Mantid

Script Repository Detailed Design Document

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Author | Issue | Date | Description |
| Gesner Passos |  | 18-Feb-2013 | 2nd version written |
| Gesner Passos |  | 09-Apr-2013 | Detailed GUI interaction |

# Introduction

## Purpose of this Document

This document describes the detailed design of the Script Repositoy toolbox for Mantid framework.

It will form the basis of the development of this aspect of the framework and act as a guide for maintaining the system.

## Scope of this Document

todo

## Context of this issue

Provide a version control support for the *MantidScripts* and Reduction/Analysis algorithms being developed and used in the context of Mantid from the users and instrument scientists. Give the Mantid Development Team, the opportunity to help to improve the quality and the usage of the Mantid framework by getting in touch with the *MantidScripts* being used. Allow the Development Team to switch off the old python api.

## Definition of the Terms

# *MantidScripts*: Used to define the python scripts supposed to run using the Mantid API to reduce or analyse data.

# ScriptRepository Toolbox

## Context

The goals presented [here](#_Context_of_this) has been analysed by the Development Team, and in order to allow the users and scientist to share their *MantidScripts* a git repository was created at: [mantidproject/scripts](https://github.com/mantidproject/scripts). Unfortunately, this strategy has never launched off, we suppose mainly because the users and scientist:

* were not aware of its existence, or did not see any benefit of it;
* could not understand the technical issues related to downloading and uploading files at github.
* would get confused trying to figure out which files/directories could interest them.
* need a github account and special permission to be able to publish their files.

The first version of this document presented as strategy the development of a User Interface that could just wrap the usage of the git as local repository for the users. This strategy showed some drawbacks, mainly:

* As a distributed version control, it is necessary to download the whole repository what take a considerably time for an ‘empty folder’
* The wrapper would use non-standard commands to deal with the local repository, what would cause confusion for the users and advanced users.

**A new strategy will be presented in this document** that uses a web-server as mediator between the mantid local copies and the git repository. This web-server will provide some services that will be used to deal with the main requirements of the Script Repository.

Following there is a list of requirements for this new module.

* [Script Repository GUI](#_Script_Repository_GUI)
* [Script Repository Service](#_Requirements_of_Script)
* [Script Repository WebServer](#_Requirements_for_Script)

The first one, is that will allow the user to deal with the files inside the script folder. While the second, will be responsible for background services as downloading scripts, checking updates, etc.

### Script Repository Service

#### Requirements:

##### RS1: Connection to [Script Repository WebServer](#_Requirements_for_Script) through http

##### RS2: Create a local folder that will be the ‘local repository’

##### RS3: List all files available at the [Script Repository WebServer](#_Requirements_for_Script) and locally (new creations).

##### RS4: Ability to ignore some file patterns. (for example pyc files)

##### RS5: Download single file or folders recursively.

##### RS6: Provide general information for files (remotely or local):

###### Description

###### Author

###### Last modified date

###### version identifier

##### RS7: Manage the versions of a file instance: up-to-date, locally modified, new version available, new version available and locally modified, local only.

##### RS8: Allow files to be marked for automatically updates. 🡪 NOT IMPLEMENTED YET.

##### RS9: Never override local changes. If the user decides to download a new version of a file he has changed, a backup of his own copy must be produced.

##### RS10: Upload single file or folders recursively. 🡪 NOT IMPLEMENTED YET.

##### RS11: Export its functionalities to Mantid Python API. (note: algorithms download upload)

##### RS12: Smart usage of the connection, providing a fast responsively system. Do not heavily depend on the [Script Repository WebServer](#_Requirements_for_Script) for all requests. This means that it should not request all interaction requests to the server, for example, give me the description of the current file should be avoided as server/client interaction.

