ToscaWidgets2 Documentation

Release 2.3.0

Alessandro Molina

Oct 12, 2021

Contents

ToscaWidgets is a HTML Widgets generation and management library.

It allows to create reusable widgets to show in web pages and manages the dependencies of the widgets like Javascript and CSS that those widgets might need to properly display and behave

```
class HelloWidget(twc.Widget):
    inline_engine_name = "kajiki"
    template = """
        <i>Hello ${w.name}</i>
    """

    name = twc.Param(description="Name of the greeted entity")
```

Widgets can then be displayed within your web pages to create reusable components or forms:

```
>>> HelloWidget(name="World").display()
<i>Hello World</i>
```

Widgets have support for:

- Templating based on Kajiki, Mako, Genshi and Jinja2
- Resources, to bring in Javascript and CSS dependencies they need.
- Parameters, to configure their behaviour.
- Validation, to ensure proper data was provided and show validation errors to users.
- Hooks, to drive their behaviour at runtime.

ToscaWidgets2 also provides a `tw2.forms` package with ready to use widgets to display Forms with input validation.

CHAPTER 1

Content

1.1 Getting Started

1.1.1 Enabling ToscaWidgets

ToscaWidgets is designed to work within a web request life cycle, so some of its features rely on the a current request object to be able to work a keep track of the state of widgets or resources for the whole duration of the request.

For this reason, to start using ToscaWidgets you need to wrap your WSGI application in the *tw2.core.middleware.TwMiddleware*, which is also used to configure ToscaWidgets itself:

```
def application(environ, start_response):
    response_headers = [('Content-type', 'text/plain')]
    start_response("200 OK", response_headers)
    return [b"Hello World!"]

from tw2.core.middleware import TwMiddleware
application = TwMiddleware(application)

from wsgiref.simple_server import make_server
httpd = make_server('', 8000, application)
print("Serving on port 8000...")
httpd.serve_forever()
```

You can also provide all options available to configure ToscaWidgets (those listed in *tw2.core.middleware.Config*) to `TwMiddleware` as keyword arguments to change ToscaWidgets configuration:

```
from tw2.core.middleware import TwMiddleware
application = TwMiddleware(application, debug=False)
```

Note: Debug mode is enabled by default in ToscaWidgets, so make sure you provide `debug=False` on production to leverage templates caching and other speedups.

Now that the middleare is in place, you can easily display any widget you want into your application:

```
from tw2.forms import SingleSelectField

def application(environ, start_response):
    widget = SingleSelectField(options=[1, 2, 3])
    output = widget.display()

    response_headers = [('Content-type', 'text/html')]
    start_response("200 OK", response_headers)
    return [b"<h1>Hello World!</h1>",
            b"<p>Pick one of the options</p>",
            output.encode('ascii')]

from tw2.core.middleware import TwMiddleware
application = TwMiddleware(application)

from wsgiref.simple_server import make_server
httpd = make_server('', 8000, application)
print("Serving on port 8000...")
httpd.serve_forever()
```

See *Widgets* and *Forms* to get started creating widgets and forms.

class tw2.core.middleware.**TwMiddleware**(*app*, *controllers=None*, ***config*)

ToscaWidgets middleware

This performs three tasks:

- Clear request-local storage before and after each request. At the start of a request, a reference to the middleware instance is stored in request-local storage.
- Proxy resource requests to ResourcesApp
- Inject resources

1.1.2 Configuration Options

class tw2.core.middleware.**Config**(***kw*)

ToscaWidgets Configuration Set

translator The translator function to use. (default: no-op)

default_engine The main template engine in use by the application. Widgets with no parent will display correctly inside this template engine. Other engines may require passing displays_on to `Widget.display()`. (default:string)

inject_resoures Whether to inject resource links in output pages. (default: True)

inject_resources_location A location where the resources should be injected. (default: head)

serve_resources Whether to serve static resources. (default: True)

res_prefix The prefix under which static resources are served. This must start and end with a slash. (default: /resources/)

res_max_age The maximum time a cache can hold the resource. This is used to generate a Cache-control header. (default: 3600)

serve_controllers Whether to serve controller methods on widgets. (default: True)

controller_prefix The prefix under which controllers are served. This must start and end with a slash. (default: /controllers/)

bufsize Buffer size used by static resource server. (default: 4096)

params_as_vars Whether to present parameters as variables in widget templates. This is the behaviour from ToscaWidgets 0.9. (default: False)

debug Whether the app is running in development or production mode. (default: True)

validator_msgs A dictionary that maps validation message names to messages. This lets you override validation messages on a global basis. (default: {})

encoding The encoding to decode when performing validation (default: utf-8)

auto_reload_templates Whether to automatically reload changed templates. Set this to False in production for efficiency. If this is None, it takes the same value as debug. (default: None)

preferred_rendering_engines List of rendering engines in order of preference. (default: ['mako','genshi','jinja','kajiki'])

strict_engine_selection If set to true, TW2 will only select rendering engines from within your preferred_rendering_engines, otherwise, it will try the default list if it does not find a template within your preferred list. (default: True)

rendering_engine_lookup A dictionary of file extensions you expect to use for each type of template engine. Default:

```
{
        'mako':['mak', 'mako'],
        'genshi':['genshi', 'html'],
        'jinja':['jinja', 'html'],
        'kajiki':['kajiki', 'html'],
}
```

script_name A name to prepend to the url for all resource links (different from res_prefix, as it may be shared across and entire wsgi app. (default: '')

1.2 Widgets

Widgets are small self-contained components that can be reused across the same web page or across multiple pages.

A widget typically has a state (its value) a configuration (its params) a template that describes what should be displayed, one ore more resources (javascript or CSS) needed during display and might have some logic that has to be executed every time the widget is displayed.

1.2.1 Using Widgets

A typical widget will look like:

```
class MyWidget(tw2.core.Widget):
    template = "mypackage.widgets.templates.mywidget"
```

Which will look for a template named `mywidget.html` into the `templates` python package within the `widgets` package of the `mypackage` application. The extension expected for the file depends on the template engine used:

- mako: `.mako`

- kajiki: `.kajiki`
- jinja: `.jinja`
- genshi: `.genshi`

The template engine used to render the provided template depends on the `default_engine` option provided when configuring *tw2.core.middleware.TwMiddleware*.

In case you don't want to save the template into a separate file you can also set the `inline_engine_name` option to one of the template engines and provide the template as a string:

```
class HelloWidgetTemplate(tw2.core.Widget):
    inline_engine_name = "kajiki"
    template = """
        <i>Hello <span py:for="i in range(1, 4)">${i}, </span></i>
    """
```

Displaying a widget is as simple as calling the `Widget.display()`:

```
HelloWidgetTemplate.display()
```

Widget value

Each Widget has a special paramter, which is `value`. This parameter contains the current state of the widget. Value will usually be a single value or a dictionary containing multiple values (in case of *tw2.core.widgets.CompoundWidget*).

You can use the value to drive what the widget should show once displayed:

```
class HelloWidgetValue(tw2.core.Widget):
    inline_engine_name = "kajiki"
    template = """
        <i>Hello ${w.value}</i>
    """

>>> HelloWidgetValue.display(value='World')
Markup('<i>Hello World</i>')
```

`tw2.core.CompoundWidget` can contain multiple subwidgets (children) and their value is typically a `dict` with values for each one of the children:

```
class CompoundHello(tw2.core.CompoundWidget):
    inline_engine_name = "kajiki"
    template = """
        <div py:for="c in w.children">
            ${c.display()}
        </div>
    """

    name = HelloWidgetValue()
    greeter = tw2.core.Widget(inline_engine_name="kajiki",
                              template="<span>From ${w.value}</span>")

>>> CompoundHello(value=dict(name="Mario", greeter="Luigi")).display()
Markup('<div><span>Hello Mario</span></div><div><span>From Luigi</span></div>')
```

Children of a compound widget (like *Forms*) can be accessed both as a list iterating over `w.children` or by name using `w.children.childname`.

Parameters

Widgets might require more information than just their value to display, or might allow more complex kind of configurations. The options required to configure the widget are provided through *tw2.core.Param* objects that define which options each widget supports.

If you want your widget to be configurable, you can make available one or more options to your Widget and allow any user to set them as they wish:

```
class HelloWidgetParam(tw2.core.Widget):
    inline_engine_name = "kajiki"
    template = """
        <i>Hello ${w.name}</i>
    """

    name = tw2.core.Param(description="Name of the greeted entity")
```

The parameters can be provided any time by changing configuration of a widget:

```
>>> w = HelloWidgetParam(name="Peach")
>>> w.display()
Markup('<i>Hello Peach</i>')
>>> w2 = w(name="Toad")
>>> w2.display()
Markup('<i>Hello Toad</i>')
```

Or can be provided at `display` time itself:

```
>>> HelloWidgetParam.display(name="Peach")
Markup('<i>Hello Peach</i>')
```

Deferred Parameters

When a widget requires a parameter that is not available before display time. That parameter can be set to a *tw2.core.Deferred* object.

Deferred objects will accept any callable and before the widget is displayed the callable will be executed to fetch the actual value for the widget:

```
>>> singleselect = SingleSelectField(options=tw2.core.Deferred(lambda: [1,2,3]))
>>> singleselect.options
<Deferred: <Deferred>>
>>> singleselect.display()
Markup('<select ><option value=""></option>\n <option value="1">1</option>\n <option␣
↪value="2">2</option>\n <option value="3">3</option>\n</select>')
```

`Deferred` is typically used when loading data from the content of a database to ensure that the content is the one available at the time the widget is displayed and not the one that was available when the application started:

```
>>> userpicker = twf.SingleSelectField(
...     options=twc.Deferred(lambda: [(u.user_id, u.display_name) for u in model.
↪DBSession.query(model.User)])
... )
>>> userpicker.display()
Markup('<select ><option value=""></option>\n <option value="1">Example manager</
↪option>\n <option value="2">Example editor</option>\n</select>')
```

Builtin Widgets

The `tw2.core` packages comes with the basic buildin blocks needed to create your own custom widgets.

class tw2.core.widgets.**Widget**(***kw*)
Base class for all widgets.

classmethod req(***kw*)
Generate an instance of the widget.

Return the validated widget for this request if one exists.

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

classmethod get_link()
Get the URL to the controller . This is called at run time, not startup time, so we know the middleware if configured with the controller path. Note: this function is a temporary measure, a cleaner API for this is planned.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

iteritems()
An iterator which will provide the params of the widget in key, value pairs.

controller_path = <functools.partial object>

add_call = <functools.partial object>

display = <functools.partial object>

generate_output(*displays_on*)
Generate the actual output text for this widget.

By default this renders the widget's template. Subclasses can override this method for purely programmatic output.

displays_on The name of the template engine this widget is being displayed inside.

Use it like this:

```
class MyWidget(LeafWidget):
    def generate_output(self, displays_on):
        return "<span {0}>{1}</span>".format(self.attrs, self.text)
```

classmethod validate(*params*, *state=None*)
Validate form input. This should always be called on a class. It either returns the validated data, or raises a `ValidationError` exception.

class `tw2.core.widgets.`**LeafWidget**(***kw*)
A widget that has no children; this is the most common kind, e.g. form fields.

class `tw2.core.widgets.`**CompoundWidget**(***kw*)
A widget that has an arbitrary number of children, this is common for layout components, such as `tw2.forms.TableLayout`.

classmethod post_define()
Check children are valid; update them to have a link to the parent.

prepare()
Propagate the value for this widget to the children, based on their id.

class `tw2.core.widgets.`**RepeatingWidget**(***kw*)
A widget that has a single child, which is repeated an arbitrary number of times, such as `tw2.forms.GridLayout`.

classmethod post_define()
Check child is valid; update with link to parent.

prepare()
Propagate the value for this widget to the children, based on their index.

class `tw2.core.widgets.`**DisplayOnlyWidget**(***kw*)
A widget that has a single child. The parent widget is only used for display purposes; it does not affect value propagation or validation. This is used by widgets like `tw2.forms.FieldSet`.

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class `tw2.core.widgets.`**Page**(***kw*)
An HTML page. This widget includes a `request()` method that serves the page.

