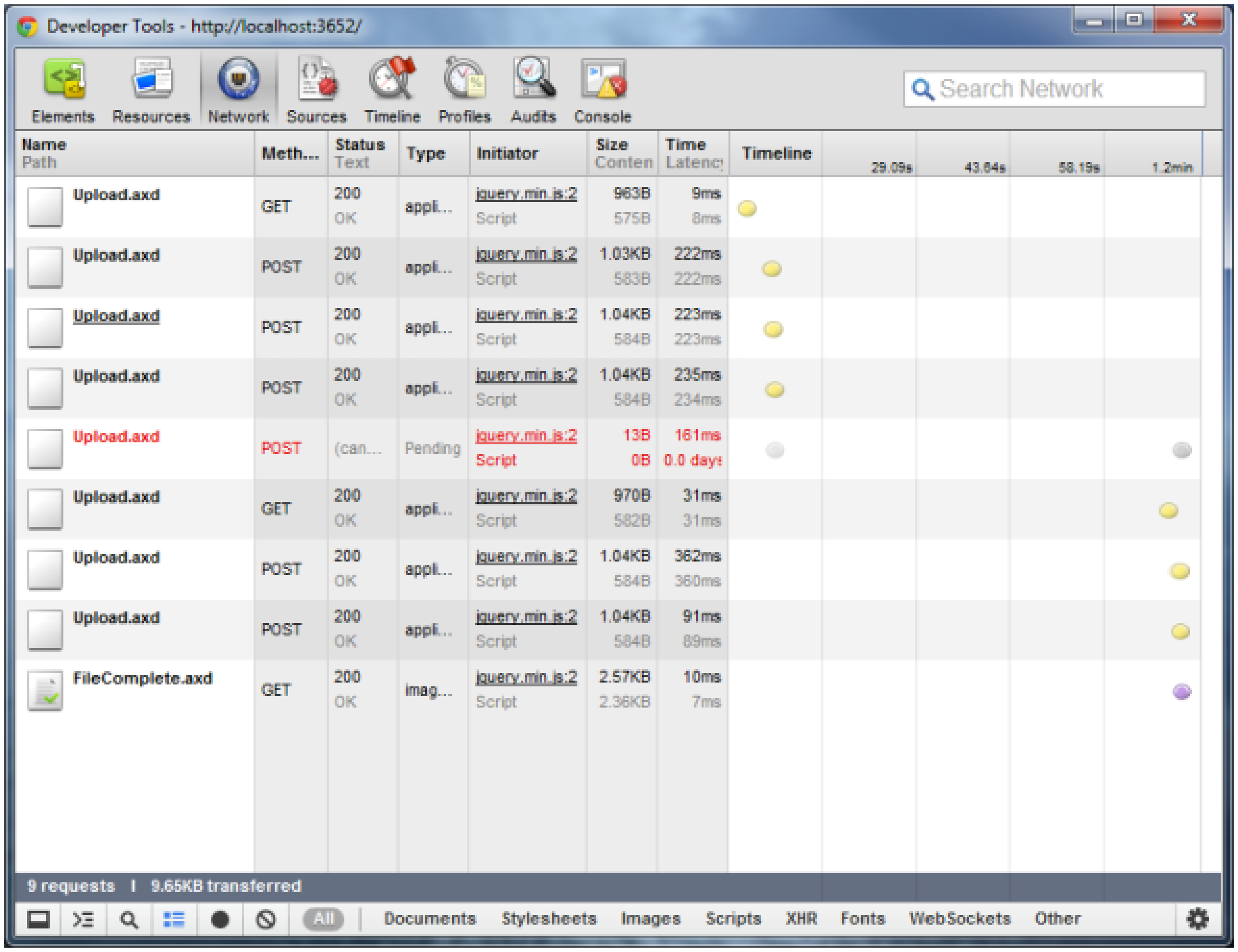


Name / Path	Meth...	Status / Text	Type	Initiator	Size / Content	Time / Latency	Timeline
Upload.axd	GET	200 OK	appli...	jquery.min.js:2 Script	963B 575B	9ms 8ms	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.03KB 583B	222ms 222ms	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	223ms 223ms	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	235ms 234ms	
Upload.axd	POST	(can...	Pending	jquery.min.js:2 Script	13B 0B	161ms 0.0 days	

5 requests | 4.06KB transferred

When the file upload reached 18.6 MB, I've cancelled it. Thus we can see that fourth `Upload.axd` POST request (in red) was canceled. In theory, the file will be devided in 6 chunks, 5 chunks of 5 MB and 1 chunk of 1.25 MB. But due to the abort operation, it will be a little bit different.

Ok, I launched the upload of the same file again. So, let's have a look at this network capture:



Name Path	Meth...	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
Upload.axd	GET	200 OK	appli...	jquery.min.js:2 Script	963B 575B	9ms 8ms	●
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.03KB 583B	222ms 222ms	●
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	223ms 223ms	●
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	235ms 234ms	●
Upload.axd	POST	(can...	Pending	jquery.min.js:2 Script	13B 0B	161ms 0.0 days	●
Upload.axd	GET	200 OK	appli...	jquery.min.js:2 Script	970B 582B	31ms 31ms	●
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	362ms 360ms	●
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.04KB 584B	91ms 89ms	●
FileComplete.axd	GET	200 OK	imag...	jquery.min.js:2 Script	2.57KB 2.36KB	10ms 7ms	●

9 requests | 9.65KB transferred

We can see that the upload was resumed and that the other chunks that were missing have been appended to the partial file already uploaded in the server just before the abort operation. The file was uploaded in 6 server calls, but due to the abort operation the size of the last Blob was bigger that 1.25 MB. Indeed, its size was 26.25 MB - (18.6 MB + 5 MB) = 2.65 MB.

And finally when all chunks were combined into one file, we can see that the `FileComplete.axd` fired in order to give the complete thumbnail to the client.

An in depth look in automatic retry

In this section we'll see how automatic retries work.

Before to start talking about that, I'll start by explaining how I added automatic retries functionality.

What's an automatic retry?

An automatic retry occurs when a file upload fails (network cable unplugged, internet connection broken, or other failures). The automatic retry will automatically resume the file upload after retrieving the bytes that have been uploaded just before the failure.

An option allows to set the number of retries and another option allows to set the timeout before the file upload is resumed in order to prevent endless loops. The timeout is increased after every retry in order to extend the waiting time.

What changed on the client side?

In order to add this functionality, the fail callback has been overriden in order to handle automaticlly resuming in case of failures. The override is illustrated below:

```
if (autoRetry) {
    $('#fileupload').fileupload({
        maxRetries: maxRetries,
        retryTimeout: retryTimeout, // Milliseconds!
        fail: function (e, data) {
             var fu = $(this).data('fileupload'),
             retries = data.context.data('retries') || 0,
             retry = function () {
                $.getJSON(handler, { f: data.files[0].name })
                    .done(function (file) {
                        data.uploadedBytes = file && file.size;
                        // clear the previous data
                        data.data = null;
                        data.submit();
                    })
                    .fail(function () {
                        fu._trigger('fail', e, data);
                    });
            };
            if (data.errorThrown !== 'abort' &&
            data.errorThrown !== 'uploadedBytes' &&
            retries < fu.options.maxRetries) {
                //alert(retries);
                retries++;
                data.context.data('retries', retries);
                window.setTimeout(retry, retries * fu.options.retryTimeout);
                return;
            }
            data.context.removeData('retries');
            $.blueimpUI.fileupload.prototype.options.fail.call(this, e,
data);
        }
    });
}
```

When an upload fails, the `uploadedBytes` is set and the file is submitted. This operation occurs until the maximum number of retries is reached. `window.setTimeout` is used to launch the retry operation.

`maxRetries` and `retryTimeout` are retrieved as follows:

```
maxRetries = parseInt('<%= MaxRetries %>'),
retryTimeout = parseInt('<%= RetryTimeout %>'), // in Milliseconds!
```

That's it for the client side.

What changed on the server side?

Nothing changed on the server side.

An in depth look

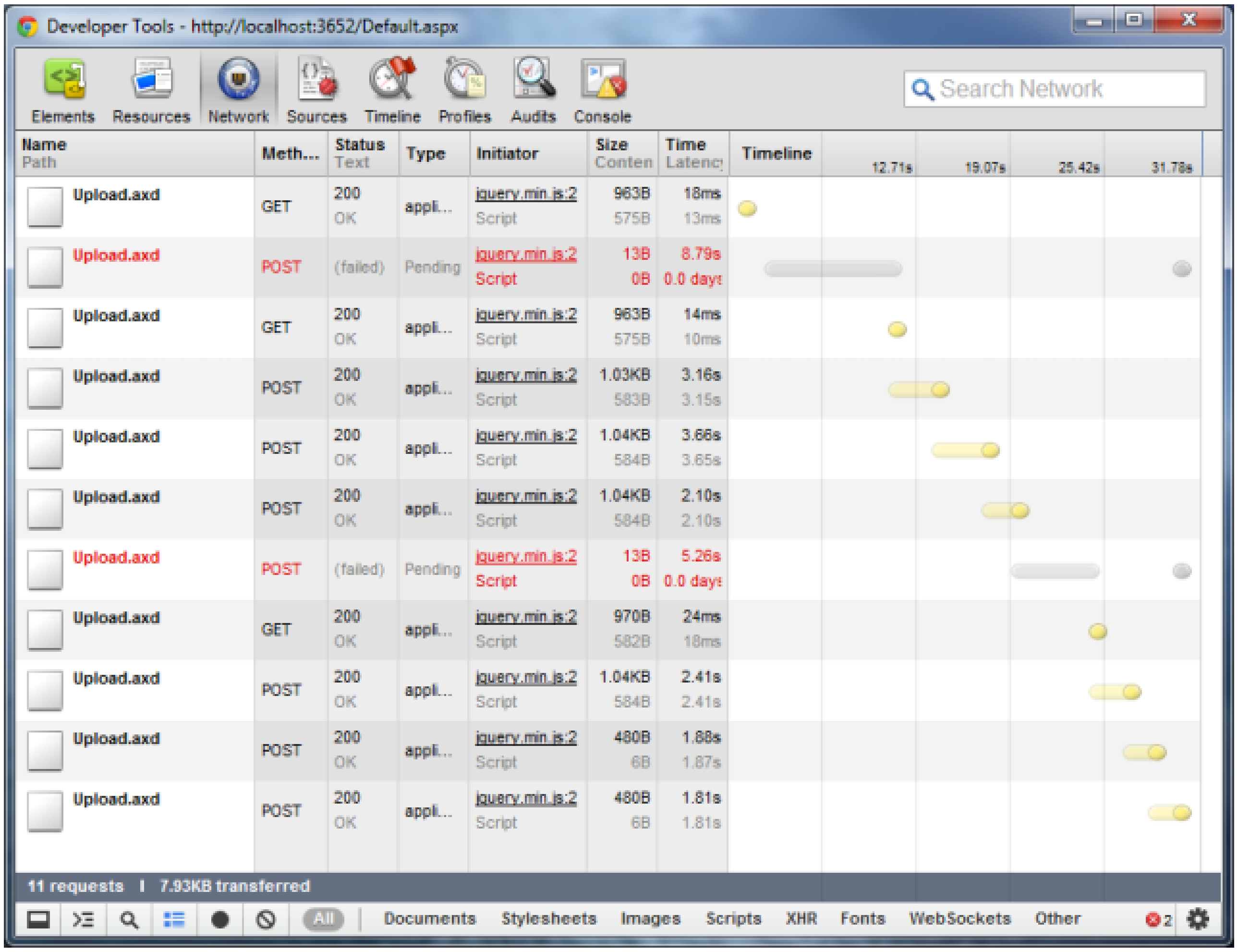
In this section we'll see how automatic retries work.

