OLD Documentation

Release 1.0a1

Joel Dunham

March 09, 2013

CONTENTS

Welcome to the documentation for the Online Linguistic Database version 1.0.

CHAPTER

ONE

GETTING STARTED

This section describes what the Online Linguistic Database does and how to get and install it.

1.1 About

The Online Linguistic Database (OLD) is software that facilitates collaborative storing, searching, processing and analyzing of linguistic fieldwork data.

Linguistic fieldwork stands to benefit significantly from inter-researcher collaboration and data-sharing. The OLD arose as a response to a lack of multi-user cross-platform tools for language documentation and analysis.

1.1.1 Purpose

The OLD seeks to facilitate achievement of the following objectives.

1. Language data can be shared easily between researchers.
2. Language data are intelligently structured (balancing an allowance for theoretical and methodological variation with capacity for easy retrieval and re-purposing.)
3. Language data are highly searchable.
4. Access to language data can be controlled via authentication and authorization.
5. Language data can be re-purposed. E.g., word list data recorded, transcribed and analyzed by a phonetician can be used by a syntactician, anthropologist, educator and/or community member.
6. Language data are digitized and available for digital processing, e.g., parsing, automated information extraction, corpus analysis, comparative cross-linguistic analysis.

1.1.2 What is it?

The OLD is a program for creating collaborative language documentation *web services* [1]. A web service is like a web site or web application, insofar as it runs on a web server and responds to HTTP requests. However, a web service differs from a traditional web application in that it expects to communicate with other programs and not, directly, with human users.

The benefit of this design strategy is that a single web service can form a useful component of a variety of different applications with different goals. For example, an OLD web service for language *L* could serve data to a mobile

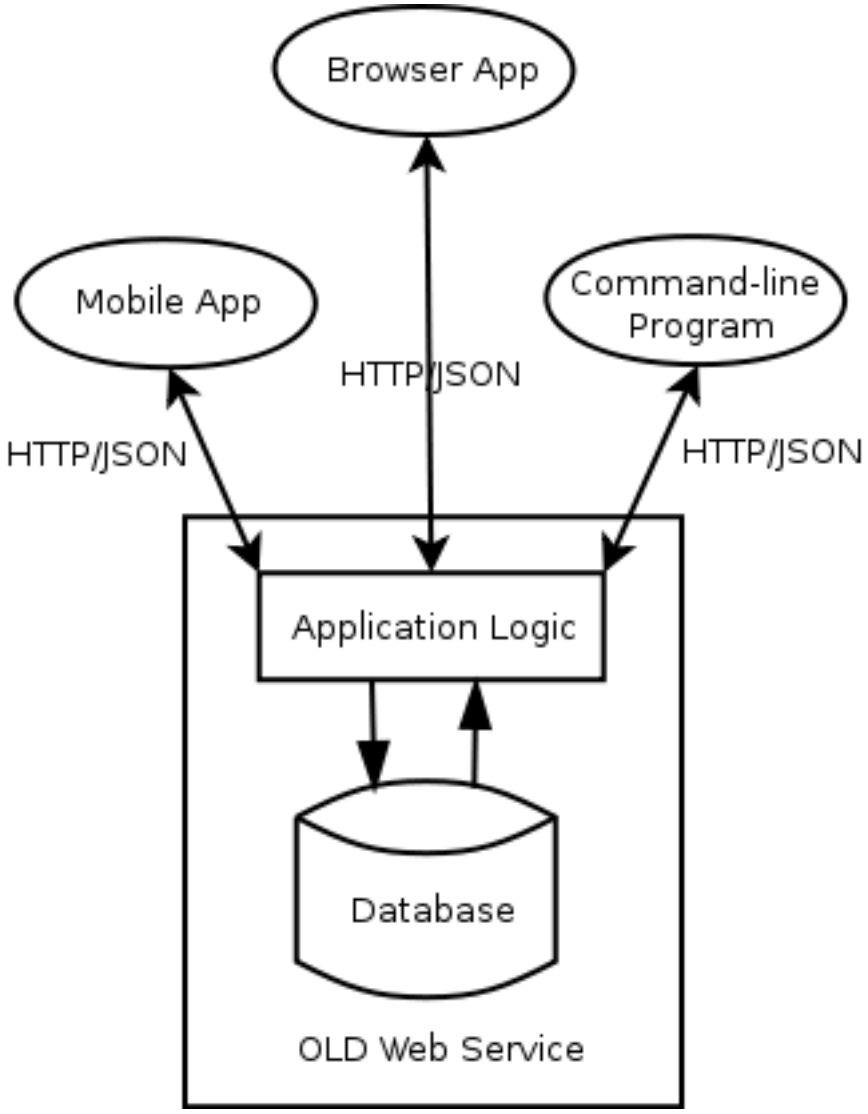
[1] Note that this is a break from previous versions of the OLD. In versions 0.1 through 0.2.7, the OLD was a traditional web application, i.e., it served HTML pages as user interface and expected user input as HTML form requests.

application that helps users to learn *L*. At the same time, researchers could be collaboratively entering, searching and processing data on the OLD web service via a desktop application and/or a browser-based one.

The OLD will be packaged with an in-browser user-friendly application. However, since these two applications will not be interdependent their documentation will be kept separate also.

The OLD is intended to be set up on a web server. However, it can also easily be installed on a personal computer for, say, developmental or experimental purposes. For detailed installation instructions see the *Installation & Configuration* section.

The OLD is, at its core, a database-driven application. It is essentially an interface to a relational database with a specific data structure, or schema. The schema was designed with the goals of linguistic fieldwork in mind. An OLD web service receives input in the form of HTTP requests with parameters encoded (usually) as JavaScript Object Notation (JSON). The application logic authenticates and authorizes the request and then, depending on the type of request, queries or updates the database and returns an HTTP response with JSON in the response body. This is illustrated in the diagram below.



1.1.3 Core features

1. User authentication and authorization.
2. Multi-user resource creation, retrieval, update and deletion (where a "resource" is something like a linguistic form or a syntactic category).
3. Input validation (e.g., ensuring that morpheme segmentation strings do not contain characters outside of a specified phonemic inventory and set of morpheme delimiters).
4. Application-wide settings, i.e., validation settings, specifications of inventories & orthographies, object and meta-language identification, etc.
5. Data processing (e.g., copying and reduction of image and audio files, generation of category strings based on the categories of component morphemes, phrase-morpheme auto-linking, etc.)
6. Resource search, i.e., open-ended, nested boolean search with substring, exact and regular expression matches against specified fields.
7. (Linguistic analysis: phonology & corpora specification, automatic morphological modeling and morphological parser creation, syntactic parser specification & generation.)

1.1.4 Technologies

The OLD is written in Python [2], using the Pylons web framework. It exposes a RESTful API based on the Atom Publishing Protocol, as implemented by the Routes URL routing component of Pylons. The relational database management system (RDBMS) may be MySQL or SQLite (others are, in principle, possible also). SQLAlchemy provides a Pythonic interface (ORM) to the RDBMS.

1.1.5 Who should read this manual?

This document will be of use to anyone wishing to understand the inner workings of the OLD.

It will be useful, in particular, to system administrators who want to know how to acquire, configure, install and serve an OLD web service.

It will also be useful to developers who would like to contribute to the code or create user-facing applications that interact with OLD web services. Developers will also benefit from reading the API documentation.

End users who wish to know more about the data structures of the OLD or its linguistic analysis and language processing components will also find this manual helpful. Typically, end users of an OLD-based system will interact with an OLD web service not directly but via a user interface-focused application. Such users may want to consult the documentation for the latter application before exploring the present document.

1.1.6 License

The OLD is open source software licensed under Apache 2.0.

1.2 Installation & Configuration

This section explains how to get, install and configure an OLD application. An overview of the process:

1. Download and install the OLD.

[2] The OLD works with Python 2.6 and 2.7 but not with Python <= 2.5. It has not been tested with Python 3.

2. Generate an OLD config file and edit it.
3. Run the setup command to create the database tables and directory structure.
4. Serve the application and test that it is working properly.

Note that these installation instructions assume a Unix-like system, e.g., Linux or Mac OS X. If you are using Windows [3], please refer to the Pylons or the virtualenv documentation for instructions on how to create and activate a Python virtual environment and install and download a Pylons application.

1.2.1 QuickStart

For the impatient, here is the quickest way to install, configure and serve an OLD application. Before blindly issuing the following commands, however, it is recommended that you read the detailed instructions in the following sections

```
virtualenv --no-site-packages env
source env/bin/activate
easy_install onlinelinguisticdatabase
mkdir xyzold
cd xyzold
paster make-config onlinelinguisticdatabase production.ini
paster setup-app production.ini
paster serve production.ini
```

Open a new terminal window and run the basic test script to ensure that the OLD application is being served and is operating correctly:

```
python _requests_tests.py
```

You should see `All requests tests passed.` as output. Congratulations.

1.2.2 Download

Pre-packaged eggs of stable OLD releases can be downloaded from the Python Package Index.

The easiest way to get and install the OLD is via the Python command-line program Easy Install. Before issuing the following command, read the *Create a virtual Python environment* and consider installing the OLD in a virtual environment. To download and install the OLD with Easy Install, run:

```
sudo easy_install onlinelinguisticdatabase
```

For developers, the full source code for the OLD can be found on GitHub. To clone the OLD repository, first install Git and then run:

```
git clone git://github.com/jrwdunham/old.git
```

See below for detailed instructions.

[3] The OLD has not been tested on Windows. Some alterations to the source may be required in order to get it running on a Windows OS. To be clear, this does *not* mean that users running a Windows OS will not be able to use a production OLD web application. A live OLD application is a web service and users with any operating system should be able to interact with it, assuming an internet connection is available. What this does mean is that the OLD, as is, may not run on a Windows *server*.

1.2.3 Install

Create a virtual Python environment

It is recommended that the OLD be installed in a virtual Python environment. A virtual environment is an isolated Python environment within which you can install the OLD and its dependencies without inadvertently rendering other programs unworkable by, say, upgrading *their* dependencies in incompatible ways. If you do not want to install the OLD and its dependencies in a virtual environment, skip this section.

Use virtualenv to create a virtual Python environment. First, follow the steps on the aforementioned web site to install virtualenv. If you already have `easy_install` or `pip` installed, you can just run one of the following commands at the terminal:

```
pip install virtualenv
easy_install virtualenv
```

Otherwise, you can download the `virtualenv` archive, decompress it, move into the directory and install it manually, i.e.,

```
cd virtualenv-X.X
python setup.py install
```

Once virtualenv is installed, create a virtual environment in a directory called `env` (or any other name) with the following command:

```
virtualenv --no-site-packages env
```

The virtual environment set up in `env` is packaged with a program called `easy_install` which, as its name suggests, makes it easy to install Python packages and their dependencies. We will use the virtual environment's version of `easy_install` to install the OLD and its dependencies into the virtual environment.

There are two ways to do this. The more explicit and verbose way is to specify the path to the executables in the virtual environment directory. That is, to run the virtual environment's `python`, `easy_install` or `pip` executables, you would run one of the following commands.

```
/path/to/env/bin/python
/path/to/env/bin/easy_install
/path/to/env/bin/pip
```

The easier way (on Posix systems) is to activate the Python virtual environment by running the `source` command with the path to the `activate` executable in your virtual environment as its first argument. That is, run:

```
source /path/to/env/bin/activate
```

If the above command was successful, you should see the name of your virtual environment directory in parentheses to the left of your command prompt, e.g., `(env)username@host:~$`. Now invoking `python`, `easy_install`, `paster`, `pip`, etc. will run the relevant executable in your virtual environment.

Install the OLD

The easiest way to install the OLD is via Easy Install, as in the command below. (Note that from this point on I am assuming that you have activated a virtual environment in one of the two ways described above or have elected not to use a virtual environment.)

```
easy_install onlinelinguisticdatabase
```

You can also use `pip` to install it:

```
pip install onlinelinguisticdatabase
```

Once the install has completed, you should see `Finished processing dependencies for onlinelinguisticdatabase`. (If you used `pip`, you will see something like `Successfully installed onlinelinguisticdatabase`.) This means that the OLD and all of its dependencies have been successfully installed.

If you have downloaded the OLD source code and need to install the dependencies, then move to the root directory of the source, i.e., the one containing the `setup.py` file, and run:

```
python setup.py develop
```

1.2.4 Configure

Generate the config file

Once the OLD is installed, it is necessary to configure it. This is done by generating a default config file and making any desired changes. When the OLD's setup script is run, several directories will be created in the same directory as the config file. Therefore, it is a good idea to create the config file in its own directory. I use the convention of naming production systems using the ISO 639-3 three-character id of the object language. To illustrate, I will use the fictitious language id *xyz* and will name the directory `xyzold`, the MySQL database `xyzold` and the MySQL user `xyzuser`. If following this convention, replace "xyz" with the Id of the language your OLD application will be documenting. To make a new directory called `xyzold` and change to it, issue the following commands.

```
mkdir xyzold
cd xyzold
```

The first step in configuring the OLD is creating a config file. To create a config file named `production.ini`, run:

```
paster make-config onlinelinguisticdatabase production.ini
```

By default, the OLD is set to serve at 127.0.0.1 on port 5000, the Pylons interactive debugger is turned off and the database (RDBMS) is set to SQLite (a database called `production.db` will be created in the current directory). These defaults are good for verifying that everything is working ok. On a production system you will need to change the `host` and `port` values in the config file as well as set the database to MySQL. If you want to get up and running with MySQL right away, see the *Set up MySQL/MySQLdb* section; otherwise, continue on to *Edit the config file*.

Developers will not need to generate a config file. The `test.ini` and `development.ini` config file should already be present in the root directory of the source. See the *Developers* section for details.

Set up MySQL/MySQLdb

The OLD can be configured to use either MySQL or SQLite as its relational database management system (RDBMS).

While SQLite is easy to install (both the SQLite library and the `pysqlite` language binding are built into the Python language), it is not recommended for multi-user concurrent production systems. Therefore, a production OLD setup should have MySQL installed. The following instructions assume that you have successfully installed the MySQL server on your system.

First login to MySQL as root:

```
mysql -u root -p<root_password>
```

Then create a database to store your OLD data:

```
mysql> create database xyzold default character set utf8;
```

Now create a MySQL user with sufficient access to the above-created database. In the first command, `xyzuser` is the username and `4R5gvC9x` is the password.

```
mysql> create user 'xyzuser'@'localhost' identified by '4R5gvC9x';
mysql> grant select, insert, update, delete, create, drop on xyzold.* to 'xyzuser'@'localhost';
mysql> quit;
```

Make sure that the above commands worked:

```
mysql -u xyzuser -p4R5gvC9x
mysql> use xyzold;
mysql> show tables;
```

Now MySQL is set up with a database called `xyzold` (with UTF-8 as its default character set) and a user `xyzuser` who has access to `xyzold`. The next step is to make sure that the python module `MySQLdb` is installed. Enter a Python prompt (using your virtual environment, if applicable) and check:

```
python
>>> import MySQLdb
```

If you see no output, then `MySQLdb` is installed. If you see `ImportError: No module named MySQLdb`, then you need to install `MySQLdb`.

Installing `MySQLdb` can be tricky. On some Linux distributions, it is necessary to first install `python-dev`. On distros with the Advanced Packaging Tool, you can run the following command.

```
apt-get install python-dev
```

Once `python-dev` is installed, run the following to install `MySQLdb` (remembering to activate the virtual environment, if necessary).

```
easy_install MySQL-python
```

Note that it is also possible to use `easy_install` to install `MySQLdb` at the same time as you install the OLD. Instead of running `easy_install onlinelinguisticdatabase` as above, run the following command:

```
easy_install onlinelinguisticdatabase[MySQL]
```

Edit the config file

The config file (whose creation was described in *Generate the config file*) is where an OLD app is configured. Open the config file (e.g., `production.ini`) and make any desired changes. While the config file is self-documenting, this section supplements that documentation.

(Note that once the OLD is downloaded and installed, it may be used to run several distinct OLD web services, e.g., for different languages. To do this, repeat the configuration steps with different settings. For example, to create two OLD web services, one for language *xyz* and one for language *abc*, create two directories, `xyzold` and `abcold`, generate a config file in each, and edit each config file appropriately, following these instructions.)

The host and port where the application will be served are configured here. The defaults of `127.0.0.1` (i.e., localhost) and `5000` are fine for initial setup and testing. During deployment and server configuration, the host will certainly need to be changed and the port probably also.

The `set debug = false` line should be left as is on a production setup. However, for initial testing it is a good idea to comment out this line with a hash mark (i.e., `#set debug = false`) so that errors can be debugged. When the line is commented out and an error occurs, Pylons will generate a detailed error report with a web interface that can be accessed by navigating to the link printed to the console (i.e., stderr).

The `sqlalchemy.url` parameter will need to be changed, depending on the relational database setup needed. If SQLite will be used, then the `sqlalchemy.url = sqlite:///production.db` line should remain uncommented. Change the database name, if desired; i.e., change `production.db` to, say, `mydb.sql`.

If MySQL will be used, then the first step is to comment out the SQLite line, and uncomment the *two* MySQL lines:

```
#sqlalchemy.url = sqlite:///production.db
sqlalchemy.url = mysql://username:password@localhost:3306/dbname
sqlalchemy.pool_recycle = 3600
```

Then, change the first MySQL line so that it contains the appropriate values for your MySQL setup. E.g., using the example setup from *Set up MySQL/MySQLdb* would involve changing it to the following:

```
sqlalchemy.url = mysql://xyzuser:4R5gvC9x@localhost:3306/xyzold
```

The only other values you may want to change are `password_reset_smtp_server`, `create_reduced_size_file_copies` and `preferred_lossy_audio_format`.

Uncomment the `password_reset_smtp_server = smtp.gmail.com` line if you want the system to send emails using a Gmail account specified in a separate `gmail.ini` config file.

Set `create_reduced_size_file_copies` to `0` if you do *not* want the system to create copies of images and .wav files with reduced sizes. Note that in order for the reduced-copies functionality to succeed with images and .wav files it is necessary to install the Python Imaging Library (PIL) and FFmpeg, respectively (see the *Soft dependencies* section below).

Finally, set the `preferred_lossy_audio_format` to `mp3` instead of `ogg` if you would like to create .mp3 copies of your users' .wav files instead of .ogg ones. (Note that a default installation of FFmpeg may not be able to convert .wav to .mp3 without installation of some additional libraries.)

1.2.5 Setup

Once the OLD has been installed and a config file has been created and edited, it is time to run the `setup` command. This will generate the tables in the database, insert some defaults (e.g., some users and useful tags) and create the requisite directory structure. To set up an OLD application, move to the directory containing the config file (e.g., `xyzold` containing `production.ini`) and run the `paster setup-app` command:

```
cd xyzold
paster setup-app production.ini
```

If successful, the output should be `Running setup_app() from onlinelinguisticdatabase.websetup`. By default, the OLD sends logs to `application.log` so if you run `cat application.log` you should see something like the following.

```
Environment loaded.
Retrieving ISO-639-3 languages data.
Creating a default administrator, contributor and viewer.
Tables created.
Creating default home and help pages.
Generating default application settings.
Creating some useful tags and categories.
Adding defaults.
OLD successfully set up.
```

If you now enter the database and poke around, you will see that the tables have been created and the defaults inserted.

```
mysql -u xyzuser -p4R5gvC9x
mysql> use xyzold;
```

```
mysql> show tables;
mysql> select username from user;
```

You should also see two new directories (`analysis` and `files`), the application log file `application.log` and Python script `_requests_tests.py`.

1.2.6 Serve

To begin serving an OLD application, use Paster's `serve` command:

```
paster serve production.ini
```

The output should be something like the following.

```
Starting server in PID 7938.
serving on http://127.0.0.1:5000
```

If you visit `http://127.0.0.1:5000` in a web browser, you should see `{"error": "The resource could not be found."}` displayed. If you visit `http://127.0.0.1:5000/forms` in a web browser, you should see `{"error": "Authentication is required to access this resource."}`. These error responses are to be expected: the first because no resource was specified in the request URL and the second because authentication is required before forms can be read. Congratulations, this means an OLD application has successfully been set up and is being served locally.

When `paster setup-app` is run, a Python script called `_requests_tests.py` is created in the current working directory. This script uses the Python Requests module to test that a live OLD application is working correctly. Assuming that you have run `paster serve` and an OLD application is being served locally on port 5000, running the following command will run the `_requests_tests` script:

```
python _requests_tests.py
```

If everything is working correctly, you should see `All requests tests passed.` (Note that if you have changed the config file, i.e., the host or port values, then you will need to change the values of the `host` and/or `port` variables in `_requests_tests.py` to match.)

1.2.7 Soft dependencies

In order to create smaller copies of image files and .wav files, the OLD uses the Python Imaging Library (PIL) and the FFmpeg command-line program. If you would like your OLD application to automatically create reduced-size images and lossy (i.e., .ogg or .mp3) copies of .wav files, then these programs should be downloaded and installed using the instructions on the above-linked pages. I provide brief instructions here.

PIL

To install PIL, download and decompress the source. Then change to the root folder and run `setup.py install` (remembering to use your `virtualenv` Python executable, if necessary):

```
cd Imaging-1.1.7
python setup.py install
```

The OLD accepts .jpg, .png and .gif image file uploads. If you want to test whether the PIL install can resize all of these formats, create a test file of each format and run something like the following. If successful, you will have created a smaller version of each image:

```
>>> import Image
>>> im = Image.open('large_image.jpg')
>>> im.thumbnail((500, 500), Image.ANTIALIAS)
>>> im.save('small_image.jpg')
```

FFmpeg

FFmpeg is a command-line tool that can convert .wav files to the lossy formats .ogg and .mp3. It can be somewhat tricky to install FFmpeg properly and some installs will not support .mp3 creation by default. For Debian 6.0 (Squeeze), I can recommend this tutorial.

Once `ffmpeg` is installed, you can check whether .wav-to-.ogg and .wav-to-.mp3 conversion is working by ensuring you have a file called `old_test.wav` in the current directory and issuing the following commands:

```
ffmpeg -i old_test.wav old_test.ogg
ffmpeg -i old_test.wav old_test.mp3
```

If successful, you will have created a .ogg and a .mp3 version of your .wav file.

1.2.8 Deploy

Deploying an OLD application means getting a domain name, serving the application on the world wide web and setting up some admin scripts. There are many possible ways to achieve this. In my production systems I have followed the approach of using Apache to proxy requests to Pylons as described in Chapter 21: Deployment of The Pylons Book and have had success with that. I review that approach here.

Assuming Apache 2, `mod_proxy` and `mod_proxy_http` are installed, you first enable the latter two:

```
sudo a2enmod proxy
sudo a2enmod proxy_http
```

Then you create a config file such as the one below in `/etc/apache2/sites-available/` or in the equivalent location for your platform. I used the config file below for an OLD application deployed for documenting the Okanagan language. The domain name is *okaold.org*. I saved the file as `/etc/apache2/sites-available/okaold.org` and created the error logs directory, i.e., `/home/old/log`. The only configuration necessary for the *OLD* config file (i.e., the `production.ini` file whose creation was detailed in *Generate the config file*) is to ensure that the `host` variable is set to `localhost` and the `port` variable is set to `8081`.

```
NameVirtualHost *
# OKA - Okanagan
<VirtualHost *>
    ServerName okaold.org
    ServerAlias www.okaold.org

    # Logfiles
    ErrorLog /home/old/log/error.log
    CustomLog /home/old/log/access.log combined

    # Proxy
    ProxyPreserveHost On
    ProxyPass / http://localhost:8081/ retry=5
    ProxyPassReverse / http://localhost:8081/
    <Proxy *>
        Order deny,allow
        Allow from all
```

```
    </Proxy>
</VirtualHost>
```

Now you can start serving the OLD application with Paster. In order to keep the server running after you exit the shell, you must invoke `paster serve` in daemon mode, as follows:

```
paster serve --daemon production.ini start
```

Now disable the default Apache configuration, enable the virtual host config file just created (in this case `okaold.org`) and restart Apache:

```
sudo a2dissite default
sudo a2ensite okaold.org
sudo /etc/init.d/apache2 restart
```

You might also want the `paster serve` script to log error messages, which you can do by specifying a file to log to using the `--log-file` option. You can also use the `--pid-file` option to store the process ID of the running server in a file so that other tools know which server is running:

```
paster serve --daemon --pid-file=/home/old/okaold.pid --log-file=/home/old/log/paster-okaold.log proc
```

As well as specifying `start`, you can use a similar command with `stop` or `restart` to stop or restart the running daemon, respectively.

The Pylons Book also explains how to Create init scripts and how to use `crontab` to restart a paster server that is serving an OLD/Pylons application (should that) ever be necessary. See the referenced sections for details.

You may also wish to write admin scripts to monitor an OLD application to ensure that it is functioning properly and to email you if not. I may include a guide for doing that at some future data.

Finally, it is a good idea to make regular backups of the database and the `files` and `analysis` directories of your OLD application. In my production systems I have used MySQL database replication to create a mirror of my production database on a second server in a different location. I then use the standard Unix utility `rsync` to create live copies of the `files` and `analysis` directories on that same second server. A Python script is run periodically on the second server to perform a `mysqldump` of the relevant databases. I will further document my backup setup at a later date.

1.2.9 Developers

This section provides an overview of the OLD for developers. It covers (1) how to download the source and install the dependencies, (2) the structure of the source, (3) how to write and compile the documentation to HTML and PDF, (4) the creation of Python version-specific virtual environments and (5) the building of OLD releases as eggs or archives.

For detailed documentation on developing a Pylons application, consult the excellent documentation for the Pylons framework, i.e., The Pylons Book.

Download & depencency installation

This subsection details how to get the OLD source and install its dependencies. To download the most up-to-date source code, make sure you have Git installed and run:

```
git clone git://github.com/jrwdunham/old.git
```

To install the dependencies, move to the newly created `old` directory and run:

```
python setup.py develop
```

Directory structure

The `onlinelinguisticdatabase` directory contains all of the files and directories that will be packaged into the final distribution. Its subdirectories are `config`, `controllers`, `lib`, `model`, `public`, and `tests`. This section gives an overview of the contents of these directories and the `websetup.py` file.

The `websetup.py` file controls how an OLD application is set up. That is, when `paster setup-app config_file.ini` is run (or when `nosetests` is run), the contents of `websetup.py` determine what database tables are created, what defaults are entered into them and what directories are created.

The `config` directory houses the `deployment.ini_tmpl` and `routing.py` files. The former is the template used to generate the config file when something like `paster make-config production.ini` is run. The `routing.py` module is where the mappings from URL paths to OLD controller actions are specified. When a new controller is created or the interface to an existing controller needs to be changed, the `routing.py` file must be edited.

The `controllers` directory holds a module for each OLD controller. For example, the `controllers/forms.py` module defines a `FormsController` class; the methods of this class (the controller's *actions*) return values which determine the content of particular responses. The `index` method (action) of the `FormsController` class, for example, returns a list of all form models in the database; since `config/routing.py` maps `GET /forms` to `FormsController.index`, it is this list of forms that is returned when `GET /forms` is requested.

The `lib` directory holds modules that define functionality used by multiple controllers. The `utils.py` module defines a large number of widely-used functions, classes and data structures; these are made available in controllers under the `h` namespace, e.g., the value of `h.markupLanguages` is the list of valid markup language string values, as defined in `utils.py`. The `auth.py` script holds the decorators that control authentication and authorization. The `schemata.py` module contains the validators that are applied against user input. The other modules in the `lib` directory are mentioned in this document where appropriate; consult the docstrings for more information.

The `model` directory contains a module for each SQLAlchemy model used by the OLD. For example, `model/file.py` houses the `File` class which defines the attributes of the file model and their implementation as columns and relations in a relational database. The `model/model.py` is special; it defines the `Model` class from which all of the other models inherit a number of methods. Note that in order to make a model available in the `onlinelinguisticdatabase.model` namespace, it must be imported in `model/__init__.py`.

The `public` directory may contain static files, HTML, CSS and JavaScript. Since the client-side OLD application has not yet been implemented, the `public` directory contains, at present, only the `iso_639_3_languages_data` which stores the tab-delimited files containing the ISO-639-3 dataset.