### Script Repository WebServer

#### Requirements:

##### RW1: Accept read-only connection.

##### RW3: Provide information from the mantidscripts/scripts files and folders.

A nice approach here would be having a JSON file that summarize the files and folders information from the repository, that could be downloaded by the clients ([Script Repository Service](#_Requirements_of_Script)) allowing them to cope with requirement [RS12](#_RS11:_Smart_usage). (soap , restfull)

##### RW2: Allow download from all the files of mantidscripts/scripts repository

##### RW3: Ensure it is up-to-date with the mantidscripts/scripts repository

##### RW4: Provide a upload interface to publish the file/folders. (Security?) 🡪 NOT IMPLEMENTED YET.

##### RW5: Ensure that uploaded files are pushed to mantidscripts/scripts repository. 🡪 NOT IMPLEMENTED YET.

### Script Repository GUI🡪 NOT IMPLEMENTED YET.

#### Requirements:

##### SG1: Allow user to choose where to install ScriptRepository

##### SG2: List all the files available at the repository in a Tree View based Widget.

##### SG3: Show the description of the files/scripts

##### SG4: Allow the user to choose the files and directories that he want to download. He won’t be interested having all the files locally if he wants just one *MantidScripts*

##### SG5: Allow user to choose the strategy for updatings:

###### Update automatically the script when a new remote version is available

###### Alert the user when a new remote version is available

##### SG6: Allow the user to publish his own *MantidScripts*

##### SG7: Provide user clear messages on misbehaviour of the system (for example, no internet connection)

## Script Repository WebServer

We will distinguish the service provided by the webserver in two categories. One is related to enabling the download of the files and folders and the other is related to accept the upload of the files and folders.

### Download Service

The download service is simplified by letting the Apache Server to deal with almost all the requests. The strategy here is to clone the [mantidproject/scripts](https://github.com/mantidproject/scripts) repository inside one folder at the web server.

[http://download.mantidproject.org/master\_builds/scripts\_repo](http://download.mantidproject.org/master_builds/scripts_repo/)

Through this, the access to the files and folders inside the repository is as easy as adding its relative path to the URL above.

So, the web server must only perform 2 tasks:

1. Update the repository fetching changes on the main repository.
2. Provide an up-to-date [repository.json](http://download.mantidproject.org/master_builds/repository.json) file.

These services will be provided by a Jenkins job called [sync\_script\_repository](http://download.mantidproject.org/jenkins/view/All/job/is_sync_scripts_repo/).

This job was configured to check for the github [mantidproject/scripts](https://github.com/mantidproject/scripts) every minute and trigger the python script called [scriptrepositoryparser.py](http://github.com/mantidproject/scripts/system/scriptrepositoryparser.py) that walk through all the entries of the repository and creates the repository.json file.

### Upload Service

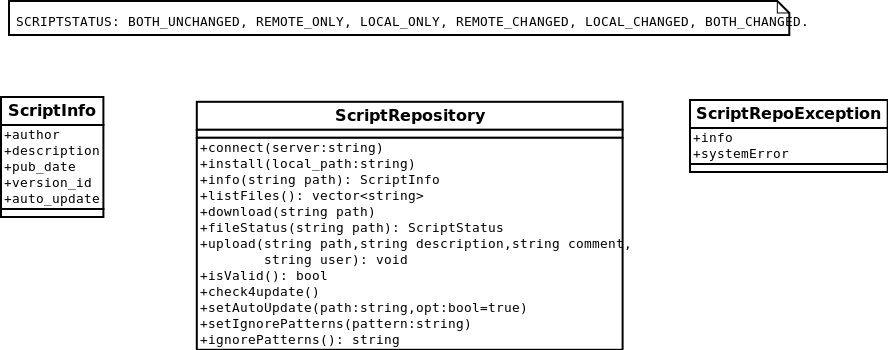
In order to allow the upload, the webserver will host a form that allows the user to provide the file, its relative path, the author, and a description of the changes. In order to avoid spams, a hidden field with a key will also be requested to process the form (this is also to force the interaction through the mantid interface). After submitting this form, it will be processed, included inside the repository, pushed to the central repository.

The form will be served as a normal http form, but internally a python-server-page will handle it. Calling the required procedures from git commands (git add, git commit, git push).

## Script Repository Service

### Definition:

### UML:



### Description:

#### ScriptRepoException:

The exceptions will be reported through the ScriptRepoException class that will provide a non technical information (what()) and a more specific and technical information (systemError()) that will allow us to deal with requirement [SG7](#_SG7:_Provide_user).

