WTForms Documentation

Release 2.0.2dev

Thomas Johansson, James Crasta

August 10, 2014

This is the documentation for WTForms 2.0.2dev, generated July 01, 2014.

For a quick introduction, as well as download/installation instructions, check out the *Crash Course*.

Coming from WTForms 1.x? Check out *What's New in WTForms 2*.

API

CHAPTER 1

Forms

Forms provide the highest level API in WTForms. They contain your field definitions, delegate validation, take input, aggregate errors, and in general function as the glue holding everything together.

1.1 The Form class

class `wtforms.form.`**`Form`**

Declarative Form base class.

Construction

`__init__`(*formdata=None*, *obj=None*, *prefix=''*, *data=None*, *meta=None*, ***kwargs*)

Parameters

- **formdata** – Used to pass data coming from the enduser, usually *request.POST* or equivalent. formdata should be some sort of request-data wrapper which can get multiple parameters from the form input, and values are unicode strings, e.g. a Werkzeug/Django/WebOb MultiDict
- **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.
- **prefix** – If provided, all fields will have their name prefixed with the value.
- **data** – Accept a dictionary of data. This is only used if *formdata* and *obj* are not present.
- **meta** – If provided, this is a dictionary of values to override attributes on this form's meta instance.
- ****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.

Initialize a Form. This is usually done in the context of a view/controller in your application. When a Form is constructed, the fields populate their input based on the formdata, obj, and kwargs.

Note Backing-store objects and kwargs are both expected to be provided with the values being already-coerced datatypes. WTForms does not check the types of incoming object-data or coerce them like it will for *formdata* as it is expected this data is defaults or data from a backing store which this form represents. See the section on *using Forms* for more information.

Properties

data
A dict containing the data for each field.

Note that this is generated each time you access the property, so care should be taken when using it, as it can potentially be very expensive if you repeatedly access it. Typically used if you need to iterate all data in the form. If you just need to access the data for known fields, you should use *form.<field>.data*, not this proxy property.

errors
A dict containing a list of errors for each field. Empty if the form hasn't been validated, or there were no errors.

Note that this is a lazy property, and will only be generated when you first access it. If you call `validate()` after accessing it, the cached result will be invalidated and regenerated on next access.

meta
This is an object which contains various configuration options and also ability to customize the behavior of the form. See the *class Meta* doc for more information on what can be customized with the class Meta options.

Methods

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

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.

One common usage of this is an edit profile view:

```
def edit_profile(request):
    user = User.objects.get(pk=request.session['userid'])
    form = EditProfileForm(request.POST, obj=user)

    if request.POST and form.validate():
        form.populate_obj(user)
        user.save()
        return redirect('/home')
    return render_to_response('edit_profile.html', form=form)
```

In the above example, because the form isn't directly tied to the user object, you don't have to worry about any dirty data getting onto there until you're ready to move it over.

__iter__()
Iterate form fields in creation order.

```
{% for field in form %}
    <tr>
        <th>{{ field.label }}</th>
        <td>{{ field }}</td>
    </tr>
{% endfor %}
```

__contains__(*name*)
Returns *True* if the named field is a member of this form.

_get_translations()

Deprecated since version 2.0: *_get_translations* is being removed in WTForms 3.0, use *Meta.get_translations* instead.

Override in subclasses to provide alternate translations factory.

Must return an object that provides gettext() and ngettext() methods.

1.2 Defining Forms

To define a form, one makes a subclass of `Form` and defines the fields declaratively as class attributes:

```
class MyForm(Form):
    first_name = StringField(u'First Name', validators=[validators.input_required()])
    last_name  = StringField(u'Last Name', validators=[validators.optional()])
```

Field names can be any valid python identifier, with the following restrictions:

- Field names are case-sensitive.
- Field names may not begin with "_" (underscore)
- Field names may not begin with "validate"

1.2.1 Form Inheritance

Forms may subclass other forms as needed. The new form will contain all fields of the parent form, as well as any new fields defined on the subclass. A field name re-used on a subclass causes the new definition to obscure the original.

```
class PastebinEdit(Form):
    language = SelectField(u'Programming Language', choices=PASTEBIN_LANGUAGES)
    code     = TextAreaField()

class PastebinEntry(PastebinEdit):
    name = StringField(u'User Name')
```

1.2.2 In-line Validators

In order to provide custom validation for a single field without needing to write a one-time-use validator, validation can be defined inline by defining a method with the convention *validate_fieldname*:

```
class SignupForm(Form):
    age = IntegerField(u'Age')

    def validate_age(form, field):
        if field.data < 13:
            raise ValidationError("We're sorry, you must be 13 or older to register")
```

1.3 Using Forms

A form is most often constructed in the controller code for handling an action, with the form data wrapper from the framework passed to its constructor, and optionally an ORM object. A typical view begins something like:

```
def edit_article(request):
    article = Article.get(...)
    form = MyForm(request.POST, article)
```

A typical CRUD view has a user editing an object that needs various fields updated. The Form would have fields describing the fields to be updated and the validation rules, where the attribute names of the fields match those of the attribute names on the object. The second parameter to the Form, the *obj* parameter, is used to populate form defaults on the initial view.

Note: While we did pass an object as the data source, this object data is only used if there is no POST data. If there is any POST data at all, then the object data is ignored. This is done for security and consistency reasons.

This pattern is mostly a convenience since most application controllers don't separate GET and POST requests into separate view methods.

The constructed form can then validate any input data and generate errors if invalid. Typically, the validation pattern in the view looks like:

```
if request.POST and form.validate():
    form.populate_obj(article)
    article.save()
    return redirect('/articles')
```

Note that we have it so `validate()` is only called if there is POST data. The reason we gate the validation check this way is that when there is no POST data (such as in a typical CRUD form) we don't want to cause validation errors.

Inside the gated block, we call `populate_obj()` to copy the data onto fields on the 'article' object. We also then redirect after a successful completion. The reason we redirect after the post is a best-practice associated with the Post/Redirect/Get design pattern.

If there is no POST data, or the data fails to validate, then the view "falls through" to the rendering portion. The Form object can be passed into the template and its attributes can be used to render the fields and also for displaying errors:

```
return render('edit.html', form=form, article=article)
```

So there we have a full simple "edit object" page setup which illustrates a best-practice way of using WTForms. This is by no means the only way to use WTForms, but just an illustration of how the various features work.

Here is the full code for the view we just made:

```
def edit_article(request):
    article = Article.get(...)
    form = MyForm(request.POST, article)

    if request.POST and form.validate():
        form.populate_obj(article)
        article.save()
        return redirect('/articles')

    return render('edit.html', form=form, article=article)
```

1.4 Low-Level API

Warning: This section is provided for completeness; and is aimed at authors of complementary libraries and those looking for very special behaviors. Don't use *BaseForm* unless you know exactly *why* you are using it.

For those looking to customize how WTForms works, for libraries or special applications, it might be worth using the `BaseForm` class. *BaseForm* is the parent class of `Form`, and most of the implementation logic from Form is actually handled by BaseForm.

The major difference on the surface between *BaseForm* and *Form* is that fields are not defined declaratively on a subclass of *BaseForm*. Instead, you must pass a dict of fields to the constructor. Likewise, you cannot add fields by inheritance. In addition, *BaseForm* does not provide: sorting fields by definition order, or inline *validate_foo* validators. Because of this, for the overwhelming majority of uses we recommend you use Form instead of BaseForm in your code.

What *BaseForm* provides is a container for a collection of fields, which it will bind at instantiation, and hold in an internal dict. Dict-style access on a BaseForm instance will allow you to access (and modify) the enclosed fields.

class `wtforms.form.`**`BaseForm`**

Base Form Class. Provides core behaviour like field construction, validation, and data and error proxying.

Construction

`__init__`(*fields*, *prefix=''*, *meta=<DefaultMeta>*)

Parameters

- **fields** – A dict or sequence of 2-tuples of partially-constructed fields.
- **prefix** – If provided, all fields will have their name prefixed with the value.
- **meta** – A meta instance which is used for configuration and customization of WTForms behaviors.

```
form = BaseForm({
    'name': StringField(),
    'customer.age': IntegerField("Customer's Age")
})
```

Because BaseForm does not require field names to be valid identifiers, they can be most any python string. We recommend keeping it simple to avoid incompatibility with browsers and various form input frameworks where possible.

Properties

`data`

see `Form.data`

`errors`

see `Form.errors`

Methods

`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.

Since BaseForm does not take its data at instantiation, you must call this to provide form data to the enclosed fields. Accessing the field's data before calling process is not recommended.

validate(*extra_validators=None*)

Validates the form by calling *validate* on each field.

Parameters **extra_validators** – If provided, is a dict mapping field names to a sequence of callables which will be passed as extra validators to the field's *validate* method.

Returns *True* if no errors occur.

__iter__()

Iterate form fields in creation order.

Unlike `Form`, fields are not iterated in definition order, but rather in whatever order the dict decides to yield them.

__contains__(*name*)

Returns *True* if the named field is a member of this form.

__getitem__(*name*)

Dict-style access to this form's fields.

__setitem__(*name*, *value*)

Bind a field to this form.

```
form['openid.name'] = StringField()
```

Fields can be added and replaced in this way, but this must be done **before** `process()` is called, or the fields will not have the opportunity to receive input data. Similarly, changing fields after `validate()` will have undesired effects.

__delitem__(*name*)

Remove a field from this form.

The same caveats apply as with `__setitem__()`.

CHAPTER 2

Fields

Fields are responsible for rendering and data conversion. They delegate to validators for data validation.

2.1 Field definitions

Fields are defined as members on a form in a declarative fashion:

```
class MyForm(Form):
    name    = StringField(u'Full Name', [validators.required(), validators.length(max=10)])
    address = TextAreaField(u'Mailing Address', [validators.optional(), validators.length(max=200)])
```

When a field is defined on a form, the construction parameters are saved until the form is instantiated. At form instantiation time, a copy of the field is made with all the parameters specified in the definition. Each instance of the field keeps its own field data and errors list.

The label and validators can be passed to the constructor as sequential arguments, while all other arguments should be passed as keyword arguments. Some fields (such as `SelectField`) can also take additional field-specific keyword arguments. Consult the built-in fields reference for information on those.

2.2 The Field base class

class `wtforms.fields.`**`Field`**

Stores and processes data, and generates HTML for a form field.

Field instances contain the data of that instance as well as the functionality to render it within your Form. They also contain a number of properties which can be used within your templates to render the field and label.

Construction

`__init__`(*label=None*, *validators=None*, *filters=()*, *description=u''*, *id=None*, *default=None*, *widget=None*, *_form=None*, *_name=None*, *_prefix=u''*, *_translations=None*, *_meta=None*)

Construct a new field.

Parameters

- **label** – The label of the field.
- **validators** – A sequence of validators to call when *validate* is called.
- **filters** – A sequence of filters which are run on input data by *process*.
- **description** – A description for the field, typically used for help text.

- **id** – An id to use for the field. A reasonable default is set by the form, and you shouldn't need to set this manually.
- **default** – The default value to assign to the field, if no form or object input is provided. May be a callable.
- **widget** – If provided, overrides the widget used to render the field.
- **_form** – The form holding this field. It is passed by the form itself during construction. You should never pass this value yourself.
- **_name** – The name of this field, passed by the enclosing form during its construction. You should never pass this value yourself.
- **_prefix** – The prefix to prepend to the form name of this field, passed by the enclosing form during construction.
- **_translations** – A translations object providing message translations. Usually passed by the enclosing form during construction. See *I18n docs* for information on message translations.
- **_meta** – If provided, this is the 'meta' instance from the form. You usually don't pass this yourself.

If *_form* and *_name* isn't provided, an `UnboundField` will be returned instead. Call its `bind()` method with a form instance and a name to construct the field.

Validation

To validate the field, call its *validate* method, providing a form and any extra validators needed. To extend validation behaviour, override *pre_validate* or *post_validate*.

`validate` (*form*, *extra_validators=()*)

Validates the field and returns True or False. *self.errors* will contain any errors raised during validation. This is usually only called by *Form.validate*.

Subfields shouldn't override this, but rather override either *pre_validate*, *post_validate* or both, depending on needs.

Parameters

- **form** – The form the field belongs to.
- **extra_validators** – A sequence of extra validators to run.

`pre_validate` (*form*)

Override if you need field-level validation. Runs before any other validators.

Parameters form – The form the field belongs to.

`post_validate` (*form*, *validation_stopped*)

Override if you need to run any field-level validation tasks after normal validation. This shouldn't be needed in most cases.

Parameters

- **form** – The form the field belongs to.
- **validation_stopped** – *True* if any validator raised StopValidation.

`errors`

If *validate* encounters any errors, they will be inserted into this list.

Data access and processing

To handle incoming data from python, override *process_data*. Similarly, to handle incoming data from the outside, override *process_formdata*.

process (*formdata*[, *data*])

Process incoming data, calling process_data, process_formdata as needed, and run filters.

If *data* is not provided, process_data will be called on the field's default.

Field subclasses usually won't override this, instead overriding the process_formdata and process_data methods. Only override this for special advanced processing, such as when a field encapsulates many inputs.

process_data (*value*)

Process the Python data applied to this field and store the result.

This will be called during form construction by the form's *kwargs* or *obj* argument.

Parameters **value** – The python object containing the value to process.

process_formdata (*valuelist*)

Process data received over the wire from a form.

This will be called during form construction with data supplied through the *formdata* argument.

