WTForms-Alchemy Documentation

Release 0.16.2

Konsta Vesterinen

Mar 15, 2017

Contents

WTForms-Alchemy is a WTForms extension toolkit for easier creation of model based forms. Strongly influenced by Django ModelForm.

CHAPTER 1

Introduction

What for?

Many times when building modern web apps with SQLAlchemy you'll have forms that map closely to models. For example, you might have a Article model, and you want to create a form that lets people post new article. In this case, it would be time-consuming to define the field types and basic validators in your form, because you've already defined the fields in your model.

WTForms-Alchemy provides a helper class that let you create a Form class from a SQLAlchemy model.

Differences with wtforms.ext.sqlalchemy model_form

WTForms-Alchemy does not try to replace all the functionality of wtforms.ext.sqlalchemy. It only tries to replace the model_form function of wtforms.ext.sqlalchemy by a much better solution. Other functionality of .ext.sqlalchemy such as QuerySelectField and QuerySelectMultipleField can be used along with WTForms-Alchemy.

Now how is WTForms-Alchemy ModelForm better than wtforms.ext.sqlachemy's model_form?

- Provides explicit declaration of ModelForms (much easier to override certain columns)
- Form generation supports Unique and NumberRange validators
- Form inheritance support (along with form configuration inheritance)
- Automatic SelectField type coercing based on underlying column type
- By default uses wtforms_components SelectField for fields with choices. This field understands None values and renders nested datastructures as optgroups.
- Provides better Unique validator
- Supports custom user defined types as well as type decorators
- Supports SQLAlchemy-Utils datatypes
- Supports ModelForm model relations population

- Smarter field exclusion
- Smarter field conversion
- Understands join table inheritance
- Better configuration

Installation

```
pip install WTForms-Alchemy
```

The supported python versions are 2.6, 2.7 and 3.3.

QuickStart

Lets say we have a model called User with couple of fields:

```
import sqlalchemy as sa
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from wtforms_alchemy import ModelForm

engine = create_engine('sqlite:///:memory:')
Base = declarative_base(engine)
Session = sessionmaker(bind=engine)
session = Session()

class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.BigInteger, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(sa.Unicode(255), nullable=False)
```

Now we can create our first ModelForm for the User model. ModelForm behaves almost like your ordinary WTForms Form except it accepts special Meta arguments. Every ModelForm must define model parameter in the Meta arguments.:

```
class UserForm(ModelForm):
    class Meta:
        model = User
```

Now this ModelForm is essentially the same as

```
class UserForm(Form):
    name = TextField(validators=[DataRequired(), Length(max=100)])
    email = TextField(validators=[DataRequired(), Length(max=255)])
```

In the following chapters you'll learn how WTForms-Alchemy converts SQLAlchemy model columns to form fields.

CHAPTER 2

Column to form field conversion

Basic type conversion

By default WTForms-Alchemy converts SQLAlchemy model columns using the following type table. So for example if an Unicode column would be converted to TextField.

The reason why so many types here convert to wtforms_components based fields is that wtforms_components provides better HTML5 compatible type handling than WTForms at the moment.

SQAlchemy column type	Form field
BigInteger	wtforms_components.fields.IntegerField
Boolean	BooleanField
Date	wtforms_components.fields.DateField
DateTime	wtforms_components.fields.DateTimeField
Enum	wtforms_components.fields.SelectField
Float	FloatField
Integer	wtforms_components.fields.IntegerField
Numeric	wtforms_components.fields.DecimalField
SmallInteger	wtforms_components.fields.IntegerField
String	TextField
Text	TextAreaField
Time	wtforms_components.fields.TimeField
Unicode	TextField
UnicodeText	TextAreaField

WTForms-Alchemy also supports many types provided by SQLAlchemy-Utils.

SQAlchemy-Utils type	Form field
ArrowType	wtforms_components.fields.DateTimeField
ChoiceType	wtforms_components.fields.SelectField
ColorType	wtforms_components.fields.ColorField
CountryType	wtforms_alchemy.fields.CountryType
EmailType	wtforms_components.fields.EmailField
IPAddressType	wtforms_components.fields.IPAddressField
PasswordType	wtforms.fields.PasswordField
PhoneNumberType	wtforms_components.fields.PhoneNumberField
URLType	wtforms_components.fields.StringField + URL validator
UUIDType	wtforms.fields.TextField + UUID validator
WeekDaysType	wtforms_components.fields.WeekDaysField

SQAlchemy-Utils range type	Form field
DateRangeType	wtforms_components.fields.DateIntervalField
DateTimeRangeType	wtforms_components.fields.DateTimeIntervalField
IntRangeType	wtforms_components.fields.IntIntervalField
NumericRangeType	wtforms_components.fields.DecimalIntervalField

Excluded fields

By default WTForms-Alchemy excludes a column from the ModelForm if one of the following conditions is True:

- Column is primary key
- Column is foreign key
- Column is DateTime field which has default value (usually this is a generated value)
- Column is of TSVectorType type
- Column is set as model inheritance discriminator field

Using include, exclude and only

If you wish the include some of the excluded fields described in the earlier chapter you can use the 'include' configuration parameter.

