Uploading Files to the Web Server

Logi Info
v12.0 - Nov 2015

Logi applications include the ability to allow users to upload files for storage on the web server. Logi Info provides elements specifically for this purpose and this document provides guidance in using them. Topics include:

- About Uploading Files
- Identifying Files to Upload
- Saving Uploaded Files
- Adjusting the Maximum Upload Size Limit

About Uploading Files

Logi Info provides a mechanism for uploading files from a user's computer to the web server, using the RFC 1867 Form-based File Upload in HTML protocol. This protocol was published in 1995 yet continues to function cleanly. With it, files are transmitted individually, without encryption, and immediately (there is no deferred transmission).

The upload process first completes the file transmission then makes information about the file (type, size, etc.) available. At that point, developers can use code to test the uploaded file and delete it, if desired, as a kind of "retroactive filtering".

Upload Considerations

The process of uploading a file to your web server consists of two steps: first, at the browser, the user has to *identify* a file on his file system to be uploaded, and then, second, at the web server, the uploaded file has to be *named* and *saved*. Logi Info provides elements that make this all fairly easy to implement. However, developers first need to consider these issues:

- **What file types will you allow to be uploaded?** For security or virus-protection reasons, you may want to limit uploads to non-executable file types, prohibiting those such as .exe, .com, .bat, and perhaps even macro-capable Microsoft Office types such as .docx and .xlsx. It's a good idea to identify the smallest set of acceptable file types and

refuse all others.

- **Should you restrict the size of uploaded files?** For most developers, this is a question of available storage space. The default maximum file size that can be uploaded under .NET is 4MB (although this can be increased if necessary - see below). You may also want to consider a limit on the *number* of files each user can upload.

  Another critical issue is that the actual upload transmission is not very efficient and large uploads can take a considerable amount of time. The process doesn't provide any feedback that would allow you to display a progress bar, either, so your users will have no way of knowing what's going on. Finally, if the upload is too large or the session times out, your application will throw an exception that's hard to handle gracefully.

- **How will the uploaded files be stored and identified?** Uploaded files are first stored as temporary files in your application's *rdDownload* folder and are subsequently renamed and relocated. Developers need to plan for storage locations on the web server and for a method of providing unique names to the files (the Logi @Function.GUID~ token can be very helpful for this purpose). You could, for example, use a database table to hold information about the uploaded files such as their original names, sizes, and GUID-based identifiers.

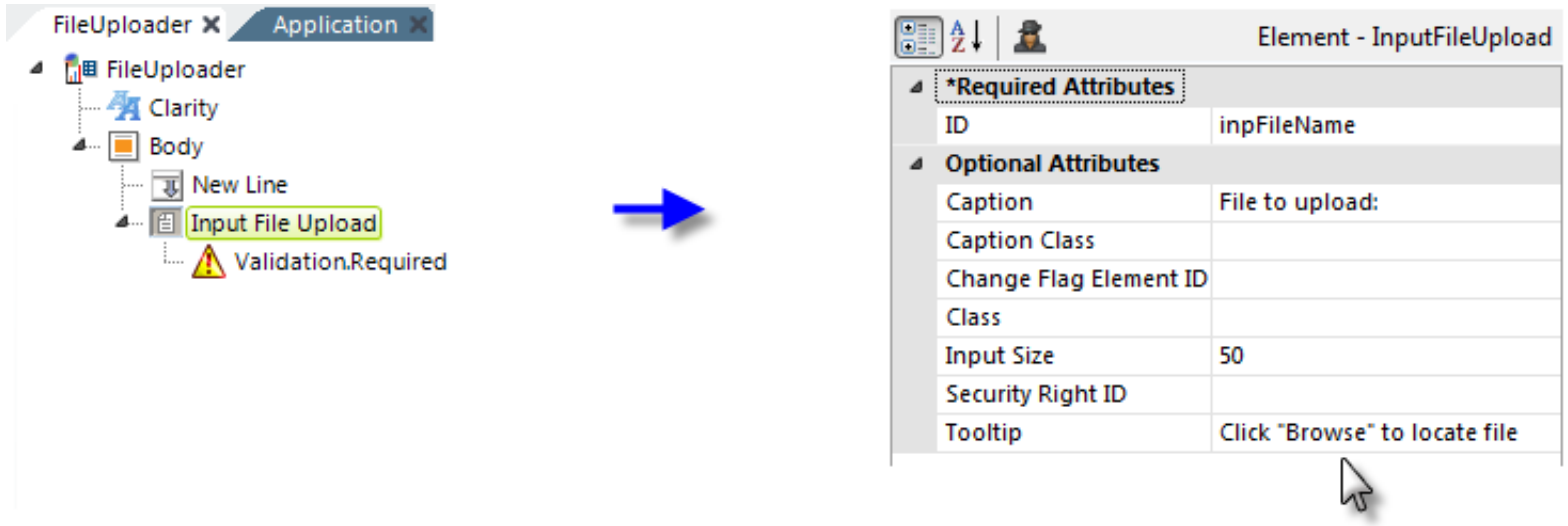
- **How will the uploaded file be managed?** Assuming you're not going to just store uploaded files forever, you'll probably need to develop a way of managing them. Here again, a database table can be useful for identifying uploaded files for deletion.