Parameters **valuelist** – A list of strings to process.

data

Contains the resulting (sanitized) value of calling either of the process methods. Note that it is not HTML escaped when using in templates.

raw_data

If form data is processed, is the valuelist given from the formdata wrapper. Otherwise, *raw_data* will be *None*.

object_data

This is the data passed from an object or from kwargs to the field, stored unmodified. This can be used by templates, widgets, validators as needed (for comparison, for example)

Rendering

To render a field, simply call it, providing any values the widget expects as keyword arguments. Usually the keyword arguments are used for extra HTML attributes.

__call__ (***kwargs*)

Render this field as HTML, using keyword args as additional attributes.

This delegates rendering to `meta.render_field` whose default behavior is to call the field's widget, passing any keyword arguments from this call along to the widget.

In all of the WTForms HTML widgets, keyword arguments are turned to HTML attributes, though in theory a widget is free to do anything it wants with the supplied keyword arguments, and widgets don't have to even do anything related to HTML.

If one wants to pass the "class" argument which is a reserved keyword in some python-based templating languages, one can do:

```
form.field(class_="text_blob")
```

This will output (for a text field):

```
<input type="text" name="field_name" value="blah" class="text_blob" id="field_name" />
```

Note: Simply coercing the field to a string or unicode will render it as if it was called with no arguments.

`__html__`()

Returns a HTML representation of the field. For more powerful rendering, see the `__call__()` method.

Many template engines use the __html__ method when it exists on a printed object to get an 'html-safe' string that will not be auto-escaped. To allow for printing a bare field without calling it, all WTForms fields implement this method as well.

Message Translations

`gettext`(*string*)

Get a translation for the given message.

This proxies for the internal translations object.

Parameters **string** – A unicode string to be translated.

Returns A unicode string which is the translated output.

`ngettext`(*singular*, *plural*, *n*)

Get a translation for a message which can be pluralized.

Parameters

- **singular** (*str*) – The singular form of the message.
- **plural** (*str*) – The plural form of the message.
- **n** (*int*) – The number of elements this message is referring to

Properties

`name`

The HTML form name of this field. This is the name as defined in your Form prefixed with the *prefix* passed to the Form constructor.

`short_name`

The un-prefixed name of this field.

`id`

The HTML ID of this field. If unspecified, this is generated for you to be the same as the field name.

`label`

This is a `Label` instance which when evaluated as a string returns an HTML `<label for="id">` construct.

`default`

This is whatever you passed as the *default* to the field's constructor, otherwise None.

`description`

A string containing the value of the description passed in the constructor to the field; this is not HTML escaped.

`errors`

A sequence containing the validation errors for this field.

`process_errors`

Errors obtained during input processing. These will be prepended to the list of errors at validation time.

`widget`

The widget used to render the field.

`type`

The type of this field, as a string. This can be used in your templates to do logic based on the type of field:

```
{% for field in form %}
    <tr>
    {% if field.type == "BooleanField" %}
        <td></td>
        <td>{{ field }} {{ field.label }}</td>
    {% else %}
        <td>{{ field.label }}</td>
        <td>{{ field }}</td>
    {% end %}
    </tr>
{% endfor %}
```

flags
An object containing boolean flags set either by the field itself, or by validators on the field. For example, the built-in `InputRequired` validator sets the *required* flag. An unset flag will result in `False`.

```
{% for field in form %}
    <tr>
        <th>{{ field.label }} {% if field.flags.required %}*{% endif %}</th>
        <td>{{ field }}</td>
    </tr>
{% endfor %}
```

meta
The same *meta object* instance as is available as `Form.meta`

filters
The same sequence of filters that was passed as the `filters=` to the field constructor. This is usually a sequence of callables.

2.3 Basic fields

Basic fields generally represent scalar data types with single values, and refer to a single input from the form.

class `wtforms.fields.`**`BooleanField`**(*default field arguments*, *false_values=None*)
Represents an `<input type="checkbox">`.

Parameters **false_values** – If provided, a sequence of strings each of which is an exact match string of what is considered a "false" value. Defaults to the tuple `('false', '')`

class `wtforms.fields.`**`DateField`**(*default field arguments*, *format='%Y-%m-%d'*)
Same as DateTimeField, except stores a *datetime.date*.

class `wtforms.fields.`**`DateTimeField`**(*default field arguments*, *format='%Y-%m-%d %H:%M:%S'*)
A text field which stores a *datetime.datetime* matching a format.

For better date/time fields, see the `dateutil extension`

class `wtforms.fields.`**`DecimalField`**(*default field arguments*, *places=2*, *rounding=None*, *use_locale=False*, *number_format=None*)
A text field which displays and coerces data of the *decimal.Decimal* type.

Parameters

- **places** – How many decimal places to quantize the value to for display on form. If None, does not quantize value.
- **rounding** – How to round the value during quantize, for example *decimal.ROUND_UP*. If unset, uses the rounding value from the current thread's context.

- **use_locale** – If True, use locale-based number formatting. Locale-based number formatting requires the 'babel' package.
- **number_format** – Optional number format for locale. If omitted, use the default decimal format for the locale.

class `wtforms.fields.`**`FileField`**(*default field arguments*)

Can render a file-upload field. Will take any passed filename value, if any is sent by the browser in the post params. This field will NOT actually handle the file upload portion, as wtforms does not deal with individual frameworks' file handling capabilities.

Example usage:

```
class UploadForm(Form):
    image        = FileField(u'Image File', [validators.regexp(u'^[^/\\]\.jpg$')])
    description  = TextAreaField(u'Image Description')

    def validate_image(form, field):
        if field.data:
            field.data = re.sub(r'[^a-z0-9_.-]', '_', field.data)

def upload(request):
    form = UploadForm(request.POST)
    if form.image.data:
        image_data = request.FILES[form.image.name].read()
        open(os.path.join(UPLOAD_PATH, form.image.data), 'w').write(image_data)
```

class `wtforms.fields.`**`FloatField`**(*default field arguments*)

A text field, except all input is coerced to an float. Erroneous input is ignored and will not be accepted as a value.

For the majority of uses, `DecimalField` is preferable to FloatField, except for in cases where an IEEE float is absolutely desired over a decimal value.

class `wtforms.fields.`**`IntegerField`**(*default field arguments*)

A text field, except all input is coerced to an integer. Erroneous input is ignored and will not be accepted as a value.

class `wtforms.fields.`**`RadioField`**(*default field arguments*, *choices*=[], *coerce*=*unicode*)

Like a SelectField, except displays a list of radio buttons.

Iterating the field will produce subfields (each containing a label as well) in order to allow custom rendering of the individual radio fields.

```
{% for subfield in form.radio %}
    <tr>
        <td>{{ subfield }}</td>
        <td>{{ subfield.label }}</td>
    </tr>
{% endfor %}
```

Simply outputting the field without iterating its subfields will result in a `<ul>` list of radio choices.

class `wtforms.fields.`**`SelectField`**(*default field arguments*, *choices*=[], *coerce*=*unicode*, *option_widget*=*None*)

Select fields keep a *choices* property which is a sequence of *(value, label)* pairs. The value portion can be any type in theory, but as form data is sent by the browser as strings, you will need to provide a function which can coerce the string representation back to a comparable object.

Select fields with static choice values:

```
class PastebinEntry(Form):
    language = SelectField(u'Programming Language', choices=[('cpp', 'C++'), ('py', 'Python'), (
```

Note that the *choices* keyword is only evaluated once, so if you want to make a dynamic drop-down list, you'll want to assign the choices list to the field after instantiation. Any inputted choices which are not in the given choices list will cause validation on the field to fail.

Select fields with dynamic choice values:

```
class UserDetails(Form):
    group_id = SelectField(u'Group', coerce=int)

def edit_user(request, id):
    user = User.query.get(id)
    form = UserDetails(request.POST, obj=user)
    form.group_id.choices = [(g.id, g.name) for g in Group.query.order_by('name')]
```

Note we didn't pass a *choices* to the `SelectField` constructor, but rather created the list in the view function. Also, the *coerce* keyword arg to `SelectField` says that we use `int()` to coerce form data. The default coerce is `unicode()`.

Advanced functionality

SelectField and its descendants are iterable, and iterating it will produce a list of fields each representing an option. The rendering of this can be further controlled by specifying *option_widget=*.

class `wtforms.fields.`**`SelectMultipleField`**(*default field arguments*, *choices=[]*, *coerce=unicode*, *option_widget=None*)

No different from a normal select field, except this one can take (and validate) multiple choices. You'll need to specify the HTML *size* attribute to the select field when rendering.

The data on the SelectMultipleField is stored as a list of objects, each of which is checked and coerced from the form input. Any inputted choices which are not in the given choices list will cause validation on the field to fail.

class `wtforms.fields.`**`SubmitField`**(*default field arguments*)

Represents an `<input type="submit">`. This allows checking if a given submit button has been pressed.

class `wtforms.fields.`**`StringField`**(*default field arguments*)

This field is the base for most of the more complicated fields, and represents an `<input type="text">`.

```
{{ form.username(size=30, maxlength=50) }}
```

2.4 Convenience Fields

class `wtforms.fields.`**`HiddenField`**(*default field arguments*)

HiddenField is a convenience for a StringField with a HiddenInput widget.

It will render as an `<input type="hidden">` but otherwise coerce to a string.

HiddenField is useful for providing data from a model or the application to be used on the form handler side for making choices or finding records. Very frequently, CRUD forms will use the hidden field for an object's id.

Hidden fields are like any other field in that they can take validators and values and be accessed on the form object. You should consider validating your hidden fields just as you'd validate an input field, to prevent from malicious people playing with your data.

class `wtforms.fields.`**`PasswordField`**(*default field arguments*)

A StringField, except renders an `<input type="password">`.

Also, whatever value is accepted by this field is not rendered back to the browser like normal fields.

class `wtforms.fields.`**`TextAreaField`**(*default field arguments*)
This field represents an HTML `<textarea>` and can be used to take multi-line input.

2.5 Field Enclosures

Field enclosures allow you to have fields which represent a collection of fields, so that a form can be composed of multiple re-usable components or more complex data structures such as lists and nested objects can be represented.

class `wtforms.fields.`**`FormField`**(*form_class*, *default field arguments*, *separator='-'*)
Encapsulate a form as a field in another form.

Parameters

- **form_class** – A subclass of Form that will be encapsulated.
- **separator** – A string which will be suffixed to this field's name to create the prefix to enclosed fields. The default is fine for most uses.

FormFields are useful for editing child objects or enclosing multiple related forms on a page which are submitted and validated together. While subclassing forms captures most desired behaviours, sometimes for reusability or purpose of combining with *FieldList*, FormField makes sense.

For example, take the example of a contact form which uses a similar set of three fields to represent telephone numbers:

```
class TelephoneForm(Form):
    country_code = IntegerField('Country Code', [validators.required()])
    area_code    = IntegerField('Area Code/Exchange', [validators.required()])
    number       = StringField('Number')

class ContactForm(Form):
    first_name   = StringField()
    last_name    = StringField()
    mobile_phone = FormField(TelephoneForm)
    office_phone = FormField(TelephoneForm)
```

In the example, we reused the TelephoneForm to encapsulate the common telephone entry instead of writing a custom field to handle the 3 sub-fields. The *data* property of the mobile_phone field will return the `data` dict of the enclosed form. Similarly, the *errors* property encapsulate the forms' errors.

class `wtforms.fields.`**`FieldList`**(*unbound_field*, *default field arguments*, *min_entries=0*, *max_entries=None*)
Encapsulate an ordered list of multiple instances of the same field type, keeping data as a list.

```
>>> authors = FieldList(StringField('Name', [validators.required()]))
```

Parameters

- **unbound_field** – A partially-instantiated field definition, just like that would be defined on a form directly.
- **min_entries** – if provided, always have at least this many entries on the field, creating blank ones if the provided input does not specify a sufficient amount.
- **max_entries** – accept no more than this many entries as input, even if more exist in formdata.

Note: Due to a limitation in how HTML sends values, FieldList cannot enclose `BooleanField` or `SubmitField` instances.

append_entry([*data*])
Create a new entry with optional default data.

Entries added in this way will *not* receive formdata however, and can only receive object data.

pop_entry()
Removes the last entry from the list and returns it.

entries
Each entry in a FieldList is actually an instance of the field you passed in. Iterating, checking the length of, and indexing the FieldList works as expected, and proxies to the enclosed entries list.

Do not resize the entries list directly, this will result in undefined behavior. See *append_entry* and *pop_entry* for ways you can manipulate the list.

__iter__()

__len__()

__getitem__(*index*)

`FieldList` is not limited to enclosing simple fields; and can indeed represent a list of enclosed forms by combining FieldList with FormField:

```
class IMForm(Form):
    protocol = SelectField(choices=[('aim', 'AIM'), ('msn', 'MSN')])
    username = StringField()

class ContactForm(Form):
    first_name   = StringField()
    last_name    = StringField()
    im_accounts = FieldList(FormField(IMForm))
```

2.6 Custom Fields

While WTForms provides customization for existing fields using widgets and keyword argument attributes, sometimes it is necessary to design custom fields to handle special data types in your application.

Let's design a field which represents a comma-separated list of tags:

```
class TagListField(Field):
    widget = TextInput()

    def _value(self):
        if self.data:
            return u', '.join(self.data)
        else:
            return u''

    def process_formdata(self, valuelist):
        if valuelist:
            self.data = [x.strip() for x in valuelist[0].split(',')]
        else:
            self.data = []
```

The *_value* method is called by the `TextInput` widget to provide the value that is displayed in the form. Overriding the `process_formdata()` method processes the incoming form data back into a list of tags.