In the following example we include the field 'author_id' in the ArticleForm (by default it is excluded since it is a foreign key column).

```
class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True, nullable=False)
    name = sa.Column(
        sa.Unicode(255),
        nullable=False
    )
    author_id = sa.Column(sa.Integer, sa.ForeignKey(User.id))
    author = sa.orm.relationship(User)


class ArticleForm(Form):
```

```
    class Meta:
        include = ['author_id']
```

If you wish the exclude fields you can either use 'exclude' or 'only' configuration parameters. The recommended way is using only, since in most cases it is desirable to explicitly tell which fields the form should contain.

Consider the following model:

```
class Article(Base):
    __tablename__ = 'article'

    id = sa.Column(sa.Integer, primary_key=True, nullable=False)
    name = sa.Column(
        sa.Unicode(255),
        nullable=False
    )
    content = sa.Column(
        sa.UnicodeText
    )
    description = sa.Column(
        sa.UnicodeText
    )
```

Now let's say we want to exclude 'description' from the form. This can be achieved as follows:

```
class ArticleForm(Form):
    class Meta:
        exclude = ['description']
```

Or as follows (the recommended way):

```
class ArticleForm(Form):
    class Meta:
        only = ['name', 'content']
```

Adding/overriding fields

Example:

```
from wtforms.fields import TextField, IntegerField
from wtforms.validators import Email

class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(
        sa.Unicode(255),
        nullable=False
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

```
    email = TextField(validators=[Optional()])
    age = IntegerField()
```

Now the UserForm would have three fields:

- name, a required TextField
- email, an optional TextField
- age, IntegerField

Type decorators

WTForms-Alchemy supports SQLAlchemy TypeDecorator based types. When WTForms-Alchemy encounters a TypeDecorator typed column it tries to convert it to underlying type field.

Example:

```
import sqlalchemy as sa
from wtforms.fields import TextField, IntegerField
from wtforms.validators import Email


class CustomUnicodeType(sa.types.TypeDecorator):
    impl = sa.types.Unicode

class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    name = sa.Column(CustomUnicodeType(100), primary_key=True)


class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the name field of UserForm would be a simple TextField since the underlying type implementation is Unicode.

CHAPTER 3

Type specific conversion

Numeric type

WTForms-Alchemy automatically converts Numeric columns to DecimalFields. The converter is also smart enough to convert different decimal scales to appropriate HTML5 input step args.

```
class Account(Base):
    __tablename__ = 'event'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    balance = sa.Column(
        sa.Numeric(scale=2),
        nullable=False
    )

class AccountForm(ModelForm):
    class Meta:
        model = Account
```

Now rendering AccountForm.balance would return the following HTML:

<input type='decimal' required step="0.01">

Arrow type

WTForms-Alchemy supports the ArrowType of SQLAlchemy-Utils and converts it to HTML5 compatible DateTime-Field of WTForms-Components.

```
from sqlalchemy_utils import ArrowType


class Event(Base):
    __tablename__ = 'event'
```

```
    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    start_time = sa.Column(
        ArrowType(),
        nullable=False
    )

class EventForm(ModelForm):
    class Meta:
        model = Event
```

Now the EventForm is essentially the same as:

```
class EventForm(Form):
    start_time = DateTimeField(validators=[DataRequired()])
```

Choice type

WTForms-Alchemy automatically converts `sqlalchemy_utils.types.choice.ChoiceType` to WTForms-Components SelectField.

```
from sqlalchemy_utils import ChoiceType


class Event(Base):
    __tablename__ = 'event'
    TYPES = [
        (u'party', u'Party'),
        (u'training, u'Training')
    ]

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    type = sa.Column(ChoiceType(TYPES))


class EventForm(ModelForm):
    class Meta:
        model = Event
```

Now the EventForm is essentially the same as:

```
from wtforms_alchemy.utils import choice_type_coerce_factory


class EventForm(Form):
    type = SelectField(
        choices=Event.TYPES,
        coerce=choice_type_coerce_factory(Event.type.type),
        validators=[DataRequired()]
    )
```

