Rebar Documentation

Release 0.1

Nathan Yergler

May 03, 2017

Contents

Rebar makes your Django forms stronger.

Rebar provides form-related tools for Django which enable complex form interactions.

CHAPTER 1

Installation

You can install Rebar using *pip* or *easy_install*:

```
$ pip install rebar
```

Once installed, Rebar provides three primary tools:

- Form Groups, for working with heterongenous collections of forms.
- State Validators, for validating that form data is in a specific state.
- Testing tools, which provide convenient helpers for writing unit tests for forms.

CHAPTER 2

Documentation

Form Groups

Rebar Form Groups provide a way to manipulate a heterogenous set of Forms or FormSets as a single entity.

> **Warning:** Restrictions
>
> To treat a set of heterogenous Forms or FormSets as a single entity, some restrictions are necessary. Specifically, every member of a Form Group will receive the same arguments at instantiation.
>
> This means if you're using ModelForms and ModelFormSets, *every* member has the same `instance`. If your data model does not meet this requirement, you'll need to work around it to use Form Groups.

Declaration

Form Groups classes are created using a factory function, much like Form Sets. The only required argument is the sequence of members for the Form Group. A Form Group may contain any number of Forms or FormSets.

Let's take an example where we might split form validation for a Contact into basic information and address information.

```
from django import forms

class ContactForm(forms.Form):

    first_name = forms.CharField(
        label="First Name")
    last_name = forms.CharField(
        label="Last Name")
    email = forms.EmailField(
        label="Email Address")
```

```
class AddressForm(forms.Form):

    street = forms.CharField()
    city = forms.CharField()
    state = forms.CharField()
```

A Form Group allows you to combine the two and treat them as one.

```
from rebar.group import formgroup_factory

ContactFormGroup = formgroup_factory(
    (
        (ContactForm, 'contact'),
        (AddressForm, 'address'),
    ),
)
```

The `ContactFormGroup` class can now be instantiated like any other form.

```
>>> ContactFormGroup()
<rebar.group.FormGroup ...>
```

Using Form Groups

Form Groups attempt to "look" as much like a single form as possible. Note that I say *as possible*, since they are a different creature, you can't use them completely without knowledge. The goal is to make them similar enough to work with Django's class based views

Accessing Member Forms

Once you've instantiated a Form Group, its members are accesible either by index or name.

```
>>> form_group = ContactFormGroup()
>>> form_group.contact
<ContactForm ...>
>>> form_group.address
<AddressForm ...>
>>> form_group[0] == form_group.contact
True
```

The members are provided in the order of declaration.

Form Prefixes

Form Groups have a prefix, much like FormSets, and sets the prefix on each member.

```
>>> form_group = ContactFormGroup()
>>> form_group.prefix
'group'
>>> form_group.contact.prefix
'group-contact'
```

You can also override the default prefix.

```
>>> form_group = ContactFormGroup(prefix='contact')
>>> form_group.prefix
'contact'
>>> form_group.contact.prefix
'contact-contact'
```

Validation

FormGroups use a similar approach to validation as FormSets. Calling `is_valid()` on a FormGroup instance will return `True` if all members are valid.

The `errors` property is a list of ErrorLists, in group member order.

Just as FormSets support a clean method for performing any validation at the set level, FormGroups provide a `clean` hook for performing any validation across the entire group. In order to take advantage of this hook, you'll need to subclass `FormGroup`.

```
from django.core.exceptions import ValidationError
from rebar.group import FormGroup

class BaseInvalidFormGroup(FormGroup):

    def clean(self):
        raise ValidationError("Group validation error.")
```

This class is passed to `formgroup_factory` and used as the base class for the new Form Group.

```
InvalidFormGroup = formgroup_factory(
    (
        (ContactForm, 'contact'),
        (AddressForm, 'address'),
    ),
    formgroup=BaseInvalidFormGroup,
)
```

When you instantiate the form group with data, any errors raised by the `clean` method are available as "group errors":

```
>>> bound_formgroup = InvalidFormGroup(data={})
>>> bound_formgroup.is_valid()
False
>>> bound_formgroup.group_errors()
[u'Group validation error.']
```

There are two things to note about group level validation:

- Unlike `Form.clean()`, the return value of `FormGroup.clean()` is unimportant
- Unlike accessing the `errors` property of Forms, FormSets, or FormGroups, `FormGroup.group_errors()` *does not* trigger validation.

Passing Extra Arguments

Most arguments that you pass to a Form Group will be passed in to its members, as well. Sometimes, however, you want to pass arguments to specific members of the Form Group. The `member_kwargs` parameter allows you to do this.

`member_kwargs` is a dict, where each key is the name of a Form Group member, and the value is a dict of keyword arguments to pass to that member.

