c2cgeoform Documentation

Release 2.0

Camptocamp

Dec 17, 2021

Contents:

1 Prerequisites **1**

2 User guide **3**
- 2.1 Creating a c2cgeoform project . . . 3
- 2.2 Defining the model for a form . . . 4
- 2.3 Create the views for your model . . . 5
- 2.4 Configure the grid . . . 6
- 2.5 Understanding the schemas . . . 7
- 2.6 Configure the widgets . . . 9
- 2.7 Using custom templates . . . 9
- 2.8 Writing tests . . . 10
- 2.9 Internationalization . . . 10

3 Developer guide **11**
- 3.1 Clone the project . . . 11
- 3.2 Run the checks . . . 11
- 3.3 Run the tests . . . 11
- 3.4 Serve the c2cgeoform_demo project . . . 12
- 3.5 Deploy the c2cgeoform_demo on demo server . . . 12

4 Indices and tables **15**

Python Module Index **17**

Index **19**

CHAPTER 1

Prerequisites

The following system packages must be installed on your system:

- `python3-dev`
- `python-virtualenv`
- `libpq-dev` (header files for PostgreSQL)
- `gettext`

On Windows, you should install `make` using Cygwin (and put the bin folder into the path). For Python, please use Python >= 3.x.

You will also need NodeJS which can be installed by NVM : https://github.com/creationix/nvm#install-script

CHAPTER 2

User guide

2.1 Creating a c2cgeoform project

This page describes how to create a c2cgeoform project. A c2cgeoform project is basically a Pyramid project with c2cgeoform enabled in the project.

2.1.1 Install c2cgeoform

```
git clone git@github.com:camptocamp/c2cgeoform.git
cd c2cgeoform
make build
```

On Windows, you should use the https way to clone the repository:

```
git clone https://github.com:camptocamp/c2cgeoform.git
```

2.1.2 Create a Pyramid project using c2cgeoform scaffold

Note that if PYTHONPATH does not exists as an environment variable, template files (*_tmpl) are not rendered in new project folder.

```
export PYTHONPATH=$PYTHONPATH
.build/venv/bin/pcreate -s c2cgeoform ../c2cgeoform_project
```

2.1.3 Initialize a git repository

Make your new project folder a git repository.

```
cd ../c2cgeoform_project
git init
git add .
git commit -m 'Initial commit'
```

2.1.4 Install the project and its dependencies

```
make build
```

2.1.5 Set up database

First of all you need to have a PostGIS database for the project. Create the database:

```
sudo -u postgres psql -c "CREATE USER \"www-data\" WITH PASSWORD 'www-data';"

sudo -u postgres createdb c2cgeoform_project
sudo -u postgres psql -d c2cgeoform_project -c 'CREATE EXTENSION postgis;'
sudo -u postgres psql -c 'GRANT ALL ON DATABASE c2cgeoform_project TO "www-data";'
```

When you do have a Postgres role and a PostGIS database edit the `development.ini` and `production.ini` files and set `sqlachemy.url` appropriately. For example:

```
sqlalchemy.url = postgresql://www-data:www-data@localhost:5432/c2cgeoform_project
```

Now create the tables:

```
make initdb
```

Note that this will launch the python script `c2cgeoform_project/scripts/initializedb.py`. You will have to customize this thereafter.

2.1.6 Run the development server

You are now ready to run the application:

```
make serve
```

Visit the following URLs to verify that the application works correctly: http://localhost:6543/excavations/new and http://localhost:6543/excavations.

2.2 Defining the model for a form

The underlying schema for a `c2cgeoform` form is defined as a SQLAlchemy model. A simple definition is shown below:

```
from sqlalchemy import (Column, Integer, Text)
import deform
from uuid import uuid4
```

(continues on next page)

(continued from previous page)

```
from c2cgeoform.models import Base


class Comment(Base):
    __tablename__ = 'comments'
    __colanderalchemy_config__ = {
        'title': 'A very simple form'
    }

    id = Column(Integer, primary_key=True, info={
        'colanderalchemy': {
            'widget': deform.widget.HiddenWidget()
        }})

    hash = Column(Text, unique=True, default=lambda: str(uuid4(), info={
        'colanderalchemy': {
            'widget': HiddenWidget()
        }})

    name = Column(Text, nullable=False, info={
        'colanderalchemy': {
            'title': 'Name'
        }})

    comment = Column(Text, nullable=True, info={
        'colanderalchemy': {
            'title': 'Comment',
            'widget': deform.widget.TextAreaWidget(rows=3),
        }})
```

This SQLAlchemy model is enriched with properties for ColanderAlchemy, for example to set a title for a field, use a specific Deform widget or use a Colander validator.

