Watson - Form

Release 2.0.0

September 30, 2014

Contents

Make working with HTML forms more tolerable.

CHAPTER 1

Build Status

CHAPTER 2

Installation

```
pip install watson-form
```

CHAPTER 3

Testing

Watson can be tested with py.test. Simply activate your virtualenv and run `python setup.py test`.

CHAPTER 4

Contributing

If you would like to contribute to Watson, please feel free to issue a pull request via Github with the associated tests for your code. Your name will be added to the AUTHORS file under contributors.

CHAPTER 5

Table of Contents

5.1 Usage

Note: Examples below are from when watson-form is being used within watson-framework. Feel free to ignore references to watson.framework if you are not using it in that capacity.

Forms are defined in a declarative way within Watson. This means that you only need to define fields you want without any other boilerplate code.

```
from watson import form
from watson.form import fields

class Login(form.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    submit = fields.Submit(value='Login', button_mode=True)
```

Which when implemented in a view would output:

```
<html>
    <body>
        <form>
            <label for="username">Username</label><input type="text" name="username" />
            <label for="password">Password</label><input type="text" name="password" />
            <button type="submit">Login</button>
        </form>
    </body>
</html>
```

5.1.1 Field types

Fields are referenced by their HTML element name. Whenever a field is defined within a form any additional keyword arguments are used as attributes on the element itself. Current fields that are included are:

Field	Output
Input	<input type="" />
Button	<button></button>
Textarea	<textarea></textarea>
Select	<select></select>

There are also a bunch of convenience classes as well which may add additional validators and filters to the field.

Field	Output
Input	<input type="" />
Radio	<input type="radio" />
Checkbox	<input type="checkbox" />
Text	<input type="text" />
Date	<input type="date" />
Email	<input type="email" />
Hidden	<input type="hidden" />
Csrf	<input type="csrf" />
Password	<input type="password" />
File	<input type="file" />
Submit	<input type="submit" /> or <button>Submit</button>

These can all be imported from the `watson.form.fields` module.

5.1.2 Populating and binding objects to a form

Form data can be populated with any standard Python dict.

```
form = forms.Login()
form.data = {'username': 'Simon'}
```

These values can then be retrieved by:

```
form.username.value  # Simon
```

If the field has been through the validation/filter process, you can still retrieve the original value that was submitted by:

```
form.username.original_value  # Simon
```

Binding an object to the form

Sometimes it's worth being able to bind an object to the form so that any posted data can automatically be injected into the object. This is a relatively simple task to achieve:

Object entities

```
class User(object):
    username = None
    password = None
    email = None
```

Edit user form

```
from watson import form
from watson.form import fields

class User(forms.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    email = fields.Email(label='Email Address')
```

Controller responsible for saving the user

```
from watson.framework import controllers
from app import forms
```

```
class Login(controllers.Rest):
    def POST(self):
        user = User()
        form = forms.User('user')
        form.bind(user)
        form.data = self.request.post
        if form.is_valid():
            user.save()  # save the updated user data
```

When is_valid() is called the POST'd data will be injected directly into the User object. While this is great for simple CRUD interfaces, things can get more complex when an object contains other objects. To resolve this we have to define a mapping to map the flat post data to the various objects (we only need to define the mapping for data that isn't a direct mapping).

A basic mapping consists of a dict of key/value pairs where the value is a tuple that denotes the object 'tree'.

```
mapping = {
    'field_name': ('attribute', 'attribute', 'attribute')
}
```

We'll take the same example from above, but modify it slightly so that our User object now also contains a Contact object (note that some of this code such as the entities would be handled automatically by your ORM of choice).

Object entities

```
class User(object):
    username = None
    password = None
    contact = None

    def __init__(self):
        self.contact = Contact()

class Contact(object):
    email = None
    phone = None
```

Edit user form

```
from watson import form
from watson.form import fields

class User(forms.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    email = fields.Email(label='Email Address')
    phone = fields.Email(label='Phone Number')
```

Controller responsible for saving the user

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def POST(self):
        user = User()
        form = forms.User('user')
        form.bind(user, mapping={'email': ('contact', 'email'), 'phone': ('contact', 'phone')})
        form.data = self.request.post
```

```
        if form.is_valid():
            user.save()  # save the updated user data
```

5.1.3 Filters and Validators

Filters and validators allow you to sanitize and modify your data prior to being used within your application. By default, all fields have the Trim filter which removes whitespace from the value of the field.

Note: Filters and Validators are from watson-filters and watson-validators respectively.

When the is_valid() method is called on the form each field is filtered, and then validated.

To add new validators and filters to a field you simply add them as a keyword argument to the field definition.

