FileSystem plug-in(FSP) Developer Guide

Reference Documentation

Version 1.0

Author: Rolando Santamaría Masó

Control of versions

Fecha	Versión	Descripción	Autor
25/07/2012	1.0	Document creation	Rolando Santamaría Masó

Table of Contents

Introducing the FileSystem plug-in

The FileSystem plug-in(FSP for now) is an extension for the jWebSocket framework to provide support for the basics “file uploads management” operations in WebSocket applications.

Using the FSP you can easily save files from the client to the server, load in the client previously saved files, remove or list previously stored files and also send files directly to other connected clients.

WebSocket applications and file uploading

The WebSocket technology changed the traditional way to create web applications. File uploads in one of the operations that has been “updated” to re-use the benefits of a permanent and bidirectional socket connection between the client and the server. The web browsers also started supporting a new File API that allows the web applications to read and write files in the client-side.

A WebSocket file upload algorithm using the FSP can be summarized as following:

- Select a file (or many)
- Read the file content (Base64 encoding)
- Save the file(s) in the server

FSP is very easy to use

Saving a file in the server:

```
//The file
var FILENAME = “test.txt”;
var FILECONTENT = “This is the test.txt content.”;
//Calling the “fileSave” method in the jWebSocket client instance
client.fileSave( FILENAME, FILECONTENT, {
  encode: true,
  scope: jws.SCOPE_PRIVATE,
  OnResponse: function( aResponse ) {
    alert(“File saved successfully”);
  }
});
```

Loading a file from the server:

```
//The file
var FILENAME = “test.txt”;
//Calling the “fileLoad” method in the jWebSocket client instance
client.fileLoad( FILENAME, jws.FileSystemPlugIn.ALIAS_PRIVATE, {
  OnResponse: function( aResponse ) {
    alert(“The file content is: ” + aResponse.data)
  }
});
```

The “scope” argument

Some operations with the FSP requires an argument called “scope”. The scope argument indicates if the file will be private to the owner user or will be public for reading or modification purposes to others. The users can only store files in their private file-system directory (private alias targeted) or in a user's common public directory (public alias targeted).

The supported scope argument values are:

- jws.SCOPE_PRIVATE: The file will be stored in the private alias.
- jws.SCOPE_PUBLIC: The file will be stored in the public alias.

The “alias” argument

The term “alias” in the FSP represents a unique short name associated to a directory path. Example: `“privateDir” => “${JWEBSOCKET_HOME}filesystem/private/{username}/”.`

There are two core aliases in the FSP, these are:

- privateDir (`jws.FileSystemPlugIn.ALIAS_PRIVATE`): Targets the user's private directory.
- publicDir (`jws.FileSystemPlugIn.ALIAS_PUBLIC`): Targets the user's common public directory.

The FSP supports however infinite aliases, but the users can only execute the following operations on them:

- fileLoad: Load a file from the alias.
- fileExists: Check if a file exists on the alias.
- fileGetFilelist: Retrieve the alias file list.

The aliases allows the applications to share with the clients read-only directories. Usage example: The server-side application PDF report's directory.

The core

The JavaScript API

- **fileGetFilelist:** `function( aAlias, aFilemasks, aOptions );`
  - Description: Retrieves the file list from a given alias.
  - Arguments:
    - `String: aAlias: The alias value.`
    - `Array: aFilemasks: The filtering filemasks. Example:["*.txt"]`
    - `Object: aOptions: Optional arguments for the raw client sendToken method.`
    - `Boolean: aOptions.recursive: Recursive file listing flag. Default value is FALSE.`
  - Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “getFilelist”,
  alias: aAlias,
  recursive: aOptions.recursive,
  filemasks: aFilemasks
}
```

  - OnSuccess callback response example:

```
{
  "files":[
    {
      "filename":"test.txt",
      "size":48,
      "modified":"2012-07-26T23:20:39Z",
      "hidden":false,
      "canRead":true,
      "canWrite":true
    }
  ],
  "code":0,
  "msg":"ok",
  "type":"response",
  "utid":11,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"getFilelist"
}
```

- **fileSave:** `function( aFilename, aData, aOptions );`
  - Description: Saves a file in a given scope
  - Arguments:
    - `String: aFilename: The filename value. Example: test.txt`
    - `String: aData: The file content. Could be base64 encoded`

optionally.

- Object: aOptions: Optional arguments for the raw client sendToken method.
- String: aOptions.scope: The scope value. Default value is jws.SCOPE_PRIVATE.
- Boolean: aOptions.encode: Indicates if the file content require to be encoded internally before send. Default value is TRUE.
- String: aOptions.encoding: The encoding method. Currently "base64" is only supported and enabled by default.
- Boolean: aOptions.notify: Indicates if the server should notify the file save to connected clients. Default value is FALSE.

◦ Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “save”,
  scope: aOptions.scope,
  encoding: aOptions.encoding,
  notify: aOptions.notify,
  data: aData,
  filename: aFilename
}
```

