File Upload

The Struts 2 framework provides built-in support for processing file uploads that conform to RFC 1867, "Form-based File Upload in HTML". When correctly configured the framework will pass uploaded file(s) into your Action class. Support for individual and multiple file uploads are provided. When a file is uploaded it will typically be stored in a temporary directory. Uploaded files should be processed or moved by your Action class to ensure the data is not lost. Be aware that servers may have a security policy in place that prohibits you from writing to directories other than the temporary directory and the directories that belong to your web application.

2

Dependencies

The Struts 2 framework leverages add-on libraries to handle the parsing of uploaded files. These libraries are not included in the Struts distribution, you must add them into your project. The libraries needed are:

Library	URL	Struts 2.0.x	Struts 2.1.x	Struts 2.5.x
Commons-FileUpload	http://commons.apache.org/fileupload/	1.1.1	1.2.1	1.3.2
Commons-IO	http://commons.apache.org/io/	1.0	1.3.2	2.4

If you are using Maven then you can add these libraries as dependencies in your project's pom.xml.

xmlStruts 2.0.x File Upload Dependencies<dependency> <groupId>commons-fileupload</groupId> <artifactId>commons-fileupload</artifactId> <version>1.1.1</version> </dependency> <dependency> <groupId>commons-io</groupId> <artifactId>commons-io</artifactId> <version>1.0< /version> </dependency> xmlStruts 2.1.x File Upload Dependencies<dependency> <groupId>commons-fileupload</groupId> <artifactId>commons-fileupload</artifactId> <version>1.2.1</version> </dependency> <dependency> <groupId>commons-io</groupId> <artifactId>commons-io</artifactId> <version>1.3.2</version> </dependency>

Basic Usage

The `org.apache.struts2.interceptor.FileUploadInterceptor` class is included as part of the `defaultStack`. As long as the required libraries are added to your project you will be able to take advantage of of the Struts 2 fileUpload capability. Configure an Action mapping for your Action class as you typically would.

Example action mapping:

xml<action name="doUpload" class="com.example.UploadAction"> <result name="success">good_result.jsp</result> </action>

A form must be create with a form field of type file, `<INPUT type="file" name="upload">`. The form used to upload the file must have its encoding type set to multipart/form-data, `<FORM action="doUpload" enctype="multipart/form-data" method="post">`. The standard procedure for adding these elements is by using the Struts 2 tag libraries as shown in the following example:

Example JSP form tags:{snippet:id=example-form|lang=xml|javadoc=true|url=org.apache.struts2.interceptor.FileUploadInterceptor}The fileUpload interceptor will use setter injection to insert the uploaded file and related data into your Action class. For a form field named **upload** you would provide the three setter methods shown in the following example:

Example Action class:

javapackage com.example; import java.io.File; import com.opensymphony.xwork2.ActionSupport; public class UploadAction extends ActionSupport { private File file; private String contentType; private String filename; public void setUpload(File file) { this.file = file; } public void setUploadContentType(String contentType) { this.contentType = contentType; } public void setUploadFileName(String filename) { this.filename = filename; } public String execute() { //... return SUCCESS; } }

The purpose of each one of these methods is described in the table below. Notice that if you have multiple file form elements with different names you would be required to have another corresponding set of these methods for each file uploaded.

Method Signature	Description
setX(File file)	The file that contains the content of the uploaded file. This is a temporary file and `file.getName()` will not return the original name of the file
setXContentType(String contentType)	The mime type of the uploaded file
setXFileName(String fileName)	The actual file name of the uploaded file (not the HTML name)

Uploading Multiple Files

As mentioned in the previous section one technique for uploading multiple files would be to simply have multiple form input elements of type file all with different names. This would require a number of setter methods that was equal to 3 times the number of files being uploaded. Another option is to use Arrays or java.util.Lists. The following examples are taken from the Showcase example application that is part sample applications you can download at http://struts.apache.org/download.cgi. For the Action mapping details see `struts-fileupload.xml` in the sample application download.