The `tests` directory contains all of the test modules. When the `nosetests` command is run, it is the modules here that define the tests. For example, `tests/functional/test_forms.py` defines a `TestFormsController` class whose methods test the various actions (or functionalities) of the forms controller. For example, the `test_create` method of the `TestFormsController` class simulates `POST /forms` requests and confirms that the system behaves as expected. When testing new funcionality, new tests should be defined in `tests/functional` or existing tests should be supplemented. Note the `_toggle_tests.py` script which does not define tests but provides an easy way to turn large numbers of them on or off. For example, `./onlinelinguisticdatabase/tests/functional/_toggle_tests.py on` will turn all tests on and `./onlinelinguisticdatabase/tests/functional/_toggle_tests.py off` will turn them all off. See its docstrings for further usage instructions. Finally, the `tests` directory also contains the `_requests_tests.py` script which defines some simple tests (using the Requests module) which (as described in the *Serve* section) can be run on a live OLD application to ensure that it is working correctly.

The `websetup.py` module defines the `setup_app` function that is called when the OLD is set up, i.e., when `paster setup-app config_file.ini` is issued. The behaviour of the setup process is determined by the name of the config file. If `test.ini` is the config file (as is the case when `nosetests` is run), then test-specific setup will be performed, i.e., all database tables will be dropped and then re-created. Otherwise, only the tables that do not already exist will be created.

Documentation

This section reviews the OLD documentation creation process. The OLD documentation (i.e., this document) is written using Sphinx and the reStructuredText lightweight markup language. In order to edit and build the documentation, Sphinx must be installed:

```
easy_install sphinx
```

The reStructuredText source files for the OLD documentation are the `.rst`-suffixed files in the `docs` directory. The Sphinx documentation has a good overview of the reStructuredText syntax. Once the source files have been edited, build the documentation HTML (in `docs/_build/html`) by moving to the `docs` directory and running:

```
sphinx-build -b html . ./_build/html
```

To generate a LaTeX version of the documentation in `docs/_build/latex`, run (from the `docs` directory):

```
sphinx-build -b latex . ./_build/latex
```

If `pdflatex` is installed [4], generate a PDF of the documentation by moving to `docs/_build/latex` and running:

```
pdflatex -interaction=nonstopmode OLD.tex
```

Virtualenv & Python distros

In order to test whether the OLD works on different Python versions or to build distributions for those versions, it is necessary to create virtual environments for each such Python distribution.

The pythonbrew utility facilitates the building and installation of different Pythons in a user's home directory. Install `pythonbrew` using the instructions on its web site.

Now run `pythonbrew install` to install the desired Pythons. For example, to install Python 2.4.6, 2.5.6 and 2.7.3, run:

```
pythonbrew install 2.4.6
pythonbrew install 2.5.6
pythonbrew install 2.7.3
```

Once complete, new Python executables should be installed in `~/.pythonbrew/pythons/Python-2.4.6`, `~/.pythonbrew/pythons/Python-2.5.6`, etc. For example, to launch the Python 2.5.6 interactive console, run:

```
~/.pythonbrew/pythons/Python-2.5.6/bin/python
```

To create a virtual environment using one of these Pythons, run `virtualenv` with the `-p` option followed by the path to the desired Python executable. It is also a good idea to choose a name for the virtual environment that makes it easy to tell what version of Python it uses. For example:

```
virtualenv -p ~/.pythonbrew/pythons/Python-2.5.6/bin/python env-2.5.6
```

Make sure that the new virtual environment has the correct python:

```
~/env-2.5.6/bin/python --version
```

Note that the OLD works with Python 2.6 and 2.7 but not with 2.4 or 2.5. It has not been tested with Python 3.

[4] See this page for an overview of how to use the TeX command-line utilities.

Releases

This section explains how to build stable OLD releases and how to upload them to PyPI.

To build an egg or a source distribution of a stable release, run the following two commands, respectively:

```
python setup.py bdist_egg
python setup.py sdist
```

Each of these commands will create a new archive in the `dist` directory.

In order to build an OLD egg distribution and upload it to PyPI in one command, run the following command. (Note that you will need the OLD's PyPI password in order to be permitted to do this.)

```
python setup.py bdist_egg register upload
```

To create and upload the source distribution to PyPI (so that, e.g., Pip can be used to install the OLD), run:

```
python setup.py sdist register upload
```

CHAPTER

TWO

ARCHITECTURE

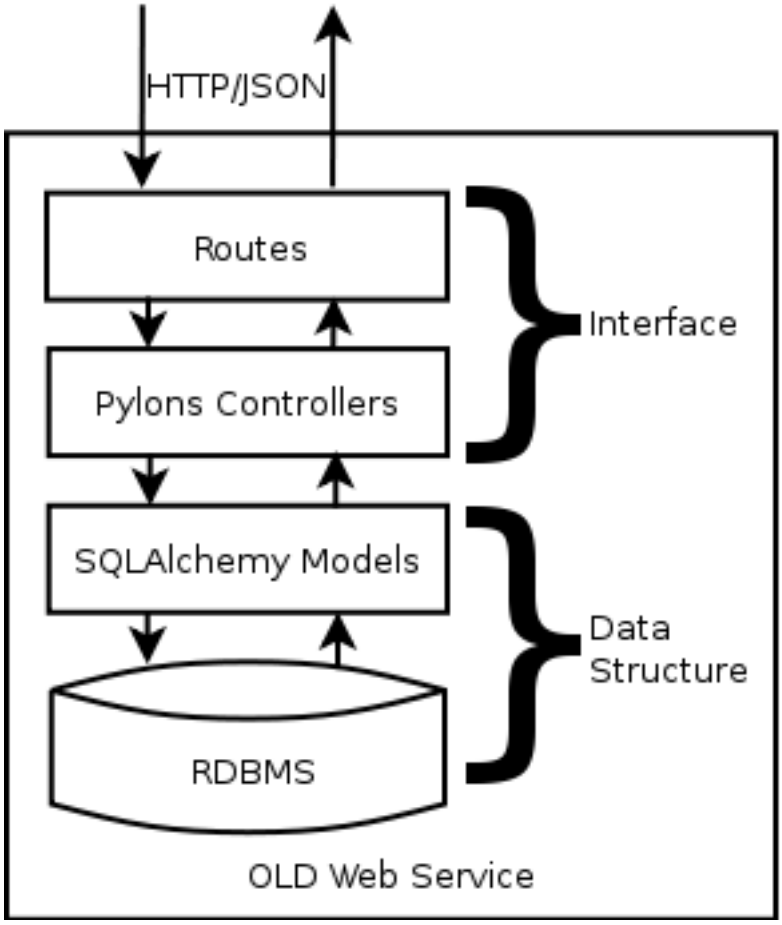
The architecture section provides details about the design of the OLD.

2.1 Introduction

An OLD web service consists of a data structure for storing the artifacts of linguistic fieldwork and analysis and a read-write interface to that data structure.

A major design principle of the OLD is that as much work as possible should be delegated to the user-facing applications so that the OLD web service can focus on providing secure and responsive multi-user concurrent access to a central data structure. In some cases, technological restrictions currently inherent to particular platforms (e.g., the inability of browser-based JavaScript applications to call external programs) have required server-side implementation of features that might otherwise be implemented client-side (e.g., morphological parsing, PDF creation using TeX).

The diagram below illustrates the core components of an OLD application.



When an OLD web application receives HTTP requests, the Routes component decides which Pylons controller will handle the request. This decision is based on the HTTP method of the request and the URL. Routes and the controllers conspire to create a RESTful interface to the data structure *qua* a set of resources. That is, a POST request to `www.xyz-old.org/forms` will be interpreted as a request to create a new form resource while the same URL with a GET method will be interpreted as a request to read (i.e., retrieve) all of the form resources. The first request will be routed to the `create` action (i.e., method) of the `forms` controller (i.e., class) while the second will be routed to the `index` action of that same controller. The authentication, authorization, input validation, data processing, linguistic analysis and database updates and queries are all handled by the controllers.

As illustrated in the diagram, the Routes and Controllers components can be conceptually grouped together as the *interface* of an OLD web service. The *Interface* section details this interface.

SQLAlchemy provides an abstraction over the tables and relations of the underlying database. Tables, their columns and the relations between them (i.e., the schema) are declared using Python data structures called *models* and interaction with the database is accomplished entirely via these. This not only simplifies interaction with the database (from the Python programmer's point of view) but also makes it easier to use different RDBMSs (e.g., SQLite, MySQL) with minimal changes to the application logic.

As illustrated in the diagram, the Models and RDBMS components can be conceptually grouped together as the *data structure* of an OLD web service. The *Data Structure* section describes and argues for the utility of the data structure of the OLD.

2.2 Interface

This section details the RESTful interface to the OLD data structure as well as resource search, authentication and authorization, input validation and notable data processing functionality. That is, it explains what kind of effect one can expect from requesting a particular URL (with a particular HTTP method and a particular JSON payload) of an OLD web service.

2.2.1 RESTful API

The OLD exposes a RESTful interface to its data structure. In the context of the OLD, the term *RESTful* [1] refers to the fact URLs are used consistently to refer to OLD resources and that HTTP methods dictate the action to be performed on the resource. For example, URLs of the form `/forms` and `/forms/id` are always routed to the forms controller which provides the interface for the form resources. If the HTTP method is GET and the URL is `/forms`, the system will *return* all form resources; the same URL with a POST method will cause the system to *create* a new form resource (using JSON data passed in the request body). The URL `/forms/id` with a PUT method will result in an *update* to the form resource with `id=id` while a DELETE method on the same URL will cause that resource to be *deleted*.

This pattern is detailed in the following table.

HTTP Method	URL	Effect	Parameters
GET	/forms	Read all forms	optional GET params
GET	/forms/id	Read form with id=id	
GET	/forms/new	Get data for creating a new form	optional GET params
GET	/forms/id/edit	Get data for editing form with id=id	optional GET params
DELETE	/forms/id	Delete form with id=id	
POST	/forms	Create a new form	JSON object
PUT	/forms/id	Update form with id=id	JSON object

The benefit of this consistent interface is that, once you know what resources the OLD exposes, it is clear how to create new ones, retrieve all or one in particular, update one or delete one. The resources of the OLD are listed in the table below.

Resource (URL)	SEARCH-able	Read-only	Additional actions
applicationsettings			
collections	Yes		Yes
collectionbackups	Yes	Yes	
elicitationmethods			
files	Yes		Yes
forms	Yes		Yes
formbackups	Yes	Yes	
formsearchs	Yes		
languages	Yes	Yes	
orthographies			
pages			
phonologies			
rememberedforms*	Yes		
sources	Yes		
speakers			
syntacticcategories			
tags			
users			

[1] See this StackOverflow page for a discussion on what exactly REST means and read Fielding's thesis for the source of the term.

As indicated by the "SEARCH-able" column in the above table, some OLD resources can be searched using a non-standard [2] SEARCH method with the relevant URL. The table below uses the files resources to illustrate the search interface. The details of the search feature (e.g., the format of JSON search parameters) are laid out in the *Search* section.

Note: `POST /resources/search` is a synonym for `SEARCH /resources`; this is to allow for search requests from clients that do not allow specification of non-standard HTTP methods.

HTTP Method	URL	Effect	Parameters
SEARCH	/files	Search files	JSON object
POST	/files/search	Search files	JSON object
GET	/files/new_search	Get data for searching files	

Requests to `GET /resources/new_search` return a JSON object which summarizes the data structure of the relevant resource, thus facilitating query construction.

For the read-only resources (cf. the third column in the resources table), the only standard requests that are valid are `GET /resources` and `GET /resources/id`. Since these read-only resources also happen to be searchable, the search-related requests of the table above are valid for them as well.

The core OLD resources (i.e., forms, files and collections) deviate from the RESTful standard in having additional valid URLs associated. For example, the forms resource has a `remember` action such that `POST /forms/remember` will result in the system associating the forms referenced in the request body to the user making the request (i.e., the user remembers those forms). Similarly, the files resource has a `serve` action such that `GET /files/serve/id` will return the file data for the file with `id=id`. These additional actions are described in the subsections for the relevant resources/controllers below.

Aside from those described above, the only additional valid URL/method combinations of an OLD web service have to do with authentication and the `login` controller. These are detailed in the *Authentication & authorization* section.

All other requests to an OLD web service will result in a response with a sensible HTTP error code and a JSON message in the response body that gives further information on the error.

GET /resources

Requests of the form `GET /resources`, e.g., `GET /forms`, return all resources of the type specified in the URL. These requests are routed to the `index` action of the controller for the resource.

The order of the returned resources may be specified via "orderBy"-prefixed parameters in the URL query string. For example, a request such as `GET /forms?orderByModel=Form&orderByAttribute=id&orderByDirection=desc` will return all form resources sorted by id in descending order. These ordering parameters are processed in exactly the same way as those passed as an array during resource search requests (see *Ordering results*).

It is also possible to request that the resources returned be paginated. This is accomplished by passing "page" and "itemsPerPage" parameters in the URL query string. For example, `GET /files?page=3&itemsPerPage=50` will return a JSON representation of files 101 through 150. Of course, ordering and pagination parameters may both be supplied in a single request.

GET /resources/id

Requests of the form `GET /resources/id`, e.g., `GET /collections/43`, return a JSON object representation of the resource with the specified id. These requests are routed to the `show` action of the controller for the resource.

[2] The WebDAV standard includes a SEARCH method so this is not entirely without precedent.

GET /resources/new

Requests of the form `GET /resources/new`, e.g., `GET /forms/new`, return a JSON object containing all of the data necessary to create new resources of the specified type. These requests are routed to the `new` action of the controller for the relevant resource. For example, when creating a new form resource, it is helpful to know the set of valid grammaticality values, elicitation method names, users, sources, etc. of the system. Therefore, a request to `GET /forms/new` will return a JSON object of the form listed below, where the values of the attributes are arrays containing the relevant data.

```
{
    "grammaticalities": [ ... ],
    "elicitationMethods": [ ... ],
    "tags": [ ... ],
    "syntacticCategories": [ ... ],
    "speakers": [ ... ],
    "users": [ ... ],
    "sources": [ ... ]
}
```

This is really just a convenience that saves the trouble of making multiple requests (e.g., to `GET /tags`, `GET /sources`, etc.)

Parameters in the query string can be used to alter the content of the response so that only certain datasets are returned. If the URL query string is not empty, then only the attributes of the response object that have non-empty parameters in the query string will be returned. For example, the request `GET /forms/new?sources=y&tags=y` will result in a response object of the same form as above except that only the `sources` and `tags` attributes will have non-empty arrays for values.

If the value of a parameter in the URL query string is a valid ISO 8601 datetime string of the form `YYYY-MM-DDTHH:MM:SS`, then the value of the corresponding attribute in the response object will be non-empty only so long as the input datetime does *not* match the most recent `datetimeModified` value of the specified resources. This permits the requesting of only novel data. For example the request `GET /forms/new?sources=2013-02-22T23:28:43` will return nothing but source resources and even these only if there are such that have been updated or created more recently than 2013-02-22T23:28:43.

Some resources have very simple data structures (e.g., tags) and, therefore, requests of the form `GET /resources/new` on such resources will return an empty JSON object.

GET /resources/id/edit

Requests of the form `GET /resources/id/edit` return the resource with the specified id as well as all data required to update that resource. These requests are routed to the `edit` action of the relevant controller. Such requests can be thought of as a combination of `GET /resources/id` and `GET /resources/new`. The JSON object in the response body is of the form

```
{"resourceName": {...}, "data": {...}}
```

where the value of the `resourceName` attribute is the same object as that returned by `GET /resources/id` and the value of the `data` attribute is the same as that returned by `GET /resources/new`. Parameters supplied in the URL query string have the same effect as those supplied to `GET /resources/new` requests (cf. *GET /resources/new*).

DELETE /resources/id

Requests of the form `DELETE /resources/id` result in the resource with the specified id being deleted from the database. Such requests are routed to the `delete` action of the relevant controller. The form and collection

resources are special in that they are first saved to a backup table before being deleted; thus these types of resources can be restored after deletion. The response body of a successful deletion request is a JSON object representation of the content of the resource. As mentioned above, only administrators and their enterers may delete form, file and collection resources.

POST /resources

Requests of the form `POST /resources` result in the creation of a resource of the specified type using the data supplied as a JSON object in the request body. These requests are routed to the `create` action of the relevant controller. The input data are first validated (as detailed in *Input validation*). If successful, a JSON object representation of the newly created resource is returned.

Note: All resources receive, upon successful POST and PUT requests, a value for a `datetimeModified` attribute which is a Coordinated Universal Time (UTC) timestamp. For creation requests on form, file and collection resources, the user who made the request is recorded in the `enterer` attribute of the resource.

PUT /resources/id

Requests of the form `PUT /resources/id` result in the updating of the resource of the specified type with the specified id. The data used to update the resource are supplied as a JSON object in the request body. These requests are routed to the `update` action of the relevant controller. As with the POST requests described above, the input data are validated before the update can occur. If successful, a JSON object representation of the newly updated resource is returned. Upon successful update, the previous versions of form and collection resources are saved to special backup tables of the database (i.e., `formbackup` and `collectionbackup`.)

JSON

As a general rule, the OLD communicates via JSON. JSON is a widely-used standard for converting certain data types and (nested) data structures to and from strings. Strings, numbers, arrays (lists) and associative arrays (dictionaries) can all be serialized to a JSON string. For example, a Python dictionary, i.e., a set of key/value pairs such as `{'transcription': 'dog', 'translations': [{'transcription': 'chien'}]}` when converted to JSON would be `'{"transcription": "dog", "translations": [{"transcription": "chien"}]}'`. In most cases, when an OLD web service requires user input, that input is expected to be JSON in the request body [3].

2.2.2 Search

The OLD provides a powerful search interface to a subset of its resources: collections, collectionbackups, files, forms, formbackups, formsearches, languages, rememberedforms and sources. This interface allows for an unlimited number of filter expressions conjoined via boolean operators into a hierarchical structure of unbounded depth where each filter expression references a resource attribute, a relation and a pattern.

In terms of implementation, search expressions are JSON objects that are mapped to SQLAlchemy query objects which produce SQL queries. In relational database-speak, the OLD search interface permits multi-table queries while taking care of the joins and subqueries automatically. The `SQLAQueryBuilder` class in `lib/SQLAQueryBuilder.py` handles the conversion from JSON search expression objects [4] to SQLAlchemy query objects.

[3] In contrast to POST, PUT and DELETE requests, HTTP GET requests are not, canonically, supposed to possess contentful request bodies; therefore, when optional parameters are permissible on such requests, the OLD will expect GET parameters in the URL string.

[4] Actually, the search actions of the relevant controllers convert the JSON string to a Python dictionary using the `loads` function of the `simplejson` module.

Valid search requests (e.g., `SEARCH /forms`) must contain in the request body a JSON object representing the query. The query object has a 'query' attribute whose value is another object which has a mandatory 'filter' attribute and an optional 'orderBy' attribute. The values of `request.body.query.filter` and `request.body.query.orderBy` are both arrays, the former representing the hierarchy of filter expressions conjoined by boolean operators and the latter representing a simple SQL `ORDER BY` clause:

```
{
    "query": {
        "filter": [ ... ],
        "orderBy": [ ... ]
    }
}
```

Filter expression syntax

OLD query filters are sets of simple filter expressions configured into a hierarchical structure using negation, conjunction and disjunction. Their syntax is simple and can be described via the following context-free grammar.

```
filterExpression         ::=   simpleFilterExpression | complexFilterExpression
simpleFilterExpression   ::=   "[" modelName "," attributeName "," relationName "," pat
                               "[" modelName "," attributeName "," attributeModelAttrib
complexFilterExpression  ::=   "[", "not" "," filterExpression "]" |
                               "[", "and" "," "[" filterExpression ("," filterExpressio
                               "[", "or" "," "[" filterExpression ("," filterExpression
```

That is, a `filterExpression` is either (1) a `simpleFilterExpression` or (2) an array whose first element is the string "not" and whose second element is another `filterExpression` or (3) an array whose first element is one of the strings "and" or "or" and whose second element is an array of one or more filter expressions.

Simple filter expressions

In plain English, a simple filter expression is something like "the transcription contains the character 'a'". A `simpleFilterExpression` is an array with four or five elements. If four, then the first is the name of an OLD model, the second the name of a valid attribute of that model, the third a relation and the fourth a pattern or value. Consider the simple filter expression below (where the forms resources are being searched, i.e., `SEARCH /forms`).

```
["Form", "transcription", "like", "%a%"]
```

This expression is mapped to the SQLAlchemy `query` object:

```
query(model.Form).filter(model.Form.transcription.like(u'%a%'))
```

which generates the SQL that follows.

```
SELECT * FROM form WHERE transcription LIKE '%a%';
```

A request to `SEARCH /forms` with this `simpleFilterExpression` in the request body would return all form resources whose transcription attribute contains the character "a".

When a simple filter expression has five elements, the second is assumed to be the name of a relational attribute, i.e., an attribute that references another model, while the third is an attribute of the referenced model. For example, the `Form` model has an `enterer` attribute whose value is a `User` model and a `User` model has a `firstName` attribute. Therefore, to find all form resources with enterers whose first name begins with "J" or "S", we construct the simple filter expression

```
["Form", "enterer", "firstName", "regex", "^[JS]"]
```

which maps to the SQLAlchemy query object:

```
query(model.Form).filter(model.Form.enterer.has(User.firstName.op('regexp')(u'^[JS]')))
```

The two following simple filter expressions return all forms lacking enterers and all forms having them, respectively.

```
["Form", "enterer", "=", null]
["Form", "enterer", "!=", null]
```

Some relational attributes of OLD models reference *collections*, i.e., lists of zero or more models of a given type. For example, OLD forms can be associated to one or more files, i.e., the `Form` model has a `files` attribute whose value is a collection of `File` objects. Since `File` objects have `id` attributes, we can use the filter expression below to retrieve all forms associated to files with one of the following ids: 1, 2, 33, 5.

```
["Form", "files", "id", "in", [1, 2, 33, 5]]
```

The four-element filter expression below returns the same result set as the five-element one one above. This is because the OLD knows that the `Form` model is being queried and that the only relation between the `Form` and `File` models is captured by the `files` attribute of the `Form` model. [5]

```
["File", "id", "in", [1, 2, 33, 5]]
```

The two following simple filter expressions return all forms lacking files and all forms having one or more, respectively.

```
["Form", "files", "=", null]
["Form", "files", "!=", null]
```

Complex filter expressions

Complex filter expressions are built from simple filter expressions using "not", "and" and "or".

The following complex filter expression uses "not" to return all form resources that do not have "a" in their transcriptions.

```
["not", ["Form", "transcription", "like", "%a%"]]
```

Conjoined and disjoined filter expressions are exemplified below.

```
['and', [['Form', 'transcription', 'like', '%a%'],
         ['Form', 'elicitor', 'id', '=', 13]]]
['or', [['Form', 'transcription', 'like', '%a%'],
        ['Form', 'dateElicited', '<', '2012-01-01']]]
```

Finally, an example of a complex filter expression involving multiple levels of embedding.

```
['and', [['Translation', 'transcription', 'like', '%1%'],
         ['not', ['Form', 'morphemeBreak', 'regex', '[28][5-7]']],
         ['or', [['Form', 'datetimeModified', '<', '2012-03-01T00:00:00'],
                 ['Form', 'datetimeModified', '>', '2012-01-01T00:00:00']]]]]
```

[5] Note that while the results returned will be the same, the SQLAlchemy query object constructed and the SQL issued to the database will be distinct. That is, the filter expression `["Form", "files", "id", "in", [1, 2, 33, 5]]` maps to the SQLAlchemy query `query(model.Form).filter(model.Form.files.any(model.File.id.in_([1, 2, 33, 5])))` while `["File", "id", "in", [1, 2, 33, 5]]` maps to `fileAlias = aliased(File)` and `Session.query(Form).filter(fileAlias.id.in_([1, 2, 33, 5])).outerjoin(fileAlias, Form.files)`.

Filter relations

OLD search requests permit the relations listed below.

- equality ("=" or "__eq__")
- inequality ("!=" or "__ne__")
- like ("like" [6])
- regular expression ("regex" or "regexp")
- less than ("<" or "__lt__")
- less than or equal to ("<=" or "__le__")
- greater than (">" or "__gt__")
- greater than or equal to (">=" or "__ge__")
- one of ("in" or "in_")

Note: Some relations can be referenced by more than one name as indicated in the brackets.

Most of these relations should be self-explanatory. However, the *like* and *regular expression* relations merit further discussion.

The *like* relation

The "like" relation is simply the SQL `LIKE` operator. The pattern following the "like" relation may contain the wildcard characters "%" and "_". The percent sign matches zero or more of any character while the underscore matches exactly one instance of any character. These wildcards are illustrated via some typical use cases below.

Find all forms whose transcription contains "t":

```
["Form", "transcription", "like", "%t%"]
```

Find all forms whose transcription begins with "T":

```
["Form", "transcription", "like", "T%"]
```

Find all forms whose transcription ends with "t":

```
["Form", "transcription", "like", "%t"]
```

Find all forms that contain "k", followed by any single character, followed by "t":

```
["Form", "transcription", "like", "%k_t%"]
```

Note: As indicated by the above examples, OLD filter expressions are case-sensitive.

---

[6] Substring pattern match is effected via the SQL `LIKE` relation. TALK ABOUT WILDCARDS HERE

The *regexp* relation

The "regexp" (a.k.a. "regex") relation implements regular expression matching. [7] Regular expressions are tools for specifying complex patterns on strings. As with the "like" relation described above, certain characters and constructions in "regexp" search patterns have special meanings.

By default, regular expressions perform a substring match. That is, an OLD filter expression like the one that follows will return all forms that contain the string "it" anywhere in the value of their transcription attribute.

```
["Form", "transcription", "regex", "it"]
```

We can refer to the beginning or end of the string using the anchors "^" and "$". For example, the following two filter expressions find all forms whose transcription begins with "T" or ends with "s", respectively.

```
["Form", "transcription", "regex", "^T"]
["Form", "transcription", "regex", "s$"]
```

The period "." matches any character. For example, the OLD filter expression below will match all forms that have "kat", "kit", "kst", "kqt", etc. in their transcription values.

```
["Form", "transcription", "regex", "k.t"]
```

It is also possible to specify a pattern that matches a limited set of characters using character classes, i.e., sequences of characters enclosed in square brackets. For example, the following OLD filter expression will match all forms whose transcription value contains "k", followed by a vowel, followed by "t". (Of course, unicode characters are permitted as well so accented and IPA vowels could be specified here also.)

```
["Form", "transcription", "regex", "k[aeiou]t"]
```

If the caret character "^" is the first character in the character class, then the class matches any character except those it contains. For example, the following OLD filter expression will match all forms whose transcriptions contain a "k", followed by *anything but* a "q" or another "k", followed by a "t".

```
["Form", "transcription", "regex", "k[^qk]t"]
```

The vertical bar "|" is the alternation metacharacter. It matches either the string to its left or the string to its right. For example, the following OLD filter expression will return all forms containing a translation that contains either "the cat ran" or "the dog ran".

```
["Form", "translations", "transcription", "the (cat|dog) ran"]
```

Regular expressions also support quantification. That is, it is possible to specify that a pattern zero or one times (using "?"), zero or more times (using "*"), one or more times (using "+"), exactly *n* times (using "{n}"), between *n* and *m* times (using "{n,m}") and *n* or more times (using "{n,}").

For example, to find all forms whose transcription is a single word with one syllable whose nucleus is transcribed using exactly two vowels, an OLD filter expression like the following might be appropriate.

```
["Form", "transcription", "regex", "^[ptkmns][aeiou]{2}[ptkmns]$"]
```

Quantifiers could also be used to filter resources by the length of one of their fields. For example, to find all forms whose transcriptions contain at least five but no more than ten characters, one could use the following OLD filter expression.

```
["Form", "transcription", "regex", "^.{5,10}$"]
```

[7] With MySQL as RDBMS, the "regexp" relation is simply the standard MySQL `REGEXP` operator, i.e., an implementation of POSIX extended regular expressions. Since SQLite does not implement a `REGEXP` operator, the OLD supplies one using the standard `re` Python module. The table on this page does a good job of detailing the difference between these two regular expression implementations.