2.6.1 Fields With Custom Constructors

Custom fields can also override the default field constructor if needed to provide additional customization:

```
class BetterTagListField(TagListField):
    def __init__(self, label='', validators=None, remove_duplicates=True, **kwargs):
        super(BetterTagListField, self).__init__(label, validators, **kwargs)
        self.remove_duplicates = remove_duplicates

    def process_formdata(self, valuelist):
        super(BetterTagListField, self).process_formdata(valuelist)
        if self.remove_duplicates:
            self.data = list(self._remove_duplicates(self.data))

    @classmethod
    def _remove_duplicates(cls, seq):
        """Remove duplicates in a case insensitive, but case preserving manner"""
        d = {}
        for item in seq:
            if item.lower() not in d:
                d[item.lower()] = True
                yield item
```

When you override a Field's constructor, to maintain consistent behavior, you should design your constructor so that:

- You take *label='', validators=None* as the first two positional arguments
- Add any additional arguments your field takes as keyword arguments after the label and validators
- Take ***kwargs* to catch any additional keyword arguments.
- Call the Field constructor first, passing the first two positional arguments, and all the remaining keyword args.

2.6.2 Considerations for overriding process()

For the vast majority of fields, it is not necessary to override `Field.process()`. Most of the time, you can achieve what is needed by overriding `process_data` and/or `process_formdata`. However, for special types of fields, such as form enclosures and other special cases of handling multiple values, it may be needed.

If you are going to override `process()`, be careful about how you deal with the `formdata` parameter. For compatibility with the maximum number of frameworks, we suggest you limit yourself to manipulating formdata in the following ways only:

- Testing emptiness: `if formdata`
- Checking for key existence: `key in formdata`
- Iterating all keys: `for key in formdata` (note that some wrappers may return multiple instances of the same key)
- Getting the list of values for a key: `formdata.getlist(key)`.

Most importantly, you should not use dictionary-style access to work with your formdata wrapper, because the behavior of this is highly variant on the wrapper: some return the first item, others return the last, and some may return a list.

2.7 Additional Helper Classes

class `wtforms.fields.`**`Flags`**

Holds a set of boolean flags as attributes.

Accessing a non-existing attribute returns False for its value.

Usage:

```
>>> flags = Flags()
>>> flags.required = True
>>> 'required' in flags
True
>>> 'nonexistent' in flags
False
>>> flags.fake
False
```

class `wtforms.fields.`**`Label`**

On all fields, the *label* property is an instance of this class. Labels can be printed to yield a `<label for="field_id">Label Text</label>` HTML tag enclosure. Similar to fields, you can also call the label with additional html params.

`field_id`

The ID of the field which this label will reference.

`text`

The original label text passed to the field's constructor.

CHAPTER 3

Validators

A validator simply takes an input, verifies it fulfills some criterion, such as a maximum length for a string and returns. Or, if the validation fails, raises a `ValidationError`. This system is very simple and flexible, and allows you to chain any number of validators on fields.

class `wtforms.validators.`**`ValidationError`**(*message=u'‘*, **args*, ***kwargs*)
Raised when a validator fails to validate its input.

class `wtforms.validators.`**`StopValidation`**(*message=u'‘*, **args*, ***kwargs*)
Causes the validation chain to stop.

If StopValidation is raised, no more validators in the validation chain are called. If raised with a message, the message will be added to the errors list.

3.1 Built-in validators

class `wtforms.validators.`**`DataRequired`**(*message=None*)
Checks the field's data is 'truthy' otherwise stops the validation chain.

This validator checks that the `data` attribute on the field is a 'true' value (effectively, it does `if field.data`.) Furthermore, if the data is a string type, a string containing only whitespace characters is considered false.

If the data is empty, also removes prior errors (such as processing errors) from the field.

NOTE this validator used to be called *Required* but the way it behaved (requiring coerced data, not input data) meant it functioned in a way which was not symmetric to the *Optional* validator and furthermore caused confusion with certain fields which coerced data to 'falsey' values like `0`, `Decimal(0)`, `time(0)` etc. Unless a very specific reason exists, we recommend using the `InputRequired` instead.

Parameters **message** – Error message to raise in case of a validation error.

This also sets the `required` `flag` on fields it is used on.

class `wtforms.validators.`**`Email`**(*message=None*)
Validates an email address. Note that this uses a very primitive regular expression and should only be used in instances where you later verify by other means, such as email activation or lookups.

Parameters **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`EqualTo`**(*fieldname*, *message=None*)
Compares the values of two fields.

Parameters

- **fieldname** – The name of the other field to compare to.
- **message** – Error message to raise in case of a validation error. Can be interpolated with *%(other_label)s* and *%(other_name)s* to provide a more helpful error.

This validator can be used to facilitate in one of the most common scenarios, the password change form:

```
class ChangePassword(Form):
    password = PasswordField('New Password', [InputRequired(), EqualTo('confirm', message='Passw
    confirm  = PasswordField('Repeat Password')
```

In the example, we use the InputRequired validator to prevent the EqualTo validator from trying to see if the passwords do not match if there was no passwords specified at all. Because InputRequired stops the validation chain, EqualTo is not run in the case the password field is left empty.

class `wtforms.validators.`**`InputRequired`**(*message=None*)

Validates that input was provided for this field.

Note there is a distinction between this and DataRequired in that InputRequired looks that form-input data was provided, and DataRequired looks at the post-coercion data.

This also sets the `required` `flag` on fields it is used on.

class `wtforms.validators.`**`IPAddress`**(*ipv4=True*, *ipv6=False*, *message=None*)

Validates an IP address.

Parameters

- **ipv4** – If True, accept IPv4 addresses as valid (default True)
- **ipv6** – If True, accept IPv6 addresses as valid (default False)
- **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`Length`**(*min=-1*, *max=-1*, *message=None*)

Validates the length of a string.

Parameters

- **min** – The minimum required length of the string. If not provided, minimum length will not be checked.
- **max** – The maximum length of the string. If not provided, maximum length will not be checked.
- **message** – Error message to raise in case of a validation error. Can be interpolated using *%(min)d* and *%(max)d* if desired. Useful defaults are provided depending on the existence of min and max.

class `wtforms.validators.`**`MacAddress`**(*message=None*)

Validates a MAC address.

Parameters **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`NumberRange`**(*min=None*, *max=None*, *message=None*)

Validates that a number is of a minimum and/or maximum value, inclusive. This will work with any comparable number type, such as floats and decimals, not just integers.

Parameters

- **min** – The minimum required value of the number. If not provided, minimum value will not be checked.
- **max** – The maximum value of the number. If not provided, maximum value will not be checked.

- **message** – Error message to raise in case of a validation error. Can be interpolated using *%(min)s* and *%(max)s* if desired. Useful defaults are provided depending on the existence of min and max.

class `wtforms.validators.`**`Optional`**(*strip_whitespace=True*)

Allows empty input and stops the validation chain from continuing.

If input is empty, also removes prior errors (such as processing errors) from the field.

Parameters **strip_whitespace** – If True (the default) also stop the validation chain on input which consists of only whitespace.

This also sets the `optional` `flag` on fields it is used on.

class `wtforms.validators.`**`Regexp`**(*regex*, *flags=0*, *message=None*)

Validates the field against a user provided regexp.

Parameters

- **regex** – The regular expression string to use. Can also be a compiled regular expression pattern.
- **flags** – The regexp flags to use, for example re.IGNORECASE. Ignored if *regex* is not a string.
- **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`URL`**(*require_tld=True*, *message=None*)

Simple regexp based url validation. Much like the email validator, you probably want to validate the url later by other means if the url must resolve.

Parameters

- **require_tld** – If true, then the domain-name portion of the URL must contain a .tld suffix. Set this to false if you want to allow domains like *localhost*.
- **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`UUID`**(*message=None*)

Validates a UUID.

Parameters **message** – Error message to raise in case of a validation error.

class `wtforms.validators.`**`AnyOf`**(*values*, *message=None*, *values_formatter=None*)

Compares the incoming data to a sequence of valid inputs.

Parameters

- **values** – A sequence of valid inputs.
- **message** – Error message to raise in case of a validation error. *%(values)s* contains the list of values.
- **values_formatter** – Function used to format the list of values in the error message.

class `wtforms.validators.`**`NoneOf`**(*values*, *message=None*, *values_formatter=None*)

Compares the incoming data to a sequence of invalid inputs.

Parameters

- **values** – A sequence of invalid inputs.
- **message** – Error message to raise in case of a validation error. *%(values)s* contains the list of values.
- **values_formatter** – Function used to format the list of values in the error message.

3.2 Custom validators

We will step through the evolution of writing a length-checking validator similar to the built-in `Length` validator, starting from a case-specific one to a generic reusable validator.

Let's start with a simple form with a name field and its validation:

```
class MyForm(Form):
    name = StringField('Name', [InputRequired()])

    def validate_name(form, field):
        if len(field.data) > 50:
            raise ValidationError('Name must be less than 50 characters')
```

Above, we show the use of an *in-line validator* to do validation of a single field. In-line validators are good for validating special cases, but are not easily reusable. If, in the example above, the *name* field were to be split into two fields for first name and surname, you would have to duplicate your work to check two lengths.

So let's start on the process of splitting the validator out for re-use:

```
def my_length_check(form, field):
    if len(field.data) > 50:
        raise ValidationError('Field must be less than 50 characters')

class MyForm(Form):
    name = StringField('Name', [InputRequired(), my_length_check])
```

All we've done here is move the exact same code out of the class and as a function. Since a validator can be any callable which accepts the two positional arguments form and field, this is perfectly fine, but the validator is very special-cased.

Instead, we can turn our validator into a more powerful one by making it a factory which returns a callable:

```
def length(min=-1, max=-1):
    message = 'Must be between %d and %d characters long.' % (min, max)

    def _length(form, field):
        l = field.data and len(field.data) or 0
        if l < min or max != -1 and l > max:
            raise ValidationError(message)

    return _length

class MyForm(Form):
    name = StringField('Name', [InputRequired(), length(max=50)])
```

Now we have a configurable length-checking validator that handles both minimum and maximum lengths. When `length(max=50)` is passed in your validators list, it returns the enclosed *_length* function as a closure, which is used in the field's validation chain.

This is now an acceptable validator, but we recommend that for reusability, you use the pattern of allowing the error message to be customized via passing a `message=` parameter:

```
class Length(object):
    def __init__(self, min=-1, max=-1, message=None):
        self.min = min
        self.max = max
        if not message:
            message = u'Field must be between %i and %i characters long.' % (min, max)
        self.message = message
    def __call__(self, form, field):
        l = field.data and len(field.data) or 0
        if l < self.min or self.max != -1 and l > self.max:
            raise ValidationError(self.message)

length = Length
```

In addition to allowing the error message to be customized, we've now converted the length validator to a class. This wasn't necessary, but we did this to illustrate how one would do so. Because fields will accept any callable as a validator, callable classes are just as applicable. For complex validators, or using inheritance, you may prefer this.

We aliased the `Length` class back to the original `length` name in the above example. This allows you to keep API compatibility as you move your validators from factories to classes, and thus we recommend this for those writing validators they will share.

3.2.1 Setting flags on the field with validators

Sometimes, it's useful to know if a validator is present on a given field, like for use in template code. To do this, validators are allowed to specify flags which will then be available on the `field's flags object`. Some of the built-in validators such as `Required` already do this.

To specify flags on your validator, set the `field_flags` attribute on your validator. When the Field is constructed, the flags with the same name will be set to True on your field. For example, let's imagine a validator that validates that input is valid BBCode. We can set a flag on the field then to signify that the field accepts BBCode:

```
# class implementation
class ValidBBCode(object):
    field_flags = ('accepts_bbcode', )

    pass # validator implementation here

# factory implementation
def valid_bbcode():
    def _valid_bbcode(form, field):
        pass # validator implementation here

    _valid_bbcode.field_flags = ('accepts_bbcode', )
    return _valid_bbcode
```

Then we can check it in our template, so we can then place a note to the user:

```
{{ field(rows=7, cols=70) }}
{% if field.flags.accepts_bbcode %}
    <div class="note">This field accepts BBCode formatting as input.</div>
{% endif %}
```

Some considerations on using flags:

- Flags can only set boolean values, and another validator cannot unset them.
- If multiple fields set the same flag, its value is still True.
- Flags are set from validators only in `Field.__init__()`, so inline validators and extra passed-in validators cannot set them.

CHAPTER 4

Widgets

Widgets are classes whose purpose are to render a field to its usable representation, usually XHTML. When a field is called, the default behaviour is to delegate the rendering to its widget. This abstraction is provided so that widgets can easily be created to customize the rendering of existing fields.

Note All built-in widgets will return upon rendering a "HTML-safe" unicode string subclass that many templating frameworks (Jinja2, Mako, Genshi) will recognize as not needing to be auto-escaped.

4.1 Built-in widgets

class `wtforms.widgets.`**`ListWidget`**`(`*html_tag=u'ul', prefix_label=True*`)`

Renders a list of fields as a *ul* or *ol* list.

This is used for fields which encapsulate many inner fields as subfields. The widget will try to iterate the field to get access to the subfields and call them to render them.

If *prefix_label* is set, the subfield's label is printed before the field, otherwise afterwards. The latter is useful for iterating radios or checkboxes.

class `wtforms.widgets.`**`TableWidget`**`(`*with_table_tag=True*`)`

Renders a list of fields as a set of table rows with th/td pairs.

If *with_table_tag* is True, then an enclosing <table> is placed around the rows.

