pyramid_crud Documentation

Release 0.1.3

Florian Rüchel

June 21, 2016

Contents

1 Links **3**

2 Table of Contents **5**

2.1 Introduction . . . 5
2.1.1 Installation . . . 5
2.1.2 A Word on Dependencies . . . 5
2.1.3 QuickStart . . . 6
2.2 Usage . . . 6
2.2.1 Configuration . . . 7
2.2.2 Views . . . 7
2.2.3 Forms . . . 15
2.2.4 Templates . . . 20
2.2.5 Utility Functions . . . 23
2.2.6 Adding Actions to Forms . . . 23
2.2.7 Help Topics . . . 26
2.2.8 Examples . . . 28
2.3 Development . . . 28
2.3.1 Building Documentation . . . 28
2.3.2 Running Tests . . . 28
2.3.3 Contributing . . . 29

3 Indices and tables **31**

Python Module Index **33**

This software is a framework with the attempt to replicate a behavior similar to Django's Generic Views and Admin Pages.

It aims to provide a simple yet configurable interface to get a CRUD (Create, Read, Update, Delete) interface on persisted data.

This library is an **unofficial** extension to Pyramid. This is not likely to change unless the libraries dependencies are decoupled as described in A Word on Dependencies.

Note: This library is in an early phase and contributions are welcome that fix bugs or add missing features. Just please make sure to keep it as clean as possible. Also always take a look at how Django achieves the desired functionality (if present), because they have some good ideas on keeping the code clean and readable.

CHAPTER 1

Links

- Documentation
- Source Code
- Package on PyPI

CHAPTER 2

Table of Contents

2.1 Introduction

2.1.1 Installation

You can install `pyramid_crud` using pip:

```
pip install pyramid_crud
```

Or you can fetch the current sources and install it manually:

```
git clone https://github.com/Javex/pyramid_crud
cd pyramid_crud
python setup.py install
```

2.1.2 A Word on Dependencies

This library currently relies on certain other libraries. Therefore, it only supports a certain use-case (or at least only integrates well there). Thus, if your application stack differs from the libraries listed below, make sure to read this section to see which parts can be changed and which cannot.

- Pyramid
- WTForms
- SQLAlchemy
- WTForms-Alchemy
- Mako

The Mako integration is very loose, allowing for arbitrary templates to be used as long as they are registered properly with Pyramid.

WTForms on the other hand is more tightly integrated. It should be easily possible to write an adapter that replicates the WTForms interface and allows integration with other form libraries but this library was not designed for it. However, I am happy to accept pull requests that change this behavior to allow arbitrary form libraries as long as the code stays clean and the interface does not require major changes. Finally, there is no requirement for you to use WTForms in the rest of your application: You can simply rely on WTForms only for this library. As long as you don't deviate from the default mechanisms you will not even have to concern yourself with WTForms at all.

SQLAlchemy is also very tightly bound to the library. Both the form and the view part rely on SQLAlchemy and its interface. However, seeing as SQLAlchemy is basically *the* go-to ORM outside of Django, I don't see a need except if NoSQL databases are desired.

Pyramid is, of course, at the core of this library and there are currently no plans to decouple it to allow arbitrary frameworks the usage of this library. Again, I accept pull requests for this, but I find it much more likely that a split into a new library that provides this functionality independent of a web framework and separate integration into different frameworks is the way to go if this is desired. If you want to work on something like this, please contact me, so we can coordinate on this.

2.1.3 QuickStart

For this quickstart we assume you already have an application with models that you want to enable CRUD for.

First you have to include `pyramid_crud` in your `.ini` file:

```
pyramid.includes =
    ...
    pyramid_crud
    ...
```

```
from pyramid_crud.forms import CSRFModelForm
from pyramid_crud.views import CRUDView
from .models import MyModel


class MyModelForm(CSRFModelForm):
    class Meta:
        model = MyModel

class MyModelView(CRUDView):
    form = MyModelForm
    url_path = '/mymodel'
```

That gets you started: We create a form and a set of views for our form. Now start your application and visit the application on the path `/mymodel`. You should see a list of present instances and also buttons to delete them and add new instances. Finally, you can also click the first columns element to edit an item. Go ahead, play around with it. Afterwards, you can head to *Usage* and start configuring the associated parts to behave the way you need it to.

2.2 Usage

After you have read the *QuickStart* you can now head into the special configuration. This chapter is split into four sections. *Views* are the center of configuration. Here you define which form to use, how templates are located and under which route it should be available. Views also needs *Forms*. These define the associated model and configure which fields are displayed. Finally, if the default templates don't suit you (e.g. you want to integrate your own style or you don't use Mako), you can visit the *Templates* section to see how to change the default templates.

There are also some *Examples* that show possible applications. You can refer there to see how your goal can be realized in pratice. To change the global behavior of the library refer to *Configuration*. This is useful if the default global application behavior does not suit you.

2.2.1 Configuration

There are several global settings with which you can configure the behavior of this library. All settings use the prefix `crud.`

Static View URL Prefix

The application needs to serve static assets to display the default templates properly (specifically, it uses Bootstrap). These assets need their own prefix to avoid routing conflicts with your other static files. Thus, this setting allows you to define a custom prefix. By default, it is `/static/crud` which should be fine for most applications (as `static` is a very common name, you can have all your CSS and JS files under this). However, if this does not fit your use case, use this setting to change it.

If this is `None`, no additional static view will be registered. This is useful if you roll your own theme anyway (see *Create a Complete Theme*) and you set up your own static views for it.

Config File Setting Name
`crud.static_url_prefix`

2.2.2 Views

Add a New View

Configuration

The main configuration of the library is done on the view. By subclassing *`CRUDView`* for each new view you can create an individual configuration that turns your model & form into a fully accessible CRUD interface. The available confiugration parameters are described on the class:

class `pyramid_crud.views.`**`CRUDView`**(*request*)

The base class for all views. Subclassing directly from this gets you a new view configuration for a single model & form. If you specify `__abstract__` on it, the class will not be configured at all and you can use it as your own base class.

Note: Configuration is done by Pyramid the moment you call `pyramid.config.Configurator.scan()` in a way similar to what the `pyramid.view.view_config` decorator does. If you want to completely disable this behavior, set *view_configurator_class* to `None`. Then no route configuration will be done and you have to set up views and routes yourself. This is an advanced technique not recommended for beginners.

The following attributes can be defined to override behavior of the view:

Form Mandatory argument that specifies the form class for which this view should be created. This must be a form as described in *Forms*.

url_path Mandatory arguments if the default *view_configurator_class* is used. It determines the base path under which this view should be available.

So for example, if this is `/myitems` then the list view will be reached under the `/myitems` path whereas the new view will be under `/myitems/new`.