Note: Regular expressions will treat unicode combining characters as separate characters. Since the OLD applies unicode canonical decomposition normalization [8] on all input, a string like "á" will be interpreted by the regular expression parser as containing two strings, the "a" and the COMBINING ACCUTE ACCENT (u+0301) character. Keep this in mind when using regular expression quantifiers to filter based on string length or when using character sets. In the latter case, it is usually safer to use parentheses and the alternation metacharacter than character sets. To illustrate, consider the two examples below. The first OLD filter expression will match "oao", "oio" and "óo", which is probably not what was intended. The second filter expression will match "oao" and "oío", which is probably what was intended.

```
["Form", "transcription", "regex", "o[aí]o"]
["Form", "transcription", "regex", "o(a|í)o"]
```

Ordering results

In making a search request of an OLD web service, it is possible to specify the order in which the results are returned. This is accomplished by specifying an `orderBy` attribute for the JSON `query` object that is passed as input in the body of the request. Remember that OLD search requests must contain an object of the following form (where the `orderBy` attribute is optional).

```
{"query": {
    "filter": [ ... ] ,
    "orderBy": [ ... ]}}
```

The value of the `orderBy` attribute is an array containing exactly three strings where the first is the name of a model/resource, the second the name of an attribute of the model and the third is a direction, i.e., "asc" or "desc". For example, the following JSON object passed in the body of a request to `SEARCH /forms` would return all forms whose transcription begins with "p" ordered by id in descending order.

```
{"query": {
    "filter": ["Form", "transcription", "regex", "^p"],
    "orderBy": ["Form", "id", "desc"]}}
```

2.2.3 Non-standard API

This section describes the valid requests that are not covered by the standard RESTful and search interfaces documented in the previous sections. A subset of OLD resources possess such supplemental interfaces. This section is organized by resource.

Forms

Form resources represent linguistic forms and are the core of an OLD web service. The non-standard interfaces of form resources are described here.

`GET /forms/history/id`

Requests to `GET /forms/history/id` are routed to the `history` action of the `forms` controller. Such requests return a JSON object representing the history, or previous versions, of the form with the specified id. The id parameter

[8] Cf. http://unicode.org/reports/tr15/

can be the integer id or the Universally Unique Identifier (UUID) of the form. [9] The JSON object returned is of the form

```
{"form": { ... }, "previousVersions": [ ... ]}
```

where the value of the "form" attribute is the JSON representation of the form while the value of "previousVersions" is an array of objects representing the previous versions of the form. If the form has been deleted, the value of the "form" attribute will be `null` and if the form has not been updated or deleted, the value of the "previousVersions" attribute will be an empty array.

POST /forms/remember

Requests to `POST /forms/remember` are routed to the `remember` action of the `forms` controller and cause the forms referenced in the request body to be appended to the `rememberedForms` collection of the user making the request. The expected input is an object of the form

```
{"forms": [id1, id2, ... ]}
```

where `id1`, `id2`, etc. are form integer ids.

PUT /forms/update_morpheme_references

Requests to `PUT /forms/update_morpheme_references` regenerates values for the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes of *all* forms in the system. (See the *Morphological processing* and *Form* sections for details on these attributes.) The response generated by this request contains a JSON array of ids corresponding to the forms that were updated. Only administrators are authorized to make this request.

> **Warning:** It should not be necessary to request the regeneration of morpheme references via this request since this should already be accomplished automatically by the call to `updateFormsContainingThisFormAsMorpheme` on all successful update and create requests on form resources. This interface is, therefore, deprecated (read: use it with caution) and may be removed in future versions of the OLD.

Files

OLD file resources are representations of binary files stored on a filesystem. From a linguist's point of view, they are the audio/video records of linguistic fieldwork, the images (or audio or video) used as stimuli, PDFs of relevant papers or handouts, etc. – anything that is relevant to a piece or a collection of language data. Multiple file resources can be associated to a given form or collection resource. Thus, for example, a form representing a sentence could be associated to a large audio recording of an elicitation session, a smaller audio recording of just the sentence being uttered, an image used to illustrate a context for a speaker, etc. See the *File* section for more details on files.

GET /files/serve/id

Requests to `GET /files/serve/id` return the file data of the file resource with the given id, assuming the authenticated user is authorized to access that resource. If the file with the specified id is a subinterval-referencing file,

[9] Since some RDBMSs reuse primary key integers when a record is deleted, it is not possible to associate forms and collections to their backups via their integer id attributes. Therefore, both form and collection resources have UUID attributes and are associated to their backup objects via both `form_id`/`collection_id` and `UUID` attributes. The safest way, therefore, to request all of the backups of a given form/collection, therefore is to pass the UUID to the relevant `history` GET request.

the file data of the parent file is returned; if the file data are hosted externally, an explanatory error message is returned. (See the *File* for an explanation of subinterval-referencing and externally hosted files.)

GET /files/serve_reduced/id

Requests to `GET /files/serve_reduced/id` return the file content of the reduced-size copy of the file which was created by the OLD upon file creation. If there is no reduced-size copy of the file, the OLD returns an error message. These requests handle subinterval-referencing and externally hosted files in the same way as described in the above subsection.

Collections

Collections are documents that can reference forms and are useful for creating records of elicitation sessions or for writing papers using data stored on an OLD application. See the *Collection* section for more details on collections.

GET /collections/history/id

Requests to `GET /collections/history/id` are routed to the `history` action of the `collections` controller and return a JSON object representing the history, or previous versions, of the collection with the specified id. The id parameter can be the integer id or the Universally Unique Identifier (UUID) of the collection. [10] The JSON object returned is of the form

```
{"collection": { ... }, "previousVersions": [ ... ]}
```

where the value of the "collection" attribute is the JSON representation of the collection while the value of "previousVersions" is an array of objects representing the previous versions of the collection. If the collection has been deleted, the value of the `collection` attribute will be `null` and if the collection has not been updated or deleted, the value of the `previousVersions` attribute will be an empty array.

Application settings

The application-wide settings for an OLD application are stored as application settings objects. These resources have non-standard interfaces insofar as only administrators are permitted to create, update or delete them. Other types of users can only read them, i.e., request `GET /applicationsettings` and `GET /applicationsettings/id`. The application settings resources are also unique in that the most recently created one (i.e., that with the largest id) is designated as the *active* application settings and is the one that affects the behaviour of the rest of the application. Therefore, application-wide behaviour may be configured either by updating the active application settings resource or by creating a new (and hence active) one. The latter approach is recommended since the previously created application settings resources will provide a history of previous configurations.

Users

User resources represent the users (i.e., administrators, contributors and viewers) of an OLD application. The interface to this resource is non-standard in that only administrators are authorized to create or delete user resources and a user resource can only be updated by administrators and the holder of the user account. See the *User* section for more details on users.

Remembered forms

Each OLD user has a `rememberedForms` attribute whose value is a collection of zero or more form resources that the user has memorized. Since these collections can grow quite large, they are treated as a resources of their own and are not affected by interactions with user resources. The interface to the remembered forms resources are non-standard in that ...

GET /rememberedforms/id

Requests to `GET /rememberedforms/id` return the array of forms remembered by the user with the supplied id. Such requests are routed to the `show` action of the `rememberedforms` controller. Ordering and pagination parameters may be provided in the query string of this request in exactly the same way as with standard `GET /resources` requests of conventional resources (cf. *GET /resources*).

UPDATE /rememberedforms/id

Requests to `UPDATE /rememberedforms/id` are routed to the `update` action and set the remembered forms of the user with the supplied id to the set of forms referenced in the JSON array of form ids sent in the request body. This type of request accomplishes creation, updating and deletion of a remembered form "resource". Only administrators and the user with the supplied id can make licit requests to `UPDATE /rememberedforms/id`. As with requests to `POST /forms/remember`, requests to `UPDATE /rememberedforms/id` should contain a JSON request body of the form `{"forms": [16, 28, 385]}`.

Note: The `remember` action of the forms controller has a similar, but more restricted, effect, i.e., requests to `POST /forms/remember` can add forms to (but not delete them from) the remembered forms collection of the user who makes the request.

SEARCH /rememberedforms/id

Requests to `SEARCH /rememberedforms/id` return all form resources remembered by the user with the supplied id and which match the JSON search filter passed in the request body. These requests are routed to the `search` action. Requests to `POST /rememberedforms/id/search` have the same effect as those to `SEARCH /rememberedforms/id`.

Note: The same effect can be achieved by conjoining the filter expression `["Memorizer", "id", "=", id]` to an existing search on form resources, i.e., a request to `SEARCH /forms`.

2.2.4 Authentication & authorization

Speakers of endangered languages and their communities often require that the language data gathered by researchers not be made available to the public at large. Therefore, authentication (i.e., a username and password) is required in order to access data on an OLD web service [10].

In addition to authentication, the OLD possesses a role-based system of authorization. The three roles are *administrator*, *contributor* and *viewer*.

[10] Future versions of the OLD may make authentication a configurable option, thus allowing publicization of all data. Another possibility is that the system could allow users to tag some data as public and that these data could be accessed without authentication. A final possibility would be to publicize all data but allow some data to be encrypted such that only authenticated users could decrypt them.

Viewers are only able to perform read requests, e.g., view all form resources, retrieve a particular file resource, search the collections resources, etc.

Contributors have read and write access to most resources, with some restrictions. Contributor *U1* is not permitted to delete a form, file or collection entered by contributor *U2*. Only administrators and *U1* can delete a form, file or collection entered by *U1*. In addition, only administrators and user *U1* are permitted to update the user resource representing *U1*.

Administrators have unrestricted access to read and write any resource. Only administrators can create or delete users and only administrators have write access to application settings resources.

Separate from the role-based division of users is a classification into restricted and unrestricted users. While administrators are, by default, always unrestricted, the application settings can specify a subset of contributors and viewers as unrestricted. Only unrestricted users are permitted to access restricted objects, i.e., forms, files or collections tagged with the "restricted" tag. Users not classified as unrestricted (i.e., restricted users) are unable to access restricted objects in any way. Since core objects can be associated to one another (e.g., a form can be associated to multiple files), restricted status can spread from object to object. For example, an unrestricted form becomes restricted as soon as it is associated to a restricted file.

The `login` controller effects authentication. Its interface is detailed in the following table.

HTTP Method	URL	Effect	Parameters
POST	/login/authenticate	Attempt to authenticate	JSON object
GET	/login/logout	De-authenticate	
POST	/login/email_reset_password	Email a newly generated password to the user	JSON object

`POST /login/authenticate` attempts authentication using the provided input, i.e., a JSON object on the request body of the form `{"username": " ... ", "password": " ... "}`. If successful, authenticated status is persisted across requests via a cookie-based `session` object where the value of `session['user']` is the user model of the authenticated user.

A `GET /login/logout` request removes the `'user'` key from the `session` object associated with the cookie passed in the request. That is, it de-authenticates, or logs out, the user.

A `POST /login/email_reset_password` request with a JSON object in the request body of the form `{"username": " ... "}` attempts to create a new, randomly generated password for the user with the provided username and notify the user via email of the change. If the server is unable to send email, the password will not be reset and a JSON error message will be returned in the response.

Note: If an SMTP mail server cannot be used, it is possible (as detailed in the comments of the config file that is generated when `paster make-config` is run) to configure an OLD application to send email via a specified Gmail account.

For more details on the authentication and authorization scheme of the OLD, please consult the API documentation and/or the source code. Most relevant are the `lib/auth.py`, `controllers/login.py`, `controllers/forms.py`, `controllers/files.py` and `controllers/oldcollections.py` modules.

2.2.5 Input validation

When users attempt to create a new resource or update an existing one, the OLD attempts to validate the input. If validation fails, the status code of the response is set to 400 and a JSON object explaining the issue(s) is returned, i.e., an object of the form `{'error': 'error message'}` or `{'errors': {'field name 1': 'error message 1', 'field name 2': 'error message 2'}}`.

Standard validation

Standard validation is validation on user input that is applied by all OLD applications in the same way.

Some representative examples will illustrate. All forms require some string in their transcription field and at least one translation. References to other OLD resources via their ids are validated for existence; e.g., when an elicitor for a form is specified via a user id, then validation ensures that the id corresponds to a user in the database. User-supplied values for date fields must be in `mm/dd/yyyy` format. Emails must be correctly formatted. Files uploaded must be one of the allowed file types (e.g., .jpg, .wav) of the OLD.

The Pylons controller classes that control the creation and updating of resources ensure that all such validation is passed before these requests can succeed. The validators that encode these validations are written using the FormEncode library and can be found in the `lib/schemata.py` module of the OLD source. For further information on input validation, consult the *Data Structure* section, the API documentation and/or the source code.

Object language validation

In addition to the standard validation described above, particular OLD applications can control how, or whether, transcriptions of the object language are validated. The relevant form attributes are `transcription`, `phoneticTranscription`, `narrowPhoneticTranscription` and `morphemeBreak`. By configuring the OLD application's settings, adminstrators can control what types of strings are permitted in these fields. This is useful for when groups of researchers want to ensure that, say, all morpheme segmentation strings (i.e., `morphemeBreak` values) are restricted to sequences of phonemes from the specified inventory plus the specified morpheme delimiters.

The table below shows how object language transcription validation is configured.

Form attribute	Relevant inventory or orthography	Validation parameter
transcription	storageOrthography	orthographicValidation
phoneticTranscription	broadPhoneticInventory	broadPhoneticValidation
narrowPhoneticTranscription	narrowPhoneticInventory	narrowPhoneticValidation
morphemeBreak	phonemicInventory*	morphemeBreakValidation

The validation parameter column lists the attributes of the application settings resource that control whether the form attribute in the first column should be validated against the relevant inventory or orthography. Each of the attributes in the validation parameter column can have one of three possible values: `None`, `Warning` or `Error`. Only if the attribute is set to `Error` will inventory/orthography-based validation occur.

For example, if the current application settings resource has `orthographicValidation` set to `Error`, then input validation will ensure that form transcriptions contain only graphemes (i.e., characters or character sequences) from the storage orthography plus punctuation characters and the space character.

When validation is enabled on the phonetic transcription fields, only graphs from the specified inventory plus the space character are permitted (i.e., no punctuation).

The `morphemeBreak` attribute's validation settings are slightly more complex since it is possible to choose between the storage orthography or the phonemic inventory when configuring validation. This is done by setting the `morphemeBreakIsOrthographic` attribute of the application settings resource to `true` in the former case and `false` in the latter. For example, if `morphemeBreakIsOrthographic` is set to `false` and `morphemeBreakValidation` is set to `Error`, then input to the `morphemeBreak` field will be rejected if it contains characters outside of the specified phonemic inventory, the specified morpheme delimiters and the space character.

As implied in the above discussion, the application settings resource has `morphemeDelimiters` and `punctuation` attributes for specifying sets of valid morpheme delimiters and punctuation, respectively.

Sometimes it is desirable to include foreign words in the object language transcriptions while still permitting validation against inventories and orthographies on these fields. For example, in a system where `morphemeBreak` validation is enabled and the phonemic inventory is /p/, /t/, /k/, /i/, /a/, /u/, it might be desirable to allow a `morphemeBreak` value

of “ki dog katti” but prohibit “ki dog kotti”. The OLD permits this via the special “foreign word” tag on form resources. When a form is tagged as a foreign word, its transcription values affect validation. So, if the system were to contain a foreign word form with “dog” as its `morphemeBreak` value, then validation would correctly allow both instances of “dog” in the above two examples while disallowing the latter example because of the illicit “o” in “kotti”. The function `updateApplicationSettingsIfFormIsForeignWord` is called in the `forms` controller upon successful create and update requests and is responsible for updating the validators with the foreign word information.

2.2.6 Processing

When requests cause resources to be created or updated, the OLD may perform some additional processing that may affect the values of certain attributes of the target resource or even of other resources. The notable data processing functionalities are listed below and are detailed in their own subsections.

- the generation of values for form attributes related to morphological analysis
- the updating of transcription validators when foreign words are entered
- the resolution and cacheing of collection-collection and collection-form cross-references
- the creation of reduced-size copies of the binary files of file resources

Morphological processing

Values for four attributes of form resources related to morphological analysis are generated on create and update requests. These are the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes. The function `compileMorphemicAnalysis` in the `forms` controller is responsible for generating these values.

The values of the `morphemeBreakIDs` and `morphemeGlossIDs` attributes are arrays that hold references to other forms that match the morphemes indicated in the user-defined `morphemeBreak` and `morphemeGloss` attributes. Each array has one array per word in the relevant field, each word array has one array per morpheme and each morpheme array has one array per match found. Matches are ordered triples where the first element is the id of the match, the second is the `morphemeBreak` or `morphemeGloss` value of the match and the third is the `syntacticCategory.name` of the match or `null` if no category is specified. As illustration, consider a database containing the following forms.

id	transcription	morphemeBreak	morphemeGloss	syntacticCategory.name
1	chien	chien	dog	N
2	s	s	PL	Agr
3	s	s	PL	Num
4	le	le	the	D
5	cour	cour	run	V
6	ent	ent	3.PL	Agr
7	les chiens courent	le-s chien-s cour-ent	the-PL dog-PL run-3PL	S

When the form with id 7 is entered, the system will generate the following arrays for the `morphemeBreakIDs` and `morphemeGlossIDs` attributes.

```
morphemeBreakIDs = [
    [
        [[4, 'the', 'D']],
        [[2, 'PL', 'Agr'], [3, 'PL', 'Num']]
    ],
    [
        [[1, 'dog', 'N']],
        [[2, 'PL', 'Agr'], [3, 'PL', 'Num']]
    ],
```

```
    [
        [[5, 'run', 'V']],
        [[6, '3.PL', 'Agr']]
    ]
]
morphemeGlossIDs = [
    [
        [[4, 'le', 'D']],
        [[2, 's', 'Agr'], [3, 's', 'Num']]
    ],
    [
        [[1, 'chien', 'N']],
        [[2, 's', 'Agr'], [3, 's', 'Num']]
    ],
    [
        [[5, 'cour', 'V']],
        []
    ]
]
```

Note: The `morphemeBreakIDs[0][1]` value contains two match triples because the second morpheme of the first word in the `morphemeBreak` line, i.e., "s", matches two forms, i.e., the forms with ids 2 and 3. Similarly, `morphemeGlossIDs[0][1]` contains two analogous match triples, the difference in this case being that the morpheme's phonemic/orthographic representation is listed and not its gloss. In contrast, the morpheme break "ent" matches form 6, hence the single match triple in `morphemeBreakIDs[2][1]`, whereas "3PL" matches nothing, hence the absence of matches in `morphemeGlossIDs[2][1]`.

The purpose of the `morphemeBreakIDs` and `morphemeGlossIDs` attributes is that they record the extent to which the morphemic analysis of a given form is in accordance with the lexical items listed in the database. If these values were not generated server-side upon create and update requests, then for any user-facing application to display such information would require many requests and database queries each time a form were displayed. The information in these two attributes is quite valuable in that it can be used to immediately inform users when the lexical items implicit in their morphological analyses are not yet listed in the database or when small differences in, say, glossing conventions are masking underlying consensus in analysis.

At the same time as the `morphemeBreakIDs` and `morphemeGlossIDs` values are generated, so too are the values for the `syntacticCategoryString` and `breakGlossCategory` attributes. These values for our example form 7 from above would be:

```
syntacticCategoryString = 'D-Agr N-Agr V-Agr'
breakGlossCategory = 'le|the|D-s|PL|Agr chien|dog|N-s|PL|Agr cour|run|V-ent|3PL|Agr'
```

The value of the `syntacticCategoryString` attribute is a string of syntactic category names corresponding to the string of morphemes in the morphemic segmentation.[#f11]_ Since the syntactic category string can be used to filter form resources on search requests, its generation facilitates search based on high-level morphological patterns. For example, using the syntactic category string, one could use regular expressions to search for all forms consisting of an NP followed by a VP.

Note: Given our example dataset, `'D-Num N-Num V-Agr'` is a reasonable (and perhaps preferable) syntactic category string value. However, the system has no way of knowing this and therefore when there are two matches for a morpheme (as there are for "s") it arbitrarily chooses the syntactic category of the lexical form with the lowest id.

The value of `breakGlossCategory` is a string that unambiguously represents the morphemic analysis of the form. Each morpheme is taken to be a triplet consisting of a phonemic representation (i.e., the `morphemeBreak` value), a semantic representation (i.e., the `morphemeGloss` value) and a categorial value (i.e.,

the `syntacticCategory.name` value). These break-gloss-category triplets are delimited by the vertical bar "|" and each such triplet is joined using the morpheme delimiters of the `morphemeBreak` value.

This attribute makes it possible to search for forms that contain a specific morpheme. Consider the case where one wanted to find all forms containing the morpheme "s" glossed as "PL" of category "Num". Performing a regular expression search on the `morphemeBreak` line for the pattern `-s( |-|$)` (i.e., "-s" followed by a space, "-" or the end of the string) would be insufficient since it might also find forms containing an "s" morpheme with a different gloss. Conjoining the above regular expression filter with another on the `morphemeGloss` line with the pattern `-PL( |-|$)` would still be insufficient since it would (contra what is desired) match a form with a `morphemeBreak` value of "le-s oiseau-x" and a `morphemeGloss` value of "the-plrl bird-PL". By searching the forms according to those whose `breakGlossCategory` value matches the regular expression `-s\|PL\|Num( |-|$)`, one can be assured of finding all and only all the forms containing the morpheme "s"/"PL"/"Num"

Given the above discussion, it is evident that an update to an existing lexical form, the creation of a new one or the updating of the name of a syntactic category may require updating the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and/or `breakGlossCategory` values of a number of different forms. The OLD accomplishes this by calling `updateFormsContainingThisFormAsMorpheme` whenever a form is created or updated. This function first assesses whether the newly created/updated form is lexical and, if so, it selects all forms whose morphological analyses implicitly reference the lexical form and updates the relevant fields appropriately. Care is taken to reduce database select queries to an absolute minimum with the end result being that the majority of calls to `updateFormsContainingThisFormAsMorpheme` will require only one select query, i.e., the one to find all of the forms that reference the lexical item just created/updated. In addition, when the name of a (lexical) syntactic category is changed, `updateFormsContainingThisFormAsMorpheme` is called on each form that has that category.

Foreign words

Whenever a form is created, updated or deleted, the forms controller calls `updateApplicationSettingsIfFormIsForeignWord`. This function is responsible for updating the transcription validators of the application settings if the form is a foreign word. As described in *Object language validation*, forms tagged with the "foreign word" tag will create exceptions to the user-defined object language transcription validation. For example, if a form is entered with `transcription`, `morphemeBreak` and `morphemeGloss` values of "John", "John" and "John" and is tagged as a "foreign word", then the system will allow the string "John" to be included in the `transcription` field of other forms even if validation is set to reject forms whose transcriptions contain, say, "J" or "h".

Note: It is desirable to be able to enter such a lexical entry as "John" with a category of, say, "PN" since doing so will result in sensible `syntacticCategoryString` values for forms containing "John" in their `morphemeBreak` value.

Collection references

The `contents` attribute of collections is a string that may contain references to forms and other collections. These references determine the value of the `contentsUnpacked`, `html` and `forms` attributes.

When the value of the `contents` attribute of an existing collection is updated, the `update` action calls `updateCollectionsThatReferenceThisCollection` in order to update the `contentsUnpacked`, `html` and `forms` values of all of the collections that reference the updated collection. This same function is called when a collection is deleted; in this case, all references to the deleted collection are removed from any collections that were referencing it and the appropriate values are updated. Similarly, when a form is deleted, the `delete` action calls `updateCollectionsReferencingThisForm` and all references to the to-be-deleted form are removed from any collections that reference it.

See the *Collection* section for more details on collection references and the attributes whose values depend on them.

Lossy file copies

When new file models are created with locally stored file data, the OLD may create reduced-size copies of certain file types and store them, by default, in `files/reduced_files/`. Such lossy copies are created when `create_reduced_size_file_copies` is set to a truthy value (e.g., "1") in the config file and if the relevant utilities are installed, i.e., for images the Python Imaging Library and for WAV files the FFmpeg command-line utility. See the *Soft dependencies* and *File* sections for more details.

2.3 Data Structure

This page describes the data structure of the OLD. The OLD data structure is a representation of the artifacts of linguistic fieldwork and their properties. This data structure is implemented as tables and their inter-relations in a relational database. However, it is here presented using the language of *model objects* and their *attributes*, i.e., using the conceptual structure of the object-relational mapping provided by SQLAlchemy.

The prototypical OLD model object is the `form` which represents a linguistic form, i.e., a morpheme, word, phrase or sentence elicited by a linguistic fieldworker. Some of the representative attributes of the form model are `transcription`, `morphemeBreak`, `morphemeGloss`, `translations`, `grammaticality`, `speaker` and `dateElicited`.

This exposition is structured according to the models defined by the OLD.[11] Each section begins with an overview of the model. The attributes of the model are described and justified in alphabetically ordered subsections. Included in these subsections are specifications of what constitutes a licit[12] value for each attribute as well as the methods of construction for system-generated values. Each model section details the format of the input expected upon create or update requests as well as the format of the model when returned. Note that all of the attributes of the objects in the input descriptions must be present. In general, unspecified values should be represented as empty strings or JSON `null`. If the expected value is an array of ids of a given model, then unspecified is indicated by an empty array (`[]`). For example, the JSON object used to create a form resource with no elicitor and no files associated would (with other attributes omitted) look like `{"elicitor": null, "files": []}`.

The `id` and `datetimeModified` attributes are common to all models and are therefore described here in order to avoid repetition. The former is the integer value created by the RDBMS each time a new row is created in a table. Each model has an `id` value that is unique among all other models of that type. The larger the `id` value the more recently added is the model. The `datetimeModified` attribute holds a datetime value. It is a UTC timestamp generated by the application logic whenever a model is created or updated. Datetime values are returned by OLD web services as strings in ISO 8601 format, e.g., "2010-01-29T09:33:27".

A note on the terminology of *resources*, *controllers*, *models* and *tables*. There is a near 1-to-1-to-1-to-1 correspondence between the *resources* exposed by an OLD application, the *controllers* that facilitate interaction with them, the *models* that enode their structure and the RDBMS *tables* where their data are stored. For example, form resources are accessed via the `forms` controller and the data for each form is represented internally as a `form` model object which is persisted to a `form` table in the database. Some resources, such as the `rememberedforms` quasi-resource described in *Interface*, have no corresponding model or table while some tables, e.g., the `formtag` table that stores the many-to-many relations between the `form` and `tag` tables, have no model or controller. (Note that because of a naming conflict, the controller responsible for OLD collections resources is in `controllers/oldcollections.py` not `controllers/collections.py`.)

[11] The models are defined in the `model` directory of the source code. Each model has its own appropriately named module where it is declared. The form model, for example, is declared in `model/form.py`.

[12] The code that validates user input is located in `lib/schemata.py`.

Note finally that the OLD treats all strings as unicode. Data input to the database or written to disk are UTF-8 encoded. The OLD applies unicode canonical decomposition normalization [13] to all string data (including user input, search query patterns and system-generated data). This means that the character "a" will be stored as "LATIN SMALL LETTER A" (U+0061) followed by the combining character "COMBINING ACCUTE ACCENT" (U+0301) even when it is entered as the canonically equivalent "LATIN SMALL LETTER A WITH ACUTE" (U+00E1). Such normalization allows search and other functionality to work despite superficial differences in user input.

2.3.1 ApplicationSettings

An application settings model stores system-wide application settings. These settings affect such things as how input is validated, what the morpheme delimiters are, what the valid grammaticality values are, what the name of the language being studied is, etc.

