clld Documentation

Release 3.3.3

Robert Forkel

Dec 19, 2017

Contents

CHAPTER 1

The Project

The goal of the Cross-Linguistic Linked Data project (CLLD) is to help record the world's language diversity heritage. This is to be facilitated by developing, providing and maintaining interoperable data publication structures.

For more information refer to the project's website at clld.org.

CHAPTER 2

The clld framework

Underlying all applications built within the project to publish datasets is the clld framework - a python package providing functionality to build and maintain CLLD apps.

2.1 Overview

clld provides

- a common core database model clld.db.models.common,
- a pyramid application scaffold,
- a core web application implemented in the pyramid framework clld.web.app,
- scripts exploiting the core database model,
- libraries for common problems when working with linguistic databases.

Online documentation is at readthedocs, source code and issue tracker at GitHub.

Contents:

2.1.1 Getting started

Requirements

Starting with version 0.13 clld works with python 2.7 and 3.4. It has been installed and run successfully on Ubuntu 12.04, Mac OSX (see install_mac) and Windows (see install_win). While it might be possible to use sqlite as database backend, all production installations of clld and most development is done with postgresql 9.1. To retrieve the clld software from GitHub, git must be installed on the system.

Installation

To install the python package from pypi run

    pip install clld

To install from a git repository, you may run the following commands in an activated virtualenv:

```
git clone git@github.com:clld/clld.git
cd clld
python setup.py develop
```

Alternatively, you may want to fork `clld` first and then work with your fork.

Bootstrapping a `clld` app

A `clld` app is a python package implementing a pyramid web application.

The `clld` package provides a pyramid application scaffold to create the initial package directory layout for a `clld` app:

```
pcreate -t clld_app myapp
```

Note: The `pcreate` command has been installed with pyramid as a dependency of `clld`.

This will create a python package `myapp` with the following layout:

```
(clld)robert@astroman:~/venvs/clld$ tree myapp/
myapp/                              # project directory
├── development.ini                 # deployment settings
├── fabfile.py                      # fabric tasks for managing the application
├── MANIFEST.in
├── myapp                           # package directory
│   ├── adapters.py                 # custom adapters
│   ├── appconf.ini                 # custom application settings
│   ├── assets.py                   # registers custom static assets with the clld␣
↪framework
│   ├── datatables.py               # custom datatables
│   ├── __init__.py                 # contains the main function
│   ├── interfaces.py               # custom interface specifications
│   ├── locale                      # locale directory, may be used for custom␣
↪translations
│   │   └── myapp.pot
│   ├── maps.py                     # custom map objects
│   ├── models.py                   # custom database objects
│   ├── scripts
│   │   ├── initializedb.py         # database initialization script
│   │   └── __init__.py
│   ├── static                      # custom static assets
│   │   ├── project.css
│   │   └── project.js
│   ├── templates                   # custom mako templates
│   │   ├── dataset                 # custom templates for resources of type Dataset
│   │   │   └── detail_html.mako    # the home page of the app
│   │   └── myapp.mako              # custom site template
│   ├── tests
```

```
│   │   ├── __init__.py
│   │   ├── test_functional.py
│   │   └── test_selenium.py
│   └── views.py
├── README.txt
├── setup.cfg
└── setup.py
```

Running:

```
cd myapp
python setup.py develop
```

will install your app as Python package in development mode, i.e. will create a link to your app's code in the `site-packages` directory.

Now edit the configuration file, `myapp/development.ini` providing a setting `sqlalchemy.url` in the `[app:main]` section. The SQLAlchemy engine URL given in this setting must point to an existing (but empty) database if the `postgresql` dialect is chosen.

Running:

```
python myapp/scripts/initializedb.py development.ini
```

will then create the database for your app. Whenever you edit the database initialization script, you have to re-run the above command.

Note: If you are using PostgreSQL as rdbms the above command will not automatically drop an existing database, so before running it, you have to drop and re-create and empty database "by hand".

You are now ready to run:

```
pserve --reload development.ini
```

and navigate with your browser to http://127.0.0.1:6543 to visit your application.

The next step is populating the database.

Populating the database

The `clld` framework does not provide any GUI or web interface for populating the database. Instead, this is assumed to be done with a script. You can edit `clld/scripts/initializedb.py` to fill the database with your data and run:

```
python myapp/scripts/initializedb.py development.ini
```

Adding objects to the database is done by instantiating model objects and adding them to `clld.db.meta.DBSession`. (This session is already initialized when your code in `initializedb.py` runs.) For more information about database objects read the chapter *Declarative base and mixins*.

A minimal example (building upon the default `main` function in `initializedb.py` as created for the app skeleton) adding just two *Value* objects may look as follows

```
def main(args):
    data = Data()
```

```
    dataset = common.Dataset(id=myapp.__name__, domain='myapp.clld.org')
    DBSession.add(dataset)

    # All ValueSets must be related to a contribution:
    contrib = common.Contribution(id='contrib', name='the contribution')

    # All ValueSets must be related to a Language:
    lang = common.Language(id='lang', name='A Language', latitude=20, longitude=20)

    param = common.Parameter(id='param', name='Feature 1')

    # ValueSets group Values related to the same Language, Contribution and
    # Parameter
    vs = common.ValueSet(id='vs', language=lang, parameter=param,␣
↪contribution=contrib)

    # Values store the actual "measurements":
    DBSession.add(common.Value(id='v1', name='value 1', valueset=vs))
    DBSession.add(common.Value(id='v2', name='value 2', valueset=vs))
```

A more involved example, creating instances of all core model classes, is available in chapter *Populating the database of a clld app*.

The `data` object present in the `main` function in `initializedb.py` is an instance of

class clld.scripts.util.**Data**(***kw*)

Dictionary, serving to store references to new db objects during data imports.

The values are dictionaries, keyed by the name of the model class used to create the new objects.

```
>>> data = Data()
>>> l = data.add(common.Language, 'l', id='abc', name='Abc Language')
>>> assert l == data['Language']['l']
```

add(*model*, *key*, ***kw*)

Create an instance of a model class to be persisted in the database.

Parameters

- **model** – The model class we want to create an instance of.
- **key** – A key which can be used to retrieve the instance later.
- **kw** – Keyword parameters passed to model class for initialisation.

Returns The newly created instance of model class.

Thus, you can create objects which you can reference later like

```
data.add(common.Language, 'mylangid', id='1', name='French')
data.add(common.Unit, 'myunitid', id='1', language=data['Language']['mylangid'])
```

Note: Using `data.add` for all objects may not be a good idea for big datasets, because keeping references to all objects prevents garbage collection and will blow up the memory used for the import process. Some experimentation may be required if you hit this problem. As a general rule: only use `data.add` for objects that you actually need to lookup lateron.

Note: All model classes derived from *clld.db.meta.Base* have an integer primary key `pk`. This primary key

is defined in such a way (at least for PostgreSQL and SQLite) that you do not have to specify it when instantiating an object (although you may do so).

The dataset

Each `clld` app is assumed to serve a dataset, so you must add an instance of *`clld.db.models.common.Dataset`* to your database. This dataset is assumed to have a publisher and a license. Information about the publisher and the license should be part of the data, as well as other metadata about the dataset.

A note on files

A `clld` app may have static data files associated with its resources (e.g. soundfiles). The `clld` framework is designed to store these files in the filesystem and just keep references to them in the database. While this does require a more complex import and export process, it helps keeping the database small, and allows serving the static files directly from a webserver instead of having to go through the web application (which is still possible, though).

