LetsMT!

Platform for Online Sharing of Training Data and Building User Tailored MT
www.letsmt.eu/

Project no. 250456

Deliverable D2.2
SMT resource repository and data processing facilities ready for integration

Version No. v1.0

June 30, 2011

Document Information

Deliverable number:	D2.2
Deliverable title:	SMT resource repository and data processing facilities ready for integration
Due date of deliverable according to DoW:	M16 – June, 2011
Actual submission date of deliverable:	June 30, 2011
Main Author(s):	Jörg Tiedemann, Matthias Zumpe, Valters Sics
Participants:	Uppsala, Tilde
Workpackage:	WP2
Workpackage title:	SMT resource repository and data processing facilities
Workpackage leader:	Uppsala
Dissemination Level:	PU
Version:	v1.0
Keywords:	data repository, data processing, resources

Version History

Version	Date	Status	Name of Author (Partner)	Contributions	Description/ Approval Level
v0.1	05/17/2010	initial	Uppsala		
v0.2	06/01/2010		Uppsala		review version
v0.3	06/20/2010		Uppsala, Tilde		pre-final
v1.0	06/20/2010		Uppsala, Tilde		final

EXECUTIVE SUMMARY

This is a description of the implementation of the LetsMT! resource repository. It includes a detailed specification of the webservice API and the implementation of the storage backend of the repository software. The webservice API enables the access to all LetsMT! resources from the web frontend and from certified clients that perform offline processes like data import and conversion and SMT model training. The backend implements the physical storage using a version-controlled file system and a database for metadata and permission control. Data processing facilities are integrated with the backend implementations.

Contents

List of Figures

List of Tables

1 Task Description

The task described here was to implement a resource repository with data processing facilities and general storage capacities for the LetsMT! platform. All information and training resources will be stored in the SMT repository. Users will be able to browse the repository and search appropriate resources for SMT training. The repository will store meta information about all resources and allows permission control to protect or to share uploaded data resources. Meta information can be used to select resources according to the user's needs. Another important part of the software concerns the implementation of data processing facilities. They are used to import data in various formats into the internal data storage with its unified data storage format. This includes mechanisms for validation, evaluation, transformation and normalization of data files. Tools for pre-processing training data are also provided that will be used by the SMT training pipeline. The system supports the upload of monolingual and parallel data resources. Additionally, it provides the possibility to align parallel documents on the sentence level in order to create the necessary resources for training translation models.

2 System Overview

Figure 1 illustrates the general architecture of the resource repository and its integration into the LetsMT! platform.

The LetsMT! resource repository has a web API that is implemented as a REST (Representational State Transfer) service with HTTP request messages and XML response messages. The Web API gives access to the LetsMT! resource repository which consists mainly of a revision control system (Subversion), a database (TokyoCabinet) and a batch-queuing system (SGE, Oracle Grid Engine). The purpose of the Web API is to enable the interaction with the repository system for uploading and downloading data, requesting and searching information and triggering batch processes. The LetsMT! resource repository system is implemented in Perl and uses the Apache server and mod_perl to handle the requests and responses to and from the client system.

Here is a list of some important terms used in this documentation:

- (Resource) Repository: The collection of all subversion repositories belonging to the LetsMT! system
- Slot: A single subversion repository containing all files belonging to the same *corpus* or *resource collection*.

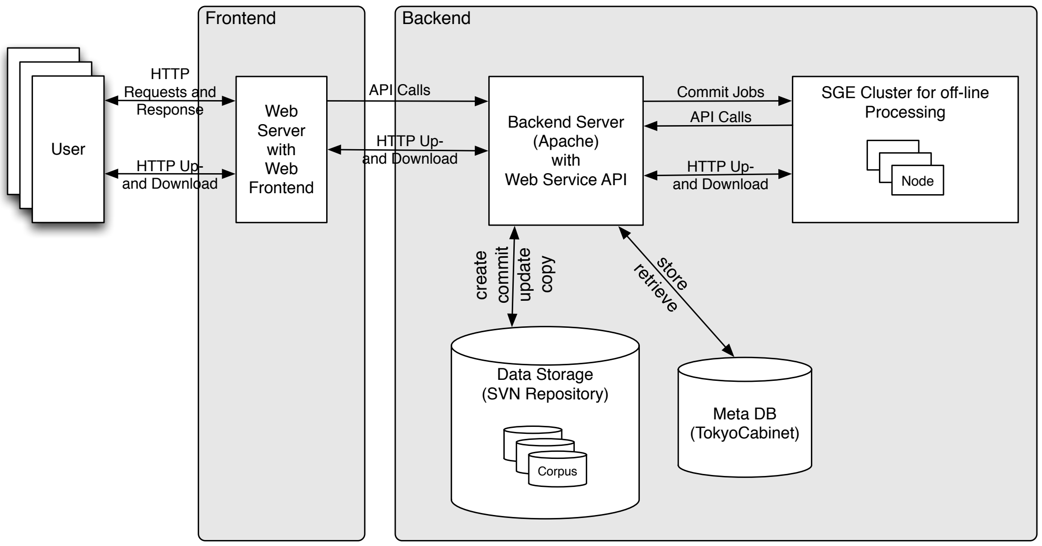

Figure 1: The system overview.

- Branch: A directory within a Slot that by convention has the name of its owning user. Only this user has write permission to that directory. A branch contains a collection of resources.
- Corpus: A special type of resource collection referring to parallel and/or monolingual documents that will be used as training data for SMT models. Corpora have a special file structure in the repository that divides the collection into different parts (more information can be found in section 5.2).
- Resource: Can be anything within the repository: a slot, a branch, a directory or a file

3 The Web Service API

The web service APIs are the interfaces to the SMT Resource Repository and are organized into the following collection of modules as seen in table 1, more closely described in section 3.3:

Module	Description
LetsMT::Repository::API::Access	Used to change and query access settings of resources (files and directories).
LetsMT::Repository::API::Admin	Provides extra information about the system, for example the status of the database.
LetsMT::Repository::API::Group	Used to create and delete user groups, add/delete users to/from groups.
LetsMT::Repository::API::Job	Used to start and manage data processing jobs like alignment and data conversion.
LetsMT::Repository::API::MetaData	Used to set, get, delete and search meta data that describes and augments the resources stored in the repository.
LetsMT::Repository::API::Storage	Can be considered the main API. It is used to create and delete resources in the repository, upload and download data, trigger import and copy resources.

Table 1: Web service API Modules.

3.1 Web Service design

As a resource repository mainly relies on the CRUD operations - create, read, update, delete, we chose to use the style of a RESTful web service. 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.

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.

3.1.1 Request Format

A call to the web service APIs is in HTTP format and has the following parts:

```
https://<host>/<API-name>/<path>?<argument=value>&<key=value>
```

where

1. <host> is the serving host,
2. <API-name> addresses the API,
3. <path> addresses the resource or collection of resources.

It may be suffixed with an <argument>, <key> or a list of both where each of them may have a specific value, depending on the operations defined in the API. Appendix —refappendix:api contains a list of example requests with their corresponding responses.

3.1.2 Response Format

The response to all requests, with the exception of download, is an XML formatted string containing a status and an optional list tag.

The `type` attribute of the `status` tag indicates if the request was successful ('ok') or resulted in an error ('error'). In case of an error the `code` attribute is set to one of the LetsMT! codes from table 2 otherwise it is set to '0'. The `location` attribute states the request URL as it was send to the server and the `operation` attribute shows the method used. The content of the `status` tag gives a more descriptive, human readable explanation of the outcome of the request where necessary.

In case of a GET requests and if information can be returned it is listed inside the `list` tag as `entry` tags. Where applicable the `list` and `entry` tags have their own `path` and/or `kind` attributes. The `path` attribute describes the logical location of the resource in the LetsMT! system.

If the response to a GET request is a file download there is no XML returned and only the file is transmitted.

```
letsmt-ws version='X'>
  <status type='ok|error' code='XXX' operation='GET|PUT|
  POST|DELETE' location='/...'>
  </status>
  <list path="/...">
    <entry kind="dir|file|group|...">
      ...
    </entry>
    <entry kind="dir|file|group|...">
      ...
    </entry>
    ...
  </list>
</letsmt-ws>
```

3.1.3 Response Codes

Table 2 gives an overview of the response and error codes used by the LetsMT! system in the status tag of the response. It also shows the HTTP status code send along with the response and the clear text status message.

LetsMT! Code	HTTP Status	HTTP Status Meaning	LetsMT! Status
0	200	OK	<depending on the request made>
0	201	Created	<depending on the request made>
0	202	Accepted	<depending on the request made>
1	409	Conflict	User '...' already member of group"
2	403	Forbidden	Failed to add user '...' to group"
3	404	Not Found	Group '...' not found"
4	409	Conflict	... exists already"
6	403	Forbidden	Can not find/read ..."
7	500	Internal Server Error	DB error: ...
8	500	Internal Server Error	System level failure: ..."
9	403	Forbidden	init failure ...
10	403	Forbidden	failed to create group ..."
11	409	Conflict	Other error: ...
12	400	Bad Request	Missing ...
13	501	Not Implemented	Not implemented: ...
14	403	Forbidden	Permission denied for ...
15	403	Forbidden	... not a valid user
16	409	Conflict	Failed to delete ...
17	400	Bad Request	Invalid ..."

Table 2: Response and Error codes, HTTP status and their meaning

3.2 Security

In order to ensure data security of the LetsMT! system, the client systems using the LetsMT! web services and the users accessing the client systems have to be authenticated. This is a two parted approach with separated responsibility.

3.2.1 Authentication of a Client System

A client system to the LetsMT! web services is authenticated using Secure Socket Layer (SSL) client certificates, a secure and widely accepted method, used by, for instance, banking and financial systems.

A new installation of a client system that will access the resource repository server, needs a signed client certificate in order to successfully connect to the web service. This certificate has to be requested from the operator of the LetsMT!

system. For such a request, a private client key needs to be created and with this a certificate request can be generated. The detailed steps and commands for this procedure are further described on the LetsMT! Wiki page on `http://opus.lingfil.uu.se/letsmt-trac/wiki/SSL`

3.2.2 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.

3.2.3 Access Control

Access control is implemented using the two access attributes: user (uid) and group (gid). A user has read and write access to a resource, a user in a group gets only read access. User is always the owner who created a branch, which is done either by uploading new material or copying an existing, shared branch. The group is by default set to the users own group that has the same name as the user. The group setting can be changed later on by the owner of the resource in order to give read access to another group of users. The granularity of the access control is on the branch level. This means that the permissions defined for a branch is applied to all files and directories contained in the branch.

3.3 The API Modules

This section describes each API module, its purpose, available commands, gives examples of commands and their return values and explains their internal implementation where appropriate.

3.3.1 Access

The Access API allows to get and change settings that control the access permissions of a branch. Any branch can only have one group name assigned to the group property at a time. All users that are members of this group have read access to all branches that have this group setting. Every branch also has an owner property set to a user name which gives write permission to that user. However, this cannot be changed by the Access API. In other words, one can only adjust read permissions with the Access API by changing the group settings. Table 3 gives an overview about the available commands.

Description	Method	Path	Arguments	Response	Response Code
Get group of a resource	GET	/<path>	uid=<uid>	status, list	Success: 200
Set group of a resource	PUT	/<path>	uid=<uid>, gid=<gid>	status	Success: 201

Table 3: Overview of commands for the Access API.