Ok, to make this work I enabled automatic retries by setting `EnableChunkedUploads`, `Resume` and `AutoRetry` to True, `MaxChunkSize` to 5 MB, `RetryTimeout` to 500 milliseconds and `MaxRetries` to 100, and played with the famous video file `Wildlife.wmv` HD (26.25 MB) in Google Chrome.

Below the options configuration:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
AcceptFileTypes=".*"
    SequentialUploads="true" EnableChunkedUploads="true"
MaxChunkSize="5000000" Resume="true"
    AutoRetry="true" RetryTimeout="500" MaxRetries="100" />
```

Let's have a look at this network capture:



Name Path	Meth...	Status Text	Type	Initiator	Size Conten	Time Latency
Upload.axd	GET	200 OK	appl...	jquery.min.js:2 Script	963B 575B	18ms 13ms
Upload.axd	POST	(failed)	Pending	jquery.min.js:2 Script	13B 0B	8.79s 0.0 days
Upload.axd	GET	200 OK	appl...	jquery.min.js:2 Script	963B 575B	14ms 10ms
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	1.03KB 583B	3.16s 3.15s
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	1.04KB 584B	3.66s 3.65s
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	1.04KB 584B	2.10s 2.10s
Upload.axd	POST	(failed)	Pending	jquery.min.js:2 Script	13B 0B	5.26s 0.0 days
Upload.axd	GET	200 OK	appl...	jquery.min.js:2 Script	970B 582B	24ms 18ms
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	1.04KB 584B	2.41s 2.41s
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	480B 6B	1.88s 1.87s
Upload.axd	POST	200 OK	appl...	jquery.min.js:2 Script	480B 6B	1.81s 1.81s

11 requests | 7.93KB transferred

When the upload was launched, I forced a failure (first `Upload.axd` request in red). We can notice that after this failure the upload resumed automatically. Then when the upload status was about ~50% I forced another failure again (second red `Upload.axd` request). We can notice that after this second failure the upload resumed automatically again. And finally, that the uploaded bytes before the first failure were appended to the uploaded bytes before the second failure to the uploaded bytes after the second failure. Thus, the uploaded bytes were combined chunk by chunk into one file without any data loss.

An in depth look in large file upload

In this section we'll study the behavior of the user control on large file upload.

The user control is based on the jQuery File Upload plugin which is designed to upload large files, the default maximum allowed file size for uploads is unlimited and can be set through the `MaxFileSize` option. That is to say, the only limitation is browser capabilities when chunked uploads are disabled. We'll see the why?

Chunked uploads enabled

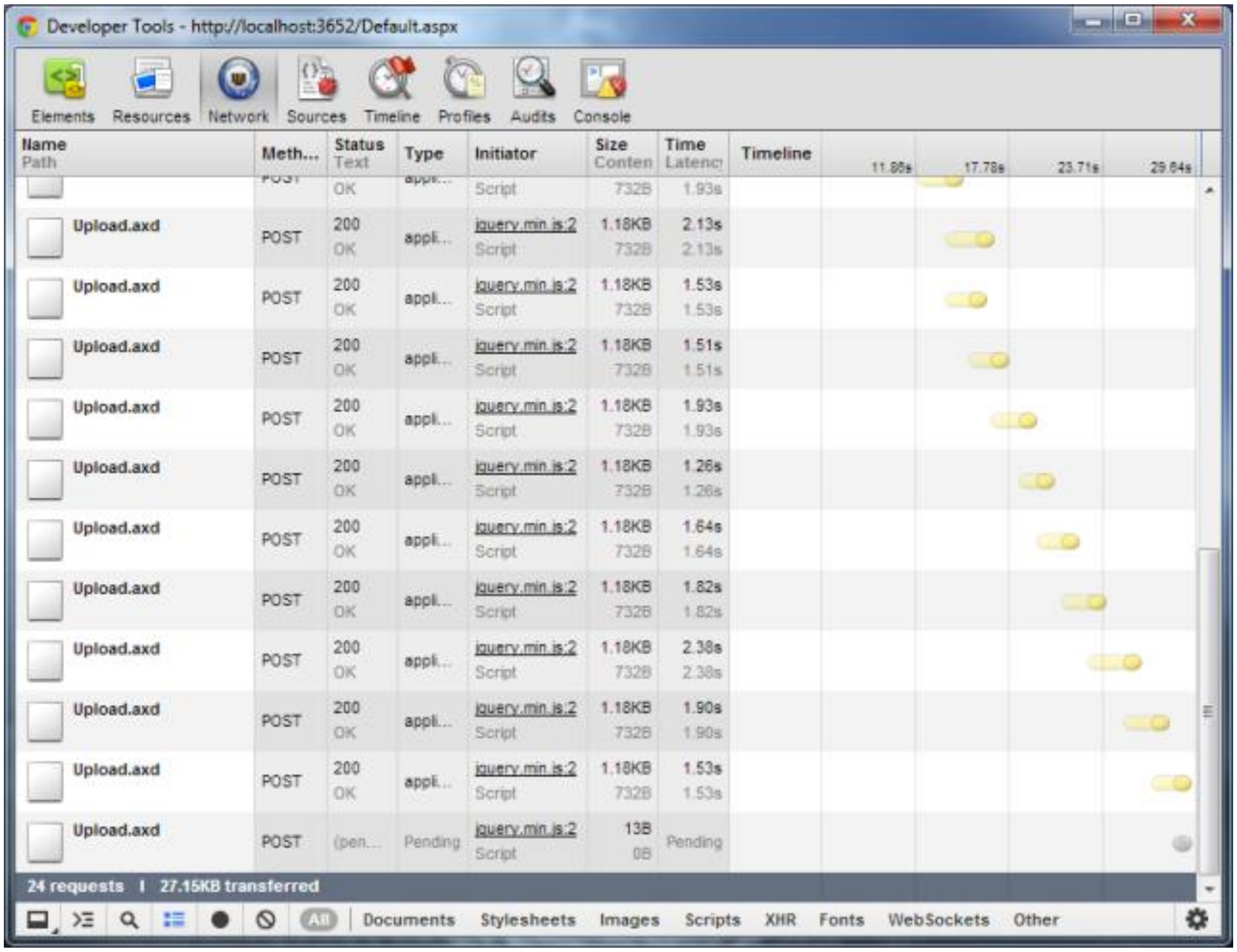
I've tested the user control by uploading files of 10 GB and 25 GB through chunked uploads, and uploads were successful perfectly. Let's dig into these uploads.

Ok, to make this work I enabled chunked uploads by setting `EnableChunkedUploads` to True, set up `MaxChunkSize` to 15 MB, and started playing with a file of 10 GB in Google Chrome.

Below the options configuration:

```
<jQuery:MultipleFileUpload ID="MultipleFileUpload" runat="server"
AcceptFileTypes=".*" SequentialUploads="false" EnableChunkedUploads="true"
MaxChunkSize="15000000" Resume="true" AutoRetry="true" RetryTimeout="500"
MaxRetries="100" OnFileUploadDone="fileuploaddone" LimitConcurrentUploads="2"
ForceIframeTransport="false" AutoUpload="false" PreviewAsCanvas="true" />
```

When upload was just starting, I took a network capture. Let's have a look at this:



Developer Tools - http://localhost:3652/Default.aspx

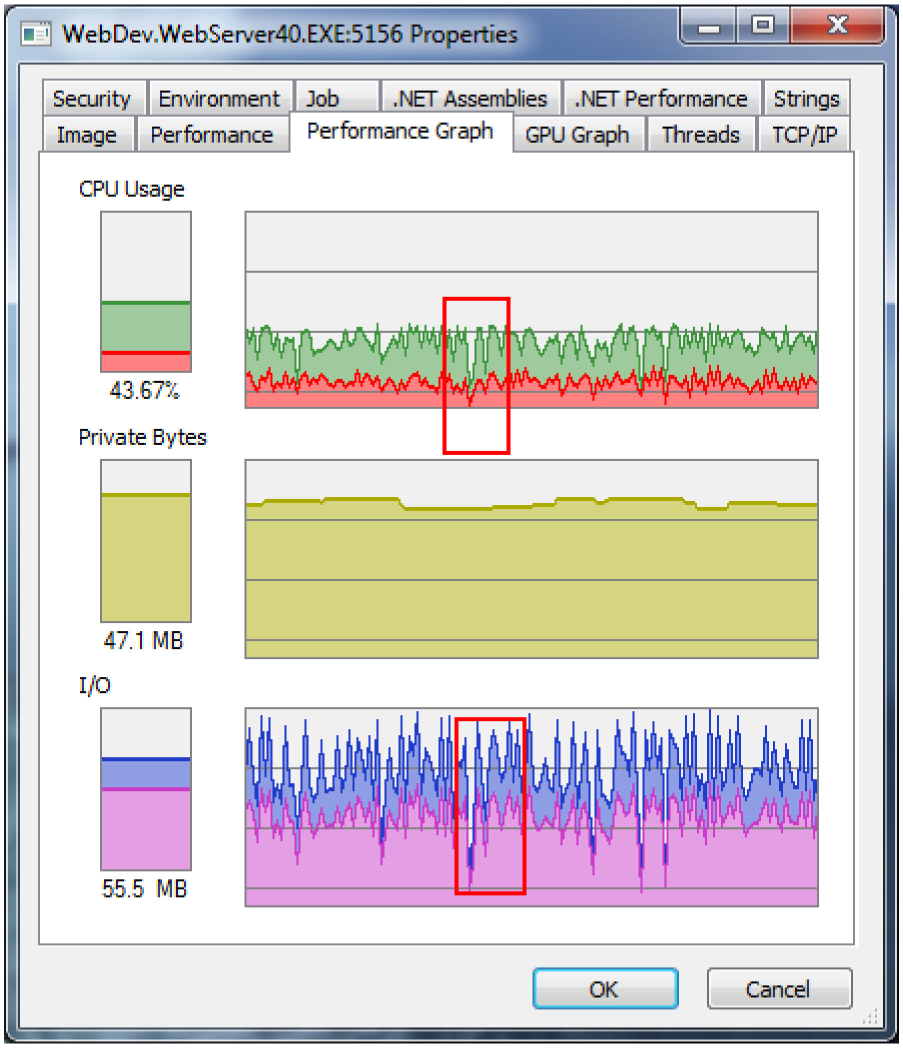
Name / Path	Meth...	Status / Text	Type	Initiator	Size / Conten	Time / Latency	Timeline
	POST	OK	appli...	Script	732B	1.93s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	2.13s 2.13s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.53s 1.53s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.51s 1.51s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.93s 1.93s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.26s 1.26s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.64s 1.64s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.82s 1.82s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	2.38s 2.38s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.90s 1.90s	
Upload.axd	POST	200 OK	appli...	jquery.min.js:2 Script	1.18KB 732B	1.53s 1.53s	
Upload.axd	POST	(pen...	Pending	jquery.min.js:2 Script	13B 0B	Pending	

24 requests | 27.15KB transferred

We can notice that chunked uploads are in action at the beginning of upload. The file of 10 GB is being uploaded to the server chunk by chunk through HTTP requests. Each chunk is 15 MB size, the file is splitted into 667 chunks of 15 MB. Thus, the user control fired 667 HTTP requests, each request contains a chunk. And finally, chunks were combined one by one into one file through the `Upload.axd` HTTP handler. When upload was working, I was watching the server storage folder and I've been able to see the size of the file size growing up chunk by chunk until the 10 GB were reached through 667 HTTP requests.

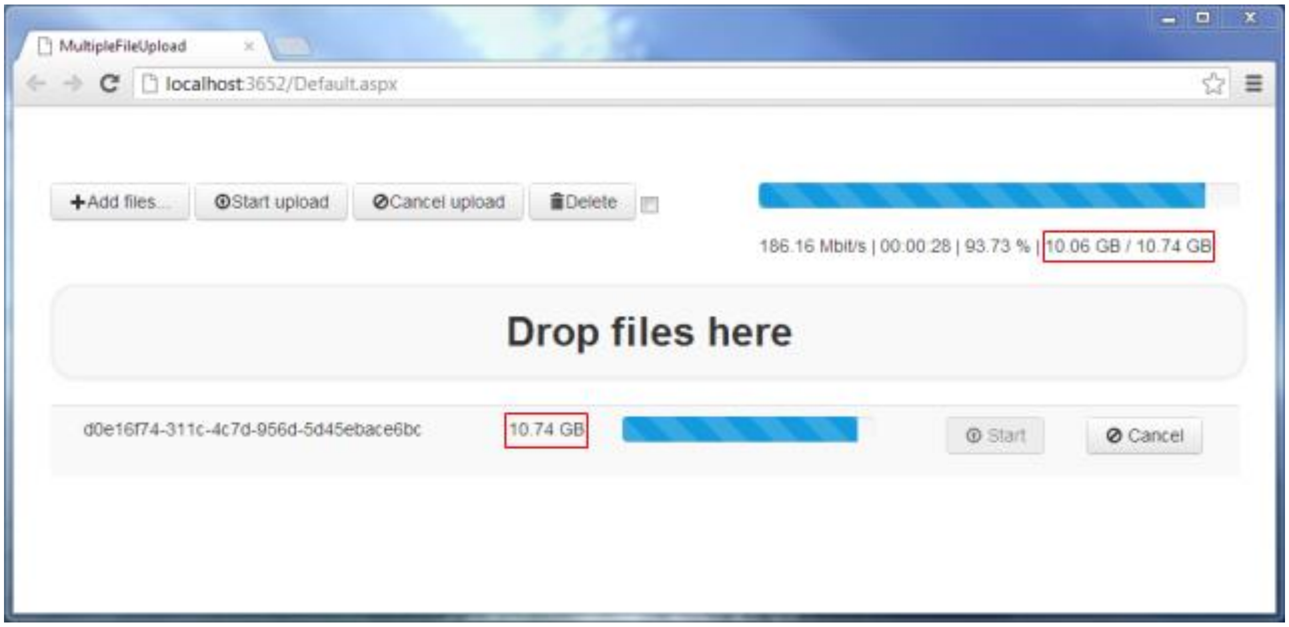
Ok, the explanation above is from a network perspective. But, what about CPU, I/O and RAM on the server?

Well, I also monitored the CPU, memory and I/O usage of the Web server. I took a capture through ProcessExplorer.exe when the upload was about finishing. Let's have a look at this capture:



We can notice that uploading a file of 10 GB consumes just a small amount of memory ~47.1 MB on the Web server. We can also notice through the graphs highlighted in red rectangles that each HTTP request taken by the `Upload.axd` HTTP handler consumes CPU and I/O to store the chunk on the Web server.

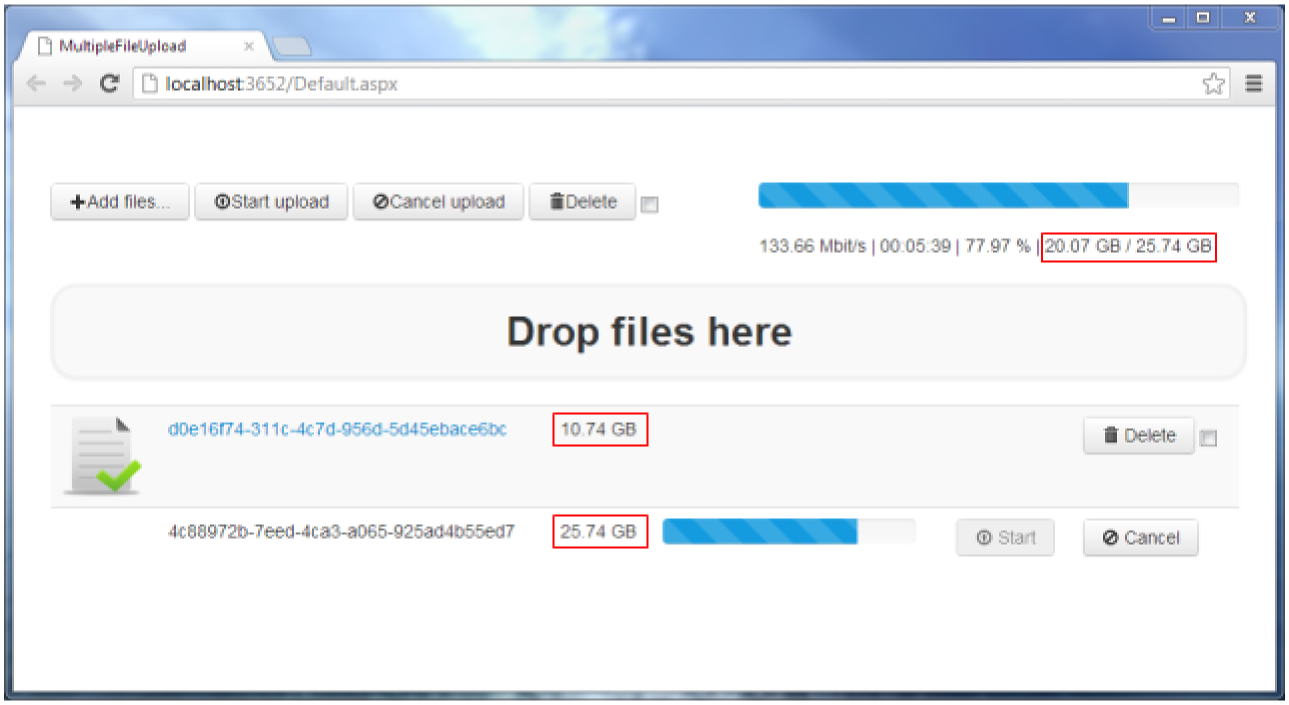
When upload were about to finish, I took a screenshot of the Web page:



186.16 Mbit/s | 00:00:28 | 93.73 % | 10.06 GB / 10.74 GB

d0e16f74-311c-4c7d-956d-5d45ebace6bc 10.74 GB

When upload finished, I launched another upload of a 25 GB size file:



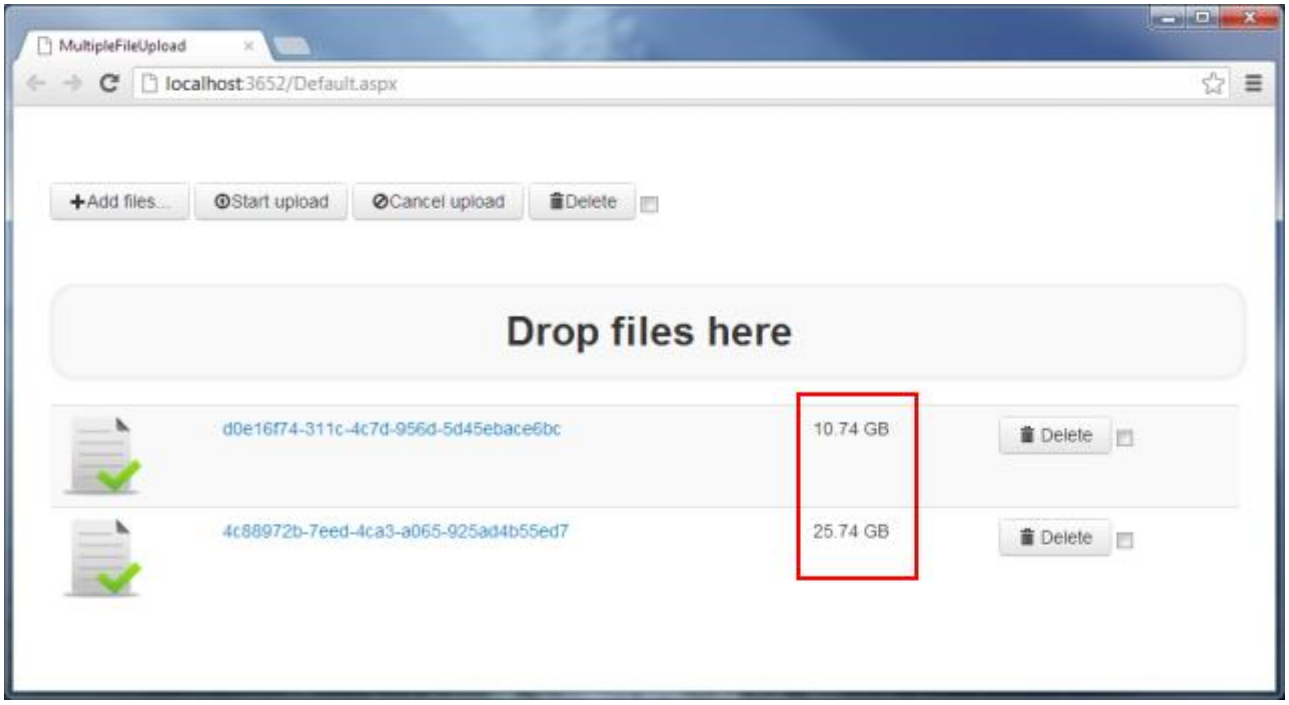
133.66 Mbit/s | 00:05:39 | 77.97 % | 20.07 GB / 25.74 GB