Requests to create or update application settings resources must contain a JSON object of the following form.

```
{
    "broadPhoneticInventory": "",
    "broadPhoneticValidation": "",
    "grammaticalities": "",
    "inputOrthography": null, // integer id of a valid orthography model, or null or "" if unspecifie
    "metalanguageId": "",
    "metalanguageInventory": "",
    "metalanguageName": "",
    "morphemeBreakIsOrthographic": "",
    "morphemeBreakValidation": "",
    "morphemeDelimiters": "",
    "narrowPhoneticInventory": "",
    "narrowPhoneticValidation": "",
    "objectLanguageId": "",
    "objectLanguageName": "",
    "orthographicValidation": "",
    "outputOrthography": null, // integer id of a valid orthography model, or null or "" if unspecifi
    "phonemicInventory": "",
    "punctuation": "",
    "storageOrthography": null, // integer id of a valid orthography model, or null or "" if unspeci
    "unrestrictedUsers": [] // array of ids of valid user models, or [] if none are unrestricted
}
```

Application settings representations returned by the OLD are JSON objects of the following form.

```
{
    "broadPhoneticInventory": "",
    "broadPhoneticValidation": "",
    "datetimeModified": "",
    "grammaticalities": "",
    "id": 1,
    "inputOrthography": {}, // object representation of an orthography model
    "metalanguageName": "",
    "metalanguageId": "",
    "metalanguageInventory": "",
    "morphemeBreakIsOrthographic": "",
    "morphemeBreakValidation": "",
    "morphemeDelimiters": "",
    "narrowPhoneticInventory": "",
    "narrowPhoneticValidation": "",
    "objectLanguageId": "",
```

[13] Cf. http://unicode.org/reports/tr15/ and http://en.wikipedia.org/wiki/Unicode_equivalence.

```
    "objectLanguageName": "",
    "orthographicValidation": "",
    "outputOrthography": {}, // object representation of an orthography model
    "phonemicInventory": "",
    "punctuation": "",
    "storageOrthography": {}, // object representation of an orthography model
    "unrestrictedUsers": [] // array of objects representing user models
}
```

broadPhoneticInventory

The value of the `broadPhoneticInventory` attribute is a comma-delimited string representing the inventory of graphemes (i.e., single characters or strings of characters) that should be used to construct broad phonetic transcriptions, i.e., to construct values for the `phoneticTranscription` attribute of form models. The space character should not be included as a grapheme since the validation functionality will allow it by default.

broadPhoneticValidation

The `broadPhoneticValidation` attribute determines how or whether the input to the `phoneticTranscription` attribute of forms is validated. The permissible values of the `broadPhoneticValidation` attribute, as defined in the `validationValues` tuple of `lib/utils.py`, are "Error", "Warning" and "None". If the value is "Error", then the OLD will not permit a form to be created or updated if its `phoneticTranscription` value cannot be constructed using the graphemes in the broad phonetic inventory plus the space character. See the *Object language validation* section for more details.

grammaticalities

The `grammaticalities` attribute holds a comma-delimited list of grammaticality values that will be the available options for the `grammaticality` attributes of form models and the `grammaticality` attributes of translation models. The default value for this field is "*,#,?" as defined in the `generateDefaultApplicationSettings` function of `lib/utils.py`.

inputOrthography

The `inputOrthography` is a reference to an existing orthography model object. An orthography is essentially a list of graphemes (like an inventory) but with some extra settings (cf. the *Orthography* section). The purpose of a system-wide input orthography is to allow for the possibility that users will enter form transcriptions (and possibly also morpheme segmentations) using one orthography (i.e., the input orthography) but that these transcriptions will be translated into another orthography (i.e., the storage orthography) for storage in the database. When outputing the forms, the system would then re-translate them from the storage orthography into the output orthography. Previous OLD applications implemented this orthography conversion server-side. However, with the new architecture of the OLD >= 1.0 this added complication seems best implemented client-side as user-specific orthography conversion. Therefore, the `inputOrthography` attribute of the `ApplicationSettings` model may be removed in future versions of the OLD.

metalanguageId

The value of the `metalanguageId` attribute is a three-character language Id from the ISO 639-3 standard which unambiguously identifies the metalanguage of the application, i.e., the language used in the analysis and documentation of the object language. The OLD language resources contain the ISO 639-3 data; that

is, requesting `GET /languages` (or `SEARCH /languages`, `GET /applicationsettings/new` or `GET /applicationsettings/edit/id`) will return a JSON array containing all of the languages identified in the ISO 639-3 standard. The default value for the `metalanguageId` attribute is "eng".

metalanguageInventory

The value of the `metalanguageInventory` attribute is a comma-delimited string representing the inventory of graphemes (i.e., single characters or strings of characters) that should be used to construct the translations in the `translations` attribute of form models. Note that the OLD is not set up to use the inventory in the `metalanguageInventory` attribute for validation.

metalanguageName

The value of the `metalanguageName` is the name of the language that is used in the analysis (and translation) of the language under study (the object language). The default value for this attribute is "English".

morphemeBreakIsOrthographic

The value of the `morphemeBreakIsOrthographic` attribute controls what characters the system will expect to find in the values of the `morphemeBreak` attribute of forms. If `morphemeBreakIsOrthographic` is set to "true" (or "yes", "on" or "1"), then the system will expect the `morphemeBreak` value to be constructed using the graphemes defined in the `storageOrthography` attribute; if it is set to "false" (or "no", "off" or "0"), the system will expect graphemes from the `phonemicInventory` in the value of this attribute.

morphemeBreakValidation

The `morphemeBreakValidation` attribute determines how or whether the input to the `morphemeBreak` attribute of forms is validated. The permissible values of the `morphemeBreakValidation` attribute, as defined in the `validationValues` tuple of `lib/utils.py`, are "Error", "Warning" and "None". If the value is "Error", then the OLD will not permit a form to be created or updated if its `morphemeBreak` value cannot be constructed using the graphemes of the relevant orthography/inventory (cf. the `morphemeBreakIsOrthographic` attribute) plus the space character. See the *Object language validation* section for more details.

morphemeDelimiters

The `morphemeDelimiters` attribute holds a comma-delimited list of characters that the system should expect users will employ when segmenting morpheme transcriptions or morpheme glosses in the `morphemeBreak` and `morphemeGloss` fields, respectively. The default value for this attribute, as defined in the `generateDefaultApplicationSettings` function of `lib/utils.py`, is "-,=". If morpheme break validation is enabled, then these delimiter characters will be permitted in the `morphemeBreak` values in addition to the graphemes of the specified orthography/inventory. See the *Object language validation* section for more details.

narrowPhoneticInventory

The value of the `narrowPhoneticInventory` attribute is a comma-delimited string representing the inventory of graphemes (i.e., single characters or strings of characters) that should be used to construct narrow phonetic transcriptions, i.e., to construct values for the `narrowPhoneticTranscription` attribute of form models. The space character should not be included as a grapheme since the validation functionality will allow it by default.

narrowPhoneticValidation

The narrowPhoneticValidation attribute determines how or whether the input to the narrowPhoneticTranscription attribute of forms is validated. The permissible values of the narrowPhoneticValidation attribute, as defined in the validationValues tuple of lib/utils.py, are "Error", "Warning" and "None". If the value is "Error", then the OLD will not permit a form to be created or updated if its narrowPhoneticTranscription value cannot be constructed using the graphemes in the narrow phonetic inventory plus the space character. See the *Object language validation* section for more details.

objectLanguageId

The value of the objectLanguageId attribute is a three-character language Id from the ISO 639-3 standard which unambiguously identifies the language being documented using the application, i.e., the object language. The OLD language resources contain the ISO 639-3 data; that is, requesting GET /languages (or SEARCH /languages, GET /applicationsettings/new or GET /applicationsettings/edit/id) will return a JSON array containing all of the languages identified in the ISO 639-3 standard.

objectLanguageName

The value of the objectLanguageName is the name of the language that is being documented and analyzed using the OLD web service.

orthographicValidation

The orthographicValidation attribute determines how or whether the input to the transcription attribute of forms is validated. The permissible values of the orthographicValidation attribute, as defined in the validationValues tuple of lib/utils.py, are "Error", "Warning" and "None". If the value is "Error", then the OLD will not permit a form to be created or updated if its transcription value cannot be constructed using the graphemes in the storage orthography plus the space character and the specified punctuation. See the *Object language validation* section for more details.

outputOrthography

The outputOrthography is a reference to an existing orthography model object. An orthography is essentially a list of graphemes (like an inventory) but with some extra settings (cf. the *Orthography* section). The purpose of a system-wide output orthography is to allow for the possibility that users will enter form transcriptions (and possibly also morpheme segmentations) using one orthography (i.e., the input orthography) but that these transcriptions will be translated into another orthography (i.e., the storage orthography) for storage in the database. When outputing the forms, the system would then re-translate them from the storage orthography into the output orthography. Previous OLD applications implemented this orthography conversion server-side. However, with the new architecture of the OLD >= 1.0 this added complication seems best implemented client-side as user-specific orthography conversion. Therefore, the outputOrthography attribute of the ApplicationSettings model may be removed in future versions of the OLD.

phonemicInventory

The value of the phonemicInventory attribute is a comma-delimited string representing the inventory of phonemes that should be used to construct morpheme segmentations in the morphemeBreak attribute of form resources. See the *Object language validation* section for more details on configuring input validation for the morphemeBreak attribute of forms.

punctuation

The `punctuation` attribute holds a string representing a list of punctuation characters. There is no delimiter: each character in the string is considered a punctuation character. Thus the default value of `.,;:!?’"\’”[]{}()-` results in the following characters being identified as valid punctuation: FULL STOP, COMMA, SEMICOLON, COLON, EXCLAMATION MARK, QUESTION MARK, APOSTROPHE, QUOTATION MARK, LEFT SINGLE QUOTATION MARK, RIGHT SINGLE QUOTATION MARK, LEFT DOUBLE QUOTATION MARK, RIGHT DOUBLE QUOTATION MARK, LEFT SQUARE BRACKET, RIGHT SQUARE BRACKET, LEFT CURLY BRACKET, RIGHT CURLY BRACKET, LEFT PARENTHESIS, RIGHT PARENTHESIS, HYPHEN-MINUS. When orthographic validation is enabled, the system will allow the punctuation characters specified here to occur in the values of the `transcription` attribute of forms.

storageOrthography

The `storageOrthography` is a reference to an existing orthography model object. An orthography is essentially a list of graphemes (like an inventory) but with some extra settings (cf. the *Orthography* section). The storage orthography defines the character sequences that should be used to create form `transcription` values. If the `morphemeBreakIsOrthographic` attribute is set to “true”, then the form `morphemeBreak` values should also be constructed out of the graphemes defined in the `storageOrthography` (plus the morpheme delimiters specified in `morphemeDelimiters`). See the *Object language validation* section for details on how to configure orthography/inventory-based validation for form transcription attributes.

The system-wide storage orthography is also a component in an orthography conversion feature. Orthography conversion allows for the possibility that users will enter form transcriptions (and possibly also morpheme segmentations) using one orthography (i.e., the input orthography) but that these transcriptions will be translated into another orthography (i.e., the storage orthography) for storage in the database. When outputing the forms, the system would then re-translate them from the storage orthography into the output orthography. Previous OLD applications implemented this orthography conversion server-side. However, with the new architecture of the OLD >= 1.0 this added complication seems best implemented client-side as user-specific orthography conversion.

unrestrictedUsers

The `unrestrictedUsers` attribute is a collection of user models which identifies the set of users that are to be identified as *unrestricted*. Such users are authorized to access restricted form, file and collection resources while contributors and viewers who are not unrestricted (i.e., who are *restricted*) are unable to view (or, *a fortiori*, update) such resources. See the *Authentication & authorization* section for more details on authorization based on the “restricted” classification.

2.3.2 Collection

OLD collection models are documents that can contain both text (with markup) and references to form models in their `contents` attribute. They can be used for a number of purposes: to create a simple list of forms, to write an academic paper or a lesson plan, to document a conversation or narrative, etc. The value of the `contents` attribute is a document written using one of the lightweight markup languages reStructuredText or Markdown. OLD collections can embed other OLD collections via reference. As reStructuredText or MarkDown documents, they can be converted to HTML and, in the case of collections written using reStructuredText, they can be converted to (Xe)LaTeX (whence to PDF) and Open Document Format (i.e., .odt; whence to Word, i.e., .doc).

Collection creation and update requests must contain a JSON object of the following form.

```
{
    "contents": "",
    "dateElicited": "",
```

```
    "description": "",
    "elicitor": null, // valid user model id or null
    "files": [] // array of valid file model ids or []
    "markupLanguage": "",
    "source": null, // valid source model id or null
    "speaker": null, // valid speaker model id or null
    "tags": [], // array of valid tag model ids or []
    "title": "My Collection",
    "type": "",
    "url": "",
}
```

Collection representations returned by the OLD are JSON objects of the following form.

```
{
    "contents": "",
    "contentsUnpacked": "",
    "dateElicited": "",
    "datetimeEntered": "",
    "datetimeModified": "",
    "description": "",
    "elicitor": null, // an object representation of a user or null
    "enterer": { ... }, // an object representation of a user
    "files": [], // an array of object representations of files or []
    "forms": [], // an array of object representations of forms or []
    "html": "",
    "id": 1,
    "markupLanguage": "",
    "source": null, // an object representation of a source or null
    "speaker": null, // an object representation of a speaker or null
    "tags": [], // an array of object representations of tags or []
    "title": "",
    "type": "",
    "url": "",
    "UUID": ""
}
```

contents

The value of the `contents` attribute is a string that constitutes the content of the collection. If markup is used, it should be the markup specified in the `markupLanguage` attribute.

The value of this attribute can contain references to form models in the database. These references are strings like `form[136]` or `Form[136]`, i.e., the string "form" or "Form", followed by a left bracket "[", followed by a valid form model id, followed by a right bracket "]". The reference "form[136]" would result in the form with id 136 being associated to the collection, i.e., `collection.forms` would contain that form.

Note that the value of the `contents` attribute need not contain any markup or other text. That is, it may simply be a string consisting of references to forms.

Here is an example of a well-formed `contents` value that uses the MarkDown markup language and contains a reference to the form with id 136:

```
Chapter 2
=========

Section containing a list
-------------------------
```

```
* Item 1
* Item 2

Section containing forms
------------------------

form[136]
```

It is also possible to reference another collection within the value of the `contents` attribute. This causes the contents of first collection to behave as though it contained the contents of the referenced collection in its contents value at the point of reference. For example, consider collection *C2* below which references collection *C1* (with id 3) from above.

```
Chapter 1
=========

Section containing prose
------------------------

Blah blah pied piping ... blah blah.

Section containing forms
------------------------

form[135]

collection[3]
```

When collection *C2* is created, the `collections` controller will generate the following value for `contentsUnpacked`:

```
Chapter 1
=========

Section containing prose
------------------------

Blah blah pied piping ... blah blah.

Section containing forms
------------------------

form[135]

Chapter 2
=========

Section containing a list
-------------------------

* Item 1
* Item 2

Section containing forms
------------------------

form[136]
```

The above `contentsUnpacked` value will be used to extract the form references of the collection and to generate

the value of the `html` attribute. That is, collection *C2* will be associated to forms 135 and 136. Note that collection-collection references can be nested, i.e., collections can reference collections which reference other collections, etc.

contentsUnpacked

The value of the `contentsUnpacked` attribute is the value of the `contents` attribute when all of its collection references are replaced with the contents of the collections referred to. These referred-to collections can refer to others in turn and all such references are replaced by the appropriate `contents` values. The form models associated to a collection are calculated by gathering all of the form references in the value of the `contentsUnpacked` attribute.

A result of collection-to-collection referencing is that the `contents` and `forms` values of a collection may be altered by updates to other collections. The forms controller handles this by calling `updateCollectionsThatReferenceThisCollection` upon successful update requests.

dateElicited

The `dateElicited` attribute is a user-supplied date value which indicates the date when the collection was elicited. The date must be in mm/dd/yyyy format. This is applicable to collections that represent records of events, e.g., elicitation sessions, recordings of stories, etc.

datetimeEntered

The value of the `datetimeEntered` attribute is a UTC timestamp generated by the system when a collection is created. Note that this value is distinct from the `datetimeModified` attribute that is common to all model types since that value is generated upon creation *and* update requests while the `datetimeEntered` value is only generated upon creation requests and is not altered thereafter.

description

The value of the `description` attribute is a user-supplied string that describes the collection.

elicitor

The `elicitor` attribute references a valid user model who is the elicitor of the collection. This attribute may not be appropriate for all collection types.

enterer

The `enterer` attribute references the user model whose account was used to create the collection. This value is generated automatically by the system upon collection creation.

files

A collection may be associated to zero or more files via the `files` attribute which references a collection [14] of file models. Files are OLD objects that represent a binary file (e.g., an audio, video or image file) along with metadata. An example use case would be a collection that represents an elicitation session and which is associated to one or more

[14] Note the distinction between OLD *collections* which are a type of model and *collections* in the ORM sense where the term refers to a type of model attribute which references a set of zero or more other models. E.g., `form.files` is a collection of file models and is an example of a collection in the second sense.

files whose file data are large audio recordings of the session. See the *File* section for details on the structure of file models.

forms

A collection may be associated to zero or more forms. These are stored in the `forms` attribute, which references a collection of form models. Whereas files are associated to an OLD collection by specifying an array of file ids in the `files` attribute of the JSON object passed to collection create/update requests, forms are associated indirectly, that is by being referenced in the value of the `contents` attribute of the collection (cf. the *contents* section).

html

The value of the `html` attribute is a string of HTML that is generated by the system using the value of the `contentsUnpacked` attribute and the markup-to-HTML function corresponding to theh markup language specified in the `markupLanguage` attribute. Note that while the HTML could be generated in the user-facing application, there is not, to my knowledge, a JavaScript implementation of the reStructuredText markup-to-HTML algorithm; therefore the HTML generation is performed server-side. Note also that form references are left as-is, which is to say that no HTML representation of the form data is generated. This is left as a task for the user-facing application since applications will have their own method(s) of displaying forms.

markupLanguage

The value of the `markupLanguage` attribute is one of "Markdown" or "reStructuredText" as defined in the `markupLanguages` variable of `lib/utils.py`. Markdown and reStructuredText are *lightweight markup languages*. A lightweight markup language is a markup language (i.e., a system for annotating a document) that is designed to be easy to read in its raw form. If no value is specified, "reStructuredText" will be the default.

source

The `source` attribute references a valid source model that indicates the textual (or other) source of the collection. This is useful for when the content of a collection is taken from another document and that fact needs to be attributed. The structure of the source model is based on the BibTeX format. See the *Source* section for details.

speaker

The `speaker` attribute references a valid speaker model who is the speaker or consultant of the collection. As with attributes like `elicitor`, the `speaker` attribute may not be appropriate for all collection types.

tags

A collection may be associated to zero or more tags and these associations are stored in the `tags` attribute. Tags are user-defined models that can be used to arbitrarily categorize other OLD models. If a collection is to be restricted, the special "restricted" tag should be associated to it. See the *Tag* section for details.

title

The value of the `title` attribute is a string that is the title of the collection. All collections must have a title and no title may exceed 255 characters.

type

The value of the `type` attribute is used to classify the collection and may affect how it is displayed or exported. The permitted values, as defined in `collectionTypes` in `lib/utils.py`, are "story", "elicitation", "paper", "discourse" and "other". If no value is specified, `null` is the default.

url

The value of the `url` attribute is not actually a valid URL but something more akin to the *path* component of a URL. That is, it is a string composed of any of the 26 letters of the English alphabet (including upper-case versions), the underscore "_", the forward slash "/" and the hyphen "-". The `url` value must not exceed 255 characters. At present the OLD qua web service does not make use of this attribute. However, it may be used by a user-facing application to allow users to navigate to a specific collection using something more meaningful than an integer id. For example, on a web application front-end to an OLD web service with the URL `http://www.xyz-old.org`, one might navigate to a representation of the collection entitled "Magnum Opus" by entering `http://www.xyz-old.org/magnum_opus` in the address bar (where "magnum_opus" is the value of the `url` attribue.)

UUID

The value of the `UUID` attribute is a universally unique identifier (UUID), i.e., a number represented by 32 hexadecimal digits displayed in five groups using four hyphens. A valid UUID is a 36-character string that looks like `aba3ea8d-b56f-4934-a8f7-68cba500f411`. The collections controller (i.e, `oldcollections`) randomly generates a UUID value for each newly created collection model. These values are used to associate collection backups to the collections they backup.

2.3.3 CollectionBackup

A collection backup model is created whenever a collection model is updated or deleted. These models cannot be created directly, i.e., `POST /collectionbackups` is not a valid request. The collection backup model receives all of the attributes of the model that it backs up. It also has some additional attributes, viz. `collection_id` and `backuper`. The value of the `collection_id` attribute is the value of the `id` attribute of the collection that was backed up to create the present collection backup model. The value of the `backuper` attribute is a JSON object representing the user who created the backup (by deleting or updating the collection). In general, the values of the relational attributes of the collection (i.e., the attributes that refer to other models) are converted to JSON object representations in the collection backup model. For example, the value of the `speaker` attribute is such a JSON object and the value of the `files` attribute is a JSON array of such objects representing file models. Since form models have many attributes and since collection models will, typically, be associated to many form models, the `forms` attribute of a collection backup model is simply a JSON array of form `id` values. If the collection has just been deleted, then the value of the `datetimeModified` value of the collection backup will be the UTC datetime at the time of deletion.

Collection backup representations returned by the OLD are JSON objects of the following form.

```
{
    "backuper": { ... } // an object representation of a user
    "collection_id": 1
    "contents": "",
    "contentsUnpacked": "",
    "dateElicited": "",
    "datetimeEntered": "",
    "datetimeModified": "",
    "description": "",
```

```
    "elicitor": null, // an object representation of a user or null
    "enterer": { ... }, // an object representation of a user
    "files": [], // an array of object representations of files
    "forms": [], // an array of object representations of forms
    "html": "",
    "id": 1,
    "markupLanguage": "",
    "source": null, // an object representation of a source or null
    "speaker": null, // an object representation of a speaker or null
    "tags": [], // an array of object representations of tags
    "title": "",
    "type": "",
    "url": "",
    "UUID": ""
}
```

2.3.4 ElicitationMethod

Elicitation method objects represent a set of tags for categorizing the way in which a form was elicited. For example, sometimes a researcher asks a consultant "How do you say 'Every man loves a woman.'?" An elicitation method used to categorize forms elicited in this way might have a `name` value of "translated English". Sometimes a researcher asks a consultant "Does this sound like a good sentence: 'Il y a une femme que tous les hommes aiment.'?" The elicitation method for such forms might have a name of "judged object language utterance of researcher".

Elicitation method creation and update requests must contain a JSON object of the following form.

```
{
    "description": "",
    "name": ""
}
```

Elicitation method representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "description": "",
    "id": 1,
    "name": ""
}
```

description

The value of the `description` attribute is a user-supplied string that describes the elicitation method and (perhaps) provides guidance on its use.

name

The value of the `name` attribute is an obligatory, user-supplied string of no more than 255 characters which must be unique among all other elicitation method names.

2.3.5 File

OLD file model objects are binary files with metadata. From the language researcher's point of view, they are the audio/video recordings of linguistic fieldwork as well as image, audio or video files that may be used to elicit speech

or even the documents (such as PDFs of handouts or pedagogical materials) that are in some way related to language data.

There are three types of file models and while each share a common core of metadata-related attributes, they have attributes unique to their type as well. *Local* files are stored on the filesystem (by default, in the `files/` directory) of the machine serving an OLD applicaton. *Subinterval-referencing* files get their file content from a local audio/video file (their `parentFile`) and have `start` and `end` attributes which reference start and end positions in the parent file. *Externally hosted* files have content stored on another server and have `url` attributes for locating that content. The form of the input passed with create requests will determine which type of file model is created. Whatever the type of file being created, the URL and HTTP method for such requests remains the same, i.e., `POST /files`.

When creating a *local* OLD file, it is necessary to upload a binary file to the OLD.[15] The traditional way of doing this in web applications is to specify the `Content-Type` of the HTTP request as `multipart/form-data` and pass the binary file data in the body of the request in a special format. When using this method, additional parameters are restricted to simple name-value pairs – hierarchical JSON objects are not permitted. Therefore, when one is using the `multipart/form-data` approach and when the file ought to be associated to multiple tag or form models, the parameter names should make use of the following convention: <attribute_name>-<index>. That is, to associate the tags with `id` values 2 and 36 to a file one is creating, the body of the request should contain a parameter named "tags-0" with a value of "2" and another parameter named "tags-1" with a value of "36". Similarly, associating a new file to multiple forms using the `multipart/form-data` approach will require parameter names like "forms-0", "forms-1", "forms-2", etc. When using this approach, at least the following set of parameters must be included.

Parameter name	Comments
filename	required
dateElicited	format mm/dd/yyyy
description	possibly empty string describing the file
elicitor	id of a valid elicitor model, or empty string
forms-0	id of a valid form model, or empty string
speaker	id of a valid speaker model, or empty string
tags-0	id of a valid tag model, or empty string
utteranceType	one of the allowed utterance types

The other way of creating a local OLD file is to set the `Content-Type` of the request to `application/json` and send all input as a JSON object, as is done with all other creation and update requests to an OLD web service. Under this approach, the binary file is converted to a string using Base64 encoding and that string is the value of the `base64EncodedFile` attribute of the JSON object passed in the request body. Because it is inefficient to Base64-encode large files on the client and then decode them in memory on the server, requests to `POST /files` with a request body that is greater than 20MB [16] will be rejected with a 400 error code. File creation requests for *local* files using the `application/json` content type must contain a JSON object of the following form.

```
{
    "base64EncodedFile": ""
    "dateElicited": "",
    "description": "",
    "elicitor": null, // valid user model id or null
    "filename": "",
    "forms": [], // array of valid form model ids or []
    "speaker": null, // valid speaker model id or null
    "tags": [], // array of valid tag model ids or []
    "utteranceType": "",
}
```

Note that once a local file model has been created the value of its `filename` attribute cannot be changed, nor can its file data. That is, requests to `PUT /files` should contain an object just like that presented above except

[15] Note that updates to a local file model/resource cannot alter the binary data of the file model. That is, if the wrong file is uploaded, it is necessary to delete the miscreated file and to create a new one with the correct file data.

[16] Technically, such requests will be rejected if the length of the request body (as a Python unicode object) is greater than 20971520.

that the `base64EncodedFile` and `filename` attributes ought to be removed as they will simply be ignored by the controller handling the request. In contrast, when requesting an update to an externally hosted or subinterval-referencing file, the input object may contain new values for all of the attribues permitted on create requests (see below).

Requests to create subinterval-referencing files are identified by the presence of a `parentFile` attribute in the request parameters. Creation requests for these types of files must contain a JSON object in the body of the request of the following form.

```
{
    "dateElicited": "",
    "description": "",
    "elicitor": null, // valid user model id or null
    "end": 4.7, // integer or float representing the end of the interval in seconds
    "filename": "",
    "forms": [], // array of valid form model ids or []
    "name": "",
    "parentFile": 1, // valid id of a local OLD audio/video file
    "speaker": null, // valid speaker model id or null
    "start": 3.5, // integer or float representing the start of the interval in seconds
    "tags": [], // array of valid tag model ids or []
    "utteranceType": "",
}
```

Requests to create externally hosted files are identified by the presence of a `url` attribute in the request parameters. Creation requests for these types of files must contain a JSON object in the body of the request of the following form.

```
{
    "dateElicited": "",
    "description": "",
    "elicitor": null, // valid user model id or null
    "filename": "",
    "forms": [], // array of valid form model ids or []
    "MIMEtype": "",
    "name": "",
    "parentFile": 1, // valid id of a local OLD file
    "password": "",
    "speaker": null, // valid speaker model id or null
    "tags": [], // array of valid tag model ids or []
    "url": "http://vimeo.com/13452",
    "utteranceType": "",
}
```

File representations returned by the OLD are JSON objects of the following form.