The ScriptRepoException inherits from std::exception, and its what methods provide the user information, that may be shown in the mantid logging systems, or a Qt message box widget, while a more technical information will be available through the method: systemError. Another method, called, filePath will also be used to identify the file and line that triggered the exception for the developers.

#### Connect

Allow the Script Repository Service to check the connection with the Server. This method ensures that the server is up, the network connection is available, and the link is available. [(RS1)](#_RS1:_Connection_to)

#### Install (path)

The installation consists in the creation of the folder that will get the repository of the scripts and MantidScripts. It may also have some ‘hidden’ files that will allow the Script Repository Service to cope with its functions. [(RS2)](#_RS2:_Create_a)

Argument: path – The folder to be created inside the local file system.

#### ListFiles():vector<string>

Return the list of files inside the repository. It provides a file-system like path for all the files, folders that are inside the local repository as well as remotely. For example, the listFiles for the mantidproject/scripts would be something like:

TofConv/ , TofConv/TofConverter/, …., TofConv/README.txt, TofConv/TofConverter.py, Development/, …

But, the files inside the folder must be inserted in this list as well. [(RS3)](#_RS3:_List_all)

##### Comments on the Implementation

This method is responsible for creating a local cache of all the files/folders and their current status. It must merge two list of files (the remote files and the local files). The list of the remote files will be available at repository.json, while the list of local files must be produced through inspection inside the local folder.

After having the list of both (remote) and (local), the list could be ordered, in order to be easier to merge the list. Having it ordered, a loop through both list in reverse order would be enough to produce an ordered list and the status for all the files.

Example:

A/, A/B, D, E -> remote

A/,A/C -> localy

E: greater than A/C -> insert (E)

D: greater than A/C ->insert (D)

A/B: lower than A/C ->insert (A/C) dir status = C

A/B: greater than A/ -> insert(A/B) dir status = C && B

A/ : equal A/ -> insert (A/) with curr status.

Finished.

#### Info(path): ScriptInfo

Get general information from files/folder through the ScriptInfo struct. This ScriptInfo has the following information: [(RS6)](#_RS6:_Provide_general)

##### Author

A nick name is necessary (that may allow the Mantid Team) to recognize the responsible for this file/folder.

##### Description

Probably, the best place to keep the description of the files are inside themselves. This would also encourage good practices, related to documentation of MantidScripts and projects. A rule is suggested here:

###### Directories:

The description of the directories is the description of \_\_init\_\_.py (if is a python module) or the description inside README file inside the folder.

###### README:

The content of the README file. For README file we accept any extension (.txt, .md, no extension)

###### MantidScripts:

Python modules (MantidScripts) have a recognized way to documentation. So, first of all, it will take the \_\_doc\_\_ of the module. If this one is not available, it will try to parse the module to get the first group of comments (that are usually used for description).

##### PubDate

The date of the last changing of this document. This is the same value given by the Operative System about the date of this entry.

For folders, it is the date …

#### fileStatus(path)

Provide information on the current status of the file. Currently, the following status are recognized for files:

* BothUnchanged: Local copy of the file and the central repository are iqual.
* RemoteOnly: No local copy of the file
* LocalOnly: It does not exists in the central repository
* RemoteChanged: Local copy is out-dated, there is a new version of the file inside the central repository.
* LocalChanged: Local copy was modified after the download, but no new version is available at the central repository.
* BothChanged: Local copy was modified, and there is a new version available at the repository.

For folders, the rules are:

* RemoteOnly: All children remote only
* LocalOnly: All children local only
* BothUnchanged: All children both unchanged
* LocalChanged: at least one children local changed (others must be bothunchanged)
* RemoteChanged: at least one children local changed (others must be bothunchanged)
* BothChanged: if none of the others applies.

##### Comments on the Implementation

In order to figure out the current status of the file, the following information will be considered:

* The Remote JSON file: repository.json
* The Local JSON file: local.json
* The local file itself

###### Repository.json

{

"README.md":

{

"pub\_date": "timestamp",

"description": "the description",

"directory": False

},

"TofConv":

{

"pub\_date": "timestamp",

"description": "the description",

"directory": True

}

}

###### local.json

{

“README.md”:

{

"downloaded\_pubdate": "01/02/2012 10:01:25",

“downloaded\_date”: "05/02/2012 12:01:25",

“auto\_update”: False

},

"TofConv":

{

"downloaded\_date": "01/02/2012 10:01:25",

"downloaded\_pubdate": "01/01/2012 09:01:25",

"auto\_update ": True

}

}

If the file is not inside repository.json, it is LocalOnly.

