ToscaWidgets 2 Documentation

Release 2.2.4

Paul Johnston, Alberto Valverde
Contributors

May 16, 2018

Contents

ToscaWidgets aims to be a practical and useful widgets framework that helps people build interactive websites with compelling features, faster and easier. Widgets are re-usable web components that can include a template, server-side code and JavaScripts/CSS resources. The library aims to be: flexible, reliable, documented, performant, and as simple as possible. For changes since ToscaWidgets 0.9, see *What's New in ToscaWidgets 2?*.

You can see the available widgets in the Widget Browser.

ToscaWidgets 2 library packages follow the same naming convention, for example:

- tw2.core – Core functionality – no end-usable widgets here.
- tw2.forms – Basic forms library
- tw2.dynforms – Dynamic forms – client-side and Ajax
- tw2.sqla – SQLAlchemy database interface, similar to Sprox and Rum
- tw2.yui – tw2 wrappers around Yahoo User Interface widgets
- tw2.jquery – tw2 wrappers around jQuery core functionality.
- tw2.jqplugins.ui – tw2 wrappers around jQuery-UI widgets.
- tw2.jit – tw2 wrappers around the Javascript Infovis Toolkit.
- ... and many more.

Online Resources

- Live demos – Pick and choose from available libraries from the tw2 Widget Browser.
- Tutorials for doing –
  - Dynamic database-driven forms with tw2 and Pyramid.
  - Dynamic database-driven forms with *tw2 and TurboGears 2*.
  - Dynamic database-driven forms with *tw2 all by its standalone self*.
  - Interactive relationship graphs with tw2.jit and Pyramid.
  - Interactive relationship graphs with tw2.jit and TurboGears 2.1.
  - Database-aware jqgrid, with jqplot and portlets in a TG2.1 app.
  - Bubble charts with tw2.protovis.
- Nightly run test results.
- Email list: toscawidgets-discuss.
- IRC channel: `#toscawidgets` on `irc.freenode.net`
- Bug tracker: The toscawidgets github account. * (All ToscaWidgets 2 issues should go here, regardless of which component the issue exists in. However, ToscaWidgets 0.9 bugs must not go on this tracker.)
- *Changelog*

Contents

CHAPTER 1

Tutorial

1.1 Installation

First of all, you need Python - version 2.5, 2.6 or 2.7. The recommended way to install ToscaWidgets is using pip. Once you have pip itself installed, you should issue (with sudo if required):

```
pip install tw2.dynforms tw2.devtools tw2.sqla genshi elixir
```

This install the widget libraries and a number of dependencies. Once this is complete, try running the widget browser to check this worked. Issue:

```
gearbox tw2.browser
```

And browse to `http://localhost:8080/`, where you should be able to see the installed widgets.

If you have any problems during install, try asking on the group.

1.2 Using ToscaWidgets

ToscaWidgets can be used with a web framework, such as Pylons or TurboGears, or it can be used standalone, with ToscaWidgets itself as the framework. There are separate tutorials depending on how you want to use the library:

1.2.1 Standalone Tutorial

Note: The files created in this tutorial can be downloaded as a `.zip file`.

Installing ToscaWidgets2

Your operating system may provide another way of installing ToscaWidgets2 (*yum*, *apt-get*, etc...). Failing that, you can use *pip*:

```
$ pip install tw2.core tw2.forms tw2.dynforms tw2.devtools tw2.sqla tw2.jqplugins.
↪jqgrid
```

TW2 supports many different *templating engines*. For this tutorial we'll be writing `genshi` templates, so install support for that as well:

```
$ pip install genshi
```

In this tutorial, we're also going to be showing off some of the database features of `tw2.sqla`. For our application we're going to use `elixir`; install it too:

```
$ pip install elixir
```

Building a Page

To get started, we'll build a simple "Hello World" application. First, create `myapp.py` with the following:

```
import tw2.core
import tw2.devtools

class Index(tw2.core.Page):
    template = 'genshi:./index.html'

tw2.devtools.dev_server()
```

Here were are creating a Page widget, called Index, with a template specified. Index is a special name that matches the root URL. We need to create the template, so in the same directory, create `index.html` with the following content:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
    <head>
        <title>My Application</title>
    </head>
    <body>
        <h1>My Application</h1>
        <p>Hello World!</p>
    </body>
</html>
```

This is a Genshi template. This simple example just uses static HTML, but the template system has many other features to explore. Now, start the application:

```
python myapp.py
```

And browse to `http://localhost:8000/` to check that it's working.

To go beyond simple static HTML, the first step is to have Python code executed when the page is generated. The `fetch_data` method is called at this time, and we can override this to hook the event. We will add simple code that reflects the received request. Add the following to *Index* in `myapp.py`:

