File Upload 2.0 Component Specification

1. Design

The File Upload component supports web-based file uploads. Files are received via HTTP requests. Applications implementing the component have the ability to store uploaded files on the file system.

This is a complete redesign of version 1.0. Despite the rewrite, most of the functionality provided in version 1.0 is retained. The most obvious enhancement is to store the uploaded files in the centralized File System Server. This will allow the file upload to work in a clustered environment, in case the user's subsequent requests are served from another node.

This design is a great improvement as far as flexibility is concerned. In version 1.0, the parsing of requests is restricted solely to the HTTP 1.1 standard, and an instance has to be created for every request. In version 2.0, the component supports pluggable parsing implementation through the *RequestParser* interface, so that it can still be used on a different protocol. One *FileUpload* instance is now sufficient to serve multiple requests concurrently. This fits well in the servlet environment where only one instance is needed during the lifetime of the servlet. Other added features include uploading files to memory, and some file operations with a specified file id, such as retrieval and removal of files.

The challenge of this design lies in figuring out the commonality between local file system and remote file system, and possibly another kind of persistence to store the uploaded files in general. As such, the *FileUpload* and *UploadedFile* are defined as the base classes to abstract the uploader and uploaded file respectively. *MemoryFileUpload* works with the memory to produce *MemoryUploadedFile* instances. *LocalFileUpload* and *RemoteFileUpload* work with the file systems to produce *LocalUploadedFile* instances.

Another decision to make is the internals of the uploaded file object, while making sure the object is serializable. In other words, the internals must be serializable themselves. The problem is we can't store the connection inside the object for remote file access. Since the requirement assumes that the uploaded files are not modified in the remote server, it is safe to assume that the downloaded file is an accurate representation. Considering also the performance gain, the approach to cache the remote files in the local file system is chosen, so that we can simply store the serializable File inside the uploaded file object.

Note that extra file information such as form file name and content type is available immediately after parsing the request. It will not available again when retrieving an uploaded file using a file id later. The component does not do the job of storing these pieces of information in this version. If necessary to retrieve them later, users should store them in some way more effective.

Finally, it is worth mentioning that the Object Factory component is not used to instantiate the classes. For one thing, it requires a completely different configuration structure which may not be convenient for those who get used to the previous version. For another, it requires the configurable attributes to be specified in the constructor. The argument list may look too long for those classes supporting 5 attributes or more.

1.1 Design Patterns

Abstract factory pattern is used in the FileUpload class. Different implementations of FileUpload produce different implementations of UploadedFile.

Strategy pattern is used through the RequestParser interface. It allows different parsing strategy implementations plugged to the FileUpload class.

1.2 Industry Standards

RFC 1867 (http://www.ietf.org/rfc/rfc1867.txt)

1.3 Required Algorithms

1.3.1 Overview of HTTP Request

The first thing is to understand what the request looks like when uploading files using HTTP 1.1 protocol. An example is given here for illustration. The parsing of request is based on the data format.

First, the content type of the request should have been set to “multipart/form-data; boundary=[some stuff]”. This is how you can ascertain that you're really dealing with a properly encoded upload post. The boundary value is probably of the form ----------18788734234, where the digits are randomly generated. This boundary is a MIME boundary; it's guaranteed not to appear anywhere in the data except between the multiple parts of the data.

The data itself appears in blocks that are made up of lines separated by CR/LF pairs. It looks like this more or less. The line numbers are shown here for explanation purpose, and should not appear in real data.

```
01 -----------------------------18788734234
02 Content-Disposition: form-data; name="parameter_name"
03
04 [parameter value here]
05 -----------------------------18788734234
06 Content-Disposition: form-data; name="form_file_name"; filename="file.gif"
07 Content-Type: image/gif
08
09 [file contents here]
10 -----------------------------18788734234--
```

These points should be noted:

- The resulting boundary value should have an extra “--“ in front of the value extracted from the content type.
- The end of the data is marked by an extra “--“ after the boundary value.
- The end of the header in each block is marked by an empty line after the boundary line.
- The end of the parameter value or file contents in each block is marked by the occurrence of the boundary line.
- The absence of “filename=…” in “Content-Disposition” header indicates that this is a parameter block. Otherwise this should be a file block.
- The “Content-type” header in the file block is optional.