If the file is not inside the local.json, but it is inside repository.json it is RemoteOnly.

All the others files will be inside local.json and repository.json.

If the downloaded\_pubdate field of local.json is the same of pubdate in repository.json this mean that there is no new version available at the repository.

If they differ, this means that there is a new version available at the repository.

If the downloaded\_date differ from the current date of the file provided by the operative system, this may indicate that the file was changed. (There are some occasions that the file was not changed, it has just be saved again, what changes its date, but we will not deal with this case).

If there is a new version and the file was changed locally: BOTHUNCHANGED.

If there is a new version and the file was not changed locally: REMOTE\_CHANGED.

No new version and file was changed locally: LOCAL\_CHANGED.

No new version and not local change: BOTH\_UNCHANGED.

The definition of the status is not straight forward for folders, but it is for files. Because the status of the folders depend on the status of every file. But, it is necessary to be able to recognize files that have been changed locally as soon as it gets changed. The approach here is the following. The [listFiles](#_ListFiles():vector<string>) could be responsible for creating a cache of the status of every file and folder. After this, for each call to fileStatus, if the entry is a file, the current status will be evaluated, if it is the same of the cached value, it will be returned, if it is different, the current status will be returned, but the system will trigger the [listFiles](#_ListFiles():vector<string>) again. 🡪 NOT IMPLEMENTED YET.

#### Download(path)

The download get the current file available at the Script Repository WebServer and place it inside the folder (using the same relative path). [(RS5)](#_RS5:_Download_single)

If a folder is chosen, that, it must download all the children files of the folder, recursively.

If a file is chosen, than, it must download the single file.

If the file already exists locally, then, if the file has not been changed, the download will only exchange the current file with the new one. If the file has been locally changed, them, this method will back up the current file and then download the new one.

##### Comments on the Implementation

Having the copy of the git repository inside the path of the server, means that we are able to download the file by having its url. So, for a webserver running at [http://download.mantidproject.org](http://download.mantidproject.org/), if we have the script repository at path: master\_builds, then:

<http://download.mantidproject.org/master_builds/scripts_repo/README.md> returns the README.md file.

The Script Repository Service shall have a JSON local file to keep track of downloaded files. This file will be hidden for the operative system. An example of an entry of this file is available [here](#_local.json).

After downloading a new file, this method must update the JSON local file, either producing a new entry or updating the dates (the downloaded\_date will be the date returned by the operative system for the current file, while the downloaded\_pubdate will be a copy of the pubdate of repository.json file entry, the md5sum will be validated from the repository.json and will be updated).

###### Download folders

Folders will not be downloaded, they will be just created locally with the same name of the remote folder.

###### Downloading files

Before downloading, the status of the file will be checked: (through the [fileStatus method](#_fileStatus(path)))

* RemoteOnly files: they will be downloaded, create the new file/folder entry and update the local.json file.
* RemoteChanged files: the download will replace the current file.
* LocalChanged and BothChanged: the download will backup the current file and download the new version.

#### setAutoUpdate(path,bool) 🡪 NOT IMPLEMENTED YET.

Define if a file or folder should be updated automatically as soon as the ScriptRepository finds a new version available. [RS8](#_RS8:_Allow_files)

#### Upload(path, comment, description, author)

The upload will fill the form provided by the Web Service ([link](#_Upload_Service)), using the Poco library. It does not allow to submit a folder, only one file each time.

### Implementation Details

#### Internal Struct to keep information about the files

The Script Repository Service must track the information on the central repository as well as the local folder. As, explained, this will be done through the repository.json file provided by the central repository, and through the recursive inspection of the local folder.

To keep the information of these entries, the following class is suggested:

class RepositoryEntry{

Bool remote;

Bool local;

Bool directory;

DateTime current\_date;

String description;

STATUS status;

Int downloaded\_version;

DateTime downloaded\_date;

Bool auto\_upate;

String author;

DateTime pub\_date;

}

Some of these values will be available at repository.json, others will be available at local.json and others must be asked to the real file entry locally. So, from the repository.json, the following information is available:

* Remote: The file is present remotely.
* Directory: The entry is a directory.
* Description: When the entry is present remotely, this information will be available.
* Author: Definition of the author.
* PubDate: The date of the latest version.