```
def fetch_data(self, req):
    self.req = str(req)
```

The `req` variable supplied is a WebOb `Request` object. To access a URL parameter, use `req.GET['myparam']`. In this example, we simply use the string representation of the request.

And in `index.html`, change the `<p>Hello World!</p>` line to:

```
<pre>$w.req</pre>
```

The `$w` variable refers to the Python widget object that called the template.

Restart the application and refresh your browser to see this.

Building a Form

We'll now create a simple form. In this example we're going to create a movie database. First, add to the top of `myapp.py`:

```
import tw2.forms
```

Now, add another class to this file, before `tw2.devtools.dev_server()`:

```
class Movie(tw2.forms.FormPage):
    title = 'Movie'
    class child(tw2.forms.TableForm):
        title = tw2.forms.TextField(validator=tw2.core.Required)
        director = tw2.forms.TextField()
        genre = tw2.forms.CheckBoxList(
            options=['Action', 'Comedy', 'Romance', 'Sci-fi'])
        class cast(tw2.forms.GridLayout):
            extra_reps = 5
            character = tw2.forms.TextField()
            actor = tw2.forms.TextField()
```

Before we explain this code, restart the application and browse to `http://localhost:8000/movie` to see how the form looks. Have a go an entering values and submitting; notice the difference when you specify a title compared to not.

The *FormPage* widget adds functionality beyond *Page* in that it handles POST requests and redisplays the form on validation failures. The *TableForm* widget displays the form, including the submit button, and does the table layout. The form fields are specified in a fairly self-explanatory manner, noting that validation is specified for *title*. Here, *GridLayout* is used as a kind of sub-form, which allows multiple cast members to be specified.

The form does not look particularly appealing. To try to improve this, lets add some CSS. We'll start with something simple; create `myapp.css` with the following:

```
th {
    vertical-align: top;
    text-align: left;
    font-weight: normal;
}

ul {
    list-style-type: none;
}
```

(continues on next page)

(continued from previous page)

```
.required th {
    font-weight: bold;
}
```

Notice the use of the "required" class. TableForm applies this to rows that contain a field that is required.

Before TableForm will inject `myapp.css` into the page, we'll have to add it to the list of resources. Add the following to the top of the *Movie* class definition just above the line `title = 'Movie'`:

```
resources = [tw2.core.CSSLink(filename='myapp.css')]
```

Restart `myapp.py` and browse to `http://localhost:8000/movie` to see the new css in action.

Connecting to a Database

The next step is to save movies to a database. To do this, we'll use SQLAlchemy and Elixir to define a database model. Create `model.py` with the following:

```
import elixir, tw2.sqla
elixir.session = tw2.sqla.transactional_session()
elixir.metadata = elixir.sqlalchemy.MetaData('sqlite:///myapp.db')
```

This is code is required to set up the database connection. It will use an SQLite database, `myapp.db` in the current directory. Now, add the code to define our tables (still to `model.py`):

```
class Movie(elixir.Entity):
    title = elixir.Field(elixir.String)
    director = elixir.Field(elixir.String)
    genre = elixir.ManyToMany('Genre')
    cast = elixir.OneToMany('Cast')

class Genre(elixir.Entity):
    name = elixir.Field(elixir.String)
    def __unicode__(self):
        return self.name

class Cast(elixir.Entity):
    movie = elixir.ManyToOne(Movie)
    character = elixir.Field(elixir.String)
    actor = elixir.Field(elixir.String)
```

Finally, a small piece of boilerplate code is required at the bottom:

```
elixir.setup_all()
```

This defines three tables - Movie, Genre and Cast, with relations between them. To learn more about the Elixir syntax, read the Elixir tutorial. The next step is to create our database. In the python interpreter, issue:

```
import model
model.elixir.create_all()
```

We'll now add the genres to the database:

```
model.Genre(name='Action')
model.Genre(name='Comedy')
```

(continues on next page)

(continued from previous page)

```
model.Genre(name='Romance')
model.Genre(name='Sci-fi')
model.elixir.session.commit()
```

Now, exit the Python interpreter, and update `myapp.py` to connect the *Movie* form to the database. At the top of the file add:

```
import tw2.sqla
import model
```

Replace `class Movie(tw2.forms.FormPage):` with:

```
class Movie(tw2.sqla.DbFormPage):
    entity = model.Movie
```

Add a line just below the `class child(tw2.forms.TableForm):` line that reads:

```
id = tw2.forms.HiddenField
```

And replace `genre = tw2.forms.CheckBoxList(...)` with:

```
genre = tw2.sqla.DbCheckBoxList(entity=model.Genre)
```

Finally, we need to enable the wrapper that automatically commits transactions after each request. Replace `tw2.devtools.dev_server()` with:

```
tw2.devtools.dev_server(repoze_tm=True)
```

With this done, restart the application and try submitting a movie.

Front Page

We want a front page that provides a list of our movies, and the ability to click on a movie to edit it. We can use a GridLayout for this; replace the *Index* class in `myapp.py` with:

```
class Index(tw2.sqla.DbListPage):
    entity = model.Movie
    title = 'Movies'
    newlink = tw2.forms.LinkField(link='movie', text='New', value=1)
    class child(tw2.forms.GridLayout):
        title = tw2.forms.LabelField()
        id = tw2.forms.LinkField(link='movie?id=$', text='Edit', label=None)
```

When you browse to /, you will see a list of movies that have been submitted, and be able to edit each one. When you're done editing, we want to redirect back to this front page, so add the following to the *Movie* class:

```
redirect = '/'
```

This gives our application just enough functionality to be a basic movie tracking system.

GrowingGrid

The list of cast is somewhat limited; there's no easy way to delete a row, any you can't add more than five people at once. We can use a widget from tw2.dynforms to improve this. GrowingGridLayout is a dynamic grid that can grow client-side. Be aware that tw2.dynforms requires your site's visitors to have JavaScript enabled.

To use this, update `myapp.py`; at the top of the file add:

```
import tw2.dynforms
```

Replace this:

```
class cast(tw2.forms.GridLayout):
    extra_reps = 5
```

With:

```
class cast(tw2.dynforms.GrowingGridLayout):
```

Finally, change this:

```
class child(tw2.forms.TableForm):
```

To this:

```
class child(tw2.dynforms.CustomisedTableForm):
```

jQuery's jqGrid

There are a lot of *non-core* TW2 widget libraries out there, and just to give you a taste, we'll use one to add one more view to our Movie app.

In your handy-dandy terminal, run:

```
$ pip install tw2.jqplugins.jqgrid
```

Go back to editing `myapp.py` and add to the top:

```
import tw2.jqplugins.jqgrid
```

And add another two whole classes near the bottom of the file but above `tw2.devtools.dev_server(repoze_tm=True)`:

```
class GridWidget(tw2.jqplugins.jqgrid.SQLAjqGridWidget):
    entity = model.Movie
    excluded_columns = ['id']
    prmFilter = {'stringResult': True, 'searchOnEnter': False}
    pager_options = { "search" : True, "refresh" : True, "add" : False, }
    options = {
        'url': '/db_jqgrid/',
        'rowNum':15,
        'rowList':[15,30,50],
        'viewrecords':True,
        'imgpath': 'scripts/jqGrid/themes/green/images',
        'width': 900,
        'height': 'auto',
    }

tw2.core.register_controller(GridWidget, 'db_jqgrid')

class Grid(tw2.core.Page):
    title = 'jQuery jqGrid'
    child = GridWidget
```

1.2.2 TurboGears 2 Tutorial

Note: The files created in this tutorial can be downloaded as a .zip file, a .tar file, or can be cloned from a github repository.

Note: Various parts of this tutorial differ based on the version of TurboGears you are using. If you are unsure about the exact version, fire up a Python interpreter and type:

```
>>> import tg
>>> tg.__version__
'2.1.1'
```

Please also consult the accompanying TurboGears documentation for your version: TurboGears 2.2 or TurboGears 2.3.

Enabling ToscaWidgets

First, you need to create a TurboGears project. The full instructions are in the TurboGears documentation, briefly (assuming you have virtualenvwrapper installed):

```
$ mkvirtualenv --no-site-packages tw2-and-tg2
$ pip install tg.devtools
```

TurboGears<=2.2	TurboGears>=2.3
`$ paster quickstart -s -n -m myapp`	`$ gearbox quickstart -s -n -m myapp`

This command creates a new TurboGears project named myapp in the directory myapp using the SQLAlchemy object relational mapper (-s), no authentication (-n) and the Mako templating engine (-m).

Now change to the newly created directory:

```
$ cd myapp
```

Then open `setup.py` and add the following to the `install_requires=[...]` entry:

```
install_requires=[

    ...