For better understanding, here shows the method calls of the RequestParser in the correct sequence, and where the input stream positions upon each call based on the above example.

```
parser.parseRequest(...);
// at the start of line 02

while (parser.hasNextFile()) {
    // at the start of line 09
    // should have parsed the parameter

    parser.writeNextFile(...);
    // at the end of line 10
    // should have parsed the file
}
```

1.3.2 Overview of Local File Upload

Local file upload is just a matter of saving the uploaded files into a target directory in the local file system. The file id is assigned as the name of the saved file. If overwriting of files is allowed, the remote file name will be directly used as the saved file name. If overwriting of files is not allowed, the saved file name will be composed as “unique_id + remote_file_name”, where the unique id is generated by the GUID Generator component. The GUID Generator ensures the uniqueness of id across all servers.

1.3.3 Overview of Remote File Upload

Remote file upload is slightly more complex. It first saves the uploaded files into a temporary directory in the local file system, using the unique file names as described above. The FileSystemClient is then used to upload the saved files from the local file system to the FileSystemServer. The saved files act as a cache for subsequent download requests, and are deleted from the temporary directory upon the exit of JVM. The file id is assigned as the one provided by the FileSystemServer.

Retrieval of uploaded file with a specified file id reverses the above process. First it looks for the saved file under the temporary directory in the local file system. If not, the FileSystemClient is then called to download the file from the FileSystemServer into the temporary directory. An input stream is opened with the downloaded file. Make sure the downloaded files are deleted upon the exit of the JVM.

The following sub-sections show how to use the FileSystemClient to perform file-related operations. They are provided in the demo section of FileSystemServer’s component specification.

1.3.3.1 Upload a file

```
String requestId = fsClient.uploadFile(fileLocation,fileName);
FileUploadCheckStatus status =
fsClient.getFileUploadCheckStatus(requestId, true);
if (status==FileUploadCheckStatus.UPLOAD_ACCEPTED) {
       while(fsClient.isFileTransferWorkerAliva(requestId)) {
           Thread.sleep(500);
       }
       ResponseMessage response =
       (ResponseMessage) fsClient.receiveResponse(requestId,true);
       if (response != null && response.getException()==null) {
              String fileId = (String) response.getResult();
       } else {
              // handle the exception
       }
}
```

1.3.3.2 Retrieve a file

```
String requestId = fsClient.retrieveFile(fileId,fileLocation);
while(fsClient.isFileTransferWorkerAlive(requestId)) {
    Thread.sleep(500);
}
```

```
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       String fileName = (String) response.getResult();
       // use the file with the resulted file name
} else {
       // handle the exception
}
```

1.3.3.3 Get file name

```
String requestId = fsClient.getFileName(fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       String fileName = (String) response.getResult();
} else {
       // handle the exception
}
```

1.3.3.4 Remove a file

```
String requstId = fsClient.removeFile(fileId);
ResponseMessage response =
(ResponseMessage) fsClient.receiveResponse(requestId,true);
if (response.getException()==null) {
       // response.getResult()==null
} else {
       // handle the exception
}
```

1.4 Component Class Overview

FileUpload (abstract class):

A class abstracting the parsing of ServletRequest to produce the FileUploadResult. The information parsed in the ServletRequest includes uploaded files and form parameters. Implementations should define the strategy to store the uploaded files to persistence (e.g. in memory or some kind of persistent storage).

The uploaded file can be retrieved later (e.g. after restarting the application) using a file id. The file id should be unique to every uploaded file. Implementations should also define how to assign this file id if applicable.

MemoryFileUpload:

This class represents the memory storage of the uploaded files. This class works by saving the uploaded files into memory (using a byte array). Overwriting of files is not allowed.

MemoryFileUpload can be used for fast retrieval of data (without I/O) when the file size is small. It cannot be used in a clustered environment however. It is strongly recommended to limit the file size by specifying the single_file_limit property in the configuration.

LocalFileUpload:

This class represents the persistent storage of the uploaded files in the local file system. This class works by saving the uploaded files into a default or specified directory (which should be among the allowed directories). In addition, the user can specify whether overwriting of files (based on the remote file name) is desirable. If overwrite is false, a unique file name is generated for each uploaded file and this is used as the file id. If overwrite is true, the remote file name will simply be used as the file id. This will allow uploads with the same remote file name under a request or different requests.

LocalFileUpload can be used to store files whose size is moderately large. The advantage is the saving of memory and persistent storage. It cannot be used in the clustered environment however.