From the local.json, the following information is available:

* DownloadedPubDate: The date of the downloaded data.
* DownloadedDate: The date of the file as soon as it was created just after download.
* AutoUpdate: The auto update option.

From the files and folders locally, the following information is available:

* Local: The file is local.
* Directory: This is a directory.
* CurrentDate: The date given by the operative system for this file.

A class is used, just to provide an easy constructor, to set up the default values.

The RepositoryEntry will be kept inside an ordered map, to allow an easy way to retrieve information on these entries. And it will be fulfilled through parsing the three entries that may change its values, to it will first parse the repository.json, than the local.json and finally, the local directory.

void parseCentralRepository()

void parseDownloadedEntries()

void parseLocalRepository()

## Script Repository GUI

### UML

### Description

The ScriptRepositoryGui will use the Model-View programming defined by Qt (see [link](http://qt-project.org/doc/qt-4.8/model-view-programming.html)).

Data

Model

View

Based on <http://qt-project.org/doc/qt-4.8/images/modelview-overview.png>.

The Framework deals with 4 components: Data, Model, View and Delegates. The data is the real data, and refers to the [ScriptRepository Service](#_Script_Repository_Service). The View refers to the User interface itself. The figure 1, shows a proposed interface. The interface is a [QTreeView](http://qt-project.org/doc/qt-4.8/qtreeview.html) and a [QTextBrowser](http://qt-project.org/doc/qt-4.8/qtextbrowser.html).

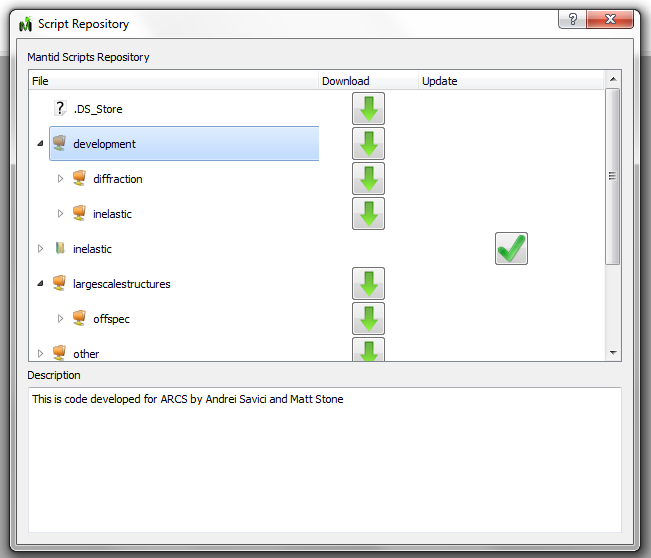
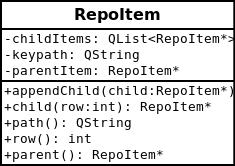


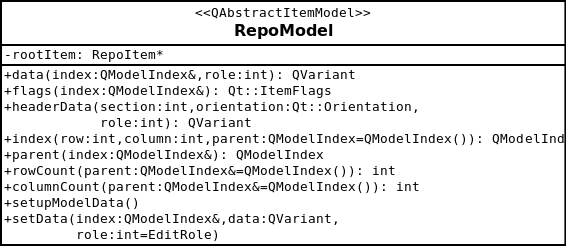
Figure The interface for Script Repository GUI

The QTreeView knows how to deal with model ([QAbstractItemModel](http://qt-project.org/doc/qt-4.8/qabstractitemmodel.html)), which must provide an abstraction of the Data. And, if necessary, the QTreeView may delegate the rendering and the interaction with the Model for the Delegates.

The data is a hierarchical data, because the files/folders have their parents folders, and each entry will have some information (columns). At ScriptRepositoryGUI the Model will be defined by the RepoModel class which specialize the [QAbstractItemModel](http://qt-project.org/doc/qt-4.8/qabstractitemmodel.html). The interaction with the Model and the view is done through the QModelIndex, which has a method called internalPointer that may be used to get access to the data structure. The real need to interact with the ScriptRepository service is the file/folder path. So, having the path as a key, the RepoModel may provide all the information required from the QAbstractItemModel.