To specify where in the filesystem these static files are stored, a configuration setting `clld.files` must point to a directory on the local filesystem. This setting is evaluated when a file's "create" method is called, or its URL is calculated.

Note that there's an additional category of static files - downloads - which are treated differently because they are not considered primary but derived data which can be recreated at any time. To separate these concerns physically, downloads are typically stored in a different directory than primary data files.

Deployment

TODO: clld.environment == 'production', webassets need to be built. gunicorn + nginx

Examples

A good way explore how to customize a `clld` app is by looking at the code of existing apps. These apps are listed at http://clld.org/datasets.html and each app links to its source code repository on GitHub (in the site footer).

2.1.2 Populating the database of a `clld` app

In the following we will show how to create instances of all core model classes, thus populating the database of a `clld` app. The code snippets should be understood as living inside the `main` function of an app's `scripts.initializedb` module.

Metadata

```
data = Data()

dataset = common.Dataset(id=myapp.__name__, domain='myapp.clld.org')
DBSession.add(dataset)

# All ValueSets must be related to a contribution:
contrib = common.Contribution(id='contrib', name='the contribution')
```

```
# All ValueSets must be related to a Language:
data.add(common.Language, 'eng', id='eng', name='English', latitude=52.0, longitude=0.
↪0)
data.add(common.Language, 'abk', id='abk', name='Abkhaz', latitude=43.08,␣
↪longitude=41.0)
```

Note: We use a *clld.scripts.util.Data* instance and its `add` method to create objects we want to reference lateron.

Language-level parameters and values

Structural databases like WALS are best modeled using *clld.db.models.common.Parameter* objects for structural features and *clld.db.models.common.Value* objects for a single value assignment. So code to add WALS-like data could look as follows:

```
feature1 = common.Parameter(id='1A', name='Consonant Inventories')

# ValueSets group Values related to the same Language, Contribution and Parameter
vs = common.ValueSet(id='1A-eng', language=data['Language']['eng'],␣
↪parameter=feature1, contribution=contrib)

# Values store the actual "measurements":
DBSession.add(common.Value(id='1A-eng', name='Average', valueset=vs))
```

Parameters often allow only values from a fixed domain. This can be modeled using `clld.db.models.common.DomainElement` objects:

```
feature2 = common.Parameter(id='9A', name='The velar nasal')

# We add a DomainElement for Paramter feature2 ...
no_velar_nasal = common.DomainElement(id='9A-1', name='No velar nasal',␣
↪parameter=feature2)

vs = common.ValueSet(id='1A-abk', language=data['Language']['abk'],␣
↪parameter=feature2, contribution=contrib)

# ... and reference this DomainElement when creating a Value:
DBSession.add(common.Value(id='1A-abk', valueset=vs, domainelement=no_velar_nasal))
```

Unit-level parameters and values

Lexical databases typically provide information on words or lexemes. This kind of data can be modeled using *clld.db.models.common.Unit* and *clld.db.models.common.UnitParameter* objects.

```
# We model words as units of a language:
unit = common.Unit(id='unit', name='hand', language=data['Language']['eng'])

# Part of speech is a typical parameter which can be "measured" for words or lexemes.
pos = common.UnitParameter(id='pos', name='part of speech')

DBSession.add(common.UnitValue(id='unit-pos', name='noun', unit=unit,␣
↪unitparameter=pos, contribution=contrib))
```

Note: We could have used `clld.db.models.common.UnitDomainElement` objects to model a controlled list of valid part-of-speech values.

2.1.3 Resources

Resources are a central concept in `clld`. While we may use the term resource also for single instances, more generally a resource is a type of data implementing an interface to which behaviour can be attached.

The default resources known in a `clld` app are listed in `clld.RESOURCES`, but it is possible to extend this list when configuring a custom app (see *Adding a resource*).

Resources have the following attributes:

name a string naming the resource.

interface class specifying the interface the resource implements.

model core model class for the resource.

Behaviour may be tied to a resource either via the `name` (as is the case for *Routes*) or via the `interface` (as is the case for *Adapters*).

Models

Each resource is associated with a db model class and optionally with a custom db model derived from the default one using joined table inheritance.

Adapters

Adapters are basically used to provide representations of a resource. Thus, if we want to provide the classification tree of a Glottolog languoid in newick format, we have to write and register an adapter. This kind of adapter is generally implemented as subclass of *`clld.web.adapters.base.Representation`* or *`clld.web.adapters.base.Index`*.

For the builtin resources a couple of adapters are registered by default:

- a template-based adapter to render the details page,
- a JSON representation of the resource (based on `clld.web.adapters.base.JSON`).
- a CSV representation of a resource index (`clld.web.adapters.csv.CsvAdapter`).

Routes

The `clld` framework uses URL dispatch to map default views to URLs for resources.

For each resource the following routes and views (and URLs) are registered by default:

- an index view for the route `<name>s` and the URL `/<name>s`,
- an alternative index view for the route `<name>s_alt` and the URL pattern `/<name>s.{ext}`,
- a details view for the route `<name>` and the URL pattern `/<name>s/{id}`,
- an alternative details view for the route `<name>_alt` and the URL `/<name>s/{id}.{ext}`.

Views

We distinguish two classes of views for resources:

- index views, implemented in `clld.web.views.index_view()`, serve rendered adapters registered for the interface `IIndex` and a particular resource. They typically require a corresponding `DataTable` subclass to be instantiated as context object when the view is executed.
- detail views, implemented in `clld.web.views.detail_view()`, serve rendered adapters registered for the interface `IRepresentation` and a particular resource. The resource instance with the `id` passed in the request will be fetched from the database as context object of the view.

Templates

The adapters associated with resources may use templates to render the response. In particular this is the case for the HTML index and detail view.

Providing custom data for a reources details template

Since the view rendering a resources details representations is implemented in clld core code, clld applications may need a way to provide additional context for the templates. This can be done by implementing an appropriately named function in the app.util which will be looked up and called in a BeforeRender event subscriber.

Requesting a resource

The flow of events when a resource is requested from a `clld` app is as follows (we don't give a complete rundown but only highlight the deviations from the general pyramid request processing flow):

1. When a route for a resource matches, the corresponding factory function is called to obtain the context of the request. For index routes this context object is an instance of a DataTable, for a details route this is an instance of the resource's model class (or a custom specialization of this model).
2. For index routes `clld.web.views.index_view()` is called, for details routes `clld.web.views.resource_view()`.
3. Both of these look up the appropriate adapter registered for the context, instantiate it and call its `render_to_response` method. The result of this call is returned as `Response`.
4. If this method uses a standard template renderer the listener for the `BeforeRender` event will look for a function in `myapp.util` with a name of `<resource_name>_<template_basename>`, e.g. `dataset_detail_html` for the template `templates/dataset/detail_html.mako`. If such a function exists, it will be called with the current template variables as keyword parameters. The return value of the function is expected to be a dictionary which will be used to update the template variables.

2.1.4 Data modeling

This chapter describes how to model cross-linguistic data using the core resources available in the `clld` framework. While it is possible to extend the core data model in various ways, sticking to core resources for comparable concepts will ensure re-usability of the data, because all of the data publication mechanisms implemented in `clld` will be available.

Dataset

Each `clld` app is assumed to serve a cross-linguistic dataset. The *clld.db.models.common.Dataset* object holds metadata about the dataset, e.g. the publisher and license and relations to editors.

Languages

Languages are the core objects which are described in datasets served by `clld` apps. *clld.db.models.common.Language* - like most other objects - are at the most basic level described by a name, an optional description and an optional geographical coordinate.