How and if this parameter is used depends entirely on the implementation of the configurator but it is recommended to keep this parameter for custom implementations as well.

dbsession Return the current SQLAlchemy session. By default this expects a `dbsession` attribute on the `request` object. It is **mandatory** that you either attach the attribute using an event or override this attribute (you can use a `property` if you like).

list_display A tuple if items which should be displayed on the list view. By default a single column of the models `__str__` method is used. There are several possibilities of what you might specify here (the options will be tried in this order):

- A string representing an attribute or callable on the model. If this attribute is callable, it will be called and get no additional arguments (the first argument will already be `self`, the model instance).

  For example, with a normal field on the model:

```
class Model(Base):
     id = Column(Integer, primary_key=True,
                 info={'label': 'ID'})

class View(CRUDView):
     list_display = ('id',)
```

  In this example there will be a single column in the list view. Its title will be "ID" and its value will be the value of the `id` field in the database.

  Similarly, with a callable:

```
class Model(Base):
    id = Column(Integer, primary_key=True)

    def id_plus_one(self):
        return self.id + 1
    id_plus_one.info = {'label': 'ID+1'}

class View(CRUDView):
    list_display = ('id_plus_one',)
```

- A generic callable function. This function will be called with a single argument: The instance of the model. For example:

```
class Model(Base):
    id = Column(Integer, primary_key=True)

def id_plus_one(obj):
    return obj.id + 1
id_plus_one.info  = {'label': 'ID+1'}

class View(CRUDView):
    list_display = (id_plus_one,)
```

- A string representing a method on the view. This will behave in the same way as for the function callable above except that it must be a string. For example:

```
class Model(Base):
    id = Column(Integer, primary_key=True)

class View(CRUDView):
    list_display = ('id_plus_one',)
```

```
            def id_plus_one(self, obj):
                return obj.id + 1
            id_plus_one.info = {'label': 'ID+1'}
```

Some additional notes on the way this attribute behaves:

- Some additional configuration is possible on each attribute, regardless of how it is specified. For information on this see *The Info Dictionary*.
- A class `columnn-<attr-name>` is placed on each on each of the <th> fields in the column heading to allow application of CSS attributes, e.g. to set the width of a column.
- If the attribute `info` cannot be found on the attribute (at the class level, not instance level), default value is determined as the column heading. If name of the column is `__str__` then the name of the model class is fetched. If it is directly callable (in case of a generic callable function), then the name of the function is used. In all other cases the provided string is used. To make for a prettier format, it additionally replaces any underscores by spaces and captializes each word.

list_display_links Specify which of the displayed columns should be turned into links that open the edit view of that instance. By default, the first column is used.

This should be any kind of iterable, preferrably a tuple or set for performance reasons.

Example:

```
    class MyView(CRUDView):
        list_display = ('column1', 'column2', 'column3')
        list_display_links = ('column1', 'column3')
```

This configuration will turn the columns `column1` and `column3` into links.

actions: An optional list of action callables or view method names for the dropdown menu. See *Adding Actions to Forms* for details on how to use it.

theme A theme is just a collection of template files inside a directory and this is the name of that directory. The recommended way is to use asset specification to unambigously identify the package. By default the bootstrap template is used and so this is set to `pyramid_crud:templates/mako/bootstrap`. If you want to roll your own theme, you can overwrite this. But if you only want to copy a single template and modify it, you should check out *Templates*.

template_ext Which file extension to use for templates. By default, Mako templates are used and so the extension is `.mako` but any renderer that is recognized by pramid can be used.

template_* You can specify any name here, e.g. `template_list` and the *`CRUDView.get_template_for()`* method will use this when calling it with `list` as the action parameter. This is useful for overwriting specific templates but keeping the default behavior for the rest.

Note: The name "ext" for an action is thus not allowed (as `template_ext` is another configuration). Just don't define an action with that name.

This way is also impossible for templates in subdirectories, for example `fieldsets/horizontal.mako` since a slash ("/") cannot be used on a path. Currently the only way is to overwrite *`CRUDView.get_template_for()`*.

---

view_configurator_class A class that configures all views and routes for this view class. The default implementation is *ViewConfigurator* which covers basic route & view configuration. However, if you need more advanced functionalities like, for example, permissions, you can change this parameter. See the documentation on *ViewConfigurator* for details on how to achieve that.

There are also some attributes which you can access. All of them are available on the instance, but only some are also available on the class (in this case, it is noted on the attribute).

routes A dictionary mapping action names to routes. Action names are such as `list` or `edit` and they all have unique route names that can be given to `request.route_url`. You can use it like this:

```
url = request.route_url(view.routes["list"])
```

This will return a URL to the list view.

The routes dictionary is populated by the *view_configurator_class*.

This can be accessed at the class and instance level.

request The current request, an instance of `pyramid.request.Request`.

View & Route Setup

Setting up views and routes is delegated to a special configurator class that creates a route & view for each available view, i.e. list, edit, new and delete. Since you often need to change the routes and views to match your needs, you can subclass this and start overwriting its behavior. The interface is very simple:

Note: There is a slight overhead to configuring views like this because it requires the creation of an additional class. However, approaches like configuring parameters directly on the view are inflexible and setting awkward callables (in theory the most pythonic way) look ugly. Thus, this method is both flexible and easy to read.

---

class `pyramid_crud.views.`**`ViewConfigurator`**(*config*, *view_class*)

The standard implementation of the view configuration. It performs the most basic configuration of routes and views without any extra functionality.

This is sufficient in many cases, but there are several applications where you might want to completely or partially change this behavior. Any time you want to pass additional arguments to `pyramid.config.Configurator.add_route()` or `pyramid.config.Configurator.add_view()` you can just subclass this and override the specific methods.

All the public methods must always be implemented according to their documentation or the configuration of views and routes will fail. If you are unsure, you can take a look at the default implementation. It is just a very thin wrapper around the above mentioned methods.

During instantiation the arguments `config` representing an instance of `pyramid.config.Configurator` and `view_class` being your subclassed view class are given to the instance and stored under these values as its attributes.

From the `view_class` parameter you can access the complete configuration as documented on *CRUDView*. `config` should then be used to add routes and views and possibly other configuration you might need.

`ViewConfigurator.`**`configure_list_view`**`()`

Configure the "list" view by setting its route and view. This method must call `add_view` to configure the view and `add_route` to connect a route to it. Afterwards, it must return the name of the configured route that links route and view. This will then be stored in the view's `route` dictionary under the "list" key.

```
    def configure_list_view(self):
        self.config.add_view('myview-list',
                             renderer='list.mako',)
        self.config.add_route('myview-list', self.view_class.url_path)
        return 'myview-list'
```

This does a few things:

- It sets up the view under the alias `myview-list` with the template `list.mako`. Note that the default configuration uses a theme and absolute paths while this configures a template that needs to be in `mako.directories`.
- It connects the alias to the configured route via the *url_path* configuration parameter (the list view is just the base route in this case, but that is totally up to you).
- It returns this alias from the function so that it can be stored in the `routes` dictionary on the view.