Uploading Multiple Files using Arrays

multipleUploadUsingArray.jsp Notice all file input types have the same name.

html<s:form action="doMultipleUploadUsingArray" method="POST" enctype="multipart/form-data"> <s:file label="File (1)" name="upload" /> <s:file label="File (2)" name="upload" /> <s:file label="FIle (3)" name="upload" /> <s:submit cssClass="btn btn-primary"/> </s:form>

MultipleFileUploadUsingArrayAction.java

javapublic class MultipleFileUploadUsingArrayAction extends ActionSupport { private File[] uploads; private String[] uploadFileNames; private String[] uploadContentTypes; public String upload() throws Exception { System.out.println("\n\n upload2"); System.out.println("files:"); for (File u : uploads) { System.out.println("*** " + u + "\t" + u.length()); } System.out.println("filenames:"); for (String n : uploadFileNames) { System.out.println ("*** " + n); } System.out.println("content types:"); for (String c : uploadContentTypes) { System.out.println("*** " + c); } System.out.println("\n\n"); return SUCCESS; } public File[] getUpload() { return this.uploads; } public void setUpload(File[] upload) { this.uploads = upload; } public String[] getUploadFileName() { return this.uploadFileNames; } public void setUploadFileName(String[] uploadFileName) { this.uploadFileNames = uploadFileName; } public String[] getUploadContentType() { return this.uploadContentTypes; } public void setUploadContentType(String[] uploadContentType) { this.uploadContentTypes = uploadContentType; } }

Uploading Multiple Files using Lists

multipleUploadUsingList.jsp Notice all file input types have the same name.

xml<s:form action="doMultipleUploadUsingList" method="POST" enctype="multipart/form-data"> <s:file label="File (1)" name="upload" /> <s:file label="File (2)" name="upload" /> <s:file label="FIle (3)" name="upload" /> <s:submit cssClass="btn btn-primary"/> </s:form>

MultipleFileUploadUsingListAction.java

javapublic class MultipleFileUploadUsingListAction extends ActionSupport { private List<File> uploads = new ArrayList<File>(); private List<String> uploadFileNames = new ArrayList<String>(); private List<String> uploadContentTypes = new ArrayList<String>(); public List<File> getUpload() { return this.uploads; } public void setUpload(List<File> uploads) { this.uploads = uploads; } public List<String> getUploadFileName() { return this.uploadFileNames; } public void setUploadFileName(List<String> uploadFileNames) { this.uploadFileNames = uploadFileNames; } public List<String> getUploadContentType() { return this.uploadContentTypes; } public void setUploadContentType(List<String> contentTypes) { this.uploadContentTypes = contentTypes; } public String upload() throws Exception { System.out.println("\n\n upload1"); System.out.println ("files:"); for (File u : uploads) { System.out.println("*** " + u + "\t" + u.length()); } System.out.println("filenames:"); for (String n : uploadFileNames) { System.out.println("*** " + n); } System.out.println("content types:"); for (String c : uploadContentTypes) { System.out.println("*** " + c); } System. out.println("\n\n"); return SUCCESS; } }

Advanced Configuration

The Struts 2 `default.properties` file defines several settings that affect the behavior of file uploading. You may find in necessary to change these values. The names and default values are:

nonestruts.multipart.parser=jakarta struts.multipart.saveDir= struts.multipart.maxSize=2097152
Please remember that the **struts.multipart.maxSize** is the size limit of the whole request, which means when you uploading multiple files, the sum of their size must be below the **struts.multipart.maxSize**!

In order to change theses settings you define a constant in your applications `struts.xml` file like so:

xml<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN" "http://struts.apache.org/dtds/struts-2.0.dtd"> <struts> <constant name="struts.multipart.maxSize" value="1000000" /> ... </struts>

Additionally the `fileUpload` interceptor has settings that can be put in place for individual action mappings by customizing your interceptor stack.

xml<action name="doUpload" class="com.example.UploadAction"> <interceptor-ref name="basicStack"/> <interceptor-ref name="fileUpload"> <param name="allowedTypes">text/plain</param> </interceptor-ref> <interceptor-ref name="validation"/> <interceptor-ref name="workflow"/> <result name="success">good_result.jsp</result> </action>

File Size Limits

There are two separate file size limits. First is `struts.multipart.maxSize` which comes from the Struts 2 `default.properties` file. This setting exists for security reasons to prohibit a malicious user from uploading extremely large files to file up your servers disk space. This setting defaults to approximately 2 megabytes and should be adjusted to the maximum size file (2 gigs max) that your will need the framework to receive. If you are uploading more than one file on a form the `struts.multipart.maxSize` applies to the combined total, not the individual file sizes. The other setting, `maximumSize`, is an interceptor setting that is used to ensure a particular Action does not receive a file that is too large. Notice the locations of both settings in the following example:

xml<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN" "http://struts.apache.org/dtds/struts-2.0.dtd"> <struts> <constant name="struts.multipart.maxSize" value="1000000" /> <action name="doUpload" class="com.example.UploadAction"> <interceptor-ref name="basicStack"/> <interceptor-ref name="fileUpload"> <param name="maximumSize" >500000</param> </interceptor-ref> <interceptor-ref name="validation"/> <interceptor-ref name="workflow"/> <result name="success" >good_result.jsp</result> </action> </struts>