Hidden fields will not be displayed with a row, instead the field will be pushed into a subsequent table row to ensure XHTML validity. Hidden fields at the end of the field list will appear outside the table.

class `wtforms.widgets.`**`Input`**`(`*input_type=None*`)`

Render a basic `<input>` field.

This is used as the basis for most of the other input fields.

By default, the *_value()* method will be called upon the associated field to provide the `value=` HTML attribute.

class `wtforms.widgets.`**`TextInput`**

Render a single-line text input.

class `wtforms.widgets.`**`PasswordInput`**`(`*hide_value=True*`)`

Render a password input.

For security purposes, this field will not reproduce the value on a form submit by default. To have the value filled in, set *hide_value* to *False*.

class `wtforms.widgets.`**`HiddenInput`**

Render a hidden input.

class `wtforms.widgets.`**`CheckboxInput`**

Render a checkbox.

The `checked` HTML attribute is set if the field's data is a non-false value.

class `wtforms.widgets.`**`FileInput`**

Renders a file input chooser field.

class `wtforms.widgets.`**`SubmitInput`**

Renders a submit button.

The field's label is used as the text of the submit button instead of the data on the field.

class `wtforms.widgets.`**`TextArea`**

Renders a multi-line text area.

rows and *cols* ought to be passed as keyword args when rendering.

class `wtforms.widgets.`**`Select`**(*multiple=False*)

Renders a select field.

If *multiple* is True, then the *size* property should be specified on rendering to make the field useful.

The field must provide an *iter_choices()* method which the widget will call on rendering; this method must yield tuples of *(value, label, selected)*.

4.2 Widget-Building Utilities

These utilities are used in WTForms widgets to help render HTML and also in order to work along with HTML templating frameworks. They can be imported for use in building custom widgets as well.

`wtforms.widgets.`**`html_params`**(***kwargs*)

Generate HTML attribute syntax from inputted keyword arguments.

The output value is sorted by the passed keys, to provide consistent output each time this function is called with the same parameters. Because of the frequent use of the normally reserved keywords *class* and *for*, suffixing these with an underscore will allow them to be used.

In addition, the values `True` and `False` are special:

- `attr=True` generates the HTML compact output of a boolean attribute, e.g. `checked=True` will generate simply `checked`
- `attr=`False` will be ignored and generate no output.

```
>>> html_params(name='text1', id='f', class_='text')
'class="text" id="f" name="text1"'
>>> html_params(checked=True, readonly=False, name="text1", abc="hello")
'abc="hello" checked name="text1"'
```

class `wtforms.widgets.`**`HTMLString`**

This is an "HTML safe string" class that is returned by WTForms widgets.

For the most part, HTMLString acts like a normal unicode string, except in that it has a *__html__* method. This method is invoked by a compatible auto-escaping HTML framework to get the HTML-safe version of a string.

Usage:

```
HTMLString('<input type="text" value="hello">')
```

__html__()

Give an HTML-safe string.

This method actually returns itself, because it's assumed that whatever you give to HTMLString is a string with any unsafe values already escaped. This lets auto-escaping template frameworks know that this string is safe for HTML rendering.

4.3 Custom widgets

Widgets, much like validators, provide a simple callable contract. Widgets can take customization arguments through a constructor if needed as well. When the field is called or printed, it will call the widget with itself as the first argument and then any additional arguments passed to its caller as keywords. Passing the field is done so that one instance of a widget might be used across many field instances.

Let's look at a widget which renders a text field with an additional class if there are errors:

```
class MyTextInput(TextInput):
    def __init__(self, error_class=u'has_errors'):
        super(MyTextInput, self).__init__()
        self.error_class = error_class

    def __call__(self, field, **kwargs):
        if field.errors:
            c = kwargs.pop('class', '') or kwargs.pop('class_', '')
            kwargs['class'] = u'%s %s' % (self.error_class, c)
        return super(MyTextInput, self).__call__(field, **kwargs)
```

In the above example, we extended the behavior of the existing `TextInput` widget to append a CSS class as needed. However, widgets need not extend from an existing widget, and indeed don't even have to be a class. For example, here is a widget that renders a `SelectMultipleField` as a collection of checkboxes:

```
def select_multi_checkbox(field, ul_class='', **kwargs):
    kwargs.setdefault('type', 'checkbox')
    field_id = kwargs.pop('id', field.id)
    html = [u'<ul %s>' % html_params(id=field_id, class_=ul_class)]
    for value, label, checked in field.iter_choices():
        choice_id = u'%s-%s' % (field_id, value)
        options = dict(kwargs, name=field.name, value=value, id=choice_id)
        if checked:
            options['checked'] = 'checked'
        html.append(u'<li><input %s /> ' % html_params(**options))
        html.append(u'<label for="%s">%s</label></li>' % (field_id, label))
    html.append(u'</ul>')
    return u''.join(html)
```

class Meta

the *class Meta* paradigm allows WTForms features to be customized, and even new behaviors to be introduced. It also supplies a place where configuration for any complementary modules can be done.

Typical usage looks something like:

```
class MyForm(Form):
    class Meta:
        csrf = True
        locales = ('en_US', 'en')

    name = TextField(...)
    # and so on...
```

For the majority of users, using a class Meta is mostly going to be done for customizing options used by the default behaviors, however for completeness the entire API of the *Meta* interface is shown here.

class `wtforms.meta.`**`DefaultMeta`**

This is the default Meta class which defines all the default values and therefore also the 'API' of the class Meta interface.

Configuration

`csrf` **= False**

Setting `csrf` to *True* will enable CSRF for the form. The value can also be overridden per-instance via instantiation-time customization (for example, if csrf needs to be turned off only in a special case)

```
form = MyForm(request.form, meta={'csrf': False})
```

`csrf_class` **= None**

If set, this is a class which is used to implement CSRF protection. Read the *CSRF Documentation* to get more information on how to use.

`csrf_field_name` **= 'csrf_token'**

The name of the automatically added CSRF token field.

`locales` **= False**

Setting to a sequence of strings specifies the priority order of locales to try to find translations for built-in messages of WTForms. If the value *False*, then strings are not translated (the translations provider is a dummy provider)

`cache_translations` **= True**

If *True* (the default) then cache translation objects. The default cache is done at class-level so it's shared with all class Meta.

Advanced Customization

Usually, you do not need to override these methods, as they provide core behaviors of WTForms.

build_csrf(*form*)
Build a CSRF implementation. This is called once per form instance.

The default implementation builds the class referenced to by `csrf_class` with zero arguments. If *csrf_class* is `None`, will instead use the default implementation `wtforms.csrf.session.SessionCSRF`.

Parameters **form** – The form.

Returns A CSRF implementation.

get_translations(*form*)
Override in subclasses to provide alternate translations factory. See the i18n documentation for more.

Parameters **form** – The form.

Returns An object that provides gettext() and ngettext() methods.

bind_field(*form*, *unbound_field*, *options*)
bind_field allows potential customization of how fields are bound.

The default implementation simply passes the options to `UnboundField.bind()`.

Parameters

- **form** – The form.
- **unbound_field** – The unbound field.
- **options** – A dictionary of options which are typically passed to the field.

Returns A bound field

wrap_formdata(*form*, *formdata*)
wrap_formdata allows doing custom wrappers of WTForms formdata.

The default implementation detects webob-style multidicts and wraps them, otherwise passes formdata back un-changed.

Parameters

- **form** – The form.
- **formdata** – Form data.

Returns A form-input wrapper compatible with WTForms.

render_field(*field*, *render_kw*)
render_field allows customization of how widget rendering is done.

The default implementation calls `field.widget(field, **render_kw)`

CHAPTER 6

CSRF Protection

The CSRF package includes tools that help you implement checking against cross-site request forgery ("csrf"). Due to the large number of variations on approaches people take to CSRF (and the fact that many make compromises) the base implementation allows you to plug in a number of CSRF validation approaches.

CSRF implementations are made by subclassing `CSRF`. For utility, we have provided one possible CSRF implementation in the package that can be used with many frameworks for session-based hash secure keying, `SessionCSRF`.

6.1 Using CSRF

CSRF in WTForms 2.0 is now driven through a number of variables on `class Meta`. After choosing a CSRF implementation, import it and configure it on the class Meta of a subclass of Form like such:

```
from somemodule import SomeCSRF

class MyBaseForm(Form):
    class Meta:
        csrf = True  # Enable CSRF
        csrf_class = SomeCSRF  # Set the CSRF implementation
        csrf_secret = b'foobar'  # Some implementations need a secret key.
        # Any other CSRF settings here.
```

And once you've got this set up, you can define your forms as a subclass of *MyBaseForm*:

```
class UserForm(MyBaseForm):
    name = TextField()
    age = IntegerField()

def view():
    form = UserForm(request.POST)
    if request.POST and form.validate():
        pass # Form is valid and CSRF succeeded

    return render('user.html', form=form)
```

There is a special field inside the CSRF form (called `csrf_token` by default) which you need to make sure you render in your template:

```
<form action="/user" method="POST">
{{ form.csrf_token }}
{% if form.csrf_token.errors %}
    <div class="warning">You have submitted an invalid CSRF token</div>
```

```
{% endif %}
<div>{{ form.name }} {{ form.name.label }}</div>
<div>{{ form.age }}{{ form.age.label }}</div>
```

Remember, with the *class Meta* you can always override variables in a sub-class or at the constructor for special-cases:

```
class SearchForm(MyBaseForm):
    """
    We expect search queries to come externally, thus we don't want CSRF
    even though it's set up on the base form.
    """
    class Meta:
        # This overrides the value from the base form.
        csrf = False
```

6.2 How WTForms CSRF works

Most CSRF implementations hinge around creating a special token, which is put in a hidden field on the form named *csrf_token*, which must be rendered in your template to be passed from the browser back to your view. There are many different methods of generating this token, but they are usually the result of a cryptographic hash function against some data which would be hard to forge.

class `wtforms.csrf.core.`**`CSRFTokenField`**`(`**args*, ***kw*`)`

A subclass of HiddenField designed for sending the CSRF token that is used for most CSRF protection schemes.

Notably different from a normal field, this field always renders the current token regardless of the submitted value, and also will not be populated over to object data via populate_obj

`__init__``(`**args*, ***kw*`)`

`current_token` **= None**

`_value``()`

We want to always return the current token on render, regardless of whether a good or bad token was passed.

`populate_obj``(`**args*`)`

Don't populate objects with the CSRF token

`pre_validate``(`*form*`)`

Handle validation of this token field.

`process``(`**args*`)`

class `wtforms.csrf.core.`**`CSRF`**

`setup_form``(`*form*`)`

Receive the form we're attached to and set up fields.

The default implementation creates a single field of type `field_class` with name taken from the `csrf_field_name` of the class meta.

Parameters **form** – The form instance we're attaching to.

Returns A sequence of *(field_name, unbound_field)* 2-tuples which are unbound fields to be added to the form.

generate_csrf_token (*csrf_token_field*)
Implementations must override this to provide a method with which one can get a CSRF token for this form.

A CSRF token is usually a string that is generated deterministically based on some sort of user data, though it can be anything which you can validate on a subsequent request.

Parameters **csrf_token_field** – The field which is being used for CSRF.

Returns A generated CSRF string.

validate_csrf_token (*form*, *field*)
Override this method to provide custom CSRF validation logic.

The default CSRF validation logic simply checks if the recently generated token equals the one we received as formdata.

Parameters

- **form** – The form which has this CSRF token.
- **field** – The CSRF token field.

field_class = <class 'wtforms.csrf.core.CSRFTokenField'>
The class of the token field we're going to construct. Can be overridden in subclasses if need be.

6.3 Creating your own CSRF implementation

Here we will sketch out a simple theoretical CSRF implementation which generates a hash token based on the user's IP.

Note This is a simplistic example meant to illustrate creating a CSRF implementation. This isn't recommended to be used in production because the token is deterministic and non-changing per-IP, which means this isn't the most secure implementation of CSRF.

First, let's create our CSRF class:

```
from wtforms.csrf.core import CSRF
from hashlib import md5

SECRET_KEY = '1234567890'

class IPAddressCSRF(CSRF):
    """
    Generate a CSRF token based on the user's IP. I am probably not very
    secure, so don't use me.
    """
    def setup_form(self, form):
        self.csrf_context = form.meta.csrf_context
        return super(IPAddressCSRF, self).setup_form(form)

    def generate_csrf_token(self, csrf_token):
        token = md5(SECRET_KEY + self.csrf_context).hexdigest()
        return token

    def validate_csrf_token(self, form, field):
        if field.data != field.current_token:
            raise ValueError('Invalid CSRF')
```

Now that we have this taken care of, let's write a simple form and view which would implement this:

```
class RegistrationForm(Form):
    class Meta:
        csrf = True
        csrf_class = IPAddressCSRF

    name = StringField('Your Name')
    email = StringField('Email', [validators.email()])

def register(request):
    form = RegistrationForm(
        request.POST,
        meta={'csrf_context': request.ip}
    )

    if request.method == 'POST' and form.validate():
        pass # We're all good, create a user or whatever it is you do
    elif form.csrf_token.errors:
        pass # If we're here we suspect the user of cross-site request forgery
    else:
        pass # Any other errors

    return render('register.html', form=form)
```

And finally, a simple template:

```
<form action="register" method="POST">
    {{ form.csrf_token }}
    <p>{{ form.name.label }}: {{ form.name }}</p>
    <p>{{ form.email.label }}: {{ form.email }}</p>
    <input type="submit" value="Register">
</form>
```

Please note that implementing CSRF detection is not fool-proof, and even with the best CSRF protection implementation, it's possible for requests to be forged by expert attackers. However, a good CSRF protection would make it infeasible for someone from an external site to hijack a form submission from another user and perform actions as them without additional a priori knowledge.