d0e16f74-311c-4c7d-956d-5d45ebace6bc 10.74 GB

4c88972b-7eed-4ca3-a065-925ad4b55ed7 25.74 GB

Likewise, the same behavior as the first.

When all uploads finished, I took another screenshot:



When chunked uploads are enabled and when the browser supports the HTML5 Blob API, don't worry about file size on large file upload. There's no limitation.

Chunked uploads are supported by Chrome, Mozilla Firefox 4+ and Microsoft Internet explorer 10.

Chunked uploads disabled

This case is more complicated. Indeed, as I said previously the only limitation in case of large file uploads when chunked uploads are disabled is the browser.

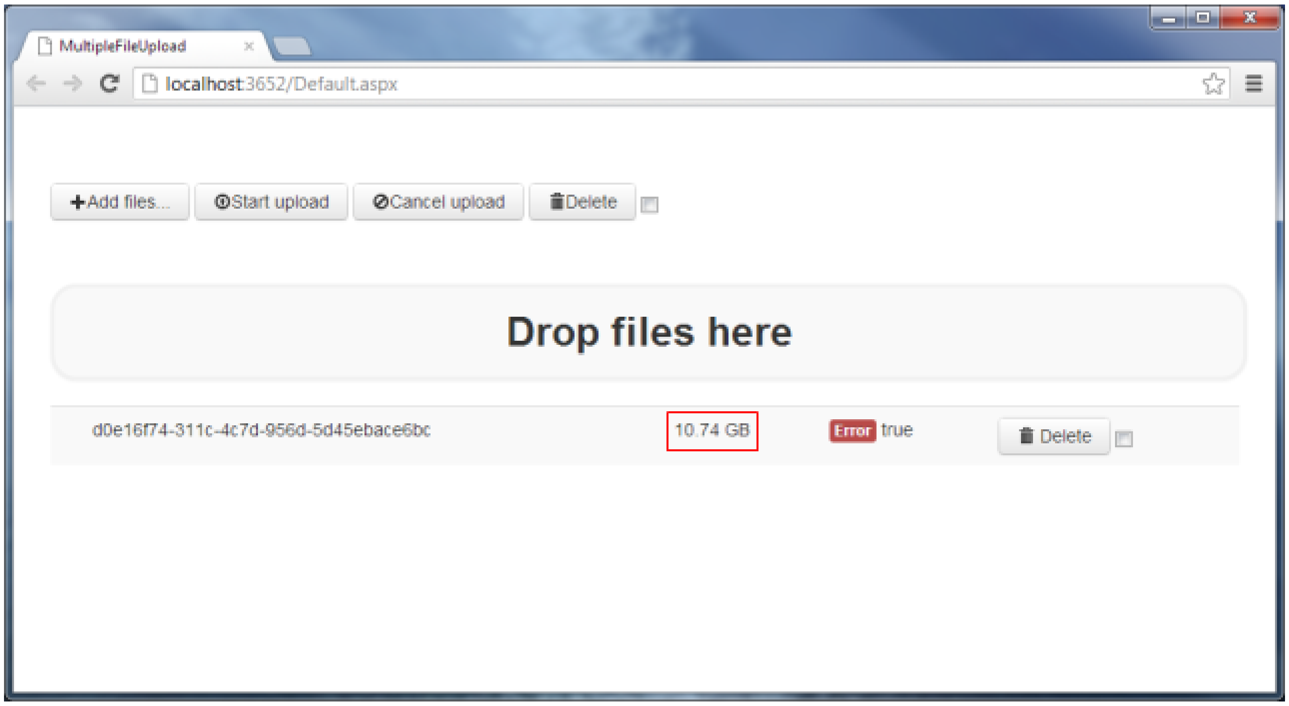
Well, in this case the maximum limit is 4 GB. Why? The jQuery File Upload plugin allows to upload > 4 GB, but this restriction comes from browsers when chunked uploads are disabled. This might be fixed in future updates to those browsers:

- Firefox has a current limit of 2^{32} bytes for file uploads, If the `Content-Length` of the upload post exceeds 2^{32} bytes, Firefox reports a false `Content-Length`:
  - https://bugzilla.mozilla.org/show_bug.cgi?id=660159

- Google Chrome reports the correct `Content-Length`, but continues to send 0's if a `Content-Length` of more than 2^{32} bytes has been exceeded:
  - http://code.google.com/p/chromium/issues/detail?id=139815

Similar browser restrictions might apply to other browsers.

If you disable chunked uploads and try to upload a file bigger than 4 GB on Google Chrome for example, you'll get the following:



d0e16f74-311c-4c7d-956d-5d45ebace6bc 10.74 GB Error true

Conclusion

In conclusion, if you enable chunked uploads and use Google Chrome, FireFox 4+, Microsoft Internet Explorer 10 or other browsers supporting the HTML5 Blob API you don't have to worry about the file size (10 GB, 25 GB, 100GB ...). I haven't tried the 100 GB and higher, but it should work, even bigger should work too (Tell me in the forum if you do it). On the other hand, when chunked uploads are disabled, the only limitation is the browser. And due to some restrictions that might be fixed in future updates, the maximum limit in this case is 4 GB.

HTTP protocol might not be the best protocol for uploading large files, It would be very interresting to implement a protocol on top of the HTTP upload layer of the browser in order to make large files uploads faster. The same approach have been adopted with FTP protocol (in the Open Source World, UFTP for instance) and other protocols.

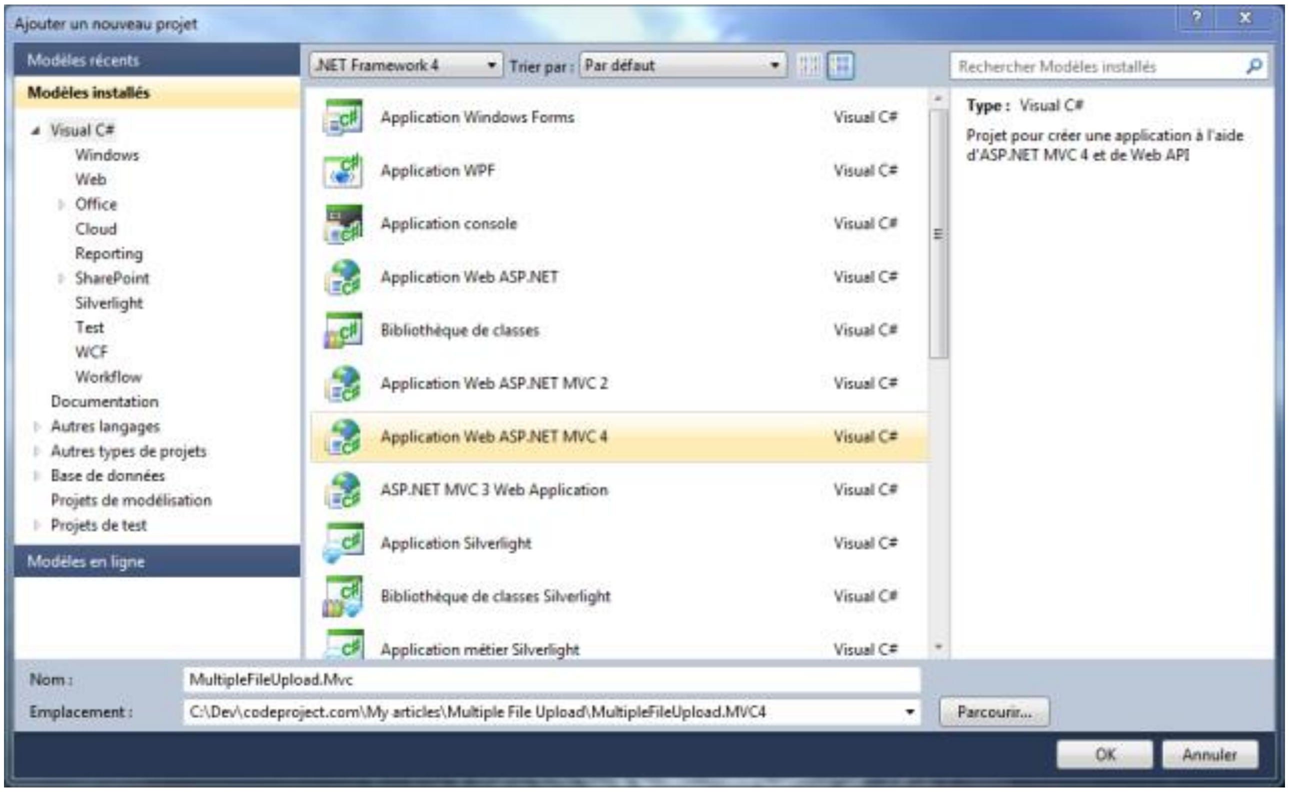
An in depth look in ASP .NET Web API and MVC 4

I had a new challenge in the forum. The challenge was about making a version with ASP .NET Web API. I accepted the challenge with pleasure and released a new version using the ASP .NET Web API and the ASP .NET MVC 4 framework.

In this section is described how the set up of this MVC 4 Web API version has been made from scratch.

Init

First of all, a new MVC 4 Web project was created:


Ajouter un nouveau projet
Modèles récents
Modèles installés