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

class tw2.core.**Param**(*description=Default*, *default=Default*, *request_local=Default*, *attribute=Default*, *view_name=Default*)

A parameter for a widget.

description A string to describe the parameter. When overriding a parameter description, the string can include `$$` to insert the previous description.

default The default value for the parameter. If no defalt is specified, the parameter is a required parameter. This can also be specified explicitly using tw.Required.

request_local Can the parameter be overriden on a per-request basis? (default: True)

attribute Should the parameter be automatically included as an attribute? (default: False)

view_name The name used for the attribute. This is useful for attributes like *class* which are reserved names in Python. If this is None, the name is used. (default: None)

The class takes care to record which arguments have been explictly specifed, even if to their default value. If a parameter from a base class is updated in a subclass, arguments that have been explicitly specified will override the base class.

class tw2.core.**Deferred**(*fn*)

This class is used as a wrapper around a parameter value. It takes a callable, which will be called every time the widget is displayed, with the returned value giving the parameter value.

1.2.2 Resources

ToscaWidgets comes with resources management for widgets too.

Some widgets might be complex enough that they need external resources to work properly. Typically those are CSS stylesheets or Javascript functions.

The need for those can be specified in the `Widget.resources` param, which is a list of resources the widget needs to work properly

The *tw2.core.middleware.TwMiddleware* takes care of serving all the resources needed by a widget through a `tw2.core.resources.ResourcesApp`. There is not need to setup such application manually, having a `TwMiddleware` in place will provide support for resources too.

When a widget is being prepared for display, all resources that it requires (as specified by `tw2.core.Widget.resources`) are registered into the current request and while the response page output goes through the middleware it will be edited to add the links (or content) of those resources as specified by their location.

Note: If a resource was already injected into the page during current request and another widget requires it, it won't be injected twice. ToscaWidgets is able to detect that it's the same resource (thanks to the resource `id`) and only inject that once.

To add resources to a widget simply specify them in `tw2.core.Widget.resources`:

```
class HelloWidgetClass(twc.Widget):
    inline_engine_name = "kajiki"
    template = """
        <i class="${w.css_class}">Hello ${w.name}</i>
    """

    name = twc.Param(description="Name of the greeted entity")
    css_class = twc.Param(description="Class used to display content", default="red")

    resources = [
        twc.CSSSource(src="""
            .red { color: red; }
            .green { color: green; }
            .blue { color: blue; }
        """)
    ]
```

Once the page where the widget is displayed is rendered, you will see that it begins with:

```
<!DOCTYPE html>
<html>
<head><style type="text/css">
            .red { color: red; }
            .green { color: green; }
            .blue { color: blue; }
        </style>
<meta content="width=device-width, initial-scale=1.0" name="viewport">
<meta charset="utf-8">
```

Which contains the CSS resource you specified as a dependency of your widget.

In case you are using a solution to package your resources into bundles like WebPack, WebAssets or similar, you might want to disable resources injection using `inject_resoures=False` option provided to *tw2.core.middleware.TwMiddleware* to avoid injecting resources that were already packed into your bundle.

Builtin Resource Types

class tw2.core.resources.**ResourceBundle**(***kw*)
Just a list of resources.

Use it as follows:

```
>>> jquery_ui = ResourceBundle(resources=[jquery_js, jquery_css])
>>> jquery_ui.inject()
```

class tw2.core.resources.**Resource**(***kw*)
A resource required by a widget being displayed.

`location` states where the resource should be injected into the page. Can be any of `head`, `headbottom`, `bodytop` or `bodybottom` or `None`.

class tw2.core.resources.**Link**(***kw*)
A link to a file.

The `link` parameter can be used to specify the explicit link to a URL.

If omitted, the link will be built to serve `filename` from `modname` as a resource coming from a python distribution.

classmethod guess_modname()
Try to guess my modname.

If I wasn't supplied any modname, take a guess by stepping back up the frame stack until I find something not in tw2.core

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

class tw2.core.resources.**DirLink**(***kw*)
A whole directory as a resource.

Unlike *JSLink* and *CSSLink*, this resource doesn't inject anything on the page.. but it does register all resources under the marked directory to be served by the middleware.

This is useful if you have a css file that pulls in a number of other static resources like icons and images.

class tw2.core.resources.**JSLink**(***kw*)
A JavaScript source file.

By default is injected in whatever default place is specified by the middleware.

class tw2.core.resources.**CSSLink**(***kw*)
A CSS style sheet.

By default it's injected at the top of the head node.

class tw2.core.resources.**JSSource**(***kw*)
Inline JavaScript source code.

By default is injected before the </body> is closed

class tw2.core.resources.**CSSSource**(***kw*)
Inline Cascading Style-Sheet code.

By default it's injected at the top of the head node.

1.3 Forms

ToscaWidgets provides all the widgets related to building HTML Forms in the `tw2.forms` package.

While `tw2.core` implements the foundation for declaring any kind of widget, the `tw2.forms` is specialised in widgets that are needed to create HTML forms.

1.3.1 Form

A form is usually created by declaring a subclass of a `tw2.forms.Form`. Within the form a `child` attribute that specifies the *Form Layout* (how the fields shoulb be arranged graphically) through a subclass of `tw2.forms.`

`Layout` and then within `child` all the fields of the form can be declared:

```
import tw2.core as twc
import tw2.forms as twf


class MovieForm(twf.Form):
    class child(twf.TableLayout):
        title = twf.TextField()
        director = twf.TextField(value='Default Director')
        genres = twf.SingleSelectField(options=['Action', 'Comedy', 'Romance', 'Sci-fi
↪'])

    action = '/save_movie'
```

The form must also provide an `action` attribute to specify where the form should be submitted.

Note: If you are going to use ToscaWidgets with TurboGears you probably want the action to be a `tg.lurl` to ensure that prefix of your application is retained.

Form Buttons

By default, each form comes with a **submit** button.

The submit button can be replaced by setting the form `submit` attribute:

```
class NameForm(twf.Form):
    class child(twf.TableLayout):
        name = twf.TextField()

    action = '/save_name'
    submit = twf.SubmitButton(value="Save Name")
```

Multiple buttons can also be provided for the form by setting the `buttons` attribute:

```
class NameForm(twf.Form):
    class child(twf.TableLayout):
        name = twf.TextField()

    action = '/save_name'
    buttons = [
        twf.SubmitButton(value="Save Name"),
        twf.ResetButton(),
        twf.Button(value="Say Hi", attrs=dict(onclick="alert('hi')"))
    ]
```

Dynamic Forms

Children can be added and removed dynamically from forms using the *`Widget.post_define()`* and *`Widget.prepare()`* methods.

For example to change children of a form based on an option, *`Widget.post_define()`* can be used:

```
class GrowingMovieForm(twf.Form):
    class child(twf.TableLayout):
        @classmethod
        def post_define(cls):
            if not cls.parent:
                return

            children = []

            for count in range(cls.parent.num_contacts):
                class person_fieldset(twf.TableFieldSet):
                    id = "person_%s" % count
                    label = "Person #%s" % count
                    name = twf.TextField(validator=twc.Validator(required=True))
                    surname = twf.TextField()

                children.append(person_fieldset(parent=cls))

            cls.children = children

    action = '/save_contacts'
    num_contacts = twc.Param(default=1)

fivefieldsform = GrowingMovieForm(num_contacts=5)
```

Note: Use the same `fivefieldsform` object for both display and validation. Trying to make a new `GrowingMovieForm` might not work even though `num_contacts` is always set to `5`.

This will not work btw if you need to take action at display time. In such case `Widget.prepare()` is needed, for example to have a text field that suggests the placeholder based on its original value:

```
class DynamicText(twf.Form):
    class child(twf.TableLayout):
        text = twf.TextField(placeholder="Put text here")

    action = "/save_movie"

    def prepare(self):
        super(DynamicText, self).prepare()

        if self.child.children.text.value:
            self.child.children.text.attrs = dict(
                self.child.children.text.attrs,
                placeholder="Put text here (was %s)" % self.child.children.text.value
            )
```

Note: `Widget.prepare()` is usually involved when setting a state that depends on the current request. For example current value of a field, or something else that is known only in current request. The resulting state of the widget is also only valid in current request, a different request might have nothing in common. Keep this in mind when using validation, as validation usually happens in a different request from the one that displayed the widget.

class `tw2.forms.widgets.`**Form**(***kw*)

A form, with a submit button. It's common to pass a TableLayout or ListLayout widget as the child.

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

submit
alias of `tw2.core.params.SubmitButton_s_s`

1.3.2 Validating Forms

When you submit a form, it will send its data to the endpoint you specified through the `action` parameter.

Before using it, you probably want to make sure that the data that was sent is correct and display back to the user error messages when it is not.

This can be done through *Validation* and thanks to the fact that Forms remember which form was just validated in the current request.

For each field in the form it is possible to specify a `validator=` parameter, which will be in charge of validation for that field:

```
class ValidatedForm(twf.Form):
    class child(twf.TableLayout):
        number = twf.TextField(placeholder="a number (1, 2, 3, 4)",
                               validator=twc.validation.IntValidator())
        required = twf.TextField(validator=twc.Required)
```

To validate the data submitted through this form you can use the `tw2.forms.widgets.Form.validate()` method.

If the validation passes, the method will return the validated data:

```
>>> ValidatedForm.validate({'numer': 5, 'required': 'hello'})
{'numer': 5, 'required': 'hello'}
```

If the validation fails, it will raise a *tw2.core.validation.ValidationError* exception:

```
Traceback (most recent call last):
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 106, in wrapper
```

(continues on next page)

(continued from previous page)

```
        d = fn(self, *args, **kw)
    File "/home/amol/wrk/tw2.core/tw2/core/widgets.py", line 718, in _validate
        raise vd.ValidationError('childerror', exception_validator)
tw2.core.validation.ValidationError
```

Such error can be trapped to get back the validated widget, the value that was being validated and the error message for each of its children:

```
>>> try:
...     ValidatedForm.validate({'numer': 'Hello', 'required': ''})
... except tw2.core.validation.ValidationError as e:
...     print(e.widget.child.value)
...     for c in e.widget.child.children:
...         print(c.compound_key, ':', c.error_msg)

{'numer': 'Hello', 'required': ''}
numer : Must be an integer
required : Enter a value
```

Also, trying to display back the form that was just validated, will print out the error message for each field:

```
>>> try:
...     ValidatedForm.validate({'numer': 'Hello', 'required': ''})
... except tw2.core.validation.ValidationError as e:
...     print(e.widget.display())

<form enctype="multipart/form-data" method="post">
    <span class="error"></span>

    <table>
    <tr class="odd error" id="numer:container">
        <th><label for="numer">Numer</label></th>
        <td>
            <input id="numer" name="numer" placeholder="a number (1, 2, 3, 4)" type=
↪"text" value="Hello"/>
            <span id="numer:error">Must be an integer</span>
        </td>
    </tr><tr class="even required error" id="required:container">
        <th><label for="required">Required</label></th>
        <td>
            <input id="required" name="required" type="text" value=""/>
            <span id="required:error">Enter a value</span>
        </td>
    </tr>
    </table>
    <input type="submit" value="Save"/>
</form>
```

For convenience, you can also recover the currently validated instance of the form anywhere in the code. Even far away from the exception that reported the validation error.

This can be helpful when you are isolating validation into a separate Aspect of your application and then you need to recover the form instance that includes the errors to display into your views.

To retrieve the currently validated widget, you can just use `tw2.core.widget.Widget.req()`:

```
>>> try:
...     ValidatedForm.validate({'numer': 'Hello', 'required': ''})
... except tw2.core.validation.ValidationError as e:
...     print(e.widget)
...     print(ValidatedForm.req())

<__main__.ValidatedForm object at 0x7f9432e5e080>
<__main__.ValidatedForm object at 0x7f9432e5e080>
```

As you can see `ValidatedForm.req()` returns the same exact instance that `e.widget` was. That's because when `Widget.req()` is used and there is a validated instance of that same exact widget in the current request, ToscaWidgets will assume you are trying to access the widget you just validated and will return that one instace of building a new instance.

If you want a new instance, you can still do `ValidatedForm().req()` instead of `ValidatedForm.req()`:

```
>>> try:
...     ValidatedForm.validate({'numer': 'Hello', 'required': ''})
... except tw2.core.validation.ValidationError as e:
...     print(e.widget)
...     print(ValidatedForm().req())

<__main__.ValidatedForm object at 0x7f9432e5e080>
<tw2.core.params.ValidatedForm_d object at 0x7f9432420940>
```

Keep in mind that this only keeps memory of the *last* widget that failed validation. So in case multiple widgets failed validation in the same request, you must used `tw2.core.validation.ValidationError.widget` to access each one of them.

1.3.3 Form Layout

A layout specifies how the fields of the form should be arranged.