In addition, it's important to understand that very often, the more strict the CSRF protection, the higher the chance of false positives occurring (ie, legitimate users getting blocked by your CSRF protection) and choosing a CSRF implementation is actually a matter of compromise. We will attempt to provide a handful of usable reference algorithms built in to this library in the future, to allow that choice to be easy.

Some tips on criteria people often examine when evaluating CSRF implementations:

- **Reproducability** If a token is based on attributes about the user, it gains the advantage that one does not need secondary storage in which to store the value between requests. However, if the same attributes can be reproduced by an attacker, then the attacker can potentially forge this information.
- **Reusability**. It might be desired to make a completely different token every use, and disallow users from re-using past tokens. This is an extremely powerful protection, but can have consequences on if the user uses the back button (or in some cases runs forms simultaneously in multiple browser tabs) and submits an old token, or otherwise. A possible compromise is to allow reusability in a time window (more on that later).
- **Time Ranges** Many CSRF approaches use time-based expiry to make sure that a token cannot be (re)used beyond a certain point. Care must be taken in choosing the time criteria for this to not lock out legitimate users. For example, if a user might walk away while filling out a long-ish form, or to go look for their credit card, the time for expiry should take that into consideration to provide a balance between security and limiting user inconvenience.

- **Requirements** Some CSRF-prevention methods require the use of browser cookies, and some even require client-side scripting support. The webmaster implementing the CSRF needs to consider that such requirements (though effective) may lock certain legitimate users out, and make this determination whether it is a good idea to use. For example, for a site already using cookies for login, adding another for CSRF isn't as big of a deal, but for other sites it may not be feasible.

6.4 Session-based CSRF implementation

A provided CSRF implementation which puts CSRF data in a session.

This can be used fairly comfortably with many *request.session* type objects, including the Werkzeug/Flask session store, Django sessions, and potentially other similar objects which use a dict-like API for storing session keys.

The basic concept is a randomly generated value is stored in the user's session, and an hmac-sha1 of it (along with an optional expiration time, for extra security) is used as the value of the csrf_token. If this token validates with the hmac of the random value + expiration time, and the expiration time is not passed, the CSRF validation will pass.

class `wtforms.csrf.session.`**`SessionCSRF`**

Meta Values

- `csrf_secret` A byte string which is the master key by which we encode all values. Set to a sufficiently long string of characters that is difficult to guess or bruteforce (recommended at least 16 characters) for example the output of `os.urandom(16)`.
- `csrf_time_limit` if *None*, tokens last forever (not recommended.) Otherwise, set to a `datetime.timedelta` that will define how long CSRF tokens are valid for. Defaults to 30 minutes.
- `csrf_context` This should be a `request.session`-style object. Usually given in the Form constructor.

6.4.1 Example

```
from wtforms.csrf.session import SessionCSRF
from datetime import timedelta

class MyBaseForm(Form):
    class Meta:
        csrf = True
        csrf_class = SessionCSRF
        csrf_secret = b'EPj00jpfj8Gx1SjnyLxwBBSQfnQ9DJYe0Ym'
        csrf_time_limit = timedelta(minutes=20)

class Registration(MyBaseForm):
    name = StringField()

def view(request):
    form = Registration(request.POST, meta={'csrf_context': request.session})
    # rest of view here
```

Note that request.session is passed as the `csrf_context` override to the meta info, this is so that the CSRF token can be stored in your session for comparison on a later request.

6.4.2 Example Integration

WTForms primitives are designed to work with a large variety of frameworks, and as such sometimes things seem like they are more work to use, but with some smart integration, you can actually clean up your code substantially.

For example, if you were going to integrate with Flask, and wanted to use the SessionCSRF implementation, here's one way to get the CSRF context to be available without passing it all the time:

```
from flask import request
from wtforms.csrf.session import SessionCSRF

class MyBaseForm(Form):
    class Meta:
        csrf = True
        csrf_class = SessonCSRF
        csrf_secret = app.config['CSRF_SECRET_KEY']

        @property
        def csrf_context(self):
            return request.session
```

Now with any subclasses of MyBaseForm, you don't need to pass in the csrf context, and on top of that, we grab the secret key out of your normal app configuration.

CHAPTER 7

Extensions

WTForms ships with a number of extensions that make it easier to work with other frameworks and libraries, such as Django.

7.1 Appengine

WTForms includes support for AppEngine fields as well as auto-form generation from models.

Deprecated since version 2.0: `wtforms.ext.appengine` is now deprecated, and will be removed in WTForms 3.0. For an appengine module that will continue to be updated, see WTForms-Appengine

7.1.1 Model Forms

See the module docstring for examples on how to use `model_form()`.

`wtforms.ext.appengine.db.`**`model_form`**(*model*, *base_class=Form*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*)

Creates and returns a dynamic `wtforms.Form` class for a given `db.Model` class. The form class can be used as it is or serve as a base for extended form classes, which can then mix non-model related fields, subforms with other model forms, among other possibilities.

Parameters

- **model** – The `db.Model` class to generate a form for.
- **base_class** – Base form class to extend from. Must be a `wtforms.Form` subclass.
- **only** – An optional iterable with the property names that should be included in the form. Only these properties will have fields.
- **exclude** – An optional iterable with the property names that should be excluded from the form. All other properties will have fields.
- **field_args** – An optional dictionary of field names mapping to keyword arguments used to construct each field object.
- **converter** – A converter to generate the fields based on the model properties. If not set, `ModelConverter` is used.

7.1.2 Datastore-backed Fields

class `wtforms.ext.appengine.fields.`**`ReferencePropertyField`**(*default field arguments*, *reference_class=None*, *get_label=None*, *allow_blank=False*, *blank_text=''*)

A field for `db.ReferenceProperty`. The list items are rendered in a select.

Parameters

- **reference_class** – A db.Model class which will be used to generate the default query to make the list of items. If this is not specified, The *query* property must be overridden before validation.
- **get_label** – If a string, use this attribute on the model class as the label associated with each option. 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.
- **allow_blank** – If set to true, a blank choice will be added to the top of the list to allow *None* to be chosen.
- **blank_text** – Use this to override the default blank option's label.

class `wtforms.ext.appengine.fields.`**`StringListPropertyField`**(*default field arguments*)

A field for `db.StringListProperty`. The list items are rendered in a textarea.

class `wtforms.ext.appengine.fields.`**`GeoPtPropertyField`**(*default field arguments*)

7.1.3 NDB

WTForms includes support for NDB models and can support mapping the relationship fields as well as generating forms from models.

class `wtforms.ext.appengine.fields.`**`KeyPropertyField`**(*default field arguments*, *reference_class=None*, *get_label=None*, *allow_blank=False*, *blank_text=''*)

A field for `ndb.KeyProperty`. The list items are rendered in a select.

Parameters

- **reference_class** – A db.Model class which will be used to generate the default query to make the list of items. If this is not specified, The *query* property must be overridden before validation.
- **get_label** – If a string, use this attribute on the model class as the label associated with each option. 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.
- **allow_blank** – If set to true, a blank choice will be added to the top of the list to allow *None* to be chosen.
- **blank_text** – Use this to override the default blank option's label.

`wtforms.ext.appengine.ndb.`**`model_form`**(*model*, *base_class=Form*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*)

Creates and returns a dynamic `wtforms.Form` class for a given `ndb.Model` class. The form class can be used as it is or serve as a base for extended form classes, which can then mix non-model related fields, subforms with other model forms, among other possibilities.

Parameters

- **model** – The `ndb.Model` class to generate a form for.
- **base_class** – Base form class to extend from. Must be a `wtforms.Form` subclass.
- **only** – An optional iterable with the property names that should be included in the form. Only these properties will have fields.
- **exclude** – An optional iterable with the property names that should be excluded from the form. All other properties will have fields.
- **field_args** – An optional dictionary of field names mapping to keyword arguments used to construct each field object.
- **converter** – A converter to generate the fields based on the model properties. If not set, `ModelConverter` is used.

7.2 Dateutil

For better date-time parsing using the python-dateutil package, `wtforms.ext.dateutil` provides a set of fields to use to accept a wider range of date input.

class `wtforms.ext.dateutil.fields.`**`DateTimeField`**(*default field arguments*, *parse_kwargs=None*, *display_format='%Y-%m-%d %H:%M'*)

DateTimeField represented by a text input, accepts all input text formats that *dateutil.parser.parse* will.

Parameters

- **parse_kwargs** – A dictionary of keyword args to pass to the dateutil parse() function. See dateutil docs for available keywords.
- **display_format** – A format string to pass to strftime() to format dates for display.

class `wtforms.ext.dateutil.fields.`**`DateField`**(*default field arguments*, *parse_kwargs=None*, *display_format='%Y-%m-%d'*)

Same as the DateTimeField, but stores only the date portion.

7.3 Django

This extension provides templatetags to make it easier to work with Django templates and WTForms' html attribute rendering. It also provides a generator for automatically creating forms based on Django ORM models.

Deprecated since version 2.0: `wtforms.ext.django` is now deprecated, and will be removed in WTForms 3.0. For Django support that will continue to be updated, see WTForms-Django

7.3.1 Templatetags

Django templates does not allow arbitrarily calling functions with parameters, making it impossible to use the html attribute rendering feature of WTForms. To alleviate this, we provide a templatetag.

Adding `wtforms.ext.django` to your INSTALLED_APPS will make the wtforms template library available to your application. With this you can pass extra attributes to form fields similar to the usage in jinja:

```
{% load wtforms %}

{% form_field form.username class="big_text" onclick="do_something()" %}
```

Note By default, using the *{{ form.field }}* syntax in django models will be auto-escaped. To avoid this happening, use Django's *{% autoescape off %}* block tag or use WTForms' *form_field* template tag.

7.3.2 Model forms

`wtforms.ext.django.orm.`**`model_form`**(*model*, *base_class=Form*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*)

Create a wtforms Form for a given Django model class:

```
from wtforms.ext.django.orm import model_form
from myproject.myapp.models import User
UserForm = model_form(User)
```

Parameters

- **model** – A Django ORM model class
- **base_class** – Base form class to extend from. Must be a `wtforms.Form` subclass.
- **only** – An optional iterable with the property names that should be included in the form. Only these properties will have fields.
- **exclude** – An optional iterable with the property names that should be excluded from the form. All other properties will have fields.
- **field_args** – An optional dictionary of field names mapping to keyword arguments used to construct each field object.
- **converter** – A converter to generate the fields based on the model properties. If not set, `ModelConverter` is used.

`model_form()` attempts to glean as much metadata as possible from inspecting the model's fields, and will even attempt to guess at what validation might be wanted based on the field type. For example, converting an *EmailField* will result in a `StringField` with the `Email` validator on it. if the *blank* property is set on a model field, the resulting form field will have the `Optional` validator set.

Just like any other Form, forms created by model_form can be extended via inheritance:

```
UserFormBase = model_form(User)

class UserForm(UserFormBase):
    new_pass     = PasswordField('', [validators.optional(), validators.equal_to('confirm_pass')
    confirm_pass = PasswordField()
```

When combined with `form iteration`, model_form is a handy way to generate dynamic CRUD forms which update with added fields to the model. One must be careful though, as it's possible the generated form fields won't be as strict with validation as a hand-written form might be.

7.3.3 ORM-backed fields

While linking data to most fields is fairly easy, making drop-down select lists using django ORM data can be quite repetitive. To this end, we have added some helpful tools to use the django ORM along with wtforms.

class `wtforms.ext.django.fields.`**`QuerySetSelectField`**(*default field args*, *queryset=None*, *get_label=None*, *allow_blank=False*, *blank_text=u''*)

Given a QuerySet either at initialization or inside a view, will display a select drop-down field of choices. The *data* property actually will store/keep an ORM model instance, not the ID. Submitting a choice which is not in the queryset will result in a validation error.

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 the blank choice can be set by specifying the *blank_text* parameter.

```
class ArticleEdit(Form):
    title    = StringField()
    column   = QuerySetSelectField(get_label='title', allow_blank=True)
    category = QuerySetSelectField(queryset=Category.objects.all())

def edit_article(request, id):
    article = Article.objects.get(pk=id)
    form = ArticleEdit(obj=article)
    form.column.queryset = Column.objects.filter(author=request.user)
```

As shown in the above example, the queryset can be set dynamically in the view if needed instead of at form construction time, allowing the select field to consist of choices only relevant to the user.

class `wtforms.ext.django.fields.`**`ModelSelectField`**(*default field args*, *model=None*, *get_label=''*, *allow_blank=False*, *blank_text=u''*)

Like a QuerySetSelectField, except takes a model class instead of a queryset and lists everything in it.

7.4 SQLAlchemy

This extension provides SelectField integration with SQLAlchemy ORM models, similar to those in the Django extension.