    "tw2.core",
    "tw2.forms",
    "tw2.dynforms",
    "tw2.sqla",
    "tw2.jqplugins.jqgrid",
    ],
```

Once that's done, install your dependencies by running:

```
$ pip install -e .
```

TurboGears 2.0

Edit `myapp/config/middleware.py`, add `import tw2.core as twc` to the top of the file, and replace the line:

```
app = make_base_app(global_conf, full_stack=True, **app_conf)
```

with the following two lines:

```
custom = lambda app : twc.make_middleware(app, default_engine='mako')
app = make_base_app(global_conf, wrap_app=custom, full_stack=True, **app_conf)
```

TurboGears 2.1

Edit `myapp/config/app_cfg.py` and add at the end:

```
base_config.use_toscawidgets2 = True
```

TurboGears 2.2 or greater

ToscaWidgets 2 is fully integrated and supported by these TurboGears versions, so no changes are needed.

Verifying the installation

To check whether the installation worked:

TurboGears<=2.2	TurboGears>=2.3
`$ paster serve development.ini`	`$ gearbox serve`

Building a Form

We'll create a movie database as in the *Standalone Tutorial* example. First, let's create a movie controller and mount it from our root controller.

Create a new file `myapp/controllers/movie.py` with the contents:

```
from tg import expose, request

from myapp.lib.base import BaseController
from myapp import model

__all__ = ['MovieController']

import tw2.core
```

(continues on next page)

(continued from previous page)

```
import tw2.forms


class MovieForm(tw2.forms.FormPage):
    title = 'Movie'

    class child(tw2.forms.TableForm):
        title = tw2.forms.TextField(validator=tw2.core.Required)
        director = tw2.forms.TextField()
        genres = tw2.forms.CheckBoxList(options=['Action', 'Comedy', 'Romance', 'Sci-
↪fi'])

        class cast(tw2.forms.GridLayout):
            extra_reps = 5
            character = tw2.forms.TextField()
            actor = tw2.forms.TextField()


class MovieController(BaseController):
    @expose('myapp.templates.widget')
    def movie(self, *args, **kw):
        w = MovieForm(redirect='/movie/').req()
        return dict(widget=w, page='movie')
```

Add another new file `myapp/templates/widget.mak` with the contents:

```
<html>
<%inherit file="local:templates.master"/>

<%def name="title()">
  TurboGears 2 and ToscaWidgets 2, like jelly and jam with no bread:  Great!
</%def>

<body>
${widget.display() | n}
</body>
</html>
```

And open up the existing file `myapp/controllers/root.py` and add, just below the `from myapp.controllers.error import ErrorController` line:

```
from myapp.controllers.movie import MovieController
```

And just below the `error = ErrorController()` line:

```
movie = MovieController()
```

With those three file edits in place, you should be able to restart the application with `paster serve development.ini`/`gearbox serve` (there is a `--reload` option for convenience) and point your browser at http://localhost:8080/movie/movie.

The form does not look particularly appealing. To try to improve this, lets add some CSS. We'll start with something simple; create `myapp/public/css/myapp.css` with the following:

```
th {
    vertical-align: top;
```

(continues on next page)

(continued from previous page)

```
    text-align: left;
    font-weight: normal;
    padding: 3px;
}

ul {
    list-style-type: none;
}

.required th {
    font-weight: bold;
}

th label {
    font-weight: bold;
}

td label {
    display: inline;
}
```

Notice the use of the `required` class. TableForm applies this to rows that contain a field that is required.

Before TableForm will inject `myapp.css` into the page, we'll have to add it to the list of resources. Add the following to the top of the *MovieForm* class definition in `myapp/controllers/movie.py` just above the line `title = 'Movie'`:

```
resources = [tw2.core.CSSLink(link='/css/myapp.css')]
```

Restart `paster/gearbox` and browse to http://localhost:8080/movie/movie to see the new css in action.

Connecting to a Database

Note: Be aware that the following describes a different approach (using than the one recommended in the TurboGears documentation!

The next step is to save movies to a database. To do this, we'll use only SQLAlchemy (and not elixir as in the *Standalone Tutorial* tutorial). SQLAlchemy is built into TurboGears by default. Edit `myapp/config/app_config.py` and add near the top:

```
from tw2.core.middleware import ControllersApp as TW2ControllersApp
```

and add at the very bottom:

```
base_config.custom_tw2_config['controllers'] = TW2ControllersApp()
base_config.custom_tw2_config['controller_prefix'] = '/tw2_controllers/'
base_config.custom_tw2_config['serve_controllers'] = True
```

Next add a brand new file `myapp/model/movie.py` with the contents:

```
from sqlalchemy import Table, ForeignKey, Column
from sqlalchemy.types import Unicode, Integer
from sqlalchemy.orm import relation, backref
```

(continues on next page)

(continued from previous page)