To allow identification of languages across apps or even domains, languages can be associated with any number of alternative `clld.db.models.common.Identifier`; typically glottocodes or iso 639-3 codes or alternative names.

Parameters

clld.db.models.common.Parameter objects are used to model language parameters, i.e. phenomena (aka features) which can be measured across languages. Single datapoints, i.e. measurements of the parameter for a single language are modeled as instances of *clld.db.models.common.Value*. To support multiple measurements for the same (language, parameter) pair, values are grouped in a *clld.db.models.common.ValueSet*, and it is the valueset that is related to language and parameter.

Enumerated domain

`clld` supports enumerated domains. Elements of the domain of a parameter can be modeled as `clld.db.models.common.DomainElement` instances and each value must then be related to one domain element.

The `clld` framework will then use the `domain` property of a parameter to select behaviour suitable for enumerated domains only, e.g. loading values associated with one domain element as separate layer when displaying a parameter map.

Typed values

The `clld` framework is agnostic with regard to the types of values, i.e. as far as default functionality is concerned the only properties required of a value are a `name` and an `id` (and optionally a `description`). To simply store typed data for values multiple mechanisms are available.

- Storing typed data in the `jsondata` dictionary: This accomodates all data types which can be serialized as JSON, i.e. numbers, booleans, arrays, dictionaries.
- If the data for a value comes as a list or dictionary of strings, it can also be stored as `clld.db.models.common.Value_data` instances.
- Finally there's the option to store data related to a value as files, i.e. as instances of `clld.db.models.common.Value_files`.

2.1.5 Customizing a CLLD app

Extending or customizing the default behaviour of a CLLD app is basically what pyramid calls configuration. So, since the `clld_app` scaffold is somewhat tuned towards imperative configuration, this means calling methods on the config object returned by the call to `clld.web.app.get_configurator()` in the apps `main` function. Since

the config object is an instance of the pyramid Configurator this includes all the standard ways to configure pyramid apps, in particular adding routes and views to provide additional pages and funtionality with an app.

Wording

Most text displayed on the HTML pages of the default app can be customized using a technique commonly called localization. I.e. the default is set up in an "internationalized" way, which can be "localized" by providing alternative "translations".

These translations are provided in form of a PO file which can be edited by hand or with tools such as Poedit.

The workflow to create alternative translations for core terms of a CLLD app is as follows:

1. Extract terms from your code to create the app specific translations file `myapp/locale/en/LC_MESSAGES/clld.po`:

   ```
   python setup.py extract_messages
   ```

2. Look up the terms available for translation in `clld/locale/en/LC_MESSAGES/clld.po`. If the term you want to translate is found, go on. Otherwise file an issue at https://github.com/clld/clld/issues

3. Initialize a localized catalog for your app running:

   ```
   python setup.py init_catalog -l en
   ```

4. When installing `clld` tools have been installed to extract terms from python code files. To make the term available for extraction, include code like below in `myapp`.

```
# _ is a recognized name for a function to mark translatable strings
_ = lambda s: s
_('term you wish to translate')
```

5. Extract terms from your code and update the local `myapp/locale/en/LC_MESSAGES/clld.po`:

   ```
   python setup.py extract_messages
   python setup.py update_catalog
   ```

6. Add a translation by editing `myapp/locale/en/LC_MESSAGES/clld.po`.

7. Compile the catalog:

   ```
   python setup.py compile_catalog
   ```

If you restart your app you should see your translation at places where previously the core term appeared. Whenever you want to add translations, you have to go through steps 3–6 above.

Static Pages

TODO: reserved route names, . . .

Templates

The default CLLD app comes with a set of Mako templates (in `clld/web/templates`) which control the rendering of HTML pages. Each of these can be overridden locally by providing a template file with the same path (relative to the `templates` directory); i.e. to override `clld/web/templates/language/detail_html.mako`

– the template rendered for the details page of languages (see *Templates*) – you'd have to provide a file `myapp/templates/language/detail_html.mako`.

Static assets

CLLD Apps may provide custom css and js code. If this code is placed in the default locations `myapp/static/project.[css|js]`, it will automatically be packaged for production. Note that in this case the code should not contain any URLs relative to the file, because these may break in production.

Additionally, you may provide the logo of the publisher of the dataser as a PNG image. If this file is located at `myapp/static/publisher_logo.png` it will be picked up automatically by the default application footer template.

Other static content can still be placed in the `myapp/static` directory but must be explicitely included on pages making use of it, e.g. with template code like:

```
<link href="${request.static_url('myapp:static/css/introjs.min.css')}" rel="stylesheet
↪">
<script src="${request.static_url('myapp:static/js/intro.min.js')}"></script>
```

Menu Items

Registering non-default menu items can only be done wholesale, i.e. replacing the whole main menu by calling the `register_menu` method of the config object.

register_menu(*items) #*

Parameters **items** – (name, factory) pairs, where factory is a callable that accepts the two parameters (ctx, req) and returns a pair (url, label) to use for the menu link and name is used to compare with the `active_menu` attribute of templates.

Datatables

A main building block of CLLD apps are dynamic data tables. Although there are default implementations which may be good enough in many cases, each data table can be fully customized as follows.

1. Define a customized datatable class in `myapp/datables.py` inheriting from either *clld.web.datatables.base.DataTable* or one of its subclasses in `clld.web.datatables`.

2. Register this datatable for the page you want to display it on by adding a line like the following to the function `myapp.datatables.includeme`:

```
config.register_datatable('routename', DataTableClassName)
```

The `register_datatable` method of the config object has the following signature:

register_datatable(*route_name*, *cls*)

Parameters

- **route_name** (*str*) – Name of the route which maps to the view serving the data (see *Routes*).
- **cld** (*class*) – Python class inheriting from *clld.web.datatables.base.DataTable*.

Datatables are always registered for the routes serving the data. Often they are displayed on the corresponding resource's index page, but sometimes you will want to display a datatable on some other page, e.g. a list of parameter values on the parameter detail's page. This can be done be inserting a call to *clld.web.app.CldRequest.get_datatable()* to create a datatable instance which can then be rendered calling its `render` method.

As an example, the code to render a values datatable restricted to the values for a particular parameter instance `param` would look like

```
request.get_datatable('values', h.models.Value, parameter=param).render()
```

Customize column definitions

Overwrite *clld.web.datatables.base.DataTable.col_defs()*.

Customize query

Overwrite *clld.web.datatables.base.DataTable.base_query()*.

Data model

The core `clld` data model can be extended for CLLD apps by defining additional mappings in `myapp.models` in two ways:

1. Additional mappings (thus additional database tables) deriving from *clld.db.meta.Base* can be defined.

Note: While deriving from *clld.db.meta.Base* may add some columns to your table which you don't actually need (e.g. `created`, ...), it is still important to do so, to ensure custom objects behave the same as core ones.

---

2. Customizations of core models can be defined using joined table inheritance:

```
from sqlalchemy import Column, Integer, ForeignKey
from zope.interface import implementer
from clld.interfaces import IContribution
from clld.db.meta import CustomModelMixin
from clld.db.models.common import Contribution

@implementer(IContribution)
class Chapter(Contribution, CustomModelMixin):
    """Contributions in WALS are chapters chapters. These comprise a set of features␣
↪with
    corresponding values and a descriptive text.
    """
    pk = Column(Integer, ForeignKey('contribution.pk'), primary_key=True)
    # add more Columns and relationships here
```

Note: Inheriting from *clld.db.meta.CustomModelMixin* takes care of half of the boilerplate code necessary to make inheritance work. The primary key still has to be defined "by hand".