This can be specified by having `Form.child` inherit from a specific layout class:

```
class NameForm(twf.Form):
    class child(twf.TableLayout):
        name = twf.TextField()
```

or:

```
class NameForm(twf.Form):
    class child(twf.ListLayout):
        name = twf.TextField()
```

Custom Layouts

A custom layout class can also be made to show the children however you want:

```
class Bootstrap3Layout(twf.BaseLayout):
    inline_engine_name = "kajiki"
    template = """
<div py:attrs="w.attrs">
    <div class="form-group" py:for="c in w.children_non_hidden" title="${w.hover_help
↪and c.help_text or None}" py:attrs="c.container_attrs" id="${c.compound_id}
↪:container">
```

(continues on next page)

(continued from previous page)

```
        <label for="${c.id}" py:if="c.label">$c.label</label>
        ${c.display(attrs={"class": "form-control"})}
        <span id="${c.compound_id}:error" class="error help-block" py:content="c.
↪error_msg"/>
    </div>
    <py:for each="c in w.children_hidden">${c.display()}</py:for>
    <div id="${w.compound_id}:error" py:content="w.error_msg"></div>
</div>"""

class BootstrapNameForm(twf.Form):
    class child(Bootstrap3Layout):
        name = twf.TextField()

    submit = twf.SubmitButton(css_class="btn btn-default")
```

Complex Layouts

In case of complex custom layouts, you can even specify the layout case by case in the form itself with each children in a specific position accessing the children using `w.children.child_name`:

```
class OddNameForm(twf.Form):
    class child(twf.BaseLayout):
        inline_engine_name = "kajiki"
        template = """
        <div py:attrs="w.attrs">
            <div py:with="c=w.children.name">
                ${c.display()}
                <span id="${c.compound_id}:error" py:content="c.error_msg"/>
            </div>
            <div py:with="c=w.children.surname">
                ${c.display()}
                <span id="${c.compound_id}:error" py:content="c.error_msg"/>
            </div>

            <py:for each="ch in w.children_hidden">${ch.display()}</py:for>
            <div id="${w.compound_id}:error" py:content="w.error_msg"></div>
        </div>
        """

        name = twf.TextField()
        surname = twf.TextField()
```

class tw2.forms.widgets.**BaseLayout**(***kw*)

The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

prepare()

Propagate the value for this widget to the children, based on their id.

class tw2.forms.widgets.**ListLayout**(***kw*)

Arrange widgets and labels in a list.

The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

class tw2.forms.widgets.**TableLayout**(**kw)
Arrange widgets and labels in a table.

The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

class tw2.forms.widgets.**GridLayout**(**kw)
Arrange labels and multiple rows of widgets in a grid.

child
alias of tw2.core.params.RowLayout_s

class tw2.forms.widgets.**RowLayout**(**kw)
Arrange widgets in a table row. This is normally only useful as a child to GridLayout.

prepare()
Propagate the value for this widget to the children, based on their id.

1.3.4 Bultin Form Fields

tw2.forms package comes with a bunch of builtin widgets that can help you build the most common kind of forms.

class tw2.forms.widgets.**FormField**(**kw)
Basic Form Widget from which each other field will inherit

name
Name of the field

required
If the field is required according to its validator (read-only)

class tw2.forms.widgets.**TextFieldMixin**(**kw)
Misc mixin class with attributes for textual input fields

maxlength = None
Maximum length of the field

placeholder = None
Placeholder text, until user writes something.

class tw2.forms.widgets.**InputField**(**kw)
A generic <input> field.

Generally you won't use this one, but will rely on one of its specialised subclasses like *TextField* or Checkbox.

type = None
Input type

value = None
Current value of the input

required = None
Add required attributed to the input.

autofocus = None
Add autofocus attributed to the input.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**PostlabeledInputField**(**kw)
Inherits InputField, but with a text label that follows the input field

text = None
Text to display in the label after the field.

text_attrs = {}
Attributes of the label displayed after to the field.

class tw2.forms.widgets.**TextField**(**kw)
A simple text field where to input a single line of text

size = None
Add size attribute to the HTML field.

class tw2.forms.widgets.**TextArea**(**kw)
A multiline text area

rows = None
Add a rows= attribute to the HTML textarea

cols = None
Add a cols= attribute to the HTML textarea

class tw2.forms.widgets.**CheckBox**(**kw)
A single checkbox.

Its value will be True or Folse if selected or not.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**RadioButton**(**kw)
A single radio button

checked = False
If the radio button is checked or not.

class tw2.forms.widgets.**PasswordField**(**kw)
A password field. This never displays a value passed into the widget, although it does redisplay entered values on validation failure. If no password is entered, this validates as EmptyField.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**FileValidator**(***kw*)
Validate a file upload field

extension Allowed extension for the file

class tw2.forms.widgets.**FileField**(***kw*)
A field for uploading files. The returned object has (at least) two properties of note:

- **filename:** the name of the uploaded file
- **value:** a bytestring of the contents of the uploaded file, suitable for being written to disk

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**HiddenField**(***kw*)
A hidden field.

Typically this is used to bring around in the form values that the user should not be able to modify or see. Like the ID of the entity edited by the form.

class tw2.forms.widgets.**IgnoredField**(***kw*)
A hidden field. The value is never included in validated data.

class tw2.forms.widgets.**LabelField**(***kw*)
A read-only label showing the value of a field. The value is stored in a hidden field, so it remains through validation failures. However, the value is never included in validated data.

class tw2.forms.widgets.**LinkField**(***kw*)
A dynamic link based on the value of a field. If either *link* or *text* contain a $, it is replaced with the field value. If the value is None, and there is no default, the entire link is hidden.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**Button**(***kw*)
Generic button. You can override the text using *value* and define a JavaScript action using *attrs['onclick']*.

class tw2.forms.widgets.**SubmitButton**(***kw*)
Button to submit a form.

class tw2.forms.widgets.**ResetButton**(**kw)
Button to clear the values in a form.

class tw2.forms.widgets.**ImageButton**(**kw)

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**HTML5PatternMixin**(**kw)
HTML5 mixin for input field regex pattern matching

See http://html5pattern.com/ for common patterns.

TODO: Configure server-side validator

class tw2.forms.widgets.**HTML5MinMaxMixin**(**kw)
HTML5 mixin for input field value limits

TODO: Configure server-side validator

class tw2.forms.widgets.**HTML5StepMixin**(**kw)
HTML5 mixin for input field step size

class tw2.forms.widgets.**HTML5NumberMixin**(**kw)
HTML5 mixin for number input fields

class tw2.forms.widgets.**EmailField**(**kw)
An email input field (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**UrlField**(**kw)
An url input field (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**NumberField**(**kw)
A number spinbox (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**RangeField**(**kw)
A number slider (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**SearchField**(**kw)
A search box (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**ColorField**(**kw)
A color picker field (HTML5 only).

Will fallback to a normal text input field on browser not supporting HTML5.

class tw2.forms.widgets.**SelectionField**(***kw*)
Base class for single and multiple selection fields.

The *options* parameter must be a list; it can take several formats:

- A list of values, e.g. `['', 'Red', 'Blue']`
- A list of (code, value) tuples, e.g. `[(0, ''), (1, 'Red'), (2, 'Blue')]`
- A mixed list of values and tuples. If the code is not specified, it defaults to the value. e.g. `['', (1, 'Red'), (2, 'Blue')]`
- Attributes can be specified for individual items, e.g. `[(1, 'Red', {'style':'background-color:red'})]`
- A list of groups, e.g. `[('group1', [(1, 'Red')]), ('group2', ['Pink', 'Yellow'])]`

Setting `value` before rendering will set the default displayed value on the page. In ToscaWidgets1, this was accomplished by setting `default`. That is no longer the case.

options = None
List of options to pick from in the form `[(id, text), (id, text), ...]`

prompt_text = None
Prompt to display when no option is selected. Set to `None` to disable this.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**MultipleSelectionField**(***kw*)

item_validator = None
Validator that has to be applied to each item.

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**SingleSelectField**(***kw*)
Specialised *SelectionField* to pick one element from a list of options.

class tw2.forms.widgets.**MultipleSelectField**(***kw*)
Specialised *SelectionField* to pick multiple elements from a list of options.

size = None
Number of options to show

class tw2.forms.widgets.**SelectionList**(***kw*)

class tw2.forms.widgets.**SeparatedSelectionTable**(***kw*)

class tw2.forms.widgets.**RadioButtonList**(**kw)

class tw2.forms.widgets.**CheckBoxList**(**kw)

class tw2.forms.widgets.**SelectionTable**(**kw)

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**VerticalSelectionTable**(**kw)

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**RadioButtonTable**(**kw)

class tw2.forms.widgets.**SeparatedRadioButtonTable**(**kw)

class tw2.forms.widgets.**VerticalRadioButtonTable**(**kw)

class tw2.forms.widgets.**CheckBoxTable**(**kw)

class tw2.forms.widgets.**SeparatedCheckBoxTable**(**kw)

class tw2.forms.widgets.**VerticalCheckBoxTable**(**kw)

class tw2.forms.widgets.**BaseLayout**(**kw)
The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

prepare()
Propagate the value for this widget to the children, based on their id.

class tw2.forms.widgets.**TableLayout**(**kw)
Arrange widgets and labels in a table.

The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

class tw2.forms.widgets.**ListLayout**(**kw)
Arrange widgets and labels in a list.

The following CSS classes are used, on the element containing both a child widget and its label.

odd / even On alternating rows. The first row is odd.

required If the field is a required field.

error If the field contains a validation error.

class tw2.forms.widgets.**RowLayout**(**kw)
Arrange widgets in a table row. This is normally only useful as a child to GridLayout.

prepare()
Propagate the value for this widget to the children, based on their id.

class tw2.forms.widgets.**StripBlanks**(**kw)

to_python(*value*, *state=None*)
Convert an external value to Python and validate it.

class tw2.forms.widgets.**GridLayout**(**kw)
Arrange labels and multiple rows of widgets in a grid.

child
alias of tw2.core.params.RowLayout_s

class tw2.forms.widgets.**Spacer**(**kw)
A blank widget, used to insert a blank row in a layout.

class tw2.forms.widgets.**Label**(**kw)
A textual label. This disables any label that would be displayed by a parent layout.

class tw2.forms.widgets.**Form**(**kw)
A form, with a submit button. It's common to pass a TableLayout or ListLayout widget as the child.

classmethod post_define()
This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

submit
alias of tw2.core.params.SubmitButton_s_s

prepare()
This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

class tw2.forms.widgets.**FieldSet**(**kw)
A field set. It's common to pass a TableLayout or ListLayout widget as the child.

class tw2.forms.widgets.**TableForm**(**kw)
Equivalent to a Form containing a TableLayout.

child
alias of tw2.core.params.TableLayout_s

submit
alias of tw2.core.params.SubmitButton_s_s_s

class tw2.forms.widgets.**ListForm**(**kw)
Equivalent to a Form containing a ListLayout.

child
alias of tw2.core.params.ListLayout_s

submit
alias of tw2.core.params.SubmitButton_s_s_s

class tw2.forms.widgets.**TableFieldSet**(**kw)
Equivalent to a FieldSet containing a TableLayout.

child
alias of tw2.core.params.TableLayout_s

class tw2.forms.widgets.**ListFieldSet**(**kw)
Equivalent to a FieldSet containing a ListLayout.

child
alias of tw2.core.params.ListLayout_s

class tw2.forms.widgets.**FormPage**(**kw)
A page that contains a form. The request method performs validation, redisplaying the form on errors. On success, it calls validated_request.

1.4 Validation

ToscaWidgets provides validation support for all the data that needs to be displayed into widgets or that has to come from submitted forms.

Setting a validator for a widget (or a form field) can be done through the tw2.core.Widget.validator param.

Validators are typically used in the context of forms and can be used both to tell ToscaWidgets how a python object should be displayed in HTML result:

```
>>> import tw2.core as twc
>>> import tw2.forms as twf
>>>
>>> w = twf.TextField(validator=twc.validation.DateValidator(format="%Y/%m/%d"))
>>> w.display(datetime.datetime.utcnow())
Markup('<input value="2019/04/04" type="text"/>')
```

Or to tell ToscaWidgets how the data coming from a submitted form should be converted into Python:

```
>>> class MyDateForm(twf.Form):
...   class child(twf.TableLayout):
...     date = twf.TextField(validator=twc.validation.DateValidator(format="%Y/%m/%d
↪"))
```

(continues on next page)

(continued from previous page)

```
...
>>> MyDateForm.validate({'date': '2019/5/3'})
{'date': datetime.date(2019, 5, 3)}
```

1.4.1 Validators

A validator is a class in charge of two major concerns:

- Converting data from the web to python and back to the web
- Validating that the data is what you expected.