For example:

```
>>> form_group = ContactFormGroup(
...     member_kwargs={
...         'address': {
...             'prefix': 'just_address',
...         },
...     },
... )
>>> form_group.contact.prefix
'group-contact'
>>> form_group.address.prefix
'just_address'
```

In this example we override the prefix argument. A more realistic application is when you have a heavily customized form subclass that requires some additional piece of information.

Form Groups in Views

Using in Class Based Views

Form Groups are designed to be usable with Django's class based views. The group class can be specified as the form_class for an edit view. If you need to pass additional arguments, you can override the get_form_kwargs method to add the `member_kwargs`.

Rendering Form Groups

Form Groups do not provide shortcuts for rendering in templates. The shortest way to emit the members is to simply iterate over the members:

```
{% for form in formgroup.forms %}

    {{ form.as_p }}

{% endfor %}
```

Form Groups do provide media definitions that roll-up any media found in members.

State Validators

There are times when it's useful to have validation logic above and beyond what a form applies. For example, a CMS might have very lax validation for saving a document in progress, but enforce additional validation for publication.

State Validators provide a method for encapsulating validation logic for states above and beyond basic form or model validation.

State Validators are made up of individual validation functions, and provides error information about failures, if any. They are similar to the validators in *Forms* or *Models*, in that you describe validators on a field basis. State Validators can be used to validate forms, models, and dicts of values.

Creating Validators

A State Validator consists of a collection of validation functions. A validation function takes an input value and raises a *ValidationError* if the value is invalid.

For example, validation functions that enforce a field as "required" and one that require an integer value might be written as follows.

```
from django.core.exceptions import ValidationError


def required(value):
    if not bool(value):
        raise ValidationError("This field is required.")

def is_int(value):
    try:
        int(value)
    except (ValueError, TypeError):
        raise ValidationError("An integer is required.")
```

These are wrapped into a State Validator using the *statevalidator_factory()*.

```
from rebar.validators import statevalidator_factory

AgeValidator = statevalidator_factory(
    {
        'age': (required, is_int,),
    },
)
```

`statevalidator_factory` takes a dict which maps field names to one or more validator functions, and returns a *StateValidator* class.

Note that when validating, *all* validators will be called for each field, regardless of whether a preceeding validator raises an exception. The goal is to collect all errors that need to be corrected.

Validating Data

Once a StateValidator class has been created, it can be used to validate data. State Validators can validate a simple dict of fields, a form, or a model. When validating a form, its `cleaned_data` will be validated if it is bound, otherwise the initial data values will be used.

```
>>> validator = AgeValidator()
>>> validator.is_valid({'age': 10})
True
>>> validator.is_valid({'age': 'ten'})
False
>>> validator.is_valid({})
False
```

Accessing Errors

In addition to determining if the data is valid or not, the errors raised by the validation functions can be retrieved through the `errors` method. *StateValidator.errors()* returns a dict, where the keys correspond to field names, and the values are a sequence of errors. `errors` returns an empty dict if there are no errors.

```
>>> validator = AgeValidator()
>>> validator.errors({'age': 10})
{}
>>> validator.errors({'age': 'ten'})
{'age': [u'An integer is required.']}
>>> validator.errors({})
{'age': [u'This field is required.', u'An integer is required.']}
```

Enabling & Disabling Validators

State Validators may be disabled or re-enabled. A disabled validator is always valid, and returns no errors.

```
>>> validator.enabled
True
>>> validator.disable()
>>> validator.is_valid({})
True
>>> validator.errors({})
{}
```

```
>>> validator.enabled
False
>>> validator.enable()
>>> validator.is_valid({})
False
```

Testing with Rebar

Rebar includes some helpers for writing unit tests for Django Forms.

Generating Form Data

A common pattern for testing Forms and Formsets is to create a dict of form data to use when instantiating the form. When dealing with a large form, or a form with lots of initial values, keeping the test in sync can be cumbersome.

rebar.testing.flatten_to_dict() takes a Form, Formset, or FormGroup, and returns its fields as a dict. This allows you to create an unbound form, flatten it, and use the resulting dict as data to test with.

If passed a FormSet, the return of `flatten_to_dict` will include the ManagementForm.

For example, if you have a form with a required field:

```
from django import forms

class NameForm(forms.Form):

    first_name = forms.CharField(required=True)
```

You can create an unbound version of the form to generate the form data dict from.

```
>>> from rebar.testing import flatten_to_dict
>>> unbound_form = NameForm()
>>> form_data = flatten_to_dict(unbound_form)
```

```
>>> form_data['first_name'] = 'Larry'
>>> NameForm(data=form_data).is_valid() is True
True
```

This is an obviously oversimplified example, but `flatten_to_dict` allows you to focus your test on the fields that actually matter in that context.

If a ModelForm is passed to `flatten_to_dict` with a foreign key, the related object's primary key, if any, will be used as the value for that field. This correlates with how Django treats those fields in form proessing.

Empty Form Data for Formsets

FormSets allow you to create a view that contains multiple instances of the same form. FormSets have a convenience property, empty_form, which returns an empty copy of the form, with its index set to the placeholder `__prefix__`.