The following is an example for requesting the access settings of a certain branch, e.g. `/access/slot_name_1/user_id_2` :

```
GET 'https://host:443/ws/access/slot_name_1/
user_id_2?uid=user_id_2'
```

The result would be:

```
<letsmt-ws version="3">
  <list path="slot_name_1/user_id_2">
    <entry kind="branch" path="slot_name_1/user_id_2">
      <group>group_id_2</group>
      <owner>user_id_2</owner>
    </entry>
  </list>
  <status code="0" location="/access/slot_name_1/user_id_2"
  operation="GET" type="ok"></status>
</letsmt-ws>
```

The command for setting the group name of a branch to a new value, e.g. `/access/slot_name_2/user_id_2?uid=user_id_2&gid=group_3` could be:

```
PUT 'https://host:443/ws/access/slot_name_2/
user_id_2?uid=user_id_2&gid=group_3'
```

```
<letsmt-ws version="3">
  <status code="0" location="/access/slot_name_2/user_id_2"
  operation="PUT" type="ok">Set group to 'group_2'</status>
</letsmt-ws>
```

The group property is internally represented by a key-value pair that is stored in the meta database along with every entry (ID) that represents a branch. For example in the PUT example above the meta data ID `/slot_name_2/user_id_2` would get the key `grp` set to the value `group_3` .

3.3.2 Admin

The Admin API provides, as the name implies, administrative functionality. Currently it can return the absolute SVN path of a specific resource and status information about the meta database. Further options and support of additional maintenance tasks are planned for future releases. Table 4 shows the currently available commands.

Description	**Method**	**Path**	**Arguments**	**Response**	**Response Code**
Get svn path of a resource	GET	/<path>	type=svnpath	status, list	Success: 200
Get meta database status	GET		type=db_status	status, list	Success: 200

Table 4: Overview of commands for the Admin API.

An example for requesting the SVN path of a resource (e.g. `/slot/user/file.xml`) would be:

```
GET 'https://host/ws/admin/slot/user/file.xml?type=svnpath'
```

The response to this would be:

```
<letsmt-ws version="3">
  <list path="file:///slot/user/file.xml">
    <entry kind="svn path">
      file:///var/lib/letsmt/www-data/slot/user/file.xml
    </entry>
  </list>
  <status code="0" location="/admin/slot/user/file.xml"
          operation="GET" type="ok"></status>
</letsmt-ws>
```

3.3.3 Group

The Group API provides commands to list, create and delete groups and to add users to groups and to remove them. Every user is by default in the group 'public' and in his own group that has the same name as his user name. By adding a user to a new group the user gets read access to every branch that has that group set. Internally, groups are stored in a dedicated database including information about group owners and members. The group database uses the same database management system as the metadatabase (see section 6) with all its transaction support and logging facilities. The physical database file is located at `LETSMTDISKROOT/groups.tct` (see section 8.2 for information on environment variables in the system). Table 5 shows all available commands in the Group API.

Description	Method	Path	Arguments	Response	Response Code
Get all groups	GET	/		status, list	Success: 200
Get users of a group	GET	/<group>		status, list	Success: 200
Add user to existing group	PUT	/<group>/<user_to_add>	uid=<uid>	status	Success: 201
Create group and add effective user as first member	POST	/<group>	uid=<uid>	status	Success: 201
Delete a group	DELETE	/<group>	uid=<uid>	status	Success: 201
Delete a user in a group	DELETE	/<group>/<user_name>	uid=<uid>	status	Success: 201

Table 5: Overview of commands for the Admin API.

3.3.4 Job

The Job API handles offline jobs that run on the high-performance cluster. The High-Performance Cluster Demon (HPCD) service is an internal LetsMT! service that enables remote access to the LetsMT! cluster running the Oracle Grid Engine (SGE).[1] The service provides an interface to submit certain tasks to the grid engine and to retrieve status information of running and queued jobs. The HPCD service is implemented as an XML.RPC service in Python. The Job API calls this service with appropriate commands. Table 6 shows all available calls in the Job API.

Description	Method	Arguments	Response	Response Code
Submit a new job	POST	uid=<uid>[,<para>]	status	Success: 201
Submit/restart a job	PUT	uid=<uid>[,<para>]	status	Success: 201
Get the job status	GET	uid=<uid>	status,list	Success: 200
Get SMT training chart	GET	uid=<uid>,type=chart	binary	Success: 200
Remove a job	DELETE	uid=<uid>	status	Success: 201

Table 6: Overview of commands for the Job API. The request path determines the location of the job description file (see below). <para> are SGE parameters (key-value pairs) such as walltime and queue-name that can be understood by the High-Performance Cluster Demon (HPCD) that retrieves job requests.

The path of the request specifies the job description file which needs to be stored in the resource repository using the following simple XML format:

[1]Oracle Grid Engine, previously known as Sun Grid Engine (SGE): http://gridengine.org/

```
<letsmt-job>
  <wallTime>expected walltime</wallTime>
  <workDir>local/working/directory</workDir>
  <commands>
    <command>first-command-to-execute arguments</command>
    <command>second-command-to-execute arguments</command>
    ...
  </commands>
  <logfiles>
    <stdout>/storage/location/of/job/stdout</stdout>
    <stderr>/storage/location/of/job/stdout</stderr>
  </logfiles>
</letsmt-job>
```

Commands are sequentially executed in the order they are specified in the job description file. Additional job-specific parameters can be given such <wallTime>, <workDir> and <logfiles>. The logfile parameters specify the location of the log-files **after** finishing the job on the grid engine and uploading them to the resource repository. Default values will be used for all unspecified parameters.

The response format of GET requests depends on the arguments of the API call. The common response is status information about the job as provided by the HPCD. The `type=chart` argument allows to fetch the SMT training chart produced by the Moses experiment management system in binary format. POST requests allow to submit a job (as specified by the job description file and its location) exactly once. In other words, POST requests fail if the same job is submitted twice. PUT requests on the other hand allow to restart already submitted jobs. With this respect, PUT acts as a combination of DELETE (if the job is already queued) and a fresh POST request. Communication between the Job API and the HPCD is handled via the HPCD interface and its responses that will be stored in the metadata connected with the job description file (for example, the job ID at the grid engine).

3.3.5 Meta Data

The MetaData API gives access to the meta data stored for each resource. The storage path is used as the unique ID for database entries and arbitrary key value pairs can be stored in each of them. More details about the metadata store canbe found in section 6. In the API, there are commands to read, write and search meta data. Meta data access permissions are based on the group attribute which can be manipulated via the Access (section 3.3.1) and Group (section 3.3.3) API. Each user can only write meta data for resources that he owns (owner attribute set to users name) and read meta data from resources where he is in the group that is set for the resource.

There are two methods for retrieving meta data: by path (= database ID) and by key/value. The path/ID method can be used to list all meta data entries for a given resource The key/value method is used to search the whole meta database for matching paths/IDs based on the conditions given by keys or key-value combinations. The key/value method returns a list of matching paths/IDs and not the meta data of these paths/IDs. The key/values method furthermore accepts several special key words (CONDITIONS) that can be prefixed to the key (see list below). An example for a meta search by key/value with the CONDITION that the value of the key 'owner' should start with 'Bo' would be:

```
GET 'https://host:443/ws/metadata?STARTS_WITH_owner=Bo'
```

As mentioned before, the key/value search method returns by default only the IDs/paths of matching entries not the actual meta data records. To change that behavior the argument `action=list_all` can be appended to the request to get a listing of the IDs/paths and their full meta data records. This listing, however, can get quite long and may be time consuming to produce and fetch and should, therefore, be used with care.

It is also possible to restrict a database search to specific resources in the respository. For this, the special argument `type=recursive` is used. Database queries specified by key/value pairs are then restricted to entries within the resource specified by the path. Only the matching entries within that resource will be returned.

Description	Method	Path	Arguments	Response	Response Code
Get meta data of a resource by path	GET	/<path>	uid=<uid>	status, list	Success: 200
Search meta data by key/value	GET	/	uid=<uid>, <key>[=<value>], [action=list_all]	status, list	Success: 200
Search meta data by key/value with condition	GET	/<path>	uid=<uid>, [action=list_all], CONDITION_<key>[=<value>]	status, list	Success: 200
Search meta data by key/value recursively within a given path	GET	/<path>	uid=<uid>, type=recursive, <key>[=<value>], [action=list_all]	status, list	Success: 200
Add operations over search results	GET	/[<path>]	uid=<uid>, <key>[=<value>], action=OPERATION_key, [type=recursive]	status, list	Success: 200
Create/Extend meta data entry	PUT	/<path>	uid=<uid>, <key>=<value>	status	Success: 201
Create/Overwrite meta data entry	POST	/<path>	uid=<uid>, <key>=<value>	status	Success: 201
Delete meta data entry, key or value	DELETE	/<path>	uid=<uid>, [<key>[=<value>]]	status	Success: 201

Table 7: Overview of commands for the MetaData API.

CONDITION_ for the search mode can be one of:

ONE_OF_ (interprets the value of the selected key and the query as lists of (comma separated) strings and tries to look for data entries that contain at least one of the given strings)

ALL_OF_ (similar to ONE_OF_ but requires all query strings to be part of the data record)

STARTS_WITH_ (looks for key values that start with the given string)

ENDS_WITH_ (looks for key values that end with the given string)

MAX_ (interprets the selected field as a numeric value and looks for data entries that have a value at most as high as the given one)

MIN_ (a similar numeric comparison but looks for values equal or higher than the given one)

NOT_ (negation of the standard string-matching search)

Look at the following examples that illustrate some additional examples of the conditions explained above:

- Search for English-French parallel data:
  `...?uid=user&ALL_OF_language=en,fr&resource-type=parallel`

- Search for data sets that are in the groups 'public' or 'test':
  `...?uid=user&ONE_OF_gid=public,test`

- Search for data sets that are smaller than 1000 sentences:
  `...?uid=user&MAX_size=1000`

It is up to the LetsMT! front-end how to use the metadata and what kind of semantics to connect to specific keys. The API does not restrict the data model in any way and any key (except '_ID_' which is used for internal purposes) can be used in list context, for string comparisons or numeric queries.