- **What About Virus Protection?** If your web server's anti-virus software conducts "on access" scans of all files, any uploaded files stored on the web server should be examined automatically. If not, you'll want to consider how you can screen uploaded files for viruses.

Back to the top

Identifying Files to Upload

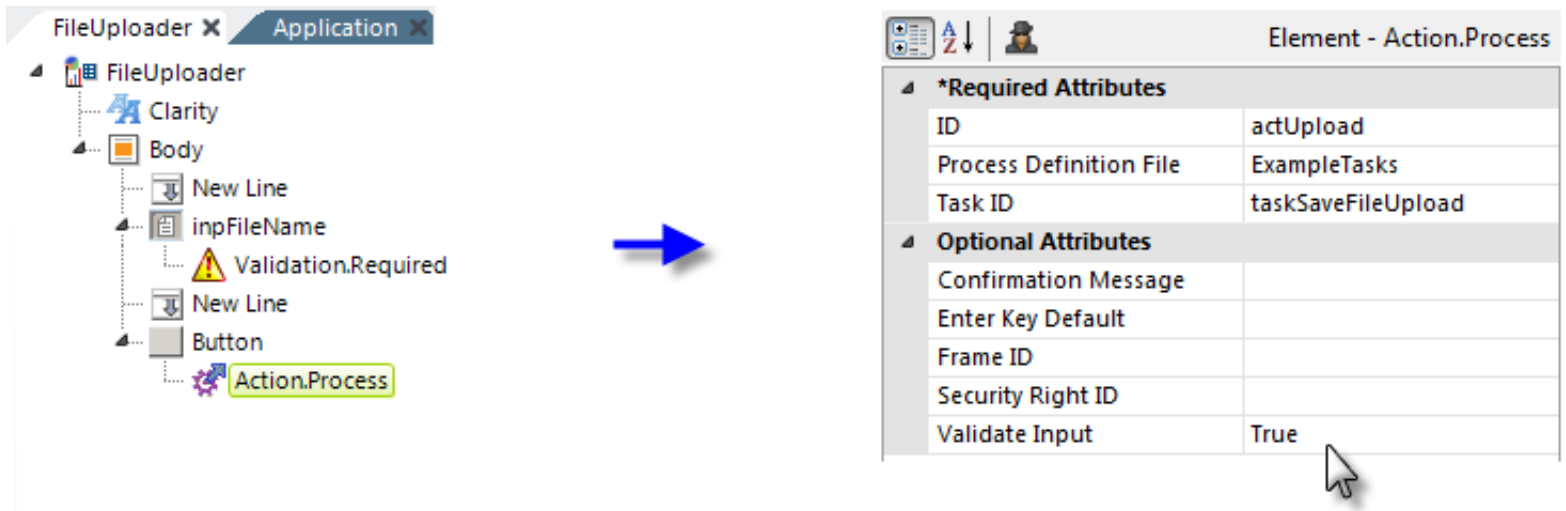
The upload process begins with the user *identifying* the file to upload. This can be done very easily using the **Input File Upload** element in a report definition. This element displays both an input text box and a "Browse" button that allows users to browse their local file system.



1. As shown above, add an **Input File Upload** element to a report and set its attributes as desired.
2. Add a **Validation.Required** element beneath it to ensure that the filename value is not empty.

The example above shows what the **Input File Upload** element looks when displayed. Clicking the Browse... button opens a file explorer window, allowing the user to find and select a local file. The fully-qualified file name (drive letter, complete path, and full file name) is entered into the text box. It's not possible to enter this information by typing.

