# Communications API

## Overview

In this lab, you’ll implement a Web application that allows users to upload multiple files to the Web server simultaneously. Bulk upload is a common requirement in many Web applications.

The user can select as many files as they want to upload; the application will buffer the file data client-side until the user is ready to upload the files. When the user starts the upload operation, the application will create a series of simultaneous Ajax requests that will post all the files to the server in parallel.

The application will take advantage of the new features in XMLHttpRequest Level 2 to receive progress events during each file upload operation, to give the user visual feedback about how each file is going.

## Source folders

* C:\Html5Css3JsDev\Student\15-CommunicationsApi
* C:\Html5Css3JsDev\Solutions\15-CommunicationsApi

## Roadmap

There are 7 exercises in this lab, of which the last exercise is "if time permits". Here's a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

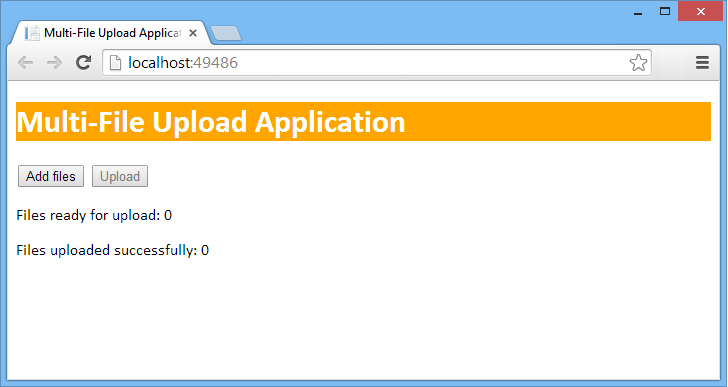
1. Familiarization with the solution application
2. Understanding the server-side starter code
3. Understanding the client-side starter code
4. Preparing form data for a file upload operation
5. Uploading form data to the server
6. Running the application
7. Additional suggestions (if time permits)

## Exercise 1: Familiarization with the solution application

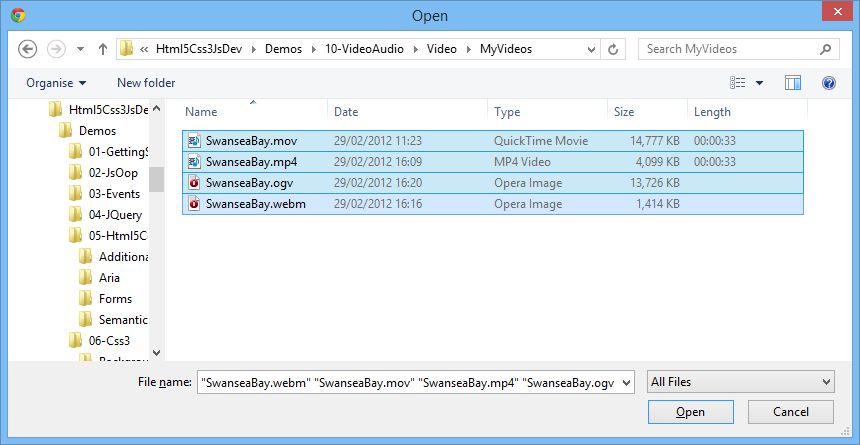
Start Visual Studio and open the *Solution* project for this lab, to get an idea of what you’re aiming for. Here are the details of the project:

* *Folder:*C:\Html5Css3JsDev\Solutions\15-CommunicationsApi\MultiFileUploadApp
* *Project name:*MultiFileUploadApp.sln