7.4.1 ORM-backed fields

These fields are provided to make it easier to use data from ORM objects in your forms.

```
def enabled_categories():
    return Category.query.filter_by(enabled=True)

class BlogPostEdit(Form):
    title    = StringField()
    blog     = QuerySelectField(get_label='title')
    category = QuerySelectField(query_factory=enabled_categories, allow_blank=True)

def edit_blog_post(request, id):
    post = Post.query.get(id)
    form = BlogPostEdit(obj=post)
    # Since we didn't provide a query_factory for the 'blog' field, we need
    # to set a dynamic one in the view.
    form.blog.query = Blog.query.filter(Blog.author == request.user).order_by(Blog.name)
```

class `wtforms.ext.sqlalchemy.fields.`**`QuerySelectField`**(*default field args*, *query_factory=None*, *get_pk=None*, *get_label=None*, *allow_blank=False*, *blank_text=u''*)

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.ext.sqlalchemy.fields.`**`QuerySelectMultipleField`**(*default field args*, *query_factory=None*, *get_pk=None*, *get_label=None*)

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.

7.4.2 Model forms

It is possible to generate forms from SQLAlchemy models similarly to how it can be done for Django ORM models.

`wtforms.ext.sqlalchemy.orm.`**`model_form`**(*model*, *db_session=None*, *base_class=<class 'wtforms.form.Form'>*, *only=None*, *exclude=None*, *field_args=None*, *converter=None*, *exclude_pk=True*, *exclude_fk=True*, *type_name=None*)

Create a wtforms Form for a given SQLAlchemy model class:

```
from wtalchemy.orm import model_form
from myapp.models import User
UserForm = model_form(User)
```

Parameters

- **model** – A SQLAlchemy mapped model class.
- **db_session** – An optional SQLAlchemy Session.
- **base_class** – Base form class to extend from. Must be a `wtforms.Form` subclass.

- **only** – An optional iterable with the property names that should be included in the form. Only these properties will have fields.
- **exclude** – An optional iterable with the property names that should be excluded from the form. All other properties will have fields.
- **field_args** – An optional dictionary of field names mapping to keyword arguments used to construct each field object.
- **converter** – A converter to generate the fields based on the model properties. If not set, `ModelConverter` is used.
- **exclude_pk** – An optional boolean to force primary key exclusion.
- **exclude_fk** – An optional boolean to force foreign keys exclusion.
- **type_name** – An optional string to set returned type name.

7.5 CSRF

Deprecated since version 2.0: wtforms.ext.csrf is deprecated, as CSRF is now provided as a part of the core of WTForms. This module exists as-is for backwards compatibility and will be removed in WTForms 3.0. See the *CSRF Docs*.

The CSRF package includes tools that help you implement checking against cross-site request forgery ("csrf"). Due to the large number of variations on approaches people take to CSRF (and the fact that many make compromises) the base implementation allows you to plug in a number of CSRF validation approaches.

CSRF implementations are made by subclassing `SecureForm`. For utility, we have provided one possible CSRF implementation in the package that can be used with many frameworks for session-based hash secure keying, `SessionSecureForm`.

All CSRF implementations hinge around creating a special token, which is put in a hidden field on the form named 'csrf_token', which must be rendered in your template to be passed from the browser back to your view. There are many different methods of generating this token, but they are usually the result of a cryptographic hash function against some data which would be hard to forge.

class `wtforms.ext.csrf.form.`**`SecureForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form that enables CSRF processing via subclassing hooks.

`generate_csrf_token`(*csrf_context*)

Implementations must override this to provide a method with which one can get a CSRF token for this form.

A CSRF token should be a string which can be generated deterministically so that on the form POST, the generated string is (usually) the same assuming the user is using the site normally.

Parameters **csrf_context** – A transparent object which can be used as contextual info for generating the token.

`validate_csrf_token`(*field*)

Override this method to provide custom CSRF validation logic.

The default CSRF validation logic simply checks if the recently generated token equals the one we received as formdata.

7.5.1 Session-based CSRF implementation

A provided CSRF implementation which puts CSRF data in a session.

This can be used fairly comfortably with many *request.session* type objects, including the Werkzeug/Flask session store, Django sessions, and potentially other similar objects which use a dict-like API for storing session keys.

The basic concept is a randomly generated value is stored in the user's session, and an hmac-sha1 of it (along with an optional expiration time, for extra security) is used as the value of the csrf_token. If this token validates with the hmac of the random value + expiration time, and the expiration time is not passed, the CSRF validation will pass.

Usage

First, create a SessionSecureForm subclass that you can use as your base class for any forms you want CSRF support for:

```
from wtforms.ext.csrf.session import SessionSecureForm

class MyBaseForm(SessionSecureForm):
    SECRET_KEY = 'EPj00jpfj8Gx1SjnyLxwBBSQfnQ9DJYe0Ym'
    TIME_LIMIT = timedelta(minutes=20)
```

Now incorporate it into any form/view by further subclassing:

```
class Registration(MyBaseForm):
    name = StringField()

def view(request):
    form = Registration(request.POST, csrf_context=request.session)
    # rest of view here
```

Note that request.session is passed as the `csrf_context=` parameter, this is so that the CSRF token can be stored in your session for comparison on a later request.

class `wtforms.ext.csrf.session.`**`SessionSecureForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A provided CSRF implementation which puts CSRF data in a session. Must be subclassed to be used.

Class Attributes

`SECRET_KEY`
Must be set by subclasses to a random byte string that will be used to generate HMAC digests.

`TIME_LIMIT`
If None, CSRF tokens never expire. If set to a `datetime.timedelta`, this is how long til a generated token expires. Defaults to `timedelta(minutes=30)`

7.6 I18n

Deprecated since version 2.0: I18n extension is now deprecated in favor of using the built-in locales support on the form.

class `wtforms.ext.i18n.form.`**`Form`**(**args*, ***kwargs*)

Base form for a simple localized WTForms form.

NOTE this class is now un-necessary as the i18n features have been moved into the core of WTForms, and will be removed in WTForms 3.0.

This will use the stdlib gettext library to retrieve an appropriate translations object for the language, by default using the locale information from the environment.

If the LANGUAGES class variable is overridden and set to a sequence of strings, this will be a list of languages by priority to use instead, e.g:

```
LANGUAGES = ['en_GB', 'en']
```

One can also provide the languages by passing *LANGUAGES=* to the constructor of the form.

Translations objects are cached to prevent having to get a new one for the same languages every instantiation.

Additional Help

CHAPTER 8

FAQ

This contains the most commonly asked questions about WTForms. The most current version of this document can always be found on the WTForms Website.

8.1 Does WTForms work with [library here]?

The answer is most likely **yes**. WTForms tries to provide as usable an API as possible. We've listed here some of the known libraries to work with WTForms, but if it's not listed, it doesn't mean it won't work.

- **Request/Form Input**
  - Django
  - Webob (Includes Pylons, Google App Engine, Turbogears)
  - Werkzeug (Includes Flask, Tipfy)
  - any other cgi.FieldStorage-type multidict
- **Templating Engines**
  - Jinja2
  - Mako
  - Django Templates (To get the full power of WTForms in your templates, you will need to use the Django `extension`.)
  - Genshi
- **Database Objects**
  - Pretty much any ORM or object-DB should work, as long as data objects allow attribute access to their members.

    Special support is there for SQLAlchemy, Google App Engine, and Django collections via `extensions`.

8.2 Does WTForms support unicode?

Simple answer: Yes.

Longer answer: WTForms uses unicode strings throughout the source code, and assumes that form input has already been coerced to unicode by your framework (Most frameworks already do this.) WTForms fields render to unicode strings by default, and therefore as long as your templating engine can work with that, you should have no unicode issues.

8.3 What versions of Python are supported?

WTForms supports Python 2.6-2.7 and 3.2+ with a single codebase. Without 2to3 tool.

8.4 How can I contribute to WTForms?

WTForms is not that scary. Really. We try to keep it as succint and readable as possible. If you feel like you have something to contribute to WTForms, let us know on the mailing list. For bugs and feature requests, you can file a ticket on the project page.

8.5 How do I mark in a template when a field is required?

Some validators (notably Required and Optional) set flags on the fields' `flags` object. To use this in a template, you can do something like:

```
{% for field in form %}
    {{ field }}
    {% if field.flags.required %}*{% endif %}{{ field.label }}
{% endfor %}
```

8.6 Does WTForms handle file uploads?

Currently, it does not. This is because WTForms strives to be framework-agnostic, and every web framework handles file uploads somewhat differently. WTForms has a `FileField` which will let you render a file input widget, but the rest is up to you. An example use in a django-ish framework:

```
class MyForm(Form):
    image = FileField()

def my_view(request):
    form = MyForm(request.POST)
    file_wrapper = request.FILES[form.image.name]
    # Do things with your file wrapper now
```

Using `form.image.name` is an easy way to know what input name was generated for your file input, even if the form is prefixed.

8.7 Why does blank input not go back to the default value?

A key design decision of WTForms was that form data -always- takes precedence when there's a form submission. That is, if a field exists on a form, and a form was posted, but that field's value was missing, it will not revert to a default, but instead store an empty value (and in some cases cause a validation error.)

This is for a number of reasons:

1. Security. If a form reverted to defaults on missing data, then an evil user could potentially cause problems by submitting a hand-coded form with key missing fields.
2. Bug-finding. If you omitted a field in your template, it might fall through to the default and you'd possibly miss it.
3. Consistency.

See the following mailing list posts for more discussion on the topic:

- http://groups.google.com/group/wtforms/browse_frm/thread/6755a45a13878e9
- http://groups.google.com/group/wtforms/msg/fa409c8c89b6f62d

8.8 How do I... [convoluted combination of libraries]

You'll probably want to check out our *Solving Specific Problems* doc.

Solving Specific Problems

What follows is a collection of recipes that will help you tackle specific challenges that may crop up when using WTForms along with various other python frameworks.

9.1 Prelude: Poke it with a Stick!

The aim of WTForms is not to do it all, but rather to stick to the basics, while being compatible with as many frameworks as possible. We attempt to place useful things in the API so that developers can get what they want out of it, if the default behaviour is not desired.

For example, many fields in WTForms are iterable to allow you to access enclosed fields inside them, providing you another way to customize their rendering. Many attributes on the fields are readily available for you to use in your templates. We encourage you to use the introspection abilities of the python interpreter to find new ways to manipulate fields. When introspection fails, you should try reading the source for insight into how things work and how you can use things to your advantage.

If you come up with a solution that you feel is useful to others and wish to share it, please let us know via email or the mailing list, and we'll add it to this document.

9.2 Removing Fields Per-instance

Sometimes, you create a form which has fields that aren't useful in all circumstances or to all users. While it is indeed possible with form inheritance to define a form with exactly the fields you need, sometimes it is necessary to just tweak an existing form. Luckily, forms can have fields removed post-instantiation by using the `del` keyword:

```python
class MagazineIssueForm(Form):
    title  = StringField()
    year   = IntegerField('Year')
    month  = SelectField(choices=MONTHS)

def edit_issue():
    publication = get_something_from_db()
    form = MagazineIssueForm(...)

    if publication.frequency == 'annual':
        del form.month

    # render our form
```

Removing a field from a form will cause it to not be validated, and it will not show up when iterating the form. It's as if the field was never defined to begin with. Note that you cannot add fields in this way, as all fields must exist on the form when processing input data.

9.3 Dynamic Form Composition

This is a rare occurrence, but sometimes it's necessary to create or modify a form dynamically in your view. This is possible by creating internal subclasses:

```
def my_view():
    class F(MyBaseForm):
        pass

    F.username = StringField('username')
    for name in iterate_some_model_dynamically():
        setattr(F, name, StringField(name.title()))

    form = F(request.POST, ...)
    # do view stuff
```

For more form composition tricks, refer to this mailing list post

9.4 Rendering Errors

In your template, you will often find yourself faced with the repetitive task of rendering errors for a form field. Here's a Jinja2 macro that may save you time:

```
{% macro with_errors(field) %}
    <div class="form_field">
    {% if field.errors %}
        {% set css_class = 'has_error ' + kwargs.pop('class', '') %}
        {{ field(class=css_class, **kwargs) }}
        <ul class="errors">{% for error in field.errors %}<li>{{ error|e }}</li>{% endfor %}</ul>
    {% else %}
        {{ field(**kwargs) }}
    {% endif %}
    </div>
{% endmacro %}

Usage: {{ with_errors(form.field, style='font-weight: bold') }}
```

9.5 Specialty Field Tricks

By using widget and field combinations, it is possible to create new behaviours and entirely new ways of displaying a form input to the user.

A classic example is easily supported using the *widget*= keyword arg, such as making a hidden field which stores and coerces integer data:

```
user_id = IntegerField(widget=HiddenInput())
```

Alternatively, you can create a field which does this by subclassing:

```
class HiddenInteger(IntegerField):
    widget = HiddenInput()
```

Some fields support even more sophisticated customization.For example, what if a multiple-select was desired where instead of using a multi-row `<select>`, a series of checkboxes was used? By using widgets, one can get that behavior very easily:

```
class MultiCheckboxField(SelectMultipleField):
    """
    A multiple-select, except displays a list of checkboxes.

    Iterating the field will produce subfields, allowing custom rendering of
    the enclosed checkbox fields.
    """
    widget = widgets.ListWidget(prefix_label=False)
    option_widget = widgets.CheckboxInput()
```

By overriding *option_widget*, our new multiple-select when iterated will now produce fields that render as checkboxes.

CHAPTER 10

Crash Course

So you've cracked your knuckles and started working on that awesome python webapp you want to write. You get through writing a few pages and finally you need to tackle that loathsome task: form input handling and validation. Enter WTForms.

But why do I need *yet another* framework? Well, some webapp frameworks take the approach of associating database models with form handling. While this can be handy for very basic create/update views, chances are not every form you need can map directly to a database model. Or maybe you already use a generic form handling framework but you want to customize the HTML generation of those form fields, and define your own validation.