```
{
    "dateElicited": "",
    "datetimeEntered": "",
    "datetimeModified": "",
    "description": "",
    "elicitor": null, // integer id of a valid user model
    "end": null, // number or null
    "enterer": 1, // integer id of a valid user model
    "filename": "",
    "forms": [], // array of valid ids of form models
    "id": 1,
    "lossyFilename": "",
    "MIMEtype": "",
    "name": "",
```

```
    "parentFile": null,  // integer id of a valid (audio/video) file model
    "password": "",
    "size": null, // integer representing the size of the file in bytes
    "speaker": null, // integer id of a valid speaker model
    "start": null, // number or null
    "tags": [], // array of valid ids of tag models
    "url": "",
    "utteranceType": ""
}
```

dateElicited

The `dateElicited` attribute is a user-supplied date value which indicates the date when the file was elicited, if applicable, e.g., when a recording of an elicitation was made. The date must be in mm/dd/yyyy format.

datetimeEntered

The value of the `datetimeEntered` attribute is a UTC timestamp generated by the system when a file is created. Note that this value is distinct from the `datetimeModified` attribute that is common to all model types since that value is generated upon creation *and* update requests while the `datetimeEntered` value is only generated upon creation requests and is not altered thereafter.

description

The value of the `description` attribute is a user-supplied string that describes the file.

elicitor

The `elicitor` attribute references a valid user model who is the elicitor of the file, if applicable.

end

The value of the `end` attribute is a number (integer or float) representing the end of the subinterval in seconds of a subinterval-referencing file. For example, consider the subinterval-referencing file *F2* which references the audio file *F1* as its parent file. A value of 3.7 for the `end` attribute of *F1* means that the content of *F1* is a portion of the audio file of *F2* which ends at 3.7 seconds. Note that only subinterval-referencing files should have values for the `end` attribute.

enterer

The `enterer` attribute references the user model whose account was used to create the file. This value is generated automatically by the system upon file creation.

filename

The `filename` attribute holds the name of the file as it is stored in the filesystem. When a local file is created, a non-empty `filename` value must be provided in the input parameters. While unicode (i.e., non-ASCII) characters are permitted in the `filename` value, the system removes certain characters (QUOTATION MARK (”), APOSTROPHE (‘), the path separator (/ on Unix systems) and the null byte) and replaces spaces with underscores. If a file with the resulting name already exists in the directory that holds local file data (the `files/` directory by default), then

the system will alter the name (by inserting an underscore followed by a string of eight random characters between the end of the file name and its extension) until a unique one is found. The resulting string becomes the value of the `filename` attribute. So, for example, if a file create request contains "john's file.wav" as the value of the `filename` parameter and if `files/johns_file.wav` already exists, then the file data will be saved to something like `files/johns_file_3Df6Nop0.wav` and the value of the `filename` attribute of the file model will be "johns_file_3Df6Nop0.wav".

forms

A file model may be associated to zero or more forms. On file create and update requests, associated forms are specified by providing an array of valid form ids as the value of the `forms` attribute. When JSON object representations of file models are returned, the value of the `forms` attribute is an array of JSON objects representing the associated forms.

lossyFilename

If the OLD is configured to create reduced-size copies of uploaded files and if the requisite dependencies are installed (i.e., PIL or FFmpeg), then the system will create reduced-size (i.e., lossy) copies of the files in `files/reduced_files/` and the `lossyFilename` attribute will return the name of the reduced-size copy in that directory. For example, if in the config file `create_reduced_size_file_copies` is set to "1" and `preferred_lossy_audio_format` is set to "ogg" and if FFmpeg is installed, then a WAV file uploaded and saved to `files/my_file.wav` will have a lossy copy in `files/reduced_files/my_file.ogg` and the value of `lossyFilename` will be "my_file.ogg".

MIMEtype

MIMEtypes, also known as Internet Media Types, are standardized strings used to categorize types of binary files. An OLD web service will ascertain the MIMEtype of an uploaded file using the python-magic module and the contents of the file. If the MIMEtype is in the list of allowed MIMEtypes (as defined in `allowedFileTypes` of `lib/utils.py`), then the value of the `MIMEtype` attribute will be assigned to the ascertained MIMEtype string. The valid MIME/Internet Media types are listed in the table below.

Internet media type	Common extension(s)	Name
application/pdf	.pdf	Portable Document Format
image/gif	.gif	GIF image
image/jpeg	.jpg, jpeg	JPEG JFIF image
image/png	.png	Portable Network Graphics
audio/mpeg	.mp3	MP3 or other MPEG audio
audio/ogg	.ogg	Ogg Vorbis, Speex, Flac and other audio
audio/x-wav	.wav, .wave	WAV audio
video/mpeg	.mpeg	MPEG-1 video with multiplexed audio
video/mp4	.mp4	MP4 video
video/ogg	.ogg, .ogv	Ogg Theora or other video (with audio)
video/quicktime	.mov, .qt	QuickTime video
video/x-ms-wmv	.wmv	Windows Media Video

name

Externally hosted and subinterval-referencing files may supply a value for the `name` attribute. Since these types of files do not have values for the `filename` attribute, the `name` attribute can be useful in identifying them. For local files the system automatically sets the `name` attribute to the value of the `filename` attribute. If a subinterval-referencing

file creation request does not include a non-empty `name` value, then the value assigned to that attribute is the value of the `filename` attribute of the subinterval-referencing file's parent file.

parentFile

Subinterval-referencing files are identified by possession of a non-empty `parentFile` attribute. The value of this attribute is a reference to an existing local file. The parent file must be an audio or video file. The subinterval-referencing file gets its file data from its parent file.

password

The `password` attribute can be specified for externally hosted file models that require a password in order for the external host to serve the file. Note that this value will be available to all users of the system and should *not* therefore be a password used for other purposes, e.g., to log in to the OLD web service itself.

size

Local file models have a value for the `size` attribute which is an integer representing the size of the binary file in bytes. This is calculated upon a successful file creation request.

speaker

The `speaker` attribute references a valid speaker model who is the speaker or consultant of the file. This is appropriate in cases where the file is, say, an audio recording of a speaker telling a story or a recording of an elicitation session with a particular consultant.

start

The value of the `start` attribute is a number (integer or float) representing the beginning of the subinterval in seconds of a subinterval-referencing file. For example, consider the subinterval-referencing file *F2* which references the audio file *F1* as its parent file. A value of 2.1 for the `start` attribute of *F1* means that the content of *F1* is a portion of the audio file of *F2* begins at 2.1 seconds. Note that only subinterval-referencing files should have values for the `start` attribute.

tags

A file may be associated to zero or more tags. Tags are user-defined models that can be used to arbitrarily categorize other OLD models. If a file is to be restricted, then the special "restricted" tag should be associated to id. See the *Tag* section for more details on the tag model.

url

Externally hosted files are identified by possession of a non-empty value for the `url` attribute. The value should be a valid URL that will serve the content of the file when requested. This value will allow user-facing applications to display (i.e., embed) the file content of externally hosted file models.

utteranceType

Files that represent recordings of utterances should be categorized using the `utteranceType` attribute. Valid values, as defined in the `utteranceTypes` tuple of `lib/utils.py` are "None", "Object Language Utterance", "Metalanguage Utterance" and "Mixed Utterance". If the value of this attribute on input is an empty string or `null`, then its value will be `null`.

Here is a potential use case scenario for this attribute. Consider an OLD web service that is being used to study the Blackfoot language and imagine a file model *F1* whose binary data is a WAV file audio recording of a speaker saying "oki", which means "hello" in Blackfoot. Now imagine a second file, *F2* whose binary data is another WAV file recording of the speaker saying "hello". Assume that the `utteranceType` value of *F1* is "Object Language Utterance" (since it is a recording of an utterance of the object language, i.e., Blackfoot) and assume that the `utteranceType` value of *F2* is "Metalanguage Utterance" (since it is a recording of an utterance in the language of analysis and translation, i.e., English). Now imagine a form *F* whose transcription is "oki" and whose only translation is "hello" and which is associated to files *F1* and *F2*. If there are a good number of forms like *F*, then an application making use of this OLD web service would be able to reasonably assume that *F1*, being an object language utterance associated to *F* is a recording of a speaker uttering the linguistic form that is transcribed in *F*. Such an application could then use such forms to automatically generate audio/textual language learning games or talking dictionaries.

2.3.6 Form

An OLD form model represents a linguistic form in a very general sense; that is, it can represent a lexical item abstracted from any elicitation or recording event as well as a word, phrase or sentence uttered on a particular occasion by a particular speaker.

Form creation and update requests must contain a JSON object of the following form.

```
{
    "comments": "",
    "dateElicited": "" // string of the form mm/dd/yyyy
    "elicitationMethod": null, // valid elicitation method model id or null
    "elicitor": null, // valid user model id or null
    "files": [], // array of valid file model ids or []
    "translations": [{"transcription": "hello", "grammaticality": ""}],
    "grammaticality": "",
    "morphemeBreak": "",
    "morphemeGloss": "",
    "narrowPhoneticTranscription": "",
    "phoneticTranscription": "",
    "source": null, // valid source model id or null
    "speaker": null, // valid speaker model id or null
    "speakerComments": "",
    "status": "",
    "syntacticCategory": null, // valid syntactic category model id or null
    "tags": [], // array of valid tag model ids or []
    "transcription": "oki",
    "verifier": null // valid user model id or null
}
```

Forms representations returned by the OLD are JSON objects of the following form.

```
{
    "breakGlossCategory": "",
    "comments": "",
    "dateElicited": "",
    "datetimeEntered": "", // system-generated ISO 8601-formatted datetime
    "datetimeModified": "", // system-generated ISO 8601-formatted datetime
```

```
    "elicitationMethod": null, // an object representation of an elicitation method or null
    "elicitor": null, // an object representation of a user or null
    "enterer": { ... }, // an object representation of a user
    "files": [], // an array of object representations of files or []
    "translations": [{...}], // an array of object representations of translations
    "grammaticality": "",
    "id": 1, // the integer id assigned by the database
    "morphemeBreak": "",
    "morphemeBreakIDs": null, // an array or null
    "morphemeGloss": "",
    "morphemeGlossIDs": null, // an array or null
    "narrowPhoneticTranscription": "",
    "phoneticTranscription": "",
    "source": null, // an object representation of a source or null
    "speakerComments": "",
    "speaker": null, // an object representation of a speaker or null
    "status": "",
    "syntacticCategory": null, // an object representation of a syntactic category or null
    "syntacticCategoryString": "",
    "tags": [], // an array of object representations of tags or []
    "transcription": "bonjour",
    "UUID": "1025b514-5781-4dce-8715-8c2590119546", // generated by the system
    "verifier": null, // an object representation of a user or null
}
```

breakGlossCategory

The `breakGlossCategory` attribute stores a system-generated string which merges the values of the `morphemeBreak`, `morphemeGloss` and `syntacticCategoryString` attributes. For example, the `breakGlossCategory` value of a form with "chien-s" as its morpheme segmentation, "dog-PL" as its morpheme gloss string and "N-Num" as its syntactic category would be "chien|dog|N-s|PL|Num". Since the `breakGlossCategory` value is searchable, it can be used to filter forms according to presence/absence of a specific morpheme. See the *Morphological processing* section for details on the structure of this value and its method of generation.

collections

A form may be associated to zero or more collections. Collections are documents that typically reference, and are associated to, multiple forms. Note that such associations are *not* created during form creation or updating but during collection creation. See the *Collection* section for details.

comments

The `comments` attribute is an open-ended field that may contain any comments about the form or any data that do not fit neatly into the standard attributes of the form resource. If multiple forms are to be tagged or classified in some way, it is better to use the `tags` attribute for this purpose and not the `comments` attribute.

dateElicited

The `dateElicited` attribute is a user-supplied date value which indicates the date when the form was elicited. The date must be in mm/dd/yyyy format. For abstract lexical forms this value may not be appropriate.

datetimeEntered

The value of the `datetimeEntered` attribute is a UTC timestamp generated by the system when a form is created. Note that this value is distinct from the `datetimeModified` attribute that is common to all model types since that value is generated upon creation *and* update requests while the `datetimeEntered` value is only generated upon creation requests and is not altered thereafter.

elicitationMethod

The `elicitationMethod` attribute references a valid elicitation method model that classifies the way in which the form was elicited. See the *ElicitationMethod* section for details.

elicitor

The `elicitor` attribute references a valid user model who is the elicitor of the form.

enterer

The `enterer` attribute references the user model whose account was used to enter the form. This value is generated automatically by the system upon form creation.

files

A form may be associated to zero or more files via the `files` attribute which references a collection of file models. Files are OLD objects that represent a binary file (e.g., an audio, video or image file) along with metadata (e.g., a description or the size of the file). See the *File* section for details on the structure of file models. To associate a form to files upon form create/update requests, pass an array of valid file ids as the value of the `files` attribute of the input object. When a form is output by an OLD application, the value of the `files` attribute of the output object will be an array containing JSON object representations of any associated file models.

translations

A form model must have at least one translation but may have more. The translations of a form are each translation model objects that are listed in the `translations` attribute of the form. (In the relational database schema, the `form` and `translation` tables are in a one-to-many relationship.) Forms with multiple translations, e.g., sentences with multiple valid translations, should use separate translation models for each such translation. Translation models can also have grammaticalities (cf. the `grammaticality` attribute) – this feature may be used to indicate a translation that is not appropriate to a grammatical form. Thus, as a simplistic example, “chien” may be translationed as “dog” and “*wolf” using two translation models.

grammaticality

The `grammaticality` attribute stores the grammaticality value assigned to the form. This is a forced-choice attribute whose options are defined by the users of the system in the `grammaticalities` attribute of the active application settings resource. Usually, the available grammaticalities will be a list such as “*”, ”?”, “#”, “**”, etc.

memorizers

The memorizers attribute holds a collection of zero or more user models corresponding to the users who have memorized, or remembered, this form. See the section on the remembered forms resource (*Remembered forms*) for details on how memorize a form.

morphemeBreak

The morphemeBreak attribute holds a representation of the morphological analysis of a linguistic form, i.e., a morphemic segmentation. Maximum length is 255 characters. The system will expect words to be split by whitespace and morphemes by the delimiters specified in the morphemeDelimiters attribute of the active application settings. By specifying appropriate values for the morphemeBreakValidation, morphemeBreakIsOrthographic and phonemicInventory or storageOrthography attributes of the active application settings resource, it is possible to ensure that data input to this attribute are validated against the specified orthography/inventory and delimiters.

morphemeBreakIDs

The value of the morphemeBreakIDs attribute is a system-generated JSON array that contains references to all matches found for each morpheme listed in the morphemeBreak attribute. See the *Morphological processing* section for details on the structure of this value and its method of generation.

morphemeGloss

The morphemeGloss attribute holds a string of morpheme glosses corresponding to the phonemic representations stored in the morphemeBreak field. Maximum length is 255 characters. As with the morphemeBreak field, the gloss "words" in this field should be delimited using whitespace and the glosses within words should be delimited using the specified morpheme delimiters.

morphemeGlossIDs

The value of the morphemeGlossIDs attribute is a system-generated JSON array that contains references to all matches found for each morpheme gloss listed in the morphemeGloss attribute. See the *Morphological processing* section for details on the structure of this value and its method of generation.

narrowPhoneticTranscription

The narrowhoneticTranscription attribute holds a narrow phonetic transcription of the linguistic form. Maximum length is 255 characters. By specifying a value for the narrowPhoneticInventory attribute of the active application settings and setting that same resource's narrowPhoneticValidation attribute to "Error", it is possible to configure narrowhoneticTranscription validation so that values not generable using the specified inventory are rejected. See *Object language validation*.

phoneticTranscription

The phoneticTranscription attribute holds a phonetic transcription of the linguistic form. By convention, this is a *broad* phonetic transcription. Maximum length is 255 characters. By specifying a value for the broadPhoneticInventory attribute of the active application settings and setting that same resource's broadPhoneticValidation attribute to "Error", it is possible to configure phoneticTranscription validation so that values not generable using the specified inventory are rejected. See *Object language validation*.

semantics

The value of the `semantics` attribute is canonically a semantic representation of the form, e.g., a denotation. Maximum length is 1023 characters. At some future point candidate values for this attribute may be auto-generated.

source

The `source` attribute references a valid source model that indicates the textual (or other) source of the form. This is useful for when data are taken from papers or dictionaries and need to be attributed. The source model is based on the BibTeX format. See the *Source* section for details.

speaker

The `speaker` attribute references a valid speaker model who is the speaker or consultant of the form.

speakerComments

The `speakerComments` attribute holds comments made about the form by the speaker or consultant.

status

The `status` attribute encodes the status of the form with respect to its verification. At present, the two licit values are "tested" and "requires testing". Usage of this attribute permits researchers to enter forms not yet tested in order to prepare for a planned elicitation session.

syntacticCategory

The `syntacticCategory` attribute references a valid syntactic category model that categorizes the form. For example, a form like "chien" might have a `syntacticCategory` value which references a syntactic category model whose `name` attribute is "N". See the *SyntacticCategory* section for details.

syntacticCategoryString

The `syntacticCategoryString` attribute holds a system-generated value which is a string of syntactic category names corresponding to the morphemes specified by the creator/updater of the form. That is, the system inspects the values of the `morhemeBreak` and `morphemeGloss` fields and searches the database for matches to the specified morpheme/gloss pairs; the names of the syntactic categories of the matches are used to generate the value for the `syntacticCategoryString` attribute. By searching forms based on patterns in this field it is possible to filter the database according to higher-level morphological or syntactic patterns. See the *Morphological processing* section for further details on how this value is generated.

syntax

The value of the `syntax` attribute is canonically a syntactic representation of the form, e.g., a phrase structure tree in bracket notation. Maximum length is 1023 characters. At some future point candidate values for this attribute may be auto-generated.

tags

A form may be associated to zero or more tags. Tags are user-defined models that can be used to arbitrarily categorize other OLD models. An example usage would be to define a tag model with a `name` value of "VP ellipsis" and use that tag to categorize forms that exhibit the phenomenon. If a form is to be restricted, then the special "restricted" tag should be associated to id; similarly, if the form documents a foreign word, then it should be associated to the special "foreign word" tag. See the *Tag* section for more details on the tag model.

transcription

The `transcription` attribute holds transcriptions of linguistic forms. By convention, these are expected to be written in an orthography of the object language. Maximum length is 255 characters. Every form must have a transcription. It is possible to specify a storage orthography in the active application settings resource and configure form transcription validation so that values not generable using the orthography are rejected. See *Object language validation* for details.

UUID

The value of the `UUID` attribute is a universally unique identifier (UUID), i.e., a number represented by 32 hexadecimal digits displayed in five groups using four hyphens. A valid UUID is a 36-character string that looks like `aba3ea8d-b56f-4934-a8f7-68cba500f411`. The forms controller randomly generates a UUID value for each newly created form model. These values are used to associate form backups to the forms they backup.

verifier

The `verifier` attribute references a valid user model who has verified the form. This is useful, for example, in a case where one researcher finds that a form they have elicited has already been stored in the database and they do not want to record a duplicate entry. Oftentimes, however, it is desirable to enter a duplicate entry.

2.3.7 FormBackup

A form backup model is created whenever a form model is updated or deleted. These models cannot be created directly, i.e., `POST /formbackups` is not a valid request. The form backup model receives all of the attributes of the model that it backs up. It also has some additional attributes, viz. `form_id` and `backuper`. The value of the `form_id` attribute is the value of the `id` attribute of the form that was backed up to create the present form backup model. The value of the `backuper` attribute is a JSON object representing the user who created the backup (by deleting or updating the form). In general, the values of the relational attributes of the form (i.e., the attributes that refer to other models) are converted to JSON object representations in the form backup model. For example, the value of the `speaker` attribute is such a JSON object and the value of the `files` attribute is a JSON array of such objects representing file models. If the form has just been deleted, then the value of the `datetimeModified` value of the form backup will be the UTC datetime at which the backup occurred.

Form backup representations returned by the OLD are JSON objects of the following form.

```
{
    "backuper": null, // an object representation of an elicitation method or null
    "breakGlossCategory": "",
    "comments": "",
    "dateElicited": "",
    "datetimeEntered": "",
    "datetimeModified": "",
    "elicitationMethod": null, // an object representation of an elicitation method or null
```

```
    "elicitor": null, // an object representation of an elicitation method or null
    "enterer": null, // an object representation of an elicitation method or null
    "files": [], // an array of objects representing file models or []
    "form_id": 1,
    "translations": [], // an array of objects representing translation models or []
    "grammaticality": "",
    "id": 1,
    "morphemeBreak": "",
    "morphemeBreakIDs": null, // an array or null
    "morphemeGloss": "",
    "morphemeGlossIDs": null, // an array or null
    "narrowPhoneticTranscription": "",
    "phoneticTranscription": "",
    "source": null, // an object representation of an elicitation method or null
    "speaker": null, // an object representation of an elicitation method or null
    "speakerComments": "",
    "syntacticCategory": null, // an object representation of an elicitation method or null
    "syntacticCategoryString": ""
    "tags": [], // an array of objects representing tag models or []
    "transcription": "",
    "UUID": "",
    "verifier": null, // an object representation of an elicitation method or null
}
```

2.3.8 FormSearch

The form search model stores searches on form resources so that these searches can be saved for later use and shared with other users of the system.

Requests to create or update application settings resources must contain a JSON object of the following form.

```
{
    "description": u"",
    "name": u"returns all transitive verbs", // obligatory string
    "search": {...}, // an object representing an OLD form query
}
```

Form search representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "description": "",
    "id": 1,
    "name": "returns all transitive verbs",
    "search": { ... }, // an object representing an OLD form query
    "searcher": { ... } // object representation of a user model
}
```

description

The value of the `description` attribute is a user-supplied string that describes the search resource.

name

The value of the `name` attribute is a user-supplied string used to identify the search resource. Names are obligatory, may not exceed 255 characters and no two searches may have the same name.

search

The value of the `search` attribute is the JSON object representing the search. If the user-supplied search object is not well-formed, the system will prevent the form search resource from being created or updated. The search object is an object with an obligatory `filter` attribute and an optional `orderBy` attribute (see below). The values of both of these attributes are arrays. The definitions of what constitutes well-formed "filter" and "orderBy" arrays are provided in the *Search* section.

```
{
    "filter": [ ... ],
    "orderBy": [ ... ]
}
```

searcher

The `searcher` attribute references the user model whose account was used to create the form search. This value is generated automatically by the system upon form search creation.

2.3.9 Translation

Translations are translations of forms into the metalanguage. A form model can have multiple translations and each of these translations is a translation model. Each translation model has `transcription` and `grammaticality` attributes. In relational database terminology, the form and translation tables are in a one-to-many relationship; that is, a form may have many translations but each translation has one and only one form. When a form is deleted, so too are its translations.

Translations are created not directly (i.e., there is no "translations" resource) but upon form create and update requests. The input JSON object of such requests has a `translations` attribute whose value is an array of objects with `transcription` and `grammaticality` attributes, e.g.,

```
{
    "translations": [
        {"transcription": "dog", "grammaticality": ""},
        {"transcription": "wolf", "grammaticality": "*"}
    ]
}
```

2.3.10 Language

Each language model represents a language in the ISO 639-3 standard. These models are created in the database when `paster setup-app` is run during the initial set up of the application. The data are taken from the tab-delimited text file `public/iso_639_3_languages_data/iso_639_3.tab`. Existing language models cannot be updated and new ones cannot be created. The purpose of this resource is to provide options for the metalanguage and object language id and name attributes of application settings resources.

The language models are unique among OLD models in lacking an `id` attribute. Instead they have `Id` attributes whose values are the unique three-character strings used to identify the language. The other attribute of note is the `Ref_Name` attribute whose value is the reference name of the language. The standard makes it clear that no special importance

should be given to the reference name; OLD administrators are encouraged to use whatever language names seem most appropriate, despite what the value of `Ref_Name` may be. However, care should be taken to attempt to identify the correct `Id` value for the language being documented via an OLD web service so that this information is unambiguous.

For completeness, the attributes of language models are listed here: `Id`, `Part2B`, `Part2T`, `Part1`, `Scope`, `Type`, `Ref_Name`, `Comment`, `datetimeModified`. See http://www-01.sil.org/iso639-3/download.asp for the semantics of these attributes.

2.3.11 `Orthography`

An orthography model is a representation of the graphemes used in a particular writing system. The OLD makes use of orthography models in order to effect input validation on the `transcription` and `morphemeBreak` attributes of form models. Previous versions of the OLD implemented orthography conversion functionality server-side, thus allowing users to enter transcriptions in one orthography and have it converted to a string in another (storage) orthography. However, this functionality will now be the responsibility of any user-facing applications that make use of OLD web services.

Requests to create or update orthography resources must contain a JSON object of the following form.

```
{
    "initialGlottalStops": true
    "lowercase": false,
    "name": "Standard Orthography",
    "orthography": "p, t, k, n, s, i, o, a",
}
```

Orthography representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "id": 1,
    "initialGlottalStops": true,
    "lowercase": false,
    "name": "",
    "orthography": ""
}
```

`initialGlottalStops`

The value of the `initialGlottalStops` is a boolean with `True` as the default. The user-supplied input may be a truthy string (i.e., “true”, “on”, “yes” or “1”), JSON `true`, a falsey string (i.e., “false”, “off”, “no” or “0”) or JSON `false`. This attribute encodes whether the orthography marks glottal stops at the beginning of words and can be useful for orthography conversion algorithms.

`lowercase`

The value of the `lowercase` is a boolean with `False` as the default. The user-supplied input may be a truthy string (i.e., “true”, “on”, “yes” or “1”), JSON `true`, a falsey string (i.e., “false”, “off”, “no” or “0”) or JSON `false`. This attribute encodes whether the orthography uses only lowercase characters and can be useful for orthography conversion algorithms and for reducing the number of graphemes that must be specified in the `orthography` attribute.

name

The `name` attribute holds a name for the orthography. The name must be unique among orthography names and may not exceed 255 characters. The name should facilitate identification of the orthography.

orthography

The value of the `orthography` attribute is a comma-delimited list of strings representing the graphemes of the orthography. A non-empty value for this attribute is required.

Previous versions of the OLD drew significance from the ordering of the graphemes (i.e., for sorting & alphabetization) and also encouraged bracketing of graphemes into equivalence classes for the purpose of sorting (i.e., "a" and "a" would be sorted equivalently if the orthography contained "..., [a, a], ..."). The OLD web service now leaves orthography conversion to the user-facing applications; therefore, additional conventions for orthography specification (such as the significance of ordering and equivalence bracketing) should be detailed in the documentation of those applications.

As described in the *Object language validation* and *ApplicationSettings* sections, orthography models and, in particular, the values of their `orthography` attributes are used in input transcription validation.

2.3.12 Page

A page model can be used to allow users to create web pages using a specified markup language. Some of the attributes (e.g., `heading` or `name`) may be removed or renamed in future versions of the OLD.

Requests to create or update page resources must contain a JSON object of the following form.

```
{
    "content": u"",
    "heading": u"",
    "markupLanguage": u"",
    "name": u""
}
```

Page representations returned by the OLD are JSON objects of the following form.

```
{
    "content": "",
    "datetimeModified": "",
    "heading": "",
    "html": "",
    "id": 1,
    "markupLanguage": "",
    "name": ""
}
```

content

The `content` attribute holds a string representing the content of the page written in the specified markup language.

heading

The value of the `heading` attribute is a user-supplied string, no longer than 255 characters, which could be used as a heading or title for the page.

html

The value of the `html` attribute is the HTML generated from the user-supplied `content` value using the markup-to-HTML function corresponding to the specified markup language.

markupLanguage

The value of the `markupLanguage` attribute is one of "Markdown" or "reStructuredText" as defined in the `markupLanguages` variable of `lib/utils.py`. Markdown and reStructuredText are *lightweight markup languages*. A lightweight markup language is a markup language (i.e., a system for annotating a document) that is designed to be easy to read in its raw form. The system will expect the value of the `content` attribute to contain markup in the specified markup language and will choose a markup-to-HTML function corresponding to that markup language when generating the HTML of the page. If no value is specified, "reStructuredText" will be the default.

name

The value of the `name` attribute is a string used to identify the page. This value may not exceed 255 characters and a non-empty value must be provided.

2.3.13 Phonology

OLD phonology models are representations of a phonology for the object language. That is, they specify the relationship between underlying representations (e.g., the value of the `morphemeBreak` attribute) and surface representations (e.g., the value of the `transcription`, `phoneticTranscription` or `narrowPhoneticTranscription` attributes) of form models.

The intention is to use the user-specified phonologies to compile finite-state transducer implementations of the phonologies and to use these transducers in the construction of morphological parsers and in functionality that compares surface strings and underlying strings and informs users of incompatibilities. At present this functionality is not yet implemented in the OLD.

Requests to create or update phonology resources must contain a JSON object of the following form.

```
{
    "description": "",
    "name": "",
    "script": ""
}
```

Phonology representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeEntered": "",
    "datetimeModified": "",
    "description": "",
    "enterer": { ... }, // object representation of a user
    "id": 1,
    "modifier": null, // object representation of a user or null
    "name": "",
    "script": "",
}
```

datetimeEntered

The value of the `datetimeEntered` attribute is a UTC timestamp generated by the system when a phonology is created. Note that this value is distinct from the `datetimeModified` attribute that is common to all model types since that value is generated upon creation *and* update requests while the `datetimeEntered` value is only generated upon creation requests and is not altered thereafter.

description

The value of the `description` attribute is an open-ended, user-supplied description of the phonology.

enterer

The `enterer` attribute references the user model whose account was used to create the phonology. This value is generated automatically by the system upon phonology creation.

modifier

The `modifier` attribute references the user model whose account was used to perform the most recent update on the phonology. This value is generated automatically by the system upon successfuly phonology update requests.

name

The value of the obligatory `name` attribute is a unique string, not to exceed 255 characters, that identifies the phonology.

script

The `script` attribute holds a user-supplied string constituting the rules or specification of the phonology. The intention is for the OLD to make use of the FST compiler package called Foma. When this is implemented, the OLD will expect the `script` value to contain a valid Foma script and will attempt to compile it, returning an error on create/update requests if the compile attempt fails.

2.3.14 Source

Sources are references to texts that can be cited in the `source` attribute of form and collection models. The source schema is that of the BibTeX file format. The OLD validates input to source create and update requests in adherence to the BibTeX format. That is, a source of a given type (i.e., a BibTeX entry type) must have values for all of the required attributes of that type. For example, a source with a `type` value of "article" must have values for its `author`, `title`, `journal` and `year` attributes.