## Build and run the application. The web page appears as follows:



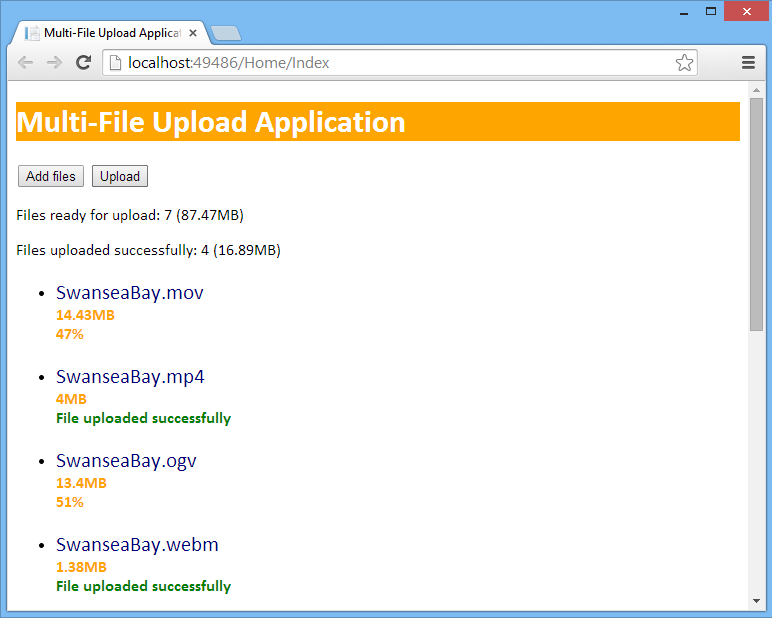
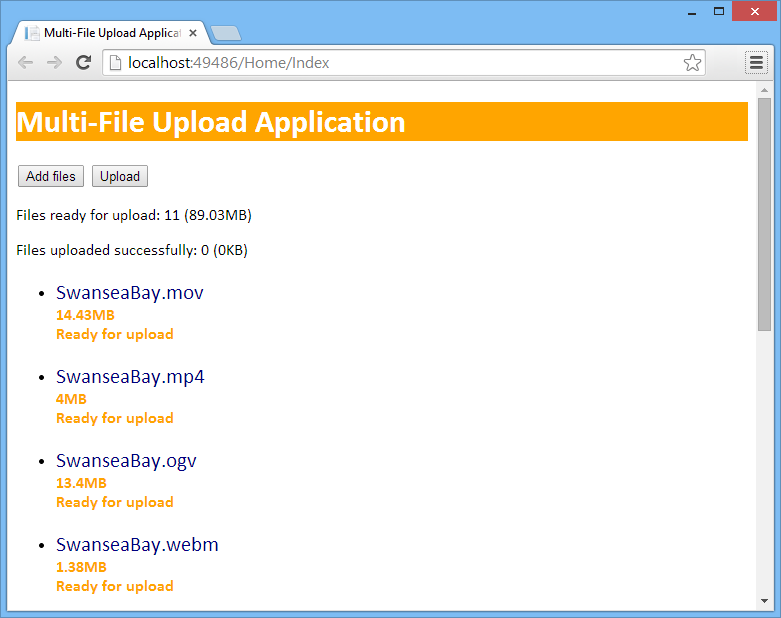
## The idea of the application is that it allows the user to upload multiple files from the client computer to the server. Click the Add files button and select several files. For example:



Select a lot of large files (maximum 20MB per file), so that the file-upload operation will take a measurable amount of time.

## When you've selected the files, the web page displays the details for the files in a bulleted list. For each file, it displays the file name, file size, and status (this is "Ready for upload" for each file initially). Near the top of the page, it also indicates how many files are ready for upload, and their total size. Also notice that the Upload button is now enabled, because there are files ready for upload:

## Click the Upload button, to commence uploading files to the server. Each file is uploaded as a separated Ajax HTTP POST request. The application makes use of XMLHttpRequest Level 2 upload progress events, so you can see the progress of each of the simultaneous file upload requests. (Note: The file upload operation is very quick, so you'll need to upload quite a lot of large files in order for the whole operation to take long enough for you to observe the progress update messages):



As each individual file upload is completed, the message "File uploaded successfully" is displayed in green for that file, and the statistics at the top of the web page are updated to indicate how many files (and how many bytes) have been uploaded so far. If any errors occur, a red error message is displayed. After all the files have been uploaded, the *Upload* button is disabled again – you can click *Add files* again if you want to upload more files.

When you're happy with how the solution application works, close the browser. Open File Explorer and go to the following folder:

C:\Html5Css3JsDev\Solutions\15-CommunicationsApi

\MultiFileUploadApp\MultiFileUploadApp\App\_Data

This is the server folder where the server stores all the uploaded files. Verify it contains all of the files you uploaded from the client browser.