`ViewConfigurator.`**`configure_edit_view`**`()`
This method behaves exactly like *`ViewConfigurator.configure_list_view()`* except it must configure the edit view, i.e. the view for editing existing objects. It must return the name of the route as well that will then be stored under the "edit" key.

`ViewConfigurator.`**`configure_new_view`**`()`
This method behaves exactly like *`ViewConfigurator.configure_list_view()`* except it must configure the new view, i.e. the view for adding new objects. It must return the name of the route as well that will then be stored under the "new" key.

There are also some *helper methods* available.

The Info Dictionary

Each object can have an optional info dictionary attached (and in most cases you will want one). The idea is based on the idea of WTForms-Alchemy's Form Customization and actually just extends it. Several attributes used by this library support inclusion of extra information in this dict. The following options can be set and/or read and some are automatically defined if you do not provide a value. The follwoing values are available:

label This is taken over from WTForms-Alchemy but is used in more places. Instead of being just used as the label on a form, it is also used as a column heading in the list view. Each object should have one, but some functions set it (for example, the column header function associated with *list_display* provides a default). For specific behavior on this regarding different views you should consult the associated documentation. While you should normally set it, not setting it will invent some hopefully nice-looking strings for the default usage (basically list and edit views).

description Used on form fields to describe a field more in-depth than a label can. This text may be arbitrarily long. It is not displayed on all templates (see *Fieldset Templates*).

css_class A css class which should be set on this element's context. Currently this is only used for the list view where the `th` element gets this class so you can style your table based on individual columns. See the documentation on *list_display* for more info.

bool This value is not always set, but when it is set, it indicates if this item is a boolean type. Currently this is only set for the list headings and there it is unused but can be adapted by custom templates.

func This is only used with actions and defines the callable which executes an action. It is part of the dict returned by `_all_actions` on the view.

API

The classes, methods and attributes described here are normally not used directly by the user of the library and are just here for the sake of completeness.

CRUDView

The following methods refer to specific views:

CRUDView.**list**()
List all items for a Model. This is the default view that can be overridden by subclasses to change its behavior.

Returns A dict with a single key `items` that is a query which when iterating over yields all items to be listed.

CRUDView.**delete**(*query*)
Delete all objects in the `query`.

CRUDView.**edit**()
The default view for editing an item. It loads the configured form and model. In edit mode (i.e. with an already existing object) it requires a matchdict mapping primary key names to their values. This has to be ensured during route configuring by setting the correct pattern. The default implementation takes correctly care of this.

Returns

In case of a GET request a dict with the key `form` denoting the configured form instance with data from an optional model loaded and a key `is_new` which is a boolean flag indicating whether the actual action is `new` or `edit` (allowing for templates to display "New Item" or "Edit Item").

In case of a POST request, either the same dict is returned or an instance of `HTTPFound` which indicates success in saving the item to the database.

Raises **ValueError** – In case of an invalid, missing or unmatched action. The most likely reason for this is the missing button of a form, e.g. by the name `save`. By default the following actions are supported: `save`, `save_close`, `save_new` and additionally anything that starts with `add_` or `delete_` (these two are for internal form handling and inline deletes/adds).

Addtionally, the following helper methods are used internally during several sections of the library:

CRUDView.**redirect**(*route_name=None*, **args*, ***kw*)
Convenience function to create a redirect.

Parameters **route_name** – The name of the route for which to create a URL. If this is `None`, the current route is used.

All additional arguments and keyword arguments are passed to `pyramid.request.Request.route_url()`.

Returns An instance of `pyramid.httpexceptions.HTTPFound` suitable to be returned from a view to create a redirect.

classmethod CRUDView.**get_template_for**(*action*)
Return the name of the template to be used. By default this uses the template in the folder `theme` with the name `action + template_ext`, so for example in the default case for a list view: "pyramid_crud:templates/mako/bootstrap/list.mako".

This method basically just appends the given action to a base path and appends the file extension. As such, it is perfectly fine, to define relative paths here:

```
view.get_template_for('fieldsets/horizontal')
```

You can also change single templates by statically defining `action_template` on the view class where `action` is replaced by a specific action, e.g. `list`. So say, for example, you want to only change the default list template and keep the others. In that case, you would specify `list_template = "templates/my_crud_list.mako"` and the list template would be loaded from there (while still loading all other templates from their default location).

Parameters **`action`** – The action, e.g. `list` or `edit`.

`CRUDView.`**`_get_request_pks`**`()`
Get an ordered dictionary of primary key names matching to their value, fetched from the request's matchdict (not the model!).

Parameters **`names`** – An iterable of names which are to be fetched from the matchdict.

Returns An `OrderedDict` of the given `names` as keys with their corresponding value.

Raises **`ValueError`** – When only some primary keys are set (it is allowed to have all or none of them set)

`CRUDView.`**`_get_route_pks`**`(`*obj*`)`
Get a dictionary mapping primary key names to values based on the model (contrary to *`_get_request_pks()`* which bases them on the request).

Parameters **`obj`** – An instance of the model.

Returns A dict with primary key names as keys and their values on the object instance as the values.

`CRUDView.`**`_edit_route`**`(`*obj*`)`
Get a route for the edit action based on an objects primary keys.

Parameters **`obj`** – The instance of a model on which the routes values should be based.

Returns A URL which can be used as the routing URL for redirects or displaying the URL on the page.

`CRUDView.`**`iter_head_cols`**`()`
Get an iterable of column headings based on the configuration in `list_display`.

`CRUDView.`**`iter_list_cols`**`(`*obj*`)`
Get an iterable of columns for a given obj suitable as the columns for a single row in the list view. It uses the `list_display` option to determine the columns.

ViewConfigurator

In addition to the methods described above, the default implementation has a few helper methods. These are not required in any case since they are only called by the above methods. However, since these methods are used to factor out common tedious work, you might either use or override them and possibly not even touch the default implementations above.

`ViewConfigurator.`**`_configure_view`**`(`*action*, *route_action=None*, **args*, ***kw*`)`
Configure a view via `pyramid.config.Configurator.add_view()` while passing any additional arguments to it.

Parameters

- **`action`** – The name of the attribute on the view class that represents the action. For example, in the default implementation the `list` action corresponds to *`CRUDView.list()`*. If you rename them, e.g. by naming the `list` action "_my_list", this would have to be `_my_list` regardless of the name of the action.

- **route_action** – An optional parameter that is used as the name base for the route. If this is missing, it will take the same value as `action`. In the default implementation it is used to distinguish between `new` and `edit` which use the same action, view and template but different route names.

Overriding this method allows you to change the view configuration for all configured views at once, i.e. you don't have to change the public methods at all. Just look at their default implementation to see the parameters they use.