```
from watson import form
from watson.form import fields
from watson import validators

class Login(form.Form):
    username = fields.Text(label='Username', validators=[validators.Length(min=10)])
    password = fields.Password(label='Password', validators=[validators.Required()])
    # required can actually be set via required=True
    submit = fields.Submit(value='Login', button_mode=True)
```

For a full list of validators and filters check out filters and validators in the reference library.

5.1.4 Validating post data

Validating forms is usually done within a controller. We'll utilize the Login form above to demonstrate this...

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def GET(self):
        form = forms.Login('login_form', action='/login')
        form.data = self.redirect_vars
        # populate the form with POST'd data to avoid the PRG issue
        # we don't really need to do this
        return {
            'form': form
        }

    def POST(self):
        form = forms.Login('login_form')
        form.data = self.request.post
        if form.is_valid():
            self.flash_messages.add('Successfully logged in')
            self.redirect('home')
        else:
            self.redirect('login')
```

With the above code, when a user hits /login, they are presented with a login form from the GET method of the controller. As they submit the form, the code within the POST method will execute. If the form is valid, then they will be redirected to whatever the 'home' route displays, otherwise they will be redirected back to the GET method again.

Errors upon validating

When is_valid() is called, all fields will be filtered and validated, and any subsequent error messages will be available via form.errors.

5.1.5 Protecting against CSRF (Cross site request forgery)

Cross site request forgery is a big issue with a lot of code bases. Watson provides a simple way to protect your users against it by using a decorator.

```
from watson import form
from watson.form import fields
from watson.form.decorators import has_csrf

@has_csrf
class Login(form.Form):
    username = fields.Text(label='Username')
    password = fields.Password(label='Password')
    submit = fields.Submit(value='Login', button_mode=True)
```

The above code will automatically add a new field (named csrf_token) to the form, which then will need to be rendered in your view. You will also need to pass the session into the form when it is instantiated so that the csrf token can be saved against the form.

```
from watson.framework import controllers
from app import forms

class Login(controllers.Rest):
    def GET(self):
        form = forms.Login('login_form', action='/login', session=self.request.session)
        form.data = self.redirect_vars
        return {
            'form': form
        }
```

As the form is validated (via is_valid()), the token will automatically be processed against the csrf validator.

5.2 Reference Library

5.2.1 watson.form.decorators

watson.form.decorators.**has_csrf**(*cls*)

Adds csrf protection to the form.

Adds a new field named 'csrf_token' to the form and overrides the set data method to retrieve the correct token.

If a form is csrf protected then a session object must be passed to the __init__ method so that a token can be created (if not already).

Example:

```
@has_csrf
class MyForm(Form):
    username = fields.Text(required=True)
    password = fields.Password(required=True)
```

5.2.2 watson.form.fields

class watson.form.fields.**Button**(*name=None*, *value=None*, *label=None*, *label_attrs=None*, ***kwargs*)
Creates a button, can be used instead of Input(type="button").

class watson.form.fields.**Checkbox**(*name=None*, *values=None*, *value=None*, ***kwargs*)
Creates a checkbox input.

Example:

```
field = Checkbox(name='test', label='My Radio Options', values=(('Test', 1), ('Testing', 2)))
str(field)

<fieldset>
    <legend>My Checkbox Options</legend>
    <label for="test_0">Test<input id="test_0" name="test" type="checkbox" /></label>
    <label for="test_1">Testing<input id="test_1" name="test" type="checkbox" /></label>
</fieldset>

field = Checkbox(name='test', label='My Checkbox', values=1)
str(field)=None

<label for="test"><input type="checkbox" name="test" value="1" />My Checkbox</label>
```

__init__(*name=None*, *values=None*, *value=None*, ***kwargs*)
Initializes the checkbox.

If a value is specified, then that value out of the available values will be checked. If multiple values are specified, then a checkbox group will be created.

Parameters

- **name** (*string*) – the name of the field
- **values** (*tuple|list*) – the values to be used
- **value** (*mixed*) – the value for the field

class watson.form.fields.**Csrf**(*name='csrf_token'*, *value=None*, ***kwargs*)
Creates an <input type="hidden" /> element for use in csrf protection.

__init__(*name='csrf_token'*, *value=None*, ***kwargs*)

class watson.form.fields.**Date**(*name=None*, *value=None*, *format='%Y-%m-%d'*, ***kwargs*)
Creates an <input type="date" /> element.

__init__(*name=None*, *value=None*, *format='%Y-%m-%d'*, ***kwargs*)

render(***kwargs*)
Format the date in the format the HTML5 spec requires.

class watson.form.fields.**Definition**(*class_*, **args*, ***kwargs*)
Placeholder form element which allows for the creation of new form elements when the form is instantiated.