To give an example, here's how one could model the many-to-many relation between words and meanings often encountered in lexical databases:

```
from clld import interfaces
from clld.db.models import common
from clld.db.meta import CustomModelMixin

@implementer(interfaces.IParameter)
class Meaning(CustomModelMixin, common.Parameter):
    pk = Column(Integer, ForeignKey('parameter.pk'), primary_key=True)

@implementer(interfaces.IValueSet)
class SynSet(CustomModelMixin, common.ValueSet):
    pk = Column(Integer, ForeignKey('valueset.pk'), primary_key=True)

@implementer(interfaces.IUnit)
class Word(CustomModelMixin, common.Unit):
    pk = Column(Integer, ForeignKey('unit.pk'), primary_key=True)

@implementer(interfaces.IValue)
class Counterpart(CustomModelMixin, common.Value):
    """a counterpart relates a meaning with a word
    """
    pk = Column(Integer, ForeignKey('value.pk'), primary_key=True)

    word_pk = Column(Integer, ForeignKey('unit.pk'))
    word = relationship(Word, backref='counterparts')
```

The definitions of `Meaning`, `Synset` and `Word` above are not strictly necessary (because they do not add any relations or columns to the base classes) and are only added to make the semantics of the model clear.

Now if we have an instance of `Word`, we can iterate over its meanings like this

```
for counterpart in word.counterparts:
    print counterpart.valueset.parameter.name
```

A more involved example for the case of tree-structured data is given in *Handling Trees*.

Adding a resource

You may also want to add new resources in your app, i.e. objects that behave like builtin resources in that routes get automatically registered and view and template lookup works as explained in *Requesting a resource*. An example for this technique are the families in e.g. WALS.

The steps required to add a custom resource are:

1. Define an interface for the resource in `myapp/interfaces.py`:

```
from zope.interface import Interface

class IFamily(Interface):
    """marker"""
```

2. Define a model in `myapp/models.py`.

```
@implementer(myapp.interfaces.IFamily)
class Family(Base, common.IdNameDescriptionMixin):
    pass
```

3. Register the resource in `myapp.main`:

```
config.register_resource('family', Family, IFamily)
```

4. Create templates for HTML views, e.g. `myapp/templates/family/detail_html.mako`,
5. and register these:

```
from clld.web.adapters.base import adapter_factory
...
config.register_adapter(adapter_factory('family/detail_html.mako'), IFamily)
```

Custom maps

The appearance of *Maps* in `clld` apps depends on various factors which can be tweaked for customization:

- the Python code that renders the HTML for the map,
- the GeoJSON data which is passed as map layers,
- the JavaScript code implementing the map.

GeoJSON adapters

GeoJSON in `clld` is just another type of representation of a resource, thus it is created by a suitable adapter, usually derived from `clld.web.adapters.geojson.GeoJSON`.

Map classes

Maps in `clld` are implemented as subclasses of *`clld.web.maps.Map`*. These classes tie together behavior implemented in javascript (based on leaflet) with Python code used to assemble the map data, options and legends.

The following `clld.web.maps.Map.options` are recognized:

name	type	default	description
sidebar	`bool`	`False`	whether the map is rendered in the sidebar
show_labels	`bool`	`False`	whether labels are shown by default
no_showlabels	`bool`	`False`	whether the control to show labels should be hidden
no_popup	`bool`	`False`	whether clicking on markers opens an info window
no_link	`bool`	`False`	whether clicking on markers links to the language page
info_route	`str`	`'language_alt'`	name of the route to query for info window contents
info_query	`dict`	`{}`	query parameters to pass when requesting info window content
hash	`bool`	`False`	whether map state should be tracked via URL fragment
max_zoom	`int`	`6`	maximal zoom level allowed for the map
zoom	`int`	`5`	zoom level of the map
center	`(lat, lon)`	`None`	center of the map
icon_size	`int`	`20 if sidebar else 30`	size of marker icons in pixels
icons	`str`	`'base'`	name of a javascript marker factory function
on_init	`str`	`None`	name of a javascript function to call when initialization is done
base_layer	`str`	`None`	name of a base layer which should be selected upon map load

Custom URLs

When an established database is ported to CLLD it may be necessary to support legacy URLs for its resources (as was the case for WALS). This can be achieved by passing a `route_patterns` dict, mapping route names to custom patterns, in the settings to `clld.web.app.get_configurator()` like in the following example from WALS:

```
def main(global_config, **settings):
    settings['route_patterns'] = {
        'languages': '/languoid',
        'language': '/languoid/lect/wals_code_{id:[^/\.]+}',
    }
    config = get_configurator('wals3', **dict(settings=settings))
```

Downloads

TODO

Misc Utilities

http://www.muthukadan.net/docs/zca.html#utility

- IMapMarker
- ILinkAttrs
- ICtxFactoryQuery

2.1.6 Interfaces

clld makes heavy use of the zope.interfaces and the Zope Component Architecture - in particular in via pyramid's registry - to bind behaviour to objects.

2.1.7 Database

The `clld` database models are declared using SQLAlchemy's declarative extension. In particular we follow the approach of mixins and custom base class, to provide building blocks with enough shared commonality for custom data models.

Declarative base and mixins

class clld.db.meta.**Base**(*jsondata=None*, ***kwargs*)

The declarative base for all our models.

classmethod get(*value*, *key=None*, *default=<NoDefault>*, *session=None*)

Convenience method to query a model where exactly one result is expected.

e.g. to retrieve an instance by primary key or id.

Parameters

- **value** – The value used in the filter expression of the query.
- **key** (*str*) – The key or attribute name to be used in the filter expression. If None is passed, defaults to *pk* if value is `int` otherwise to *id*.

history()
return result proxy to iterate over previous versions of a record.

jsondata = Column(None, JSONEncodedDict(), table=None)
To allow storage of arbitrary key,value pairs with typed values, each model provides a column to store JSON encoded dicts.

jsondatadict
Deprecated convenience function.

Use jsondata directly instead, which is guaranteed to be a dictionary.

pk = Column(None, Integer(), table=None, primary_key=True, nullable=False)
All our models have an integer primary key which has nothing to do with the kind of data stored in a table. 'Natural' candidates for primary keys should be marked with unique constraints instead. This adds flexibility when it comes to database changes.

update_jsondata(***kw*)
Convenience function.

Since we use the simple JSON encoded dict recipe without mutation tracking, we provide a convenience method to update

class clld.db.meta.**CustomModelMixin**
Mixin for customized classes in our joined table inheritance scheme.

Note: With this scheme there can be only one specialized mapper class per inheritable base class.

---

class clld.db.models.common.**IdNameDescriptionMixin**
Mixin for 'visible' objects, i.e. anything that has to be displayed.

In particular all *Resources* fall into this category.

Note: Only one of *clld.db.models.common.IdNameDescriptionMixin.description* or *clld.db.models.common.IdNameDescriptionMixin.markup_description* should be supplied, since these are used mutually exclusively.

---

description = Column(None, Unicode(), table=None)
A description of the object.

id = Column(None, String(), table=None)
A `str` identifier of an object which can be used for sorting and as part of a URL path; thus should be limited to characters valid in URLs, and should not contain '.' or '/' since this may trip up route matching.

markup_description = Column(None, Unicode(), table=None)
A description of the object containing HTML markup.

name = Column(None, Unicode(), table=None)
A human readable 'identifier' of the object.

While the above mixin only adds columns to a model, the following mixins do also add relations between models, thus have to be used in combination, tied together by naming conventions.

class clld.db.models.common.**DataMixin**
Provide a simple way to attach key-value pairs to a model class given by name.

class clld.db.models.common.**HasDataMixin**
Adds a convenience method to retrieve the key-value pairs from data as dict.