Color type

```
from sqlalchemy_utils import ColorType


class CustomView(Base):
    __tablename__ = 'view'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    background_color = sa.Column(
        ColorType(),
        nullable=False
    )

class CustomViewForm(ModelForm):
    class Meta:
        model = CustomView
```

Now the CustomViewForm is essentially the same as:

```
from wtforms_components import ColorField


class CustomViewForm(Form):
    color = ColorField(validators=[DataRequired()])
```

Country type

```
from sqlalchemy_utils import CountryType


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    country = sa.Column(CountryType, nullable=False)


class UserForm(ModelForm):
    class Meta:
        model = User
```

The UserForm is essentially the same as:

```
from wtforms_components import CountryField


class UserForm(Form):
    country = CountryField(validators=[DataRequired()])
```

Email type

```
from sqlalchemy_utils import EmailType


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    email = sa.Column(EmailType, nullable=False)


class UserForm(ModelForm):
    class Meta:
        model = User
```

The good old wtforms equivalent of this form would be:

```
from wtforms_components import EmailField


class UserForm(Form):
    email = EmailField(validators=[DataRequired()])
```

Password type

Consider the following model definition:

```
from sqlalchemy_utils import PasswordType


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    password = sa.Column(
        PasswordType(
            schemes=['pbkdf2_sha512']
        ),
        nullable=False
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the UserForm is essentially the same as:

```
class UserForm(Form):
    name = TextField(validators=[DataRequired(), Length(max=100)])
    password = PasswordField(validators=[DataRequired()])
```

Phonenumber type

WTForms-Alchemy supports the PhoneNumberType of SQLAlchemy-Utils and converts it automatically to WTForms-Components PhoneNumberField. This field renders itself as HTML5 compatible phonenumber input.

Consider the following model definition:

```
from sqlalchemy_utils import PhoneNumberType


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    phone_number = sa.Column(PhoneNumberType())


class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the UserForm is essentially the same as:

```
from wtforms_components import PhoneNumberField


class UserForm(Form):
    name = TextField(validators=[DataRequired(), Length(max=100)])
    password = PhoneNumberField(validators=[DataRequired()])
```

URL type

WTForms-Alchemy automatically converts SQLAlchemy-Utils URLType to StringField and adds URL validator for it.

Consider the following model definition:

```
from sqlalchemy_utils import URLType


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True, autoincrement=True)
    website = sa.Column(URLType())


class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the UserForm is essentially the same as:

```
from wtforms_components import StringField
from wtforms.validators import URL
```

```
class UserForm(Form):
    website = StringField(validators=[URL()])
```

CHAPTER 4

Form customization

Custom fields

If you want to use a custom field class, you can pass it by using form_field_class parameter for the column info dictionary.

Example

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    color = sa.Column(
        sa.String(7),
        info={'form_field_class': ColorField},
        nullable=False
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the 'color' field of UserForm would be a custom ColorField.

Forcing the use of SelectField

Sometimes you may want to have integer and unicode fields convert to SelectFields. Probably the easiest way to achieve this is by using choices parameter for the column info dictionary.

Example

```
class User(Base):
    __tablename__ = 'user'
```

```
    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    age = sa.Column(
        sa.Integer,
        info={'choices': [(i, i) for i in xrange(13, 99)]},
        nullable=False
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Here the UserForm would have two fields. One TextField for the name column and one SelectField for the age column containing range of choices from 13 to 99.

Notice that WTForms-Alchemy is smart enough to use the right coerce function based on the underlying column type, hence in the previous example the age column would convert to the following SelectField.

```
SelectField('Age', coerce=int, choices=[(i, i) for i in xrange(13, 99)])
```

For nullable unicode and string columns WTForms-Alchemy uses special null_or_unicode coerce function, which converts empty strings to None values.

Field descriptions

Example:

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(
        sa.Unicode(255),
        nullable=False,
        info={'description': 'This is the description of email.'}
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the 'email' field of UserForm would have description 'This is the description of email.'

Field labels

Example:

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(
        sa.Unicode(100), primary_key=True, nullable=False,
        info={'label': 'Name'}
    )
```

```
class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the 'name' field of UserForm would have label 'Name'.