`ViewConfigurator.`**`_configure_route`**(*action*, *suffix*, **args*, ***kw*)

Set up a route via `pyramid.config.Configurator.add_route()` while passing all addtional arguments through to it.

Parameters

- **action** – The action upon which to base the route name. It must be the same as `route_action` on *`_configure_view()`*.
- **suffix** – The suffix to be used for the actual path. It is appended to the `url_path` directly. This may be empty (as is the case for the default list view) but must always be explicitly specified. The result of this will be passed to `add_route` and so may (and often will) include parameters such as `/{id}`.

Overriding this method can be done in the same manner as described for *`_configure_view()`*.

Warning: Some methods on the view require primary keys of the object in question in the `matchdict` of the request. To guarantee this, the routes have to be correctly set up, i.e. each route that requires this primary key (or keys, depending on the model) has to have a pattern where each primary key name appears once. The default implementation takes care of this via *`_get_route_pks()`*, but if you change things you have to ensure this yourself.
Which methods require which values is documented on the respective views of *`CRUDView`*.

`ViewConfigurator.`**`_get_route_name`**(*action*)

Get a name for a route of a specific action. The default implementation provides the fully quallyfied name of the view plus the action, e.g. `mypackage.views.MyView.list` (in this case, the action is "list" for the class "MyView" in the module "mypackage.views").

Note: In theory this implementation is ambigous, because you could very well have two classes with the same name in the same module. However, this would be a very awkward implementation and is not recommended anyway. If you really choose to do such a thing, you should probably find a better way of naming your routes.

`ViewConfigurator.`**`_get_route_pks`**()

Get a string representing all primary keys for a route suitable to be given as suffix to *`_configure_route()`*. Some examples will probably best describe the default behavior.

In the case of a model with a single primary key `id`, the result is the very simple string `{id}`. If you add this to a route, the primary key of the object will be in the `matchdict` under the key `id`.

If you have a model with multiple primary keys, say composite foreign keys, called `model1_id` and `model2_id` then the result would be `{model1_id},{model2_id}`. The order is not important on this one as pyramids routing system will fully take care of it.

Note: If you have some kind of setup where one of the primary keys may contain a comma, this implementation is likely to fail and you have to change it. However, in most cases you will **not** have a comma and this should be fine.

2.2.3 Forms

Add a New Form

Configuration

To configure a normal form, you subclass the *ModelForm*. On this subclass there are several options you can/must override. The mandatory options are listed first, followed by a list of optional configuration parameters. Finally, you can of course always override the methods on the form.

class pyramid_crud.forms.**ModelForm**(*formdata=None*, *obj=None*, **args*, ***kw*)
Base-class for all regular forms.

The following configuration options are available on this form in addition to the full behavior described for WTForms-Alchemy

---

Note: While this class can easily be the base for each form you want to configure, it is strongly recommended to use the *CSRFModelForm* instead. It is almost no different than this form except for a new csrf_token field. Thus it should never hurt to subclass it instead of this form.

Meta This is the only mandatory argument. It is directly taken over from WTForms-Alchemy so you should check out their documentation on this class as it will provide you with a complete overview of what's possible here.

inlines A list of forms to use as inline forms. See *Inline Forms / One-To-Many*.

fieldsets Optionally define fieldsets to group your form into categories. It requires a list of dictionaries and in each dictionary, the following attributes can/must be set:

- title: A title to use for the fieldset. This is required but may be the empty string (then no title is displayed).
- fields: A list of field names that should be displayed together in a fieldset. This is required.
- template: The name of the fieldset template to load. This must be the name of a file in the fieldsets directory of the current theme **without** a file extension. It defaults to horizontal which uses bootstraps horizontal forms for each fieldset. See *Fieldset Templates* for details on available templates.

title Set the title of your form. By default this returns the class name of the model. It is used in different places such as the title of the page.

title_plural: The plural title. By default it is the title with an "s" appended, however, you somtimes might want to override it because "Childs" just looks stupid ;-)

name: The name of this form. By default it uses the lowercase model class name. This is used internally und you normally do not need to change it.

get_dbsession: Unfortunately, you have to define this classmethod on the form to get support for the unique validator. It is documented in Unique Validator. This is a limitation we soon hope to overcome.

fieldsets
See inline documentation for ModelForm

get_fieldsets()
Get a list of all configured fieldsets, setting defaults where they are missing.

populate_obj(*obj*)
Populates the attributes of the passed *obj* with data from the form's fields.

Note This is a destructive operation; Any attribute with the same name as a field will be overridden. Use with caution.

populate_obj_inline(*obj*)
Populate all inline objects. It takes the usual `obj` argument that is the **parent** of the inline fields. From these all other values are derived and finally the objects are updated.

Note: Right now this assumes the relationship operation is a `append`, thus for example set collections won't work right now.

---

primary_keys
Get a list of pairs `name, value` of primary key names and their values on the current object.

process(*formdata=None*, *obj=None*, ***kwargs*)
Take form, object data, and keyword arg input and have the fields process them.

Parameters

- **formdata** – Used to pass data coming from the enduser, usually *request.POST* or equivalent.
- **obj** – If *formdata* is empty or not provided, this object is checked for attributes matching form field names, which will be used for field values.
- **data** – If provided, must be a dictionary of data. This is only used if *formdata* is empty or not provided and *obj* does not contain an attribute named the same as the field.
- ****kwargs** – If *formdata* is empty or not provided and *obj* does not contain an attribute named the same as a field, form will assign the value of a matching keyword argument to the field, if one exists.

process_inline(*formdata=None*, *obj=None*, ***kwargs*)
Process all inline fields. This sets the global attribute `inline_fields` which is a dict-like object that contains as keys the name of all defined inline fields and as values a pair of `inline, inline_forms` where `inline` is the inline which the name refers to and `inline_forms` is the list of form instances associated with this inline.

validate()
Validates the form by calling *validate* on each field, passing any extra *Form.validate_<fieldname>* validators to the field validator.

validate_inline()
Validate all inline forms. Implicitly called by *validate()*.

This will also fill the `form.errors` dict with additional error messages based on invalid inline fields using the same naming pattern used for naming inline fields for display and form submission, i.e. `inlinename_index_fieldname`.

Thus, if errors exist on an inline field, they can be fetched from the global errors dict the same way regular errors are present in it.

Fieldset Templates

You can configure custom fieldset templates on the *fieldsets* configuration parameter by setting the "template" key for a fieldset. The following fieldsets are available:

horizontal A typical horizontal display that renders each form field in its own row with a label before the field.

grid A grid display that renders the field first and then displays the label. All fields are next to each other and line breaks only happen at the edge of the screen. This is a good template for a fieldset that consists only of checkboxes. This will not display the "description" of a field.

Inline Forms / One-To-Many

class `pyramid_crud.forms.`**`BaseInLine`**(*formdata=None*, *obj=None*, **args*, ***kw*)

Base-class for all inline forms. You normally don't subclass from this directly unless you want to create a new inline type. However, all inline types share the attributes inherited by this template.