OLD source models have attributes corresponding to all of the standard BibTeX field names as well as attributes corresponding to some non-standard ones. The full list of source attributes is given below. In general, the source attribute names match their BibTeX field name counterparts exactly. The exceptions to this are the `key`, `keyField`, `type` and `typeField` attributes which correspond to BibTex key, "key" field name, entry type and "type" field name, respectively. See the relevant subsections below for details.

Like all other OLD models, sources have `id` and `datetimeModified` attributes. Source models also have a `file` attribute for referencing an OLD file model.

At some point, the OLD may specify a syntax for citing source models within the value of the `contents` attribute of collection models.

Requests to create or update source resources must contain a JSON object of the following form. Source representations returned by the OLD are JSON objects of the same form, with the addition of `id`, `datetimeModified` and `crossrefSource` attributes. The value of the `crossrefSource` attribute is either `null` (if no `crossref` value was supplied by the user) or a JSON object representing the cross-referenced source.

```
{
    "abstract": "",
    "address": "",
    "affiliation": "",
    "annote": "",
    "author": "",
    "booktitle": "",
    "chapter": "",
    "contents": "",
    "copyright": "",
    "crossref": "",
    "edition": "",
    "editor": "",
    "file": null, // valid file model id or null on input; object on output
    "howpublished": "",
    "institution": "",
    "ISBN": "",
    "ISSN": "",
    "journal": "",
    "key": "chomsky67",
    "keyField": "",
    "keywords": "",
    "language": "",
    "location": "",
    "LCCN": "",
    "month": "",
    "mrnumber": "",
    "note": "",
    "number": "",
    "organization": "",
    "pages": "",
    "price": "",
    "publisher": "",
    "school": "",
    "series": "",
    "size": "",
    "title": "",
    "type": "book",
    "typeField": "",
    "url": "",
    "volume": "",
    "year": ""
}
```

The descriptions of the BibTeX field names given in the subsections below are taken, with some modifications, from Kopka.2004. The restrictions on lengths of attribute values are imposed (somewhat arbitrarily) by the OLD and are not part of the BibTeX format.

abstract

An abstract of the work. Maximum length is 1000 characters.

address

Usually the address of the publisher or other type of institution. For major publishing houses, it is recommended that this information be omitted entirely. For small publishers, on the other hand, you can help the reader by giving the complete address. Maximum length is 1000 characters.

affiliation

The author's affiliation. Maximum length is 255 characters.

annote

An annotation. It is not used by the standard bibliography styles, but may be used by others that produce an annotated bibliography.

author

The name(s) of the author(s), in the format described in Kopka.2004. There are two basic formats: (1) *Given Names Surname* and (2) *Surname, Given Names*. For multiple authors, use the formats just specified and separated each such formatted name by the word "and". Maximum length is 255 characters.

booktitle

Title of a book, part of which is being cited. See Kopka.2004 for details on how to type titles. For book entries, use the title field instead. Maximum length is 255 characters.

chapter

A chapter (or section or whatever) number. Maximum length is 255 characters.

contents

A table of contents. Maximum length is 255 characters.

copyright

Copyright information. Maximum length is 255 characters.

crossref

The `key` value of another source to be cross-referenced. Any attribute values that are missing from the source model are inherited from the source cross-referenced via the `crossref` attribute. Maximum length is 1000 characters.

If a valid `key` value is supplied as the value of the `crossref` attribute, the system will use the attributes of the cross-referenced source when validating the input. That is, a source whose `type` value is, for example, "inproceedings" would normally fail validation if it lacks a value for its `booktitle` attribute; however, if it cross-references another source whose `type` value is "proceedings" and which has a content-ful `booktitle` value, then it will pass validaton. If a valid `crossref` value is passed on input, then, on output, the value of `crossrefSource` will be an object representing the cross-referenced source.

crossrefSource

The value of the `crossrefSource` attribute is either `null` or the source model that is cross-referenced via the `crossref` attribute. That is, a valid `crossref` value passed on input will cause the system to set the cross-referenced source as the value of the `crossrefSource` attribute. When returning a JSON representation of the original source, the value of the `crossrefSource` attribute will be a JSON object representing the cross-referenced source.

edition

The edition of a book – for example, "Second". This should be an ordinal, and should have the first letter capitalized, as shown here; the standard styles convert to lower case when necessary. Maximum length is 255 characters.

editor

Name(s) of editor(s), typed as indicated in Kopka.2004. At its most basic, this means either as *Given Names Surname* or *Surname, Given Names* and using "and" to separate multiple editor names. If there is also a value for the `author` attribute, then the `editor` attribute gives the editor of the book or collection in which the reference appears. Maximum length is 255 characters.

file

Source models may reference an OLD file model object via the `file` attribute, thus permitting the association to a source of a document containing the source text itself. Note that the `file` attribute does not correspond to a standard BibTeX field name.

howpublished

How something strange has been published. The first word should be capitalized. Maximum length is 255 characters.

institution

The sponsoring institution of a technical report. Maximum length is 255 characters.

ISBN

The International Standard Book Number. Maximum length is 20 characters.

ISSN

The International Standard Serial Number. Used to identify a journal. Maximum length is 20 characters.

journal

A journal name. Abbreviations are provided for many journals. Maximum length is 255 characters.

key

The OLD source `key` field is the BibTeX key, i.e., the unique string used to unambiguously identify a source. Usually some type of convention is established for creating `key` values, e.g., the first author's last name in lowercase followed by the year of publication: "chomsky57". Maximum length is 1000 characters. All sources must have a valid `key` value and this value must be unique among source `key` values. A valid `key` value is any combination of ASCII letters, numerals and symbols (except the comma).

keyField

Used for alphabetizing, cross referencing, and creating a label when the `author` information is missing. This field should not be confused with the source's `key` attribute. Maximum length is 255 characters.

keywords

Key words used for searching or possibly for annotation. Maximum length is 255 characters.

language

The language the document is in. Maximum length is 255 characters.

location

A location associated with the entry, such as the city in which a conference took place. Maximum length is 255 characters.

LCCN

The Library of Congress Call Number. Maximum length is 20 characters.

month

The month in which the work was published or, for an unpublished work, in which it was written. Maximum length is 100 characters.

mrnumber

The Mathematical Reviews number. Maximum length is 25 characters.

note

Any additional information that can help the reader. The first word should be capitalized. Maximum length is 1000 characters.

number

The number of a journal, magazine, technical report, or of a work in a series. An issue of a journal or magazine is usually identified by its volume and number; the organization that issues a technical report usually gives it a number; and sometimes books are given numbers in a named series. Maximum length is 100 characters.

organization

The organization that sponsors a conference or that publishes a manual. Maximum length is 255 characters.

pages

One or more page numbers or range of numbers, such as 42–111 or 7,41,73–97 or 43+ (the "+" in this last example indicates pages following that don't form a simple range). Maximum length is 100 characters.

price

The price of the document. Maximum length is 100 characters.

publisher

The publisher's name. Maximum length is 255 characters.

school

The name of the school where a thesis was written. Maximum length is 255 characters.

series

The name of a series or set of books. When citing an entire book, the `title` attribute gives its title and an optional `series` attribute gives the name of a series or multi-volume set in which the book is published. Maximum length is 255 characters.

size

The physical dimensions of a work. Maximum length is 255 characters.

title

The work's title, typed as explained in the Kopka.2004. Maximum length is 255 characters.

type

The value of the OLD source `type` attribute is the BibTeX entry type, e.g., "article", "book", etc. The valid entry types and their required fields are specified as the keys of the `entryTypes` dictionary in `lib/bibtex.py`. A valid `type` value is obligatory for all source models. The chosen `type` value will determine which other attributes must also possess non-empty values, cf. the table below.

type	required attributes
article	author, title, journal, year
book	author or editor, title, publisher, year
booklet	title
conference	author, title, booktitle, year
inbook	author or editor, title, chapter or pages, publisher, year
incollection	author, title, booktitle, publisher, year
inproceedings	author, title, booktitle, year
manual	title
mastersthesis	author, title, school, year
misc	
phdthesis	author, title, school, year
proceedings	title, year
techreport	author, title, institution, year
unpublished	author, title, note

typeField

The type of a technical report—for example, "Research Note". Maximum length is 255 characters.

url

The universal resource locator for online documents; this is not standard but supplied by more modern bibliography styles. Maximum length is 1000 characters.

volume

The volume of a journal or multi-volume book. Maximum length is 100 characters.

year

The year of publication or, for an unpublished work, the year it was written. Generally it should consist of four numerals, such as 1984.

2.3.15 Speaker

An OLD speaker model represents a speaker or consultant who is the source of a linguistic form or collection thereof or who is the speaker on a recording.

Requests to create or update speaker resources must contain a JSON object of the following form.

```
{
    "dialect": "",
    "firstName": "John",
    "lastName": "Doe",
    "markupLanguage": ""
    "pageContent": ""
}
```

Speaker representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "dialect": "",
    "firstName": "",
    "html": "",
    "id": 1,
    "lastName": "",
    "markupLanguage": "",
    "pageContent": ""
}
```

dialect

The value of the `dialect` attribute is a string denoting the dialect of the speaker. The value may not exceed 255 characters.

Note that for abstract lexical forms, where it does not make sense to specify a speaker, dialects can be specified via tags – perhaps with a special syntax to facilitate search, e.g., "dialect:dialect_name".

firstName

The `firstName` attribute holds the first name of the speaker. A value is obligatory and cannot exceed 255 characters.

html

The value of the `html` attribute is a string of HTML that is generated by the system using the value of the `pageContent` attribute and the markup language specified in the `markupLanguage` attribute.

lastName

The `lastName` attribute holds the last name of the speaker. A value is obligatory and cannot exceed 255 characters.

markupLanguage

The value of the `markupLanguage` attribute is one of "Markdown" or "reStructuredText" as defined in the `markupLanguages` variable of `lib/utils.py`. Markdown and reStructuredText are *lightweight markup languages*. A lightweight markup language is a markup language (i.e., a system for annotating a document) that is designed to be easy to read in its raw form. This value determines which markup-to-HTML function is employed when the system attempts to generate the `html` value from the user-supplied `pageContent` value. If no value is specified, "reStructuredText" will be the default.

pageContent

The value of the `pageContent` attribute is a string that can be used to construct a web page for the speaker. Future versions of the OLD will probably include `markupLanguage` and `html` attributes so that speaker creators can specify a markup language that the system can use to generate and cache the HTML.

2.3.16 SyntacticCategory

Syntactic category models are used to categorize form models into morphological or syntactic classes.

Requests to create or update syntactic category resources must contain a JSON object of the following form.

```
{
    "description": "",
    "name": "",
    "type": ""
}
```

Syntactic category representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "description": "",
    "id": "",
    "name": "",
    "type": ""
}
```

description

The value of the `description` attribute can be used to describe the category and/or clarify its intended usage.

name

The `name` attribute holds the name of the category. Example names might be "N", "S", "Agr", "VP", "V"', "Noun", "Sentence", "CP", etc. A non-empty value for this attribute is obligatory, must be unique among other syntactic category `name` values and may not exceed 255 characters.

type

Syntactic categories are themselves categorized via the `type` attribute. Valid values, as defined in the `syntacticCategoryTypes` tuple of `lib/utils.py` are "lexical", "phrasal" and "sentential". An input value of `null` or the empty string will result in `null` as value. The purpose of this attribute is to help the system to better understand the categorization. This categorization could be useful for functionality that, say, seeks to induce a grammar of the morphology of the language. The available syntactic category types may change in future versions of the OLD.

2.3.17 Tag

Tags are general-purpose, user-defined models that can be associated to forms, files and collections. Any form, file or collection may have zero or more tags associated to it. Example usage of a tag would be to create tags for linguistic phenomena relevant to ones research; searches could then make reference to the presence or absence of this tag.

There are two special tags that are identified by their `name` values; these are the "restricted" and "foreign word" tags. These tags cannot be deleted via the interface (and should not be forcefully deleted by administrators using the RDBMS as this may have unintended consequences). The usage of the restricted and foreign word tags are described in the *Authentication & authorization* and *Object language validation* sections, respectively.

Requests to create or update tag resources must contain a JSON object of the following form.

```
{
    "description": "",
    "name": ""
}
```

Tag representations returned by the OLD are JSON objects of the following form.

```
{
    "datetimeModified": "",
    "description": "",
    "id": "",
    "name": ""
}
```

description

The value of the `description` attribute can be used to describe the tag and/or clarify its intended usage.

name

The `name` attribute holds the name of the tag. Example names might be "VP ellipsis", "double object" or "needs verification". A non-empty value for this attribute is obligatory, must be unique among other tag `name` values and may not exceed 255 characters.

2.3.18 User

User models represent the authorized users of an OLD web service. Authenticating to an OLD web service means supplying values for `username` and `password` attributes that match those of an existing user model. Only users with a `role` value of "administrator" are authorized to create new users. An authenticated user is permitted to update her own user model; however, only administrators can change the value of the `username` attribute.

Requests to create or update user resources must contain a JSON object of the following form. Note that on update, setting the values of the `username` and `password` attributes to `null` will cause the system to leave those values unchanged.

```
{
    "affiliation": "",
    "email": "",
    "firstName": "",
    "inputOrthography": null,
    "lastName": "",
    "markupLanguage": "",
    "outputOrthography": null
    "pageContent": "",
    "password": "",
    "password_confirm": "",
    "role": "",
    "username": "",
}
```

User representations returned by the OLD are JSON objects of the following form. Note that the `password` attribute is never present and that the `username` attribute is present only in the return value of DELETE, POST and PUT requests.

```
{
    "affiliation": "",
    "datetimeModified": "",
    "email": "",
    "firstName": "",
    "html": "",
    "id": 1,
    "inputOrthography": null, // object representation of an orthography model or null
    "lastName": "",
    "markupLanguage": "",
    "outputOrthography": null, // object representation of an orthography model or null
    "pageContent": "",
    "role": "",
    "username": ""
}
```

affiliation

The value of the `affiliation` attribute is a string representing the school or institution with which the user is affiliated. A value here is optional. Maximum allowable length is 255 characters.

email

The `email` attribute holds the email address of the user. A valid email must be provided. Maximum allowable length is 255 characters.

firstName

The value of the `firstName` attribute is the first name(s) of the user. A value here is obligatory. Maximum allowable length is 255 characters.

html

The value of the `html` attribute is a string of HTML that is generated by the system using the value of the `pageContent` attribute and the markup language specified in the `markupLanguage` attribute.

inputOrthography

The `inputOrthography` is a reference to an existing orthography model object. The purpose of a user-specific input orthography is to allow for the possibility that users will enter form transcriptions (and possibly also morpheme segmentations) using one orthography (i.e., their input orthography) but that these transcriptions will be translated into another orthography (i.e., the system-wide storage orthography) for storage in the database. When outputing the forms, the system would then re-translate them from the storage orthography into the user's output orthography. Previous OLD applications implemented this user-specific orthography conversion server-side. However, with the new architecture of the OLD >= 1.0 this added complication seems best implemented client-side.

lastName

The value of the `lastName` attribute is the last name of the user. A value here is obligatory. Maximum allowable length is 255 characters.

markupLanguage

The value of the `markupLanguage` attribute is one of "Markdown" or "reStructuredText" as defined in the `markupLanguages` variable of `lib/utils.py`. Markdown and reStructuredText are *lightweight markup languages*. A lightweight markup language is a markup language (i.e., a system for annotating a document) that is designed to be easy to read in its raw form. This value determines which markup-to-HTML function is employed when the system attempts to generate the `html` value from the user-supplied `pageContent` value. If no value is specified, "reStructuredText" will be the default.

outputOrthography

The `outputOrthography` is a reference to an existing orthography model object. The purpose of a user-specific input orthography is to allow for the possibility that users will enter form transcriptions (and possibly also morpheme segmentations) using one orthography (i.e., their input orthography) but that these transcriptions will be translated into another orthography (i.e., the system-wide storage orthography) for storage in the database. When outputing the forms, the system would then re-translate them from the storage orthography into the user's output orthography. Previous OLD applications implemented this user-specific orthography conversion server-side. However, with the new architecture of the OLD >= 1.0 this added complication seems best implemented client-side.

pageContent

The `pageContent` attribute holds a string representing the content of the user's page. This content should be written using the markup language specified in the `markupLanguage` attribute.

password

When creating a user, a valid value for the `password` attribute must be supplied. A valid password is composed of at least eight characters but no more than 255. It must contain either at least one printable character not in the printable ASCII range or one symbol, one digit, one uppercase letter and one lowercase letter. For example, "dave.Smith1" is a valid password, as is "philippe.gagne". (The latter contains a non-ASCII character.)

The users controller stores the password in the database encrypted using the `PassLib` module's implementation of the PBKDF2 key derivation function and the value of the `salt` attribute. During authentication attempts, the system applies the same encryption to the supplied password values and authentication succeeds if the encrypted password string from the request matches the encrypted password of the specified user. This means that even administrators of the system are unable to view any user passwords in their unencrypted form.

When specifying a new password, the input object passed in the request must also contain a `password_confirm` attribute whose value exactly matches that of the object's `password` attribute.

rememberedForms

The value of the `rememberedForms` attribute is a collection of form models that the user has "remembered". See the *Remembered forms* section for details on how to modify the value of this attribute. Note that this attribute is not included in the JSON object representation of user models. Retrieving a user's remembered forms requires a separate request to the `rememberedforms` resource.

role

The `role` attribute is used to classify users and is the basis for the authorization functionality. Every user must have a value for the `role` attribute. Valid values are "administrator", "contributor" and "viewer". Administrators have unrestricted access to all requests on all resources, contributors have read and write access to almost all resources and viewers have only read access. See the *Authentication & authorization* section for more details on roles and authorization.

salt

A value for the `salt` attribute is generated by the system when a user is created. This value is a randomly generated UUID. The salt aids in the secure encryption of the password.

username

The value of the `username` attribute is a string consisting of letters of the English alphabet, numbers and the underscore. Each user must have a unique `username` value and no two usernames may be the same. Only an administrator can update the username of a user model.

CHAPTER

THREE

API DOCUMENTATION

This section contains the API documentation automatically extracted from the source code.

3.1 onlinelinguisticdatabase

All of the Online Linguistic Database code lives here.

3.1.1 controllers

OLD controllers process user requests and return responses. The response body almost invariably consists of a JSON object.

applicationsettings

Contains the `ApplicationsettingsController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.applicationsettings.`**ApplicationsettingsController**

Generate responses to requests on application settings resources.

REST Controller styled on the Atom Publishing Protocol.

The most recently created application settings resource is considered to be the *active* one.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

Note: Only administrators are authorized to create, update or delete application settings resources.

---

create()

Create a new application settings resource and return it.

Url `POST /applicationsettings`

Request body JSON object representing the application settings to create.

Returns the newly created application settings.

delete(*id*)

Delete an existing application settings and return it.

Url `DELETE /applicationsettings/id`

Parameters **id** (*str*) – the `id` value of the application settings to be deleted.

Returns the deleted application settings model.

edit(*id*)

Return an application settings and the data needed to update it.

Url `GET /applicationsettings/edit` with optional query string parameters

Parameters **id** (*str*) – the `id` value of the application settings that will be updated.

Returns

a dictionary of the form:

```
{"applicationSettings": {...}, "data": {...}}
```

where the value of the `applicationSettings` key is a dictionary representation of the application settings and the value of the `data` key is a dictionary containing the objects necessary to update an application settings, viz. the return value of `ApplicationsettingsController.new()`.

Note: This action can be thought of as a combination of `ApplicationsettingsController.show()` and `ApplicationsettingsController.new()`. See `getNewApplicationSettingsData()` to understand how the query string parameters can affect the contents of the lists in the `data` dictionary.

---

index()

Get all application settings resources.

Url `GET /applicationsettings`

Returns a list of all application settings resources.

new()

Return the data necessary to create a new application settings.

Url `GET /applicationsettings/new` with optional query string parameters

Returns A dictionary of lists of resources

Note: See `getNewApplicationSettingsData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

---

show(*id*)

Return an application settings.

Url `GET /applicationsettings/id`

Parameters **id** (*str*) – the `id` value of the application settings to be returned.

Returns an application settings model object.

update(*id*)

Update an application settings and return it.

Url `PUT /applicationsettings/id`

Request body JSON object representing the application settings with updated attribute values.

Parameters **id** (*str*) – the `id` value of the application settings to be updated.

Returns the updated application settings model.

onlinelinguisticdatabase.controllers.applicationsettings.**createNewApplicationSettings**(*data*)
Create a new application settings.

Parameters **data** (*dict*) – the application settings to be created.

Returns an SQLAlchemy model object representing the application settings.

onlinelinguisticdatabase.controllers.applicationsettings.**getNewApplicationSettingsData**(*GET_*
Return the data necessary to create a new application settings or update an existing one.

Parameters **GET_params** – the `request.GET` dictionary-like object generated by Pylons which contains the query string parameters of the request.

Returns A dictionary whose values are lists of objects needed to create or update application settings.

If `GET_params` has no keys, then return all required data. If `GET_params` does have keys, then for each key whose value is a non-empty string (and not a valid ISO 8601 datetime) add the appropriate list of objects to the return dictionary. If the value of a key is a valid ISO 8601 datetime string, add the corresponding list of objects *only* if the datetime does *not* match the most recent `datetimeModified` value of the resource. That is, a non-matching datetime indicates that the requester has out-of-date data.

onlinelinguisticdatabase.controllers.applicationsettings.**updateApplicationSettings**(*application*
data)
Update an application settings.

Parameters

- **applicationSettings** – the application settings model to be updated.
- **data** (*dict*) – representation of the updated application settings.

Returns the updated application settings model or, if `changed` has not been set to `True`, then `False`.

collectionbackups

Contains the `CollectionbackupsController`.

class onlinelinguisticdatabase.controllers.collectionbackups.**CollectionbackupsController**
Generate responses to requests on collection backup resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

Note: Collection backups are created when updating and deleting collections; they cannot be created directly and they should never be deleted. This controller facilitates searching and getting of collection backups only.

---

index()
Get all collection backup resources.

Url `GET /collectionbackups`

Returns a list of all collection backup resources.

new_search()
Return the data necessary to search the collection backup resources.

Url `GET /collectionbackups/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of collection backup resources matching the input JSON query.

Url `SEARCH /collectionbackups` (or `POST /collectionbackups/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a collection backup.

Url `GET /collectionbackups/id`

Parameters **id** (*str*) – the `id` value of the collection backup to be returned.

Returns a collection backup model object.

elicitationmethods

Contains the `ElicitationmethodsController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.elicitationmethods.`**ElicitationmethodsController**
Generate responses to requests on elicitation method resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new elicitation method resource and return it.

Url `POST /elicitationmethods`

Request body JSON object representing the elicitation method to create.

Returns the newly created elicitation method.

delete(*id*)
Delete an existing elicitation method and return it.

Url `DELETE /elicitationmethods/id`

Parameters **id** (*str*) – the `id` value of the elicitation method to be deleted.

Returns the deleted elicitation method model.

edit(*id*)
Return an elicitation method and the data needed to update it.

Url `GET /elicitationmethods/edit` with optional query string parameters

Parameters **id** (*str*) – the `id` value of the elicitation method that will be updated.

Returns

a dictionary of the form:

```
{"elicitationMethod": {...}, "data": {...}}
```

where the value of the `elicitationMethod` key is a dictionary representation of the elicitation method and the value of the `data` key is a dictionary containing the objects necessary to update an elicitation method, viz. `{}`.

index()

Get all elicitation method resources.

Url `GET /elicitationmethods` with optional query string parameters for ordering and pagination.

Returns a list of all elicitation method resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()

Return the data necessary to create a new elicitation method.

Url `GET /elicitationmethods/new`

Returns an empty dictionary

show(*id*)

Return an elicitation method.

Url `GET /elicitationmethods/id`

Parameters **id** (*str*) – the `id` value of the elicitation method to be returned.

Returns an elicitation method model object.

update(*id*)

Update an elicitation method and return it.

Url `PUT /elicitationmethods/id`

Request body JSON object representing the elicitation method with updated attribute values.

Parameters **id** (*str*) – the `id` value of the elicitation method to be updated.

Returns the updated elicitation method model.

onlinelinguisticdatabase.controllers.elicitationmethods.**createNewElicitationMethod**(*data*)

Create a new elicitation method.

Parameters **data** (*dict*) – the elicitation method to be created.

Returns an SQLAlchemy model object representing the elicitation method.

onlinelinguisticdatabase.controllers.elicitationmethods.**updateElicitationMethod**(*elicitationMeth*
data)

Update an elicitation method.

Parameters

- **elicitationMethod** – the elicitation method model to be updated.
- **data** (*dict*) – representation of the updated elicitation method.

Returns the updated elicitation method model or, if `changed` has not been set to `True`, `False`.

error

Contains the `ErrorController`.

class `onlinelinguisticdatabase.controllers.error.`**ErrorController**
Generate JSON error objects as required.

The `StatusCodeRedirect` middleware forwards to `ErrorController` when error-related status codes are returned from the application.

This behaviour can be altered by changing the parameters to the `StatusCodeRedirect` middleware in the `config/middleware.py` file.

document()
Return a JSON object representing the error.

Instead of returning an HTML error document (the Pylons default), return the JSON object that the controller has specified for the response body. If the response body is not valid JSON, then it has been created by Routes; make it into valid JSON.

files

Contains the `FilesController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.files.`**FilesController**
Generate responses to requests on file resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new file resource and return it.

Url `POST /files`

Request body JSON object *or* conventional POST parameters containing the attribute values of the new file.

Content type `application/json` *or* `multipart/form-data`.

Returns the newly created file.

Note: There are three types of file and four types of file creation request.

1. **Local file with** `multipart/form-data` **content type.** File data are in the request body and the file metadata are structured as conventional POST parameters.
2. **Local file with** `application/json` **content type.** File data are Base64-encoded and are contained in the same JSON object as the metadata, in the request body.
3. **Subinterval-referencing file with** `application/json` **content type.** All parameters provided in a JSON object. No file data are present; the `id` value of an existing *audio/video* parent file must be provided in the `parentFile` attribute; values for `start` and `end` attributes are also required.

4. **Externally hosted file with** `application/json` **content-type.** All parameters provided in a JSON object. No file data are present; the value of the `url` attribute is a valid URL where the file data are being served.

`delete`(*id*)

Delete an existing file and return it.

Url `DELETE /files/id`

Parameters **id** (*str*) – the `id` value of the file to be deleted.

Returns the deleted file model.

Note: Only administrators and a file's enterer can delete it.

`edit`(*id*)

Return a file and the data needed to update it.

Url `GET /files/edit` with optional query string parameters

Parameters **id** (*str*) – the `id` value of the file that will be updated.

Returns

a dictionary of the form:

```
{"file": {...}, "data": {...}}
```

where the value of the `file` key is a dictionary representation of the file and the value of the `data` key is a dictionary containing the objects necessary to update a file, viz. the return value of `FilesController.new()`

Note: This action can be thought of as a combination of `FilesController.show()` and `FilesController.new()`. See `getNewEditFileData()` to understand how the query string parameters can affect the contents of the lists in the `data` dictionary.

`index`()

Get all file resources.

Url `GET /files` with optional query string parameters for ordering and pagination.

Returns a list of all file resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

`new`()

Return the data necessary to create a new file.

Url `GET /files/new` with optional query string parameters

Returns a dictionary of lists of resources.

Note: See `getNewEditFileData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

---

new_search()
Return the data necessary to search the file resources.

Url `GET /files/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of file resources matching the input JSON query.

Url `SEARCH /files` (or `POST /files/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

serve(*id*)
Return the file data (binary stream) of the file.

Parameters **id** (*str*) – the `id` value of the file whose file data are requested.

serve_reduced(*id*)
Return the reduced-size file data (binary stream) of the file.

Parameters **id** (*str*) – the `id` value of the file whose reduced-size file data are requested.

show(*id*)
Return a file.

Url `GET /files/id`

Parameters **id** (*str*) – the `id` value of the file to be returned.

Returns a file model object.

update(*id*)
Update a file and return it.

Url `PUT /files/id`

Request body JSON object representing the file with updated attribute values.

Parameters **id** (*str*) – the `id` value of the file to be updated.

Returns the updated file model.

onlinelinguisticdatabase.controllers.files.**addStandardMetadata**(*file*, *data*)
Add the standard metadata to the file model using the data dictionary.

Parameters

- **file** – file model object
- **data** (*dict*) – dictionary containing file attribute values.