Custom widgets

Example:

```
from wtforms import widgets


class User(Base):
    __tablename__ = 'user'

    name = sa.Column(
        sa.Unicode(100), primary_key=True, nullable=False,
        info={'widget': widgets.HiddenInput()}
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the 'name' field of UserForm would use HiddenInput widget instead of TextInput.

Default values

By default WTForms-Alchemy ModelForm assigns the default values from column definitions. Example

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    level = sa.Column(sa.Integer, default=1)

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the UseForm 'level' field default value would be 1.

CHAPTER 5

Validators

Auto-assigned validators

By default WTForms-Alchemy ModelForm assigns the following validators:

- InputRequired validator if column is not nullable and has no default value
- DataRequired validator if column is not nullable, has no default value and is of type *sqlalchemy.types.String*
- NumberRange validator if column if of type Integer, Float or Decimal and column info parameter has min or max arguments defined
- DateRange validator if column is of type Date or DateTime and column info parameter has min or max arguments defined
- TimeRange validator if column is of type Time and info parameter has min or max arguments defined
- Unique validator if column has a unique index
- Length validator for String/Unicode columns with max length
- Optional validator for all nullable columns

Unique validator

WTForms-Alchemy automatically assigns unique validators for columns which have unique indexes defined. Unique validator raises ValidationError exception whenever a non-unique value for given column is assigned. Consider the following model/form definition. Notice how you need to define get_session() classmethod for your form. Unique validator uses this method for getting the appropriate SQLAlchemy session.

```
engine = create_engine('sqlite:///:memory:')

Base = declarative_base()
```

```
Session = sessionmaker(bind=engine)
session = Session()


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100), nullable=False)
    email = sa.Column(
        sa.Unicode(255),
        nullable=False,
        unique=True
    )

class UserForm(ModelForm):
    class Meta:
        model = User

    @classmethod
    def get_session():
        # this method should return sqlalchemy session
        return session
```

Here UserForm would behave the same as the following form:

```
class UserForm(Form):
    name = TextField('Name', validators=[DataRequired(), Length(max=100)])
    email = TextField(
        'Email',
        validators=[
            DataRequired(),
            Length(max=255),
            Unique(User.email, get_session=lambda: session)
        ]
    )
```

If you are using Flask-SQLAlchemy or similar tool, which assigns session-bound query property to your declarative models, you don't need to define the get_session() method. Simply use:

```
Unique(User.email)
```

Using unique validator with existing objects

When editing an existing object, WTForms-Alchemy must know the object currently edited to avoid raising a ValidationError. Here how to proceed to inform WTForms-Alchemy of this case. Example:

```
obj = MyModel.query.get(1)
form = MyForm(obj=obj)
form.populate_obj(obj)
form.validate()
```

WTForms-Alchemy will then understand to avoid the unique validation of the object with this same object.

Range validators

WTForms-Alchemy automatically assigns range validators based on column type and assigned column info min and max attributes.

In the following example we create a form for Event model where start_time can't be set in the past.

```
class Event(Base):
    __tablename__ = 'event'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255))
    start_time = sa.Column(sa.DateTime, info={'min': datetime.now()})


class EventForm(ModelForm):
    class Meta:
        model = Event
```

Additional field validators

Example:

```
from wtforms.validators import Email

class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(
        sa.Unicode(255),
        nullable=False,
        info={'validators': Email()}
    )

class UserForm(ModelForm):
    class Meta:
        model = User
```

Now the 'email' field of UserForm would have Email validator.

Overriding default validators

Sometimes you may want to override what class WTForms-Alchemy uses for email, number_range, length etc. validations. For all automatically assigned validators WTForms-Alchemy provides configuration options to override the default validator.

In the following example we set a custom Email validator for User class.

```
from sqlalchemy_utils import EmailType
from wtforms_components import Email
```

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(
        EmailType,
        nullable=False,
    )

class MyEmailValidator(Email):
    def __init__(self, message='My custom email error message'):
        Email.__init__(self, message=message)


class UserForm(ModelForm):
    class Meta:
        model = User
        email_validator = MyEmailValidator
```

If you don't wish to subclass you can simply use functions / lambdas:

```
def email():
    return Email(message='My custom email error message')


class UserForm(ModelForm):
    class Meta:
        model = User
        email_validator = email
```

You can also override validators that take multiple arguments this way:

```
def length(min=None, max=None):
    return Length(min=min, max=max, message='Wrong length')


class UserForm(ModelForm):
    class Meta:
        model = User
        length_validator = length
```

Here is the full list of configuration options you can use to override default validators:

- email_validator
- length_validator
- unique_validator
- number_range_validator
- date_range_validator
- time_range_validator
- optional_validator

Disabling validators

You can disable certain validators by assigning them as *None*. Let's say you want to disable nullable columns having *Optional* validator. This can be achieved as follows:

```
class UserForm(ModelForm):
    class Meta:
        model = User
        optional_validator = None
```

CHAPTER 6

Configuration

ModelForm meta parameters

The following configuration options are available for ModelForm's Meta subclass.

include_primary_keys (default: False)

If you wish to include primary keys in the generated form please set this to True. This is useful when dealing with natural primary keys. In the following example each user has a natural primary key on its column name.