RemoteFileUpload:

This class represents the persistent storage of the uploaded files in the remote file system. The remote file system is provided by the File System Server component. This class works by saving the uploaded files using unique file names into a temporary directory in the local file system first, then calling the File System Server to upload the files. These locally saved files act as a cache, and should be removed from the temporary directory upon the exit of JVM. The file id is provided by the File System Server (assigned during upload). This will allow uploads with the same remote file name under a request or different requests. Overwriting of files is not allowed.

RemoteFileUpload can be used in a clustered environment when the request is served from another node. However the upload and download time will be slower because the data is transferred over the network, especially when the file size is large.

UploadedFile (abstract class):

This class abstracts an uploaded file stored in the persistence. Some file information and the file contents (byte data) can be retrieved. Implementations should define the way to retrieve file information, while assuring the UploadedFile object is serializable.

MemoryUploadedFile:

This class represents an uploaded file stored in the memory. Some file information and the file contents (byte data) can be retrieved from the memory. No file id is associated with a MemoryUploadedFile.

LocalUploadedFile:

This class represents an uploaded file stored in the local file system. Some file information and the file contents (byte data) can be retrieved from the locally stored file. Every such instance is identified by a file id. It is not necessary to have the file name as the file id however.

RequestParser (interface):

This interface defines the contract to parse the uploaded files and parameters from the servlet request. For uploaded file, its form file name, remote file name, content type and file contents will be retrieved. It provides the method to write the file contents to the specified output stream for storage purpose.

HttpRequestParser:

An implementation of RequestParser interface to parse the uploaded files and parameters from the servlet request using the HTTP 1.1 standard. For details of the protocol please refer to RFC 1867.

FileUploadResult:

A class that encapsulates the file upload information collected during the parsing of the request. This includes uploaded files and form parameters. The APIs for the form parameters resemble the HttpServletRequest for easy grasp of usage.

1.5 Component Exception Definitions

1.5.1 Custom Exceptions

FileUploadException:

This is the base exception class for the File Upload component. All exception classes should extend from it.

ConfigurationException:

This is an exception class to indicate any errors related to configuration, such as missing required properties or invalid property values (e.g. not a number). This exception is thrown from the constructors accepting a namespace.

PersistenceException:

This is an exception class to indicate any errors related to persistence where the uploaded files are stored. It can be used to wrap exceptions thrown from database or remote file system, for example. This exception is thrown for all operations involving persistence, such as uploading or downloading files.

FileDoesNotExistException:

This is an exception class to indicate that the requested file (specified with file id) does not exist in the persistence. This exception is thrown in the FileUpload and UploadedFile classes. These classes retrieve file information using the specified file id.

DisallowedDirectoryException:

This is an exception class to indicate that the specified directory is not allowed to write files under, since it is not one of the allowed directories. This exception is thrown in the constructors of LocalFileUpload class where directory is specified.

RequestParsingException:

This is an exception class to indicate any errors related to parsing the request, such as reading unexpected input data or violating some parsing constraints. This exception is thrown in the uploadFiles() method of FileUpload class.

FileSizeLimitExceededException:

This is an exception class to indicate the uploaded file exceeds the file size limit. This happens during the parsing of the request. This exception is thrown in the uploadFiles() method of FileUpload class.

InvalidContentTypeException:

This is an exception class to indicate invalid content type in the request. For example, according to HTTP 1.1, the content type of the request should be "multipart/form-data". Others should be treated as invalid content type. This exception is thrown in the uploadFiles() method of FileUpload class.

1.5.2 System Exceptions

IllegalArgumentException:

This exception can be thrown by methods of all classes when invalid inputs are passed to them, such as null or empty string. Normally an empty string is checked with trimmed result.

IllegalStateException:

This exception indicates that the method is invoked at an inappropriate time, such as calling the methods of the RequestParser interface in the wrong sequence. It can also be thrown in the remote classes when the FileSystemClient is disconnected.

IOException:

This exception indicates any errors related to I/O. The exception can be thrown when retrieving the input stream of the UploadedFile.

1.6 Thread Safety

This component is thread-safe. Only the RequestParser implementation keeps state information and is not thread-safe, but it will be used in a thread-safe manner by the FileUpload subclasses. The remote classes interact with the FileSystemServer using the FileSystemClient, which can be used in a multi-threaded environment once connected. The rest of the classes are thread-safe by being immutable.

The FileSystemClient connects to the File System Server in the constructor. Once constructed the communication becomes transparent to the user. It is important to explicitly call the disconnect() method after use (with the RemoteFileUpload instance) to release the resources.