In general, every SQLAlchemy model can be used as schema for a form. The only requirements are:

- The model class must contain exactly one primary key column. Tables with composite primary keys are not supported.

A more complex example for a model can be found here. For more information on how to define the model, please refer to the SQLAlchemy, ColanderAlchemy, Colander and Deform documentations.

2.3 Create the views for your model

There is already a views class created in your project by the scaffold, see file `views/excavation.py`. Let's have a look on that file content.

To ease creation of views classes, c2cgeoform comes with an abstract class that contains base methods to display grids, render forms and save data. This is why ExcavationViews extends AbstractViews for a specific SQLAlchemy model and colander schema:

```
@view_defaults(match_param='table=excavations')
class ExcavationViews(AbstractViews):

    _model = Excavation
    _base_schema = base_schema
```

Also note the `@view_defaults` which says that all the views declared in this class will only apply when the route parameter named `table` will be equal to `"excavation"`. The routes given by c2cgeoform have the following form:

- c2cgeoform_index: `{table}`
- c2cgeoform_grid: `{table}/grid.json`
- c2cgeoform_item: `{table}/{{id}}`
- c2cgeoform_item_duplicate: `{table}/{{id}}/duplicate`

Those routes are registered in the pyramid config by the `routes` module (see the `routes.py` file situated at the root of the generated project).

```
register_models(config, [
    ('excavations', Excavation)
```

To select records through urls, we also need a unique field, this is given by:

```
_id_field = 'hash'
```

And to show the table records grid we need a definition per column:

```
_list_fields = [
    _list_field('reference_number'),
    _list_field('request_date'),
    ...
]
```

Finally we need a method for each view, for a typical use case, we could have 6 views:

- `index`: Return HTML page with the grid.
- `grid`: Return records as JSON for the grid.
- `edit`: Show create or edit form for the specified record.
- `duplicate`: Show duplication form for the specified record.
- `delete`: Delete the specified record.
- `save`: Save new record or modifications to existing record.

In a typical use case, those views will only call the super class method with the same name.

2.4 Configure the grid

Grid columns can be configured using the `_list_fields` property of the views class, which is an ordered list of `ListField` objects, one for each column.

The `ListField` constructor take some parameters:

- `model`: the SQLAlchemy mapper (required if attr is an attribute name).
- `attr`: the model attribute name to use or an SQLAlchemy InstrumentedAttribute.
- `key`: an identifier for the column, default to `attribute.key`.
- `label`: text for the column header, default to colanderalchemy title for the field.
- `renderer`: callable that takes an entity of the SQLAlchemy mapper and returns a string value.

- `sort_column`: An `IntrumentedAttribute` to use in `sort_by`.
- `filter_column`: An `IntrumentedAttribute` to filter with.
- `visible`: a boolean for the initial visible state of this column.

Every time the table index page asks for data from the grid view, the `AbstractView` will create a default query using `AbstractViews._base_query` method.

If you use columns coming from relationships, this might result in sending one request to the database for each relationship and each record. In such cases, you should override the `_base_query` method to use eager loading for those relationships, for example:

```
def _base_query(self):
    return self._request.dbsession.query(Excavation).distinct(). \
        join('situations'). \
        options(subqueryload('situations'))
```

Note that you also need to `join` the relationships you use for sorting and filtering.

2.5 Understanding the schemas

ColanderAlchemy allows creating Colander schemas directly from SQLAlchemy model classes.

Additionally, c2cgeoform provides its own classes with extended features. A basic use case schema creation will look like:

```
from model import MyClass
schema = GeoFormSchemaNode(MyClass)
```

See the following API to understand what is going on behind the scene.

class c2cgeoform.schema.**GeoFormSchemaNode**(**args*, ***kw*)

An SQLAlchemySchemaNode with deferred request and dbsession properties. This will allow defining schemas that requires the request and dbsession at module-scope.