With WTForms, your form field HTML can be generated for you, but we let you customize it in your templates. This allows you to maintain separation of code and presentation, and keep those messy parameters out of your python code. Because we strive for loose coupling, you should be able to do that in any templating engine you like, as well.

10.1 Download / Installation

The easiest way to install WTForms is by using easy_install or pip:

```
easy_install WTForms
pip install WTForms
```

You can also download WTForms manually from PyPI and then run `python setup.py install`.

If you're the sort that likes to risk it all and run the latest version from Git, you can grab a packaged up version of the tip, or head over to the project page and clone the repository.

10.2 Key Concepts

- `Forms` are the core container of WTForms. Forms represent a collection of fields, which can be accessed on the form dictionary-style or attribute style.
- `Fields` do most of the heavy lifting. Each field represents a *data type* and the field handles coercing form input to that datatype. For example, *IntegerField* and *StringField* represent two different data types. Fields contain a number of useful properties, such as a label, description, and a list of validation errors, in addition to the data the field contains.
- Every field has a `Widget` instance. The widget's job is rendering an HTML representation of that field. Widget instances can be specified for each field but every field has one by default which makes sense. Some fields are simply conveniences, for example `TextAreaField` is simply a `StringField` with the default widget being a `TextArea`.

- In order to specify validation rules, fields contain a list of `Validators`.

10.3 Getting Started

Let's get right down to business and define our first form:

```
from wtforms import Form, BooleanField, StringField, validators

class RegistrationForm(Form):
    username     = StringField('Username', [validators.Length(min=4, max=25)])
    email        = StringField('Email Address', [validators.Length(min=6, max=35)])
    accept_rules = BooleanField('I accept the site rules', [validators.InputRequired()])
```

When you create a form, you define the fields in a way that is similar to the way many ORM's have you define their columns: By defining class variables which are instantiations of the fields.

Because forms are regular Python classes, you can easily extend them as you would expect:

```
class ProfileForm(Form):
    birthday  = DateTimeField('Your Birthday', format='%m/%d/%y')
    signature = TextAreaField('Forum Signature')

class AdminProfileForm(ProfileForm):
    username = StringField('Username', [validators.Length(max=40)])
    level    = IntegerField('User Level', [validators.NumberRange(min=0, max=10)])
```

Via subclassing, *AdminProfileForm* gains all the fields already defined in *ProfileForm*. This allows you to easily share common subsets of fields between forms, such as the example above, where we are adding admin-only fields to *ProfileForm*.

10.3.1 Using Forms

Using a form is as simple as instantiating it. Consider the following django-like view, using the *RegistrationForm* we defined earlier:

```
def register(request):
    form = RegistrationForm(request.POST)
    if request.method == 'POST' and form.validate():
        user = User()
        user.username = form.username.data
        user.email = form.email.data
        user.save()
        redirect('register')
    return render_response('register.html', form=form)
```

First, we instantiate the form, providing it with any data available in `request.POST`. We then check if the request is made using POST, and if it is, we validate the form, and check that the user accepted the rules. If successful, we create a new User and assign the data from the validated form to it, and save it.

Editing existing objects

Our earlier registration example showed how to accept input and validate it for new entries, but what if we want to edit an existing object? Easy:

```
def edit_profile(request):
    user = request.current_user
    form = ProfileForm(request.POST, user)
    if request.method == 'POST' and form.validate():
        form.populate_obj(user)
        user.save()
        redirect('edit_profile')
    return render_response('edit_profile.html', form=form)
```

Here, we instantiate the form by providing both request.POST and the user object to the form. By doing this, the form will get any data that isn't present in the post data from the *user* object.

We're also using the form's *populate_obj* method to re-populate the user object with the contents of the validated form. This method is provided for convenience, for use when the field names match the names on the object you're providing with data. Typically, you will want to assign the values manually, but for this simple case it's perfect. It can also be useful for CRUD and admin forms.

Exploring in the console

WTForms forms are very simple container objects, and perhaps the easiest way to find out what's available to you in a form is to play around with a form in the console:

```
>>> from wtforms import Form, StringField, validators
>>> class UsernameForm(Form):
...     username = StringField('Username', [validators.Length(min=5)], default=u'test')
...
>>> form = UsernameForm()
>>> form['username']
<wtforms.fields.StringField object at 0x827eccc>
>>> form.username.data
u'test'
>>> form.validate()
False
>>> form.errors
{'username': [u'Field must be at least 5 characters long.']}
```

What we've found here is that when you instantiate a form, it contains instances of all the fields, which can be accessed via either dictionary-style or attribute-style. These fields have their own properties, as does the enclosing form.

When we validate the form, it returns False, meaning at least one validator was not satisfied. `form.errors` will give you a summary of all the errors.

```
>>> form2 = UsernameForm(username=u'Robert')
>>> form2.data
{'username': u'Robert'}
>>> form2.validate()
True
```

This time, we passed a new value for username when instantiating UserForm, and it was sufficient to validate the form.

10.3.2 How Forms get data

In addition to providing data using the first two arguments (*formdata* and *obj*), you can pass keyword arguments to populate the form. Note though that a few names are reserved: *formdata*, *obj*, and *prefix*.

formdata takes precendence over *obj*, which itself takes precedence over keyword arguments. For example:

```
def change_username(request):
    user = request.current_user
    form = ChangeUsernameForm(request.POST, user, username='silly')
    if request.method == 'POST' and form.validate():
        user.username = form.username.data
        user.save()
        return redirect('change_username')
    return render_response('change_username.html', form=form)
```

While you almost never use all three methods together in practice, it illustrates how WTForms looks up the *username* field:

1. If a form was submitted (request.POST is not empty), process the form input. Even if there was no form input for this field in particular, if there exists form input of any sort, then we will process the form input.
2. If there was no form input, then try the following in order:
   (a) Check if *user* has an attribute named *username*.
   (b) Check if a keyword argument named *username* was provided.
   (c) Finally, if everything else fails, use the default value provided by the field, if any.

10.3.3 Validators

Validation in WTForms is done by providing a field with a set of validators to run when the containing form is validated. You provide these via the field constructor's second argument, *validators*:

```
class ChangeEmailForm(Form):
    email = StringField('Email', [validators.Length(min=6, max=120), validators.Email()])
```

You can provide any number of validators to a field. Typically, you will want to provide a custom error message:

```
class ChangeEmailForm(Form):
    email = StringField('Email', [
        validators.Length(min=6, message=_(u'Little short for an email address?')),
        validators.Email(message=_(u'That\'s not a valid email address.'))
    ])
```

It is generally preferable to provide your own messages, as the default messages by necessity are generic. This is also the way to provide localised error messages.

For a list of all the built-in validators, check the `Validators Reference`

10.3.4 Rendering Fields

Rendering a field is as simple as coercing it to a string:

```
>>> from wtforms import Form, StringField
>>> class SimpleForm(Form):
...   content = StringField('content')
...
>>> form = SimpleForm(content='foobar')
>>> str(form.content)
'<input id="content" name="content" type="text" value="foobar" />'
>>> unicode(form.content)
u'<input id="content" name="content" type="text" value="foobar" />'
```

However, the real power comes from rendering the field with its `__call__()` method. By calling the field, you can provide keyword arguments, which will be injected as html attributes in the output:

```
>>> form.content(style="width: 200px;", class_="bar")
u'<input class="bar" id="content" name="content" style="width: 200px;" type="text" value="foobar" />
```

Now let's apply this power to rendering a form in a Jinja template. First, our form:

```
class LoginForm(Form):
    username = StringField('Username')
    password = PasswordField('Password')

form = LoginForm()
```

And the template:

```
<form method="POST" action="/login">
    <div>{{ form.username.label }}: {{ form.username(class="css_class") }}</div>
    <div>{{ form.password.label }}: {{ form.password() }}</div>
</form>
```

Alternately, if you're using Django templates, you can use the *form_field* templatetag we provide in our Django extension, when you want to pass keyword arguments:

```
{% load wtforms %}
<form method="POST" action="/login">
    <div>
        {{ form.username.label }}:
        {% form_field form.username class="css_class" %}
    </div>
    <div>
        {{ form.password.label }}:
        {{ form.password }}
    </div>
</form>
```

Both of these will output:

```
<form method="POST" action="/login">
    <div>
        <label for="username">Username</label>:
        <input class="css_class" id="username" name="username" type="text" value="" />
    </div>
    <div>
        <label for="password">Password</label>:
        <input id="password" name="password" type="password" value="" />
    </div>
</form>
```

WTForms is template engine agnostic, and will work with anything that allows attribute access, string coercion, and/or function calls. The *form_field* templatetag is provided as a convenience as you can't pass arguments in Django templates.

10.3.5 Displaying Errors

Now that we have a template for our form, let's add error messages:

```
<form method="POST" action="/login">
    <div>{{ form.username.label }}: {{ form.username(class="css_class") }}</div>
    {% if form.username.errors %}
        <ul class="errors">{% for error in form.username.errors %}<li>{{ error }}</li>{% endfor %}</u
    {% endif %}

    <div>{{ form.password.label }}: {{ form.password() }}</div>
    {% if form.password.errors %}
        <ul class="errors">{% for error in form.password.errors %}<li>{{ error }}</li>{% endfor %}</u
    {% endif %}
</form>
```

If you prefer one big list of errors at the top, this is also easy:

```
{% if form.errors %}
    <ul class="errors">
        {% for field_name, field_errors in form.errors|dictsort if field_errors %}
            {% for error in field_errors %}
                <li>{{ form[field_name].label }}: {{ error }}</li>
            {% endfor %}
        {% endfor %}
    </ul>
{% endif %}
```

As error handling can become a rather verbose affair, it is preferable to use Jinja macros (or equivalent) to reduce boilerplate in your templates. (*example*)

10.3.6 Custom Validators

There are two ways to provide custom validators. By defining a custom validator and using it on a field:

```
from wtforms.validators import ValidationError

def is_42(form, field):
    if field.data != 42:
        raise ValidationError('Must be 42')

class FourtyTwoForm(Form):
    num = IntegerField('Number', [is_42])
```

Or by providing an in-form field-specific validator:

```
class FourtyTwoForm(Form):
    num = IntegerField('Number')

    def validate_num(form, field):
        if field.data != 42:
            raise ValidationError(u'Must be 42')
```

For more complex validators that take parameters, check the *Custom validators* section.

10.4 Next Steps

The crash course has just skimmed the surface on how you can begin using WTForms to handle form input and validation in your application. For more information, you'll want to check the following:

- The *WTForms documentation* has API documentation for the entire library.
- *Solving Specific Problems* can help you tackle specific integration issues with WTForms and other frameworks.

- The mailing list is where you can get help, discuss bugs in WTForms, and propose new features.

CHAPTER 11

Internationalization (i18n)

Localizing strings in WTForms is a topic that frequently comes up in the mailing list. While WTForms does not provide its own localization library, you can integrate WTForms with almost any gettext-like framework easily.

In WTForms, the majority of messages that are transmitted are provided by you, the user. However, there is support for translating some of the *built-in* messages in WTForms (such as errors which occur during data coercion) so that the user can make sure the user experience is consistent.

11.1 Translating user-provided messages

This is not actually any specific feature in WTForms, but because the question is asked so frequently, we need to address it here: **WTForms does -not- translate any user-provided strings.**

This is not to say they can't be translated, but that it's up to you to deal with providing a translation for any passed-in messages. WTForms waits until the last moment (usually validation time) before doing anything with the passed in message (such as interpolating strings) thus giving you the opportunity to e.g. change your locale before validation occurs, if you are using a suitable "lazy proxy".

Here's a simple example of how one would provide translated strings to WTForms:

```
from somelibrary import ugettext_lazy as _
from wtforms import Form, StringField, IntegerField, validators as v

class RegistrationForm(Form):
    name = StringField(_(u'Name'), [v.InputRequired(_(u'Please provide your name'))])
    age = IntegerField(
        _(u'Age'),
        [v.NumberRange(min=12, message=_(u'Must be at least %(min)d years old.'))]
    )
```

The field label is left un-perturbed until rendering time in a template, so you can easily provide translations for field labels if so desired. In addition, validator messages with format strings are not interpolated until the validation is run, so you can provide localization there as well.

11.2 Translating built-in messages

There are some messages in WTForms which are provided by the framework, namely default validator messages and errors occuring during the processing (data coercion) stage. For example, in the case of the IntegerField above, if someone entered a value which was not valid as an integer, then a message like "Not a valid integer value" would be displayed.

11.2.1 Using the built-in translations provider

WTForms now includes a basic translations provider which uses the stdlib *gettext* module to localize strings based on locale information distributed with the package. Localizations for several languages are included, and we hope that soon there will be more submitted.

To use the builtin translations provider, simply pass locale languages as locales in the *meta* section of the `constructor` of your form:

```
form = MyForm(request.form, meta={'locales': ['en_US', 'en']})
```

Alternately, if you are localizing application-wide you can define locales at meta-level in a subclass of Form:

```
class MyBaseForm(Form):
    class Meta:
        locales = ['es_ES', 'es']
```

Now simply have all your forms be a subclass of *MyBaseForm* and you will have all your default messages output in spanish.

11.2.2 Writing your own translations provider

For this case, we provide the ability to give a translations object on a subclass of Form, which will then be called to translate built-in strings.

An example of writing a simple translations object:

```
from mylibrary import ugettext, ungettext
from wtforms import Form

class MyTranslations(object):
    def gettext(self, string):
        return ugettext(string)

    def ngettext(self, singular, plural, n):
        return ungettext(singular, plural, n)

class MyBaseForm(Form):
    def _get_translations(self):
        return MyTranslations()
```

You would then use this new base Form class as the base class for any forms you create, and any built-in messages from WTForms will be passed to your gettext/ngettext implementations.