File Types

There are two ways to limit the uploaded file type, declaratively and programmatically. To declaratively limit the file type a comma separated list of allowedTypes can be specified as a fileUpload interceptor param as shown in the following example:

xml<action name="doUpload" class="com.example.UploadAction"> <interceptor-ref name="basicStack"/> <interceptor-ref name="fileUpload"> <param name="allowedTypes">image/jpeg,image/gif</param> </interceptor-ref> <interceptor-ref name="validation"/> <interceptor-ref name=" workflow"/> <result name="success">good_result.jsp</result> </action>

When the uploaded file type does not match one of the MIME types specified a field error will be created as described in the next section entitled Error Messages. Programmatically limiting the file type means using the information passed in to your Action class via the `setXContentType (String contentType)` method. The benefit to this type of approach would be that it's more flexible and no interceptor configuration would be needed if file sizes are keep under 2 megs.

Error Messages

If an error occurs several field errors will be added assuming that the action implements `com.opensymphony.xwork2.ValidationAware` or extends `com.opensymphony.xwork2.ActionSupport`. These error messages are based on several i18n values stored in struts-messages. properties, a default i18n file processed for all i18n requests. You can override the text of these messages by providing text for the following keys:

Error Key	Description
struts.messages.error.uploading	A general error that occurs when the file could not be uploaded
struts.messages.error.file.too.large	Occurs when the uploaded file is too large as specified by maximumSize.
struts.messages.error.content.type.not.allowed	Occurs when the uploaded file does not match the expected content types specified
struts.messages.error.file.extension.not.allowed	Occurs when uploaded file has disallowed extension
struts.messages.upload.error.SizeLimitExceededException	Occurs when the upload request (as a whole) exceed configured **struts.multipart.maxSize**
struts.messages.upload.error.<Exception class SimpleName>	Occurs when any other exception took place during file upload process

Temporary Directories

All uploaded files are saved to a temporary directory by the framework before being passed in to an Action. Depending on the allowed file sizes it may be necessary to have the framework store these temporary files in an alternate location. To do this change `struts.multipart.saveDir` to the directory where the uploaded files will be placed. If this property is not set it defaults to `javax.servlet.context.tempdir`. Keep in mind that on some operating systems, like Solaris, `/tmp` is memory based and files stored in that directory would consume an amount of RAM approximately equal to the size of the uploaded file.

Alternate Libraries

The `struts.multipart.parser` used by the fileUpload interceptor to handle HTTP POST requests, encoded using the MIME-type multipart /form-data, can be changed out. Currently there are two choices, jakarta and pell. The jakarta parser is a standard part of the Struts 2 framework needing only its required libraries added to a project. The pell parser uses Jason Pell's multipart parser instead of the Commons-FileUpload library. The pell parser is a Struts 2 plugin, for more details see: pell multipart plugin. There was a third alternative, cos, but it was removed due to licensing incompatibilities.

As from Struts version 2.3.18 a new implementation of `MultiPartRequest` was added - `JakartaStreamMultiPartRequest`. It can be used to handle large files, see WW-3025 for more details, but you can simple set

```
<constant name="struts.multipart.parser" value="jakarta-stream" />
```

in struts.xml to start using it.

Request validation

The `struts.multipart.validationRegex` is used to define a RegEx to be used to validate if the incoming request is a multipart request. The request must use the `POST` method and match the RegEx, by default the RegEx is defined as follow:

^multipart\\/form-data(; boundary=[a-zA-Z0-9]{1,70})?

Please read RFC1341 the **Multipart section** for more details, existing Struts `Multipart` parsers support only `multipart/form-data` content type. This option is available since Struts 2.3.11.

Disabling file upload support

You can alternatively disable the whole file upload mechanism defining a constant in `struts.xml`:

xml<constant name="struts.multipart.enabled" value="false"/>

With this constant in place, Struts will ignore a `Content-Type` header and will treat each request as an ordinary http request. This option is available since Struts 2.3.11.