Inspired on the Simple Tree Model Structure example (<http://qt-project.org/doc/qt-4.8/itemviews-simpletreemodel.html>) we will define the following classes:





The RepoItem class will be the data structure used to retrieve the path information that is required to retrieve information from the ScriptRepository Service.

The RepoModel will implement all the required methods from QAbstractItemModel in order to provide an interface for the ScriptRepository Service.

The RepoModel will define the following columns:

* Path: the path of the folder/file
* Status: String that will indicate the status of the entry: REMOTE\_ONLY, LOCAL\_ONLY, REMOTE\_CHANGED, LOCAL\_CHANGED, UPDATED, BOTH\_CHANGED. This column will accept the following actions, as string: Download, Upload.
* AutoUpdate: string that will indicate the auto update flag: true, false, empty (when the entry has not been downloaded). It will accept the following actions: setTrue, setFalse, that will be valid only for the entries that have been donloaded.

The Download action, valid only for the states REMOTE\_CHANGED, BOTH\_CHANGED, REMOTE\_ONLY, downloads a copy of the entry at the repository and put it inside the local folder. In the case of BOTH\_CHANGED it will backup the current file. So, after the user may merge the changes and publish afterwards.

The Upload action, valid only for the states LOCAL\_ONLY, LOCAL\_CHANGED, will publish the local files making them available at the central repository.

The ScriptRepository GUI will use the delegate in order to provide a better user experience, providing a more useful and nice user interface. It will do so, associating icons to the states of Status Column:

Updated - 

RemoteOnly - 

LocalOnly, LocalChanged - 

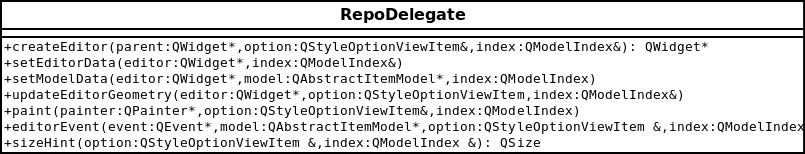
LocalChanged, BothChanged - 

Besides, clicking on these icons will trigger the actions Download and Upload as described above.

For the AutoUpdate column, it will provide a check box disabled in tri-state for non downloaded entries, and false and true for the downloaded ones.

RepoModel will also signals meta information, for example, the description for the selected entry.

To do so, the RepoDelegate class will be defined as follows:



The GUI will be created with the class ScriptRepositoryView. It is composed by the QTreeView that will allow the interaction with the RepoModel, but it will also provide some extended features:

It will show the description of the selected entry, as well as the information of the last modified date and the last author. It will allow to open the script using the Mantid Console to show the content of the MantidScripts. It must provide an easy access to the wiki page that will document the ScriptRepository as well as explaining how to retrieve old versions of files if the user need to do so.

### Implementation Details

#### RepoItem:

##### Constructor: RepoItem(QString key, RepoItem \* parent=0)

Will construct the repoitem and provide the information of who is the parent (the root will have parent = 0, and key empty).

##### appendChild(RepoItem\*)

This method will be used to update the childItems, it will be useful to mark the children of the folders. This method and the constructor will be used to setup the data information. The others method, will be used to query the data.

##### child(int row)

Return the childItems(row), wich provides a way to access the files inside the folders.

##### row():int

Returns the numbers of children (childItems.size())

##### parent():RepoItem\*

Return the parent (given at the construction)

#### RepoModel

##### Constructor: RepoModel(Object\*)

In its construction, the repomodel will access the Repository Service to populate the RepoItem objects. It will do so, iterating through the list of entries provided by [listFiles](#_ListFiles():vector<string>). It will use the constructor of RepoItem and the appendChild methods to stablish the hierarchy of the ScriptRepository with its folders and files.

##### headerData(column, Orientation, role)

The headerData method gives the RepoModel the possibility to setup the title of the columns (path, status, autoupdate) as well as some tooltips to explain how those columns should be used.