__init__(*class_*, **args*, ***kwargs*)

class watson.form.fields.**Email**(*name=None*, *value=None*, ***kwargs*)
Creates an <input type="email" /> element.

__init__(*name=None*, *value=None*, ***kwargs*)

class watson.form.fields.**FieldMixin**(*name=None*, *value=None*, *label=None*, *label_attrs=None*, ***kwargs*)

A mixin that can be used as a base to simplify the creation of fields.

When defining a field, a fully instantiated field must be created with definition=False as an argument in it's __init__ method. This is to facilitate the way fields are defined in Form objects in 2.0.0.

label
watson.form.fields.Label

the label associated with the field

html
string

the html used to render the field

validators
list

the validators that will be used to validate the value

filters
list

the filters that will be used prior to validation

__init__(*name=None*, *value=None*, *label=None*, *label_attrs=None*, ***kwargs*)
Initializes the field with a specific name.

filter()
Filter the value on the field based on the associated filters.

Set the original_value of the field to the first value stored. Note, if this is called a second time, then the original value will be overridden.

name
Convenience method to retrieve the name of the field.

original_value
Return the original value for the field.

render_with_label()
Render the field with the label attached.

validate()
Validate the value of the field against the associated validators.

Returns A list of errors that have occurred when the field has been validated.

value
Return the value for the field.

If the field has been cleaned, the original value can be retrieved with FieldMixin.original_value.

class watson.form.fields.**GroupInputMixin**(*name=None*, *values=None*, *value=None*, ***kwargs*)
A mixin for form elements that are used in a group.

Related form elements are wrapped in a fieldset, with a common legend.

__init__(*name=None*, *values=None*, *value=None*, ***kwargs*)

has_multiple_elements()
Determine whether or not a field has multiple elements.

class watson.form.fields.**Hidden**(*name=None*, *value=None*, ***kwargs*)
Creates an <input type="hidden" /> element.

__init__(*name=None*, *value=None*, ***kwargs*)

class watson.form.fields.**Input**(*name=None*, *value=None*, *label=None*, *label_attrs=None*, ***kwargs*)
Creates an <input> field.

Custom input types can be created by sending type='type' through the __init__ method.

Example:

```
input = Input(type='text')  # <input type="text" />
```

render(***kwargs*)
Render the element as html.

Does not need to be called directly, as will be called by __str__ natively.

render_with_label(***kwargs*)
Render the element as html and include the label.

Output the element and prepend the <label> to it.

class watson.form.fields.**Label**(*text*, ***kwargs*)
A <label> tag which can be automatically included with fields.

html
string

the html used to render the label

text
string

the text associated with the label

__init__(*text*, ***kwargs*)

class watson.form.fields.**Password**(*name=None*, *value=None*, ***kwargs*)
Creates an <input type="password" /> element.

__init__(*name=None*, *value=None*, ***kwargs*)

class watson.form.fields.**Radio**(*name=None*, *values=None*, *value=None*, ***kwargs*)
Creates a radio input.

Example:

```
field = Radio(name='test', label='My Radio Options', values=(('Test', 1), ('Testing', 2)))
str(field)

<fieldset>
    <legend>My Radio Options</legend>
    <label for="test_0">Test<input id="test_0" name="test" type="radio" value="1" /></label>
    <label for="test_1">Testing<input id="test_1" name="test" type="radio" value="2" /></label>
</fieldset>

field = Radio(name='test', label='My Radio', values=1)
str(field)

<label for="test"><input type="radio" name="test" values="1" />My Radio</label>
```

__init__ (*name=None*, *values=None*, *value=None*, ***kwargs*)
Initializes the radio.

If a value is specified, then that value out of the available values will be checked. If multiple values are specified, then a radio group will be created.

class watson.form.fields.**Select** (*name=None*, *options=None*, *value=None*, *multiple=False*, ***kwargs*)
Creates a select field.

html
string

the html for the outer select element

option_html
string

the individual option html element

optgroup_html
string

the optgroup html element

options
list|dict

the options available

__init__ (*name=None*, *options=None*, *value=None*, *multiple=False*, ***kwargs*)
Initializes the select field.

If the options passed through are a dict, and the value of each key is a list or tuple, then an optgroup will be rendered, using the key as the label for the optgroup.