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Documentation
Autres langages
Autres types de projets
Base de données
Projets de modélisation
Projets de test
Modèles en ligne
.NET Framework 4
Trier par : Par défaut
Rechercher Modèles installés
Application Windows Forms Visual C#
Application WPF Visual C#
Application console Visual C#
Application Web ASP.NET Visual C#
Bibliothèque de classes Visual C#
Application Web ASP.NET MVC 2 Visual C#
Application Web ASP.NET MVC 4 Visual C#
ASP.NET MVC 3 Web Application Visual C#
Application Silverlight Visual C#
Bibliothèque de classes Silverlight Visual C#
Application métier Silverlight Visual C#
Parcourir...
OK
Annuler

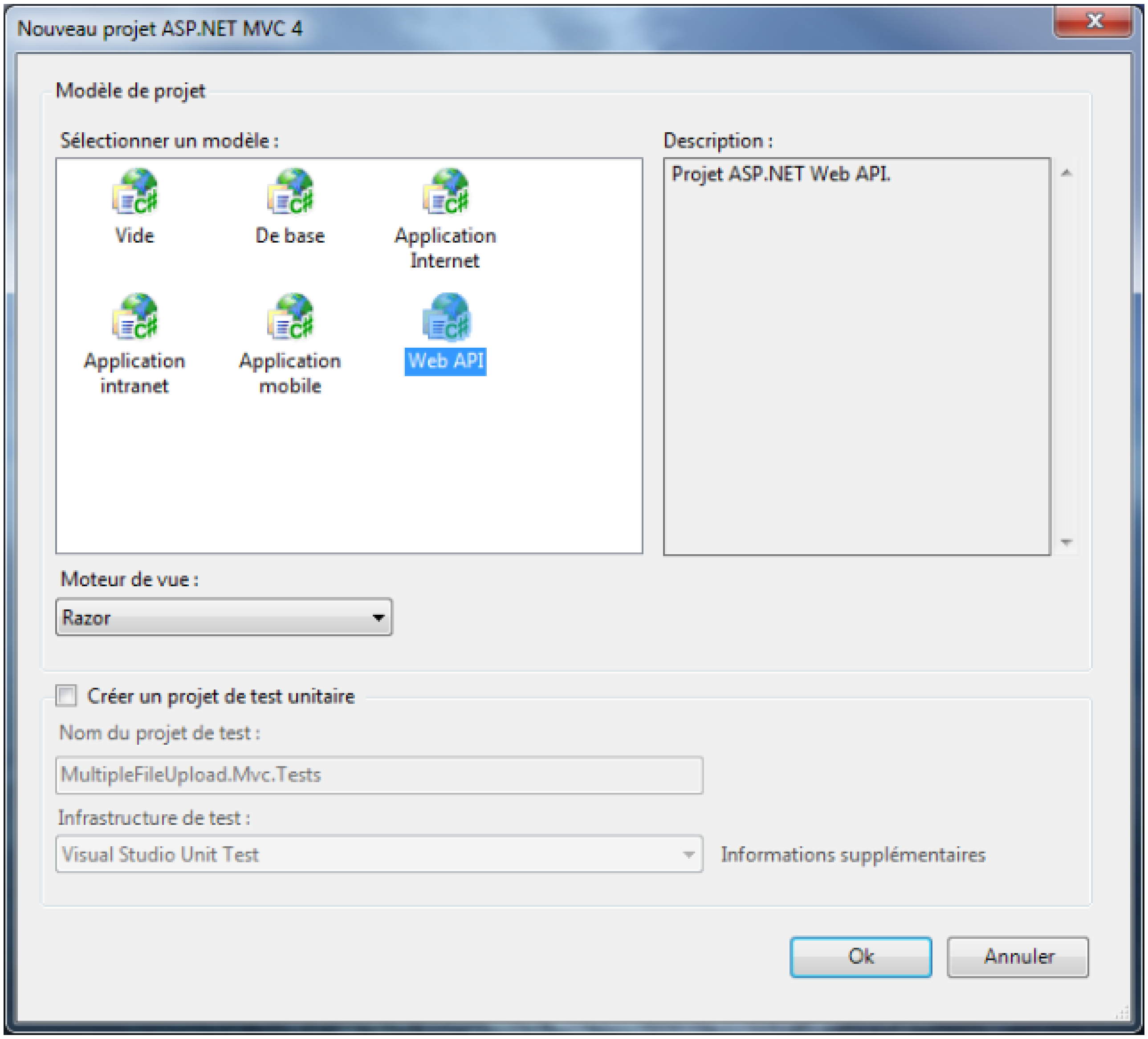
Type : Visual C#

Projet pour créer une application à l'aide d'ASP.NET MVC 4 et de Web API

Nom : MultipleFileUpload.Mvc

Emplacement : C:\Dev\codeproject.com\My articles\Multiple File Upload\MultipleFileUpload.MVC4

Then, the ASP .NET Web API model was selected:


Nouveau projet ASP.NET MVC 4

Modèle de projet

Sélectionner un modèle :

Description :

Projet ASP.NET Web API.

Moteur de vue :

Razor

Créer un projet de test unitaire

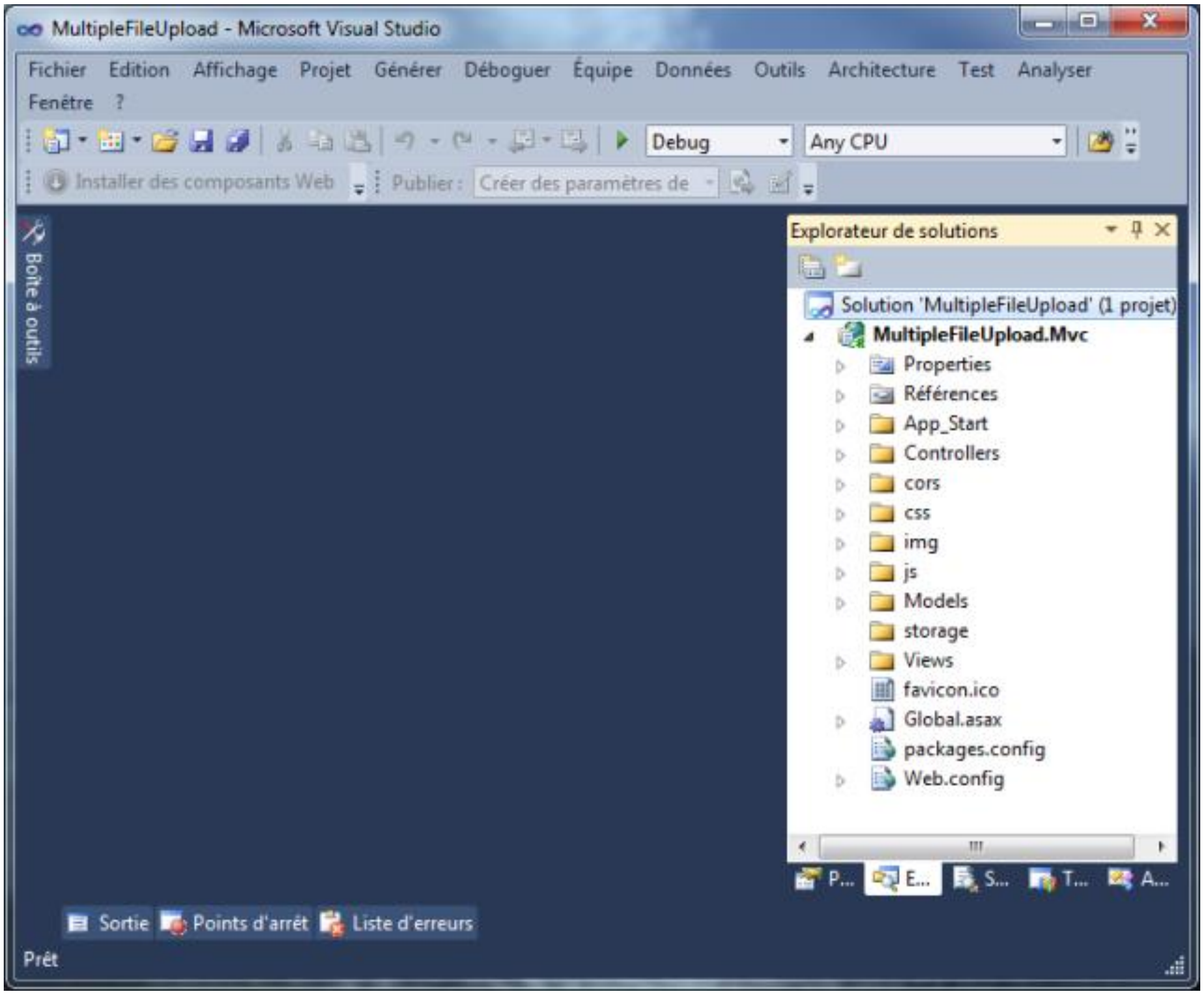
Nom du projet de test :

MultipleFileUpload.Mvc.Tests

Infrastructure de test :

Visual Studio Unit Test

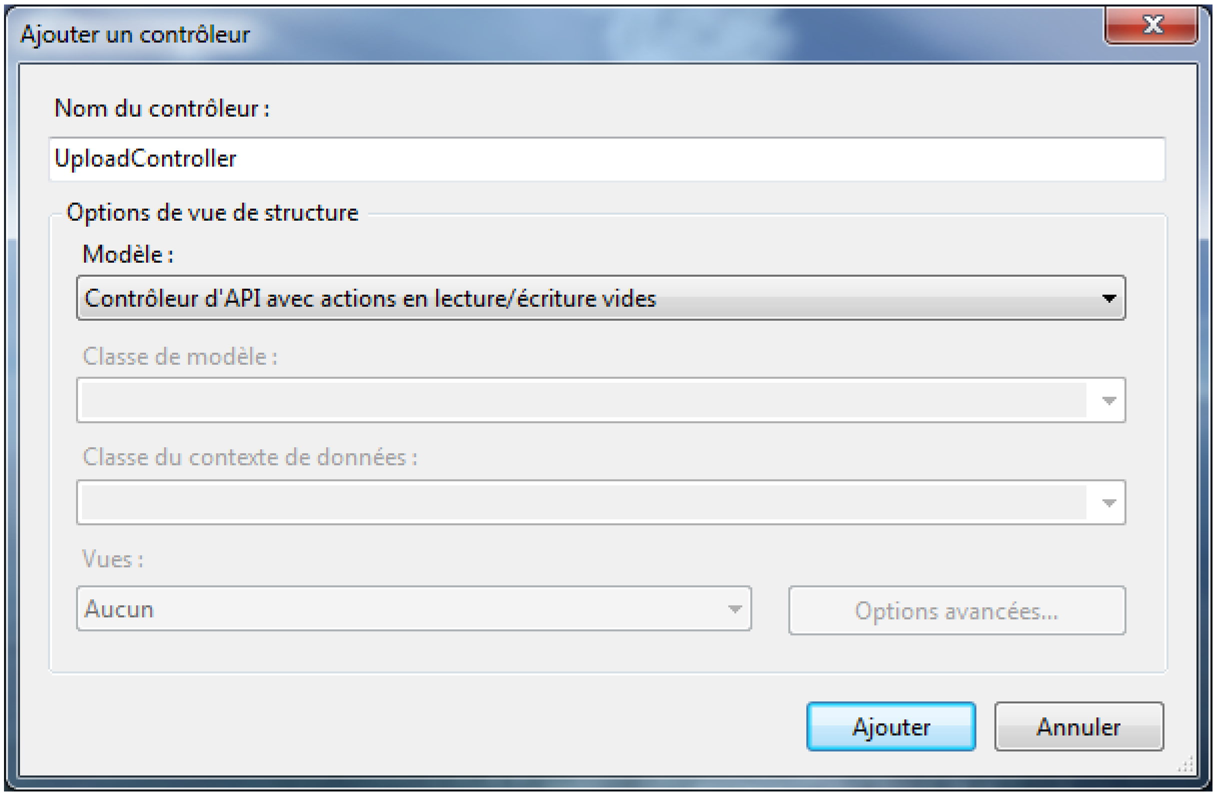
And finally the Visual Studio project was created. I added the necessary JavaScript libraries and stylesheets:


MultipleFileUpload - Microsoft Visual Studio
Fichier Edition Affichage Projet Générer Déboguer Équipe Données Outils Architecture Test Analyser Fenêtre ?
Debug
Any CPU
Installer des composants Web
Publier : Créer des paramètres de
Boîte à outils
Explorateur de solutions
Solution 'MultipleFileUpload' (1 projet)
MultipleFileUpload.Mvc
Properties
Références
App_Start
Controllers
cors
css
img
js
Models
storage
Views
favicon.ico
Global.asax
packages.config
Web.config
P... E... S... T... A...
Sortie
Points d'arrêt
Liste d'erreurs
Prêt

Controllers and Models

Ok, now we can start building the multiple file upload from scratch.

A new API controller `UploadController` has been added with the GET, POST, PUT, and DELETE actions:


Ajouter un contrôleur

Nom du contrôleur :

UploadController

Options de vue de structure

Modèle :

Contrôleur d'API avec actions en lecture/écriture vides

Classe de modèle :

Classe du contexte de données :

Vues :

Aucun

Remember in the `HttpHandler` approach, each action was handled through the `DoWork` method which runs specific jobs relying on the HTTP method. But now, no need to do all that. This is automatically handled trough the Restful architecture of the ASP .NET Web API.

Below the code the API controller at the beginning:

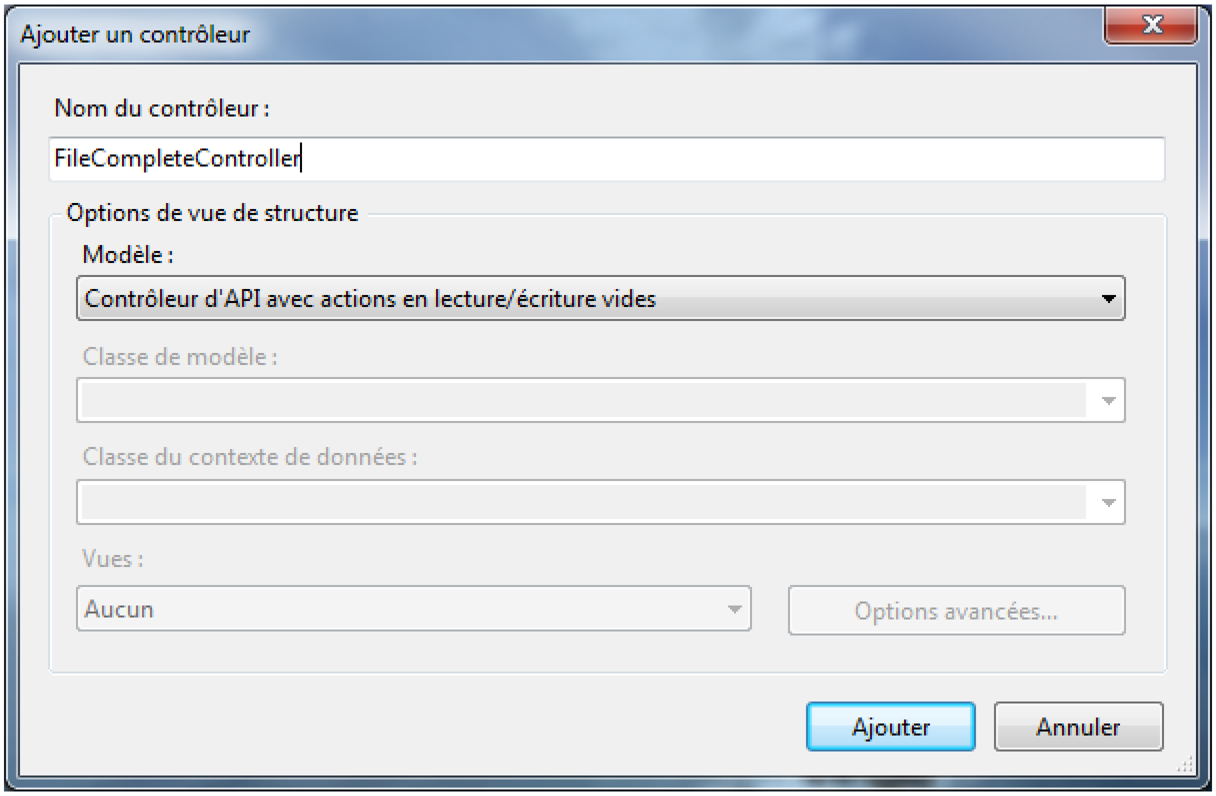
```
public class UploadController : ApiController
{
    public string StorageFolder
    {
        get
        {
            return HttpContext
                .Current
                .Server
                .MapPath(ConfigurationManager.AppSettings["StorageFolder"]);
        }
    }