Another possibility is to apply additional operations over selected keys. This can be done using the "action" argument and a prefix specifying the operation to be applied (followed by the key to be operated on). See table 7 for the exact syntax of such a request. The OPERATION_ in the *action* argument can be one of:

SUM_ (return the sum of all values for the given key for matching database entries)

MAX_ (return the maximum value for the given key among all matching database entries (together with the path to the resource with that value))

MIN_ (return the minimum value for the given key among all matching database entries (together with the path to the resource with that value))

These operations are useful to quickly find information over large amounts of matching data entries. For example, the following query can be run to obtain the size of all aligned documents for a given user:

```
../metadata?uid=user&resource-type=sentalign&action=SUM_size

<letsmt-ws version="3">
  <list path="">
    <entry type="search result">
      <SUM_size>62952992</SUM_size>
      <count>67</count>
    </entry>
  </list>
  <status code="0" location="/metadata/"
          operation="GET" type="ok">
    Found 67 matching entries
  </status>
</letsmt-ws>
```

The response code gives the cumulative sum of all size values for all entries that match the query (67 in the example) in the `<entry type="search result">`. Search operations can also be combined with recursive search requests within given resource paths. For example, the following query returns the sum of sizes of all documents in `slot/user/xml/en-fr/` and its sub-directories:

```
../metadata/slot/user/xml/en-fr/?uid=user&type=recursive&
                                 action=SUM_size

<letsmt-ws version="3">
  <list path="slot/user/xml/en-fr/">
    <entry type="search result">
      <SUM_size>290</SUM_size>
      <count>2</count>
    </entry>
  </list>
  <status code="0" location="/metadata/slot/user/xml/en-fr/"
          operation="GET" type="ok">Found 67 matching entries
  </status>
</letsmt-ws>
```

In the case of MIN_ and MAX_, the system also returns the matching resource that contains this value (in the *path* attribute):

```
../metadata/slot/user/xml/en-fr/?uid=user&type=recursive&
                                 action=MIN_size
```

```
<letsmt-ws version="3">
  <list path="slot/user/xml/en-fr/">
    <entry type="search result">
      <MIN_size path="slot/user/xml/en-fr/1996.xml">139</MIN_size>
      <count>2</count>
    </entry>
  </list>
  <status code="0" location="/metadata/slot/user/xml/en-fr"
          operation="GET" type="ok">Found 2 matching entries
  </status>
</letsmt-ws>
```

<count> in this example provides the information that the returned value is the minimum among two matching resources.

Query operators can also be combined to obtain information over more than just one key. For example, `action=SUM_size,MAX_length,MIN_height` would give all three values back (in case they exist). The response format is the same as above only with additional result tags in the entry returned:

```
<letsmt-ws version="3">
  <list path="">
    <entry type="search result">
      <MAX_length path="slot/user/min-resource">1234</MAX_length>
      <MIN_height path="slot/user/min-resource">5</MIN_height>
      <SUM_size>1025</SUM_size>
      <count>123</count>
    </entry>
  </list>
  <status code="0" location="/metadata" operation="GET" type="ok">
    Found 123 matching entries
  </status>
</letsmt-ws>
```

Note that only one operation per key is allowed, i.e. a search for `action=MAX_size, MIN_size` will not work (and only return the result of the last operation).

3.3.6 Storage

The Storage API can be considered the main repository API as it allows to create, list and delete slots, branches and directories, up- and download and view files, import them into the repository and copy branches. PUT and POST requests create slots, branches and directories as required prior to uploading a file or if no file for upload is given. Table 8 summarizes the storage commands implemented in the API.

Description	**Method**	**Path**	**Arguments**	**Response**	**Response Code**
Get listing of a resource	GET	/<path>	uid=<uid>	status, list	Success: 200
View content of a file, optional line number from/to	GET	/<path>	uid=<uid>, action=cat, [from=<number>], [to=<number>]	status, list	Success: 200
Download a resource	GET	/<path>	uid=<uid>, action=download, [type=no_archive]	download of file	Success: 200
Upload a resource and optionally import it	PUT	/<path>	uid=<uid>, [action=import]	status	Success: 201
Upload a resource or copy a branch	POST	/<path>	uid=<uid>, [action=copy&dest=<path>&gid=<gid>]	status	Success: 201
Delete a resource	DELETE	/<path>	uid=<uid>, [action=delete_meta]	status	Success: 201

Table 8: Overview of commands for the Storage API.

3.4 API-Calls from the Command Line

The API can be used from any client with a verified certificate. For example, API commands can be executed using standard command-line tools like `curl`.[2] For convenience, there is also a command line front-end that can be used to run some standard tasks:

```
letsmt_rest [OPTIONS] <command>
```

The following arguments can be specified as command-line OPTIONS:

-s slot – Name of the slot in the repository
-u user – Name/ID of the repository user
-b branch – Name of the branch within <slot> (default=<user>)
-d path – Path inside of the repository (relative to <slot>/<user>
-g group – Group name/ID
-t filetype – File type (for uploads)
-c copy_dest – Destination for copy (default=<slot>/<user>/)
-m metadata – Metadata in the form of <key>:<value>[:<key>:<value>]*
-f – Force delete (without confirmation)
-D domain – Domain of the corpus (for create)
-O owner – Owner of the data (for create), default=<user>
-P provider – Provider of the data (for create), default=<user>
-v – Verbose output

The following commands can be executed in connection with the arguments above:

- Commands for manipulating resources:

create – Create a new resource-directory (<slot>/<branch>[/<path>])
fetch – Fetch a resource (given by <slot>/<branch>[/<path>])
show – Show content of <slot>/<branch>[/<path>]
permission – Change group settings of <slot>/<user> to <group>
delete – Delete resource given by <slot>/<branch>[/<path>]
copy – Copy a <branch> in <slot> to <slot>/<copy_dest>
upload – Upload files (file name follow right after <command>)

  - If you specify only one file on the command line:
    upload destination=<slot>/<user>[/<path>]

[2] http://curl.haxx.se/

- If you specify multiple upload files (for example, file1 file2):
  upload destinations=
  <slot>/<user>/<path>/file1, <slot>/<user>/<path>/file2
- If you use absolute paths to specify upload files: the path will be ignored in the destination and only the file basename is used (<slot>/<user>/<path>/*basename(*file1*)*);

upload_corpus – Upload files to <slot>/<user>/uploads/<type>/ where <type> is either specified on the command line (-t) or taken from file extensions (check accepted upload types in the source code); the same principle for absolute and relative paths are used as with the *upload* command explained above

- Commands for manipulating groups:

create_user – Create a new user (default group = 'public')
create_group – Create a new <group> owned by <user>
add_user – Add <user> to <group> owned by <owner>
group – Show all groups or members of a specific <group>
delete_user – Delete <user> from <group> owned by <owner>

- Commands for manipulating MetaData:

showmeta Show metadata for <slot>/<branch>[/<path>]
setmeta Set metadata <metadata> for <slot>/<branch>[/<path>]
addmeta Add metadata <metadata> to <slot>/<branch>[/<path>]
delete_meta Delete all metadata from <slot>/<branch>[/<path>]

-m 'key:value' Delete a specific key-value pair
-m 'key' Delete all values for key

4 Data Upload Mechanisms

Users of the LetsMT! platform will use the public website to upload their training data, i.e., parallel, monolingual, development and evaluation data. Furthermore, users will upload files in a variety of formats to translate their contents with the LetsMT! SMT systems. All data uploads will use the same file upload mechanisms described below.

The following summarizes the file upload architecture of the LetsMT! platform from client machines to the Resource Repository:

1. The user selects a file and uploads it from the LetsMT! website;
2. The LetsMT! website forwards the file to the LetsMT! internal Application Logic Layer service;
3. The Application Logic Layer service puts the file into the Resource Repository using its Web service API.

All applications in this architecture use standard HTTP file content uploading.

Data upload in the LetsMT! website uses three different client-side upload mechanisms. Users of the LetsMT! website may use any of them depending on the client's choice, network stability, size of upload data and availability of the technology on the client's machine and web browser. These mechanisms are:

1. Standard HTML 4.0 file content upload
2. Interactive file upload using the Uploadify plugin[3]
3. Reliable large file upload using the LetsMT! Java applet

Current web standards provide no reliable mechanism to facilitate the HTTP upload of large files. Hence, the preferable method of data upload is through the LetsMT! Java applet. It provides the ability to resume the upload after temporary loss of internet connectivity by implementing the Google Data resumable protocol. Other methods do not provide such functionality.

4.1 Resumable HTTP Java file upload applet

The LetsMT! upload applet implements a reliable, platform-independent mechanism that provides the ability to resume any data upload after temporary loss of the internet connection. The solution consists of a client component (Java applet) and a sever component (ASP.NET web request handler). Both components communicate via the Google Data resumable protocol[4] that is based on HTTP/1.0 standard.

The Java applet provides an API that can be called from JavaScript to trigger file uploads and to receive upload progress events. These events are used in HTML web pages using a client-side script providing a friendly and intuitive user interface for monitoring data uploads to the web server (see figure 2 that illustrates a file upload progress).

This file upload mechanism allows to upload files up to 20 GB to the LetsMT! platform or even more if necessary. The applet sends data in multiple chunks to the

[3]http://www.uploadify.com/

[4]http://code.google.com/intl/lv/apis/gdata/docs/resumable_upload.html

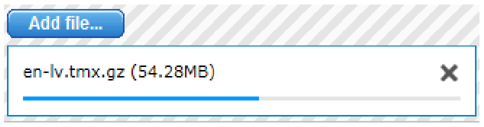

Figure 2: The user interface of the Java applet file upload progress.

server and, when the server receives one chunk, it responds and the applet sends the next chunk. In this way, it is possible to upload very large files to ASP.NET where the maximum request length is 2 GB. The applet retries sending the same chunk until it receives information from the server that data is successfully received. This enables data transfer even in cases where connections are unstable and frequently interrupted.

4.1.1 Resumable HTTP Java file upload applet API

The resumable HTTP Java file upload applet is designed as a component with an API of a number of methods and events for the communication with client-side scripting languages like JavaScript. It also implements user interface elements like a file browse button and a file open dialog.

Methods

The following methods are implemented in the LetsMT! upload applet API:

addFile - adds a file to the upload queue. Currently only single file upload are supported.
Remark: Calling this method from JavaScript may encounter security issues when uploading the file, because JavaScript typically has less security rights than Java. The Java Security Manager checks and may deny JavaScript the permission to access the file as JavaScript is in the call stack. Use the browse method instead in such cases.
Parameters: string - path to file.

browse - show the file open dialog. When the user selects a file, it is added the to upload queue.
Parameters: no parameters.

upload - start file upload.
Parameters: no parameters.

pause - pause file upload.
Parameters: no parameters.

resume - resume upload if the user has paused it. In case of upload problems, the file upload will be resumed automatically when the connection is available.
Parameters: no parameters.

cancel - cancel file upload.
Parameters: no parameters.

Events

The following events are implemented in the API:

onInitialised - the Java applet component is loaded

onFileAdded - file is added to upload queue

onProgress - report file upload progress
Event arguments:
long - total bytes to upload
long - bytes uploaded

onCompleted - file upload has completed

onError - Fatal errors occurred with the file upload or its component initialization
Event arguments: string - error description

Parameters

The upload applet configuration is done by the following parameters:

uploadUrl - File upload URL that implements the Google Resumable File Upload protocol. This is a required parameter.

maxFileSize - Maximum file size limit in bytes. 0 - no size limit. Default value - 10737418240 (10 GB).

chunkSize - Size of upload chunks in bytes. Uploading large files have to deal with server request length restrictions. This parameter allows specifying how to divide file uploads into multiple upload requests. Default value - 1048576 (1 MB).

cookie - HTTP cookie. The parameter can be used to provide information to the server in form of a HTTP cookie. LetsMT! will use it to authenticate ASP.NET sessions.

buttonImage - Path to the image to use as file browse button. The browse button is not created if the parameter is not set.

onInitialized - Client-side script method to execute when onInitialized event is fired.

onProgress - Client-side script method to execute when onProgress event is fired.

onCompleted - Client-side script method to execute when onCompleted event is fired.

onError - Client-side script method to execute when onError event is fired.

onFileAdded - Client-side script method to execute when onFileAdded event is fired.

debug - Debug parameter that should be used in web development only to get more information about the applet operation. Set to "true" to enable debug information. The default value is "false".