Inline forms are forms that are not intended to be displayed by themselves but instead are added to the *inlines* attribute of a normal form. They will then be displayed inside the normal form while editing, allowing for multiple instance to be added, deleted or modified at the same time. They are heavily inspired by Django's inline forms.

An inline form is configurable with the following attributes, additionally to any attribute provided by WTForms-Alchemy

Meta This is the standard *WTForms-Alchemy* attribute to configure the model. Check out their documentation for specific details.

relationship_name The name of the *other side* of the relationship. Determined automatically, unless there are multiple relationships between the models in which case this must be overridden by the subclass.

For example: If this is the child form to be inlined, the other side might be called `children` and this might be called `parent` (or it might not even exist, there is no need for a bidirectional relationship). The correct value would then be `children` *not* `parent`.

extra How many empty fields to display in which new objects can be added. Pay attention that often fields require intputs and thus extra field may often not be left empty. This is an intentional restriction to allow client-side validation without javascript. So only specify this if you are sure that items will always be added (note, however, that the extra attribute is not used to enforce a minimum number of members in the database). Defaults to `0`.

is_extra A boolean indicating whether this instance is an extra field or a persisted database field. Set during parent's processing.

`fieldsets`

See inline documentation for ModelForm

`get_fieldsets`()

Get a list of all configured fieldsets, setting defaults where they are missing.

classmethod `pks_from_formdata`(*formdata*, *index*)

Get a list of primary key values in the order of the primary keys on the model. The returned value is suitable to be passed to `sqlalchemy.orm.query.Query.get()`.

Parameters

- **`formdata`** – A `webob.multidict.MultiDict` that contains all parameters that were passed to the form.
- **`index`** (*int*) – The index of the element for which the primary key is desired. From this, the correct field name to get from `fromdata` is determined.

Returns A tuple of primary keys that uniquely identify the object in the database. The order is based on the order of primary keys in the table as reported by SQLAlchemy.

Return type tuple

populate_obj(*obj*)
Populates the attributes of the passed *obj* with data from the form's fields.

Note This is a destructive operation; Any attribute with the same name as a field will be overridden. Use with caution.

primary_keys
Get a list of pairs `name, value` of primary key names and their values on the current object.

process(*formdata=None*, *obj=None*, *data=None*, ***kwargs*)
Take form, object data, and keyword arg input and have the fields process them.

Parameters

- **formdata** – Used to pass data coming from the enduser, usually *request.POST* or equivalent.
- **obj** – If *formdata* is empty or not provided, this object is checked for attributes matching form field names, which will be used for field values.
- **data** – If provided, must be a dictionary of data. This is only used if *formdata* is empty or not provided and *obj* does not contain an attribute named the same as the field.
- ****kwargs** – If *formdata* is empty or not provided and *obj* does not contain an attribute named the same as a field, form will assign the value of a matching keyword argument to the field, if one exists.

validate()
Validates the form by calling *validate* on each field, passing any extra *Form.validate_<fieldname>* validators to the field validator.

class pyramid_crud.forms.**TabularInLine**(*formdata=None*, *obj=None*, **args*, ***kw*)
A base class for a tabular inline display. Each row is displayed in a table row with the field labels being displayed in the table head. This is basically a list view of the fields only that you can edit and delete them and even insert new ones.

Many-To-One & One-To-One

The opposite of the One-To-Many pattern is the Many-To-One. One-To-One looks the same from the "One" side, just that the "parent" does not have many "children" but one "child".

Both relationships are possible without any further configuration. They are automatically detected and work right away. Currently, this feature only has limited use as you cannot directly create a parent form here. Instead the parent object has to exist. Then you can go back and select it in the child's edit form.

Note: When using a model with a child as an inline, this automatic detection will not display the parent item in the inline form.

Extra Forms

CSRF

The CSRF Forms are special forms to protect you against CSRF attacks. There are two different types: The *CSRFForm* is the base for any form that wants to enable CSRF and is not limited to the usage within the scope of this library (it is not integrated with the rest of the system, it only implements a WTForms form that takes a `pyramid.request.Request` as the `csrf_context`). The *CSRFModelForm* on the other hand *is* integrated

with the rest of the library and should be used to protect a form against CSRF attacks while still maintaining the complete functionality of the *ModelForm*.

class pyramid_crud.forms.**CSRFForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Base class from which new CSRF-protected forms are derived. Only use this if you want to create a form without the extra model-functionality, i.e. is normal form.

If you want to create a CSRF-protected model form use *CSRFModelForm*.

generate_csrf_token(*csrf_context*)

Create a CSRF token from the given context (which is actually just a pyramid.request.Request instance). This is automatically called during __init__.

validate()

Validate the form and with it the CSRF token. Logs a warning with the error message and the remote IP address in case of an invalid token.

class pyramid_crud.forms.**CSRFModelForm**(*formdata=None*, *obj=None*, **args*, ***kw*)

A form that adds a CSRF token to the form. Derive from this class for security critical operations (read: you want it most of the time and it doesn't hurt).

Do not derive from this for inline stuff and other composite forms: Only the main form should use this as you only need one token per request.

All configuration is done exactly in the same way as with the *ModelForm* except for one difference: An additional csrf_context argument is required. The pre-configured views and templates already know how to utilize this field and work fine with and without it.

Fields

The library defined some special fields. Normally, there is no need to be concerned with them as they are used internally. However, they might provide useful features to a developer.

class pyramid_crud.fields.**MultiCheckboxField**(*label=None*, *validators=None*, *coerce=<type 'unicode'>*, *choices=None*, ***kwargs*)

A multiple-select, except displays a list of checkboxes.

Iterating the field will produce subfields, allowing custom rendering of the enclosed checkbox fields.

Example for displaying this field:

```
class MyForm(Form):
    items = MultiCheckboxField(choices=[('1', 'Label')]
form = MyForm()
for item in form.items:
    str(item)  # the actual field to be displayed, likely in template
```

If you don't iterate, it produces an unordered list be default (if str is called on form.items, not each item individually).

And with formdata it might look like this:

```
# Definition same as above
formdata = MultiDict()
formdata.add('items', '1')
form = MyForm(formdata)
assert form.items.data == ['1']
```

As you can see, a list is produces instead of a scalar value which allows multiple fields with the same name.

class pyramid_crud.fields.**MultiHiddenField**(*label=None*, *validators=None*, *coerce=<type 'unicode'>*, *choices=None*, ***kwargs*)

A field that represents a list of hidden input fields the same way as *MultiCheckboxField* and wtforms.fields.SelectMultipleField.

class pyramid_crud.fields.**SelectField**(*label=None*, *validators=None*, *coerce=<type 'unicode'>*, *choices=None*, ***kwargs*)

Same as wtforms.fields.SelectField with a custom validation message and the requirement that data evaluates to True (for the purpose of having an empty field that is not allowed).