Column is the column number (0->path, 1->status,2->autoupdate)

Only the Horizontal orientation is valid.

The header will provide useful data for the following role: DisplayRole, ToolTipRole

##### Flags(QModelIndex&)

The flags define what are the user interactions available for an entry of the repository. We have that the path are selectable and enabled but can not be modified, while the status and autupdate may be changed as well.

##### data(QModelIndex, role)

The data provides the data stored from ScriptRepository Service. Given the index, it will access the RepoItem associated in order to get its path, which is necessary to query the ScriptRepository Service. From the index, it will also get the column requested (path, status, autoupdate). For the path, data will return information for the following roles:

DisplayRole: the path string

DecorationRole: Icon that indicates if it is a folder/file

For the status, it will return information for the following roles:

DisplayRole: one of the following strings: REMOTE\_ONLY, LOCAL\_ONLY, REMOTE\_CHANGED, LOCAL\_CHANGED, UPDATED, BOTH\_CHANGED.

ToolTipRole: A string describing what is possible to do (Download, Upload)

For the autoupdate, it will return information for the following roles:

DisplayRole: true of false

ToolTipRole: A string describing the actions available, to set up an auto update.

##### setData(QModelIndex,data,role)

The role will be always EditRole. The data will be always a string. The setData will be valid only for columns status and autoupdate. For the status, it will accept the Download and Upload strings, for the autoupdate, it will accept the setFalse, and setTrue. This method will ask the ScriptRepository Service to handle the request, that may be:

Download a new file or update an existing one

Download a folder

Upload one file

Setup the auto update flag

##### Index, parent, columncount and rowcount

These methods will be implemented with the help of the RepoItem class.

#### RepoDelegate

The delegate responsible to provide a nicer interface for the user.

##### createEditor(QWidget\*, QStyleOptionViewItem, QModelIndex)

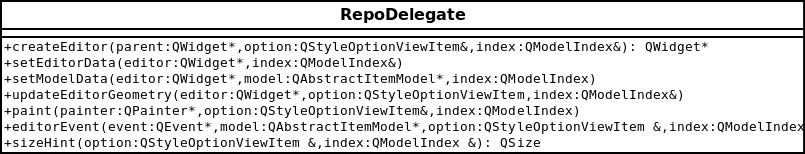
It will recognize only the index whose column is 1(status) or 2(autoupdate). For the status, it will return a pushbutton that will allow the user to click to trigger the action: Download, Upload. To the autoupdate, it will return a checkbox to allow the user to select/unselect the autoupdate option.

##### setEditorData(QWidget, QModelIndex)

Based on index, it will now if the Editor is a pushbutton or a checkbox. For the column 1, the pushbutton does not need to change the editor data. For the column 2, the checkbox, it will check if the entry has or not the autoupdate flag, and will set the checkbox accordingly.

##### setModelData(QWidget, RepoModel, QModelIndex)

This method will be called when the user triggers the push button to download or upload (column=1) or to configure the autoupdate(checkbox). It will forward the proper request to setData of RepoModel with the correspond string (Upload, Download, setTrue, setFalse).



# Uploading Strategy

For the uploading we will consider 2 possibilities: a form based page, or the proposal of a rest api. Currently, we have Python Server Pages to deal with out Mantid Web.

## Python Server Pages

If we want to continue with this technology, them, the only way to provide an upload service it through a FORM page. For example, an html form like

<form action="upload" method="post" enctype="multipart/form-data" >

Name: <input type="text" name="author"><br>

E-mail: <input type="text" name="mail"><br>

Comment:<input type="text" name="comment" ><br>

File: <input type="file" name="file">

Path:<input type="text" name="path" hidden>

<input type="submit" value="Send">

</form>

Then, inside the upload page we could deal with the posting a new file.

For example:

<%

if form.has\_key('file') and form['file'].filename:

fileitem = form['file']

# strip leading path from file name to avoid directory traversal attacks

fname = os.path.basename(fileitem.filename)

# build absolute path to files directory

target\_path = REPOPATH+form['path']

open(target\_path, 'wb').write(fileitem.file.read())

##trigger git methods for comit push….