```
from myapp.model import DeclarativeBase, metadata, DBSession

__all__ = ['Movie', 'Genre', 'Cast']

movie_genre_table = Table('movie_genre', metadata,
    Column('movie_id', Integer, ForeignKey('movies.id',
        onupdate='CASCADE', ondelete='CASCADE'), primary_key=True),
    Column('genre_id', Integer, ForeignKey('genres.id',
        onupdate='CASCADE', ondelete='CASCADE'), primary_key=True)
)


class Movie(DeclarativeBase):
    __tablename__ = 'movies'
    id = Column(Integer, primary_key=True)
    title = Column(Unicode(255))
    director = Column(Unicode(255))


class Genre(DeclarativeBase):
    __tablename__ = 'genres'
    id = Column(Integer, primary_key=True)
    name = Column(Unicode(255))
    movies = relation('Movie', secondary=movie_genre_table, backref='genres')

    def __unicode__(self):
        return unicode(self.name)


class Cast(DeclarativeBase):
    __tablename__ = 'casts'
    id = Column(Integer, primary_key=True)
    movie_id = Column(Integer, ForeignKey(Movie.id))
    movie = relation(Movie, backref=backref('cast'))
    character = Column(Unicode(255))
    actor = Column(Unicode(255))
```

Next edit `myapp/model/__init__.py` and uncomment the line that reads:

```
DeclarativeBase.query = DBSession.query_property()
```

and also add the following line to the very bottom of that file:

```
from myapp.model.movie import Movie, Genre, Cast
```

Edit `myapp/websetup/bootstrap.py` and add the following just inside the bootstrap function definition:

```
for name in ['Action', 'Comedy', 'Romance', 'Sci-fi']:
    model.DBSession.add(model.Genre(name=name))
transaction.commit()
```

And finally, get your controller ready to redirect everything as necessary. Edit `myapp/controllers/movie.py` and add to the very top:

```
import tw2.sqla
```

As well, change `class MovieForm(tw2.forms.FormPage):` to instead read:

```
class MovieForm(tw2.sqla.DbFormPage):
    entity = model.Movie
```

Just inside the definition of the child class (right above the `title =` line) add:

```
action = '/tw2_controllers/movie_submit'
id = tw2.forms.HiddenField()
```

And the last for the *MovieForm*, change `genres = tw2.forms.CheckBoxList( ... )` to:

```
genres = tw2.sqla.DbCheckBoxList(entity=model.Genre)
```

And (still in `myapp/controllers/movie.py`) inside the MovieController's movie method, just below the line `w = MovieForm(...` add the three lines:

```
w.fetch_data(request)
tw2.core.register_controller(w, 'movie_submit')
```

Now, in your command prompt run:

TurboGears<=2.2	TurboGears>=2.3
`$ paster setup-app development.ini`	`$ gearbox setup-app`

This will create and initialize your database in a sqlite DB.

We're almost done, but not quite. Nonetheless, this is a good point to restart your app and test to see if any mistakes have cropped up. Restart `paster`/`gearbox` and visit http://localhost:8080/movie/movie. Submit your first entry. It should give you an **Error 404**, but don't worry. Point your browser now to http://localhost:8080/movie/movie?id=1 and you should see the same movie entry that you just submitted.

Great – we can write to the database and read back an entry, now how about a list of entries?

Add a whole new class to `myapp/controllers/movie.py`:

```
class MovieIndex(tw2.sqla.DbListPage):
    entity = model.Movie
    title = 'Movies'
    newlink = tw2.forms.LinkField(link='/movie/movie', text='New', value=1)
    class child(tw2.forms.GridLayout):
        title = tw2.forms.LabelField()
        id = tw2.forms.LinkField(link='/movie/movie?id=$', text='Edit', label='Action
↪')
```

And add the following method to your *MovieController*:

```
@expose('myapp.templates.widget')
def index(self, **kw):
    w = MovieIndex.req()
    w.fetch_data(request)
    return dict(widget=w, page='movie')
```

Getting Fancy

And if we wanted to start getting fancy we could add:

```
<li class="${('', 'active')[page=='movie']}"><a href="${tg.url('/movie')}">Movies</a>
↪</li>
```

to the list of `<ul id="mainmenu"> ... </ul>` items in `myapp/templates/master.mak`.

We could also make things dynamic by editing `myapp/controllers/movie.py` and adding at the top:

```
import tw2.dynforms
```

replacing `class child(tw2.forms.TableForm):` with:

```
class child(tw2.dynforms.CustomisedTableForm):
```

and replacing:

```
class cast(tw2.forms.GridLayout):
    extra_reps = 5
```

with:

```
class cast(tw2.dynforms.GrowingGridLayout):
```

Getting Fancier

There are a lot of *non-core* TW2 widget libraries out there, and just to give you a taste, we'll use one to add one more view to our Movie app.

Edit `myapp/controllers/movie.py` and add the following to the top:

```
import tw2.jqplugins.jqgrid
```

Add the following class definition to the same file:

```
class GridWidget(tw2.jqplugins.jqgrid.SQLAjqGridWidget):
    id = 'grid_widget'
    entity = model.Movie
    excluded_columns = ['id']
    prmFilter = {'stringResult': True, 'searchOnEnter': False}
    pager_options = {'search': True, 'refresh': True, 'add': False, }
    options = {
        'url': '/tw2_controllers/db_jqgrid/',
        'rowNum': 15,
        'rowList': [15, 30, 50],
        'viewrecords': True,
        'imgpath': 'scripts/jqGrid/themes/green/images',
        'width': 900,
        'height': 'auto',
    }
```

And add the following method to the `MovieController` class:

```
@expose('myapp.templates.widget')
def grid(self, *args, **kw):
    tw2.core.register_controller(GridWidget, 'db_jqgrid')
    return dict(widget=GridWidget, page='movie')
```

Your template has already been loading the current jQuery library the whole time, but that would be causing us trouble now, since `tw2.jquery` also provides the library, even versioned. So you need to **delete** or comment the line that looks like follows from `myapp/templates/master.mak` and `myapp/templates/master.html`:

```
<script src="http://code.jquery.com/jquery.js"></script>
```

Redirect your browser to http://localhost:8080/movie/grid and you should see the sortable, searchable jQuery grid.

If you are using a different framework, try asking on the group.

1.3 Next Steps

This tutorial has demonstrated the basic concepts of ToscaWidgets 2. To further your knowledge, a good place to look is the widget browser. There is also comprehensive design documentation, which explains how the different parts of ToscaWidgets work.

1.4 Import Styles

The Python import statement is flexible and allows several styles to be used:

- The tutorials have import lines like `import tw2.forms`, which makes each use of an imported widget quite long-winded, e.g. `tw2.forms.TextField`.
- A common approach is to use `from tw2.forms import TextField`, which makes each usage more concise - just `TextField`. However, it's then necessary to add each widget that will be used to the import line.
- It is possible to use `from tw2.forms import *`. However, this is discouraged for several reasons, including that it makes it difficult for a reader of the code to tell where a widget has been imported from.
- An alternative approach is to use `import tw2.forms as twf`, with each use then being `twf.TextField`.

It is possible to use ToscaWidgets 2 with any of these. However, the latter form is favored, and documentation beyond the introductory tutorials will generally use that style.

CHAPTER 2

Design

2.1 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')
```

2.1.1 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

2.1.2 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.

2.2 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.

2.3 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 *Constructing Javascript from Python* for more information on how to set up your middleware for desired output.

2.4 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.

2.5 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:

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

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

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

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

class `tw2.core.`**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.

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
```

2.6 Constructing Javascript from Python

class tw2.core.**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_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_symbol**(*name=None*, *src=None*)

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

All together now

Consider the following prepare method:

```
def prepare(self):
    super(MyWidget, self).prepare()

    # Create a js object for "$(document).ready(.."
    when_ready = lambda f: twc.js_function('jQuery')(
        twc.js_symbol('document')
    ).ready(twc.js_callback(f))

    # Dicts and other primitives get translated to js properly.
    my_js_object = dict(foo="bar", hello="world")

    # This is the main function we want to execute
    payload = twc.js_function('console.log')(my_js_object)

    # Register it all with tw2's middleware for later injection
    self.add_call(when_ready(payload))
```

The above will add the following output to the bottom of the response:

```
<script type="text/javascript">
    jQuery(document).ready(function(){
        console.log({"foo": "bar", "hello": "world"})
    })
</script>
```

2.7 Middleware

The WSGI middleware has three functions:

- Request-local storage
- Configuration
- Resource injection

Configuration

In general, ToscaWidgets configuration is done on the middleware instance. At the beginning of each request, the middleware stores a reference to itself in request-local storage. So, during a request, a widget can consult request-local storage, and get the configuration for the middleware active in that request. This allows multiple applications to use ToscaWidgets, with different configurations, in a single Python environment.

Configuration is passed as keyword arguments to the middleware constructor. The available parameters are:

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: '')

2.8 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):
```

(continues on next page)

(continued from previous page)

```
    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.

2.9 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

2.10 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:
            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`.

2.10.1 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`.

2.10.2 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.

2.11 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
```

CHAPTER 3

tw2.devtools

To keep tw2.core as minimal as possible, features needed only for development are in a separate package, tw2.devtools. The features in devtools are:

- Widget browser
- Widget library quick start

3.1 Widget Browser

The browser essentially enumerates the `tw2.widgets` entrypoint. When browsing a module, it iterates through the public names in the module, and displays any that is a Widget subclass. It also imports `samples.py` for demo widgets. This can contain `page_options` - a dict that gives attributes for the body tag in the containing page.

The parameters that are displayed are: all the required parameters, plus non-required parameters that are defined on anything other than the Widget base class. Variables are never shown.

3.2 Widget Library Quick Start

To create a widget library, issue:

```
$ paster create -t tw2.library tw2.mylib
```

This creates an empty template that gets you started.