    // GET api/upload
    public IEnumerable<FileStatus> Get()
    {
        throw new NotImplementedException();
    }

    // POST api/upload
    public IEnumerable<FileStatus> Post()
    {
        throw new NotImplementedException();
    }

    // PUT api/upload
    public IEnumerable<FileStatus> Put()
    {
        throw new NotImplementedException();
    }

    // DELETE api/upload
    public void Delete(string f)
    {
        throw new NotImplementedException();
    }
}
```

`FileStatus` is our Model and exposes the following properties:

```
public class FileStatus
{
    #region Constructors

    public FileStatus()
    {
    }

    public FileStatus(FileInfo fileInfo)
    {
        SetValues(fileInfo.Name, (int)fileInfo.Length);
    }

    public FileStatus(string fileName, int fileLength)
    {
        SetValues(fileName, fileLength);
    }

    #endregion

    #region Properties

    public string group { get; set; }
    public string name { get; set; }
    public string type { get; set; }
    public int size { get; set; }
    public string progress { get; set; }
    public string url { get; set; }
    public string complete_url { get; set; }
    public string error_url { get; set; }
    public string delete_url { get; set; }
    public string delete_type { get; set; }
    public string error { get; set; }

    #endregion

    #region Helpers
    private void SetValues(string fileName, int fileLength)
    {
        name = fileName;
        type = "image/png";
        size = fileLength;
        progress = "1.0";
        url = "api/upload?f=" + fileName;
        complete_url = "api/filecomplete?f=" + fileName;
        error_url = "api/fileerror?f=" + fileName;
        delete_url = "api/upload?f=" + fileName;
        delete_type = "DELETE";
    }