The UserForm would contain two fields name and email.

```
class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(sa.Unicode(255), nullable=False)


class UserForm(ModelForm):
    class Meta:
        model = User
        include_primary_keys = True
```

exclude

Warning: Using `exclude` might lead to problems in situations where you add columns to your model and forget to exclude those from the form by using `exclude`, hence it is recommended to use `only` rather than `exclude`.

You can exclude certain fields by adding them to the exclude list.

```
class User(Base):
    __tablename__ = 'user'
```

```
    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(sa.Unicode(255), nullable=False)


class UserForm(ModelForm):
    class Meta:
        model = User
        include_primary_keys = True
        exclude = ['email']
        # this form contains only 'name' field
```

only

Generates a form using only the field names provided in `only`.

```
class UserForm(ModelForm):
    class Meta:
        model = User
        only = ['email']
```

field_args (default: {})

This parameter can be used for overriding field arguments. In the following example we force the email field optional.

```
class UserForm(ModelForm):
   class Meta:
       model = User
       field_args = {'email': {'validators': [Optional()]}}
```

include_foreign_keys (default: False)

Foreign keys can be included in the form by setting include_foreign_keys to True.

only_indexed_fields (default: False)

When setting this option to True, only fields that have an index will be included in the form. This is very useful when creating forms for searching a specific model.

include_datetimes_with_default (default: False)

When setting this option to True, datetime with default values will be included in the form. By default this is False since usually datetime fields that have default values are generated columns such as “created_at” or “updated_at”, which should not be included in the form.

validators

A dict containing additional validators for the generated form field objects.

Example:

```
from wtforms.validators import Email


class User(Base):
    __tablename__ = 'user'

    name = sa.Column(sa.Unicode(100), primary_key=True, nullable=False)
    email = sa.Column(sa.Unicode(255), nullable=False)


class UserForm(ModelForm):
```

```
    class Meta:
        model = User
        include_primary_keys = True
        validators = {'email': [Email()]}
```

datetime_format (default: '%Y-%m-%d %H:%M:%S')

Defines the default datetime format, which will be assigned to generated datetime fields.

date_format (default: '%Y-%m-%d')

Defines the default date format, which will be assigned to generated datetime fields.

all_fields_optional (default: False)

Defines all generated fields as optional (useful for update forms).

assign_required (default: True)

Whether or not to assign non-nullable fields as required.

strip_string_fields (default: False)

Whether or not to add stripping filter to all string fields.

Example

```
from werkzeug.datastructures import MultiDict


class UserForm(ModelForm):
    class Meta:
        model = User
        strip_string_fields = True


form = UserForm(MultiDict([('name', 'someone     ')]))

assert form.name.data == 'someone'
```

You can also fine-grain field stripping by using trim argument for columns. In the example below the field 'name' would have its values stripped whereas field 'password' would not.

```
from wtforms.validators import Email


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100))
    password = sa.Column(sa.Unicode(100), info={'trim': False})


class UserForm(ModelForm):
    class Meta:
        model = User
        strip_string_fields = True
```

form_generator (default: FormGenerator class)

Change this if you want to use custom form generator class.

Form inheritance

ModelForm's configuration support inheritance. This means that child classes inherit parents Meta properties.

Example:

```
from wtforms.validators import Email


class UserForm(ModelForm):
    class Meta:
        model = User
        validators = {'email': [Email()]}


class UserUpdateForm(UserForm):
    class Meta:
        all_fields_optional = True
```

Here UserUpdateForm inherits the configuration properties of UserForm, hence it would use model User and have additional Email validator on column 'email'. Also it assigns all fields as optional.

Not nullable column validation

WTForms-Alchemy offers two options for configuring how not nullable columns are validated:

- not_null_validator

  The default validator to be used for not nullable columns. Set this to *None* if you wish to disable it. By default this is *[InputRequired()]*.

- not_null_validator_type_map

  Type map which overrides the **not_null_validator** on specific column type. By default this is *ClassMap({sa.String: [InputRequired(), DataRequired()]})*.

In the following example we set *DataRequired* validator for all not nullable Enum typed columns:

```
import sqlalchemy as sa
from wtforms.validators import DataRequired
from wtforms_alchemy import ClassMap


class MyForm(ModelForm):
    class Meta:
        not_null_validator_type_map = ClassMap({sa.Enum: [DataRequired()]})
```

Customizing type conversion

You can customize the SQLAlchemy type conversion on class level with type_map Meta property.