Returns the updated file model object.

onlinelinguisticdatabase.controllers.files.**createBase64File**(*data*)
Create a local file using data from a `Content-Type: application/json` request.

Parameters

- **data** (*dict*) – the data to create the file model.

- **data['base64EncodedFile']** (*str*) – Base64-encoded file data.

Returns an SQLAlchemy model object representing the file.

onlinelinguisticdatabase.controllers.files.**createExternallyHostedFile**(*data*)
Create an externally hosted file.

Parameters

- **data** (*dict*) – the data to create the file model.
- **data['url']** (*str*) – a valid URL where the file data are served.

Returns an SQLAlchemy model object representing the file.

Optional keys of the data dictionary, not including the standard metadata ones, are `name`, `password` and `MIMEtype`.

onlinelinguisticdatabase.controllers.files.**createPlainFile**()
Create a local file using data from a `Content-Type: multipart/form-data` request.

Parameters

- **request.POST['filedata']** – a `cgi.FieldStorage` object containing the file data.
- **request.POST['filename']** (*str*) – the name of the binary file.

Returns an SQLAlchemy model object representing the file.

Note: The validator expects `request.POST` to encode list input via the `formencode.variabledecode.NestedVariables` format. E.g., a list of form `id` values would be provided as values to keys with names like `'forms-0'`, `'forms-1'`, `'forms-2'`, etc.

---

onlinelinguisticdatabase.controllers.files.**createSubintervalReferencingFile**(*data*)
Create a subinterval-referencing file.

Parameters

- **data** (*dict*) – the data to create the file model.
- **data['parentFile']** (*int*) – the `id` value of an audio/video file model.
- **data['start']** (*float/int*) – the start of the interval in seconds.
- **data['end']** (*float/int*) – the end of the interval in seconds.

Returns an SQLAlchemy model object representing the file.

A value for `data['name']` may also be supplied.

onlinelinguisticdatabase.controllers.files.**deleteFile**(*file*)
Delete a file model.

Parameters **file** – a file model object to delete.

Returns `None`.

This deletes the file model object from the database as well as any binary files associated with it that are stored on the filesystem.

onlinelinguisticdatabase.controllers.files.**getNewEditFileData**(*GET_params*)
Return the data necessary to create a new OLD file or update an existing one.

Parameters **GET_params** – the `request.GET` dictionary-like object generated by Pylons which contains the query string parameters of the request.

Returns A dictionary whose values are lists of objects needed to create or update files.

If `GET_params` has no keys, then return all relevant data lists. If `GET_params` does have keys, then for each key whose value is a non-empty string (and not a valid ISO 8601 datetime) add the appropriate list of objects to the return dictionary. If the value of a key is a valid ISO 8601 datetime string, add the corresponding list of objects *only* if the datetime does *not* match the most recent `datetimeModified` value of the resource. That is, a non-matching datetime indicates that the requester has out-of-date data.

`onlinelinguisticdatabase.controllers.files.`**`getUniqueFilePath`**(*filePath*)
Get a unique file path.

Parameters **filePath** (*str*) – an absolute file path.

Returns a tuple whose first element is the open file object and whose second is the unique file path as a unicode string.

`onlinelinguisticdatabase.controllers.files.`**`restrictFileByForms`**(*file*)
Restrict the entire file if it is associated to restricted forms.

Parameters **file** – a file model object.

Returns the file model object potentially tagged as "restricted".

`onlinelinguisticdatabase.controllers.files.`**`serveFile`**(*id*, *reduced=False*)
Serve the content (binary data) of a file.

Parameters

- **id** (*str*) – the `id` value of the file whose file data will be served.
- **reduced** (*bool*) – toggles serving of file data or reduced-size file data.

`onlinelinguisticdatabase.controllers.files.`**`updateExternallyHostedFile`**(*file*)
Update an externally hosted file model.

Parameters

- **file** – a file model object to update.
- **request.body** – a JSON object containing the data for updating the file.

Returns the file model or, if the file has not been updated, `False`.

`onlinelinguisticdatabase.controllers.files.`**`updateFile`**(*file*)
Update a local file model.

Parameters

- **file** – a file model object to update.
- **request.body** – a JSON object containing the data for updating the file.

Returns the file model or, if the file has not been updated, `False`.

`onlinelinguisticdatabase.controllers.files.`**`updateStandardMetadata`**(*file*, *data*, *changed*)
Update the standard metadata attributes of the input file.

Parameters

- **file** – a file model object to be updated.
- **data** (*dict*) – the data used to update the file model.
- **changed** (*bool*) – indicates whether the file has been changed.

Returns a tuple whose first element is the file model and whose second is the boolean `changed`.

onlinelinguisticdatabase.controllers.files.**updateSubintervalReferencingFile**(*file*)
Update a subinterval-referencing file model.

Parameters

- **file** – a file model object to update.
- **request.body** – a JSON object containing the data for updating the file.

Returns the file model or, if the file has not been updated, `False`.

formbackups

Contains the `FormbackupsController`.

class onlinelinguisticdatabase.controllers.formbackups.**FormbackupsController**
Generate responses to requests on form backup resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

Note: Form backups are created when updating and deleting forms; they cannot be created directly and they should never be deleted. This controller facilitates searching and getting of form backups only.

---

index()
Get all form backup resources.

Url `GET /formbackups`

Returns a list of all form backup resources.

new_search()
Return the data necessary to search the form backup resources.

Url `GET /formbackups/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of form backup resources matching the input JSON query.

Url `SEARCH /formbackups` (or `POST /formbackups/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a form backup.

Url `GET /formbackups/id`

Parameters **id** (*str*) – the `id` value of the form backup to be returned.

Returns a form backup model object.

forms

Contains the FormsController and its auxiliary functions.

class onlinelinguisticdatabase.controllers.forms.**FormsController**
Generate responses to requests on form resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The h.jsonify decorator converts the return value of the methods to JSON.

---

create()
Create a new form resource and return it.

Url POST /forms

Request body JSON object representing the form to create.

Returns the newly created form.

delete(*id*)
Delete an existing form and return it.

Url DELETE /forms/id

Parameters **id** (*str*) – the id value of the form to be deleted.

Returns the deleted form model.

Note: Only administrators and a form's enterer can delete it.

---

edit(*id*)
Return a form and the data needed to update it.

Url GET /forms/edit with optional query string parameters

Parameters **id** (*str*) – the id value of the form that will be updated.

Returns

a dictionary of the form:

```
{"form": {...}, "data": {...}}
```

where the value of the form key is a dictionary representation of the form and the value of the data key is a dictionary containing the objects necessary to update a form, viz. the return value of FormsController.new()

Note: This action can be thought of as a combination of FormsController.show() and FormsController.new(). See getNewEditFormData() to understand how the query string parameters can affect the contents of the lists in the data dictionary.

---

history(*id*)
Return the form with form.id==id and its previous versions.

Url GET /forms/history/id

Parameters **id** (*str*) – a string matching the id or UUID value of the form whose history is requested.

Returns

A dictionary of the form:

```
{"form": { ... }, "previousVersions": [ ... ]}
```

where the value of the `form` key is the form whose history is requested and the value of the `previousVersions` key is a list of dictionaries representing previous versions of the form.

index()
Get all form resources.

Url `GET /forms` with optional query string parameters for ordering and pagination.

Returns a list of all form resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new form.

Url `GET /forms/new` with optional query string parameters

Returns A dictionary of lists of resources

Note: See `getNewEditFormData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

---

new_search()
Return the data necessary to search the form resources.

Url `GET /forms/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

remember()
Cause the logged in user to remember the forms referenced in the request body.

Url `POST /forms/remember`

Request body A JSON object of the form `{"forms": [ ... ]}` where the value of the `forms` attribute is the array of form `id` values representing the forms that are to be remembered.

Returns A list of form `id` values corresponding to the forms that were remembered.

search()
Return the list of form resources matching the input JSON query.

Url `SEARCH /forms` (or `POST /forms/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a form.

Url `GET /forms/id`

Parameters **id** (*str*) – the `id` value of the form to be returned.

Returns a form model object.

update(*id*)
Update a form and return it.

Url `PUT /forms/id`

Request body JSON object representing the form with updated attribute values.

Parameters **id** (*str*) – the `id` value of the form to be updated.

Returns the updated form model.

update_morpheme_references()
Update the morphological analysis-related attributes of all forms.

That is, update the values of the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes of every form in the database.

Url `PUT /forms/update_morpheme_references`

Returns a list of ids corresponding to the forms where the update caused a change in the values of the target attributes.

> **Warning:** It should not be necessary to request the regeneration of morpheme references via this action since this should already be accomplished automatically by the calls to `updateFormsContainingThisFormAsMorpheme` on all successful update, create and delete requests on form resources. This action is, therefore, deprecated (read: use it with caution) and may be removed in future versions of the OLD.

onlinelinguisticdatabase.controllers.forms.**backupForm**(*formDict*, *datetimeModified=None*)
Backup a form.

Parameters

- **formDict** (*dict*) – a representation of a form model.
- **datetimeModified** (*datetime.datetime*) – the time of the form's last update.

Returns `None`

onlinelinguisticdatabase.controllers.forms.**compileMorphemicAnalysis**(*form*, *morphemeDelimiters=None*, ***kwargs*)
An error-handling wrapper arround `compileMorphemicAnalysis_()`.

Catch any error, log it and return a default 4-tuple.

Parameters

- **form** – the form model for which the morphological values are to be generated.
- **validDelimiters** (*list*) – morpheme delimiters as strings.

- **kwargs** (*dict*) – arguments that can affect the degree to which the database is queried.

Returns the output of `compileMorphemicAnalysis_()` or, if an error occurs, a 4-tuple of `None` objects.

`onlinelinguisticdatabase.controllers.forms.`**`compileMorphemicAnalysis_`**(*form*, *morphemeDelimiters=None*, ***kwargs*)

Generate values fo the morphological analysis-related attributes of a form model.

Parameters

- **form** – the form model for which the morphological values are to be generated.
- **validDelimiters** (*list*) – morpheme delimiters as strings.
- **kwargs** (*dict*) – arguments that can affect the degree to which the database is queried.

Returns a 4-tuple containing the generated values or all four `None` objects if no values can be generated.

Generate values for the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes of the input form.

For each morpheme detected in the `form`, search the database for forms whose `morphemeBreak` value matches the morpheme's phonemic form and whose `morphemeGloss` value matches the morpheme's gloss. If a perfect match is not found, searc the database for forms matching just the phonemic form or just the gloss.

Matching forms are represented as triples where the first element is the `id` value of the match, the second is its `morphemeBreak` or `morphemeGloss` value and the third is its `syntacticCategory.name` value. To illustrate, consider a form with `morphemeBreak` value u'chien-s' and `morphemeGloss` value u'dog-PL' and assume the lexical entries 'chien/dog/N/33', 's/PL/Agr/103' and 's/PL/Num/111' (where, for */a/b/c/d*, *a* is the `morphemeBreak` value, *b* is the `morphemeGloss` value, *c* is the `syntacticCategory.name` value and *d* is the `id` value. Running `compileMorphemicAnalysis()` on the target form returns the following 4-tuple `q`:

```
(
    json.dumps([[[[33, u'dog', u'N']], [[111, u'PL', u'Num'], [103, u'PL', u'Agr']]]]),
    json.dumps([[[[33, u'chien', u'N']], [[111, u's', u'Num'], [103, u's', u'Agr']]]]),
    u'N-Num',
    u'chien|dog|N-s|PL|Num'
)
```

where `q[0]` is the `morphemeBreakIDs` value, `q[1]` is the `morphemeGlossIDs` value, `q[2]` is `syntacticCategoryString` value and `q[3]` is `breakGlossCategory` value.

If `kwargs` contains a 'lexicalItems' or a 'deletedLexicalItems' key, then `compileMorphemicAnalysis()` will *update* (i.e., not re-create) the 4 relevant values of the form using only the items in `kwargs['lexicalItems']` or `kwargs['deletedLexicalItems']`. This facilitates lexical change percolation without massively redundant database queries.

`onlinelinguisticdatabase.controllers.forms.`**`createNewForm`**(*data*)

Create a new form.

Parameters **data** (*dict*) – the form to be created.

Returns an SQLAlchemy model object representing the form.

`onlinelinguisticdatabase.controllers.forms.`**`getFormAndPreviousVersions`**(*id*)

Return a form and its previous versions.

Parameters **id** (*str*) – the `id` or `UUID` value of the form whose history is requested.

Returns a tuple whose first element is the form model and whose second element is a list of form backup models.

`onlinelinguisticdatabase.controllers.forms.`**`getNewEditFormData`**(*GET_params*)
Return the data necessary to create a new OLD form or update an existing one.

Parameters **GET_params** – the `request.GET` dictionary-like object generated by Pylons which contains the query string parameters of the request.

Returns A dictionary whose values are lists of objects needed to create or update forms.

If `GET_params` has no keys, then return all data, i.e., grammaticalities, speakers, etc. If `GET_params` does have keys, then for each key whose value is a non-empty string (and not a valid ISO 8601 datetime) add the appropriate list of objects to the return dictionary. If the value of a key is a valid ISO 8601 datetime string, add the corresponding list of objects *only* if the datetime does *not* match the most recent `datetimeModified` value of the resource. That is, a non-matching datetime indicates that the requester has out-of-date data.

`onlinelinguisticdatabase.controllers.forms.`**`updateApplicationSettingsIfFormIsForeignWord`**(*form*)
Update the transcription validation functionality of the active application settings if the input form is a foreign word.

Parameters **form** – a form model object

Returns `None`

`onlinelinguisticdatabase.controllers.forms.`**`updateCollectionsReferencingThisForm`**(*form*)
Update all collections that reference the input form in their `contents` value.

When a form is deleted, it is necessary to update all collections whose `contents` value references the deleted form. The update removes the reference, recomputes the `contentsUnpacked`, `html` and `forms` attributes of the affected collection and causes all of these changes to percolate through the collection-collection reference chain.

Parameters **form** – a form model object

Returns `None`

Note: Getting the collections that reference this form by searching for those whose `forms` attribute contain it is not quite the correct way to do this because many of these collections will not *directly* reference this form – in short, this will result in redundant updates and backups.

`onlinelinguisticdatabase.controllers.forms.`**`updateForm`**(*form*, *data*)
Update a form model.

Parameters

- **form** – the form model to be updated.
- **data** (*dict*) – representation of the updated form.

Returns the updated form model or, if `changed` has not been set to `True`, then `False`.

`onlinelinguisticdatabase.controllers.forms.`**`updateFormsContainingThisFormAsMorpheme`**(*form*, *change='c*
pre-
vi-
ousVer-
sion=None

Update the morphological analysis-related attributes of every form containing the input form as morpheme.

Update the values of the `morphemeBreadIDs`, `morphemeGlossIDs`, `syntacticCategoryString`, and `breakGlossCategory` attributes of each form that contains the input form in its morphological analysis, i.e., each form whose `morphemeBreak` value contains the input form's `morphemeBreak` value as a morpheme or whose `morphemeGloss` value contains the input form's `morphemeGloss` line as a gloss. If the input form is not lexical (i.e., if it contains the space character or a morpheme delimiter), then no updates occur.

Parameters

- **form** – a form model object.
- **change** (*str*) – indicates whether the form has just been deleted or created/updated.
- **previousVersion** (*dict*) – a representation of the form prior to update.

Returns `None`

`onlinelinguisticdatabase.controllers.forms.`**`updateHasChangedTheAnalysis`**(*form*, *formDict*)

Return `True` if the update from formDict to form has changed the morphological analysis of the form.

`onlinelinguisticdatabase.controllers.forms.`**`updateMorphemeReferencesOfForm`**(*form*, *validDelimiters=None*, ***kwargs*)

Update the morphological analysis-related attributes of a form model.

Attempt to update the values of the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes of a form using only the lexical items specified in the list `kwargs['lexicalItems']` or the list `kwargs['deletedLexicalItems']`.

Parameters

- **form** – the form model to be updated.
- **validDelimiters** (*list*) – morpheme delimiters as strings.
- **kwargs['lexicalItems']** (*list*) – a list of form models.
- **kwargs['deletedLexicalItems']** (*list*) – a list of form models.

Returns the form if updated; else `False`.

`onlinelinguisticdatabase.controllers.forms.`**`updateMorphemeReferencesOfForms`**(*forms*, *validDelimiters*, ***kwargs*)

Update the morphological analysis-related attributes of a list of form models.

Attempt to update the values of the `morphemeBreakIDs`, `morphemeGlossIDs`, `syntacticCategoryString` and `breakGlossCategory` attributes of all forms in `forms` using only the lexical items specified in the list `kwargs['lexicalItems']` or `kwargs['deletedLexicalItems']`. Whenever an update occurs, backup the form and commit the changes.

Parameters

- **forms** (*list*) – the form models to be updated.
- **validDelimiters** (*list*) – morpheme delimiters as strings.
- **kwargs['lexicalItems']** (*list*) – a list of form models.
- **kwargs['deletedLexicalItems']** (*list*) – a list of form models.

Returns a list of form `id` values corresponding to the forms that have been updated.

formsearches

Contains the `FormsearchesController`.

class `onlinelinguisticdatabase.controllers.formsearches.`**`FormsearchesController`**

Generate responses to requests on form search resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

`create``()`

Create a new form search resource and return it.

Url `POST /formsearches`

Request body JSON object representing the form search to create.

Returns the newly created form search.

`delete``(`*id*`)`

Delete an existing form search and return it.

Url `DELETE /formsearches/id`

Parameters **id** (*str*) – the `id` value of the form search to be deleted.

Returns the deleted form search model.

`edit``(`*id*`)`

GET /formsearches/id/edit: Return the data necessary to update an existing OLD form search.

`index``()`

Get all form search resources.

Url `GET /formsearches` with optional query string parameters for ordering and pagination.

Returns a list of all form search resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

`new``()`

GET /formsearches/new: Return the data necessary to create a new OLD form search.

`new_search``()`

Return the data necessary to search the form search resources.

Url `GET /formsearches/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()

Return the list of form search resources matching the input JSON query.

Url `SEARCH /formsearches` (or `POST /formsearches/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

Note: Yes, that's right, you can search form searches. (No, you can't search searches of form searches :)

---

show(*id*)

Return a form search.

Url `GET /formsearches/id`

Parameters **id** (*str*) – the `id` value of the form search to be returned.

Returns a form search model object.

update(*id*)

Update a form search and return it.

Url `PUT /formsearches/id`

Request body JSON object representing the form search with updated attribute values.

Parameters **id** (*str*) – the `id` value of the form search to be updated.

Returns the updated form search model.

onlinelinguisticdatabase.controllers.formsearches.**createNewFormSearch**(*data*)

Create a new form search.

Parameters **data** (*dict*) – the form search to be created.

Returns an form search model object.

onlinelinguisticdatabase.controllers.formsearches.**updateFormSearch**(*formSearch*, *data*)

Update a form search model.

Parameters

- **form** – the form search model to be updated.
- **data** (*dict*) – representation of the updated form search.

Returns the updated form search model or, if `changed` has not been set to `True`, then `False`.

languages

Contains the `LanguagesController`.

class onlinelinguisticdatabase.controllers.languages.**LanguagesController**
Generate responses to requests on language resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

Note: The language table is populated from an ISO 639-3 file upon application setup. The language resouces are read-only. This controller facilitates searching and getting of languages resources.

---

index()
Get all language resources.

Url `GET /languages`

Returns a list of all language resources.

new_search()
Return the data necessary to search the language resources.

Url `GET /languages/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of language resources matching the input JSON query.

Url `SEARCH /languages` (or `POST /languages/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a language.

Url `GET /languages/id`

Parameters **id** (*str*) – the `id` value of the language to be returned.

Returns a language model object.

login

Contains the LoginController.

class onlinelinguisticdatabase.controllers.login.**LoginController**
Handles authentication-related functionality.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

authenticate()
Session-based authentication.

Url `POST /login/authenticate`

Request body A JSON object with `"username"` and `"password"` string values

Returns `{"authenticated": True}` on success, an error dictionary on failure.

email_reset_password()
Reset the user's password and email them a new one.

Url `POST /login/email_reset_password`

Request body a JSON object with a `"username"` attribute.

Returns a dictionary with `'validUsername'` and `'passwordReset'` keys whose values are booleans.

logout()
Logout user by deleting the session.

Url `POST /login/logout`.

Returns `{"authenticated": False}`.

oldcollections

Contains the `OldcollectionsController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.oldcollections.`**OldcollectionsController**
Generate responses to requests on collection resources.

REST Controller styled on the Atom Publishing Protocol.

The collections controller is one of the more complex ones. A great deal of this complexity arised from the fact that collections can reference forms and other collections in the value of their `contents` attribute. The propagation of restricted tags and associated forms and the generation of the html from these contents-with-references, necessitates some complex logic for updates and deletions.

Warning: There is a potential issue with collection-collection reference. A restricted user can restrict their own collection *A* and that restriction would be propagated up the reference chain, possibly causing another collection *B* (that was not created by the updater) to become restricted. That is, collection-collection reference permits restricted users to indirectly restrict collections they would otherwise not be permitted to restrict. This will be bothersome to other restricted users since they will no longer be able to access the newly restricted collection *B*.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new collection resource and return it.

Url `POST /collections`

Request body JSON object representing the collection to create.

Returns the newly created collection.

delete(*id*)
Delete an existing collection and return it.

Url `DELETE /collections/id`

Parameters **id** (*str*) – the `id` value of the collection to be deleted.

Returns the deleted collection model.

Note: Only administrators and a collection's enterer can delete it.

edit(*id*)
Return a collection and the data needed to update it.

Url `GET /collections/edit` with optional query string parameters

Parameters **id** (*str*) – the `id` value of the collection that will be updated.

Returns

a dictionary of the form:

```
{"collection": {...}, "data": {...}}
```

where the value of the `collection` key is a dictionary representation of the collection and the value of the `data` key is a dictionary containing the objects necessary to update a collection, viz. the return value of `CollectionsController.new()`

Note: This action can be thought of as a combination of `CollectionsController.show()` and `CollectionsController.new()`. See `getNewEditCollectionData()` to understand how the query string parameters can affect the contents of the lists in the `data` dictionary.

history(*id*)
Return a collection and its previous versions.

Url `GET /collections/history/id`

Parameters **id** (*str*) – a string matching the `id` or `UUID` value of the collection whose history is requested.

Returns

a dictionary of the form:

```
{"collection": { ... }, "previousVersions": [ ... ]}
```

where the value of the `collection` key is the collection whose history is requested and the value of the `previousVersions` key is a list of dictionaries representing previous versions of the collection.

index()
GET /collections: Return all collections.

new()
Return the data necessary to create a new collection.

Url `GET /collections/new` with optional query string parameters

Returns a dictionary of lists of resources.

Note: See `getNewEditCollectionData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

new_search()
Return the data necessary to search the collection resources.

Url `GET /collections/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of collection resources matching the input JSON query.

Url `SEARCH /collections` (or `POST /collections/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a collection.

Url `GET /collections/id`

Parameters **id** (*str*) – the `id` value of the collection to be returned.

Returns a collection model object.

update(*id*)
Update a collection and return it.

Url `PUT /collections/id`

Request body JSON object representing the collection with updated attribute values.

Parameters **id** (*str*) – the `id` value of the collection to be updated.

Returns the updated collection model.

onlinelinguisticdatabase.controllers.oldcollections.**addContentsUnpackedToValues**(*values*, *collectionsReferenced*)
Add a `'contentsUnpacked'` value to values and return values.

Parameters

- **values** (*dict*) – data for creating a collection.
- **collectionsReferenced** (*dict*) – keys are collection `id` values and values are collection models.

Returns `values` updated.

onlinelinguisticdatabase.controllers.oldcollections.**addFormIdsListToValues**(*values*)
Add a list of referenced form ids to values.

Parameters **values** (*dict*) – data for creating or updating a collection

Returns `values` with a `'forms'` key whose value is a list of id integers.

onlinelinguisticdatabase.controllers.oldcollections.**backupCollection**(*collectionDict*, *datetimeModified=None*)

Backup a collection.

Parameters

- **formDict** (*dict*) – a representation of a collection model.
- **datetimeModified** (*datetime.datetime*) – the time of the collection's last update.

Returns `None`

onlinelinguisticdatabase.controllers.oldcollections.**createNewCollection**(*data*, *collectionsReferenced*)

Create a new collection.

Parameters

- **data** (*dict*) – the collection to be created.
- **collectionsReferenced** (*dict*) – the collection models recursively referenced in `data['contents']`.

Returns an SQLAlchemy model object representing the collection.

onlinelinguisticdatabase.controllers.oldcollections.**generateContentsUnpacked**(*contents*, *collectionsReferenced*, *patt=None*)

Generate the `contentsUnpacked` value of a collection.

Parameters

- **contents** (*unicode*) – the value of the `contents` attribute of a collection
- **collectionsReferenced** (*dict*) – the collection models referenced by a collection; keys are collection `id` values.
- **patt** – a compiled regexp pattern object that matches collection references.

Returns a unicode object as a value for the `contentsUnpacked` attribute of a collection model.

Note: Circular, invalid and unauthorized reference chains are caught in the generation of `collectionsReferenced`.

---

onlinelinguisticdatabase.controllers.oldcollections.**getCollection**(*collectionId*, *user*, *unrestrictedUsers*)

Return the collection such that `collection.id==collectionId`.

If the collection does not exist or if `user` is not authorized to access it, raise an appropriate error.

Parameters

- **collectionId** (*int*) – the `id` value of a collection.
- **user** – a user model of the logged in user.
- **unrestrictedUsers** (*list*) – the unrestricted users of the system.

Returns a collection model object.

`onlinelinguisticdatabase.controllers.oldcollections.`**`getCollectionAndPreviousVersions`**(*id*)

Return a collection and its previous versions.

Parameters **id** (*str*) – the `id` or `UUID` value of the collection whose history is requested.

Returns a tuple whose first element is the collection model and whose second element is a list of collection backup models.

`onlinelinguisticdatabase.controllers.oldcollections.`**`getCollectionsReferenced`**(*contents*, *user=None*, *unrestrictedUsers=None*, *collectionId=None*, *patt=None*)

Return the collections (recursively) referenced by the input `contents` value.

That is, return all of the collections referenced in the input `contents` value, plus all of the collections referenced in those collections, etc.

Parameters

- **contents** (*unicode*) – the value of the `contents` attribute of a collection.
- **user** – the user model who made the request.
- **unrestrictedUsers** (*list*) – the unrestricted user models of the application.
- **collectionId** (*int*) – the `id` value of a collection.
- **patt** – a compiled regular expression object.

Returns a dictionary whose keys are collection `id` values and whose values are collection models.

`onlinelinguisticdatabase.controllers.oldcollections.`**`getCollectionsReferencedInContents`**(*collectionsReferenced*)

Get the immediately referenced collections of a collection.

Parameters

- **collection** – a collection model.
- **collectionsReferenced** (*dict*) – keys are collection `id` values and values are collection models.

Returns a list of collection models; useful in determining whether directly referenced collections are restricted.

onlinelinguisticdatabase.controllers.oldcollections.**getCollectionsReferencingThisCollectio**

Return all collections that recursively reference `collection`.

That is, return all collections that reference `collection` plus all collections that reference those referencing collections, etc.

Parameters

- **collection** – a collection model object.
- **queryBuilder** – an `SQLAQueryBuilder` instance.

Returns a list of collection models.

onlinelinguisticdatabase.controllers.oldcollections.**getContents**(*collectionId*, *collectionsReferenced*)

Return the `contents` value of the collection with `collectionId` as its `id` value.

Parameters

- **collectionId** (*int*) – the `id` value of a collection model.
- **collectionsReferenced** (*dict*) – the collections (recursively) referenced by a collection.

Returns the contents of a collection, or a warning message.

onlinelinguisticdatabase.controllers.oldcollections.**getNewEditCollectionData**(*GET_params*)

Return the data necessary to create a new OLD collection or update an existing one.

Parameters **GET_params** – the `request.GET` dictionary-like object generated by Pylons which contains the query string parameters of the request.

Returns A dictionary whose values are lists of objects needed to create or update collections.

If `GET_params` has no keys, then return all data. If `GET_params` does have keys, then for each key whose value is a non-empty string (and not a valid ISO 8601 datetime) add the appropriate list of objects to the return dictionary. If the value of a key is a valid ISO 8601 datetime string, add the corresponding list of objects *only* if the datetime does *not* match the most recent `datetimeModified` value of the resource. That is, a non-matching datetime indicates that the requester has out-of-date data.

onlinelinguisticdatabase.controllers.oldcollections.**getUnicode**(*key*, *dict_*)

Return `dict_[key]`, making sure it defaults to a unicode object.

onlinelinguisticdatabase.controllers.oldcollections.**removeReferencesToThisCollection**(*contents*, *collectionId*)

Remove references to a collection from the `contents` value of another collection.