    #endregion

}
```

The implementation of `UploadController` will be described later.

A new API controller `FileCompleteController` has been added with the GET action in order to retrieve the thumbnail in case of successful uploads:



Nom du contrôleur : FileCompleteController

Modèle : Contrôleur d'API avec actions en lecture/écriture vides

Classe de modèle :

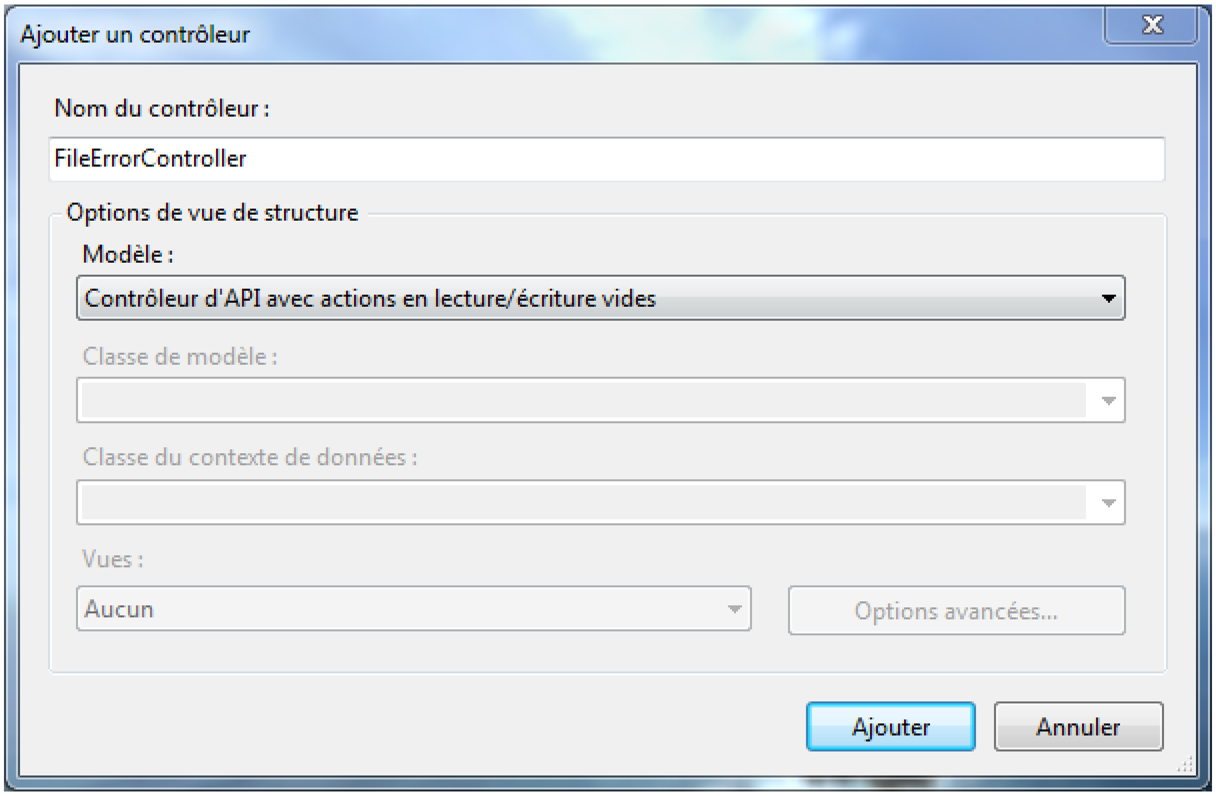
Classe du contexte de données :

Vues : Aucun

The implementation of this API controller is very simple and illustrated below:

```
public class FileCompleteController : ApiController
{
    // GET api/filecomplete
    public HttpResponseMessage Get(string f)
    {
        HttpResponseMessage response = new
HttpResponseMessage(HttpStatusCode.OK);
        response.Content = new StreamContent(
            new FileStream(HttpContext.Current.Server.MapPath("/img/file-
complete-icon.png"),
                            FileMode.Open, FileAccess.Read));
        response.Content.Headers.ContentType = new
MediaTypeHeaderValue("image/png");
        return response;
    }

}
```

A new API controller `FileErrorController` has been added with the GET action in order to retrieve the thumbnail in case of failed uploads:


Ajouter un contrôleur

Nom du contrôleur : FileErrorController

Options de vue de structure

Modèle : Contrôleur d'API avec actions en lecture/écriture vides

Classe de modèle :

Classe du contexte de données :

Vues : Aucun

The implementation of this API controller is very simple and illustrated below:

```
public class FileErrorController : ApiController
{
    // GET api/fileerror
    public HttpResponseMessage Get(string f)
    {
        HttpResponseMessage response = new
HttpResponseMessage(HttpStatusCode.OK);
        response.Content = new StreamContent(
            new FileStream(HttpContext.Current.Server.MapPath("/img/file-
error-icon.png"),
                FileMode.Open, FileAccess.Read));
        response.Content.Headers.ContentType = new
MediaTypeHeaderValue("image/png");
        return response;
    }
}
```

Now, the only thing that we need to implement to get our API controllers working is the `UploadController`. We will see the implementation of every action step by step.

Let's start with the simplest one. The GET. This action occurs when the view is loaded. Indeed, when the view is loaded, the files stored on the server storage folder are retrieved and listed in the view using Bootstrap layout. The implementation of this action is very simple and described below:

```
// GET api/upload
public IEnumerable<FileStatus> Get()
{
    return ListCurrentFiles();
}
```

`ListCurrentFiles` as its name tells, lists all the files stored in the server storage folder and is implemented as follows:

```
private IEnumerable<FileStatus> ListCurrentFiles()
{
    FileStatus[] statuses =
        (from file in new DirectoryInfo(StorageFolder).GetFiles("*",
SearchOption.TopDirectoryOnly)
            where !file.Attributes.HasFlag(FileAttributes.Hidden)
            select new FileStatus(file)).ToArray();
    return statuses;
}
```

Remember in the `HttpHandler` approach, data was serialized in JSON through `JavaScriptSerializer` and written in the HTTP response. Now, no need to do all that, the ASP .NET Web API automatically serializes the collection of `FileStatus` and sends it.

Below the implementation of the POST and PUT actions:

```
// POST api/upload
public IEnumerable<FileStatus> Post()
{
    return UploadFile();
}

// PUT api/upload
public IEnumerable<FileStatus> Put()
{
    return UploadFile();
}
```

The `UploadFile` method uploads files and returns a collection of `FileStatus`:

```
private IEnumerable<FileStatus> UploadFile()
{
    List<FileStatus> statuses = new List<FileStatus>();
    HttpRequestHeaders headers = Request.Headers;
    IEnumerable<string> values;
    HttpContext context = HttpContext.Current;

    if (!headers.TryGetValues("X-File-Name", out values))
    {
        UploadWholeFile(context, statuses);
    }
    else
    {
        UploadPartialFile(values.FirstOrDefault(), context, statuses);
    }

    return statuses;
}
```

`UploadWholeFile` and `UploadPartialFile` didn't change. The file upload can be done in another style using asynchronous tasks, I added it to my todo list.

Ok, now we can upload files, list the files already uploaded. The last thing that we need to do is to implement the DELETE action. The DELETE action deletes the file from the storage folder as follows:

```
// DELETE api/upload
public void Delete(string f)
{
    DeleteFile(f);
}
```

The file name is retrieved from query strings and passed to `DeleteFile`:

```
// Delete file from the server
private void DeleteFile(string f)
{
    string filePath = Path.Combine(StorageFolder, f);
    if (File.Exists(filePath))
    {
        File.Delete(filePath);
    }
}
```

That's it! we're done. We have all the controllers implemented.

Let's build our MVC 4 View.

View

First of all, let's build our JavaScript and stylesheets bundles:

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        // Add CSS bundle
        bundles.Add(new
StyleBundle("~/bundles/css").IncludeDirectory("~/css/","*.css"));

        // Add JS bundle
        bundles.Add(new ScriptBundle("~/bundles/js").Include("~/js/jquery.js"
                                                    ,
"~/js/vendor/jquery.ui.widget.js"
                                                    , "~/js/tmpl.js"
                                                    , "~/js/load-image.js"
                                                    , "~/js/canvas-to-
blob.js"
                                                    , "~/js/bootstrap.js"
                                                    , "~/js/bootstrap-image-
gallery.js"
                                                    , "~/js/jquery.iframe-
transport.js"
                                                    ,
"~/js/jquery.fileupload.js"
                                                    ,
"~/js/jquery.fileupload-fp.js"
                                                    ,
"~/js/jquery.fileupload-ui.js"
                                                    , "~/js/locale.js"));
        bundles.Add(new
ScriptBundle("~/bundles/main").IncludeDirectory("~/js/main", "*.js"));
    }
}
```

Then register them:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    @Styles.Render("~/bundles/css")
    <!-- Bootstrap CSS fixes for IE6 -->
    <!--[if lt IE 7]><link href="css/bootstrap-ie6.min.css" rel="stylesheet"
type="text/css" /><![endif]-->
    <!-- Shim to make HTML5 elements usable in older Internet Explorer
versions -->
    <!--[if lt IE 9]><script src="js/html5.js"
type="text/javascript"></script><![endif]-->
</head>
<body>
```