Example usage:

```
schema = GeoFormSchemaNode(MyModel)

def create_form(request, dbsession):
    return Form(
        schema = schema.bind(
            request=request,
            dbsession=request.dbsession),
        ...
    )
```

add_unique_validator(*column*, *column_id*)

Adds an unique validator on this schema instance.

column SQLAlchemy ColumnProperty that should be unique.

column_id SQLAlchemy MapperProperty that is used to recognize the entity, basically the primary key ColumnProperty.

class c2cgeoform.schema.**GeoFormManyToManySchemaNode**(*class_*, *includes=None*, **args*, ***kw*)

A GeoFormSchemaNode that properly handles many to many relationships.

includes: Default to primary key name(s) only.

objectify(*dict_*, *context=None*)
Method override that returns the existing ORM class instance instead of creating a new one.

c2cgeoform.schema.**manytomany_validator**(*node*, *cstruct*)
Validator function that checks if `cstruct` values exist in the related table.

Note that entities are retrieved using only one query and placed in SQLAlchemy identity map before looping on `cstruct`.

class c2cgeoform.ext.colander_ext.**BinaryData**
A Colander type meant to be used with `LargeBinary` columns.

Example usage

```
class Model():
    id = Column(Integer, primary_key=True)
    data = Colum(LargeBinary, info={
        'colanderalchemy': {
            'typ': colander_ext.BinaryData()
        }})
```

It is usually not used directly in application models, but through the `c2cgeoform.models.FileData` mixin, which is meant to be used with a `deform_ext.FileUploadWidget`.

The serialize method just returns `colander.null`. This is because the `FileUploadWidget`'s template does not use and need the binary data.

The deserialize method gets a Python `file` object and returns a bytes string that is appropriate for the database.

deserialize(*node*, *cstruct*)
In Colander speak: Converts a serialized value (a cstruct) into a Python data structure (a appstruct). Or: Converts a Python file stream to plain binary data.

serialize(*node*, *appstruct*)
In Colander speak: Converts a Python data structure (an appstruct) into a serialization (a cstruct).

class c2cgeoform.ext.colander_ext.**Geometry**(*geometry_type='GEOMETRY'*, *srid=-1*, *map_srid=-1*)
A Colander type meant to be used with GeoAlchemy 2 geometry columns.

Example usage

```
geom = Column(
    geoalchemy2.Geometry('POLYGON', 4326, management=True), info={
        'colanderalchemy': {
            'typ': colander_ext.Geometry(
                'POLYGON', srid=4326, map_srid=3857),
            'widget': deform_ext.MapWidget()
        }})
```

Attributes/Arguments

geometry_type The geometry type should match the column geometry type.

srid The SRID of the geometry should also match the column definition.

map_srid The projection used for the OpenLayers map. The geometries will be reprojected to this projection.

deserialize (*node*, *cstruct*)
In Colander speak: Converts a serialized value (a cstruct) into a Python data structure (a appstruct). Or: Converts a GeoJSON string into a *WKBElement*.

serialize (*node*, *appstruct*)
In Colander speak: Converts a Python data structure (an appstruct) into a serialization (a cstruct). Or: Converts a *WKBElement* into a GeoJSON string.

2.6 Configure the widgets

All Deform widgets can be used with `c2cgeoform`. See the Deform examples and widgets API reference for detailed description about available options.

Additionally, c2cgeoform provides some extra widgets:

2.7 Using custom templates

c2cgeoform distinguishes two types of templates: **views** templates and **widget** templates. - Views templates are used directly by Pyramid and provide the site structure. - Widgets templates are used by Deform to render the forms.

2.7.1 Default views templates

The default c2cgeoform views templates are located in the `templates` folder and use jinja2 syntax.

c2cgeoform comes with partial templates that are included in views templates of your project.

2.7.2 Overriding widgets templates globally

Deform widget templates are located in the `templates/widgets` folder and use the chameleon syntax.

At rendering time, Deform will search folders for the templates in order they appear in Form renderer `search_path` property. c2cgeoform configure it to:

```
default_search_paths = (
    resource_filename('c2cgeoform', 'templates/widgets'),
    resource_filename('deform', 'templates'))
```

But you can add you own widgets folder, in your package `__init__.py` file before including `c2cgeoform` using:

```
import c2cgeoform
search_paths = (
    (resource_filename(__name__, 'templates/widgets'),) +
    c2cgeoform.default_search_paths
)
c2cgeoform.default_search_paths = search_paths
```

To overwrite globally the Deform templates or the templates coming from `c2cgeoform` (like the map widget), you just need to copy the template to your application `templates/widgets` folder.