Note: It is the responsibility of the programmer to make sure conversion to a `dict` makes sense, i.e. the keys in data are actually unique, thus usable as dictionary keys.

datadict()
return `dict` of associated key-value pairs.

class clld.db.models.common.**FilesMixin**
This mixin provides a way to associate files with instances of another model class.

Note: The file itself is not stored in the database but must be created in the filesystem, e.g. using the create method.

create(*dir_*, *content*)
Write `content` to a file using `dir_` as file-system directory.

Returns File-system path of the file that was created.

mime_type = Column(None, String(), table=None)
Mime-type of the file content.

ord = Column(None, Integer(), table=None, default=ColumnDefault(1))
Ordinal to control sorting of files associated with one db object.

relpath
OS file path of the file relative to the application's file-system dir.

class clld.db.models.common.**HasFilesMixin**
Mixin for model classes which may have associated files.

files
return `dict` of associated files keyed by `id`.

Typical usage looks like

```
class MyModel_data(Base, Versioned, DataMixin):
    pass

class MyModel_files(Base, Versioned, FilesMixin):
    pass

class MyModel(Base, HasDataMixin, HasFilesMixin):
    pass
```

Core models

The CLLD data model includes the following entities commonly found in linguistic databases and publications:

class clld.db.models.common.**Dataset**(*jsondata=None*, ***kwargs*)
Represents a database.

Each project (e.g. WALS, APiCS) is regarded as one dataset; thus, each app will have exactly one Dataset object.

pk
primary key

published
date of publication

publisher_name
publisher

publisher_place
place of publication

class clld.db.models.common.**Language**(*jsondata=None*, ***kwargs*)
Languages are the main objects of discourse.

We attach a geo-coordinate to them to be able to put them on maps.

latitude
geographical latitude in WGS84

longitude
geographical longitude in WGS84

pk
primary key

class clld.db.models.common.**Parameter**(*jsondata=None*, ***kwargs*)
A measurable attribute of a language.

pk
primary key

class clld.db.models.common.**ValueSet**(*jsondata=None*, ***kwargs*)
The intersection of Language, Parameter, and optionally Contribution.

pk
primary key

source
textual description of the source for the valueset

class clld.db.models.common.**Value**(*jsondata=None*, ***kwargs*)
A measurement of a parameter for a particular language.

confidence
textual assessment of the reliability of the value assignment

frequency
Languages may have multiple values for the same parameter. Their relative frequency can be stored here.

class clld.db.models.common.**Contribution**(*jsondata=None*, ***kwargs*)
A set of data contributed within the same context by the same contributors.

pk
primary key

class clld.db.models.common.**Contributor**(*jsondata=None*, ***kwargs*)
Creator of a contribution.

pk
primary key

class clld.db.models.common.**Source**(*jsondata=None*, ***kwargs*)
A bibliographic record, cited as source for some statement.

pk
primary key

class clld.db.models.common.**Unit**(*jsondata=None*, ***kwargs*)
A linguistic unit of a language.

pk
primary key

class clld.db.models.common.**UnitParameter**(*jsondata=None*, ***kwargs*)
A measurable attribute of a unit.

pk
primary key

class clld.db.models.common.**UnitValue**(*jsondata=None*, ***kwargs*)

pk
primary key

validate_parameter_pk(*key*, *unitparameter_pk*)
Validator to sync related parameter.

We have to make sure, the parameter a value is tied to and the parameter a possible domainelement is tied to stay in sync.

Versioning

Versioned model objects are supported via the `clld.db.versioned.Versioned` mixin, implemented following the corresponding SQLAlchemy ORM Example.

Migrations

Migrations provide a mechanism to update the database model (or the data) in a controlled and repeatable way. `clld` apps use Alembic to implement migrations.

Since a migration may change the database schema, it is generally not possible to fully use ORM mechanisms in migration scripts. Instead, migration scripts typically construct SQL to be sent to the database "by hand", or using SQLAlchemy's SQL expression language. Now dropping down to these lower levels of database access makes scripts verbose and error prone. Thus, `clld` provides a module with helpers for Alembic migration scripts. Functionality for alembic scripts.

This module provides

- basic crud functionality within alembic migration scripts,
- advanced helpers for special tasks, like merging sources.

Note: Using the functionality provided in this module is not possible for Alembic scripts supposed to be run in offline mode.

---

class clld.db.migration.**Connection**(*conn*)
A wrapper around an SQLAlchemy connection.

This wrapper provides the convenience of allowing typical CRUD operations to be called passing model classes.

Additionally, it implements more complicated clld domain specific database operations.

A `Connection` will typically be instantiated in an Alembic migration script as follows:

```
from alembic import op
conn = Connection(op.get_bind())
```

all(*model*, ***where*)
return all results of a select statement.

delete(*model*, ***where*)
Run a delete statement.

execute(**args*, ***kw*)
Provide access to the underlying connection's `execute` method.

first(*model*, ***where*)
return first result of a select statement or `None`.

get(*model*, *pk*)
return row specified by primary key.

insert(*model*, ***values*)
Run an insert statement.

Returns primary key of the inserted row.

pk(*model*, *id_*, *attr='id'*)
Get the primary key of an object specified by a unique property.

Parameters

- **model** – model class.
- **id** – Value to be used when filtering.
- **attr** – Column to be used for filtering.

Returns primary key of (first) matching row.

select(*model*, ***where*)
Run a select statement and return a ResultProxy.

set_glottocode(*lid*, *gc*, *gcid=None*)
assign a unique glottocode to a language.

i.e. alternative glottocodes will be deleted.

Parameters

- **lid** – `id` of the language.
- **gc** – Glottocode to be assigned.
- **gcid** – `id` of the `Identifier` instance if one has to be created; defaults to `gc`.

update(*model*, *values*, ***where*)
Run an update statement.

2.1.8 The request object

`clld` registers a custom request factory, i.e. the request object available in view code or templates is an instance of *`clld.web.app.CsomethingRequest`*.

class clld.web.app.**CldRequest**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
Custom Request class.

ctx_for_url(*url*)
Method to reverse URL generation for resources.

I.e. given a URL, tries to determine the associated resource.

Returns model instance or `None`.

dataset
Convenient access to the Dataset object.

Properties of the *clld.db.models.common.Dataset* object an application serves are used in various places, so we want to have a reference to it.

db
Convenient access to the db session.

We make the db session available as request attribute, so we do not have to import it in templates.

get_datatable(*name*, *model*, ***kw*)
Convenient lookup and retrieval of initialized DataTable object.

Parameters

- **name** – Name under which the datatable class was registered.
- **model** – model class to pass as initialization parameter to the datatable.
- **kw** – Keyword parameters are passed through to the initialization of the datatable.

Returns *clld.web.datatables.base.DataTable* instance, if a datatable was registered for `name`.

get_map(*name=None*, ***kw*)
Convenient lookup and retrieval of initialized Map object.

Parameters **name** – Name under which the map was registered.

Returns *clld.web.maps.Map* instance, if a map was registered else `None`.

purl
Access the current request's URL.

For more convenient URL manipulations, we provide the current request's URL as purl.URL instance.

query_params
Convenient access to the query parameters of the current request.

Returns dict of the query parameters of the request URL.

resource_url(*obj*, *rsc=None*, ***kw*)
Get the absolute URL for a resource.