Only *one* file at a time can be uploaded per Input File Upload element.



3. Add **New Line** and **Button** elements, as shown above, and beneath the Button, add an **Action.Process** child element which calls the appropriate Process task. Note that its **Validate Input** attribute is set to *True* to cause the Validation.Required element to check to see if a file name was entered.

When the user clicks the button and submits the page, the upload will take place immediately and the uploaded file will be stored as a temporary file in your application's *rdDownload* folder on the web server.

At the same time, the Process task specified will be called; its role is discussed in the next section.

Back to the top

Saving Uploaded Files

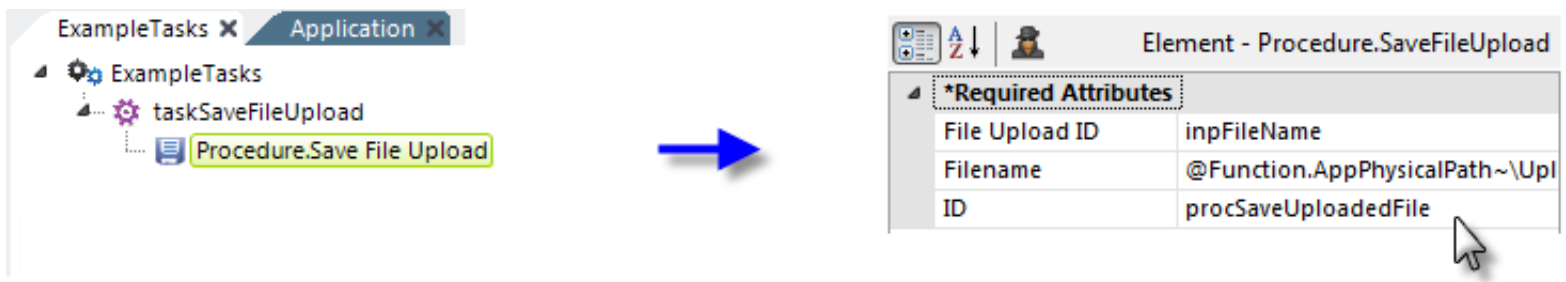
After the physical upload completes, a Process task is used to save the temporary file to the final file and determine what has been uploaded. Several useful special tokens become available in the task to facilitate the process:

Token	Description
@FileUpload.UploadFileName~	Contains the file name, with extension but stripped of any path info, of the original file. Only available for use in attributes of the Procedure.Save Uploaded File element. Note that the RFC 1867 protocol used to upload files *does not* include access to the full file *path* of the original file, only its file *name*.
@FileUpload.UploadFileExtension~	Contains the file extension of the original file, including the period. This is case-sensitive and only available for use in attributes of the Procedure.Save Uploaded File element.
@Procedure.<*MyProc*>.UploadFileName~	Contains the file name, with extension but stripped of any path info, of the original file. Available throughout the task.
@Procedure.<*MyProc*>.UploadFileExtension~	Contains the file extension of the original file, including the period. This is case-sensitive and available throughout the task.
@Procedure.<*MyProc*>.UploadFileContentType~	Contains the file's MIME or content-type string. Available throughout the task.

@Procedure.<*MyProc*>.UploadFileLength~	Contains the number of bytes uploaded (the file size). Available throughout the task.

where <*MyProc*> is the ID of the task's **Procedure.Save Uploaded File** element.

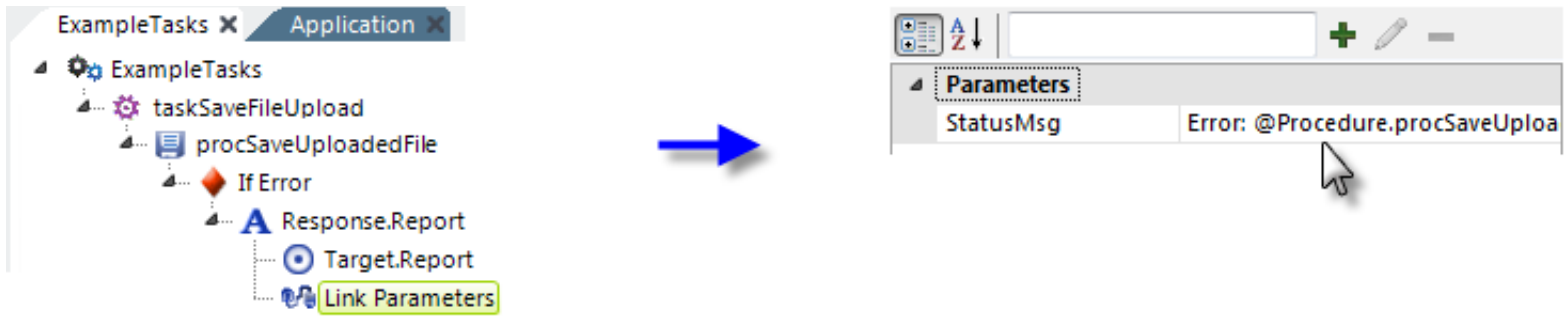
Now you're ready to create a Process task to save the uploaded file:



1. In a process definition, add a **Task** element. Its **ID** should be the same as the value entered in the Task ID attribute of the Action.Process element in the report definition in the previous section of this document.
2. Add a **Procedure.Save File Upload** element to the task.
3. Set its **File Upload ID** attribute to the ID of the Input File Upload element in the report definition.
4. Set its **Filename** attribute to the fully-qualified file name you want to save the temporary upload file to, beginning with a physical drive letter on the web server. Several tokens can be used together to provide a value for the this attribute, for example:

   ```
   @Function.AppPhysicalPath~\UploadedFiles\@FileUpload.UploadFileName~
   ```

   The value shown above will move the temporary upload file to a folder called "UploadedFiles" in your application folder and rename it to the file's original name. Note that the target folder ("UploadedFiles") must exist *in advance*; the element will not create it. If this folder is outside of your application folder, the account the web server uses to run your application (ASPNET, NETWORK SERVICE, or Application Pool) has to be granted *Write* permission to that folder.



4. To handle errors, add an **If Error** element beneath the Save Uploaded File element.
5. Add **Response.Report**, **Target.Report**, and **Link Params** elements, as shown, to redirect to the calling report in case of an error. Any error message will be available using this token:

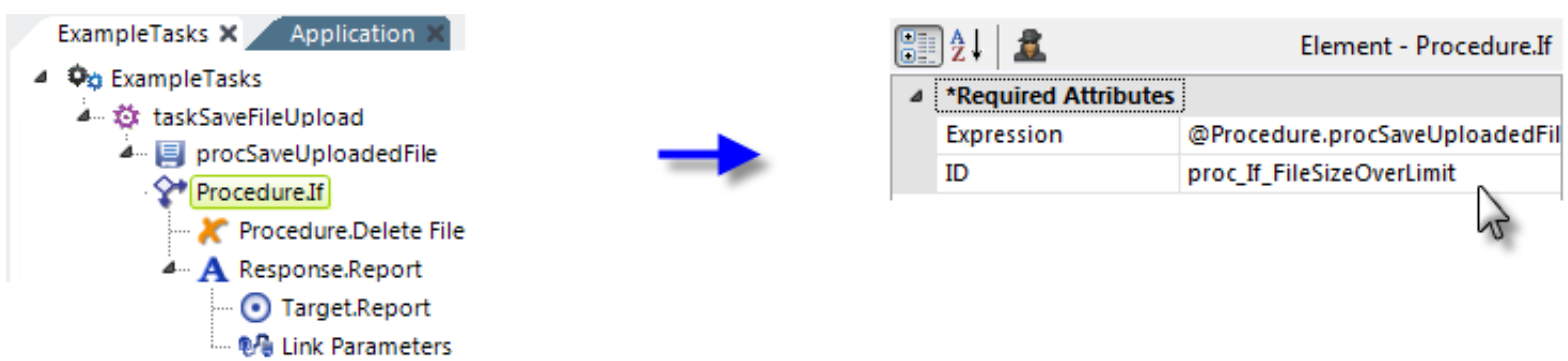
```
@Procedure.procSaveUploadedFile.ErrorMessage~
```

6. Back in the report definition, you can use an @Request token to test for and display any error message returned.

If the report is run now, the file will be uploaded and saved to the UploadedFiles folder. Note that there is a default 4MB file size limit, discussed in more detail below.

Qualifying Uploaded Files

Once an upload has occurred and the temporary file has been saved, we have an opportunity to *qualify* the uploaded file and delete it if necessary, using criteria such as file size or file type. This "post-upload" filtering isn't the most desirable arrangement but it's all that's available under the RFC 1867 protocol.

