Mantid

Script Repository Detailed Design Document

# 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 and http+proxy

##### 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.

##### 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.

##### 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?)

##### RW5: Ensure that uploaded files are pushed to mantidscripts/scripts repository.

### Script Repository GUI

#### 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

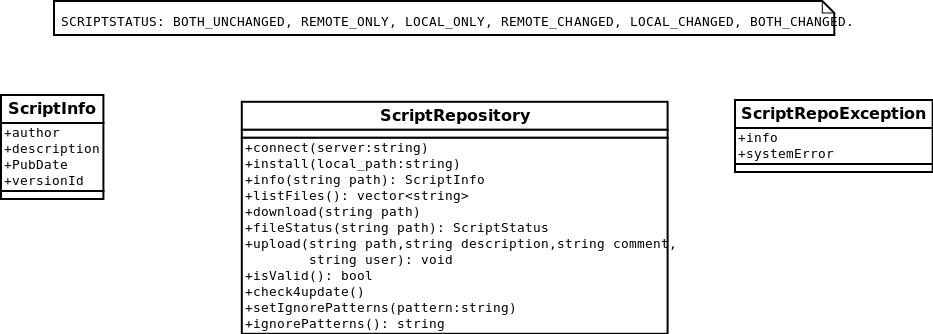
##### SG6: Allow the user to publish his own *MantidScripts*

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

## Script Repository Service

### Definition:

### UML:



### Description:

#### ScriptRepoException:

The exceptions will be reported through the ScriptRepoException class that will provide a non technical information (info) 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 throught 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":

{

"version": 2,

"date": "timestamp",

"description": "the description",

"directory": False

},

"TofConv":

{

"version": 2,

"date": "timestamp",

"description": "the description",

"directory": True

}

}

###### local.json

{

“README.md”:

{

"downloaddate": "01/02/2012 10:01:25",

“version\_downloaded”: 1

},

"TofConv":

{

"downloaddate": "01/02/2012 10:01:25",

“version\_downloaded”: 1

}

}

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 version\_downloaded field of local.json is the same of version 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 downloaddate 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.

#### 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 localhost, if we have the script repository at path: repository, then:

<http://localhost/repository/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 downloaddate will be the date returned by the operative system for the current file, while the repositorydate 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.

# 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:

do\_download\_file(string url)

Then, the unit test could be done in a specialized class for the ScriptRepositoryService that reimplements the do\_download\_file 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

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 do\_download\_file(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.