Both those step are performed through two methods:

tw2.core.validation.Validator.to_python() which is in charge of converting data from the web to Python:

```
>>> validator = twc.validation.DateValidator(required=True, format="%Y/%m/%d")
>>> validator.to_python('2019/10/3')
datetime.date(2019, 10, 3)
```

and *tw2.core.validation.Validator.from_python()* which is in charge of converting data from Python to be displahyed on a web page:

```
>>> validator.from_python(datetime.datetime.utcnow())
"2019/04/04"
```

When converting data *to python* (so for data submitted from the web to your web application) the validator does three steps:

1. Ensures that the data is not empty through `tw2.core.validation.Validator._is_empty()` if `required=True` was provided
2. Converts data to Python through `tw2.core.validation.Validator._convert_to_python()`
3. Validates that the converted data matches what you expected through `tw2.core.validation.Validator._validate_python()`

All those three methods (`is_empty`, `_convert_to_python` and `_validate_python`) can be specialised in subclasses to implement your own validators.

For example the *tw2.core.validation.IntValidator* takes care of converting the incoming text to interger-ers:

```
>>> twc.validation.IntValidator().to_python("5")
5
```

but also takes care of validating that it's within an expected range:

```
>>> twc.validation.IntValidator(min=1).to_python("0")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 236, in to_python
    self._validate_python(value, state)
File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 376, in _validate_python
    raise ValidationError('toosmall', self)
tw2.core.validation.ValidationError: Must be at least 1
```

1.4.2 Custom Validators

You can write your own validators by subclassing *tw2.core.validation.Validator*.

Those should *at least* implement the custom conversion part, to tell toscawidgets how to convert the incoming data to the type you expect:

```
class TwoNumbersValidator(twc.validation.Validator):
    def _convert_to_python(self, value, state=None):
        try:
            return [int(v) for v in value.split(',')]
        except ValueError:
            raise twc.validation.ValidationError("Must be integers", self)
        except Exception:
            raise twc.validation.ValidationError("corrupt", self)
```

This is already enough to be able to convert the incoming data to a list of numbers:

```
>>> TwoNumbersValidator().to_python("5,3")
[5, 3]
```

and to detect that numbers were actually submitted:

```
>>> TwoNumbersValidator().to_python("5, allo")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 235, in to_python
        value = self._convert_to_python(value, state)
    File "<stdin>", line 6, in _convert_to_python
tw2.core.validation.ValidationError: Must be integers
```

and to detect malformed inputs:

```
>>> TwoNumbersValidator().to_python(datetime.datetime.utcnow())
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 235, in to_python
        value = self._convert_to_python(value, state)
    File "<stdin>", line 8, in _convert_to_python
tw2.core.validation.ValidationError: Form submission received corrupted; please try␣
↪again
```

But it doesn't perform validation on the converted data. It doesn't ensure that what was provided are really two numbers:

```
>>> TwoNumbersValidator().to_python("5")
[5]
```

To do so we need to implement the validation part of the validator, which is done through `_validate_python`:

```
class TwoNumbersValidator(twc.validation.Validator):
    def _convert_to_python(self, value, state=None):
        try:
            return [int(v) for v in value.split(',')]
        except ValueError:
            raise twc.validation.ValidationError("Must be integers", self)
        except Exception:
            raise twc.validation.ValidationError("corrupt", self)
```

(continues on next page)

(continued from previous page)

```
    def _validate_python(self, value, state=None):
        if len(value) != 2:
            raise twc.validation.ValidationError("Must be two numbers", self)
```

To finally provide coverage for the case where a single number (or more than two numbers) were provided:

```
>>> TwoNumbersValidator().to_python("5")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 236, in to_python
        self._validate_python(value, state)
    File "<stdin>", line 11, in _validate_python
tw2.core.validation.ValidationError: Must be two numbers
```

You will notice by the way, that empty values won't cause validation errors:

```
>>> v = TwoNumbersValidator().to_python("")
```

Those will be converted to `None`:

```
>>> print(v)
None
```

Because by default validators have `required=False` which means that missing values are perfectly fine.

If you want to prevent that behaviour you can provide `required=True` to the validator:

```
>>> TwoNumbersValidator(required=True).to_python("")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 231, in to_python
        raise ValidationError('required', self)
tw2.core.validation.ValidationError: Enter a value
```

1.4.3 Internationalization

Validator error messages can be translated through the usage of the `msgs` lookup dictionary.

The `msgs` dictionary is a map from keywords to translated strings and it's used by ToscaWidgets to know which message to show to users:

```
from tw2.core.i18n import tw2_translation_string

class FloatValidator(twc.Validator):
    msgs = {
        "notfloat": tw2_translation_string("Not a floating point number")
    }

    def _convert_to_python(self, value, state):
        try:
            return float(value)
        except ValueError:
            raise twc.validation.ValidationError("notfloat", self)
```

You will see that when validation fails, the `"notfloat"` key is looked up into `msgs` to find the proper message:

```
>>> FloatValidator().to_python("Hello")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 235, in to_python
        value = self._convert_to_python(value, state)
    File "<stdin>", line 9, in _convert_to_python
tw2.core.validation.ValidationError: Not a floating point number
```

The entry in `msgs` is then wrapped in a `tw2.core.i18n.tw2_translation_string()` call to ensure it gets translated using the translated that was configured in *tw2.core.middleware.TwMiddleware* options.

Note: `tw2.core.i18n.tw2_translation_string()` is also available as `tw2.core.i18n._` so that frameworks that automate translatable strings collection like Babel can more easily find strings that need translation in ToscaWidgets validators.

The other purpose of `msgs` is to allow users of your validator to customise their error messages:

```
>>> FloatValidator(msgs={"notfloat": "Ahah! Gotcha!"}).to_python("Hello")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    File "/home/amol/wrk/tw2.core/tw2/core/validation.py", line 235, in to_python
        value = self._convert_to_python(value, state)
    File "<stdin>", line 9, in _convert_to_python
tw2.core.validation.ValidationError: Ahah! Gotcha!
```

In such case (when `msgs` are customised) translation of the messages is up to the user customising them. Who might want to ensure the new provided messages are still wrapped in `tw2.core.i18n.tw2_translation_string()`.

1.4.4 Builtin Validators

exception `tw2.core.validation.`**ValidationError**(*msg*, *validator=None*, *widget=None*)
Invalid data was encountered during validation.

The constructor can be passed a short message name, which is looked up in a validator's `msgs` dictionary. Any values in this, like `$val` are substituted with that attribute from the validator. An explicit validator instance can be passed to the constructor, or this defaults to *Validator* otherwise.

message
Added for backwards compatibility. Synonymous with *msg*.

`tw2.core.validation.`**catch**
alias of *tw2.core.validation.ValidationError*

`tw2.core.validation.`**unflatten_params**(*params*)
This performs the first stage of validation. It takes a dictionary where some keys will be compound names, such as "form:subform:field" and converts this into a nested dict/list structure. It also performs unicode decoding, with the encoding specified in the middleware config.

class `tw2.core.validation.`**Validator**(***kw*)
Base class for validators

required Whether empty values are forbidden in this field. (default: False)

strip Whether to strip leading and trailing space from the input, before any other validation. (default: True)

To convert and validate a value to Python, use the *to_python()* method, to convert back from Python, use *from_python()*.

To create your own validators, sublass this class, and override any of `_validate_python()`, `_convert_to_python()`, or `_convert_from_python()`. Note that these methods are not meant to be used externally. All of them may raise ValidationErrors.

to_python(*value*, *state=None*)
Convert an external value to Python and validate it.

from_python(*value*, *state=None*)
Convert from a Python object to an external value.

validate_python(*value*, *state=None*)
"Deprecated, use `_validate_python()` instead.

This method has been renamed in FormEncode 1.3 and ToscaWidgets 2.2 in order to clarify that is an internal method that is meant to be overridden only; you must call meth:*to_python* to validate values.

class tw2.core.validation.**BlankValidator**(***kw*)
Always returns EmptyField. This is the default for hidden fields, so their values are not included in validated data.

to_python(*value*, *state=None*)
Convert an external value to Python and validate it.

class tw2.core.validation.**LengthValidator**(***kw*)
Confirm a value is of a suitable length. Usually you'll use *StringLengthValidator* or *ListLengthValidator* instead.

min Minimum length (default: None)

max Maximum length (default: None)

class tw2.core.validation.**StringLengthValidator**(***kw*)
Check a string is a suitable length. The only difference to LengthValidator is that the messages are worded differently.

class tw2.core.validation.**ListLengthValidator**(***kw*)
Check a list is a suitable length. The only difference to LengthValidator is that the messages are worded differently.

class tw2.core.validation.**RangeValidator**(***kw*)
Confirm a value is within an appropriate range. This is not usually used directly, but other validators are derived from this.

min Minimum value (default: None)

max Maximum value (default: None)

class tw2.core.validation.**IntValidator**(***kw*)
Confirm the value is an integer. This is derived from *RangeValidator* so *min* and *max* can be specified.

class tw2.core.validation.**BoolValidator**(***kw*)
Convert a value to a boolean. This is particularly intended to handle check boxes.

class tw2.core.validation.**OneOfValidator**(***kw*)
Confirm the value is one of a list of acceptable values. This is useful for confirming that select fields have not been tampered with by a user.

values Acceptable values

class tw2.core.validation.**DateTimeValidator**(**kw)
Confirm the value is a valid date and time. This is derived from *RangeValidator* so *min* and *max* can be specified.

format The expected date/time format. The format must be specified using the same syntax as the Python strftime function.

class tw2.core.validation.**DateValidator**(**kw)
Confirm the value is a valid date.

Just like *DateTimeValidator*, but without the time component.

class tw2.core.validation.**RegexValidator**(**kw)
Confirm the value matches a regular expression.

regex A Python regular expression object, generated like `re.compile('^\w+$')`

class tw2.core.validation.**EmailValidator**(**kw)
Confirm the value is a valid email address.

class tw2.core.validation.**UrlValidator**(**kw)
Confirm the value is a valid URL.

class tw2.core.validation.**IpAddressValidator**(**kw)
Confirm the value is a valid IP4 address, or network block.

allow_netblock Allow the IP address to include a network block (default: False)

require_netblock Require the IP address to include a network block (default: False)

class tw2.core.validation.**UUIDValidator**(**kw)
Confirm the value is a valid uuid and convert to uuid.UUID.

class tw2.core.validation.**MatchValidator**(*other_field*, *pass_on_invalid=False*, **kw)
Confirm a field matches another field

other_field Name of the sibling field this must match

pass_on_invalid Pass validation if sibling field is Invalid

class tw2.core.validation.**CompoundValidator**(*args, **kw)
Base class for compound validators.

Child classes *Any* and *All* take validators as arguments and use them to validate "value". In case the validation fails, they raise a ValidationError with a compound message.

```
>>> v = All(StringLengthValidator(max=50), EmailValidator, required=True)
```

class tw2.core.validation.**All**(*args, **kw)
Confirm all validators passed as arguments are valid.

class tw2.core.validation.**Any**(*args, **kw)
Confirm at least one of the validators passed as arguments is valid.

1.5 Javascript Integration

ToscaWidget2 was designed to work with any Javascript framework and integrate Python and Javascript as well as possible, leading to a seamless development experience when you have to provide Javascript callbacks or functions to your Python declared widgets and forms.

1.5.1 Javascript on Display

Frequently, when a widget is displayed, you might have to run some form of initialization in JavaScript to attach dynamic behaviours to it.

This can be done by using `tw2.core.Widget.add_call()` to register a *tw2.core.js.js_function* that should be called.

A simple example might be to display a widget that shows an "Hello World" alert every time it renders:

```
import tw2.core as twc


class HelloJSWidget(twc.Widget):
    message = twc.Param(description="Message to display")
    template = "<div></div>"
    inline_engine_name = "kajiki"

    def prepare(self):
        super(HelloJSWidget, self).prepare()

        alert = twc.js.js_function("alert")
        if self.message:
            self.add_call(alert(self.message))
```

As you can see we define a new *tw2.core.js.js_function* named "alert" and we assign it to the python "alert" variable. If a message is provided, `tw2.core.Widget.add_call()` is used to register `alert(self.message)` as what should be called every time the widget is rendered.

Displaying the widget in a web page:

```
HelloJSWidget(message="Hello World").display()
```

will in fact open an alert box with the "Hello World" text.

But you are not constrained to use pre-existing Javascript functions (like `alert`), you can in fact declare your own function (or use one that was imported from a *tw2.core.resources.JSLink*).