Parameters

- **name** (*string*) – the name of the field
- **options** (*list|dict*) – the options available
- **value** (*string*) – the selected value
- **multiple** (*bool*) – whether or not to allow multiple selections

Example:

```
field = Select(name='test', options=collections.OrderedDict([('Group One', [1, 2]), ('Group
str(field)
```

```
<select name="test">
    <opgroup label="Group One">
        <option value="1">1</option>
    </optgroup>
    <opgroup label="Group Two">
        <option value="2">2</option>
    </optgroup>
</select>
```

class watson.form.fields.**Submit** (*name=None*, *value=None*, *button_mode=False*, ***kwargs*)
Creates a submit input.

button_mode
bool

whether or not to render as <button> or <input>

__init__(*name=None*, *value=None*, *button_mode=False*, ***kwargs*)

class watson.form.fields.**Text**(*name=None*, *value=None*, ***kwargs*)
Creates an <input type="text" /> element.

__init__(*name=None*, *value=None*, ***kwargs*)

class watson.form.fields.**Textarea**(*name=None*, *value=None*, *label=None*, *label_attrs=None*, ***kwargs*)
Creates a textarea field.

5.2.3 watson.form.types

class watson.form.types.**FieldDescriptor**(*name*)
Allow set/get access to the field value via Form.field_name

Fields accessed via a Form object will provide access directly to set and get the value of the field. When the field is being rendered in a template, the set/get will provide access to the field object itself. Access to the field prior to rendering can be made via Form.fields[name] where name is the attribute name of the field.

__init__(*name*)

class watson.form.types.**Form**(*name=None*, *method='post'*, *action=None*, *detect_multipart=True*, *validators=None*, ***kwargs*)
Declarative HTML <form> management.

Example:

```
from watson.form import fields
class MyForm(Form):
    text = fields.Text(name='text', label='My TextField')
    another = fields.Checkbox(name='another[]')

form = MyForm('my_form')
form.text = 'Something'

# in view
{% form.open() %}    # <form name="my_form">
{% form.text %}      # <input name="text" type="text" value="Something" />
{% form.text.render_with_label() %} # <label for="text">My TextField</label><input id="text" nam
{% form.another %}   # <input name="another[]" />
{% form.close() %}   # </form>
```

__init__(*name=None*, *method='post'*, *action=None*, *detect_multipart=True*, *validators=None*, ***kwargs*)
Inititalize the form and set some default attributes.

Parameters

- **name** (*string*) – the name of the form
- **method** (*string*) – the http method to use
- **action** (*string*) – the url to submit the form to
- **detect_multipart** (*boolean*) – automatically set multipart/form-data

bind(*obj=None*, *mapping=None*, *ignored_fields=None*, *hydrate=True*)
Binds an object to the form.

Optionally additional mapping can be specified in order to set values on any of the classes that may exist within the object. If this method is called after the data has been set on the form, then the existing data will be overridden with the attributes on the object unless hydrate is set to false.

Parameters

- **obj** (*class|dict*) – the class to bind to the form.
- **mapping** (*dict*) – the mapping between the form fields and obj attributes.
- **ignored_fields** (*list|tuple*) – fields to ignore when binding.

Example:

```
form = ...
user = User(username='test')
form.bind(user)
form.username.value  # 'test'
```

close(*include_http_request=True*)
Render the end tag of the form.

If the form has the http_request_method input then include it in the tag by default.

Parameters **include_http_request** (*boolean*) – Whether or not to include the HTTP_REQUEST_METHOD field

data
Returns a dict containing all the field values.

Used as a shorthand method to retrieve data from all the form fields rather than having to access the fields themselves.

invalidate()
Invalidate the data that has been bound on the form.

This is called automatically when data is bound to the form and sets the forms validity to invalid.

is_valid()
Determine whether or not the form and relating values are valid.

Filter all the values on the fields associated with the form, and then validate each field. Will only execute the filter/validation steps if the form has not been previously validated, or has been invalidated.

Returns boolean value depending on the validity of the form.

open(***kwargs*)
Render the start tag of the form.

Any addition kwargs will be used within the attributes.

render(*with_tag='div'*, *with_label=True*)
Output the entire form as a string.

Called automatically by the __str__ method.

Parameters

- **with_tag** (*string*) – the tag to be used to separate the elements.
- **with_label** (*boolean*) – render each field with it's label.

Returns A string representation of the form.

class watson.form.types.**FormMeta**(*name*, *bases*, *attrs*)
Assigns the FieldDescriptor objects to the Form object.

__init__(*name*, *bases*, *attrs*)

class watson.form.types.**Multipart**(*name*, *method='post'*, *action=None*, ***kwargs*)

Convenience class for forms that should be multipart/form-data.

By default, the Form class will automatically detect whether or not a field is of type file, and convert it to multipart.

__init__(*name*, *method='post'*, *action=None*, ***kwargs*)

W