Type map accepts dictionary of SQLAlchemy types as keys and WTForms field classes as values. The key value pairs of this dictionary override the key value pairs of FormGenerator.TYPE_MAP.

Let's say we want to convert all unicode typed properties to TextAreaFields instead of StringFields. We can do this by assigning Unicode, TextAreaField key value pair into type map.

```
from wtforms.fields import TextAreaField
from wtforms_alchemy import ClassMap


class User(Base):
    __tablename__ = 'user'

    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(100))


class UserForm(ModelForm):
    class Meta:
        type_map = ClassMap({sa.Unicode: TextAreaField})
```

In case the type_map dictionary values are not inherited from WTForm field class, they are considered callable functions. These functions will be called with the corresponding column as their only parameter.

Custom form base class

You can use custom base class for your model forms by using model_form_factory function. In the following example we have a UserForm which uses Flask-WTF form as a parent form for ModelForm.

```
from flask.ext.wtf import Form
from wtforms_alchemy import model_form_factory


ModelForm = model_form_factory(Form)


class UserForm(ModelForm):
    class Meta:
        model = User
```

You can also pass any form generator option to model_form_factory.

```
ModelForm = model_form_factory(Form, strip_string_fields=True)


class UserForm(ModelForm):
    class Meta:
        model = User
```

CHAPTER 7

Forms with relations

WTForms-Alchemy provides special Field subtypes ModelFormField and ModelFieldList. When using these types WTForms-Alchemy understands model relations and is smart enough to populate related objects accordingly.

One-to-one relations

Consider the following example. We have Event and Location classes with each event having one location.

```
from sqlalchemy.ext.declarative import declarative_base
from wtforms_alchemy import ModelForm, ModelFormField

Base = declarative_base()


class Location(Base):
    __tablename__ = 'location'
    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=True)

class Event(Base):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=False)
    location_id = sa.Column(sa.Integer, sa.ForeignKey(Location.id))
    location = sa.orm.relationship(Location)

class LocationForm(ModelForm):
    class Meta:
        model = Location

class EventForm(ModelForm):
    class Meta:
        model = Event
```

```
    location = ModelFormField(LocationForm)
```

Now if we populate the EventForm, WTForms-Alchemy is smart enough to populate related location too.

```
event = Event()
form = EventForm(request.POST)
form.populate_obj(event)
```

One-to-many relations

Consider the following example. We have Event and Location classes with each event having many location. Notice we are using FormField along with ModelFieldList.

```
from sqlalchemy.ext.declarative import declarative_base
from wtforms_alchemy import ModelForm, ModelFieldList
from wtforms.fields import FormField

Base = declarative_base()


class Event(Base):
    __tablename__ = 'event'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=False)


class Location(Base):
    __tablename__ = 'location'
    id = sa.Column(sa.Integer, autoincrement=True, primary_key=True)
    name = sa.Column(sa.Unicode(255), nullable=True)

    event_id = sa.Column(sa.Integer, sa.ForeignKey(Event.id))
    event = sa.orm.relationship(
        Location,
        backref='locations'  # the event needs to have this
    )


class LocationForm(ModelForm):
    class Meta:
        model = Location


class EventForm(ModelForm):
    class Meta:
        model = Event

    locations = ModelFieldList(FormField(LocationForm))
```

Now if we populate the EventForm, WTForms-Alchemy is smart enough to populate related locations too.

```
event = Event()
form = EventForm(request.POST)
form.populate_obj(event)
```

CHAPTER 8

Advanced concepts

Using WTForms-Alchemy with SQLAlchemy-Defaults

WTForms-Alchemy works wonderfully with SQLAlchemy-Defaults. When using SQLAlchemy-Defaults with WTForms-Alchemy you can define your models and model forms with much more robust syntax. For more information see SQLAlchemy-Defaults documentation.

Example

```
from sqlalchemy_defaults import LazyConfigured


class User(Base, LazyConfigured):
    __tablename__ = 'user'
    id = sa.Column(sa.Integer, primary_key=True)
    name = sa.Column(
        sa.Unicode(255),
        nullable=False,
        label=u'Name'
    )
    age = sa.Column(
        sa.Integer,
        nullable=False,
        min=18,
        max=100,
        label=u'Age'
    )


class UserForm(ModelForm):
    class Meta:
        model = User
```

Using WTForms-Alchemy with Flask-WTF

In order to make WTForms-Alchemy work with Flask-WTF you need the following snippet:

```
from flask_wtf import FlaskForm
from wtforms_alchemy import model_form_factory
# The variable db here is a SQLAlchemy object instance from
# Flask-SQLAlchemy package
from myproject.extensions import db

BaseModelForm = model_form_factory(FlaskForm)

class ModelForm(BaseModelForm):
    @classmethod
    def get_session(self):
        return db.session
```