**Exercise 2: Understanding the server-side starter code**

Go back to Visual Studio, and open the *Student* project for this lab:

* *Folder:*C:\Html5Css3JsDev\Student\15-CommunicationsApi\MultiFileUploadApp
* *Project name:*MultiFileUploadApp.sln

## Build and run the application. The web page appears as before. Try the following operations:

## Click the Add files button and select some files. Notice that the file selection functionality still works as before, i.e. the application allows you to select multiple files and displays them as a bulleted list on the web page.

* Now click the *Upload* button. A message box appears, indicating that the file upload functionality is not implemented yet. You'll implement this functionality during the lab, making use of XMLHttpRequest Level 2 file upload techniques.

Close the browser and return to Visual Studio.

We'll explain how the existing parts of the application work first, and then point out the bits you need to add yourself. The server-side functionality is already complete, i.e. we've written all the server-side code (in C#) to receive a file from the client browser, and to save it in the App\_Data folder at the server. To see this code, follow these steps in Visual Studio:

* In Solution Explorer, expand the Controllersfolder, and then double-click HomeController.cs. This is a C# class that handles all the requests from a client browser.
* The Index() method simply returns the main (only) web page in the application. We'll discuss how the web page works in Exercise 3.
* The UploadFile() method receives a file posted from the client (the client triggers this method by sending an Ajax HTTP POST request to the URL /Home/UploadFile). The client will pass two parameters in the HTTP request:
  + filename  
    This parameter indicates the name of the file on the client computer.
  + file  
    This parameter contains the contents of the file, posted from the client computer.

The UploadFile() method saves the file contents in the App\_Data folder in the server application, using the same filename as on the client computer. The method returns an "OK" HTTP status code (200) if everything is OK, or a "server error" HTTP status code (500) if any error occurs.

**Exercise 3: Understanding the client-side starter code**

In Solution Explorer, expand the Viewsfolder, expand the Homesub-folder, and then double-click Index.cshtml. This is the main (only) view page for the web application.

*Aside*: .cshtml files can contain a mixture of server-side C# code (to generate dynamic parts of a web page, e.g. from a database) and client-side HTML/JavaScript/.CSS for display in the browser. In our case, apart from the @{…} code on the first 3 lines, it's all regular client-side content, just like a normal .html page.

The first part of Index.cshtml contains the HTML mark-up for the web page. The HTML mark-up is already complete. Note the following points:

* The page has buttons named addFilesButton and uploadButton, which you've seen already. The user can click these buttons to select files, and then upload them to the server.
* There are <span> element named readyToUploadSummary and uploadedSummary, which display the number (and total size) of files ready for upload, plus the number (and total size) of files already uploaded.
* There's a file input element named fileChooser, which allows the user to choose multiple files for upload.

We've given this element a CSS class named 'hidden', which sets the display: none style to hide the file input element from view. The reason we've done this is because we want to benefit from the functionality of the file input element (i.e. we want the user to be able to choose files) but we don't want to actually display the file input element itself (because it's ugly and it looks unprofessional). We'll explain how we achieved this effect shortly.

* After the file input element, there's a <ul> element named fileList. This will display a bulleted list of all the files selected by the user. The list already contains a single <li> element with a CSS class named fileEntryTemplate, which sets the display: none style for the <li> element. Every time the user selects a file, we make a clone of this element, populate its <span> elements with information about the file, and then make it visible.

The rest of Index.cshtml contains JavaScript code to manage the user interface and to perform Ajax file upload operations to the server. All the boilerplate code is already complete, and we'll describe it first. The only bit you'll need to complete is the uploadFile() function, which we'll describe in detail in Exercises 4 and 5.

Here's a quick run-through of the existing JavaScript code. There are some interesting techniques here, so it's worth spending a few moments understanding how it works:

* First, note the global variable named filesToUpload. This is a JavaScript object, and it'll behave like a map (i.e. dictionary) containing information about all the files to upload.

Every time the user selects a file to upload, we add a new property to the filesToUpload object. The name of the property will indicate the filename, and the value of the property will be a JavaScript object describing that file.