You control the object's constructor, its lifecycle, and everything else about it, so you could, for example, pass the locale per-form instantiation to the translation object's constructor, and anything else you need to do for translations to work for you.

CHAPTER 12

WTForms Changelog

12.1 Version 2.0.1

Released July 1, 2014

- Update wheel install to conditionally install ordereddict for python 2.6.
- Doc improvements

12.2 Version 2.0

Released May 20, 2014

- Add new *class Meta* paradigm for much more powerful customization of WTForms.
- Move i18n into core. Deprecate *wtforms.ext.i18n*.
- Move CSRF into core. Deprecate *wtforms.ext.csrf*.
- Fix issue rendering SelectFields with `value=True`
- Make *DecimalField* able to use babel locale-based number formatting.
- Drop Python 3.2 support (Python3 support for 3.3+ only)
- passing `attr=False` to WTForms widgets causes the value to be ignored.
- *Unique* validator in *wtforms.ext.sqlalchemy* has been removed.

12.3 Version 1.0.5

Released September 10, 2013

- Fix a bug in validators which causes translations to happen once then clobber any future translations.
- ext.sqlalchemy / ext.appengine: minor cleanups / deprecation.
- Allow blank string and the string 'false' to be considered false values for BooleanField (configurable). This is technically a breaking change, but it is not likey to affect the majority of users adversely.
- ext.i18n form allows passing LANGUAGES to the constructor.

12.4 Version 1.0.4

Released April 28, 2013

- Add widgets and field implementations for HTML5 specialty input types.
- ext.appengine: Add NDB support.
- Add translations: Korean, Traditional Chinese

12.5 Version 1.0.3

Released January 24, 2013

- Tests complete in python 3.2/3.3.
- Localization for ru, fr.
- Minor fixes in documentation for clarity.
- FieldList now can take validators on the entire FieldList.
- ext.sqlalchemy model_form:
  - Fix issue with QuerySelectField
  - Fix issue in ColumnDefault conversion
  - Support Enum type
- Field class now allows traversal in Django 1.4 templates.

12.6 Version 1.0.2

Released August 24, 2012

- We now support Python 2.x and 3.x on the same codebase, thanks to a lot of hard work by Vinay Sajip.
- Add in ability to convert relationships to ext.sqlalchemy model_form
- Built-in localizations for more languages
- Validator cleanup:
  - Distinguish Required validator into InputRequired and DataRequired
  - Better IP address validation, including IPv6 support.
  - AnyOf / NoneOf now work properly formatting other datatypes than strings.
  - Optional validator can optionally pass through whitespace.

12.7 Version 1.0.1

Released February 29, 2012

- Fixed issues related to building for python 3 and python pre-releases.
- Add object_data to fields to get at the originally passed data.

12.8 Version 1.0

Released February 28, 2012

- Output HTML5 compact syntax by default.
- Substantial code reorg, cleanup, and test improvements
- Added ext.csrf for a way to implement CSRF protection
- ext.sqlalchemy:
  - Support PGInet, MACADDR, and UUID field conversion
  - Support callable defaults
- ext.appengine:
  - model_form now supports generating forms with the same ordering as model.
  - ReferencePropertyField now gets get_label like the other ORM fields
- Add localization support for WTForms built-in messages
- Python 3 support (via 2to3)
- Minor changes/fixes:
  - An empty label string can be specified on fields if desired
  - Option widget can now take kwargs customization
  - Field subclasses can provide default validators as a class property
  - DateTimeField can take time in microseconds
  - Numeric fields all set .data to None on coercion error for consistency.

12.9 Version 0.6.3

Released April 24, 2011

- Documentation: Substantial documentation improvements, including adding Crash Course as a sphinx document.
- ext.django: QuerySetSelectField (and ModelSelectField) now accept get_label similar to sqlalchemy equivalents.
- ext.appengine
  - model_form fixes: FloatField(#50), TimeField, DateTimeField(#55)
  - ReferencePropertyField: now properly stores model object, not key. (#48)

12.10 Version 0.6.2

Released January 22, 2011

- Bug Fixes:
  - ext.appengine: various field fixes (#34, #48), model_form changes (#41)

  - Fix issue in Optional with non-string input.
  - Make numeric fields more consistent.
- Tests: Improve test coverage substantially.

12.11 Version 0.6.1

Released September 17th, 2010

- Bug Fixes:
  - ext.appengine ReferencePropertyField (#36, #37)
  - dateutil fields: render issue (r419), and consistency issue (#35)
  - Optional validator failed when raw_data was absent (r418)
- Documentation: docs now mention HTML escaping functionality (#38)
- Add preliminary support for providing a translations object that can translate built-in validation and coercion errors (#32)

12.12 Version 0.6

Released April 25th, 2010.

- Widgets:
  - HTML is now marked as safe (using __html__) so that compatible templating engines will not auto-escape it.
- Fields:
  - Field._default is now Field.default.
  - All fields now have a *raw_data* property.
  - Fields which are select fields (including those in .ext) can be iterated to produce options, and have an option_widget kwarg.
  - Minor bugfixes and cleanup in FieldList, Select(Multiple)Field, QuerySelectField to address behavioral consistency.
  - Added FloatField, based on IntegerField.
- Extensions:
  - ext.appengine now supports FloatProperty and GeoPtProperty.
  - ext.sqlalchemy QueryMultipleSelectField changed to QuerySelectMultipleField.

12.13 Version 0.5

Released February 13th, 2010.

- Added a BaseForm class which provides the core processing and validation functionality of Form without requiring declarative subclassing.

- Fields:
  - Field labels now default to a humanized field name.
  - Fields now have a *short_name* property which is the un-prefixed name.
  - DecimalField now rounds values for display without float coercion. See docs for details on how to format decimals.
- Extensions:
  - ext.sqlalchemy.fields now has an additional QuerySelectMultipleField, and all fields can now support multiple-column primary keys.
  - ext.sqlalchemy.orm contains tools for making forms from ORM models.
  - Added ext.dateutil for flexible date-time parsing.
  - Added ext.appengine contributed by Rodrigo Moraes.
- Added AnyOf and NoneOf validators.

12.14 Version 0.4

Released October 10th, 2009.

- Fields have much greater control over input processing. Filters have been added to implement a simple way to transform input data.
- Added fields that encapsulate advanced data structures such as dynamic lists or child forms for more powerful field composing.
- Fields now use widgets for rendering.
- All built-in validators have been converted to classes to clean up the code.
- *Form.auto_populate* and *Field.populate* were renamed to *populate_obj* to clarify that they populate another object, not the Form or Field. This is an API breaking change.
- Dropped support for Python 2.3.

12.15 Version 0.3.1

Released January 24th, 2009.

- Several fixes were made to the code and tests to make WTForms compatible with Python 2.3/2.4.
- Form's properties can now be accessed via dictionary-style access such as *form['author']*. This also has the intended effect of making variable lookups in Django templates more reliable.
- Form and Field construction changes: Form now uses a metaclass to handle creating its *_unbound_fields* property, and Field construction now gives an instance of the new *UnboundField* class instead of using a partial function application. These are both internal changes and do not change the API.

12.16 Version 0.3

Released January 18th, 2009.

- Validation overhaul: Fields are now responsible for their own validation, instead of mostly relying on Form. There are also new pre_validate and post_validate hooks on subfields, adding a great deal of flexibility when dealing with field-level validation. Note that this is an API breaking change if you have any subfields that override *Field.validate*. These will need to be updated to use the new hooks.
- Changes in how *process_data* and *process_formdata* are called:
  - *process_data* no longer accepts the *has_formdata* parameter.
  - At form instantiation time, *process_data* will be called only once for each field. If a model object is provided which contains the property, then this value is used. Otherwise, a keyword argument if specified is used. Failing that, the field's default value is used.
  - If any form data is sent, *process_formdata* will be called after *process_data* for each field. If no form data is available for the given field, it is called with an empty list.
- wtforms.ext.django has been overhauled, both to mirror features and changes of the Django 1.0 release, and to add some useful fields for working with django ORM data in forms.
- The *checker* keyword argument to SelectField, SelectMultipleField, and RadioField has been renamed to *coerce* to reflect the actual functionality of this callable.

12.17 Version 0.2

Released January 13th, 2009.

- We have documentation and unit tests!
- Fields now have a *flags* property which contain boolean flags that are set either by the field itself or validators being specified on a field. The flags can then be used in checks in template or python code.
- Changed the way fields take parameters, they are no longer quasi magic. This is a breaking change. Please see the documentation for the new syntax.
- Added optional description argument to Field, accessible on the field as *description*. This provides an easy way to define e.g. help text in the same place as the form.
- Added new semantics for validators which can stop the validation chain, with or without errors.
- Added a regexp validator, and removed the not_empty validator in favour of two validators, optional and required. The new validators allow control over the validation chain in addition to checking emptiness.
- Renamed wtforms.contrib to wtforms.ext and reorganised wtforms.ext.django. This is a breaking change if you were using the django extensions, but should only require changing your imports around a little.
- Better support for other frameworks such as Pylons.

12.18 Version 0.1

Released July 25th, 2008.

- Initial release.

CHAPTER 13

Contributing to WTForms

WTForms is an open-source library and changing and evolving over time. To that end, we are supported by the contributions of many people in the python community.

13.1 How to Contribute

WTForms is now on GitHub, so all contributions should be either associated with a pull request or with a ticket & patch.

13.2 Contribution Guidelines

Code submitted should:

- Be formatted according to the PEP8 style guideline **except** that it does not need to confirm to the 79-column limit requirement of the guideline.
- Have tests
  - Unless it's a bugfix, it should pass existing tests.
  - New classes or methods should mean new unit tests or extending existing tests.
  - Bugfixes can probably do with a regression test too (a test that would fail without this fix)
- Use naming schemes consistent with WTForms conventions
- Work on all versions of Python that WTForms currently supports (and python3 without needing 2to3). Take advantage of Travis-CI for running tests on all supported Python versions.

13.3 Note on API compatibility

WTForms is a very small library, but yet it's possible to break API compatibility pretty easily. We are okay with breaking API compatibility for compelling features or major changes that we feel are worthwhile inclusions to the WTForms core, but realize that any API compatiblity break will delay the inclusion of your ticket to the next major release.

Some examples of API compatibility breaks include:

- Adding new attributes or methods to the base Form class

- Modifying the number of required arguments of core methods like `process()`
- Changing the default behavior of a field.

However, it is still possible to add new features to WTForms without breaking API compatibility. For example, if one were looking to add Babel locale support to DecimalField, it could be done so long as by default, DecimalField behaved the same as it did before. This could look something like:

1. Add a keyword arg `use_locale` to the constructor
2. Make the keyword default to `False` so the behavior without this arg is identical to the previous bevhavior.
3. Add your functionality and make sure all existing DecimalField tests work unchanged (and of course add new tests for the new functionality).

CHAPTER 14

What's New in WTForms 2

WTForms 2 is the first major version bump since WTForms 1.0. Coming with it are a number of major changes that allow far more customization of core WTForms features. This is done to make WTForms even more capable when working along with companion libraries.

14.1 New Features

- *Class Meta* paradigm allows customization of many aspects of WTForms.
- *CSRF* and *i18n* are core features not needing extensions anymore.
- Widget rendering changes:
  - Passing `<attribute name>=False` to WTForms widget rendering is now ignored, making it easier to deal with boolean HTML attributes.
  - Creating an html attribute `data-foo` can be done by passing the keyword `data_foo` to the widget.

14.2 Deprecated API's

These API's still work, but in most cases will cause a DeprecationWarning. The deprecated API's will be removed in WTForms 3.0, so write code against the new API's unless it needs to work across both WTForms 1.x and 2.x

- **Core**
  - `Form._get_translations` Use `Meta.get_translations` instead.
  - The `TextField` alias for `StringField` is deprecated.
  - `wtforms.validators.Required` is now `wtforms.validators.DataRequired`
  - `wtforms.fields._unset_value` is now `wtforms.utils.unset_value`
- **WTForms Extensions** All the extensions are being deprecated. We feel like the extensions we had would actually benefit from being pulled outside the WTForms package, because it would allow them to have a separate release schedule that suits their companion libraries.
  - `wtforms.ext.appengine` Is deprecated, see WTForms-Appengine
  - `wtforms.ext.csrf` CSRF protection is now *built in*
  - `wtforms.ext.dateutil` Is deprecated, but does not have a new home yet.
  - `wtforms.ext.django` Is deprecated. See WTForms-Django

- `wtforms.ext.i18n` i18n is now *built in*
- `wtforms.ext.sqlalchemy` Is deprecated, look at WTForms-Alchemy (docs)

14.3 Low-level Changes

Most of these changes shouldn't affect the typical library user, however we are including these changes for completeness for those who are creating companion libraries to WTForms.

- `BaseForm._fields` is now an OrderedDict, not a plain dict.
- `FormMeta` now manages an attribute called `_wtforms_meta` which is a subclass of any `class Meta` defined on ancestor form classes.
- A new keyword-param called simply `data=` to the Form constructor has been added and positioned as the place where soon we will be able to accept structured data which is neither formdata, object data, or defaults. Currently this parameter is merged with the kwargs, but the intention is to handle other structured data (think JSON).
- `Filters` on fields stop on the first ValueError, instead of continuing on to the next one.

Indices and tables:

- *genindex*
- *modindex*
- *search*