3.3 Writing a good widget library

Widget browser This is the main documentation for the library, and it needs to give a good introduction to a new user. Every widget should have a demo, and a clear description of the widget and parameters.

Example application There should be a simple example application that demonstrates the widgets in action. Ideally this should just be a single python file that works standalone.

Parameters Every widget should have a convenient set of parameters that allow common customisation with ease, and make more complex configuration possible.

Validation Every widget needs to work correctly with validation. When a form is submitted and there is a validation failure, all widgets should maintain the same appearance. This includes a widget with no value continuing to have no value.

Growing Every widget should work correctly within a tw2.dynforms Growing container.

Database interaction Some widgets (e.g. AjaxLookup, DataGrid) can benefit from built-in database interaction. In this case, the widget library must include a base widget that does no database interaction, so it can be connected to any database/ORM. The library should include a widget that interacts with SQLAlchemy, and may include a widget that interacts with any other database/ORM.

CHAPTER 4

What's New in ToscaWidgets 2?

ToscaWidgets 2 is a complete rewrite of ToscaWidgets. The primary motivation is to simplify the library, as the ToscaWidgets 1 code had become overly complex over time. The complexity made it difficult to write more complex widgets, such as tw.dynforms, and making changes to the library became a risky process. Where backwards compatibility can be readily maintained, this has been done. For example, the widgets in tw2.forms have almost identical names and parameters to tw.forms. However, in many cases it has been necessary to break backwards compatibility to produce a consistent and simple library.

4.1 Per-Request Widget Instances

A key feature for widgets is their dynamic capabilities, the ability to for a widget to adapt each time it is displayed. ToscaWidgets 2 creates a new instance of a widget every time it is used in a request. This allows widget and application code to update widget attributes in the natural, pythonic manner, without multiple threads interfering with each other. For example, you may want to customise a TableForm, so when it's "cost" field is over 100, the "insurance" field is made bold, to draw the user's attention. The code for this would be:

```
def prepare(self):
    super(MyWidget, self).prepare()
    if self.c.cost.value > 100:
        self.c.insurance.css_class = 'bold'
```

In ToscaWidgets 1, widget instances are singletons that exist for the life of the application. For thread safety, it's vital that their attributes are not modified during a request. This means that all dynamic parameters must be passed around as dictionaries, resulting in highly-complex code that is prone to bugs. The equivalent code would be:

```
def update_params(self, params):
    super(MyWidget, self).update_params(params)
    if params['value_for']('cost') > 100:
        params.setdefault('child_args', {}).setdefault('insurance', {})['css_class']␣
↪= 'bold'
```

4.2 Simplified Form Definitions

ToscaWidgets 2 places great emphasis on making the application code that defines forms as simple as possible. It allows a natural, declarative style; defining a form is as easy as:

```
class MovieForm(twf.TableForm):
    id = twf.HiddenField()
    title = twf.TextField()
```

For comparison, the equivalent code in ToscaWidgets 1 is:

```
class MovieForm(twf.TableForm):
    class children(twc.WidgetsList):
        id = twf.HiddenField()
        title = twf.TextField()
movie_form = MovieForm()
```

4.3 Widget Controller Methods

In ToscaWidgets 1, widgets are purely view components. However, ToscaWidgets 2 allows widgets to define controller methods, for example:

```
class PeopleLookupField(twy.AjaxLookupField):
    @classmethod
    def request(cls, req):
        return twc.encode(db.People.query.like('%'+req.GET.get('search')+'%'))
```

This allows Ajax widgets to be self-contained, rather than having functionality separated between the widget and a controller class.

Warning: You must include any required authorisation checks in the `request` method.

4.4 Built-in Validation

ToscaWidgets 1 uses FormEncode for validation. FormEncode is a popular validation library and works well for many use cases. However, ToscaWidgets 1 has considerable internal complexity to ensure that it interfaces correctly with FormEncode. In the past, this has been the cause of some subtle bugs related to complex forms. It also makes it difficult to correctly support validation for some complex widgets, such as HidingContainerMixin in tw.dynforms.

ToscaWidgets 2 has a built-in validation framework, which avoids much of the validation complexity in ToscaWidgets 1. It does not have as many validators as FormEncode, covering just the more common use cases. However, it is still possible to use FormEncode validators for individual fields (although not Schema or ForEach validators). This change has enabled validation support for GrowingGrid in tw2.dynforms. It also paves the way for future development of client-side validation.