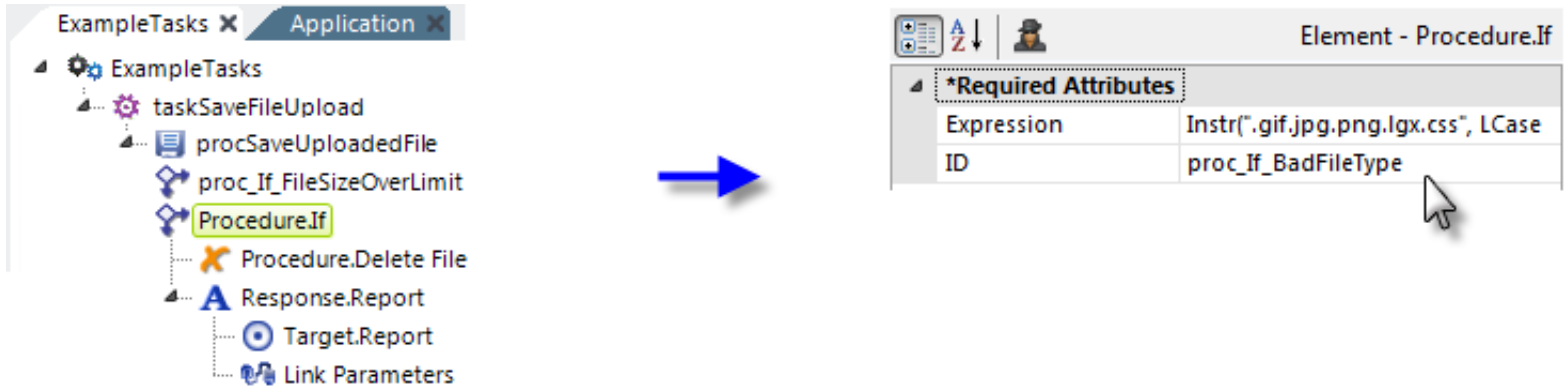


In the example above, we've added additional elements to qualify the uploaded file. A **Procedure.If** element is used to test the *size* of the uploaded file; if it's larger than 1MB, then it's deleted. The complete **Expression** attribute value used is:

```
@Procedure.procSaveUploadedFile.UploadFileLength~ > 1024000
```

Set the **Procedure.Delete File** element's **Filename** attribute to the same value as the Procedure.Save File Upload element's Filename attribute.

As before, you should also include Response, Target, and Link Parameters elements to redirect the browser to the original report and display a meaningful error message.



In the example above, more elements have been added to delete the file if the uploaded file's *extension* isn't one of five approved types. The complete **Expression** attribute value is:

```
Instr(".gif.jpg.png.lgx.css", LCase("@Procedure.procSaveUploadedFile.UploadFileExtension~")) = 0
```

In this example, the intrinsic **Instr** function is used to determine if the uploaded file's extension is in the string of concatenated approved file types. Note that the procedure token's file extension value is *case-sensitive*, which is why the **LCase** function is also used.

As before, you should also include Response, Target, and Link Parameters elements to redirect the browser to the original report and display a meaningful error message.

In conclusion, file uploading is best done under very controlled circumstances and you should be prepared for the likelihood that some uploads will fail.

Back to the top

Adjusting the Maximum Upload Size Limit

The default maximum upload size, as determined by the .NET Framework, is 4MB. To increase this, to a maximum of 2GB, you need to edit the `web.config` file in your application folder. In the `<system.web>` section, add this:

```
<httpRuntime maxRequestLength="nnnn" executionTimeout = "ssss"/>
```

where:

maxRequestLength specifies the limit for the input stream buffering threshold, in KB. This limit can be used to prevent denial of service attacks that are caused, for example, by users posting large files to the server. The default value is 4096 (4MB). To enable large file uploads you need to change the value of this attribute to the largest allowed combined file size for your application. If someone selects and uploads files with total size larger than maxRequestLength, this will result in a "Page not found" error (which is the default error of the Framework), and

executionTimeout specifies the maximum number of seconds that a request is allowed to execute before being automatically shut down by .NET. The value of this setting is ignored in Debug mode. The default in the .NET 4.x Framework is 110 seconds. To enable large file uploads, which can take large periods of time, increase the value of this property.

Back to the top

© copyright 2007-2016 Logi Analytics, Inc. all rights reserved