For example, imagine the user selects two files for upload, named File1.dat and File2.dat. The filesToUpload object will contain two properties as follows:

|  |  |
| --- | --- |
| Property name | Value |
| **'File1.dat'** | JavaScript object containing the following info about File1.dat:   * **file** property Contains the contents of the File1.dat file * **statusMessageElement** property Points to a <span> element on the web page where upload status messages will be displayed for the File1.dat file upload operation * **uploadComplete** property Indicates whether File1.dat has been completely uploaded yet (false initially) |
| **'File2.dat'** | JavaScript object containing the following info about File2.dat:   * **file** property Contains the contents of the File2.dat file * **statusMessageElement** property Points to a <span> element on the web page where upload status messages will be displayed for the File2.dat file upload operation * **uploadComplete** property Indicates whether File2.dat has been completely uploaded yet (false initially) |
| … etc. … |  |

* Next up in the JavaScript code, the $(document).ready event-handler function sets up event handlers for the web UI elements. Note the following points:
  + When the user clicks the *Add files* button, we simulate a click event on the real file input control (i.e. the fileChooser element). It's as if the user had actually clicked the file input control itself, i.e. it causes the file selection dialog box to appear. In this way, we benefit from the functionality of the file input element, without having to see its ugliness on our web page.
  + When the user completes his/her file selection, the file input element fires a change event. We handle this change event, and for each file the user has selected, we call a helper function named addFileToUploadList().

addFileToUploadList() is listed a bit further down in the JavaScript code. Take a look if you're interested – there are plenty of comments explaining how it works. The net effect is that it adds another property to the filesToUpload object, to capture information about a file to be uploaded.

* + When the user has selected all the files they're interested in and clicks the *Upload* button, we handle the click event on this button. We iterate through all the entries in the filesToUpload object. For each file that hasn't already been uploaded, we call a helper function named uploadFile(). You will implement this function in Exercises 4 and 5.
* Lower down in the JavaScript code, the displaySummaryPanel(), displayMessage(), and getFileSizeString() functions contain miscellaneous helper code used throughout the web page. Feel free to peruse these functions if you're interested – again, the comments explain how each function works.

**Exercise 4: Preparing form data for a file upload operation**

In this exercise you'll start implementing the the uploadFile() JavaScript function, so locate this function now in Index.cshtml.

The function receives a filename parameter that indicates the name of a file to upload. You can use this as a key into filesToUpload, i.e. filesToUpload[filename] gives you back a JavaScript object with the following properties for the file to be uploaded (as discussed earlier):

* file  
  Contains the contents of the file, which are to be uploaded to the server.
* statusMessageElement  
  Points to a <span> element on the web page where you will display status messages during the file upload operation.
* uploadComplete   
  Indicates whether the file has been completely uploaded yet. You will set this to true when the file upload has completed successfully.

You'll implement the first part of uploadFile() in this exercise, where you'll prepare form data for the file upload operation. You'll complete the steps indicated by the TODO comments 4a) and 4b) in the code. Here are some additional hints and tips to explain what you need to do:

1. The first step is to create a new FormData JavaScript object:

var fd = new FormData();

FormData is a new JavaScript interface introduced in the XMLHttpRequest Level 2 standard. It allows you to package up an HTTP form of data to send to the server. It simulates a real HTML form in your web page, as if you wrote this:

<form … >

… input fields to be posted to the server …

</form>

FormData allows you to package up form data programmatically. This is the ideal way to embed a JavaScript File object in the HTTP request. For full info about the FormData interface, see [https://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html#interface-formdata](https://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html%23interface-formdata).

1. Add parameters to the FormData object, to hold the filename and file parameters expected by the server. This is the code you need:

fd.append('filename', filename);

fd.append('file', filesToUpload[filename].file);

The FormData object now contains all the data it needs, ready to be sent to the server. You'll complete this functionality in the next exercise.

**Exercise 5: Uploading form data to the server**

In this exercise you'll complete the implementation of the uploadFile() function, to send form data to the server in an Ajax HTTP POST request.

You'll complete the steps indicated by the TODO comments 5a) to 5f). All your code will reside in an "immediately invoked function expression" (IIFE). An IIFE defines a private scope, and it allows us to define some private variables just for this particular Ajax request:

* statusMessageElement  
  This private variable points to the <span> element on the web page, where we will display upload progress events for this particular file upload operation.
* fileBeingUploaded  
  This private variable contains the actual file data to be uploaded in this Ajax request.
* xhr  
  This is the XMLHttpRequest object that we'll use to do the Ajax call.