4.5 Declarative Parameter Definitions

When you're creating your own widgets, you'll need to define parameters for the widgets, the variables that users of the widgets can set. You'll want to provide documentation, and default values. ToscaWidgets 2 makes this straightforward:

```
class MyWidget(twc.Widget):
    do_title = twc.Param('Whether to include a title row in the table', default=True)
```

For comparison, the equivalent code in ToscaWidgets 1 is:

```
class MyWidget(twc.Widget):
    params = {
        'do_title': 'Whether to include a title row in the table'
    }
    dotitle = True
```

The new approach used in ToscaWidgets 2 allows more metadata to be recorded about parameters, enabling new features, such as parameters that automatically become attributes.

4.6 Consistency in IDs and Names

ToscaWidgets 2 has changed how `id` and `name` parameters are generated. Just like ToscaWidgets 1, a widget's full ID is generated by combining it's ID with those of its ancestors. ToscaWidgets 1 uses underscores as the separator, which causes problems when applications use underscores in widget names. ToscaWidgets 2 uses colons as the separator, and forbids colons in widget names. In additions, a widget's full name is identical to its full id. These changes simplify the development of complex client-side widgets, such as GrowingGrid in tw2.dynforms.

4.7 Layouts Separated from Containers

ToscaWidgets 2 clearly separates the concept of form layouts and form containers. ToscaWidgets 1 combines these, so there are widgets for TableForm, ListForm, TableFieldSet and ListFieldSet. ToscaWidgets 2 has widgets for Form and FieldSet, and also for TableLayout and ListLayout. This enables more flexibility in defining new containers and layouts, and this enables the new GridLayout widget. For comptability, TableForm remains, which transparently converts to a Form widget containing a TableLayout, as do the other widgets from ToscaWidgets 1. This change affects tw2.dynforms, which now has GrowingGridLayout, instead of GrowingTableFieldSet and GrowingTableForm.

4.8 Explicitly Deferred Parameters

Sometimes it is desirable for parameters to be dynamically evaluated every time a widget is displayed. ToscaWidgets 1 automatically calls any parameter that is a callable, for example:

```
class MyWidget(twc.Widget):
    date = lambda: time.strftime('%d/%m/%Y')
```

However, in some cases parameters may be callables, but this behaviour is not desired. A common example is passing SQLAlchemy mapped classes to widgets. ToscaWidgets 2 only calls parameters that are explicitly marked as `Deferred`:

```
class MyWidget(twc.Widget):
    date = twc.Deferred(lambda: time.strftime('%d/%m/%Y'))
```

4.9 Variables in Widget Templates

In ToscaWidgets 2, the widget instance is available in the template as `$w`. Parameters must be accessed as `$w.param`. ToscaWidgets 1 made all parameters directly accessible as `$param`. The behaviour can be enabled in ToscaWidgets 2 by setting the *params_as_vars* config option.

4.10 ToscaWidgets as a Framework

ToscaWidgets 1 was always intended to be used with another web framework, primarily TurboGears and Pylons. ToscaWidgets 2 has gained features that allow it to be used as a framework in its own right. This is primarily the `Page` and `FormPage` widgets, which enable applications to be coded like this:

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

class Index(twf.FormPage):
    title = 'My app'
    class child(twf.TableForm):
        name = twf.TextField()
        email = twf.TextField(validator=twc.EmailValidator)

twc.dev_server()
```

4.11 Minor Differences

- Framework interfaces are almost completely removed; ToscaWidgets is just a piece of WSGI middleware.
- Widget constructions do not accept positional arguments, as doing so is considered bad practice when multiple inheritence is in use.
- A widget does not automatically get the `resources` from its base class.
- tw.api has been removed; just use tw2.core
- The toscawidgets simple template engine has been removed.
- Widget.__call__ is no longer an alias for display, as this causes problems for Cheetah.
- CalendarDatePicker is moved from tw2.forms to tw2.dynforms

In tw2.dynforms:

- WriteOnlyTextField is removed; tw2.forms PasswordField has similar functionality
- AjaxLookupField is removed; there are better widgets like this in libraries like YUI

4.12 ToscaWidgets 1

Python web widgets were pioneered in TurboGears and many of the key ideas remain. Once the value of widgets was realised, a move was made to create a separate library, this is ToscaWidgets. The key differences are:

- ToscaWidget is framework independent.
- Multiple template engines are supported.
- Resource links are injected by rewriting the page on output.

- The forms library is separate from the core widget library.
- The tw namespace exists for widget libraries to be located in.

ToscaWidgets had some success, but did not gain as much usage as hoped, in part due to a lack of documentation in the beginning.

CHAPTER 5

Changelog

5.1 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

5.2 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

5.3 2.2.2

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

5.4 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

5.5 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

5.6 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

5.7 2.2.0.6

5.8 2.2.0.5

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

5.9 2.2.0.4

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

5.10 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

5.11 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

5.12 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

5.13 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

5.14 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

5.15 2.1.5

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

5.16 2.1.4

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

5.17 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 6

Indices and tables

- genindex
- modindex
- search

Index