Integration into the webpage

The upload applet is integrated into the web site using the Java Network Launching Protocol (JNLP).[5] The applet integration consists of two components and an instantiation script. The following files have to be deployed into the LetsMT! web site:

1. ResumableHTTPFileUpload.jar - Java archive file that contains the compiled applet
2. ResumableHTTPFileUpload.jnlp - JNLP definition file

The JNLP definition file is an XML file that describes:

- General information of the applet
- Security permissions
- Versions of the Java virtual machine that can be used to run the applet
- Information where Java archive (.jar) file is stored and how to launch it

Content of JNLP definition file:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="" href="">
    <information>
        <title>Resumable HTTP File Upload</title>
        <vendor>Tilde, SIA</vendor>
    </information>
    <security>
                <all-permissions />
        </security>
```

[5]http://en.wikipedia.org/wiki/Java_Web_Start#Java_Network_Launching_Protocol_.28JNLP.29

```
    <resources>
        <j2se version="1.6+"
         href="http://java.sun.com/products/autodl/j2se" />
        <!-- allow any vendor -->
            <j2se version="1.6+" />

            <j2se version="1.5+"
         href="http://java.sun.com/products/autodl/j2se" />
            <!-- allow any vendor -->
            <j2se version="1.5+" />

        <jar href="ResumableHTTPFileUpload.jar" main="true" />
    </resources>
    <applet-desc
         name="Resumable HTTP File Upload Applet"
         main-class="ResumableHTTPFileUpload"
         width="300"
         height="300">
     </applet-desc>
     <update check="background"/>
</jnlp>
```

The following sample shows how to instantiate the Resumable HTTP Java file upload applet from JavaScript using the JNLP file within the HTML page:

```
<script src="http://www.java.com/js/deployJava.js"></script>
<script>
// Identifier, width and height of the applets visible container
var attributes = { id: "jnlpUploader", width: 81, height: 20 };
// Link to JNLP file and configuration properties
var parameters = { jnlp_href: "ResumableHTTPFileUpload.jnlp",
uploadUrl: "/UploadHandler.ashx",
cookie: "ASP.NET_SessionId=<% =Session.SessionID %>",
maxFileSize: 0,
chunkSize: 1048576,
buttonImage: "images/add_file_button_.png",
onInitialized: "onInitialized",
onProgress: "onProgress",
onCompleted: "onCompleted",
onError: "onError",
onFileAdded: "onFileAdded"
};
// Instantiate applet
var version = '1.5+';
deployJava.runApplet(attributes, parameters, version);
</script>
```

4.1.2 Google Data Resumable Protocol

The Google Data Resumable Protocol provides a specification for implementing protocols that allow file uploads to be resumed after temporal loss of internet connection. This section briefly describes the Google Data Resumable Protocol.

Initially, the client sends handshake signals to establish the file upload session. This request should not contain file content.

```
POST /upload HTTP/1.1
Host: example.com
Content-Length: 0
Content-Range: bytes */100
```

In the response, the server assigns an "ETag" of the just created upload session. This will identify all further upload requests of the file.

```
HTTP/1.1 308 Resume Incomplete
ETag: "vEpr6barcD"
Content-Length: 0
```

Then, the client can start an upload by placing the data of a file to the request content. The "If-Match" attribute has to contain the "ETag" of the file upload session. The "Content-Range" attribute specifies the part of the file that is uploaded within the request. Initially, the "Content-Range" may refer to all content of the file.

```
POST /upload HTTP/1.1
Host: example.com
If-Match: "vEpr6barcD"
Content-Length: 100
Content-Range: bytes 0-99/100
[bytes 0-99]
```

In case the request is terminated prior to receiving a response from the server, the file upload can be continued as follows. The client polls the server to determine which bytes it has received:

```
POST /upload HTTP/1.1
Host: example.com
If-Match: "vEpr6barcD"
Content-Length: 0
Content-Range: bytes */100
```

The server responds with the current byte range:

```
HTTP/1.1 308 Resume Incomplete
ETag: "vEpr6barcD"
Content-Length: 0
Range: 0-42
```

The client resumes the file upload where the server left off:

```
POST /upload HTTP/1.1
Host: example.com
If-Match: "vEpr6barcD"
Content-Length: 57
Content-Range: bytes 43-99/100
[bytes 43-99]
```

This time the server receives everything and sends its response:

```
HTTP/1.1 200 OK
ETag: "vEpr6barcD"
Content-Length: 10
```

4.2 Upload using Uploadify plugin

Uploadify[6] is a jQuery[7] plugin that uses Adobe Flash Player[8] objects to upload files. The main features of the plugin include user friendly interface for tracking upload progress and for uploading multiple files.

The uploadify plugin in the LetsMT! platform allows to upload files up to 2 GB in size which is the maximum single web request length allowed in ASP.NET. Uploadify does not support the resumable upload functionality if the upload is interrupted. Uploadify is a platform independent plugin and files are transferred with standard HTTP content upload mechanisms.

4.3 Standard HTML 4.0 file content upload

The HTML 4.0 file content upload mechanism[9] is the standard file upload functionality natively available in Web browsers where file upload over internet is allowed.

[6]http://www.uploadify.com/
[7]http://jquery.org/
[8]http://www.adobe.com/products/flashplayer/
[9]http://www.w3.org/TR/1998/REC-html40-19980424/interact/forms.html

This file upload will work as fall back mechanism if other file upload mechanisms fail due to security or technology reasons. Unlike the Java applet and the Uploadify mechanisms, this upload mechanism will not provide file upload progress information. It will notify users only when the upload is completed or an error occurred during the upload process (see figure 3).

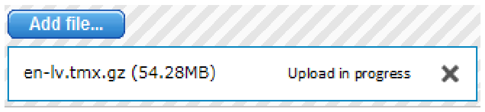

Figure 3: User interface of standard HTML file upload.

5 Data Storage

All data sets of the LetsMT! platform are stored in a revision control system. In the current implementation, we use Subversion (SVN) as it is a well tested and widely available standard system. However, the software is modular and another version control system may replace SVN or even work side-by-side with other storage backends.

5.1 Revision Control

Revision control systems are designed for dynamic repositories of textual data in multi-user environments. They typically store all repository modifications and provide tools for tracking the file history for any item in the repository. Furthermore, they naturally support data sharing and possibilities to revert to specific versions. Modifications are stored efficiently by keeping track of changes only. All of this makes them well suited for our needs in which growing resources may be accessed by multiple users.

Another important feature of common revision control systems is the possibility of creating copies of entire data branches. This is usually a cheap operation that makes it possible to obtain access to data sets without distroying data integrity for other users. Such copies are done without wasting unnecessary storage space using the same principle of storing changes only.

We use SVN as our internal storage backend and make use of its revision control features. The file/repository history can be seen as incremental backups and the branching features can be used for data sharing. Users may only modify the branches they create (and, therefore, own). However, data can be shared by copying branches to a new destination that will be owned by another user. These copies can savely be changed without altering the original data sets (which may cause unexpected behavior of data resources used by others). We add control mechanisms to manage read permissions for resources in the repository. Only branches readable for the requesting user can be copied (and viewed). In this way, we have a flexible system of sharing data (or protecting them) as we like. More details are explained below.

5.2 Data Structure

All user data is stored in a directory that can be configured via the variable `LETSMTDISKROOT` in the configuration file described in section 8.2. This directory contains all SVN repositories, one for each slot, and is the top level to a hierarchical file structure. Each SVN repository represents a slot which contains a branch for every user with write access to that slot. The branch names are by convention the name of the owning user.

Slots can be used for any kind of data. One specific use is the storage of corpus data used for training SMT systems. Within each corpus branch there are typically sub directories for special purposes: uploads, jobs and xml.

The 'uploads' directory is used for uploading new files to the repository. Only files from the 'uploads' directory can be imported and should be placed in subdirectories representing their file type, e.g. doc, tmx, pdf, etc.

The 'jobs' directory contains a file for every job that was executed, is currently running or is queued for execution (see section 3.3.4). The files in the 'jobs' directory describe the single steps each job consists of in XML format.

The 'xml' directory contains the actual corpus data which are typically the results of import jobs. There are sub-directories for every language and language pair included in the corpus. Here is an example of the general file structure in corpus branches:

```
SVN-repository (corpus, slot)
| branch_1 (user's copy of data)
| |-uploads
| |   |-doc
| |   | |-de
| |   | | |-word.oc
| |   | |-en
| |   | | |-word.oc
| |   | |-fr
| |   | |-word.oc
| |  |-tmx
| |  | |-first.tmx
| |  | |-second.tmx
| |  |-pdf
| |    |- ...
| |-jobs
|   |-job_ID_1
|   |-job_ID_2
|   |-...
| |-xml
|   |-en
|   | |-first.xml
|   | |-second.xml
|   | |-word.xml
|   |-de
|   | |-word.xml
|   |-fr
|   | |-first.xml
|   | |-second.xml
|   | |-word.xml
|   |-de-en.xml
|   |-de-en
|   | |-word.xml
|   |-de-fr.xml
|   |-de-fr
|   | |-word.xml
|   |-en-fr.xml
|   |-en-fr
|     |-first.xml
|     |-second.xml
|     |-word.xml
|-branch_2
|-...
```

6 Metadata Storage

An important design goal for developing the repository software was to allow arbitrary metadata in terms of key-value pairs stored together with resources in the repository. The focus was set on flexibility in a way that new fields and data sets in various formats can easily be added to the database during development. It has to be possible to store appropriate metadata to any resource at any location in the repository. Another important feature is that this database should still be powerful enough to allow complex search queries over the entire repository which reflects a hierarchical file structure. Standard relational database management systems do not support this degree of flexibility as they rely on pre-defined relations (tables) with fixed data types and operations over them. A recent trend is, therefore, to move from relational databased with SQL-like queries to schema-less key-value stores that do not require a fixed data model. The general idea of such a store for metadata is presented in the next section followed by some details about implementational choices in our repository package.

6.1 Key-Value Stores

A key-value store basically stores arbitrary data (values) by use of a single key. This conceptually simple strategy allows a lot of flexibility in terms of data storage without pre-defined schemas and data models. In relation to the resource repository we like to store arbitrary key-value pairs to any resource in the repository. Various kinds of information shall be stored in this way ranging from descriptive data (textual domain, ownership, language, size etc.) to status information (import/conversion status, etc), and internal information used by the LetsMT! front-end or repository back-end. Furthermore, we also need the support of repeated keys, or better, keys that may contain several values. Table 9 shows some simple examples of key-value pairs that could be stored in the database.

Resource	Key	Value
`/slot/branch/xml/en-sv`	size	10000
	resource-type	parallel
	language	en,sv
	gid	public
`/slot/branch/xml/en-fr`	size	5000
	resource-type	parallel
	language	en,fr
	gid	letsmt
`/slot/branch/xml/en`	size	200000
	resource-type	monolingual
	language	en
	domain	news
`/slot/branch/uploads/tmx/data.tmx`	status	imported
	domain	news

Table 9: Example data entries from the metadata database.

The first column in table 9 shows the data resource the metadata record is connected to. The other two columns contain keys and values. Important here is that the database is not restricted to the keys shown here but arbitrary keys can be added containing arbitrary values. Furthermore, values can also be interpreted as unordered lists, for example, in the case of *language*. Using these data sets we would like to ask complex queries such as

- Give me all public parallel data with English as either source or target language: `resource-type=parallel&gid=public&ONE_OF_language=en`
- Give me all monolingual data sets from the *news* domain that are larger than 500 sentences: `resource-type=monolingual&domain=news&MIN_size=500`

The key-value store has to be able to process such queries and to return matching resources and their associated metadata entries. The following section describes details about the implementation of such a key-value store in the LetsMT! repository.

6.2 The Metadata Manager

There are several software options for implementing the key-value store we need. We selected TokyoCabinet (`http://fallabs.com/tokyocabinet/`) that provides exactly the functionality we require. It implements an efficient schema-less data model that allows arbitrary key-value pairs associated with unique identifiers (referring to resources in our case). TokyoCabinet implements an efficient data storage that supports complex queries over the entire data set. It also comes with various APIs for common programming languages that allows straightforward integration into other systems. We implemented a metadata manager module that provides the interface between the LetsMT! Web API and the actual database (see table 10). This additional abstraction makes it possible to switch between the current choice and another implementation of a key-value store. However, the current choice with TokyoCabinet provides an efficient way of supporting the functionality specified by the LetsMT! web services.

Module	**Description**
LetsMT::Repository::MetaManager	the abstract metadata manager
LetsMT::Repository::MetaManager::TokyoCabinet	interface to Tokyo Cabinet

Table 10: Metadata manager modules.

The metadata manager modules takes care of the special operations that we need for the LetsMT! repository. It translates API calls to appropriate database operations for manipulating and retrieving data sets. For example, it takes care of copying metadata entries appropriately in case of branch-copy operations. It supports recursive deletions in case of any removal of resources. We use transactions to secure data integrity in a multi-threaded, multi-user environment. We also added logging of transactions and automatic backups to improve data security. Table 11 lists the essential files that are used by the metadata manager.

File	Description
metadata.tct	the actual database (tct = Tokyo Cabinet Table)
metadata.tct.<DoM>	daily backup of the database made in a round-robin fashion (<DoM> = Day of Month)
metadata.tct.<DoM>.log	transaction logfile (a self contained script that performs the same transactions as done after backup <DoM> up to the backup thereafter (or the current version of the database)

Table 11: Database files produced by the MetaManager module in connection with Tokyo Cabinet. All files are relative to `LETSMTDISKROOT/` .

7 Data Processing and Import

The repository software package includes several modules for importing data files and pre-processing resources. These tools can be run as batch processes offline. They use the web service API described above to communicate with the repository. Data import includes validation, conversion and normalization of documents in various formats (more details in the next section below). Pre-processing modules include basic text processing tools for sentence boundary detection and tokenization. Furthermore, we include tools for automatic sentence alignment and data export (conversion from the internal format to Moses format). The general library structure of the implementation is as follows:

LetsMT::Import — Modules for importing documents into the repository including reader modules (for external data formats), writer modules (for the internal LetsMT! format) and import handlers

LetsMT::DataProcessing — Modules for data processing including tokenizer, sentence boundary detectors (splitters) and text normalization modules

LetsMT::Export — Modules for exporting (converting) data resources to different formats

7.1 Data Import

The repository software implements various tools for importing data files in different formats. Import modules are bundled in the Perl modules `LetsMT::Import` and are called from the front-end `letsmt_import`. The general usage of this script is as follows:

```
letsmt_import -u user -s slot -p path [OPTIONS]
```

The command-line arguments specify the resource and import options that will be used. Obligatory parameters are the following:

user: The ID of the effective user requesting the import.

slot: The name of the slot that contains the resource to be imported.

path: The path to the resource to be imported relative to `slot/user/`. The resource is expected to be available in the uploads directory of the slot in a sub-directory that corresponds to the format of the document (for instance, `slot/user/uploads/tmx/`). The name of the sub-directory below `uploads/` determines the data format. Monolingual documents should be placed in appropriate sub-directories using proper language IDs within those format-specific directories. For example, English PDF files should be uploaded to `slot/user/uploads/pdf/en/` or sub-directories thereof.

-h: Show the help screen with usage information.

Other import-specific options `[OPTIONS]` are currently not implemented but will be added when necessary.

7.1.1 Resource Handlers

The main import module (`LetsMT::Import`) manages individual resource handlers for every data format supported. These handlers implement format-specific validation and conversion methods. The import module tries to convert the given resource with all possible resource handlers that match the data type of the given resource. Possible types are taken from the path information (see above) and the file extension. Any import process includes a validation step and a conversion step. If a conversion process succeeds, it will upload the newly created resources and stop. The import module also constantly updates status information in the meta information of the repository that belong to the original resource.

In the current implementation we support the following data formats with dedicated import handlers:

- Aligned parallel data:

  ```
  LetsMT::Import::TMX
  LetsMT::Import::XLIFF
  LetsMT::Import::Moses
  ```

- Monolingual text documents:

  ```
  LetsMT::Import::PDF
  LetsMT::Import::Text
  LetsMT::Import::DOC
  ```

- Compressed data and archives:

  ```
  LetsMT::Import::gz
  LetsMT::Import::zip
  LetsMT::Import::tar
  ```

Compressed data and documents in data archives will be unpacked and each individual file will be processed by appropriate import handlers. Special is also the import handler for uploads in Moses plain text formats. This format requires two aligned plain text files that need to be converted into the internal LetsMT! format. However, import handlers work on individual uploads only. Therefore, uploads in Moses format need to be done in tar-archives containing the text files with the standard file name conventions that include language IDs as their extension. The Moses import actually supports archives with more than two language in parallel, which allows the upload of parallel corpora with any number of parallel versions. Here is an example of the content of such a Moses tar file (`europarl.tar`) that could be used:

```
> tar -tf europarl.tar

europarl.en
europarl.fr
europarl.nl
europarl.sv
```

Note that the import handler does not check whether these files are correctly aligned or not. All language files must have the same number of lines and all lines need to be aligned with each other.

Similarly, TMX may contain more than two language in parallel. The import script will always create all language combinations to store sentence alignments in the internal LetsMT! format. For instance, using the Moses example from above, we obtain the following alignment files:

```
slot/user/xml/en-fr/europarl.xml
slot/user/xml/en-nl/europarl.xml
slot/user/xml/en-sv/europarl.xml
slot/user/xml/fr-nl/europarl.xml
slot/user/xml/fr-sv/europarl.xml
slot/user/xml/nl-sv/europarl.xml
```

Note that we only store alignments in one direction as they are symmetric. We always sort (alphabetically) by language IDs in order to be consistent within the entire repository.

To finish up our example, individual language files will be created as follows:

```
slot/user/xml/en/europarl.xml
slot/user/xml/fr/europarl.xml
slot/user/xml/nl/europarl.xml
slot/user/xml/sv/europarl.xml
```

Another important feature of import handlers is that they contain text processing methods. These methods influence the conversion process and can be changed and adjusted to one's needs. There are three categories supported in the current implementation:

Normalizer: Modules that can normalize a given text according to some substitution rules or any pattern matching operation.

Splitter: Modules that split a given text into sentences.

Tokenizer: Modules that split sentences into tokens (words).

Appropriate text processing tools can be specified when instantiating the import object. Default settings depend on the data format and the responsible handler that converts the resource. Most resource types apply default tools that do not alter the data in any way. This is different for the general text conversion module that includes a whitespace normalizer, a stylistic ligature converter and a generic sentence boundary detector. Conversions from PDF and MS Word documents are done via plain text formats and, therefore, apply the same default text processing modules as defined for direct conversions from text. More details on text processing modules will be given in section 7.2.

7.1.2 Status Information and Meta Data

Any import process sends runtime information to the repository in order to report the status of the process. In general, the `status` field will be used in the metadata record of the resource to be imported. Table 12 summarizes the different messages that can be found.

Status Message	Notes
importing	started to import a resource
failed to fetch	could not download the resource from the repository
failed to validate as <format>	data validation reported an error
failed to convert	file conversion failed
failed to recognize format	data format not supported (or not possible to identify)
imported	success of the entire import process

Table 12: Status messages from import processes.

The import module also uploads log files in case something went wrong with the validation of the resource (if there are any log files produced in validation). The logging information depends on the resource handler and the validation process. Currently, log files are only produces for XML validation (with the extensions '.dtd.err' for DTD validation and '.xs.err' for XML Schema validation). Further log information is available in the general log file of the repository package.

Import also produces appropriate metadata for the resources created by the conversion process. This depends on the resource type that has been produced (see table 13).

Resource Type	Key	Value
corpus files	size	<number of sentences>
	resource-type	corpusfile
	language	<language ID>
sentence alignments	size	<number of alignments>
	resource-type	sentalign
	language	<list of language IDs>
	source-language	<source language ID>
	target-language	<target language ID>

Table 13: Automatically added metadata for converted resources.

Cumulative counts for a corpus containing many documents can be retrieved by special queries over the metadata DB (see explanations of `action=SUM\_size` in section 3.3.5).

Additionally, import also updates the list of languages and language pairs included in a resource collection. This information is stored in the metadata associated with the user's branch in the current slot (`/slot/user`), see table 14.

Key	Value
langs	<list of languages>
parallel-langs	<list of language pairs>

Table 14: Global language information.

<list of languages> is a comma-separated string of language IDs (like `de, en, fr`) and, similarly, <list of language pairs> is a list of language ID pairs (like `en-de, en-sv, fr-de`).

7.1.3 Import of Parallel Corpora

The repository software supports the import of parallel corpora that are stored in the format used by the OPUS corpus. This allows to quickly include large data sets that have been coded in a format that is very similar to the one used internally by the LetsMT! project. Here, we can avoid most of the conversion tasks and simply change the links and paths in the aligned documents. The `letsmt_import` script can be used for this kind of import with the additional flag `-o` :

