LetsMT Resource Repository - Web Service API

The LetsMT resource repository is currently implemented as a RESTful server with a version-controled file system as its backend. The PDF-version of the following documentation of the current API is attached at the end of this page.

Introduction

This is part of the documentation of the LetsMT! SMT Resource Repository, WP2/Task2.1.

The SMT Resource Repository is organised into the following collection of APIs:

1. A lower level storage, which manages data storage. It has the fundamental operations of read, write, update and delete.
2. A higher level storage, which manages the extra data services needed by clients, such as preprocessing of uploaded documents into the normalised corpus format, prior to storage.
3. A storage access database, with a file registry along with access permission, ownership and group information.
4. A group database, defining groups with user memberships, constituting the basis for data sharing.

All four APIs are available as Perl modules, and more importantly, as web service APIs.

This document describes the web service APIs.

Web Service design

As a resource repository mainly rely on the CRUD operations - create, read, update, delete, we chose to use the RESTful web service style. It is simple and easy for clients to use. Resource identifiers are provided in the URL, and the operation to perform on the given resource is specified by the request method. The four HTTP request methods POST, GET, PUT and DELETE map directly to the CRUD operations. A RESTful web service also must make sure to return suitable response codes. A GET/Read request, which gets a 200-series response can and will be cached (by the browser or internet caching proxies). From this follows that GET/Read operations must not modify the state on the server, as the response to a certain request must give the same response. Caching mechanisms will not cache GET responses with codes other than in the 200-series. Neither will POST, PUT or DELETE requests or their reponses be cached.

Client actors

There are two client actors involved:

1. The system which is issuing the request to the API. This is a http client, such as a browser or another system.

2. The effective user. Each CRUD operation must be associated with an effective user, in order for the permission verification to work.

Authentication of a system

A system is authenticated using Secure Socket Layer client certificates, a secure and widely accepted method, used by for instance banking and financial systems.

A new installation of a system that will be accessing the resource repository server, needs a signed client certificate in order to connect to the web service. It can be requested from UUP. Once you have it, install it on your file system, and configure your client software to use it when issuing requests to the SMT Resource Repository. The necessary steps are described below.

To create a private client key, the following command should be used (using a recent version of OpenSSL):

```
openssl genpkey -out yourname.key -outform PEM -pass [password arg,
see docs] -algorithm rsa
```

To request a signed certificate from UUP, create a certificate request using the private key:

```
openssl req -inform PEM -outform PEM -new -key yourname.key -out
yourname.csr -passin [password arg] -passout [password arg]
```

Then send yourname.csr to UUP and wait for the signed certificate yourname.crt. During development, UUP runs its own SSL Certificate Authority which means that the LetsMT! CA certificate will be supplied by UUP. For the moment the only instance of the web service is running on a development machine at UUP. The FQDN (fully qualified domain name) refering to this host below is opus.lingfil.uu.se.

Authentication of a user

The authentication of the effective user is a responsibility that falls on the http client which is issuing requests to the repository. The effective user is supplied as an argument to the function calls, and is assumed to be authenticated by the calling system.

The chain of trust applies - only a http client trusted to manage user authentication will be issued a signed client certificate.

Access control

Access control is implemented using the three access categories: user, group and others. For each category, read and write permission can be set. User is always the owner who created the corpus, which is done either by uploading new material or copying an existing, shared corpus. The group is set to a valid group that is defined in the group database. The granularity of the access control is on the corpus level. This means that the permissions defined for the corpus is applied to all files and directories contained in the corpus. Only the owner may edit the group and the permissions of a

corpus.

Lower level storage

A call to the lower level storage web service API consists of a resource identifier, which may point out a resource or a collection of resources. It has the following parts: Â https://<host>/storage/<path>, where <host> is the serving host, `/storage/` addresses the lower level storage, and <path> addresses the resource or collection of resources. It may be suffixed with arguments, depending on the operations, see the table below. A call with an argument looks like Â https://<host>/storage/<path>[argument:value].

Description	Method	Arguments	Upload	Download	Response code
Create a resource	POST	[uid:uid]	Content	Status	Success: 201
Update a resource	PUT	[uid:uid]	Content	Status	Success: 202
Copy branch	POST	[uid:uid][type:copy][dest:<path>]([rev:n])		Status	Success: 201
List resources	GET	[uid:uid]([rev:n])		Listing	Success: 200
Download resources	GET	[uid:uid][type:download]([rev:n])		Content	Success: 200
Delete resources	DELETE	[uid:uid]		Status	Success: 202

- uid = effective user id
- type = mode of action
- rev = revision. If omitted, HEAD is used, which denotes the latest revision
- dest = the destination path

In case the request was not completed, an appropriate error from the table in the Error appendix below is raised, along with a specific description.