Then you can use the ModelForm just like before:

```
class UserForm(ModelForm):
    class Meta:
        model = User
```

CHAPTER 9

API Documentation

This part of the documentation covers all the public classes and functions in WTForms-Alchemy.

wtforms_alchemy

class wtforms_alchemy.**ModelForm**(**args*, ***kwargs*)

Standard base-class for all forms to be combined with a model. Use *model_form_factory()* in case you wish to change its behavior.

get_session: If you want to use the Unique validator, you should define this method. If you are using Flask-SQLAlchemy along with WTForms-Alchemy you don't need to set this. If you define this in the superclass, it will not be overriden.

wtforms_alchemy.**model_form_factory**(*base=<class 'wtforms.form.Form'>*, *meta=<class 'wtforms_alchemy.ModelFormMeta'>*, ***defaults*)

Create a base class for all model forms to derive from.

Parameters

- **base** – Class that should be used as a base for the returned class. By default, this is WTForms's base class wtforms.Form.
- **meta** – A metaclass to use on this class. Normally, you do not need to provide this value, but if you want, you should check out *model_form_meta_factory()*.

Returns A class to be used as the base class for all forms that should be connected to a SQLAlchemy model class.

Additional arguments provided to the form override the default configuration as described in *Custom form base class*.

class wtforms_alchemy.**ModelFormMeta**(**args*, ***kwargs*)

Meta class that overrides WTForms base meta class. The primary purpose of this class is allowing ModelForms use special configuration params under the 'Meta' class namespace.

ModelForm classes inherit parent's Meta class properties.

wtforms_alchemy.**model_form_meta_factory**(*base=<class 'wtforms.form.FormMeta'>*)

Create a new class usable as a metaclass for the *model_form_factory()*. You only need to concern yourself with this if you desire to have a custom metclass. Otherwise, a default class is created and is used as a metaclass on *model_form_factory()*.

Parameters **base** – The base class to use for the meta class. This is an optional parameter that defaults to FormMeta. If you want to provide your own, your class must derive from this class and not directly from type.

Returns A new class suitable as a metaclass for the actual model form. Therefore, it should be passed as the meta argument to *model_form_factory()*.

Example usage:

```
from wtforms.form import FormMeta


class MyModelFormMeta(FormMeta):
    # do some metaclass magic here
    pass

ModelFormMeta = model_form_meta_factory(MyModelFormMeta)
ModelForm = model_form_factory(meta=ModelFormMeta)
```

wtforms_alchemy.generator

class wtforms_alchemy.generator.**FormGenerator**(*form_class*)

Base form generator, you can make your own form generators by inheriting this class.

additional_validators(*key*, *column*)

Returns additional validators for given column

Parameters

- **key** – String key of the column property
- **column** – SQLAlchemy Column object

coerce(*column*)

Returns coerce callable for given column

Parameters **column** – SQLAlchemy Column object

create_field(*prop*, *column*)

Create form field for given column.

Parameters

- **prop** – SQLAlchemy ColumnProperty object.
- **column** – SQLAlchemy Column object.

create_fields(*form*, *properties*)

Creates fields for given form based on given model attributes.

Parameters

- **form** – form to attach the generated fields into
- **attributes** – model attributes to generate the form fields from

create_form(*form*)
Creates the form.

Parameters **form** – ModelForm instance

create_validators(*prop*, *column*)
Returns validators for given column

Parameters **column** – SQLAlchemy Column object

date_format(*column*)
Returns date format for given column.

Parameters **column** – SQLAlchemy Column object

default(*column*)
Return field default for given column.

Parameters **column** – SQLAlchemy Column object

filter_attributes(*attrs*)
Filter set of model attributes based on only, exclude and include meta parameters.

Parameters **attrs** – Set of attributes

filters(*column*)
Return filters for given column.

Parameters **column** – SQLAlchemy Column object

get_field_class(*column*)
Returns WTForms field class. Class is based on a custom field class attribute or SQLAlchemy column type.

Parameters **column** – SQLAlchemy Column object

has_index(*column*)
Whether or not given column has an index.

Parameters **column** – Column object to inspect the indexes from

length_validator(*column*)
Returns length validator for given column

Parameters **column** – SQLAlchemy Column object

range_validator(*column*)
Returns range validator based on column type and column info min and max arguments

Parameters **column** – SQLAlchemy Column object

required_validator(*column*)
Returns required / optional validator for given column based on column nullability and form configuration.

Parameters **column** – SQLAlchemy Column object

scale_to_step(*scale*)
Returns HTML5 compatible step attribute for given decimal scale.