```
letsmt_import -o -u user [-a] [-d homedir] [-f] corpus [dest]
```

The command-line arguments specify the source and the target of this upload:

user: The ID of the effective user requesting the import.

dest: The destination name of the corpus (optional). The corpus will otherwise be placed in a slot called `corpus` by default. The corpus is always put into the `public` group and will be owned by `user` .

corpus: Name of the corpus (should be the root directory of the corpus); the script expects the same sub-directory structure that is used by OPUS:

- `corpus/raw` contains untokenized corpus data in XML with sentence boundary markup (the same format as used by LetsMT!)
- `corpus/xml` contains gzipped sentence alignment files in the same format as used internally by LetsMT!; for each language pair there is a separate sub-directory with sentence alignment files in XCES format for each document pair

homedir: The (optional) home directory of the corpus collection. `corpus` should be a sub-directory of this directory. By default the script assumes that the corpus can be found in the current directory.

-f: Force overwriting existing files. By default the script skips existing files and uploads only non-existing ones.

Importing resources also triggers a query for parallel documents within the slot. Each new resource will be aligned to other resources with **exactly** the same name and path accept for the sub directory that identifies the language. Sentence alignment processes will be initiated automatically (with standard settings) if the sentence alignment file is **not** part of the same import. Here is an example:

original document: `upload/pdf/en/doc.pdf`

imported as: `xml/en/doc.xml`

already in the slot: `xml/fr/doc.xml` and `xml/de/doc.xml`

triggers alignment of: two document pairs
`upload/pdf/en/doc.xml` - `upload/pdf/fr/doc.xml` and
`upload/pdf/de/doc.xml` - `upload/pdf/en/doc.xml`

creates: `upload/pdf/en-fr/doc.xml` and `upload/pdf/de-en/doc.xml`

For more information on alignment: see section 7.3.

7.1.4 Summary of the Implementation

Table 15 summarizes all modules implemented in the current version of the repository software.

7.2 Text Processing

The repository software package includes several modules for processing textual documents. These modules include tools for typical pre-processing tasks that are necessary in order to prepare data for the use in MT training procedures. There are basically three categories of tools: Text normalizers, sentence boundary detection tools (splitter) and text tokenizer. Table 16 summarizes the tools implemented so far.

The package is easy to extend with additional modules in each category. More details are given in the source code documentation.

7.2.1 Command Line Tools

There are two command line tools for offline tokenization and de-tokenization of corpus files.

Module	Description
LetsMT::Import	main module that handles data imports
LetsMT::Import::XCESWriter	module that creates data in the internal LetsMT! format
LetsMT::Import::OPUS	module for importing OPUS corpora
Import Handlers	
LetsMT::Import::TMX	validate and convert TMX
LetsMT::Import::XLIFF	validate and convert TMX
LetsMT::Import::Moses	validate and convert TMX
LetsMT::Import::Text	validate and convert TMX
LetsMT::Import::PDF	validate PDF and extract plain text
LetsMT::Import::DOC	validate PDF and extract plain text
LetsMT::Import::tar	validate tar-archives and unpack them
LetsMT::Import::gz	validate gzipped files and unpack them
LetsMT::Import::zip	validate zip archives and unpack them
Data Readers	
LetsMT::Import::TMXReader	parse TMX files and extract aligned translation units
LetsMT::Import::XLIFFReader	parse XLIFF files and extract source and target texts
LetsMT::Import::TextReader	normalize plain text and detect sentence boundaries
LetsMT::Import::MosesReader	read aligned plain text files

Table 15: Modules for importing data resources.

```
letsmt_tokenize [OPTIONS] infile outfile
```

Here is a description of the command-line arguments and optional parameters (`OPTIONS`):

infile: The input file (with path).
outfile: The output file. Note that this file will be overwritten without warning!
-t type: Optional type of tokenizer (europarl, uplug, whitespace) to be used (default = europarl tokenizer).
-l lang: Optional language of the input document (may have some influence of the behavior of the selected tokenizer (-t). Default = en
-i informat: Optional input file format (default = xml).
-o outformat: Optional output file format (default = xml).

Another script can be used to de-tokenize a given document:

```
letsmt_detokenize [OPTIONS] infile outfile
```

Module	Description
LetsMT::DataProcessing::Normalizer::	
No	just copy (no changes)
Whitespace	remove repeated white space characters
Ligatures	replace stylistic ligatures with their multi-character equivalents
SeparateHeader	length heuristics for separating headers from paragraphs (short lines)
Chain	meta-module to build cascaded normalization chains
LetsMT::DataProcessing::Splitter::	
No	just copy (no splitting)
Lingua	sentence splitter based on Lingua::Sentence (from CPAN based on Europarl tools)
LetsMT::DataProcessing::Tokenizer::	
No	just copy (no tokenization)
Whitespace	split on whitespace characters
Europarl	tokenizer with language-specific lists of “non-breaking prefixes” based on the Europarl tools
UPlug	generic tokenizer based in regular expressions taken from the UPlug tokenizer

Table 16: List of current data processing tools.

The same optional parameters as for the tokenization script are supported and all command line arguments have the same semantics as explained above.

7.3 Sentence Alignment

Offline sentence alignment can be run with the following command line tool:

```
letsmt_align -u user -s slot -e src -f trg [-m method] [-h]
```

The command-line arguments specify the resources to be aligned and the alignment method to be used:

user: The ID of the effective user requesting the alignment.

slot: The name of the slot that contains the resources to be aligned.

src: The path to the source language resource to be aligned to the target language resource. This is relative to `slot/user` as usual.

trg: The path to the target language resource to be aligned.

method: The sentence alignment method to be used. Default is the length-based sentence aligner as proposed by [1].

-h: Show the help screen with usage information.

The `letsmt_align` script fetches the resources from the repository and aligns them locally (in a temporary working directory). The result of the sentence alignment in form of a stand-off sentence alignment file will be uploaded to the appropriate place in the repository.

The actual alignment tools are implemented in the following modules:

Module	Description
LetsMT::Align	Alignment factory for repository resources that calls appropriate alignment modules
LetsMT::Align::OneToOne	sentence alignment strictly one-to-one
LetsMT::Align::GaleChurch	A length-based sentence aligner (default)

Table 17: List of modules for automatic sentence alignment.

The OneToOne aligner is useful to create sentence alignment files for already aligned resources. In this case, sentences are aligned one-by-one in the same order they appear. The length-based aligner (GaleChurch) allows a number of parameters to influence the behavior of the algorithm. For example, prior link-type probabilities can be changed, mean and variance of the length-matching cost function can be adjusted and the search window can be changed to improve alignment efficiency (the search window limits the relative distance of sentences in bitext space to a specific maximum). New alignment methods can easily be added by implementing additional modules that can be called from `LetsMT::Align`.

The alignment modules and the `letsmt_align` tool also add information to the meta database about the result of the alignment process. This information can be used to judge the quality of the automatic process in order to detect possible errors. Figure 4 shows an example for such a metadata record added to a sentence alignment file `testslot/testuser/xml/en-sv/1988.xml`.

The information added depends very much on the alignment method applied. Typical properties useful for error detection are

alignment-cost: Total value of the alignment cost for the given bitext according to the cost function used in the alignment algorithm. Lower values are better.

alignment-link-cost: Average cost per sentence link (alignment cost divided by number of links).

link-types: Counts for all link types found. Good alignment should include mainly one-to-one alignments (1:1)

7.4 Data Export

Finally, there are several modules for exporting repositories in various data formats. The main purpose of export tools is to fetch data sets and to convert them into the

```
<letsmt-ws version="3">
  <list path=">
    <entry path="testslot/testuser/xml/en-sv/1988.xml">
      <aligner>LetsMT::Align::GaleChurch</aligner>
      <alignment-cost>125.680891718419</alignment-cost>
      <alignment-type>automatic</alignment-type>
      <average-link-cost>0.843495917573281</average-link-cost>
      <gid>testgroup</gid>
      <language>en,sv</language>
      <link-types>1:1=147,2:1=1,2:2=1</link-types>
      <nr-source-sents>152</nr-source-sents>
      <nr-target-sents>151</nr-target-sents>
      <owner>testuser</owner>
      <resource-type>sentalign</resource-type>
      <size>149</size>
      <source-document>
        testslot/testuser/xml/en/1988.xml
      </source-document>
      <source-language>en</source-language>
      <status>updated</status>
      <target-document>
        testslot/testuser/xml/sv/1988.xml
      </target-document>
      <target-language>sv</target-language>
    </entry>
  </list>
  <status code="0"
  location="/metadata/testslot/testuser/xml/en-sv/1988.xml"
  operation="GET" type="ok">Found matching path ID. Listing all of
  its properties</status>
</letsmt-ws>
```

Figure 4: Meta data added by automatic sentence alignment using the Gale & Church approach.

format used by Moses and other SMT training tools. Table 18 summarizes modules implemented in the repository package.

Module	**Description**
LetsMT::Export	main module that handles data export
LetsMT::Reader	data reader factory
LetsMT::Writer	data writer factory
Data Reader	
LetsMT::Export::Reader::Corpus	fetch and read an entire corpus (including all its files)
LetsMT::Export::Reader::XCES	read a parallel corpus specified by a given sentence alignment file (in XCES format)
LetsMT::Export::Reader::XML	read monolingual corpus data
LetsMT::Export::Reader::Converter	reader that require some kind of conversion first; currently this supports reading of PDF and DOC files
Data Writer	
LetsMT::Export::Writer::Moses	write parallel data in Moses plain text format
LetsMT::Export::Writer::Text	write monolingual data in plain text format
LetsMT::Export::Writer::XML	write monolingual data in standalone XML format

Table 18: Modules for exporting data resources.

The system can easily be extended by additional reader and writer modules in order to support additional data formats. Note that the general LetsMT::Reader module also supports plain text, TMX and XLIFF via the LetsMT::Import modules (TextReader, TMXReader, XLIFFReader). Furthermore, the LetsMT::Writer supports writing the internal LetsMT! format through the XCESWriter module in LetsMT::Import. All data conversion can also be done for local documents without fetching them from an online repository. There is a conversion tool that supports the use of export functions from the command line:

```
letsmt_convert [OPTIONS] infile outfile
```

Here is a brief overview of command line options:

infile: The input file to be converted (in case of Moses format: add another file as input just following the first one)

outfile: The output file (basename without language extension in case of Moses format; file name used in each language/language pair sub-directory in case of the xces format)

-i format: The input format (default = xml); possible values: *xml, xces, text, tmx, xliff, moses, pdf, doc, parallel* (collection of parallel documents), *monolingual* (collection of monolingual XML documents)

-o format: The output format (default = text); possible values: *text, xml, moses, align* (sentence alignment file), *xces* (sentence alignment file + aligned corpus files in XML)

-u user: Optional user ID

-s slot: Optional slot name

-b splitter: Optional sentence boundary detection used when reading (default = none); possible value: *lingua*

-t tokenizer: Optional tokenizer used while reading (default = none); possible values: *europarl, uplug, whitespace*

-n normalizer: Optional list of normalizers used while reading (default = none); possible values: a comma-separated list of *whitespace*, *header* (separate header lines), *ligatures*

7.4.1 Command Line Tools

The following command-line tool can be used to fetch data from a repository in the format needed for SMT training with Moses.