Rebar provides a convience function, *`rebar.testing.empty_form_data()`*, which takes the empty form and returns the form data dict with the prefix correctly filled in. For example, if the FormSet contains a single item, the prefix will be set to 2.

For example, assume we make a FormSet from our example `NameForm` above.

```
from django.forms import formsets
NameFormSet = formsets.formset_factory(form=NameForm)
```

When we instantiate that FormSet, it will have a single form in it, which is empty (ie, we didn't start with any forms). That forms prefix contains "1", indicating its place in the sequence.

```
>>> formset = NameFormSet()
>>> len(formset)
1
>>> formset[0].prefix
u'form-0'
```

The empty_form property contains a copy of the NameForm, with its prefix set to the `__prefix__` sentinel.

```
>>> formset.empty_form
<NameForm ...>
>>> formset.empty_form.prefix
u'form-__prefix__'
```

If we pass the FormSet to `empty_form_data`, we'll get a dict of data for the next form in the sequence.

```
>>> from rebar.testing import empty_form_data
>>> empty_form_data (formset)
{u'form-1-first_name': None}
```

You can also specify a specific index for the generated form data.

```
>>> empty_form_data (formset, index=42)
{u'form-42-first_name': None}
```

API Documentation

rebar Package

rebar Package

group Module

class rebar.group.**FormGroup**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *label_suffix=':'*, *instance=<rebar.group.Unspecified object>*, *error_class=None*, *member_kwargs=None*)

Bases: object

Form-like wrapper for a heterogenous collection of Forms.

A FormGroup collects an ordered set of Forms and/or FormSets, and provides convenience methods for validating the group as a whole.

add_prefix(*field_name*)
Return the field name with a prefix prepended.

clean()
Hook for doing formgroup-wide cleaning/validation.

Subclasses can override this to perform validation after .clean() has been called on every member.

Any ValidationError raised by this method will be accessible via formgroup.group_errors().()

errors

forms

get_default_prefix()

group_errors()
Return the group level validation errors.

Returns an ErrorList of errors that aren't associated with a particular form. Returns an empty ErrorList if there are none.

html_id(*field_name*, *form=None*)
Return the html ID for the given field_name.

is_valid()

media

save()
Save the changes to the instance and any related objects.

class rebar.group.**StateValidatorFormGroup**(**args*, ***kwargs*)

Bases: *rebar.validators.StateValidatorFormMixin*, *rebar.group.FormGroup*

Subclasses are expected to define the state_validators property, which is a mapping of states to StateValidator objects.

get_errors(**states*)

is_valid(**states*)
Returns True if no errors are thrown for the specified state.

rebar.group.**formgroup_factory**(*form_classes*, *formgroup=None*, *state_validators=None*)
Return a FormGroup class for the given form[set] form_classes.

testing Module

Tools for testing Forms, FormSets, and FormGroups.

rebar.testing.**empty_form_data**(*formset*, *index=None*)
Return a form data dictionary for a "new" form in a formset.

Given a formset and an index, return a copy of the empty form data. If index is not provided, the index of the *first* empty form will be used as the new index.

rebar.testing.**flatten_to_dict**(*item*)
Recursively flatten a Form-like object to a data dict.

Given a Form-like object such as a Form, ModelForm, FormSet, or FormGroup, flatten the members into a single dict, similar to what is provided by request.POST.

If `item` is a FormSet, all member Forms will be included in the resulting data dictionary

`validators` Module

class rebar.validators.**StateValidator**
Bases: `object`

Field Validators which must pass for an object to be in a state.

disable()
Disable the validator; when disabled, no errors will be returned.

enable()
Enable the validators.

enabled

errors(*instance*)
Run all field validators and return a dict of errors.

The keys of the resulting dict coorespond to field names. instance can be a dict (ie, form.cleaned_data), a form, a formset, or a model instance.

If instance is a form, full_clean() will be called if the form is bound.

If instance is a formset, full_clean() will be called on each member form, if bound.

is_valid(*instance*)
Return True if no errors are raised when validating instance.

instance can be a dict (ie, form.cleaned_data), a form, or a model instance. If instance is a form, full_clean() will be called.

validators = {}

class rebar.validators.**StateValidatorFormMixin**(**args*, ***kwargs*)
Bases: `object`

Mixin for adding state validators to forms.

Subclasses are expected to define the state_validators property, which is a mapping of states to StateValidator objects.

get_errors(*state*)
Return any validation errors raised for the specified state.

is_valid(**states*)
Returns True if no errors are thrown for the specified state.

state_validators = {}

rebar.validators.**statevalidator_factory**(*field_validators*, *validator=<class 'rebar.validators.StateValidator'>*)

Return a StateValidator Class with the given validators.

Developing Rebar

To run the Rebar unittests, ensure the package is set up for development (ie, you've run `python setup.py develop`), and then run:

```
$ python setup.py test
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index