%>

## Restful API

We could establish an API for the uploading that follows the directives of restful api. So, we could have:

### Insert File

* **Description**: Upload a File
* **URL-STRUCTURE:** <SERVER>/files/<path>
  + **Path:** The path to the folder the file should be uploaded to. This parameter should not point to a file.
* **Method: POST**
* **RequestBody:**
  + File to upload
* **Properties**:
  + **Description**: general description of the file to be inserted
  + **NickName**: identifier for the author
  + **Email**: email of the author.
  + **Title**: Name for the file.
* **Results:**
  + **Ok/200**
  + **404/400 errors**

### Updating Files

* **Description**: Update a File
* **URL-STRUCTURE:** <SERVER>/files/<path>
  + **Path:** The path to the file path.
* **Method: PUT**
* **RequestBody:**
  + File to upload
* **Properties**:
  + **Description**: general description of the file to be inserted
  + **NickName**: identifier for the author
  + **Email**: email of the author.
* **Results:**
  + **Ok/200**
  + **404/400 errors**

### Frameworks

In order to implement the second strategy, it is advisable to select a framework that helps the implementation of the API. There are plenty of frameworks available, I will just list some we may consider:

* Django: Oversized answer for the specific problem. But could be interesting if we plan to have for example, a catalog system for the scripts, or ‘advertising’ functionalities, how to do, tutorials, in the future.
* Botle, web.py, Flask: Allow routing and are micro-framework. It would fit better the solution for just this problem.

As an example, the following method, using Botle, is able to upload files:

@post('/files/<folder:path>')

def put\_path(folder=''):

if request.POST['file']:

upload = request.files.get('file')

author = request.POST['author']

filename = upload.filename

root = "/tmp/"

upload.save(root + folder +'/'+ filename)

# trigger git

return ""

else:

abort(404, "Error")

And a C++ function based on Poco is able to upload the file:

int upload(const std::string & fullpath, const std::string & author){

URI uri("http://localhost:8080");

std::string path("/files/myfolder");

HTTPClientSession session(uri.getHost(), uri.getPort());

HTTPRequest req(HTTPRequest::HTTP\_POST, path, HTTPMessage::HTTP\_1\_1);

HTMLForm form( HTMLForm::ENCODING\_MULTIPART);

form.add("author",author);

FilePartSource \* fil = new FilePartSource(fullpath);

form.addPart("file",fil);

form.prepareSubmit(req);

std::stringstream sst;

form.write(sst);

int size = sst.str().size();

fil->stream().clear();

fil->stream().seekg(0,std::ios::beg);

req.setContentLength(size+2);

std::ostream& ostr = session.sendRequest(req);

ostr << sst.str();

HTTPResponse response;

std::istream & rs = session.receiveResponse(response);

std::cout << "GOT RESPONSE: " << response.getStatus() << " " << response.getReason() << std::endl;

}

# Unit Tests Strategy

## Script Repository Service

In order to be able to configure a unit test for the Script Repository Service without depending on the network connection (for the interaction with the Script Repository WebServer), the following strategy is forseen.

Implement the interaction between these two services through a virtual protected method called:

doDownloadFile(string url, string local)

Then, the unit test could be done in a specialized class for the ScriptRepositoryService that reimplements the doDownloadFile to do a local copy of files from the ‘central repository’ (a local folder) to the ‘local repository’. Besides, the ‘central repository’ will be created ‘on-line’ and will be formed by one folder with the following entries:

central\_repository/repository.json

central\_repository/repository/README.md

central\_repository/repository/folderA/example.py

So, all the tests will be able to test the logic of the system, without depending on the network connection.

### Special Case🡪 NOT IMPLEMENTED YET.

A special unit test will be created (but will be pull off the normal building) that will really test the working of all the methods using the real doDownloadFile(url). The purpose of this unit test will be for the development team, to occasionally, insert this test in the builder and test if it is still working). It will also allow real test situations, as for example, firewall on the facilities, proxy problems, and so one. As a general advice, this test should be introduced at least once before a new release.

# Mantid Properties System

New properties will be introduced with the Script Repository Module, they are:

* ScriptRepository: URL for the [Script Repository WebServer](#_Script_Repository_WebServer).
* ScriptLocalRepository: Local folder where the ScriptRepository will be installed.