You will now complete the function, as indicated by the TODO comments in the code. Here are some additional hints, tips, and explanations for each step:

1. The first step is to handle the load event on the xhr object. This event always occurs when an Ajax request has completed (whether it was a GET or a POST, and whether the request succeeded or failed). You need to handle this event to see if the whole request returned a status code other than 200, indicating that an error occurred.

Here's how to handle the load event on the xhr object:

xhr.addEventListener(

'load',

function (e) {

…

}

});

In the event-handler function, e.target represents the XMLHttpRequest object. Test if its readyState property is 4, and if so test if the status property is something other than 200. This indicates the Ajax request to upload the current file has failed, so you need to display an error message for this file. You can do this as follows:

displayMessage(statusMessageElement,

'Error occurred, file not uploaded',

'errorMessage');

1. The next step is to handle progress events during the file upload operation. The ability to handle progress events during an Ajax upload operation is new in XMLHttpRequest Level 2, and is made possible via the upload property on the XMLHttpRequest object, i.e. xhr.upload in our code.

The upload property returns an XMLHttpRequestUpload object, and this object has events such as progress, load, error, and abort that allow us to monitor the progress of an Ajax upload operation.

Here's how to handle progress events during an upload:

xhr.upload.addEventListener(

'progress',

function (e) {

…

}

});

In the event-handler function, e is a ProgressEvent object and has three properties describing the upload progress:

* lengthComputable  
  Indicates whether the length of the whole HTTP upload body is known, in which case we can expect some meaningful values for the next two properties…
* loaded  
  Number of bytes uploaded to the server so far.
* total  
  Total number of bytes to be uploaded to the server (i.e. file size).

Using this information, implement the event-handler function so that it displays a meaningful message to indicate the progress of the current file upload (e.g. the percentage complete, or a generic message such as *Uploading…* if the length is not computable).

1. Next, implement the load event on the xhr.upload object. This event indicates that the file upload completed successfully. In this case, you need to do the following:

* Display a status message indicating this file has been uploaded successfully. You can call displayMessage() to achieve this effect.
* Set a flag to indicate this file has been uploaded (to prevent us from trying to upload it again later). To do this, use the fileBeingUploaded private variable to access the status information for the current file, and set its uploadComplete property to true.
* Call displaySummaryPanel() to upload the statistics at the top of the web page.

1. Implement the error event on the xhr.upload object. This event indicates that an error occurred during the upload operation. In this case, display an error message via the displayMessage() function.
2. Implement the abort event on the xhr.upload object in a similar way, to indicate the user has aborted the upload operation somehow.
3. At this point, you have implemented event handlers for all the important events that might occur, so now you're ready to actually write code to perform the Ajax operation! Follow these steps:

* Call open() on the xhr object, to open a POST request to "/Home/UploadFile". As you will recall from our earlier discussions, this URL corresponds to the server-side method that receives our data and saves the file on the server.
* Call send() on the xhr object, passing the FormData object (i.e. fd) as a parameter. This will simulate an HTML form being posted to the server, with filename and file parameters embedded.

That completes the implementation of the client-side code in Index.cshtml, so you're ready to test the application!

**Exercise 6: Running the application**

Build and run the application. The web page should appear in the browser as usual. Click the *Add files* button and choose some large files (max 20MB) to be uploaded. When you're ready, click the *Upload* button to upload the files to the server.

All being well, you should see orange progress messages for each file, followed by a green message confirming each file has been successfully uploaded. If you get any problems, press F12 in the browser and try debugging the JavaScript code to detect the problem. Ask your instructor for some help if you get stuck.

If the application has worked correctly, it should have saved all the files in the server folder. To check this out, open File Explorer and go to the following folder:

C:\Html5Css3JsDev\Student\15-CommunicationsApi

\MultiFileUploadApp\MultiFileUploadApp\App\_Data

Verify this folder contains all of the files you uploaded from the client browser.

**Exercise 7 (if time permits): Additional suggestions**

Explore the XMLHttpRequest Level 2 capability for making Ajax requests to a different domain than the domain from which the web page was downloaded.