```
    @RenderBody()
    @Scripts.Render("~/bundles/js")
    <!-- The XDomainRequest Transport is included for cross-domain file
deletion for IE8+ -->
    <!--[if gte IE 8]><script src="js/cors/jquery.xdr-
transport.js"></script><![endif]-->
    @Scripts.Render("~/bundles/main")
</body>
</html>
```

And finnaly our view:

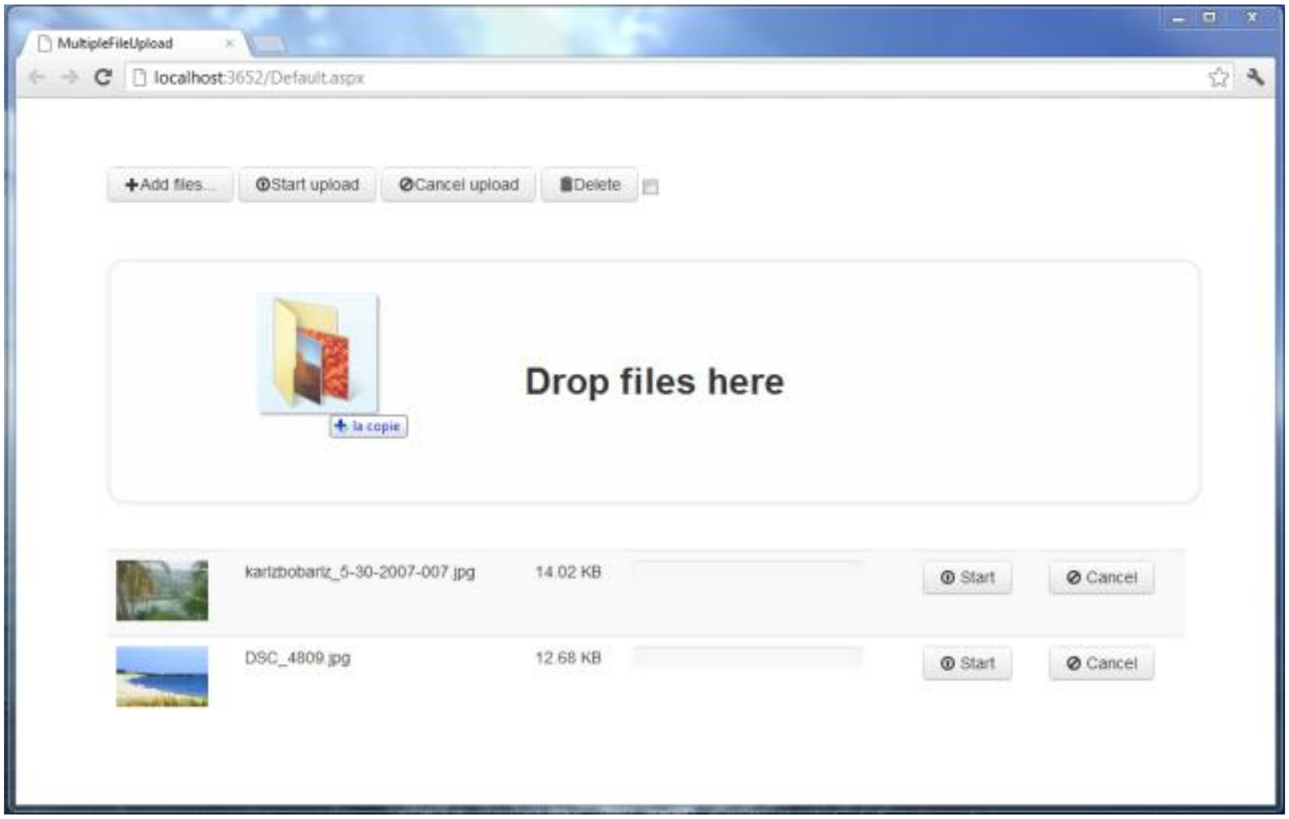
```
@{
    ViewBag.Title = "Multiple File Upload";
}
<div class="container">
    <div id="fileupload">
        <!-- The fileupload-buttonbar contains buttons to add/delete files
and start/cancel the upload -->
        <div class="row fileupload-buttonbar">
            <div class="span7">
                <!-- The fileinput-button span is used to style the file
input field as button -->
                <span class="btn btn-action fileinput-button"><i class="icon-
plus"></i><span>Add files...</span>
                    <!--Enable selecting a complete folder structure, this is
currently only supported by Google Chrome-->
                    <input type="file" name="files[]" multiple>
                </span>
                <button type="submit" class="btn btn-action start">
                    <i class="icon-upload"></i><span>Start upload</span>
                </button>
                <button type="reset" class="btn btn-action cancel">
                    <i class="icon-ban-circle"></i><span>Cancel upload</span>
                </button>
                <button type="button" class="btn btn-action delete">
                    <i class="icon-trash"></i><span>Delete</span>
                </button>
                <input type="checkbox" class="toggle">
            </div>
            <!-- The global progress information -->
            <div class="span5 fileupload-progress fade">
                <!-- The global progress bar -->
                <div class="progress progress-striped active"
role="progressbar" aria-valuemin="0"
                    aria-valuemax="100">
                    <div class="bar" style="width: 0%;">
                    </div>
                </div>
                <!-- The extended global progress information -->
                <div class="progress-extended">
                    &nbsp;</div>
            </div>
        </div>
        <!-- The loading indicator is shown during file processing -->
        <div class="fileupload-loading">
        </div>
        <br />
        <div id="dropzone" class="fade well">
            Drop files here</div>
        <br />
        <!-- The table listing the files available for upload/download -->
        <table role="presentation" class="table table-striped">
            <tbody class="files" data-toggle="modal-gallery" data-
target="#modal-gallery">
            </tbody>
        </table>
    </div>
</div>
<!-- The template to display files available for upload -->
<script id="template-upload" type="text/x-tmpl">
{% for (var i=0, file; file=o.files[i]; i++) { %}
    <tr class="template-upload fade">
        <td class="preview"><span></span></td>
        <td class="name"><span>{%=file.name%}</span></td>
        <td class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
        {% if (file.error) { %}
            <td class="error" colspan="2"><span class="label label-
important">
            {%=locale.fileupload.error%}</span>
{%=locale.fileupload.errors[file.error] || file.error%}</td>
        {% } else if (o.files.valid && !i) { %}
            <td>
                <div class="progress progress-striped active"
role="progressbar" aria-valuemin="0" aria-valuemax="100" aria-
valuenow="0"><div class="bar" style="width:0%;"></div></div>
            </td>
            <td class="start">{% if (!o.options.autoUpload) { %}
                <button class="btn btn-action">
                    <i class="icon-upload"></i>
                    <span>{%=locale.fileupload.start%}</span>
                </button>
            {% } %}</td>
        {% } else { %}
            <td colspan="2"></td>
        {% } %}
        <td class="cancel">{% if (!i) { %}
            <button class="btn btn-action">
                <i class="icon-ban-circle"></i>
                <span>{%=locale.fileupload.cancel%}</span>
            </button>
        {% } %}</td>
    </tr>
{% } %}
</script>
<!-- The template to display files available for download -->
<script id="template-download" type="text/x-tmpl">
{% for (var i=0, file; file=o.files[i]; i++) { %}
    <tr class="template-download fade">
        {% if (file.error) { %}
            <td class="preview">
                <img src="{%=file.error_url%}">
            </td>
            <td class="name"><span>{%=file.name%}</span></td>
            <td
class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
            <td class="error" colspan="2"><span class="label label-
important">
            {%=locale.fileupload.error%}</span>
{%=locale.fileupload.errors[file.error] || file.error%}</td>
        {% } else { %}
            <td class="preview">{% if (file.complete_url) { %}
                <img src="{%=file.complete_url%}">
            {% } %}</td>
            <td class="name">
                {%=file.name%}
            </td>
            <td
class="size"><span>{%=o.formatFileSize(file.size)%}</span></td>
            <td colspan="2"></td>
        {% } %}
        <td class="delete">
            <button class="btn btn-action" data-type="{%=file.delete_type%}"
data-url="{%=file.delete_url%}">
                <i class="icon-trash"></i>
                <span>{%=locale.fileupload.destroy%}</span>
            </button>
            <input type="checkbox" name="delete" value="1">
        </td>
    </tr>
{% } %}
</script>
```

And the main script:

```
/*jslint nomen: true, unparam: true, regexp: true */
/*global $, window, document */
$(function () {

    'use strict';

    var api = 'api/upload';

    //
    // Initialize the jQuery File Upload widget
    //
    $('#fileupload').fileupload({
        url: api,
        dropZone: $('#dropzone'),
        sequentialUploads: true
    });

    //
    // Enable iframe cross-domain access via redirect option
    //
    $('#fileupload').fileupload(
                    'option',
                    'redirect',
                    window.location.href.replace(/\/[^\/]*$/,
'/cors/result.html?%s')
            );

    //
    // Load existing files
    //
    $('#fileupload').each(function () {
        var that = this;
        $.getJSON(api, function (result) {
            if (result && result.length) {
                $(that).fileupload('option', 'done')
                        .call(that, null, { result: result });
            }
        });
    });

    //
    // Callbacks (if you want to use them refer to the documentation)
    //
    $('#fileupload')
            .bind('fileuploadadd', function (e, data) { /* ... */ })
            .bind('fileuploadsubmit', function (e, data) { /* ... */ })
            .bind('fileuploadsend', function (e, data) { /* ... */ })
            .bind('fileuploaddone', function (e, data) { /* ... */ })
            .bind('fileuploadfail', function (e, data) { /* ... */ })
            .bind('fileuploadalways', function (e, data) { /* ... */ })
            .bind('fileuploadprogress', function (e, data) { /* ... */ })
            .bind('fileuploadprogressall', function (e, data) { /* ... */ })
            .bind('fileuploadstart', function (e) { /* ... */ })
            .bind('fileuploadstop', function (e) { /* ... */ })
            .bind('fileuploadchange', function (e, data) { /* ... */ })
            .bind('fileuploadpaste', function (e, data) { /* ... */ })
            .bind('fileuploaddrop', function (e, data) { /* ... */ })
            .bind('fileuploaddragover', function (e) { /* ... */ });
});

//
//  dragover event
//
$(document).bind('dragover', function (e) {
    var dropZone = $('#dropzone'), timeout = window.dropZoneTimeout;
    if (!timeout) {
        dropZone.addClass('in');
    } else {
        clearTimeout(timeout);
    }
    if (e.target === dropZone[0]) {
        dropZone.addClass('thumbnail hover');
    } else {
        dropZone.removeClass('hover');
    }
    window.dropZoneTimeout = setTimeout(function () {
        window.dropZoneTimeout = null;
        dropZone.removeClass('in hover');
    }, 100);
});
```

That's it!

Our brand new MVC 4 application is finished and ready for action:



References

- JQuery File Upload documentation.
- JQuery File Upload options reference.
- JavaScript Templates documentation.
- IHttpHandler based example from Iain Ballard.
- http://www.codeproject.com/script/Articles/ArticleVersion.aspx?aid=460142&av=669590