Format: Content

Content is treated as binaries by the lower level storage.

Format: Status

Status messages are packaged in the simple XML format shown below. The type attribute declares if it is an error, warning or information message. The code attribute specifies the http response code. The operation attribute shows what http method was used in the request. The location attribute shows the resource identifier used in the request.

```
<letsmt-ws>
  <status type="error"
          code="response_code"
          operation="http method"
          location="identifier">descriptive message</status>
</letsmt-ws>
```

Format: Listing

The XML format for file listings is the format used by Subversion (`http://subversion.apache.org`). Two new values for the entry-nodes kind-attribute has been added: `slot` and `branch`.

```
<lists>
  <list path="identifier">
    <entry kind="slot|branch|directory|file">
      <name>name</name>
      <perm>drwrwrw</perm>
      <owner>root</owner>
      <group>users</group>
    </entry>
  </list>
</lists>
```

Lower level path formatting

The path expressions used in the lower level reveals the underlaying data

organisation:

```
/repository name/branch name/.../...
```

Each corpus resource is stored in a repository, refered to as a slot. Each repository has at least one branch, which serves as a root directory for the actual resource data. When a user uploads a new corpus, a new repository is created, and a branch is created named like the user. If a user wants to use a branch of a corpus shared by another user, that branch is copied to a new branch named after the user. This means that any resource access a user requests, is done to a branch named like the user.

Higher level storage

A call to the higher level storage web service API consists of a resource identifier, which may point out a resource or a collection of resources. It has the following parts: Â https://<host>/letsmt/<path>, where <host> is the serving host, `/letsmt/` addresses the higher level storage layer, and <path> addresses the resource or collection of resources. It may be suffixed with arguments, depending on the operations, see the table below. A call with an argument looks like Â https://<host>/letsmt/<path>[argument:value].

Description	Method	Arguments	Upload	Download	Response code

Upload and create (optional: archived)	POST	[uid:uid]([type:archive])	Content	Status	Success: 201
Upload updates (optional: archived)	PUT	[uid:uid]([type:archive])	Content	Status	Success: 202
Pass through to lower level	-	-	-	-	-

There are two reasons for implementing the higher level API:

- to hide the branch level in order to make access to repository resources transparent for the frontend
- to allow non-standard treatment of specific requests

One specific treatment is connected to the upload requests: These requests will automatically initiate pre-processing jobs. Pre-processing will be done off-line and the job will be scheduled on a resource-specific SGE queue (or any other job management system).

For status message formatting, see the section on lower level storage above.

For an upload and preprocessing request to be successful, the uploaded document needs to be of a valid type. The preprocessing step will take the necessary steps to turn it into normalised data which can be used in SMT.

The pass through functionality will catch all other variations (GET, DELETE), and after altering the path into the lower level path formatting, let the lower level handle the request.

Format: Content

Resource content should comply with the document types specified in the LetsMT project specification. If the optional archive type mode is used, the content is a *tar* archive, and its content is deployed in the place specified by the resource identifier path.

Higher level path formatting

The path expressions used in the higher level omits the branch name. It is unnecessary, as the effective user is always known, and user can only access branch data that they own. The branch is implicit, and handled by the higher level layer.

```
/repository name/.../...
```

Metadata database

Metadata in form of arbitrary key-value pairs can be connected to each addressable resource in the LetsMT repository. Access to the metadata is given through the metadata path: `https://<host>/metadata/<path>` where <path> is the resource within the higher level storage (relative to `https://<host>/letsmt`). Metadata is related to a specific branch of that resource which is implicitly given by the user-id argument (in the same way as it is defined in the higher level storage). In other words, only the owners of a resource-branch may manipualte metadata.

The actual key-value pairs are given as request arguments.

- Accessing metadata for specific resources (using a non-empty <path> in the request):

method	**arguments**	**description**	**Download**	**Response**
GET	[uid:user]	returns all metadata for <path>	listing	success: 200
GET	[uid:user][key:key]	returns <value> of key <key> in metadata for <path>	listing	success: 200
PUT	[uid:user]([key:value])*	set/update key/value pairs in metadata for <path>	status	success: 202
DELETE	[uid:user]([key:key])+	delete key <key> in metadata for <path>	status	success: 202
DELETE	[uid:user]	delete all metadata for <path>	status	success: 202

- The general search function (empty <path>):

```
GET https://<host>/metdata[uid:user][type:type]([key:value])*
```

Any key-value pair (within some limits) will be supported:

[type:resource-type]	resource type (parallel, monolingual, model)
[lang:language]	language (only for monolingual resources & LM's)
[srclang:language]	source language (for parallel resources & TM's / SMT models)
[trglang:language]	target language (for parallel resources & TM's / SMT models)
[owner:owner]	owner of the resource
[provider:provider]	resource-provider
[domain:domain]	subject domain
[type:type]	text type
[tag:tag]	tag (do we need to support multiple tags in queries?)
[minsize:size]	size filter (should be translated to numeric comparison of size key!)
[maxsize:size]	size filter (should be translated to numeric comparison of size key!)