For example we can change the previous widget to accept only the name of the person to greet instead of the whole message and display `"Hello SOMEONE"` always:

```
class HelloJSWidget(twc.Widget):
    greeted = twc.Param(description="Who to greet")
    template = "<div></div>"
    inline_engine_name = "kajiki"

    def prepare(self):
        super(HelloJSWidget, self).prepare()
        sayhello = twc.js.js_function('(function(target){ alert("Hello " + target); })
↪')

        if self.greeted:
            self.add_call(sayhello(self.greeted))
```

As you could see, instead of having out *tw2.core.js.js_function* point to an already existing one, we declared a new one that accepts a `target` argument and displays an alert to greet the target.

The target of the greet message is then set in `HelloJSWidget.prepare` through the `greeted` param.

Displaying such widget will lead as expected to show an alert box with "Hello" plus the name of the greeted person:

```
HelloJSWidget(greeted="Mario").display()
```

It's also for example possible to run javascript that will target the widget itself by using the `Widget.id` and `Widget.compound_id` properties to know the unique identifier of the widget in the dom.

Using such tactic we could rewrite the previous widget to always read the greeted person from the content of the `div` instead of passing it as an argument:

```
class HelloJSWidget(twc.Widget):
    greeted = twc.Param(description="Who to greet")
    template = """<div id="$w.id">${w.greeted}</div>"""
    inline_engine_name = "kajiki"

    def prepare(self):
        super(HelloJSWidget, self).prepare()
        sayhello = twc.js.js_function('(function(widget_id){ var target = document.
↪getElementById(widget_id).innerText; alert("Hello " + target); })')
        self.add_call(sayhello(self.id))
```

Note: `compound_id` is safer to use, as it avoids collions when widgets with the same id are used within different parents. But is mostly only available in form fields. On plain widgets you might need to use `id` itself.

1.5.2 Javascript Callbacks

While being able to call javascript every time the widget is displayed is essential to be able to attach advanced javascript behaviours to widgets, sometimes you will need to trigger Javascript callbacks when something happens on the widgets.

This can usually be done with *`tw2.core.js.js_callback`* to declare the javascript callback you care about.

A possible example is to run some javascript when the selected option is changed in a single select field:

```
alertpicker = twf.SingleSelectField(
    attrs={'onchange': twc.js.js_callback('alert("changed!")')},
    options=[(1, 'First'), (2, 'Second')]
)
```

1.5.3 Builtin Javascript Helpers

Python-JS interface to dynamically create JS function calls from your widgets.

This moudle doesn't aim to serve as a Python-JS "translator". You should code your client-side code in JavaScript and make it available in static files which you include as JSLinks or inline using JSSources. This module is only intended as a "bridge" or interface between Python and JavaScript so JS function **calls** can be generated programatically.

class tw2.core.js.**js_callback**(*cb*, **args*)

A js function that can be passed as a callback to be called by another JS function

Examples:

```
>>> str(js_callback("update_div"))
'update_div'
```

```
>>> str(js_callback("function (event) { .... }"))
'function (event) { .... }'
```

Can also create callbacks for deferred js calls

```
>>> str(js_callback(js_function('foo')(1,2,3)))
'function(){foo(1, 2, 3)}'
```

Or equivalently

```
>>> str(js_callback(js_function('foo'), 1,2,3))
'function(){foo(1, 2, 3)}'
```

A more realistic example

```
>>> jQuery = js_function('jQuery')
>>> my_cb = js_callback('function() { alert(this.text)}')
>>> on_doc_load = jQuery('#foo').bind('click', my_cb)
>>> call = jQuery(js_callback(on_doc_load))
>>> print call
jQuery(function(){jQuery(\"#foo\").bind(
    \"click\", function() { alert(this.text)})})
```

class tw2.core.js.**js_function**(*name*)

A JS function that can be "called" from python and added to a widget by widget.add_call() so it get's called every time the widget is rendered.

Used to create a callable object that can be called from your widgets to trigger actions in the browser. It's used primarily to initialize JS code programatically. Calls can be chained and parameters are automatically json-encoded into something JavaScript undersrtands. Example:

```
>>> jQuery = js_function('jQuery')
>>> call = jQuery('#foo').datePicker({'option1': 'value1'})
>>> str(call)
'jQuery("#foo").datePicker({"option1": "value1"})'
```

Calls are added to the widget call stack with the `add_call` method.

If made at Widget initialization those calls will be placed in the template for every request that renders the widget:

```
>>> import tw2.core as twc
>>> class SomeWidget(twc.Widget): ...
pickerOptions = twc.Param(default={})
>>> SomeWidget.add_call( ...
        jQuery('#%s' % SomeWidget.id).datePicker(SomeWidget.pickerOptions)
        ... )
```

More likely, we will want to dynamically make calls on every request. Here we will call add_calls inside the `prepare` method:

```
>>> class SomeWidget(Widget):
...     pickerOptions = twc.Param(default={})
...     def prepare(self):
...         super(SomeWidget, self).prepare()
...         self.add_call(
...             jQuery('#%s' % d.id).datePicker(d.pickerOptions)
...         )
```

This would allow to pass different options to the datePicker on every display.

JS calls are rendered by the same mechanisms that render required css and js for a widget and places those calls at bodybottom so DOM elements which we might target are available.

Examples:

```
>>> call = js_function('jQuery')("a .async")
>>> str(call)
'jQuery("a .async")'
```

js_function calls can be chained:

```
>>> call = js_function('jQuery')("a .async").foo().bar()
>>> str(call)
'jQuery("a .async").foo().bar()'
```

class tw2.core.js.**js_symbol**(*name=None*, *src=None*)

An unquoted js symbol like `document` or `window`

1.6 Design

1.6.1 Widget Overview

The main purpose of a widget is to display a functional control within an HTML page. A widget has a template to generate its own HTML, and a set of parameters that control how it will be displayed. It can also reference resources - JavaScript or CSS files that support the widget.

When defining Widgets, some parameters with be static - they will stay constant for the whole lifetime of the application. Some parameters are dynamic - they may change for every request. To ensure thread-safety, a separate widget instance is created for every request, and dynamic parameters are only set on an instance. Static parameters are set by subclassing a widget. For example:

```
# Initialisation
class MyWidget(Widget):
    id = 'myid'

# In a request
my_widget = MyWidget.req()
my_widget.value = 'my value'
```

To make initialisation more concise, the `__new__` method on `Widget` is overriden, so it creates subclasses, rather than instances. The following code is equivalent to that above:

```
# Initialisation
MyWidget = Widget(id='myid')
```

In practice, you will rarely need to explictly create an instance, using `req()`. If the `display` or `validate` methods are called on a Widget class, they automatically create an instance. For example, the following are equivalent:

```
# Explicit creation
my_widget = MyWidget.req()
my_widget.value = 'my value'
my_widget.display()
```

(continues on next page)

(continued from previous page)

```
# Implicit creation
MyWidget.display(value='my value')
```

Parameters

The parameters are how the user of the widget controls its display and behaviour. Parameters exist primarily for documentation purposes, although they do have some run-time effects. When creating widgets, it's important to decide on a convenient set of parameters for the user of the widget, and to document these.

A parameter definition looks like this:

```
import tw2.core as twc
class MyTextField(twc.Widget):
    size = twc.Param('The size of the field', default=30)
    validator = twc.LengthValidator(max=30)
    highlight = twc.Variable('Region to highlight')
```

In this case, `TextField` gets all the parameters of its base class, `tw2.core.widget` and defines a new parameter - `size`. A widget can also override parameter in its base class, either with another *tw2.core.Param* instance, or a new default value.

class tw2.core.**Param**(*description=Default*, *default=Default*, *request_local=Default*, *attribute=Default*, *view_name=Default*)

A parameter for a widget.

description A string to describe the parameter. When overriding a parameter description, the string can include `$$` to insert the previous description.

default The default value for the parameter. If no defalt is specified, the parameter is a required parameter. This can also be specified explicitly using tw.Required.

request_local Can the parameter be overriden on a per-request basis? (default: True)

attribute Should the parameter be automatically included as an attribute? (default: False)

view_name The name used for the attribute. This is useful for attributes like *class* which are reserved names in Python. If this is None, the name is used. (default: None)

The class takes care to record which arguments have been explictly specifed, even if to their default value. If a parameter from a base class is updated in a subclass, arguments that have been explicitly specified will override the base class.

class tw2.core.**Variable**(*description=Default*, ***kw*)

A variable - a parameter that is passed from the widget to the template, but cannot be controlled by the user. These do not appear in the concise documentation for the widget.

class tw2.core.**ChildParam**(*description=Default*, *default=Default*, *request_local=Default*, *attribute=Default*, *view_name=Default*)

A parameter that applies to children of this widget

This is useful for situations such as a layout widget, which adds a `label` parameter to each of its children. When a Widget subclass is defined with a parent, the widget picks up the defaults for any child parameters from the parent.

class tw2.core.**ChildVariable**(*description=Default*, ***kw*)

A variable that applies to children of this widget

Code Hooks

Subclasses of Widget can override the following methods. It is not recommended to override any other methods, e.g. display, validate, __init__.

classmethod Widget.**post_define**()

This is a class method, that is called when a subclass of this Widget is created. Process static configuration here. Use it like this:

```
class MyWidget(LeafWidget):
    @classmethod
    def post_define(cls):
        id = getattr(cls,  'id', None)
        if id and not id.startswith('my'):
            raise pm.ParameterError("id must start with 'my'")
```

post_define should always cope with missing data - the class may be an abstract class. There is no need to call super(), the metaclass will do this automatically.

Widget.**prepare**()

This is an instance method, that is called just before the Widget is displayed. Process request-local configuration here. For efficiency, widgets should do as little work as possible here. Use it like this:

```
class MyWidget(Widget):
    def prepare(self):
        super(MyWidget, self).prepare()
        self.value = 'My: ' + str(self.value)
```

Widget.**generate_output**(*displays_on*)

Generate the actual output text for this widget.

By default this renders the widget's template. Subclasses can override this method for purely programmatic output.

displays_on The name of the template engine this widget is being displayed inside.

Use it like this:

```
class MyWidget(LeafWidget):
    def generate_output(self, displays_on):
        return "<span {0}>{1}</span>".format(self.attrs, self.text)
```

Mutable Members

If a widget's `prepare()` method modifies a mutable member on the widget, it must take care not to modify a class member, as this is not thread safe. In general, the code should call `self.safe_modify(member_name)`, which detects class members and creates a copy on the instance. Users of widgets should be aware that if a mutable is set on an instance, the widget may modify this. The most common case of a mutable member is `attrs`. While this arrangement is thread-safe and reasonably simple, copying may be bad for performance. In some cases, widgets may deliberately decide not to call `safe_modify()`, if the implications of this are understood.

1.6.2 Widget Hierarchy

Widgets can be arranged in a hierarchy. This is useful for applications like layouts, where the layout will be a parent widget and fields will be children of this. There are four roles a widget can take in the hierarchy, depending on the base class used:

class tw2.core.**Widget**(**kw)
Base class for all widgets.

class tw2.core.**CompoundWidget**(**kw)
A widget that has an arbitrary number of children, this is common for layout components, such as tw2.forms.TableLayout.

class tw2.core.**RepeatingWidget**(**kw)
A widget that has a single child, which is repeated an arbitrary number of times, such as tw2.forms.GridLayout.

class tw2.core.**DisplayOnlyWidget**(**kw)
A widget that has a single child. The parent widget is only used for display purposes; it does not affect value propagation or validation. This is used by widgets like tw2.forms.FieldSet.

Value Propagation

An important feature of the hierarchy is value propagation. When the value is set for a compound or repeating widget, this causes the value to be set for the child widgets. In general, a leaf widget takes a scalar type as a value, a compound widget takes a dict or an object, and a repeating widget takes a list.

The hierarchy also affects the generation of compound ids, and validation.

Identifier

In general, a widget needs to have an identifier. Without an id, it cannot participate in value propagation or validation, and it does not get an HTML id attribute. There are some exceptions to this:

- Some widgets do not need an id (e.g. Label, Spacer) and provide a default id of None.
- The child of a RepeatingWidget must not have an id.
- An id can be specified either on a DisplayOnlyWidget, or it's child, but not both. The widget that does not have the id specified automatically picks it up from the other.

Compound IDs are formed by joining the widget's id with those of its ancestors. These are used in two situations:

- For the HTML id attribute, and also the name attribute for form fields
- For the URL path a controller widget is registered at

The separator is a colon (:), resulting in compound ids like "form:sub_form:field". **Note** this causes issues with CSS and will be changed shortly, and made configurable.