Parameters

- **obj** – A resource or the id of a resource; in the latter case `rsc` must be passed.
- **rsc** – A registered `clld.Resource`.
- **kw** – Keyword parameters are passed through to pyramid.request.Request.route_url

Returns URL

2.1.9 Page components

`clld` supports page components for web apps (i.e. parts of pages which require HTML code and JavaScript to define behavior) with the *clld.web.util.component.Component* virtual base class.

class clld.web.util.component.**Component**
Virtual base class for page components.

Components are objects that can be rendered as HTML and typically define behavior using a corresponding JavaScript object which accepts an options object upon initialization.

get_default_options()
Override this method to define default (i.e. valid across subclasses) options.

Returns JSON serializable dict

get_options()
Override this method to define final-class-specific options.

Returns JSON serializable dict

get_options_from_req()
Override this method to define options derived from request properties.

Returns JSON serializable dict

The design rationale for components is the idea to build the bridge between server and client as cleanly as possible by putting the code to collect options for a client side object and the instantiation of a these objects into one Python class (plus a mako template referenced in this class).

DataTables

DataTables are implemented as Python classes, providing configuration and server-side processing for jquery datatables.

class clld.web.datatables.base.**DataTable**(*req*, *model*, *eid=None*, ***kw*)
DataTables are used to manage selections of instances of one model class.

Often datatables are used to display only a pre-filtered set of items which are related to some other entity in the system. This scenario is supported as follows: For each model class listed in `clld.web.datatables.base.DataTable.__constraints__` an appropriate object specified either by keyword parameter or as request parameter will be looked up at datatable initialization, and placed into a datatable attribute named after the model class in lowercase. These attributes will be used when creating the URL for the data request, to make sure the same pre-filtering is applied.

Note: The actual filtering has to be done in a custom implementation of *clld.web.datatables.base.DataTable.base_query()*.

__init__(*req*, *model*, *eid=None*, ***kw*)
Initialize.

Parameters

- **req** – request object.
- **model** – mapper class, instances of this class will be the rows in the table.
- **eid** – HTML element id that will be assigned to this data table.

base_query(*query*)
Custom DataTables can overwrite this method to add joins, or apply filters.

Returns `sqlalchemy.orm.query.Query` instance.

col_defs()
Must be implemented by derived classes.

Returns list of instances of *clld.web.datatables.base.Col*.

xhr_query()
Get additional URL parameters for XHR.

Returns a mapping to be passed as query parameters to the server when requesting table data via xhr.

class clld.web.datatables.base.**Col**(*dt*, *name*, *get_object=None*, *model_col=None*, *format=None*, ***kw*)
DataTables are basically a list of column specifications.

A column in a DataTable typically corresponds to a column of an sqlalchemy model. This column can either be supplied directly via a model_col keyword argument, or we try to look it up as attribute with name "name" on self.dt.model.

format(*item*)
Called when converting the matching result items of a datatable to json.

get_obj(*item*)
Get the object for formatting and filtering.

Note: derived columns with a model_col not on self.dt.model should override this method.

order()
Called when collecting the order by clauses of a datatable's search query.

search(*qs*)
Called when collecting the filter criteria of a datatable's search query.

Maps

Maps are implemented as subclasses of *clld.web.maps.Map*, providing configuration and server-side processing for leaflet maps.

The process for displaying a map is as follows:

1. In python view code a map object is instantiated and made available to a mako template (either via the registry or directly, as template variable).
2. In the mako template, the render method of the map is called, thus inserting HTML created from the template `clld/web/templates/map.mako` into the page.
3. When the browser renders the page, *CLLD.map()* is called, instantiating a leaflet map object.
4. During initialization of the leaflet map, for each *clld.web.maps.Layer* of the map a leaflet geoJson layer is instantiated, adding data to the map.

class clld.web.maps.**Map**(*ctx*, *req*, *eid=u'map'*)
Represents the configuration for a leaflet map.

__init__(*ctx*, *req*, *eid=u'map'*)
Initialize.

Parameters

- **ctx** – context object of the current request.
- **req** – current pyramid request object.
- **eid** – Page-unique DOM-node ID.

get_layers()
Generate the list of layers.

Returns list or generator of *clld.web.maps.Layer* instances.

class clld.web.maps.**Layer**(*id_*, *name*, *data*, ***kw*)
Represents a layer in a leaflet map.

A layer in our terminology is a FeatureCollection in geojson and a geoJson layer in leaflet, i.e. a bunch of points on the map.

__init__(*id_*, *name*, *data*, ***kw*)
Initialize a layer object.

Parameters

- **id** – Map-wide unique string identifying the layer.
- **name** – Human readable name of the layer.
- **data** – A GeoJSON FeatureCollection either specified as corresponding Python dict or as URL which will serve the appropriate GeoJSON.
- **kw** – Additional keyword parameters are made available to the Layer as instance attributes.

__weakref__
list of weak references to the object (if defined)

CLLD.**map**(*eid*, *layers*, *options*)

Arguments

- **eid** (*string*) – DOM element ID for the map object.
- **layers** (*array*) – List of layer specifications.
- **options** (*object*) – Map options.

Returns CLLD.Map instance.

Adapters

Base classes for adapters.

class clld.web.adapters.base.**Index**(*obj*)
Base class for adapters implementing IIndex.

class clld.web.adapters.base.**Renderable**(*obj*)
Virtual base class for adapters.

Adapters can provide custom behaviour either by specifying a template to use for rendering, or by overwriting the render method.

class clld.web.adapters.base.**Representation**(*obj*)
Base class for adapters implementing IRepresentation.

class clld.web.adapters.base.**SolrDoc**(*obj*)
Document for indexing with Solr encoded in JSON.

2.1.10 Lib

iso

Functionality to gather information about iso-639-3 codes from sil.org.

clld.lib.iso.**get**(*path*)
Retrieve a resource from the sil site and return it's representation.

clld.lib.iso.**get_documentation**(*code*)
Scrape information about a iso 639-3 code from the documentation page.

clld.lib.iso.**get_tab**(*name*)
Generator for entries in a tab file specified by name.

clld.lib.iso.**get_taburls**()
Retrieve the current (date-stamped) file names for download files from sil.

rdf

This module provides functionality for handling our data as rdf.

class clld.lib.rdf.**ClldGraph**(**args*, ***kw*)
Augmented rdflib.Graph.

Augment the standard rdflib.Graph by making sure our standard ns prefixes are always bound.

class clld.lib.rdf.**Notation**(*name*, *extension*, *mimetype*, *uri*)

extension
Alias for field number 1

mimetype
Alias for field number 2

name
Alias for field number 0

uri
Alias for field number 3

clld.lib.rdf.**expand_prefix**(*p*)
Expand default prefixes if possible.

Parameters **p** – a qualified name in prefix:localname notation or a URL.

Returns a string URL or a URIRef

clld.lib.rdf.**properties_as_xml_snippet**(*subject*, *props*)
Serialize props of subject as RDF-XML snippet.

clld.lib.rdf.**url_for_qname**(*qname*)
Expand qname to full URL respecting our default prefixes.

bibtex

Functionality to handle bibligraphical data in the BibTeX format.

See also:

http://en.wikipedia.org/wiki/BibTeX

class clld.lib.bibtex.**Database**(*records*)

Represents a bibtex databases, i.e. a container class for Record instances.

classmethod from_file(*bibFile*, *encoding=u'utf8'*, *lowercase=False*)

Create bibtex database from a bib-file.