Parameters **scale** – an integer that defines a Numeric column's scale

select_field_kwargs(*column*)
Returns key value args for SelectField based on SQLAlchemy column definitions.

Parameters **column** – SQLAlchemy Column object

skip_column(*column*)
Whether or not to skip column in the generation process.

Parameters **column_property** – SQLAlchemy Column object

skip_column_property(*column_property*)
Whether or not to skip column property in the generation process.

Parameters **column_property** – SQLAlchemy ColumnProperty object

type_agnostic_parameters(*key*, *column*)
Returns all type agnostic form field parameters for given column.

Parameters **column** – SQLAlchemy Column object

type_specific_parameters(*column*)
Returns type specific parameters for given column.

Parameters **column** – SQLAlchemy Column object

unique_validator(*key*, *column*)
Returns unique validator for given column if column has a unique index

Parameters

- **key** – String key of the column property
- **column** – SQLAlchemy Column object

validate_attribute(*attr_name*)
Finds out whether or not given sqlalchemy model attribute name is valid. Returns attribute property if valid.

Parameters **attr_name** – Attribute name

widget(*column*)
Returns WTForms widget for given column.

Parameters **column** – SQLAlchemy Column object

wtforms_alchemy.fields

class wtforms_alchemy.fields.**QuerySelectField**(*label=None*, *validators=None*, *query_factory=None*, *get_pk=None*, *get_label=None*, *allow_blank=False*, *blank_text=u''*, ***kwargs*)
Will display a select drop-down field to choose between ORM results in a sqlalchemy *Query*. The *data* property actually will store/keep an ORM model instance, not the ID. Submitting a choice which is not in the query will result in a validation error. This field only works for queries on models whose primary key column(s) have a consistent string representation. This means it mostly only works for those composed of string, unicode, and integer types. For the most part, the primary keys will be auto-detected from the model, alternately pass a one-argument callable to *get_pk* which can return a unique comparable key. The *query* property on the field can be set from within a view to assign a query per-instance to the field. If the property is not set, the *query_factory* callable passed to the field constructor will be called to obtain a query. Specify *get_label* to customize the label associated with each option. If a string, this is the name of an attribute on the model object to use as the label text. If a one-argument callable, this callable will be passed model instance and expected to return the label text. Otherwise, the model object's *__str__* or *__unicode__* will be used. If *allow_blank* is set to *True*, then a blank choice will be added to the top of the list. Selecting this choice will result in the *data* property being *None*. The label for this blank choice can be set by specifying the *blank_text* parameter.

class wtforms_alchemy.fields.**QuerySelectMultipleField**(*label=None*, *validators=None*, *default=None*, ***kwargs*)

Very similar to QuerySelectField with the difference that this will display a multiple select. The data property will hold a list with ORM model instances and will be an empty list when no value is selected. If any of the items in the data list or submitted form data cannot be found in the query, this will result in a validation error.

wtforms_alchemy.utils

wtforms_alchemy.utils.**translated_attributes**(*model*)

Return translated attributes for current model class. See SQLAlchemy-i18n package for more information about translatable attributes.

Parameters **model** – SQLAlchemy declarative model class

class wtforms_alchemy.utils.**ClassMap**(*items=None*)

An ordered dictionary with keys as classes. ClassMap has the following charasteristics:

1. Checking if a key exists not only matches exact classes but also subclasses and objects which are instances of a ClassMap key.

2. Getting an item of ClassMap with a key matches subclasses and instances also.

__contains__(*key*)

Checks if given key exists in by first trying to find an exact match. If no exact match is found then this method iterates trhough keys and tries to check if given key is either:

1.A subclass of one of the keys

2.An instance of one of the keys

The first check has the time complexity of O(1) whereas the second check has O(n).

Example:

```
class A(object):
    pass


class B(object):
    pass


class A2(A):
    pass


class_map = ClassMap({A: 1, B: 2})
assert B in class_map
assert A in class_map
assert A2 in class_map
assert B() in class_map
assert A() in class_map
assert A2() in class_map
```

__getitem__(*key*)

Returns the item matching a key. The key matching has the same charasteristics as __contains__ method.

Example:

```
class A(object):
    pass


class B(object):
    pass


class A2(A):
    pass


class_map = ClassMap({A: 1, B: 2})
assert class_map[B] == 2
assert class_map[A] == 1
assert class_map[A2] == 1
assert class_map[B()] == 2
assert class_map[A()] == 1
assert class_map[A2()] == 1
```

CHAPTER 10

License

Python Module Index

W

Index

Symbols

A

C

D

F

G

H

L

M

Q

R

S