In generel, the id on a DisplayOnlyWidget is not included in the compound id. However, when generating the compound id for a DisplayOnlyWidget, the id is included. In addition *id_suffix* is appended, to avoid generating duplicate IDs. The *id_suffix* is not appended for URL paths, to keep the paths short. There is a risk of duplicate IDs, but this is not expected to be a problem in practice.

For children of a RepeatingWidget, the repetition is used instead of the id, for generating the compound HTML id. For the URL path, the element is skipped entirely.

Deep Children

This is a feature that helps have a form layout that doesn't exactly match the database layout. For example, we might have a sinlge database table, with fields like title, customer, start_date, end_date. We want to display this in a Form that's broken up into two FieldSets. Without deep children, the FieldSets would have to have ids, and field makes would be dotted, like info.title. The deep children feature lets us set the id to None:

```
class MyForm(twf.Form):
    class child(twc.CompoundWidget):
        class info(twf.TableFieldSet):
            id = None
```

(continues on next page)

(continued from previous page)

```
            title = twf.TextField()
            customer = twf.TextField()
        class dates(twf.TableFieldSet):
            id = None
            start_date = twf.TextField()
            end_date = twf.TextField()
```

When a value like `{'title': 'my title'}` is passed to MyForm, this will propagate correctly.

1.6.3 Template

Every widget can have a template. Toscawidgets has some template-language hooks which currently support Genshi, Mako, Jinja2, Kajiki, and Chameleon.

At one point, ToscaWidgets2 aimed to support any templating engine that supported the `buffet` interface, (an initiative by the TurboGears project to create a standard interface for template libraries). In practice though, there are more differences between template engines than the buffet interface standardises so this approach has been dropped.

The `template` parameter takes the form `engine_name:template_path`. The `engine_name` is the name that the template engine defines in the `python.templating.engines` entry point, e.g. `genshi`, `mako`, or `jinja`. The `template_path` is a string the engine can use to locate the template; usually this is dot-notation that mimics the semantics of Python's import statement, e.g. `myapp.templates.mytemplate`. Templates also allow specifications like `./template.html` which is beneficial for simple applications.

It is also possible to allow your widget to utilize multiple templates, or to have TW2 support any template language you provide a template for. To do this, simply leave the name of the template engine off of the template parameter, and TW2 will select the appropriate template, based on specifications in the TW2 middleware.

For instance, you might have a form.mak and a form.html template (mako and genshi). TW2 will render the mako template if mako is listed ahead of genshi in the middleware config's `preferred_rendering_engines`. See the documentation regarding *Enabling ToscaWidgets* for more information on how to set up your middleware for desired output.

1.6.4 Non-template Output

Instead of using a template, a widget can also override the `generate_output` method. This function generates the HTML output for a widget; by default, it renders the widget's template as described in the previous section, but can be overridden by any function that returns a string of HTML.

1.6.5 Resources

Widgets often need to access resources, such as JavaScript or CSS files. A key feature of widgets is the ability to automatically serve such resources, and insert links into appropriate sections of the page, e.g. `<HEAD>`. There are several parts to this:

- Widgets can define resources they use, using the `resources` parameter.
- When a resource is defined, it is registered with the resource server.
- When a Widget is displayed, it registers resources in request-local storage.
- The resource injection middleware detects resources in request-local storage, and rewrites the generated page to include appropriate links.
- The resource server middleware serves static files used by widgets

- Widgets can also access resources at display time, e.g. to get links
- Resources can themselves declare dependency on other resources, e.g. jquery-ui.js depends on jquery.js and must be included on the page subsequently.

Defining Resources

To define a resource, just add a `tw2.core.Resource` subclass to the widget's `resources` parameter. It is also possible to append to `resources` from within the `prepare()` method. The following resource types are available:

See *Resources*

Resources are widgets, but follow a slightly different lifecycle. Resource subclasses are passed into the `resources` parameter. An instance is created for each request, but this is only done at the time of the parent Widget's `display()` method. This gives widgets a chance to add dynamic resources in their `prepare()` method.

Using Your Own Resources

Resources that are defined by pre-existing tw2 packages can be altered globally. For instance, say that you want to use your own patched version of jquery and you want all tw2 packages that require jquery to use your version, and not the one already packaged up in `tw2.jquery`. The following code will alter `jquery_js` in not just the local scope, but also in all other modules that use it (including `tw2.jqplugins.ui`):

```
import tw2.jquery
tw2.jquery.jquery_js.link = "/path/to/my/patched/jquery.js"
```

Deploying Resources

If running behind mod_wsgi, tw2 resource provisioning will typically fail. Resources are only served when they are registered with the request-local thread, and resources are only registered when their dependant widget is displayed in a request. An initial page request may make available resource `A`, but the subsequent request to actually retrieve resource `A` will not have that resource registered.

To solve this problem (and to introduce a speed-up for production deployment), Toscawidgets2 provides an `archive_tw2_resources` distutils command:

```
$ python setup.py archive_tw2_resources \
    --distributions=myapplication \
    --output=/var/www/myapplication
```

1.6.6 Declarative Instantiation

Instantiating compound widgets can result in less-than-beautiful code. To help alleviate this, widgets can be defined declaratively, and this is the recommended approach. A definition looks like this:

```
class MovieForm(twf.TableForm):
    id = twf.HiddenField()
    year = twf.TextField()
    desc = twf.TextArea(rows=5)
```

Any class members that are subclasses of Widget become children. All the children get their `id` from the name of the member variable. Note: it is important that all children are defined like `id = twf.HiddenField()` and not `id = twf.HiddenField`. Otherwise, the order of the children will not be preserved.

It is possible to define children that have the same name as parameters, using this syntax. However, doing so does prevent a widget overriding a parameter, and defining a child with the same name. If you need to do this, you must use a throwaway name for the member variable, and specify the id explicitly, e.g.:

```
class MovieForm(twf.TableForm):
    resources = [my_resource]
    id = twf.HiddenField()
    noname = twf.TextArea(id='resources')
```

Nesting and Inheritence

Nested declarative definitions can be used, like this:

```
class MyForm(twf.Form):
    class child(twf.TableLayout):
        b = twf.TextArea()
        x = twf.Label(text='this is a test')
        c = twf.TextField()
```

Inheritence is supported - a subclass gets the children from the base class, plus any defined on the subclass. If there's a name clash, the subclass takes priority. Multiple inheritence resolves name clashes in a similar way. For example:

```
class MyFields(twc.CompoundWidget):
    b = twf.TextArea()
    x = twf.Label(text='this is a test')
    c = twf.TextField()

class TableFields(MyFields, twf.TableLayout):
    pass

class ListFields(MyFields, twf.ListLayout):
    b = twf.TextField()
```

Proxying children

Without this feature, double nesting of classes is often necessary, e.g.:

```
class MyForm(twf.Form):
    class child(twf.TableLayout):
        b = twf.TextArea()
```

Proxying children means that if `RepeatingWidget` or `DisplayOnlyWidget` have `children` set, this is passed to their `child`. The following is equivalent to the definition above:

```
class MyForm(twf.Form):
    child = twf.TableLayout()
    b = twf.TextArea()
```

And this is used by classes like `TableForm` and `TableFieldSet` to allow the user more concise widget definitions:

```
class MyForm(twf.TableForm):
    b = twf.TextArea()
```

Automatic ID

Sub classes of `Page` that do not have an id, will have the id automatically set to the name of the class. This can be disabled by setting `_no_autoid` on the class. This only affects that specific class, not any subclasses.

1.6.7 Widgets as Controllers

Sometimes widgets will want to define controller methods. This is particularly useful for Ajax widgets. Any widget can have a `request()` method, which is called with a WebOb `Request` object, and must return a WebOb

`Response` object, like this:

```
class MyWidget(twc.Widget):
    id = 'my_widget'
    @classmethod
    def request(cls, req):
        resp = webob.Response(request=req, content_type="text/html; charset=UTF8")
        # ...
        return resp
```

For the `request()` method to be called, the widget must be registered with the `ControllersApp` in the middleware. By default, the path is constructed from /controllers/, and the widget's id. A request to /controllers/ refers to a widget with id `index`. You can specify `controllers_prefix` in the configuration.

For convenience, widgets that have a `request()` method, and an `id` will be registered automatically. By default, this uses a global `ControllersApp` instance, which is also the default controllers for `make_middleware()`. If you want to use multiple controller applications in a single python instance, you will need to override this.

You can also manually register widgets:

```
twc.core.register_controller(MyWidget, 'mywidget')
```

Sometimes it is useful to dynamically acquire what URL path a Widget's controller is mounted on. For this you can use:

```
MyWidget.controller_path()
```

Methods to override

view_request Instance method - get self and req. load from db

validated_request Class method - get cls and validated data

ajax_request Return python data that is automatically converted to an ajax response

1.6.8 Validation

One of the main features of any forms library is the validation of form input, e.g checking that an email address is valid, or that a user name is not already taken. If there are validation errors, these must be displayed to the user in a helpful way. Many validation tasks are common, so these should be easy for the developer, while less-common tasks are still possible.

We can configure validation on form fields like this:

```
class child(twf.TableForm):
    name = twf.TextField(validator=twc.Required)
    group = twf.SingleSelectField(options=['', 'Red', 'Green', 'Blue'])
    notes = twf.TextArea(validator=twc.StringLengthValidator(min=10))
```

To enable validation we also need to modify the application to handle POST requests:

```
def app(environ, start_response):
    req = wo.Request(environ)
    resp = wo.Response(request=req, content_type="text/html; charset=UTF8")
    if req.method == 'GET':
        resp.body = MyForm.display().encode('utf-8')
    elif req.method == 'POST':
        try:
```

(continues on next page)

(continued from previous page)

```
            data = MyForm.validate(req.POST)
            resp.body = 'Posted successfully ' + wo.html_escape(repr(data))
        except twc.ValidationError, e:
            resp.body = e.widget.display().encode('utf-8')
    return resp(environ, start_response)
```

If you submit the form with some invalid fields, you should see error messages sidle up to each relevant field.

Whole Form Message

If you want to display a message at the top of the form, when there are any errors, define the following validator:

```
class MyFormValidator(twc.Validator):
    msgs = {
        'childerror': ('form_childerror', 'There were problems with the details you␣
↪entered. Review the messages below to correct your submission.'),
    }
```

And in your form:

```
validator = MyFormValidator()
```

Conversion

Validation is also responsible for conversion to and from python types. For example, the DateValidator takes a string from the form and produces a python date object. If it is unable to do this, that is a validation failure.

To keep related functionality together, validators also support coversion from python to string, for display. This should be complete, in that there are no python values that cause it to fail. It should also be precise, in that converting from python to string, and back again, should always give a value equal to the original python value. The converse is not always true, e.g. the string "1/2/2004" may be converted to a python date object, then back to "01/02/2004".

Validation Errors

When there is an error, all fields should still be validated and multiple errors displayed, rather than stopping after the first error.

When validation fails, the user should see the invalid values they entered. This is helpful in the case that a field is entered only slightly wrong, e.g. a number entered as "2,000" when commas are not allowed. In such cases, conversion to and from python may not be possible, so the value is kept as a string. Some widgets will not be able to display an invalid value (e.g. selection fields); this is fine, they just have to do the best they can.

When there is an error is some fields, other valid fields can potentially normalise their value, by converting to python and back again (e.g. 01234 -> 1234). However, it was decided to use the original value in this case.

In some cases, validation may encounter a major error, as if the web user has tampered with the HTML source. However, we can never be completely sure this is the case, perhaps they have a buggy browser, or caught the site in the middle of an upgrade. In these cases, validation will produce the most helpful error messages it can, but not attempt to identify which field is at fault, nor redisplay invalid values.

Required Fields

If a field has no value, if defaults to `None`. It is down to that field's validator to raise an error if the field is required. By default, fields are not required. It was considered to have a dedicated `Missing` class, but this was decided against, as `None` is already intended to convey the absence of data.

Security Consideration

When a widget is redisplayed after a validation failure, it's value is derived from unvalidated user input. This means widgets must be "safe" for all input values. In practice, this is almost always the case without great care, so widgets are assumed to be safe.

Warning: If a particular widget is not safe in this way, it must override `_validate()` and set `value` to *None* in case of error.

Validation Messages

When validation fails, the validator raises `ValidationError`. This must be passed the short message name, e.g. "required". Each validator has a dictionary mapping short names to messages that are presented to the user, e.g.:

```
msgs = {
    'tooshort': 'Value is too short',
    'toolong': 'Value is too long',
}
```

Messages can be overridden on a global basis, using `validator_msgs` on the middleware configuration. For example, the user may prefer "Value is required" instead of the default "Enter a value" for a missing field.