2.2.4 Templates

Changing a Single Template

Changing a single template is as simple as copying it from the default theme and adjusting it to your needs. For example, say you want to change the list.mako template. You copy it over and start editing.

```
cp /path/to/pyramid_crud/templates/mako/bootstrap/list.mako my_library/templates/crud/list.mako
```

After you are done editing the template, you need to configure your view to load it:

```
class MyCRUDView(CRUDView):
    ...
    template_list = 'crud/list.mako'
```

This is all assuming the crud directory can be looked up (in the example above, you would need my_library/templates to be in mako.directories).

For an explanation of each template and some additional details, see *Create a Complete Theme*.

Create a Complete Theme

The default theme uses Bootstrap which looks nice but also not really unique and does not integrate at all with your own application look. Thus, you might want to roll your own look for all views. This is easily possible with the *theme* configuration parameter.

Note: This is an advanced technique. A lot of knowledge about the rest of the library is assumed throughout this section and so it is recommended that you make yourself familiar with the rest of the documentation before taking on own themes.

It is perfectly okay to create your application with the default theme first and change it afterwards to your custom theme. This way you familiarize yourself with the library and have it much easier understanding what is going on here.

The best way to roll your own template is to copy the default template from pyramid_crud/templates/mako/bootstrap. Let's say you want to create your own theme by the name my_crud_theme. First you copy over the theme folder:

```
cp -a /path/to/pyramid_crud/templates/mako/bootstrap my_library/templates/my_crud_theme
```

Now you can directly enable your theme by configuring the *theme* variable with the setting my_crud_theme (this is assuming that this folder is in your mako.directories path). With this, you should already have your new template enabled.

Note: If you want to configure a default template, just create your own intermediate base class that defines the `theme` parameter. This isn't a very pythonic solution but it works and is very flexible.

Now you should fire up your text editor and take a look at the files in your new theme folder. Here is a description of the files used:

Note: Each file receives the usual `request` and `view` parameters pyramid passes in by default.

base.mako Contains the basic template with style sheets, flash messages and everything else that each template needs. You will define your own look here.

If you already have your own template, you don't need this file and can delete it. You then have two options for configuring a custom base template:

- You can statically set the path in each `<%include` statement in the inheriting templates or
- You can define *template_base* on the view and set it to the path of your own base template. It will then take this path as the base for all your templates and the regular base file is not needed anymore.

If you roll your own base, pay attention to the flash messages and their queue names: They are currently statically configured and so you have to read these queues or won't see any messages at all.

Also pay attention to the blocks used by inherting templates and either change them or define them in your base (e.g. `head` and `heading`).

list.mako A simple list view. It gets two arguments: The `items` parameter is a query that you can iterate over to get the object instances for each row. The `action_form` parameter is a form instance with the following fields:

action A select list where you can choose an action and execute it on multiple items at once (see *Adding Actions to Forms*).

items A field that has one checkbox field for each item in the `items` iterable. If you iterate over it, you get a single field that renders to a checkbox. In the default implementation, `zip()` is used to provide each loop iteration with a single checkbox field and the corresponding item.

csrf_token A CSRF token field. This is required and must be displayed somewhere in the form or the validation will fail.

submit A submit button that sends the form to execute the actions on the selected items.

edit.mako The view of a single item being edited. In the default implementation, this loads a fieldset for each configured fieldset on the form and then loads an inline template for each configured inline on the form. It receives the following parameters:

form A form representing the item being edited. It is an instance of your subclassed `ModelForm`. Look at the documentation for *Forms* for more information on supported methods (make sure to also checkout linked documentation from there).

is_new A boolean representing whether this is a new item or not.

delete_confirm.mako This template is invoked after the delete action was called and displays an intermediate view to make sure the user really wants to delete the selected items. It gets the following arguments:

form The same form that the list view got as `action_form`

items The list of items to be deleted.

edit_inline/*.mako Any file in this folder is considered an inline template to be included. The following parameters are given during inclusion:

inline The class that is inlined (not an instance!). It is the subclass you made from base of *BaseInLine*.

items Instances of the above `inline` parameter, each being a form to be displayed inlined.

fieldsets/*.mako This file is used by the `edit.mako` template for each fieldset that should be rendered. It gets a single `fieldset` argument which is a dict with the following keys (note that it also keeps globals of the parent):

title The title of this fieldset, usually displayed in a `<legend>` tag.

fields A list of field names on the form. Use these to retrieve the correct field from the form instance. This is used instead of iterating over the form so you can group the fields into fieldsets.

You often don't need to edit all of the files if you don't use them. For example, the `grid` fieldset is just a special case and can often go unused (you can delete it if you never use it on any fieldset). You can also often keep the default template if you like the way they do things and just style them by creating your own stylesheet using the same classes bootstrap does.

Keeping Some Templates from the Default Library

Sometimes you might want to change the complete look and overwrite most of the templates but keep some of them from the old library. You could just keep the original copy you made above but that is not a good idea because you might miss out on updates to the templates. You can abuse the `template_*` setting for this, as it works both ways: Just set it to the path of the template you want to keep. For example, to keep the `delete_confirm` template but overwrite everything else, configure your view like this:

```
class MyCRUDView(CRUDView):
    ...
    template_delete_confirm = 'pyramid_crud:templates/mako/bootstrap/delete_confirm.mako'
```

Note how this is the full asset specification of the template because it is not in any of the directories configured with `mako.directories`. Also note, that you cannot do this with templates in subdirectories (see *template_** for an explanation and solution).

Supporting Different Template Engines

Supporting another template engine is very simple. Assuming you already use them in the rest of the application, you have them set up anyway. Once you have a theme for this engine, you can just set it to the file extension of this theme.

Let's say, for example, you have created a Chameleon theme with all file names ending in `.pt`. If you have this renderer enabled properly, it will automatically be chosen correctly, if you give pyramid a path to a file ending with `.pt`. Thus, in addition to configuring your theme (see above), you just configure the *template_ext* parameter to `.pt` and are good to go. This is what your view might look like:

```
class MyCRUDView(CRUDView)
    ...
    theme = 'templates/my_chameleon_theme'
    template_ext = '.pt'
```

Now assuming the lookup is correctly configured, this will fetch the templates using the correct renderer.

2.2.5 Utility Functions

API

2.2.6 Adding Actions to Forms

Similar to Django's Admin actions, pyramid_crud also provides a way to configure specific actions.

Introduction

What are actions?

An action in the context of this library is something you perform on a list of items that might change their state (or perform anything else, really). A good example would be publishing multiple articles at once or activating multiple users.

How do you configure actions?

Actions are configured by setting the *actions* parameter on your view. Possible values here are strings or callables. If a string is provided, a method of the same name is looked up on the view and used as the callable.