Format: Status

Status information will be provided in the same simple XML format again:

```
<letsmt-ws>
  <status type="error"
          code="response_code"
          operation="http method"
          location="identifier">descriptive message</status>
</letsmt-ws>
```

Format: Listing

Listings will look similar to the listings at the lower level storage with the difference that they include arbitrary key-vakue pairs for each matched entry. The type of the resource is specified in the `kind` attribute (parallel|monolingual|document|lm|tm|model|data|unknown). Here is an example:

```
<lists>
  <list path="identifier">
    <entry kind="monolingual">
      <name>name</name>
      <provider>provider</provider>
      <group>group</group>
      <lang>language</lang>
      <domain>domain</domain>
      <type>text type</type>
      <size>size</size>
      <tags>tag1,tag2,tag3</tags>
      <rating>4.34231</rating>
    </entry>
  </list>
  <list path="identifier">
    <entry kind="parallel">
      <name>name</name>
      <provider>provider</provider>
      <group>group</group>
      <srclang>source language</srclang>
      <trglang>target language</trglang>
      <domain>domain</domain>
      <type>text type</type>
      <size>size</size>
      <tags>tag1</tags>
    </entry>
  </list>
</lists>
```

Storage access database

Group settings of a resource can be manipulated using the access database. A call to the storage access database API consists of a resource identifier, pointing out a slot. The branch is implicitly given, using the effective user id. Group settings are specified at **branch level**. The URL has the following parts: Â https://<host>/access/<path>, where <host> is the serving host, `/access/` addresses the database, and <path> addresses the resource or collection of resources. It may be suffixed with arguments, depending on the operations, see the table below. A call with an argument looks like Â https://<host>/access/<path>[argument:value].

The API currently does not include direct permission manipulations. The default settings are useful most cases:

- the creating user is the owner, and is the only user which may manipulate a branch's data (user has read and write access)
- group has read access. Only the owner user may change the group of a branch. To make a branch public, group is set to `public`.
- others have no access.

This scheme allows for most sharing situations.

Description	Method	Path	Argument	Download	Response code
Set group on a **branch**	PUT	/slot	[uid:uid][gid:group]	Status	Success: 202
Get group on a **branch**	GET	/slot	[uid:uid]	Listing	Success: 200

For status message formatting, see the section on lower level storage above.

Format: listing

The following format is used for the listing:

```
<letsmt-ws>
  <group>group name</group>
</letsmt-ws>
```

Group database

All users are initially members of the group *public* and a private group, with the same name as the user. Newly created resources are assigned to the private group by default.

A call to the group database API consists of a resource identifier, which may point out a collection of groups, a group or a user. It has the following parts: Â https://<host>/group/<path>, where <host> is the serving host, `/group/` addresses the group database, and <path> addresses the resource. The effective user is supplied as an argument, as it is only the creator of a group who may modify it. A call with an argument looks like Â https://<host>/group/<path>[uid:uid].

Description	Method	Path	Arg	Download	Response code
Add a user to a group	POST	/groupname/username	[uid:uid]	Status	Success: 201
(group created if necessary)					
List groups	GET	/	[uid:uid]	Listing	Success: 200
List users in group	GET	/groupname	[uid:uid]	Listing	Success: 200
Delete a user from a group	DELETE	/groupname/username	[uid:uid]	Status	Success: 202

For status message formatting, see the section on lower level storage above.

Format: listing

The following format is used for the listing:

```
<letsmt-ws>
  <group id="group1 name">
    <user id="user1" />
    ...
  </group>
  <group id="groupN name" />
    <user id="user1" />
    ...
```

```
  </group>
</letsmt-ws>
```

Appendix

Errors

409	user already member of group
403	failed to add user to group
403	not valid group
409	already exists
403	cannot find/read
403	exception caught
403	system level failure
403	init failure
403	failed to create group
403	other error
403	insufficient
403	not implemented
403	permission denied
403	not valid user