@param bibFile: path of the bibtex-database-file to be read.

keymap

Map bibtex record ids to list index.

class clld.lib.bibtex.**EntryType**

Bibtext entry types.

article An article from a journal or magazine. Required fields: author, title, journal, year Optional fields: volume, number, pages, month, note, key

book A book with an explicit publisher. Required fields: author/editor, title, publisher, year Optional fields: volume/number, series, address, edition, month, note, key

booklet A work that is printed and bound, but without a named publisher or sponsoring institution. Required fields: title Optional fields: author, howpublished, address, month, year, note, key

conference The same as inproceedings, included for Scribe compatibility.

inbook A part of a book, usually untitled. May be a chapter (or section or whatever) and/or a range of pages. Required fields: author/editor, title, chapter/pages, publisher, year Optional fields: volume/number, series, type, address, edition, month, note, key

incollection A part of a book having its own title. Required fields: author, title, booktitle, publisher, year Optional fields: editor, volume/number, series, type, chapter, pages, address, edition, month, note, key

inproceedings An article in a conference proceedings. Required fields: author, title, booktitle, year Optional fields: editor, volume/number, series, pages, address, month, organization, publisher, note, key

manual Technical documentation. Required fields: title Optional fields: author, organization, address, edition, month, year, note, key

mastersthesis A Master's thesis. Required fields: author, title, school, year Optional fields: type, address, month, note, key

misc For use when nothing else fits. Required fields: none Optional fields: author, title, howpublished, month, year, note, key

phdthesis A Ph.D. thesis. Required fields: author, title, school, year Optional fields: type, address, month, note, key

proceedings The proceedings of a conference. Required fields: title, year Optional fields: editor, volume/number, series, address, month, publisher, organization, note, key

techreport A report published by a school or other institution, usually numbered within a series. Required fields: author, title, institution, year Optional fields: type, number, address, month, note, key

unpublished A document having an author and title, but not formally published. Required fields: author, title, note Optional fields: month, year, key

class clld.lib.bibtex.**Record**(*genre*, *id_*, **args*, ***kw*)

A BibTeX record is an ordered dict with two special properties - id and genre.

To overcome the limitation of single values per field in BibTeX, we allow fields, i.e. values of the dict to be iterables of strings as well. Note that to support this use case comprehensively, various methods of retrieving values will behave differently. I.e. values will be

- joined to a string in __getitem__,
- retrievable as assigned with get (i.e. only use get if you know how a value was assigned),
- retrievable as list with getall

Note: Unknown genres are converted to "misc".

---

getall(*key*)
Get list of all values for key.

Returns list of strings representing the values of the record for field 'key'.

clld.lib.bibtex.**u_unescape**(*s*)
Unencode Unicode escape sequences.

Match all 3-5-digit sequences with unicode character replace all '?[u....]' with corresponding unicode

There are some decimal/octal mismatches in unicode encodings in bibtex

clld.lib.bibtex.**unescape**(*string*)
Transform latex escape sequences of type 'e into unicode.

Parameters **string** – six.text_type or six.binary_type (which will be decoded using latex+latin1)

Returns six.text_type

coins

Functionality to create Coins, i.e. context objects in spans.

See also:

http://ocoins.info/

class clld.lib.coins.**ContextObject**(*sid*, *mtx*, **data*)
A Context Object which knows how to render it's metadata as HTML span tags.

fmpxml

Functionality to retrieve data from a FileMaker server.

We use the FileMaker *Custom Web Publishing with XML* protocol.

See also:

http://www.filemaker.com/support/product/docs/12/fms/fms12_cwp_xml_en.pdf

class clld.lib.fmpxml.**Client**(*host*, *db*, *user*, *password*, *limit=1000*, *cache=None*, *verbose=True*)
Client for FileMaker's 'Custom Web Publishing with XML' feature.

get(*what*)
Retrieve data from the server.

Parameters **what** – Name of the layout from which to retrieve data.

Returns `list` of `dict` representing the data of the layout.

class clld.lib.fmpxml.**Result**(*content*)
Represents a filemaker pro xml result.

clld.lib.fmpxml.**normalize_markup**(*s*)
normalize markup in filemaker data.

2.1.11 Linked Data

CLLD applications publish Linked Data as follows:

1. VoID description deployed at <base-url>/void.ttl (also via content negotiation)
2. RDF serializations for each resource available via content negotiation or by appending a suitable file extension.
3. dumps pointed to from the VoID description

CLLD core resources provide serializations to RDF+XML via mako templates. This serialization is used as the basis for all other RDF notations. The core templates can be overwritten by applications using standard mako overrides. Custom resources can also contribute additional triples to the core serialization by specifying a __rdf__ method.

Vocabularies

Types

Resources modelled as clld.db.models.common.Language are assigned dcterm's LinguisticSystem class or additionally a subclasses of GOLD's Genetic Taxon or additionally the type skos:Concept.

clld.db.models.common.Source are assigned types from the Bibliographical Ontology.

Design decisions

1. No "303 See other"-type of redirection. While this approach may be suitable to distinguish between real-world objects and web documents, it also blows up the space of URLs which need to be maintained, and raises the requirements for an application serving the linked data (i.e. a simple web server serving static files will no longer do, at least without complicated configuration). Since we want to make sure, that the data of the CLLD project can be made available as Linked Data for as long as possible, minimizing the requirements on the hosting requirement was regarded more important than sticking to the best practice of using "303 See other"-type redirects.

2.1.12 Protocols

In addition to Linked Data, CLLD Apps implement various protocols to embed them firmly in the web fabric.

Sitemaps

view callables implementing the sitemap protocol.

See also:

http://www.sitemaps.org/

clld.web.views.sitemap.**resourcemap**(*req*)
Resource-specific JSON response listing all resource instances.

clld.web.views.sitemap.**robots**(*req*)
robots.txt response listing the sitemaps.

See also:

http://www.sitemaps.org/protocol.html#submit_robots

clld.web.views.sitemap.**sitemap**(*req*)
Resource-specific sitemap.

Note: The resource is looked up using the URL parameter `rsc`.

See also:

http://www.sitemaps.org/protocol.html#xmlTagDefinitions

clld.web.views.sitemap.**sitemapindex**(*req*)
Response listing resource-specific sitemaps.

See also:

http://www.sitemaps.org/protocol.html#index

OAI-PMH for OLAC

Support for the provider implementation of an OLAC OAI-PMH repository.

See also:

http://www.language-archives.org/OLAC/repositories.html

class clld.web.views.olac.**Institution**(*name*, *url*, *location*)

location
Alias for field number 2

name
Alias for field number 0

url
Alias for field number 1

class clld.web.views.olac.**OlacConfig**
Configuration of an applications OLAC repository.

admin(*req*)
Configure the archive participant with role admin.

Note: According to http://www.language-archives.org/OLAC/repositories.html the list of participants > must include the system administrator whose email address is given in the > <oai:adminEmail> element of the Identify response.

Parameters **req** – The current request.

Returns A suitable *Participant* instance or None.

class clld.web.views.olac.**Participant**(*role*, *name*, *email*)

email
Alias for field number 2

name
Alias for field number 1

role
Alias for field number 0

class clld.web.views.olac.**ResumptionToken**(*url_arg=None*, *offset=None*, *from_=None*, *until=None*)
Represents an OAI-PMH resumption token.

We encode all information from a List query in the resumption token so that we do not actually have to keep track of sequences of requests (in the spirit of REST).

clld.web.views.olac.**olac**(*req*)
View implementing the OLAC OAI-PMH repository protocol.

clld.web.views.olac.**olac_with_cfg**(*req*, *cfg*)
Factory function for olac views with different configurations.

If applications want to disseminate metadata for other resources than languages this function can be used to provide a second olac repository.