◦ OnSuccess callback response example:

```
{
  "type":"response",
  "code":0,
  "msg":"ok",
  "utid":5,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"save"
}
```

- **fileLoad:** function(aFilename, aAlias, aOptions);
  - ◦ Description: Loads a file from a given alias
  - ◦ Arguments:
    - String: aFilename: The filename value. Example: test.txt
    - String: aAlias: The alias value.
    - Object: aOptions: Optional arguments for the raw client sendToken method.
    - Object: aOptions.decode: Indicates if the received file content should be “base64” decoded automatically. Default value is FALSE.
  - ◦ Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “load”,
  alias: aAlias,
  filename: aFilename
```

```
}
```

- OnSuccess callback response example:

```
{
  "type":"response",
  "code":0,
  "msg":"ok",
  "utid":6,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"load",
  "data":"This is a string to be saved into the test file!",
  "decode":true
}
```

- **fileDelete:** function(aFilename, aForce, aOptions);
  - Description: Deletes a file in the user private scope.
  - Arguments:
    - String: aFilename: The filename value. Example: test.txt
    - Boolean: aForce: Force file delete flag.
    - Object: aOptions: Optional arguments for the raw client sendToken method.
  - Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “delete”,
  filename: aFilename,
  force: aForce
}
```

  - OnSuccess callback response example:

```
{
  "type":"response",
  "code":0,
  "msg":"ok",
  "utid":14,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"delete"
}
```

- **fileExists:** function(aAlias, aFilename, aOptions);
  - Description: Indicates if a custom file exists on a given alias.
  - Arguments:
    - String: aAlias: The alias value.
    - String: aFilename: The filename value. Example: test.txt
    - Object: aOptions: Optional arguments for the raw client sendToken method.

- Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “exists”,
  file: aAlias,
  filename: aFilename
}
```

- OnSuccess callback response example:

```
{
  "type":"response",
  "code":0,
  "msg":"ok",
  "utid":7,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"exists",
  "exists":true
}
```

- **fileSend:** function(aTargetId, aFilename, aData, aOptions);
  - Description: Sends a file to a targeted client
  - Arguments:
    - String: aTargetId: The targeted client identifier
    - String: aFilename: The filename value. Example: test.txt
    - String: aData: The file content. Could be base64 encoded optionally.
    - Object: aOptions: Optional arguments for the raw client sendToken method.
    - String: aOptions.encoding: The encoding method. Default value is “base64”.
  - Low level generated token:

```
{
  ns: “org.jwebsocket.plugins.filesystem”,
  type: “send”,
  encoding: aOptions.encoding,
  data: aData,
  filename: aFilename,
  targetId: aTargetId
}
```

  - OnSuccess callback response example:

```
{
  "type":"response",
  "code":0,
  "msg":"ok",
  "utid":6,
  "ns":"org.jwebsocket.plugins.filesystem",
  "reqType":"send"
}
```

- **fileLocalLoad:** function(aDOMElem, aOptions);
  - Description: This is a call back method which gets the number of files selected from the user. Construts a FileReader object that is specified in HTML 5 specification and calls its readAsDataURL with the filename obeject and reads the file content in Base64 encoded string.
  - Arguments:
    - DOMElement: aDOMElem: File Selection event object.
    - Object: aOptions: Contains success and failure callbacks to control the files load.
    - Function: aOptions.OnSuccess: Called when a file has been loaded successfully.
    - Function: aOptions.OnFailure: Called when an error occur during the file load process.

- **setFileSystemCallbacks:** function(aListeners);
  - Description: Sets the file-system plug-in lifecycle callbacks.
  - Arguments:
    - Object: aListeners: The file-system plug-in lifecycle callbacks
    - Function: aListeners.OnFileLoaded: Called when a file has been loaded.
    - Function: aListeners.OnFileSaved: Called when a file has been saved.
    - Function: aListeners.OnFileReceived: Called when a file has been received from other client.
    - Function: aListeners.OnFileSent: Called when a file has been sent to other client.
    - Function: aListeners.OnFileError: Called when an error occur during the file-system lifecycle.
    - Function: aListeners.OnLocalFileRead: Called when a file has been loaded locally.
    - Function: aListeners.OnLocalFileError: Called when an error occur during a local file load.

The FSP storage resource

The FSP uses the server-side hard disk file-system to persist the uploaded files, allowing the chances to create NFS clusters or virtual disks mounting to provide

support for applications with a high storage space requirement.
Future versions of the FSP will support databases as the file-system storage.

Security restrictions

The FSP has a certain list of authorities that the users require to interact with the FSP operations:

- `org.jwebsocket.plugins.filesystem.load`: Allows to load files from file system.
- `org.jwebsocket.plugins.filesystem.save`: Allows to save files to file system.
- `org.jwebsocket.plugins.filesystem.delete`: Allows to delete files to file system (private scope).
- `org.jwebsocket.plugins.filesystem.exists`: Allows to know if a file exists.
- `org.jwebsocket.plugins.filesystem.send`: Allows to send files from one client to another client.
- `org.jwebsocket.plugins.filesystem.getFilelist`: Allows to retrieve file lists from file system.

Developing with FSP

The FSP offer support for the basics “file uploads management” requirements in WebSocket applications, it can be used standalone as a service or in combination with custom applications to support new requirements.

Configuring the FSP

The FSP is configurable through the main jWebSocket.xml configuration file. The following example details the plug-in configuration options:

```
<!-- $JWEBSOCKET_HOME/conf/jWebSocket.xml -->
...
<plugin>
  <name>org.jwebsocket.plugins.filesystem.FileSystemPlugIn</name>
  <id>jws.filesystem</id>
  <ns>org.jwebsocket.plugins.filesystem</ns>
  <jar>jWebSocketFileSystemPlugIn-1.0.jar</jar>
  <!-- plug-in specific settings -->
  <settings>
    <!-- Spring configuration file that contains the aliases definition -->
    <setting key="spring_config">${JWEBSOCKET_HOME}conf/
             FileSystemPlugIn/filesystem.xml</setting>
    <!-- Authorization method
       -  spring: Uses the user Spring authentication to check for
required authorities.
       -  static: Uses the user static (jWebSocket.xml users definition)
          authentication to check for required authorities (rights)
    -->
    <setting key="authentication_method">static</setting>
    <!-- Required alias for the private user's directory -->
    <setting key="alias:privateDir">${JWEBSOCKET_HOME}filesystem/private/
             {username}/</setting>
    <!-- Required alias for the user's common public directory -->
    <setting key="alias:publicDir">${JWEBSOCKET_HOME}filesystem/
             public/</setting>
     <setting key="alias:webRoot">http://localhost/public/</setting>
  </settings>
  <server-assignments>
    <!-- The TokenServer identifier that will load the plug-in -->
    <server-assignment>ts0</server-assignment>
  </server-assignments>
</plugin>
...
```

The FSP settings configuration file (Spring IOC XML configuration schema):

```
<!-- $JWEBSOCKET_HOME/conf/FileSystemPlugIn/filesystem.xml -->
...
<bean id="org.jwebsocket.plugins.filesystem.settings"
      class="org.jwebsocket.plugins.filesystem.Settings">
  <property name="aliases">
    <map>
      <!-- Alias for the ReportingPlugIn reports directory (.jrxml)-->
      <entry>
        <key><value>reportRoot</value></key>
        <value>${JWEBSOCKET_HOME}conf/ReportingPlugIn/reports</value>
```

```
      </entry>
    </map>
  </property>
</bean>
...
```

Running the server

Once the FSP is configured, the jWebSocket server can be started:

jWebSocket server logs:

```
...
...
...
2012-07-27 22:03:03,028 DEBUG - AbstractJWebSocketInitializer: Plug-in
'org.jwebsocket.plugins.filesystem.FileSystemPlugIn' loaded from classpath.
2012-07-27 22:03:03,029 DEBUG - FileSystemPlugIn: Instantiating FileSystem
plug-in...
2012-07-27 22:03:03,140 INFO  - FileSystemPlugIn: Filesystem plug-in
successfully instantiated.
...
...
...
```

Using FSP in a web application

To use the FSP in a web application we require import the following JavaScript files (download first the jWebSocket 1.0 web client package from http://jwebsocket.org/downloads/downloads.htm):

web/res/js/jWebSocket.js: Contains the jWebSocket JavaScript client.
web/res/js/jwsFilesystemPlugIn: Contains the FSP extension for the jWebSocket JavaScript client.

Example using the FSP:

```
var lClient = new jws.jWebSocketJSONClient();
lClient.open(jws.JWS_SERVER_URL, {
  OnWelcome: function (){

    //The file
    var FILENAME = "test.txt";
    var FILECONTENT = "This is the test.txt content.";
    //Calling the "fileSave" method in the jWebSocket client instance
    lClient.fileSave( FILENAME, FILECONTENT, {
      encode: true,
      scope: jws.SCOPE_PRIVATE,
      OnResponse: function( aResponse ) {
        alert("File saved successfully");
      }
```

```
    });

  }
});
```

Getting support

The FSP is part of the official jWebSocket release, has been designed and developed by Alexander Schulze and Rolando Santamaría Masó, members of the jWebSocket development team.

Contact with the jWebSocket development team at:

- jWebSocket Forum: http://jwebsocket.org
- Twitter: https://twitter.com/#!/jwebsocket

Report issues at: http://jwebsocket.org/issue_report.htm