2.7.3 Use a custom template for a form or a specific widget in a form

Both the form main template and widget templates can be changed locally for a given model by giving a `template` property to the `Widget`.

```
base_schema = GeoFormSchemaNode(
    Comment,
    widget=FormWidget(template='comment'))
```

Note that it is possible to create a layout for the form fields without completely overriding the form template by giving a `fields_template` to the form schema.

```
base_schema = GeoFormSchemaNode(
    Comment,
    widget=FormWidget(fields_template='comment_fields'))
```

Here is the default one: https://github.com/camptocamp/c2cgeoform/blob/master/c2cgeoform/templates/widgets/mapping_fields.pt

2.8 Writing tests

2.9 Internationalization

CHAPTER 3

Developer guide

This page describes how to set up the development environment for working on c2cgeoform. It is for developers working on c2cgeoform itself, not for developers working on c2cgeoform-based applications.

Note that c2cgeoform is a framework with a Pyramid scaffold used to create c2cgeoform-based applications. This scaffold produce a fully functional c2cgeoform-base project: the c2cgeoform_demo project.

When running code checks and tests, these jobs are first run on the c2cgeoform framework itself. Then the c2cgeoform_demo project is generated in *.build* folder. Finally, the checks and tests are launched in this project.

Note that you should never alter the c2cgeoform_demo project itself but the c2cgeoform scaffold and regenerate the c2cgeoform_demo project.

3.1 Clone the project

```
git clone git@github.com:camptocamp/c2cgeoform.git
cd c2cgeoform
```

3.2 Run the checks

```
make check
```

3.3 Run the tests

Create the tests database:

```
sudo -u postgres psql -c "CREATE USER \"www-data\" WITH PASSWORD 'www-data';"

export DATABASE=c2cgeoform_demo_tests
sudo -u postgres psql -d postgres -c "CREATE DATABASE $DATABASE OWNER \"www-data\";"
sudo -u postgres psql -d $DATABASE -c "CREATE EXTENSION postgis;"
```

Run the framework and demo tests:

```
make test
```

3.4 Serve the c2cgeoform_demo project

You need to create a PostGIS database. For example:

```
export DATABASE=c2cgeoform_demo
sudo -u postgres psql -d postgres -c "CREATE DATABASE $DATABASE OWNER \"www-data\";"
sudo -u postgres psql -d $DATABASE -c "CREATE EXTENSION postgis;"
make initdb
```

Run the development server:

```
make serve
```

You can now open the demo project in your favorite browser: http://localhost:6543/

And there you go, you're ready to develop, make changes in c2cgeoform, run checks and tests in c2cgeoform. And finally see the results in c2cgeoform demo application.

3.5 Deploy the c2cgeoform_demo on demo server

Prepare the demo project:

```
# open a ssh connection with the GMF 2.3 server
ssh -A geomapfish-demo.camptocamp.com

# clone the c2cgeoform repository
cd /var/www/vhosts/geomapfish-demo/private
git clone git@github.com:camptocamp/c2cgeoform.git

# generate the c2cgeoform_demo project with mod_wsgi related files
APACHE_ENTRY_POINT=c2cgeoform make modwsgi
```

Create the database as to serve the development version, see: *Serve the c2cgeoform_demo project*

Include the demo project in Apache virtual host configuration:

```
echo "IncludeOptional $PWD/.build/c2cgeoform_demo/.build/apache.conf" > /var/www/
↪vhosts/geomapfish-demo/conf/c2cgeoform_demo.conf
sudo apache2ctl configtest
```

If everything goes fine, restart apache:

```
sudo apache2ctl graceful
```

You can now open the demo project in your favorite browser: https://geomapfish-demo.camptocamp.com/c2cgeoform/

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

c2cgeoform.ext.colander_ext, 8
c2cgeoform.schema, 7

Index

A

add_unique_validator() (*c2cgeoform.schema.GeoFormSchemaNode method*), 7

B

BinaryData (*class in c2cgeoform.ext.colander_ext*), 8

C

c2cgeoform.ext.colander_ext (*module*), 8
c2cgeoform.schema (*module*), 7

D

deserialize() (*c2cgeoform.ext.colander_ext.BinaryData method*), 8
deserialize() (*c2cgeoform.ext.colander_ext.Geometry method*), 8

G

GeoFormManyToManySchemaNode (*class in c2cgeoform.schema*), 7
GeoFormSchemaNode (*class in c2cgeoform.schema*), 7
Geometry (*class in c2cgeoform.ext.colander_ext*), 8

M

manytomany_validator() (*in module c2cgeoform.schema*), 8

O

objectify() (*c2cgeoform.schema.GeoFormManyToManySchemaNode method*), 8

S

serialize() (*c2cgeoform.ext.colander_ext.BinaryData method*), 8
serialize() (*c2cgeoform.ext.colander_ext.Geometry method*), 9