```
letsmt_fetch [-u user] [-v] [-h] smt-config
```

smt-config: SMT configuration file (specifying the data to be fetched and converted); more details below.

user: Optional user ID (default: take user from smt-config)

-v: Switch on verbose output (print progress)

-h: Show the help screen with usage information.

SMT configuration files use a simple standalone XML syntax to specify data sets that shall be fetched and converted to Moses training formats. Figure 5 shows an example of a configuration file.

There are several parts in the SMT configuration file illustrated in the figure. The first three tags specify the effective user (`<user>`) and the source and the target language (`<srclang>` and `<trglang>`). Parallel data sets are specified in the sections for translation models (`<tm>`), tuning data (`<tuning>`) and evaluation/test data (`<evaluation>`). Monolingual data sets are listed in the section for language models (`<lm>`). Each data section has a `name` attribute specifying the filename in which the data will be stored. Selected data resources are listed in

```
<SMT>
  <user>testuser</user>
  <srclang>sv</srclang>
  <trglang>en</trglang>
  <tm id="1" name="parallel1">
    <corpus>storage/user/EUconst/xml/en-sv</corpus>
    <corpus>storage/user/OpenOffice/xml/en-sv</corpus>
    <filter>
       <sample skip="1000" />
    </filter>
  </tm>
  <tm id="2" name="parallel2">
    <corpus>storage/user/KDE4/xml/en-sv</corpus>
    <corpus>storage/user/Europarl/xml/en-sv/ep-03-07.xml</corpus>
    <corpus>storage/user/Europarl/xml/en-sv/ep-03-08.xml</corpus>
    <corpus>storage/user/Europarl/xml/en-sv/ep-03-13.xml</corpus>
    <filter>
       <links type="1:1" />
    </filter>
  </tm>
  <lm id="1" name ="monolingual">
    <corpus>storage/user/EUconst/xml/en</corpus>
    <corpus>storage/user/KDE4/xml/en</corpus>
    <filter>
       <sample skip="1000" />
    </filter>
  </lm>
  <tuning name="tune">
    <corpus>storage/user/EUconst/xml/en-sv</corpus>
    <filter>
       <sample size="500" />
    </filter>
  </tuning>
  <evaluation name="eval">
    <corpus>storage/user/EUconst/xml/en-sv</corpus>
    <filter>
       <sample skip="500" size="500" />
    </filter>
  </evaluation>
</SMT>
```

Figure 5: An example configuration file for SMT training data.

`corpus`) tags. `letsmt_fetch` retrieves the entire corpus if the path of the corpus ends with the sub-directory specifying the language or language pair. Otherwise, it is also possible to specify individual corpus files or sentence alignment files (see, for example, the last three corpus tags in the second `tm` in figure 5).

Special filters (`<filter>`) can be used to select certain parts of the data. This is, for example, helpful to divide data into disjoint sets for training, tuning and testing. For example, in figure 5 we `skip` the first 1000 entries in the first `tm` and the `lm` (`<sample skip="1000" />`). These entries will be used for tuning and testing: The `tuning` set is taken from the same resource but uses only the first 500 entries (`size="500"`) and the `evaluation` set skips the first 500 entries and takes the following 500. Another option is to add a filter for specific sentence alignment types. For example, in the second `tm` we retrieve only sentence alignment units that are aligned one-to-one (`<links type="1:1" />`).

8 Installation, Setup and System Management

8.1 Installation

The following gives an overview of the installation process. A continuously updated and more detailed description with step-by-step instructions of the installation procedure can be found on the systems Wiki page at `http://opus.lingfil.uu.se/letsmt-trac/wiki/RepositorySetup`.

8.1.1 System Requirements

Ubuntu 10.04 LTS is the recommended Linux distribution for the LetsMT! system as development and testing is done on that system and this installation description is also based on that version. LetsMT! should however work on most other equivalent Linux distributions with some manual adjustments especially during installation.

8.1.2 Installation of Required Packages

The LetsMT! system depends on a number of open source software packages that have to be installed and partly configured before the actual repository software can be installed. The easiest way to do that is via the aptitude tool. The main required packages are apache2, perl, subversion, openssl and gridengine. There are many more smaller packages but since the list still changes under development it is best looked up on the Wiki page `http://opus.lingfil.uu.se/letsmt-trac/wiki/RepositorySetup`.

The installation of the package can be done with the commands:

```
sudo apt-get update
sudo apt-get install <list of packages>
```

8.1.3 Download of Latest Version

The latest stable release of the repository software can be downloaded at `http://opus.lingfil.uu.se/letsmt-dev/releases` as a tar package. Releases with names ending on `rc` are release candidates and should only be used for testing purposes.

8.1.4 Installation and Configuration

After the download the package can be unpacked and installed with the following command:

```
tar -xzf LetsMT-0.32.tar.gz
cd LetsMT-0.32
sudo make install
```

A few questions have to be answered during the installation process, mainly about the servers DNS name and the port for the LetsMT! web services.

8.1.5 Test of the Installation

There are test scripts for all web API modules to test their full functionality. All test scripts together form a test suite that covers all important functions of the package that should be run after every installation or upgrade to ensure proper operation. A summary of the results of the test suite is displayed after each run and shows if any of the sub test scripts failed. In such a case the failed sub test scripts can be run individually to get more detailed information to which test step exactly failed and why.

The test suite can be run from folder `perllib/LetsMT/t` inside the LetsMT! release package with the following command. However, the actual configuration must be loaded manually before with the `source` command, so that the `LETSMTROOT` variable gets replaced with the actual value (normally `/var/local`).

```
source LETSMTROOT/conf.sh
prove -r
```

If the results show that for example the meta data test failed that test script can be rerun from the folder `perllib/LetsMT/t` with the command

```
perl Repository_API_MetaData.t
```

8.2 Configuration

The configuration file `config.sh` contains a list of variable that are important for the configuration of the system. All values are set during the installation procedure and are acquired in an interactive dialog from the user where required. Table 19 shows all variables with their default value and description that are of interest for the setup and maintenance.

Variable	Default	Description
LETSMTVERSION	3	Version of the LetsMT! system
LETSMTROOT	/usr/local	Location of LetsMT! modules and binaries
LETSMT_SVN_BINDIR	/usr/local/subversion/bin	Path to SVN binary
LETSMTHOST	<hostname>	Automatically set to your servers host name
LETSMTPORT	443	Port server is listing for HTTPS connections on
LETSMTUSER	www-data	Unix user the LetsMT! system acts as
LETSMTCONF	<LETSMTROOT>/etc/repository/conf.sh	Config file to load
LOG4PERLCONF	<LETSMTROOT>/etc/repository/log4perl.conf	Configuration file for logging
PERL5LIB	/usr/local/lib/perl5	Location of Perl libraries
UPLOADDIR	/var/tmp	Place where uploads are initially stored
LETSMTDISKROOT	var/lib/letsmt/www-data	Location of all SVN repositories and meta DB
DBHOST	127.0.0.1	URL/IP for DB
DBPORT	3306	Port for DB
DBUSER	letsmt_db_user	DB user name
DBPASS	none	Password for the DB
LETSMTLOG_DIR	/var/log/letsmt	Location for all log files
LETSMT_SSL_COUNTRY	SE	SSL country setting
LETSMT_SSL_CITY	Uppsala	SSL city setting
LETSMT_SSL_COMPANY	LetsMT	SSL company setting
LETSMT_SSL_USER	developers@localhost	SSL user setting
LETSMT_SSL_USERNAME	Developers LetsMT	SSL username setting
LETSMT_SSL_DIR	/etc/ssl	SSL directory setting
LETSMT_SSL_SUBDIR	letsmt2	SSL sub-directory setting
LETSMT_CERTROOT	/etc/apache2/ssl_certs	SSL setting
LETSMT_CACERT	/etc/ssl/letsmt2/ca.crt	SSL setting
LETSMT_CERTPATH	/etc/ssl/letsmt2/newcerts	SSL setting
LETSMT_REVOCATIONPATH	/etc/ssl/letsmt2/crl	SSL setting
LETSMT_USERCERT	/etc/ssl/letsmt2/user/certificates/developers@localhost.crt	Path to SSL certificate file
LETSMT_USERCERTPASS	letsmt	SSL certificate password
LETSMT_USERKEY	/etc/ssl/letsmt2/user/keys/developers@localhost.key	Path to the SSL user key file
LETSMT_REPOS_DIR_UPLOAD	”uploads”	Directory uploads have to go to to be imported
LETSMTPUBLICGROUP	”public”	Name of the public group

Table 19: Important variables of config.sh

A few of these variables and their properties will be discussed in the following as they are of special interest for the behavior of the system:

LETSMT_SVN_BINDIR - The system comes with its own version of subversion as it uses a release that is currently not included in most Linux distributions. The `LETSMT_SVN_BINDIR` variable is used to point to that binary file.

LETSMTUSER - This variable sets the effective Linux system user for the LetsMT! system. It should be the same as the Apache server runs as and all directories the system need to write to and read from should have their permissions set to allow read and write access for this user. Specifically these are the directories set with `LETSMTDISKROOT`, `LETSMTLOG` and `UPLOADDIR`.

LETSMTCONF - This variable points to the configuration file `conf.sh` which controls most of the LetsMT! system. After a change to this file the command `source conf.sh` should be run form the directory the file resides in to activate the changed configuration.

LOG4PERLCONF - This configuration file controls the logging of the LetsMT! system and specifically the logging level which can be set via the `log4perl.logger` variable in the configuration file. It can be set to `DEBUG`, `INFO`, `WARN`, `ERROR` or `FATAL` and sets the threshold from which severity on the system should log events. See also section 8.4

UPLOADDIR - This directory is used to to temporarily store files that get uploaded via the web server before they get committed to a SVN repository but also to checkout files and prepare zip archives before download. The user set via the `LETSMTUSER` variable need read and write access to this directory.

LETSMTDISKROOT - This variable points to the directory where the SVN repositories and the meta database reside. The user set via the `LETSMTUSER` variable need read and write access to this directory. If you want to change this variable to an other location you have to move the meta database and the SVN repositories manually to the new location. The system automatically creates a new empty meta database if it doesn't find one at the expected location.

LETSMTLOG - In this directory all logging information is stored. There are logs prefixed with user names for every system user to support testing and development but in normal operation there should only be logs prefixed with the user set via the `LETSMTUSER` variable.

8.3 Backup and Restore

Since all relevant data is stored in `LETSMTDISKROOT` specifically the SVN repositories and the meta database it is this directory that requires backup. In order to get a consistent backup the Apache server should be stopped to prevent any read/write access via the webservices. It should also be made sure that no job is running on the grid engine. After that a simple file copy command on the `LETSMTDISKROOT` directory to a save location produces a backup that can also be used to restore the system. Again, the web server has to be stopped before the backup can be copied back into an empty `LETSMTDISKROOT` directory.

8.4 Logging

Logging is done via the Perl module log4perl and all log files are written to the directory configured in `conf.sh` via the `LETSMTLOG_DIR` variable, default is `/var/log/letsmt/www-data` (see section 8.2). The logging configuration is done in the file `log4perl.conf` whose location is set in the `LOG4PERLCONF` variable. There are five logging levels (`DEBUG`, `INFO`, `WARN`, `ERROR`, `FATAL`) and a threshold can be defined from which severity on logging should be done. The logging configuration can be changed on a running system as the file gets reloaded every 60 seconds if a change is detected. For more information on the configuration file and its syntax see `http://search.cpan.org/dist/Log-Log4perl/lib/Log/Log4perl.pm.`

Logging form the Apache server are handled by the server itself but is also written to the same directory and is following the same naming conventions.

As there are different logging levels and system parts that generate logs, there are also different log files with corresponding names.
`LETSMTUSER_apache_error.log` contains all errors reported by Apache server, `LETSMTUSER_debug.log` contains logging information form all logging levels for the Perl modules, `LETSMTUSER_error.log` contains only only entries form the level WARN upwards.

A API commands with examples

A.1 Access

Requesting the access settings of all branches in a slot, e.g. `/access/slot_name_1` :

```
GET 'https://host:443/ws/access/slot
        _name_1?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <list path="slot_name_1">
    <entry kind="branch" path="slot_name_1/user_id_2">
      <group>group_id_2</group>
      <owner>user_id_3</owner>
    </entry>
    <entry kind="branch" path="slot_name_1/user_id_3">
      <group>group_id_2</group>
      <owner>user_id_2</owner>
    </entry>
  </list>
  <status code="0" location="/access/slot_name_1" oper
        ation="GET" type="ok"></status>
</letsmt-ws>
```

Requesting the access settings of a certain branch, e.g. `/access/slot_name_1/user_id_2`