OpenSearch

TODO

2.1.13 Deployment of CLLD apps

The 'clldfabric' package provides functionality to ease the deployment of CLLD apps. The functionality is implemented as fabric tasks.

Overview

- The target platform assumed by these tasks is Ubuntu 12.04 LTS.
- Source code is transferred to the machines by cloning the respective github repositories.
- Apps are run by gunicorn, monitored by supervisor, behind nginx as transparent proxy.
- PostgreSQL is used as database.

Automation

We use fabric to automate deployment and other tasks which have to be executed on remote hosts.

2.1.14 Tools for CLLD apps

Archiving with ZENODO

The clld.scripts.freeze module provides support for archiving an app and its dataset with ZENODO. Complete archiving workflow.

1. run freeze_func to create a database dump as zip archive of csv files
2. commit and push the dump to the repos
3. run create_release_func to create a release of the repos (thereby triggering the zenodo hook)
4. lookup DOI created by zenodo

5. run `update_zenodo_metadata_func` to update the associated metadata at ZENODO.

`unfreeze_func` can be used to recreate an app's database from a frozen set of csv files.

datahub.io and LLOD

The `clld.scripts.llod` module provides support for creating a full RDF dump of the dataset and registration of the dataset with datahub.io (and the LLOD).

Internet Archive

The `clld.scripts.internetarchive` module provides support for enriching resources of type `Source` with metadata from the Internet Archive, thus enabling easy linking to full texts.

Google book search

TODO

2.1.15 Handling Trees

In this chapter we describe how tree-structured data my be modelled in a CLLD app. We use a technique called closure table to make efficient queries of the form "all descendants of x up to depth y" possible.

As an example we describe how the classification of languoids in Glottolog is modelled.

In the data model we extend the core Language model to include a self-referencing foreign key pointing to the parent in the classification (or Null if the languoid is a top-level family or isolate).

```
@implementer(ILanguage)
class Languoid(Language, CustomModelMixin):
    pk = Column(Integer, ForeignKey('language.pk'), primary_key=True)
    father_pk = Column(Integer, ForeignKey('languoid.pk'))
```

Then we add the closure table.

```
class ClosureTable(Base):
    __table_args__ = (UniqueConstraint('parent_pk', 'child_pk'),)
    parent_pk = Column(Integer, ForeignKey('languoid.pk'))
    child_pk = Column(Integer, ForeignKey('languoid.pk'))
    depth = Column(Integer)
```

Since data in CLLD apps typically does not change often, and if it does, then in a well-defined, hopefully scripted, way, we don't create triggers to synchronize closure table updates with updates of the parent-child relations in the main table, because triggers are typically much more prone to not being portable across databases.

Instead we include the code to update the closure table in the function `myapp.scripts.initializedb.prime_cache` whose explicit aim is to help create de-normalized data.

```
DBSession.execute('delete from closuretable')
SQL = ClosureTable.__table__.insert()

# store a mapping of pk to father_pk for all languoids:
father_map = {r[0]: r[1] for r in DBSession.execute('select pk, father_pk from␣
↪languoid')}
```

```
# we compute the ancestry for each single languoid
for pk, father_pk in father_map.items():
    depth = 1

    # now follow up the line of ancestors
    while father_pk:
        DBSession.execute(SQL, dict(child_pk=pk, parent_pk=father_pk, depth=depth))
        depth += 1
        father_pk = father_map[father_pk]
```

With this setup, we can add a method to Languoid to retrieve all ancestors:

```
def get_ancestors(self):
    # retrieve the ancestors ordered by distance, i.e. from direct parent
    # to top-level family:
    return DBSession.query(Languoid)\
        .join(TreeClosureTable, and_(
            TreeClosureTable.parent_pk == Languoid.pk,
            TreeClosureTable.depth > 0))\
        .filter(TreeClosureTable.child_pk == self.pk)\
        .order_by(TreeClosureTable.depth)
```

2.1.16 Advanced configuration

This chapter describes somewhat more advanced techniques to configure a `clld` app.

Custom map icons

`clld` uses leaflet to display maps. Thus, techniques to use custom map markers are based on corresponding mechanisms for leaflet.

Using custom leaflet markers with `clld` requires the following steps:

1. Define a javascript function in your app's `project.js` which can be used as marker factory; the signature of this function must be as follows:

`MYAPP.icon_factory` (*feature*, *size*)

Arguments

- **feature** – GeoJSON feature object.
- **size** – Size in pixels of the marker.

Returns L.Icon instance.

2. Make this function available to `clld` by assigning it to a name in `CLLD.MapIcons`:

```
CLLD.MapIcons['myname'] = MYAPP.icon_factory;
```

3. Configure a map to use the custom icons:

```
class MyMap(clld.web.maps.Map):
    def get_options(self):
        return {
            'icons': 'myname',
        }
```

The name passed as map options will be used to look up the function. This function will be called for each feature object encountered in the GeoJSON object defining a map's content, i.e. if you want to use special properties of a language or a parameter value in your algorithm to compute the appropriate marker, you will probably have to define a custom GeoJSON adapter for the map as well (see *GeoJSON adapters*).

A full example to create custom icons which display a number on top of a standard icon could look as follows:

1. In `myapp/static/project.js` add

```
MYAPP.NumberedDivIcon = L.Icon.extend({
    options: {
        number: '',
        className: 'my-div-icon'
    },
    createIcon: function () {
        var div = document.createElement('div');
        var img = this._createImg(this.options['iconUrl']);
        $(img).width(this.options['iconSize'][0]).height(this.options['iconSize'][1]);
        var numdiv = document.createElement('div');
        numdiv.setAttribute ( "class", "number" );
        $(numdiv).css({
            top: -this.options['iconSize'][0].toString() + 'px',
            left: 0 + 'px',
            'font-size': '12px'
        });
        numdiv.innerHTML = this.options['number'] || '';
        div.appendChild (img);
        div.appendChild (numdiv);
        this._setIconStyles(div, 'icon');
        return div;
    }
});

CLLD.MapIcons['numbered'] = function(feature, size) {
    return new MYAPP.NumberedDivIcon({
        iconUrl: url == feature.properties.icon,
        iconSize: [size, size],
        iconAnchor: [Math.floor(size/2), Math.floor(size/2)],
        popupAnchor: [0, 0],
        number: feature.properties.number
    });
}
```

2. In `myapp/static/project.css` add

```
.my-div-icon {
    background: transparent;
    border: none;
}

.leaflet-marker-icon .number{
    position: relative;
    font-weight: bold;
    text-align: center;
    vertical-align: middle;
}
```

2.1.17 Design

The main challenge for the `clld` framework is to balance abstraction and concreteness.

The following goals directed the design:

- There must be a core database model, which allows for as much shared functionality as possible. In particular, publication of Linked Data and integration with services such as OLAC must be implemented by the framework.
- Deployment of `clld` applications must be uniform and easy.
- User interfaces of applications for browsers must be fully customizable.
- It must be easy to re-implement legacy applications using the framework.

These constraints led to the following design decisions:

- We target Ubuntu 12.04 with postgresql 9.1 and python 2.7 as primary deployment platform. As of version 0.13 `clld` does also work with python 3.4, the version of python3 that comes packaged with Ubuntu 14.04.
- Use sqlalchemy and it's implementation of joined table inheritance to provide a core database model that can easily be extended.
- Use the pyramid framework for its extensible configuration mechanism and support of the Zope component architecture (zca).
- Use zca for pluggable functionality.
- Allow UI customization via i18n and overrideable templates.

CHAPTER 3

The applications

For examples of applications developed on top of the `clld` framework see the list of CLLD datasets.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

U

V

X