Parameters

- **contents** (*unicode*) – the value of the `contents` attribute of a collection.
- **collectionId** (*int*) – an `id` value of a collection.

Returns the modified `contents` string.

onlinelinguisticdatabase.controllers.oldcollections.**removeReferencesToThisForm**(*contents*, *formId*)

Remove references to a form from the `contents` value of another collection.

Parameters

- **contents** (*unicode*) – the value of the `contents` attribute of a collection.
- **formId** (*int*) – an `id` value of a form.

Returns the modified `contents` string.

onlinelinguisticdatabase.controllers.oldcollections.**updateCollection**(*collection*, *data*, *collectionsReferenced*)

Update a collection model.

Parameters

- **collection** – the collection model to be updated.
- **data** (*dict*) – representation of the updated collection.
- **collectionsReferenced** (*dict*) – the collection models recursively referenced in `data['contents']`.

Returns a 3-tuple where the second and third elements are invariable booleans indicating whether the collection has become restricted or has had its `contents` value changed as a result of the update, respectively. The first element is the updated collection or `False` of the no update has occurred.

onlinelinguisticdatabase.controllers.oldcollections.**updateCollectionByDeletionOfReferencedF**

Update a collection based on the deletion of a form it references.

This function is called in the `FormsController` when a form is deleted. It is called on each collection that references the deleted form and the changes to each of those collections are propagated through all of the collections that reference them, and so on.

Parameters

- **collection** – a collection model object.
- **referencedForm** – a form model object.

Returns `None`.

onlinelinguisticdatabase.controllers.oldcollections.**updateCollectionsThatReferenceThisColle**

Update all collections that reference the input collection.

Parameters

- **collection** – a collection model.
- **queryBuilder** – an `SQLAQueryBuilder` instance.
- **kwargs['contents_changed']** (*bool*) – indicates whether the input collection's `contents` value has changed.
- **kwargs['deleted']** (*bool*) – indicates whether the input collection has just been deleted.

Returns None

Update the contents, contentsUnpacked, html and/or form attributes of every collection that references the input collection plus all of the collections that reference those collections, etc. This function is called upon successful update and delete requests.

If the contents of the collection have changed (i.e., kwargs['contents_changed']==True) , then retrieve all collections that reference collection and all collections that reference those referers, etc., and update their contentsUnpacked, html and forms attributes.

If the collection has been deleted (i.e., kwargs['deleted']==True), then recursively retrieve all collections referencing collection and update their contents, contentsUnpacked, html and forms attributes.

If collection has just been tagged as restricted (i.e., kwargs['restricted']==True), then recursively restrict all collections that reference it.

In all cases, update the datetimeModified value of every collection that recursively references collection.

orthographies

Contains the OrthographiesController and its auxiliary functions.

class onlinelinguisticdatabase.controllers.orthographies.**OrthographiesController**
Generate responses to requests on orthography resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The h.jsonify decorator converts the return value of the methods to JSON.

---

create()
Create a new orthography resource and return it.

Url POST /orthographies

Request body JSON object representing the orthography to create.

Returns the newly created orthography.

delete(*id*)
Delete an existing orthography and return it.

Url DELETE /orthographies/id

Parameters **id** (*str*) – the id value of the orthography to be deleted.

Returns the deleted orthography model.

Note: Contributors can only delete orthographies that are not used in the active application settings.

---

edit(*id*)
Return an orthography and the data needed to update it.

Url GET /orthographies/edit

Parameters **id** (*str*) – the id value of the orthography that will be updated.

Returns

a dictionary of the form:

```
{"orthography": {...}, "data": {...}}
```

where the value of the `orthography` key is a dictionary representation of the orthography and the value of the `data` key is an empty dictionary.

index()
Get all orthography resources.

Url `GET /orthographies` with optional query string parameters for ordering and pagination.

Returns a list of all orthography resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new orthography.

Url `GET /orthographies/new`

Returns an empty dictionary

show(*id*)
Return an orthography.

Url `GET /orthographies/id`

Parameters **id** (*str*) – the `id` value of the orthography to be returned.

Returns an orthography model object.

update(*id*)
Update an orthography and return it.

Url `PUT /orthographies/id`

Request body JSON object representing the orthography with updated attribute values.

Parameters **id** (*str*) – the `id` value of the orthography to be updated.

Returns the updated orthography model.

Note: Contributors can only update orthographies that are not used in the active application settings.

`onlinelinguisticdatabase.controllers.orthographies.`**createNewOrthography**(*data*)
Create a new orthography.

Parameters **data** (*dict*) – the data for the orthography to be created.

Returns an SQLAlchemy model object representing the orthography.

`onlinelinguisticdatabase.controllers.orthographies.`**updateOrthography**(*orthography*, *data*)
Update an orthography.

Parameters

- **orthography** – the orthography model to be updated.
- **data** (*dict*) – representation of the updated orthography.

Returns the updated orthography model or, if `changed` has not been set to `True`, `False`.

pages

Contains the PagesController and its auxiliary functions.

class onlinelinguisticdatabase.controllers.pages.**PagesController**
Generate responses to requests on page resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The h.jsonify decorator converts the return value of the methods to JSON.

---

create()
Create a new page resource and return it.

Url POST /pages

Request body JSON object representing the page to create.

Returns the newly created page.

delete(*id*)
Delete an existing page and return it.

Url DELETE /pages/id

Parameters **id** (*str*) – the id value of the page to be deleted.

Returns the deleted page model.

edit(*id*)
Return a page and the data needed to update it.

Url GET /pages/edit

Parameters **id** (*str*) – the id value of the page that will be updated.

Returns

a dictionary of the form:

```
{"page": {...}, "data": {...}}
```

where the value of the page key is a dictionary representation of the page and the value of the data key is the list of valid markup language names.

index()
Get all page resources.

Url GET /pages with optional query string parameters for ordering and pagination.

Returns a list of all page resources.

Note: See utils.addOrderBy() and utils.addPagination() for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new page.

Url GET /pages/new.

Returns a dictionary containing the names of valid OLD markup languages.

show(*id*)
Return a page.

Url `GET /pages/id`

Parameters **id** (*str*) – the `id` value of the page to be returned.

Returns a page model object.

update(*id*)
Update a page and return it.

Url `PUT /pages/id`

Request body JSON object representing the page with updated attribute values.

Parameters **id** (*str*) – the `id` value of the page to be updated.

Returns the updated page model.

`onlinelinguisticdatabase.controllers.pages.`**createNewPage**(*data*)
Create a new page.

Parameters **data** (*dict*) – the data for the page to be created.

Returns an SQLAlchemy model object representing the page.

`onlinelinguisticdatabase.controllers.pages.`**updatePage**(*page*, *data*)
Update a page.

Parameters

- **page** – the page model to be updated.
- **data** (*dict*) – representation of the updated page.

Returns the updated page model or, if `changed` has not been set to `True`, `False`.

phonologies

Contains the `PhonologiesController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.phonologies.`**PhonologiesController**
Generate responses to requests on phonology resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new phonology resource and return it.

Url `POST /phonologies`

Request body JSON object representing the phonology to create.

Returns the newly created phonology.

delete(*id*)
Delete an existing phonology and return it.

Url `DELETE /phonologies/id`

Parameters **id** (*str*) – the `id` value of the phonology to be deleted.

Returns the deleted phonology model.

edit(*id*)
Return a phonology and the data needed to update it.

Url `GET /phonologies/edit`

Parameters **id** (*str*) – the `id` value of the phonology that will be updated.

Returns

a dictionary of the form:

```
{"phonology": {...}, "data": {...}}
```

where the value of the `phonology` key is a dictionary representation of the phonology and the value of the `data` key is an empty dictionary.

index()
Get all phonology resources.

Url `GET /phonologies` with optional query string parameters for ordering and pagination.

Returns a list of all phonology resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new phonology.

Url `GET /phonologies/new`.

Returns an empty dictionary.

show(*id*)
Return a phonology.

Url `GET /phonologies/id`

Parameters **id** (*str*) – the `id` value of the phonology to be returned.

Returns a phonology model object.

update(*id*)
Update a phonology and return it.

Url `PUT /phonologies/id`

Request body JSON object representing the phonology with updated attribute values.

Parameters **id** (*str*) – the `id` value of the phonology to be updated.

Returns the updated phonology model.

onlinelinguisticdatabase.controllers.phonologies.**createNewPhonology**(*data*)
Create a new phonology.

Parameters **data** (*dict*) – the data for the phonology to be created.

Returns an SQLAlchemy model object representing the phonology.

onlinelinguisticdatabase.controllers.phonologies.**updatePhonology**(*phonology*, *data*)
Update a phonology.

Parameters

- **page** – the phonology model to be updated.
- **data** (*dict*) – representation of the updated phonology.

Returns the updated phonology model or, if `changed` has not been set to `True`, `False`.

rememberedforms

Contains the `RememberedformsController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.rememberedforms.`**RememberedformsController**

Generate responses to requests on remembered forms resources.

REST Controller styled on the Atom Publishing Protocol.

Note: Remembered forms is a pseudo-REST-ful resource. Remembered forms are stored in the `userform` many-to-many table (cf. `model/user.py`) which defines the contents of a user's `rememberedForms` attribute (as well as the contents of a form's `memorizers` attribute). A user's remembered forms are not affected by requests to the user resource. Instead, the remembered forms resource handles modification, retrieval and search of a user's remembered forms.

Overview of the interface:

- `GET /rememberedforms/id`
- `UPDATE /rememberedforms/id`
- `SEARCH /rememberedforms/id`

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

search (*id*)

Return the remembered forms of a user that match the input JSON query.

Url `SEARCH /rememberedforms/id` (or `POST /rememberedforms/id/search`).

Parameters **id** (*str*) – the `id` value of the user whose remembered forms are searched.

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show (*id*)

Return a user's remembered forms.

Url `GET /rememberedforms/id` with optional query string parameters for ordering and pagination.

Parameters **id** (*str*) – the `id` value of a user model.

Returns a list form models.

Note: Any authenticated user is authorized to access this resource. Restricted forms are filtered from the array on a per-user basis.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

update(*id*)

Update a user's remembered forms and return them.

Url `PUT /rememberedforms/id`

Request body JSON object of the form `{"forms": [...]}` where the array contains the form `id` values that will constitute the user's `rememberedForms` collection after update.

Parameters **id** (*str*) – the `id` value of the user model whose `rememberedForms` attribute is to be updated.

Returns the list of remembered forms of the user.

Note: Administrators can update any user's remembered forms; non-administrators can only update their own.

---

sources

Contains the `SourcesController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.sources.`**SourcesController**

Generate responses to requests on source resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()

Create a new source resource and return it.

Url `POST /sources`

Request body JSON object representing the source to create.

Returns the newly created source.

delete(*id*)

Delete an existing source and return it.

Url `DELETE /sources/id`

Parameters **id** (*str*) – the `id` value of the source to be deleted.

Returns the deleted source model.

edit(*id*)

Return a source and the data needed to update it.

Url `GET /sources/edit`

Parameters **id** (*str*) – the `id` value of the source that will be updated.

Returns

a dictionary of the form:

```
{"source": {...}, "data": {...}}
```

where the value of the `source` key is a dictionary representation of the source and the value of the `data` key is the list of BibTeX entry types.

index()
Get all source resources.

Url `GET /sources` with optional query string parameters for ordering and pagination.

Returns a list of all source resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new source.

Url `GET /sources/new`.

Returns a dictionary containing the valid BibTeX entry types.

new_search()
Return the data necessary to search the source resources.

Url `GET /sources/new_search`

Returns `{"searchParameters": {"attributes": { ... }, "relations": { ... }}`

search()
Return the list of source resources matching the input JSON query.

Url `SEARCH /sources` (or `POST /sources/search`)

Request body A JSON object of the form:

```
{"query": {"filter": [ ... ], "orderBy": [ ... ]},
 "paginator": { ... }}
```

where the `orderBy` and `paginator` attributes are optional.

show(*id*)
Return a source.

Url `GET /sources/id`

Parameters **id** (*str*) – the `id` value of the source to be returned.

Returns a source model object.

Note: A source associated to a restricted file will still return a subset of the restricted file's metadata. However, restricted users will be unable to retrieve the file data of the file because of the authorization logic in the retrieve action of the files controller.

---

update(*id*)
Update a source and return it.

Url `PUT /sources/id`

Request body JSON object representing the source with updated attribute values.

Parameters **id** (*str*) – the `id` value of the source to be updated.

Returns the updated source model.

`onlinelinguisticdatabase.controllers.sources.`**createNewSource**(*data*)
Create a new source.

Parameters **data** (*dict*) – the data for the source to be created.

Returns an SQLAlchemy model object representing the source.

`onlinelinguisticdatabase.controllers.sources.`**updateSource**(*source*, *data*)
Update a source.

Parameters

- **source** – the source model to be updated.
- **data** (*dict*) – representation of the updated source.

Returns the updated source model or, if `changed` has not been set to `True`, `False`.

speakers

Contains the `SpeakersController` and its auxiliary functions.

class `onlinelinguisticdatabase.controllers.speakers.`**SpeakersController**
Generate responses to requests on speaker resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new speaker resource and return it.

Url `POST /speakers`

Request body JSON object representing the speaker to create.

Returns the newly created speaker.

delete(*id*)
Delete an existing speaker and return it.

Url `DELETE /speakers/id`

Parameters **id** (*str*) – the `id` value of the speaker to be deleted.

Returns the deleted speaker model.

edit(*id*)
Return a speaker resource and the data needed to update it.

Url `GET /speakers/edit`

Parameters **id** (*str*) – the `id` value of the speaker that will be updated.

Returns

a dictionary of the form:

```
{"speaker": {...}, "data": {...}}
```

where the value of the `speaker` key is a dictionary representation of the speaker and the value of the `data` key is an empty dictionary.

index()
Get all speaker resources.

Url `GET /speakers` with optional query string parameters for ordering and pagination.

Returns a list of all speaker resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new speaker.

Url `GET /speakers/new`.

Returns an empty dictionary.

show(*id*)
Return a speaker.

Url `GET /speakers/id`

Parameters **id** (*str*) – the `id` value of the speaker to be returned.

Returns a speaker model object.

update(*id*)
Update a speaker and return it.

Url `PUT /speakers/id`

Request body JSON object representing the speaker with updated attribute values.

Parameters **id** (*str*) – the `id` value of the speaker to be updated.

Returns the updated speaker model.

onlinelinguisticdatabase.controllers.speakers.**createNewSpeaker**(*data*)
Create a new speaker.

Parameters **data** (*dict*) – the data for the speaker to be created.

Returns an SQLAlchemy model object representing the speaker.

onlinelinguisticdatabase.controllers.speakers.**updateSpeaker**(*speaker*, *data*)
Update a speaker.

Parameters

- **speaker** – the speaker model to be updated.
- **data** (*dict*) – representation of the updated speaker.

Returns the updated speaker model or, if `changed` has not been set to `True`, `False`.

syntacticcategories

Contains the `SyntacticcategoriesController` and its auxiliary functions.

class onlinelinguisticdatabase.controllers.syntacticcategories.**SyntacticcategoriesController**
Generate responses to requests on syntactic category resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new syntactic category resource and return it.

Url `POST /syntacticcategorys`

Request body JSON object representing the syntactic category to create.

Returns the newly created syntactic category.

delete(*id*)
Delete an existing syntactic category and return it.

Url `DELETE /syntacticcategorys/id`

Parameters **id** (*str*) – the `id` value of the syntactic category to be deleted.

Returns the deleted syntactic category model.

edit(*id*)
Return a syntactic category resource and the data needed to update it.

Url `GET /syntacticcategorys/edit`

Parameters **id** (*str*) – the `id` value of the syntactic category that will be updated.

Returns

a dictionary of the form:

```
{"syntacticCategory": {...}, "data": {...}}
```

where the value of the `syntacticCategory` key is a dictionary representation of the syntactic category and the value of the `data` key is a dictionary of valid syntactic category types as defined in `onlinelinguisticdatabase.lib.utils`.

index()
Get all syntactic category resources.

Url `GET /syntacticcategorys` with optional query string parameters for ordering and pagination.

Returns a list of all syntactic category resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new syntactic category.

Url `GET /syntacticcategorys/new`.

Returns a dictionary containing the valid syntactic category types as defined in `onlinelinguisticdatabase.lib.utils`.

show(*id*)
Return a syntactic category.

Url `GET /syntacticcategorys/id`

Parameters **id** (*str*) – the `id` value of the syntactic category to be returned.

Returns a syntactic category model object.

update(*id*)
Update a syntactic category and return it.

Url `PUT /syntacticcategorys/id`

Request body JSON object representing the syntactic category with updated attribute values.

Parameters **id** (*str*) – the `id` value of the syntactic category to be updated.

Returns the updated syntactic category model.

onlinelinguisticdatabase.controllers.syntacticcategories.**createNewSyntacticCategory**(*data*)
Create a new syntactic category.

Parameters **data** (*dict*) – the data for the syntactic category to be created.

Returns an SQLAlchemy model object representing the syntactic category.

onlinelinguisticdatabase.controllers.syntacticcategories.**updateFormsReferencingThisCategory**
Update all forms that reference a syntactic category.

Parameters **syntacticCategory** – a syntactic category model object.

Returns `None`

Note: This function is only called when a syntactic category is deleted or when its name is changed.

---

onlinelinguisticdatabase.controllers.syntacticcategories.**updateSyntacticCategory**(*syntacticCateg*, *data*)
Update a syntactic category.

Parameters

- **syntacticCategory** – the syntactic category model to be updated.
- **data** (*dict*) – representation of the updated syntactic category.

Returns the updated syntactic category model or, if `changed` has not been set to `True`, `False`.

tags

Contains the `TagsController` and its auxiliary functions.

class onlinelinguisticdatabase.controllers.tags.**TagsController**
Generate responses to requests on tag resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new tag resource and return it.

Url `POST /tags`

Request body JSON object representing the tag to create.

Returns the newly created tag.

delete(*id*)
Delete an existing tag and return it.

Url `DELETE /tags/id`

Parameters **id** (*str*) – the `id` value of the tag to be deleted.

Returns the deleted tag model.

edit(*id*)
Return a tag resource and the data needed to update it.

Url `GET /tags/edit`

Parameters **id** (*str*) – the `id` value of the tag that will be updated.

Returns

a dictionary of the form:

```
{"tag": {...}, "data": {...}}
```

where the value of the `tag` key is a dictionary representation of the tag and the value of the `data` key is an empty dictionary.

index()
Get all tag resources.

Url `GET /tags` with optional query string parameters for ordering and pagination.

Returns a list of all tag resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()
Return the data necessary to create a new tag.

Url `GET /tags/new`.

Returns an empty dictionary.

show(*id*)
Return a tag.

Url `GET /tags/id`

Parameters **id** (*str*) – the `id` value of the tag to be returned.

Returns a tag model object.

update(*id*)
Update a tag and return it.

Url `PUT /tags/id`

Request body JSON object representing the tag with updated attribute values.

Parameters **id** (*str*) – the `id` value of the tag to be updated.

Returns the updated tag model.

onlinelinguisticdatabase.controllers.tags.**createNewTag**(*data*)
Create a new tag.

Parameters **data** (*dict*) – the data for the tag to be created.

Returns an SQLAlchemy model object representing the tag.

onlinelinguisticdatabase.controllers.tags.**updateTag**(*tag*, *data*)
Update a tag.

Parameters

- **tag** – the tag model to be updated.
- **data** (*dict*) – representation of the updated tag.

Returns the updated tag model or, if `changed` has not been set to `True`, `False`.

users

Contains the `UsersController` and its auxiliary functions.

class onlinelinguisticdatabase.controllers.users.**UsersController**
Generate responses to requests on user resources.

REST Controller styled on the Atom Publishing Protocol.

Note: The `h.jsonify` decorator converts the return value of the methods to JSON.

---

create()
Create a new user resource and return it.

Url `POST /users`

Request body JSON object representing the user to create.

Returns the newly created user.

Note: Only administrators are authorized to create users.

---

delete(*id*)
Delete an existing user and return it.

Url `DELETE /users/id`

Parameters **id** (*str*) – the `id` value of the user to be deleted.

Returns the deleted user model.

edit(*id*)
Return a user resource and the data needed to update it.

Url `GET /users/edit`

Parameters **id** (*str*) – the `id` value of the user that will be updated.

Returns

a dictionary of the form:

```
{"user": {...}, "data": {...}}
```

where the value of the `user` key is a dictionary representation of the user and the value of the `user` key is a dictionary of lists of resources.

Note: See `getNewUserData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

---

index()

Get all user resources.

Url `GET /users` with optional query string parameters for ordering and pagination.

Returns a list of all user resources.

Note: See `utils.addOrderBy()` and `utils.addPagination()` for the query string parameters that effect ordering and pagination.

---

new()

Return the data necessary to create a new user.

Url `GET /users/new` with optional query string parameters .

Returns a dictionary of lists of resources.

Note: See `getNewUserData()` to understand how the query string parameters can affect the contents of the lists in the returned dictionary.

---

show(*id*)

Return a user.

Url `GET /users/id`

Parameters **id** (*str*) – the `id` value of the user to be returned.

Returns a user model object.

update(*id*)

Update a user and return it.

Url `PUT /users/id`

Request body JSON object representing the user with updated attribute values.

Parameters **id** (*str*) – the `id` value of the user to be updated.

Returns the updated user model.

onlinelinguisticdatabase.controllers.users.**createNewUser**(*data*)

Create a new user.

Parameters **data** (*dict*) – the data for the user to be created.

Returns an SQLAlchemy model object representing the user.

onlinelinguisticdatabase.controllers.users.**getNewUserData**(*GET_params*)

Return the data necessary to create a new OLD user or update an existing one.

Parameters **GET_params** – the `request.GET` dictionary-like object generated by Pylons which contains the query string parameters of the request.

Returns A dictionary whose values are lists of objects needed to create or update user.

If GET_params has no keys, then return all data. If GET_params does have keys, then for each key whose value is a non-empty string (and not a valid ISO 8601 datetime) add the appropriate list of objects to the return dictionary. If the value of a key is a valid ISO 8601 datetime string, add the corresponding list of objects *only* if the datetime does *not* match the most recent datetimeModified value of the resource. That is, a non-matching datetime indicates that the requester has out-of-date data.

onlinelinguisticdatabase.controllers.users.**updateUser**(*user*, *data*)
Update a user.

Parameters

- **user** – the user model to be updated.
- **data** (*dict*) – representation of the updated user.

Returns the updated user model or, if changed has not been set to True, False.

3.1.2 lib

Modules containing functionality used by numerous other modules.

SQLAQueryBuilder

This module defines SQLAQueryBuilder. An SQLAQueryBuilder instance is used to build an SQLAlchemy query object from a Python data structure (nested lists).

The two public methods are getSQLAQuery and getSQLAFilter. Both take a list representing a filter expression as input. getSQLAQuery returns an SQLAlchemy query object, including joins and filters. getSQLAFilter returns an SQLAlchemy filter expression and is called by getSQLAQuery. Errors in the Python filter expression will cause custom OLDSearchParseErrors to be raised.

The searchable models and their attributes (scalars & collections) are defined in SQLAQueryBuilder.schema.

Simple filter expressions are lists with four or five items. Complex filter expressions are constructed via lists whose first element is one of the boolean keywords 'and', 'or', 'not' and whose second element is a filter expression or a list thereof (in the case of 'and' and 'or'). The examples below show a filter expression accepted by SQLAQueryBuilder('Form').getSQLAQuery on the second line followed by the equivalent SQLAlchemy ORM expression. Note that the target model of the SQLAQueryBuilder is set to 'Form' so all queries will be against the Form model.

1. Simple scalar queries:

```
['Form', 'transcription', 'like', '%a%']
Session.query(Form).filter(Form.transcription.like(u'%a%'))
```

2. Scalar relations:

```
['Form', 'enterer', 'firstName', 'regex', '^[JS]']
Session.query(Form).filter(Form.enterer.has(User.firstName.op('regexp')(u'^[JS]')))
```

3. Scalar relations presence/absence:

```
['Form', 'enterer', '=', 'None']
Session.query(Form).filter(Form.enterer==None)
```

4. Collection relations (w/ SQLA's collection.any() method):

```
['Form', 'files', 'id', 'in', [1, 2, 33, 5]]
Session.query(Form).filter(Form.files.any(File.id.in_([1, 2, 33, 5])))
```

5. Collection relations (w/ joins; should return the same results as (4)):

['File', 'id', 'in', [1, 2, 33, 5]] fileAlias = aliased(File) Session.query(Form).filter(fileAlias.id.in_([1, 2, 33, 5])).outerjoin(fileAlias, Form.files)

6. Collection relations presence/absence:

```
['Form', 'files', '=', None]
Session.query(Form).filter(Form.files == None)
```

7. Negation:

```
['not', ['Form', 'transcription', 'like', '%a%']]
Session.query(Form).filter(not_(Form.transcription.like(u'%a%')))
```

8. Conjunction:

```
['and', [['Form', 'transcription', 'like', '%a%'],
         ['Form', 'elicitor', 'id', '=', 13]]]
Session.query(Form).filter(and_(Form.transcription.like(u'%a%'),
                                Form.elicitor.has(User.id==13)))
```

9. Disjunction:

```
['or', [['Form', 'transcription', 'like', '%a%'],
        ['Form', 'dateElicited', '<', '2012-01-01']]]
Session.query(Form).filter(or_(Form.transcription.like(u'%a%'),
                               Form.dateElicited < datetime.date(2012, 1, 1)))
```

10. Complex:

```
['and', [['Translation', 'transcription', 'like', '%1%'],
         ['not', ['Form', 'morphemeBreak', 'regex', '[28][5-7]']],
         ['or', [['Form', 'datetimeModified', '<', '2012-03-01T00:00:00'],
                 ['Form', 'datetimeModified', '>', '2012-01-01T00:00:00']]]]]
translationAlias = aliased(Translation)
Session.query(Form).filter(and_(
    translationAlias.transcription.like(u'%1%'),
    not_(Form.morphemeBreak.op('regexp')(u'[28][5-7]')),
    or_(
        Form.datetimeModified < ...,
        Form.datetimeModified > ...
    )
)).outerjoin(translationAlias, Form.translations)
```

Note also that SQLAQueryBuilder detects the RDBMS and issues collate commands where necessary to ensure that pattern matches are case-sensitive while ordering is not.

A further potential enhancement would be to allow doubly relational searches, e.g., return all forms whose enterer has remembered a form with a transcription like 'a':

11. Scalar's collection relations:

```
['Form', 'enterer', 'rememberedForms', 'transcription', 'like', '%a%']
Session.query(Form).filter(Form.enterer.has(User.rememberedForms.any(
    Form.transcription.like('%1%'))))
```

class onlinelinguisticdatabase.lib.SQLAQueryBuilder.**SQLAQueryBuilder**(*modelName='Form'*, *primaryKey='id'*, ***kwargs*)

Generate an SQLAlchemy query object from a Python dictionary.

Builds SQLAlchemy queries from Python data structures representing arbitrarily complex filter expressions. Joins are inferred from the filter expression. The public method most likely to be used is `getSQLAQuery()`. Example usage:

```
queryBuilder = SQLAlchemyQueryBuilder()
pythonQuery = {'filter': [
    'and', [
        ['Translation', 'transcription', 'like', '1'],
        ['not', ['Form', 'morphemeBreak', 'regex', '[28][5-7]']],
        ['or', [
            ['Form', 'datetimeModified', '<', '2012-03-01T00:00:00'],
            ['Form', 'datetimeModified', '>', '2012-01-01T00:00:00']]]]]}
query = queryBuilder.getSQLAQuery(pythonQuery)
forms = query.all()
```

getSQLAFilter(*python*)
Return the SQLAlchemy filter expression generable by the input Python data structure or raise an OLDSearchParseError if the data structure is invalid.

getSQLAOrderBy(*orderBy*, *primaryKey='id'*)
The public method clears the errors and then calls the private method. This prevents interference from errors generated by previous orderBy calls.

If you can't find the information you're looking for, have a look at the index or try to find it using the search function:

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Kopka.2004] Kopka, Helmut and Daly, Patrick W. 2004. Guide to LATEX. Addison-Wesley Professional.

PYTHON MODULE INDEX

t

u