```
GET 'https://host:443/ws/access/slot_name_1/user_id_2?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <list path="slot_name_1/user_id_2">
    <entry kind="branch" path="slot_name_1/user_id_2">
      <group>group_id_2</group>
      <owner>user_id_2</owner>
    </entry>
  </list>
  <status code="0" location="/access/slot_name_1/user_id_2" operation="GET" type="ok"></status>
</letsmt-ws>
```

Setting the group name of a branch to a new value, e.g. `/access/slot_name_2/user_id_2?uid=user_id_2&gid=group_3`

```
PUT 'https://host:443/ws/access/slot_name_2/user_id_2?uid=user_id_2&gid=group_3'
```

```
<letsmt-ws version="3">
  <status code="0" location="/access/slot_name_2/user_id_2" operation="PUT" type="ok">
    Set group to 'group_3'
  </status>
</letsmt-ws>
```

A.2 Admin

Show status information about the DB: `/admin?type=db_status`

```
GET 'https://host:443/ws/admin?type=db_status'
```

```
<letsmt-ws version="3">
  <list path="/var/lib/letsmt/www-data/metadata.tct">
    <entry>
      <additional flags>none</additional flags>
      <alignment>16</alignment>
      <bucket number>131071</bucket number>
      <database type>table</database type>
      <file size>532480</file size>
      <free block pool>1024</free block pool>
      <index number>0</index number>
      <inode number>62534558</inode number>
      <modified time>2011-05-23T11</modified time>
      <options>none</options>
      <path>/var/lib/letsmt/www-data/metadata.tct</path>
      <record number>0</record number>
      <unique ID seed>0</unique ID seed>
      <used bucket number>0</used bucket number>
    </entry>
  </list>
  <status code="0" location="/admin" operation="GET" type="ok"></status>
</letsmt-ws>
```

Return the full SVN path of a given resource, e.g. `/admin/slot_name_2/user_id_2/test_file_2.xml?type=svnpath`

```
GET 'https://host:443/ws/admin/slot_name_2/user_id_2/test_file_2.xml?type=svnpath'
```

```
<letsmt-ws version="3">
  <list path="file:///slot_name_2/user_id_2/test_file_2.xml">
    <entry kind="svn path">
      file:///var/lib/letsmt/www-data/slot_name_2/user_id_2/test_file_2.xml
    </entry>
  </list>
  <status code="0" location="/admin/slot_name_2/user_id_2/test_file_2.xml"
          operation="GET" type="ok"></status>
</letsmt-ws>
```

A.3 Group

List all groups: `/group`

```
GET 'https://host:443/ws/group'
```

```
<letsmt-ws version="3">
  <list path="/">
    <entry id="group_id_1" kind="group">
      <user>user_id_1</user>
    </entry>
    <entry id="group_id_2" kind="group">
      <user>user_id_2</user>
    </entry>
  </list>
  <status code="0" location="/group" operation="GET" type="ok"></status>
</letsmt-ws>
```

Add user to a group, e.g. `/group/group_id_1/user_2?uid=user_id_1`

```
PUT 'https://host:443/ws/group/group_id_1/user_2?uid=user_id_1'
```

```
<letsmt-ws version="3">
  <status code="0" location="/group/group_id_1/user_2" operation="PUT" type="ok">
    Added user 'user_2' to group 'group_id_1'
  </status>
</letsmt-ws>
```

Get the user(s) of a group, e.g. `/group/group_id_1`

```
GET 'https://host:443/ws/group/group_id_1'
```

```
<letsmt-ws version="3">
  <list path="/group_id_1">
    <entry id="group_id_1" kind="group">
      <user>user_id_1</user>
      <user>user_id_2</user>
    </entry>
  </list>
  <status code="0" location="/group/group_id_1" operation="GET" type="ok"></status>
</letsmt-ws>
```

Create a group with the effective user as first member, e.g. `/group/group_1?uid=user_1`

```
POST 'https://host:443/ws/group/group_1?uid=user_1'
```

```
<letsmt-ws version="3">
  <status code="0" location="/group/group_1" operation="POST" type="ok">
    Created group 'group_1' with user 'user_1'
  </status>
</letsmt-ws>
```

Delete a group, e.g. `/group/group_1?uid=user_1`

```
DELETE 'https://host:443/ws/group/group_1?uid=user_1'
```

```
<letsmt-ws version="3">
  <status code="0" location="/group/group_1" operation="DELETE" type="ok"></status>
</letsmt-ws>
```

A.4 MetaData

Searching for meta data by path, e.g. `/slot_name_1`

```
GET 'https://host:443/ws/metadata/slot_name_1?uid=user_id_1'
```

```
<letsmt-ws version="3">
  <list path="">
    <entry path="slot_name_1">
      <creator>user_id_1</creator>
      <resource-type>slot</resource-type>
    </entry>
  </list>
  <status code="0" location="/metadata/slot_name_1" operation="GET" type="ok">
    Found matching path ID. Listing all of its properties
  </status>
</letsmt-ws>
```

Searching for meta data by argument, without any path, only IDs as result, e.g. `status=imported`

```
GET 'https://host:443/ws/metadata?uid=user_id_2&status=imported'
```

```
<letsmt-ws version="3">
  <list path="">
    <entry path="slot_name_2/user_id_2/uploads/test_file_1.txt" />
    <entry path="slot_name_2/user_id_2/uploads/test_file_2.txt" />
  </list>
  <status code="0" location="/metadata" operation="GET" type="ok">Found 2 matching entries</status>
</letsmt-ws>
```

Searching for meta data by path and argument, only IDs as result, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&size`

```
GET 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&size'
```

```
<letsmt-ws version="3">
  <list path="">
    <entry path="slot_name_2/test_file_2.xml" />
  </list>
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="GET" type="ok">
     Found matching path ID. Listing only the requested properties
  </status>
</letsmt-ws>
```

Searching for meta data by argument, without any path, full result, e.g. `status=imported&action=list_all`

```
GET 'https://host:443/ws/metadata?uid=user_id_2&status=imported&action=list_all'
```

```
<letsmt-ws version="3">
  <list path="">
    <entry path="slot_name_2/user_id_2/uploads/test_file_1.txt">
      <status>imported</status>
      <uid>user_id_2</uid>
    </entry>
    <entry path="slot_name_2/user_id_2/uploads/test_file_2.txt">
      <status>imported</status>
      <uid>user_id_2</uid>
    </entry>
  </list>
  <status code="0" location="/metadata" operation="GET" type="ok">Found 2 matching entries</status>
</letsmt-ws>
```

Adding new or extending existing meta data entry, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar`

```
PUT 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar'
```

```
<letsmt-ws version="3">
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="PUT" type="ok">
    Created/Extended meta data entry
  </status>
</letsmt-ws>
```

Searching for meta data by path and argument, full result,
e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&size&action=list_all`

```
GET 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&size&action=list_all'
```

```
<letsmt-ws version="3">
  <list path="">
    <entry path="slot_name_2/test_file_2.xml">
      <size>5</size>
      <uid>user_id_2</uid>
    </entry>
  </list>
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="GET" type="ok">
    Found matching path ID. Listing only the requested properties
  </status>
</letsmt-ws>
```

Adding new or overwriting existing meta data entry, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar_2`

```
POST 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar_2'
```

```
<letsmt-ws version="3">
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="POST" type="ok">
    Created/Overwrote meta data entry
  </status>
</letsmt-ws>
```

Deleting a meta data key and all its values, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&foo`

```
DELETE 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&foo'
```

```
<letsmt-ws version="3">
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="DELETE" type="ok">
    Deleted meta data entry
  </status>
</letsmt-ws>
```

Deleting a single value from a meta data key, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar`

```
DELETE 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2&foo=bar'
```

```
<letsmt-ws version="3">
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="DELETE" type="ok">
    Deleted meta data entry
  </status>
</letsmt-ws>
```

Deleting all meta data for a path, e.g. `/slot_name_2/test_file_2.xml?uid=user_id_2`

```
DELETE 'https://host:443/ws/metadata/slot_name_2/test_file_2.xml?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <status code="0" location="/metadata/slot_name_2/test_file_2.xml" operation="DELETE" type="ok">
    Deleted meta data entry
  </status>
</letsmt-ws>
```

A.5 Storage

List all slots, `/storage?uid=bob`

```
GET 'https://host:443/ws/storage?uid=bob'
```

```
<letsmt-ws version="3">
  <list path="file:///">
    <entry kind="slot">
      <name>slot_name_1</name>
      <group>users</group>
      <owner>root</owner>
      <perm>drwrwrw</perm>
    </entry>
    <entry kind="slot">
      <name>slot_name_2</name>
      <group>users</group>
      <owner>root</owner>
      <perm>drwrwrw</perm>
    </entry>
</list>
  <status code="0" location="/storage" operation="GET" type="ok"></status>
</letsmt-ws>
```

List all branches of a slot, e.g. `/storage/slot_name_2?uid=user_id_2`

```
GET 'https://host:443/ws/storage/slot_name_2?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <list path="file:///slot_name_2">
    <entry kind="branch">
      <name>user_id_1</name>
      <group>group_id_2</group>
      <owner>user_id_1</owner>
      <perm>rwr---</perm>
    </entry>
    <entry kind="branch">
      <name>user_id_2</name>
      <group>group_id_2</group>
      <owner>user_id_2</owner>
      <perm>rwr---</perm>
    </entry>
  </list>
  <status code="0" location="/storage/slot_name_2" operation="GET" type="ok"></status>
</letsmt-ws>
```

Output the content of a file, e.g. `/storage/slot_name_2/user_id_2/uploads/test_file_2.txt?uid=user_id_2&action=cat&from=1&to=3`

```
GET ’https://diates.lingfil.uu.se:4445/ws/storage/slot_name_2/user_id_2/uploads/test_file_2.txt?
uid=user_id_2&action=cat&from=1&to=3’
```

```
<letsmt-ws version="3">
  <list path="">
    <entry>This is just a test file
Nothing more
Nothing less
</entry>
  </list>
  <status code="0" location="/storage/slot_name_2/user_id_2/uploads/test_file_2.txt" operation=
  "GET" type="ok"></status>
</letsmt-ws>
```

Upload files without import, e.g. `/storage/slot_name_2/user_id_2/uploads/test.txt?uid=user_id_2`

```
PUT 'https://host:443/ws/storage/slot_name_2/user_id_2/uploads/test.txt?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <status code="0" location="/storage/slot_name_2/user_id_2/uploads/test.txt"
          operation="PUT" type="ok">
    add ok /slot_name_2/user_id_2/uploads/test.txt
  </status>
</letsmt-ws>
```

Upload files with import, e.g. `/storage/slot_name_2/user_id_2/uploads/test.txt?uid=user_id_2&action=import`

```
PUT 'https://host:443/ws/storage/slot_name_2/user_id_2/uploads/test.txt?uid=user_id_2'
```

```
<letsmt-ws version="3">
  <status code="0" location="/storage/slot_name_2/user_id_2/uploads/test.txt"
          operation="PUT" type="ok">
    add ok /slot_name_2/user_id_2/uploads/test.txt
  </status>
</letsmt-ws>
```

References

[1] William A. Gale and Kenneth Ward Church. A program for aligning sentences in bilingual corpora. *Computational Linguistics*, 19(1), 1993.