A Validator can also rename mesages, by specifying a tuple in the `msgs` dict. For example, `ListLengthValidator` is a subclass of `LengthValidator` which raises either `tooshort` or `toolong`. However, it's desired to have different message names, so that any global override would be applied separately. The following `msgs` dict is used:

```
msgs = {
    'tooshort': ('list_tooshort', 'Select at least $min'),
    'toolong': ('list_toolong', 'Select no more than $max'),
}
```

Within the messages, tags like `$min` are substituted with the corresponding attribute from the validator. It is not possible to specify the value in this way; this is to discourage using values within messages.

FormEncode

Earlier versions of ToscaWidgets used FormEncode for validation and there are good reasons for this. Some aspects of the design work very well, and FormEncode has a lot of clever validators, e.g. the ability to check that a post code is in the correct format for a number of different countries.

However, there are challenges making FormEncode and ToscaWidgets work together. For example, both libraries store the widget hierarchy internally. This makes implementing some features (e.g. `strip_name` and `tw2.dynforms.HidingSingleSelectField`) difficult. There are different needs for the handling of unicode, leading ToscaWidgets to override some behaviour. Also, FormEncode just does not support client-side validation, a planned feature of ToscaWidgets 2.

ToscaWidgets 2 does not rely on FormEncode. However, developers can use FormEncode validators for individual fields. The API is compatible in that `to_python()` and `from_python()` are called for conversion and validation, and `formencode.Invalid` is caught. Also, if FormEncode is installed, the `ValidationError` class is a subclass of `formencode.Invalid`.

Using Validators

There's two parts to using validators. First, specify validators in the widget definition, like this:

```
class RegisterUser(twf.TableForm):
    validator = twc.MatchValidator('email', 'confirm_email')
    name = twf.TextField()
    email = twf.TextField(validator=twc.EmailValidator)
    confirm_email = twf.PasswordField()
```

You can specify a validator on any widget, either a class or an instance. Using an instance lets you pass parameters to the validator. You can code your own validator by subclassing *tw2.core.Validator*. All validators have at least these parameters:

class tw2.core.**Validator**(***kw*)

Base class for validators

required Whether empty values are forbidden in this field. (default: False)

strip Whether to strip leading and trailing space from the input, before any other validation. (default: True)

To convert and validate a value to Python, use the `to_python()` method, to convert back from Python, use `from_python()`.

To create your own validators, sublass this class, and override any of `_validate_python()`, `_convert_to_python()`, or `_convert_from_python()`. Note that these methods are not meant to be used externally. All of them may raise ValidationErrors.

Second, when the form values are submitted, call `validate()` on the outermost widget. Pass this a dictionary of the request parameters. It will call the same method on all contained widgets, and either return the validated data, with all conversions applied, or raise `tw2.core.ValidationError`. In the case of a validation failure, it stores the invalid value and an error message on the affected widget.

Chaining Validators

In some cases you may want validation to succeed if any one of a number of checks pass. In other cases you may want validation to succeed only if the input passes *all* of a number of checks. For this, `tw2.core` provides the `Any` and `All` validators which are subclasses of the extendable `CompoundValidator`.

Implementation

A two-pass approach is used internally, although this is generally hidden from the developer. When `Widget.validate()` is called it first calls:

tw2.core.validation.**unflatten_params**(*params*)

This performs the first stage of validation. It takes a dictionary where some keys will be compound names, such as "form:subform:field" and converts this into a nested dict/list structure. It also performs unicode decoding, with the encoding specified in the middleware config.

If this fails, there is no attempt to determine which parameter failed; the whole submission is considered corrupt. If the root widget has an `id`, this is stripped from the dictionary, e.g. `{'myid': {'param':'value', ...}}` is converted to `{'param':'value', ...}`. A widget instance is created, and stored in request local storage. This allows compatibility with existing frameworks, e.g. the `@validate` decorator in TurboGears. There is a hook in `display()` that detects the request local instance. After creating the instance, validate works recursively, using the `_validate()`.

Widget.**_validate**(**args*, ***kw*)

Inner validation method; this is called by validate and should not be called directly. Overriding this method in widgets is discouraged; a custom validator should be coded instead. However, in some circumstances overriding is necessary.

RepeatingWidget.**_validate**(**args*, ***kw*)

The value must either be a list or None. Each item in the list is passed to the corresponding child widget for validation. The resulting list is passed to this widget's validator. If any of the child widgets produces a validation error, this widget generates a "childerror" failure.

CompoundWidget.**_validate**(**args*, ***kw*)

The value must be a dict, or None. Each item in the dict is passed to the corresponding child widget for validation, with special consideration for _sub_compound widgets. If a child returns vd.EmptyField, that value is not included in the resulting dict at all, which is different to including None. Child widgets with a key are

passed the validated value from the field the key references. The resulting dict is validated by this widget's validator. If any child widgets produce an errors, this results in a "childerror" failure.

Both `_validate()` and `to_python()` take an optional state argument. `CompoundWidget` and `RepeatingWidget` pass the partially built dict/list to their child widgets as state. This is useful for creating validators like `MatchValidator` that reference sibling values. If one of the child widgets fails validation, the slot is filled with an `Invalid` instance.

1.6.9 General Considerations

Request-Local Storage

ToscaWidgets needs access to request-local storage. In particular, it's important that the middleware sees the request-local information that was set when a widget is instatiated, so that resources are collected correctly.

The function tw2.core.request_local returns a dictionary that is local to the current request. Multiple calls in the same request always return the same dictionary. The default implementation of request_local is a thread-local system, which the middleware clears before and after each request.

In some situations thread-local is not appropriate, e.g. twisted. In this case the application will need to monkey patch request_local to use appropriate request_local storage.

pkg_resources

tw2.core aims to take advantage of pkg_resources where it is available, but not to depend on it. This allows tw2.core to be used on Google App Engine. pkg_resources is used in two places:

- In ResourcesApp, to serve resources from modules, which may be zipped eggs. If pkg_resources is not available, this uses a simpler system that does not support zipped eggs.
- In EngingeManager, to load a templating engine from a text string, e.g. "genshi". If pkg_resources is not available, this uses a simple, built-in mapping that covers the most common template engines.

Framework Interface

ToscaWidgets is designed to be standalone WSGI middeware and not have any framework interactions. However, when using ToscaWidgets with a framework, there are some configuration settings that need to be consistent with the framework, for correct interaction. Future vesions of ToscaWidgets may include framework-specific hooks to automatically gather this configuration. The settings are:

- default_view - the template engine used by the framework. When root widgets are rendered, they will return a type suitable for including in this template engine. This setting is not needed if only Page widgets are used as root widgets, as there is no containing template in that case.
- translator - needed for ToscaWidget to use the same i18n function as the framework.

Unit Tests

To run the tests, in `tw2.devtools/tests` issue:

```
nosetests --with-doctest --doctest-extension=.txt
```

1.7 Changelog

1.7.1 2.3.0

- Support overriding fields in subclasses of a *Form*

- Support for formencode validators in *CompoundValidator*

1.7.2 2.2.9

- Fix loading of templates on some systems where system encoding is not UTF8 (templates are always loaded as utf8)

1.7.3 2.2.7

- Fix support for Python3.8 removing *cgi.escape*
- Fix deprecated support for absolute paths in *resource_filename*

1.7.4 2.2.6

- New Documentation

1.7.5 2.2.5

- Added english transation, so that the gettext translator finds it and prefers english when multiple languages are involved and english is the favourite one.
- Fixed an issue with i18n translator on Python3

1.7.6 2.2.4

- Templating now uses render_unicode to render mako templates and avoid unicode dance ecc33fc
- Avoid modifying validation messages dict while iterating on it 66c7e3d
- Fix Genshi relative imports when running test suite on top directory

1.7.7 2.2.3

- Kajiki Template Engine Support
- Disallow DisplayOnlyWidget as child of RepeatingWidget as it doesn't work anyways 4c15c5a
- Flush memozation cache when auto_reload_templates in the middleware is enabled
- Fix safe_validate with FormEncode validators 3fa88ac

1.7.8 2.2.2

- Fix CompoundWidget and MatchValidator
- Fix archive_tw2_resources 8956e83
- Fix DateValidator and DateTimeValidator to be in sync with tw2.forms 06da5b9

1.7.9 2.2.1

- Merge branch 'hotfix/2.1.6' a699822e5
- compound_key was ignoring key for RepeatingWidget ed0946146
- Fix for DisplayOnlyWidget in compound_id regression 11570e42e
- All and Any validators didn't work with unicode error messages 3c177ad8d
- Merge branch 'master' of @amol-/tw2.core into develop 5254065c0

1.7.10 2.2.0.8

- Fix duplicate class name 1c133c907
- Be able to put an HTML separator between the children of a RepeatingWidget. We also need to support it for the CompoundWidget since it uses the same template db717642d
- Merge pull request #96 from LeResKP/develop 41229bf01
- Re-enable archive_tw2_resources on Python 2 56215397a

1.7.11 2.2.0.7

- – Clean up cache * Hack to fix the tests with empty value attributes for genshi cd5febe2b
- Merge pull request #95 from LeResKP/develop 9f54d72be
- Merge branch 'develop' of github.com:toscawidgets/tw2.core into develop 9142fe165

1.7.12 2.2.0.6

1.7.13 2.2.0.5

- Add a setUp method back to another base test thats missing it. 55b6061ed

1.7.14 2.2.0.4

- Restore an old setUp method for tw2.core.testbase.WidgetTest da2d9bab2

1.7.15 2.2.0.3

- Added a new validator *UUIDValidator* (+test) for UUID/GUIDs ebea7f30b
- Merge pull request #92 from RobertSudwarts/amol 481926de6
- Call me picky, but I think license belongs up there de9d87587
- Merge branch 'amol' into develop 46d68b792
- pep8 5896d4db0
- Fix tests for UUIDValidator bfc4531ec
- Handle case where response.charset is None. e1fe13460

- Merge branch 'develop' of github.com:toscawidgets/tw2.core into develop 4fec80d22

1.7.16 2.2.0.2

- Update one test now that the error message has changed. c31f52732
- Catch if a template is None. a159b6cf1
- Remove direct dependence on unittest so we can get test-generators working again. Relates to #88. f561ef33d
- Turn the css/js escaping tests into generators per engine too. c43bd4d7f
- Kajiki expects unicode these days. 16f6508c2
- Mark this test really as skipping. b59d1ff05
- Skip tests on weird kajiki behavior.... 11285aa68
- Drop python-3.2 support since our deps dont support it. 0f777ea68
- Kill kajiki. ea14b79f1
- Merge pull request #94 from toscawidgets/feature/yielding-again 30e4c4b3d
- Metadata fixups, #90 38e306f88
- Imported doc fragments from tw2.forms 894b28540

1.7.17 2.2.0.1

- Provide more info in this traceback. 77efa240f
- Variable, not Param. 03991510e
- Update TG2 tutorial to current state of affairs cb481999a
- Make some things non-required that were newly required. 14507319d
- Merge branch 'develop' of github.com:toscawidgets/tw2.core into develop f5a00e83d

1.7.18 2.2.0

- Support more webob versions. Fixes #77 e071e9d33
- Constrain webtest version for py2.5. 1214057c1
- Port to python2/python3 codebase. c1d2b7721
- Travis-CI config update. 21a35d470
- Some py3 fixes for tw2.forms. c82fb090f
- @moschlar on the ball. 8b5cdcb81
- Some setup for a port of tw2.devtools to gearbox. 08fd64a11
- Merge branch 'feature/2.2' into develop 4aef579c7
- Mention tw2.core.DirLink in the docs. Fixes #69. dce1db697
- Reference gearbox tw2.browser in the docs. 2562933ee
- Include translations in distribution. 2791169fa

- Merge pull request #82 from Cito/develop f6d1f0502
- Fix #84 in archive_tw2_resources 02eec525f
- Merge pull request #85 from toscawidgets/feature/archive_tw2_resources 8791c3236
- Add a failing test for #25. 5d7b43a9f
- Automatically assign widgets an ID. ca81db016
- Enforce twc.Required (for #25). 94e61ec52
- Deal with faulout from the twc.Required enforcement. b5063a3c7
- Merge pull request #87 from toscawidgets/feature/twc.Required 5add35cb9
- Method generators are not supported in unittest.TestCase subclasses. 30cb85826
- Support if_empty and let BoolValidator validate None to False. a9d48944a
- Merge pull request #88 from Cito/develop 2416cefb8
- Merge branch 'hotfix/2.1.6' a699822e5
- Merge branch 'hotfix/2.1.6' into develop dc99409b9
- Remove the spec file. Fedora has it now. 004c3eda6