Each callable gets two arguments: The view and the query which selects the items for which the actions should be performed. Note that a query is used instead of a list of items so that you can refine it or directly perform actions on it. If you need a list, call `.all` on it or iterate over it.

So how do I create an action exactly?

Let's work by example and take the same example Django does so we can directly see similarities and differences. Here's our definition with which we start:

```
class Article(Base):
    id = Column(Integer, primary_key=True)
    title = Column(Text)
    body = Column(Text)
    status = Column(Enum('p', 'd', 'w'))

    def __unicode__(self):
        return self.title
```

Now that we have our model, let's make a method to publish multiple articles at once:

```
def make_published(view, query):
    query.update({'status': 'p'})
    return True, None
```

Notice, how we don't pass in the request as it can be accessed with `view.request`. The view is an instance of your subclassed *`CRUDView`*. The query is an instance of `Query`. Additionally, you can see that we return a pair here. The first value indicates success of the operation, the latter value is an optional response (see *Returning Values From Actions* for a detailed explanation).

Now you might want to have a nicer title than 'Make Published' (this title is assigned by default, replacing underscores with spaces and calling `str.title()` on the result). To achieve a custom title (that will appear in the list of items), assign a label to its *info dict*:

```
def make_published(view, query):
    query.update({'status': 'p'})
    return True, None
make_published.info = {'label': "Mark selected stories as published"}
```

And how do I add it to a view?

That's easy. Here is a full configuration based on the model above:

```
class ArticleForm(ModelForm):
    class Meta:
        model = Article

class ArticleView(CRUDView):
    Form = ArticleForm
    url_path = '/articles'
    actions = [make_published]
```

See how we added the *actions* configuration directive? We gave it a list (with one item) of actions that should be available on this model.

And that's it, now you have an additional action available at your disposal. Read on for some more information, including advanced techniques, differences and what's missing in comparison to Django.

Advanced Techniques

Handling Errors

To handle exceptions, wrap your code in a `try-except-else` clause. You can then handle any exception and possibly log error messages and flash a message to the user. This allows you to shield the user from any application crashes and gives you the ability to examine the log for the cause of the error.

Nonetheless, you can still raise exceptions and they will be passed through in which case the section on *Returning Values From Actions* does not really apply (as no value is returned).

An example of an implementation that shields to user from exceptions might look like this:

```
def make_published(view, query):
    try:
        query.update({'status': 'p'})
    except:
        log.error("An error oucurred:\n%s" % format_traceback())
        self.request.session.flash("An error happened while publishing "
                                   "the article(s)")
        return False, None
    else:
        return True, None
```

This will inform the user of any failure and log the exact exception so you can investigate the problem. Note that with a perfect implementation, you would probably want to explicitly catch all possible exceptions and not use a catch-all. However, since this implementation doesn't just ignore and instead log the exception, it is not too bad to have a catch all here.

Returning Values From Actions

As already noted above, it is recommended to wrap your code in `try-except-else` blocks and return the status as a boolean. The reason for this is to allow explicit changes in application behavior based on the result of your execution.

You always have to return a pair of `(success, response)` to indicate how you would like to proceed.

`success` must be a boolean value. If it is `False` it indicates that the action was not successful. In this case the redirect is **raised** which means it is considered an exception. Any optional transaction (e.g. pyramid_tm) will see this exception and abort the transaction. Afterwards the page is redirected. The `response` value is not used in this case, so it should always be `None`.

If `success` is `True`, it is assumed that the action was successful. In this case the redirect is **returned** and the transaction is committed. Note that this is a fine distinction between success and failure and the user does not see a difference (except error messages you might give out).

However, in the case of a successful response, you might also want to change the returned value into something else (maybe redirect somewhere else or return a whole new response). This can be done by setting the `response` paramater which can really be anything that is allowed to be returned from a view.

So for example if you wanted to direct to a completely different page, you could return an instance of `HTTPFound` that achieves this. On the other hand, you maybe want to create an intermediate response. In that case, you just need to return an instance of `Response`. You could create this by calling `render_to_response` if you want to render an intermediary view from a template. This is the technique the delete action uses.

Note: The more complex it gets, the more likely it is that a redirect to an actual view is much better than manually rendering or building your response. This allows you to factor out the code from your action into a separate view but has the drawback of an additional redirect and the need to keep all the formdata alive (e.g. in the session).

Actions as Methods on the View

Instead of having an external function, you can add your action directly to the view (in most cases the recommended way). For this, you just create a method on the view instead of a function:

```
class ArticleView(CRUDView):
    ...
    def make_published(self, query):
        try:
            query.update({'status': 'p'})
        except:
            return False, None
        else:
            return True, None
    make_published = {'info': "Mark selected stories as published"}
```

Note how we renamed `view` to `self` because as a method the view reference is now actually the own instance.

Instead of providing the action as a callable, you now use a string instead:

```
class ArticleView(CRUDView):
    actions = ['make_published']
```

This will look up the action as a method on the view and call it in the same manner.

Currently Unspported Features from Django

- Site-wide actions: Currently it is not possible to add actions that are globally available. However, you can work around that by creating a custom subclass and modifiying the action list in the children during runtime, however, this is an unspported as of now and you might face some issues with mutability.
- Disabling actions: This is currently not supported at all.
- Runtime disabling/enabling of actions: While unspported, this is possible by overriding the `_all_actions` atribute. In the default implementation it behaves like a property but caches its result (using Pyramid's reify decoartor). Take a look at the default implementation to see the format of the returned value.

2.2.7 Help Topics

Transition From Django

If you are coming from Django you are probably wondering which features are offered here and to which Django features they correspond. To help you with this, here are some guidelines on getting used to this system.

`ModelAdmin` Configuration Options

Django as a myriad of configuration options on its ModelAdmin. Not all features are supported here and they are distributed differently (due to the nature of this library), so here is some guidance on getting around.

First of all, you have to know that we split things a bit more up around here because we are already based on other libraries. Roughly speaking though we can define some equivalents:

- Django's `ModelAdmin` is closest to `views.CRUDView`. Here we perform basic configuration actions, set routes and so on. However, contrary to Django not all configuration is performed here. And instead of a registration of the `ModelAdmin` with the `Model`, we define a `Form` on it that in turn links to the model.
- The `Form` subclasses `forms.ModelForm`. There is no equivalent in Django, as it creates the form automatically from the relation between the model and the admin class. However, because of the way the integration works (the form is actually created by WTForms-Alchemy) we need this form. This is the place where you configure behavior on actual form instances.
- The `Model` in turn is very close to that of Django. But since our systems are not so closely integrated, the coupling is much lower (which can be good or bad). The models are mostly just SQLAlchemy models with some additional configuration done by WTForms-Alchemy.

Now that you know how the parts are constructed take a look at the following table. It lists each Django option on `ModelAdmin` and provides the equivalent in either this library, WTForms-Alchemy or SQLAlchemy. If a behavior is not supported here yet, it is denoted by "NYI" (not yet implemented).