Note that the state of the class changes when removing files or disconnecting from the File System Server. Applications using this component should make sure that no other thread is calling the methods on the UploadedFile or RemoteFileUpload instance when performing the above actions.

2. Environment Requirements

2.1 Environment

Java 1.4

2.2 TopCoder Software Components

- **Configuration Manager 2.1.4** is used to load the configuration values of the FileUpload classes under a specified namespace.
- **File System Server 1.0** is used to save the uploaded files to the remote file system. Users can still retrieve the files if the requests are served from another node.
- **GUID Generator 1.0** is used to generate unique ids for filenames. This will allow uploads with the same remote file name under a request or different requests.
- **Base Exception 1.0** is used as a base class for all exceptions. It provides a consistent way to handle the cause exception.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.servlet.request

3.2 Configuration Parameters

Parameter	Description	Values
single_file_limit	Single file size limit for each uploaded file in the request, in bytes. A value of –1 indicates no limit.	Long integer. Optional
total_file_limit	Total file size limit for all uploaded	Long integer.

Parameter	Description	Values
	files in the request, in bytes. A value of –1 indicates no limit.	Optional
allowed_dirs	Multiple values specifying the allowed directories to write uploaded files under. Applicable to LocalFileUpload.	Directory paths. Required
default_dir	Default directory to write uploaded files under. Applicable to LocalFileUpload.	An existing directory path. Required if directory is not specified
overwrite	Whether overwriting of files is allowed by default. Applicable to LocalFileUpload.	Boolean value. Required if overwrite flag is not specified
temp_dir	Temporary directory to write uploaded files under. Applicable to RemoteFileUpload.	An existing directory path. Optional
ip_address	IP address of the file system server. Applicable to RemoteFileUpload.	Valid IP address. Required
port	Port that the file system server is listening at. Applicable to RemoteFileUpload.	Port number between 0 and 65535. Required.
message_namespace	Namespace used to create message factory. Applicable to RemoteFileUpload.	Non-empty string. Required.
handler_id	Id of the handler that will process requests on the server side Applicable to RemoteFileUpload.	Non-empty string. Required.

3.3 Dependencies Configuration

File System Server requires its own configuration.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

None

4.3 Demo

4.3.1 Creating FileUpload instances

```
// create instance for file upload to memory
FileUpload upload = new MemoryFileUpload(namespace);

// create instance for file upload to a directory in local file system,
// the directory can be specified dynamically
upload = new LocalFileUpload(namespace, dir);

// create instance for file upload to remote file system
upload = new RemoteFileUpload(namespace);
```

4.3.2 Manipulating results after parsing the request

```
// create the FileUpload instance
FileUpload upload = new LocalFileUpload(namespace, dir);

// parse the request, this will save the uploaded files
// to local file system
FileUploadResult result = upload.uploadFiles(request);

// get all the parameter names
String[] names = result.getParameterNames();

// get multiple values of the parameter
String[] values = result.getParameterValues("parameterName");

// get single values of the parameter
String value = result.getParameter("parameterName");

// get all the form file names
names = result.getFormFileNames();

// get multiple uploaded files with the form file name
UploadedFile[] files = result.getUploadedFiles("formFileName");

// get single uploaded file with the form file name
UploadedFile file = result.getUploadedFile("formFileName");

// get all the uploaded files
files = result.getAllUploadedFiles();
```

4.3.3 Performing file operations with file id

```
// create the FileUpload instance
FileUpload upload = new RemoteFileUpload(namespace);

// parse the request, this will save the uploaded files
// to file system server
FileUploadResult result = upload.uploadFiles(request);

// get the file id of an uploaded file from the result
// and store it somewhere (e.g. session) for later use
String fileId = result.getUploadedFile("formFileName").getFileId();

// later, we can retrieve the uploaded file from the file system server
// with that file id
UploadedFile file = upload.getUploadedFile(fileId);

// get the remote file name
String remoteFileName = file.getRemoteFileName();

// get the file size
```

```
long size = file.getSize();

// get the underlying input stream
InputStream in = file.getInputStream();

// close the input stream after use
in.close();

// remove the uploaded file from the file system server
upload.removeUploadedFile(fileId);

// disconnect from the file system server at the end
upload.disconnect();
```

5. Future Enhancements

- Add different implementations of persistence for uploaded files, such as database.
- Add the functionality to store the extra file information, so that they are available when retrieving the uploaded file with the file id.
- Develop different approaches to handle uploaded files with the same filename.
- Develop different parsing strategies for different protocols.