1.7.19 2.1.6

- Fix #84 in archive_tw2_resources 65493f6ab
- Support if_empty and let BoolValidator validate None to False. 4008ee77d
- 2.1.6 146d17261

1.7.20 2.1.5

- Make sure future-queued resources make it into the middleware. adb4aec79

1.7.21 2.1.4

- Simplify the validator API and make it compatible with FormEncode. 5e5f91afa
- Merge pull request #75 from Cito/develop eb74470c6

1.7.22 2.1.3

- Validation docs. 4132ff5f6
- Typo fix. Thanks Daniel Lepage. 0fbed935c
- Fixes to tests busted by the introduction of CSSSource. b795f3f2b
- More descriptive ParameterError for invalid ids. 6c06384ff
- Windows support for resource serving. 0b939179a
- Added a half-done test of the chained js feature. fe6924f89
- We won't actually deprecate tw1-style calling. f63a37c51

- Merge branch 'develop' into feature/chained-js-calls c5e3f6a1f
- Added class_or_instance properties fb9211eb0
- Revert "Added class_or_instance properties" 25df3bd3a
- Chaining js calls are back in action. eb7ef5056
- Merge branch 'feature/chained-js-calls' into develop 612d52a88
- Version for 2.0.0. 03f6d1280
- Forgot the damn classifier. a780af954
- Merge branch 'hotfix/classifier' df2556fec
- Merge branch 'hotfix/classifier' into develop 22b667946
- Add coverage to the standard test process. 99400078e
- When widgets have key they should be validated by key and not be id edc575014
- Re-added ancient/missing js_function __str__ behavior discovered in the bowels of moksha. 1d45fe424
- Demoted queued registration messages from "info" to "debug". be23347d1
- Clutch simplejson hacking. fb7c06b66
- Encoding widgets works again. 07fb3c94b
- More PEP8. b387fa470
- Found the killer test. d81926c5a
- Update to that test. 152650597
- A stab at handling function composition. Tests pass. 7ae78e03b
- This is clearly unsustainable. c96fb2898
- Solve the function composition problem. ff432f26a
- Merge branch 'feature/function-composition' into develop 5f46d5069
- Some comments in the encoder initialization. a479c7aa5
- The output of this test changes depending on what other libs are installed. 1b4306160
- Abstracted ResourceBundle out of Resource for tw2.jqplugins.ui. 56a6ba35a
- When widget has key and so gets data by key validation was still returning data by id. Now validation returns data by key when available. Also simplify CompoundWidget validation fa197ba30
- Cover only the tw2.core package 75001ec74
- Fix regression in tw2.sqla. f6089fd7f
- Revert CompoundValidation tweak. Works with tw2.sqla now. Fixes #9. 032994731
- Added a test case for amol's validation situation. 06ac1b3fb
- Supress top-level validator messages if they also apply messages to compound widget children. c144b01f3
- Correctly supress top-level validator messages. 8b15822e1
- Write test to better test CompoundWidget error reporting 74dd87075
- Handle unspecified childerror case uncovered by latest test. e94c80341
- Differentiated test names. 5a7ef40cc

- Compatibility with dreadpiratebob and percious's tree. af7a2e6b8
- Avoid receiving None instead of the object itself when object evaluates to False e8c513c3a
- 2.0.1 release. c056c88f6
- Initial RPM spec. 12cec0ed8
- Rename. 5ebc78d87
- Removed changelog. It's from the way back tw1 days. eb5fdcc65
- Skipping tests that rely on tw2.forms and yuicompressor. c7ae7984a
- We don't actually require weberror. 7b269e77e
- Include test data for koji builds. 3f61860d3
- First iteration of the new rpm. It actually built in koji. 6b924cdda
- exception value wasn't required and breaks compatibility with Python2.5 de857ce6e
- Merge pull request #16 from amol-/develop 0e9faf439
- More Py2.5 compat. 057ac45bb
- 2.0.2 release with py2.5 bugfixes for TG. bd8304957
- Specfile update for 2.0.2. d9aeb76b3
- Changed executable bit for files that should/shouldn't have it. 4d77e3043
- Exclude `.pyc` files from template directories. 4d281c684
- Version bump for rpm fixes. a76db4c94
- Remove pyc files from the sdist package. Weird. da3ddaea1
- Switched links in the doc from old blog to new blog. 8f7332fd1
- Be more careful with the multiprocessing,logging import hack. a8857267e
- Compatibility with older versions of simplejson. 64d16f234
- Test suite fixes on py2.6. e37b7e1c6
- 2.0.4 with improved py2.6 support. 7b6784e1d
- A little more succint in the middleware. 5cc582cd9
- Allow streaming html responses to pass through the middleware untouched. 3f4a5a4b9
- Simple formatting in the spec. d7020a9fa
- Version bump. 48768720b
- Stripped out explicit references to kid and cheetah. 595ba7c6c
- Removed unused reference to reset_engine_name_cache. 0e4c40e64
- Removed unnecessary "reset_engine_name_cache" 2b3ed27a7
- Removed a few leftover references to kid. 1755fd14a
- More appropriate variable name. 1c27c620a
- First rewrite of templating system. 283367bb8
- Template caching. 4d16358e0
- First stab at jinja2 support. 17d17234a

- Update to the docs. e9658290b
- Massive dos2unix pass. For good health. e74bbc42b
- PEP8. 62d256c4d
- Reference email thread regarding “displays_on” 25ffcd339
- Added support for kajiki. f809d1a5d
- Default templates for kajiki and jinja. 9a170d3cb
- More robust testing of new templates. 55f1fbe0a
- Pass filename to mako templates for easier debugging. 5e63adcbe
- More correct dotted template loading. 07b67c84d
- Added support for chameleon. fa8c160d4
- Default chameleon templates. 69de63cf6
- Updated docs with kajiki and chameleon. ef291ce4a
- Added three tests for http://bit.ly/KNYAxq 0e775ab1e
- Resurrecting the smarter logic of the “other” tw encoder. Hurray for git history. 1379196d3
- Added test for #12. Passes. b6bbf92a4
- Use __name__ in tests. fbe2b6979
- Added failing test for Issue #18. e962a03fb
- Merge pull request #21 from toscawidgets/feature/multiline-js c9e0ada6f
- Merge branch ‘develop’ into feature/template-sys b32a024c3
- Merge branch ‘develop’ into feature/issue-18 5b1c1dadf
- Guess modname in post_define. Fixes #18. d3d2aeb35
- Merge branch ‘feature/issue-18’ into develop 4f0d496fc
- Version bump - 2.0.6. ea7637a20
- Don’t check for ‘not value’ in base to_python. Messes up on cgi.FieldStorage. 204e20fbd
- Added a note to the docs about altering JSLink links. Fixes #15. 28e458fe4
- dos2unix pass on the docs/ folder. ce4f813e7
- Typo fix. 34fee8fa9
- Trying out travis-ci. 8e9414ae0
- Trying out travis-ci. abc5b4161
- Updates for testing on py2.5 and py2.6. 56ce437ef
- Merge branch ‘develop’ 0f4b81113
- Added build table to the README. 4da336497
- Merge branch ‘develop’ into feature/template-sys 832435945
- Python2.5 support. 66e93b66d
- JS and CSSSource require a .src attr. ca02d9713
- Use mirrors for travis. b504714da

- Revert "Use mirrors for travis." 9fc882050
- Fixed mako and genshi problems in new templating system found by testing against tw2.devtools. 41b8e5264
- Version bump – 2.1.0a ft. templating system rewrite. c89009332
- Ship new templates with the source dist. 2fb6cf8da
- Attribute filename for jinja and kajiki. d130c3c9f
- Provide an option for WidgetTest to exclude engines. c822b2a66
- 2.1.0a4 - Fix bug in automatic resource registration. efcd51724
- Support template inheritance at Rene van Paassen's request. fc58e929a
- Version bump for template inheritance. 6b6658870
- Fix required Keyword for Date*Validators 14196d9ce
- Bridge the tw2/formencode API divide. 547357c7f
- Make rendering_extension_lookup propagate up to templating layer 8d89dabd8
- Added test for #30. Oddly, it passes 7d1d83852
- Trying even harder to test #30. b66b59ff5
- Version bump to 2.1.0b1. 3483107a6
- Puny py2.5 has no context managers. cb1e821c8
- PEP8. Cosmetic. 50d88cc93
- Future-proofing. @amol- is a rockstar. bb006dfeb
- Conform with formencode. Fixes #28. f3bf2a821
- Improve handling of template path names under Windows. e2bbeb29c
- Borrowed backport of os.path.relpath for py2.5. Related to #30. f29337629
- Whoops. Forgot to use the new relpath. #30. f308bef92
- Use util.relpath instead of os.path.relpath. 3c302eaac
- .req() returns the validated widget is one exists. be8f39404
- Use `**kw` even when pulling in the validated widget. f78492be9
- Trying to duplicate an issue with Deferred. cefbbfd73
- Tests for #41. 7c61047b9
- Handle arguments to display() called as instance method. 86894492d
- Cosmetic. b94180f25
- Found the failing test for @amol-'s case. 284c66a38
- Allow Deferred as kwarg to .display(). d4c6dcfc6
- Second beta 2.1.0b2 to verify some bugfixes. b6ff67ab7
- Failing test for Deferred. d26389d13
- @amol-'s fix for the Deferred subclassing problem. c08c0508b
- 2.1.0. 725fd6aba
- Fixup copyright date bc509ca66

- avoid issues with unicode error messages b5a314de7
- Link to rtfd from README. 1269dff73
- Added jinja filter to take care of special case html bolean attributes such as radio checked} da25dbfaf
- Added htmlbools filter to jinja templates fb00eac66
- Fixed corner case which produced harmless but incorrect output if the special case attribute value is False 38a4505b8
- Merge pull request #48 from clsdaniel/develop 270784d5a
- Removed commented-out lines. 55af65d6c
- 2.1.1 for jinja updates and misc bugfixes. 0ff5ffcd2
- Since 2.0 autoescaping in widgets got lost due to new templates management 59f478fb5
- Mark attrs as Markup to avoid double escaping 5e138ace2
- Mark as already escape JSFuncCall too and update test to check the result for all the template engines 7c0c60ae2
- Merge pull request #49 from amol-/develop f6a3dda84
- Add proper escaping for JS and CSS sources af6d233df
- Merge pull request #50 from amol-/develop e99f82879
- Provide a Widget compound_key make available a compound_key attribute which can be used by tw2.forms as the default value for FormField name argument ee571a215
- Version bump, 2.1.2. 1b64e3f83
- Allow inline templates with no markup. de19fa2b3
- PEP8. c2da40a1b
- Test that reveals a bug in tw2.jqplugins. 6a88d0413
- Do not translate empty strings, this does not work. e4f29829d
- Merge pull request #53 from Cito/develop 168f2727f
- Add translations and passing lang via middleware a10a14e26
- Merge pull request #59 from Cito/develop cbf603238
- Inject CSS/JSSource only once. ae13c369a
- Merge pull request #61 from Cito/develop bb5c2a225
- Test blank validator for both None and empty string. 1167286c3
- Add some more translations. 32374168d
- Merge pull request #64 from Cito/develop 50fc09a24
- Fix #63. df2920d83
- Added a note about the add_call method to the design doc. e901b1243
- Reference js_* docstrings from design doc. Fixes #58. 55001c742
- General docs cleanup. 144d5cfbb
- Fix broken links to tw2.core-docs-pyramid 14e5223e2
- Fix broken links to tw2.core-docs-turbogears 55a333b1c
- Merge pull request #66 from lukasgraf/lg-doc-url-fixes 4d123d0b1

- provide compatibility with formencode validators c382eed46
- Merge pull request #71 from amol-/develop 65b9550ca
- Link to github bug tracker from docs. Fixes #67. f849b5d03
- pass on state value in validation. 7c6791d80
- Updated pyramid docs. Fixes #23. 9547108fb
- Don't let `add_call` pile-up new js resources. f1d698c55

CHAPTER 2

Online Resources

ToscaWidgets, as it was originall born from TurboGears Widgets, shares many online resources with TurboGears. If you have questions on how to use TW2 feel free to ask them in TurboGears channel or Mailing List.

- Bug tracker: GitHub.
- Gitter Channel: TurboGears Channel
- Mailing List: TurboGears Users

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

S

T

U

V

W