Django	Us
`actions`	*Adding Actions to Forms*
`actions_on_top`	NYI
`actions_on_bottomg`	NYI
`actions_selection_count`	NYI
`date_hierarchy`	NYI
`exclude`	Form.Meta.exclude
`fields`	Form.Meta.only
`fieldsets`	*ModelForm.fieldsets*
`filter_horizontal`	NYI

Continued on next page

Table 2.1 – continued from previous page

Django	Us
filter_vertical	NYI
form	*CRUDView.Form*
formfield_overrides	Adding/overriding fields
inlines	*ModelForm.inlines*
list_display	*CRUDView.list_display*
list_display_links	*CRUDView.list_display_links*
list_editable	NYI
list_filter	NYI
list_max_show_all	NYI
list_per_page	NYI
list_select_related	NYI
ordering	NYI
paginator	NYI
prepoulated_fields	NYI
preserve_filters	NYI
radio_fields	NYI
raw_id_fields	NYI
readonly_fields	NYI
save_as	NYI
save_on_top	NYI
search_fields	NYI
add_form_template	NYI
change_form_template	NYI
change_list_template	NYI
delete_confirmation_template	NYI
delete_selected_confirmation_template	NYI
object_history_template	NYI
save_model	NYI
delete_model	NYI
save_formset	NYI
get_ordering	NYI
get_search_results	NYI
save_related	NYI
get_readonly_fields	NYI
get_prepopulated_fields	NYI
get_list_display	NYI
get_list_display_links	NYI
get_fieldsets	NYI
get_list_filter	NYI
get_inline_instances	NYI
get_urls	NYI
get_form	NYI
get_formsets	NYI
formfield_for_foreignkey	NYI
formfield_for_manytomany	NYI
formfield_for_choice_field	NYI
get_changelist	NYI
get_changelist_form	NYI
get_changelist_formset	NYI

Continued on next page

Table 2.1 – continued from previous page

Django	Us
has_add_permission	NYI
has_change_permission	NYI
has_delete_permission	NYI
get_queryset	NYI
message_user	NYI
get_paginator	NYI
add_view	NYI
change_view	NYI
changelist_view	NYI
delete_view	NYI
history_view	NYI
Media	NYI

FAQ

2.2.8 Examples

2.3 Development

2.3.1 Building Documentation

To build the documentation you first need to install all documentation dependencies:

```
pip install -r docs_require.txt
```

Then you can build the documentation:

```
python setup.py build_sphinx
```

2.3.2 Running Tests

Before you can run tests, you need to install the requirements. These consist of the requirements to create docs (for doctests) and pytest:

```
pip install -r tests_require.txt
```

Note: `mock` is an unnecessary requirement for users of python 3.3 and above, but it is included in the above file unconditionally.

Now you can run your tests with:

```
python setup.py test
```

If you need more control over which tests are executed, you can also execute pytest and doctest directly:

```
py.test tests/
make -C docs/ doctest
```

Note: Our tests are also run against templates. However, as they are not python files, the test suite automatically compiles them into a temporary directory. This directory should never be checked into GitHub and also be removed before installing the library (it does not hurt, it just pollutes the directory).

Running tests against templates also is included in coverage (the reason why we need an accessible template module directory). The coverage values are reported by Travis CI to coveralls. However, since the code for this is not on GitHub, you cannot see which lines were missted online. Instead, you need to run those tests locally and get coverage output with `coverage html` after you have run the tests.

2.3.3 Contributing

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

p

pyramid_crud.fields, 19
pyramid_crud.forms, 15
pyramid_crud.views, 7

Symbols

_configure_route() (pyramid_crud.views.ViewConfigurator method), 14
_configure_view() (pyramid_crud.views.ViewConfigurator method), 13
_edit_route() (pyramid_crud.views.CRUDView method), 13
_get_request_pks() (pyramid_crud.views.CRUDView method), 13
_get_route_name() (pyramid_crud.views.ViewConfigurator method), 14
_get_route_pks() (pyramid_crud.views.CRUDView method), 13
_get_route_pks() (pyramid_crud.views.ViewConfigurator method), 14

B

BaseInLine (class in pyramid_crud.forms), 17

C

configure_edit_view() (pyramid_crud.views.ViewConfigurator method), 11
configure_list_view() (pyramid_crud.views.ViewConfigurator method), 10
configure_new_view() (pyramid_crud.views.ViewConfigurator method), 11
CRUDView (class in pyramid_crud.views), 7
CSRFForm (class in pyramid_crud.forms), 19
CSRFModelForm (class in pyramid_crud.forms), 19

D

delete() (pyramid_crud.views.CRUDView method), 12

E

edit() (pyramid_crud.views.CRUDView method), 12

F

fieldsets (pyramid_crud.forms.BaseInLine attribute), 17
fieldsets (pyramid_crud.forms.ModelForm attribute), 15

G

generate_csrf_token() (pyramid_crud.forms.CSRFForm method), 19
get_fieldsets() (pyramid_crud.forms.BaseInLine method), 17
get_fieldsets() (pyramid_crud.forms.ModelForm method), 15
get_template_for() (pyramid_crud.views.CRUDView class method), 12

I

iter_head_cols() (pyramid_crud.views.CRUDView method), 13
iter_list_cols() (pyramid_crud.views.CRUDView method), 13

L

list() (pyramid_crud.views.CRUDView method), 12

M

ModelForm (class in pyramid_crud.forms), 15
MultiCheckboxField (class in pyramid_crud.fields), 19
MultiHiddenField (class in pyramid_crud.fields), 19

P

pks_from_formdata() (pyramid_crud.forms.BaseInLine class method), 17
populate_obj() (pyramid_crud.forms.BaseInLine method), 17
populate_obj() (pyramid_crud.forms.ModelForm method), 15
populate_obj_inline() (pyramid_crud.forms.ModelForm method), 16
primary_keys (pyramid_crud.forms.BaseInLine attribute), 18

primary_keys (pyramid_crud.forms.ModelForm attribute), 16
process() (pyramid_crud.forms.BaseInLine method), 18
process() (pyramid_crud.forms.ModelForm method), 16
process_inline() (pyramid_crud.forms.ModelForm method), 16
pyramid_crud.fields (module), 19
pyramid_crud.forms (module), 15
pyramid_crud.views (module), 7

R

redirect() (pyramid_crud.views.CRUDView method), 12

S

SelectField (class in pyramid_crud.fields), 20

T

TabularInLine (class in pyramid_crud.forms), 18

V

validate() (pyramid_crud.forms.BaseInLine method), 18
validate() (pyramid_crud.forms.CSRFForm method), 19
validate() (pyramid_crud.forms.ModelForm method), 16
validate_inline() (pyramid_crud.forms.ModelForm method), 16
ViewConfigurator (class in pyramid_crud.views), 10