wheezy.web documentation

Release latest

Andriy Kornatskyy

Apr 17, 2021

Contents

1 Introduction **1**

2 Contents **3**

2.1 Getting Started 3
2.2 Examples 3
2.3 Tutorial 6
2.4 User Guide 21
2.5 Modules 36

CHAPTER 1

Introduction

wheezy.web is a lightweight, high performance, high concurrency WSGI web framework with the key features to *build modern, efficient web*:

- Requires Python 2.4-2.7 or 3.2+.
- MVC architectural pattern (push-based).
- Functionality includes routing, model update/validation, authentication/authorization, content caching with dependency, xsrf/resubmission protection, AJAX+JSON, i18n (gettext), middlewares, and more.
- Template engine agnostic (integration with jinja2, mako, tenjin and wheezy.template) plus html widgets.

It is optimized for performance, well tested and documented.

Resources:

- source code, examples and issues tracker are available on github
- documentation

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Install

wheezy.web requires python version 3.6+. It is independent of operating system. You can install it from pypi:

```
$ pip install wheezy.web
```

2.2 Examples

Since *wheezy.web* is template engine agnostic, you need specify extra requirements (per template engine of your choice):

```
$ pip install wheezy.web[jinja2]
$ pip install wheezy.web[mako]
$ pip install wheezy.web[tenjin]
$ pip install wheezy.web[wheezy.template]
```

2.2.1 Templates

Template application serves template purpose for you. It includes:

- Integration with both mako and tenjin template system.
- User registration and authentication.
- Form validation.

If you are about to start a new project it is a good starting point.

2.2.2 Hello World

hello.py shows you how to use *wheezy.web* in a pretty simple WSGI application. It no way pretend to be shortest possible and absolutely not magical:

```
""" Minimal helloworld application.
"""

from wheezy.http import HTTPResponse, WSGIApplication
from wheezy.routing import url

from wheezy.web.handlers import BaseHandler
from wheezy.web.middleware import (
    bootstrap_defaults,
    path_routing_middleware_factory,
)


class WelcomeHandler(BaseHandler):
    def get(self):
        response = HTTPResponse()
        response.write("Hello World!")
        return response


def welcome(request):
    response = HTTPResponse()
    response.write("Hello World!")
    return response


all_urls = [
    url("", WelcomeHandler, name="default"),
    url("welcome", welcome, name="welcome"),
]


options = {}
main = WSGIApplication(
    middleware=[
        bootstrap_defaults(url_mapping=all_urls),
        path_routing_middleware_factory,
    ],
    options=options,
)


if __name__ == "__main__":
    from wsgiref.simple_server import make_server

    try:
        print("Visit http://localhost:8080/")
        make_server("", 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
    print("\nThanks!")
```

Handler Contract

Let have a look through each line in this application. First of all let take a look what is a handler:

```
def welcome(request):
    response = HTTPResponse()
    response.write("Hello World!")
    return response
```

This one is not changed from what you had in wheezy.http so you are good to keep it minimal. However there is added another one (that actually implements the same handler contract internally):

```
class WelcomeHandler(BaseHandler):
    def get(self):
        response = HTTPResponse()
        response.write("Hello World!")
        return response
```

What is `get` method here? It is your response to HTTP GET request. You have post for HTTP POST, etc.

Routing

Routing is inherited from wheezy.routing. Note that both handlers are working well together:

```
all_urls = [
    url("", WelcomeHandler, name="default"),
    url("welcome", welcome, name="welcome"),
]
```

Application

`WSGIApplication` is coming from wheezy.http. Integration with wheezy.routing is provided as middleware factory (`path_routing_middleware_factory()`):

```
options = {}
main = WSGIApplication(
    middleware=[
        bootstrap_defaults(url_mapping=all_urls),
        path_routing_middleware_factory,
    ],
    options=options,
)
```

Functional Tests

You can easily write functional tests for your application using `WSGIClient` from wheezy.http (file test_hello.py).

```
from hello import main
from wheezy.http.functional import WSGIClient


```

(continues on next page)

(continued from previous page)

```
class HelloTestCase(unittest.TestCase):
    def setUp(self):
        self.client = WSGIClient(main)

    def tearDown(self):
        del self.client
        self.client = None

    def test_home(self):
        """Ensure welcome page is rendered."""
        assert 200 == self.client.get("/")
        assert "Hello World!" == self.client.content

    def test_welcome(self):
        """Ensure welcome page is rendered."""
        assert 200 == self.client.get("/welcome")
        assert "Hello World!" == self.client.content
```

For more advanced use cases refer to wheezy.http documentation, please.

Benchmark

You can add benchmark of your functional tests (file benchmark_hello.py):

```
from wheezy.core.benchmark import Benchmark


class BenchmarkTestCase(HelloTestCase):
    """
    ../../env/bin/nosetests-2.7 -qs -m benchmark benchmark_hello.py
    """

    def runTest(self):  # noqa: N802
        """Perform bachmark and print results."""
        p = Benchmark((self.test_welcome, self.test_home), 20000)
        p.report("hello", baselines={"test_welcome": 1.0, "test_home": 0.9})
```

Let run benchmark tests with `nose` (to be run from `demos/hello` directory):

```
$ ../../env/bin/nosetests-2.7 -qs -m benchmark benchmark_hello.py
```

Here is output:

```
hello: 2 x 20000
baseline throughput change target
  100.0%   11518rps  +0.0% test_welcome
   91.0%   10476rps  +1.1% test_home
----------------------------------------------------------------------
Ran 1 test in 3.686s
```

2.3 Tutorial

This tutorial will teach you the basics of building a *wheezy.web* application using your favorite text editor and python. We will use SQLite as database and python version 2.6+ or 3.2 (mainly for context manager and built-in JSON

support). *AJAX and JSON* section of tutorial require jQuery.

Estimated completion time: 30-60 minutes.

2.3.1 Prerequisites

Before you start, make sure you've installed the prerequisites listed below.

- Check python version:

```
$ python -V
Python 2.7.3
```

- Create virtual environment:

```
$ virtualenv env
```

- Install *wheezy.web* into virtual environment:

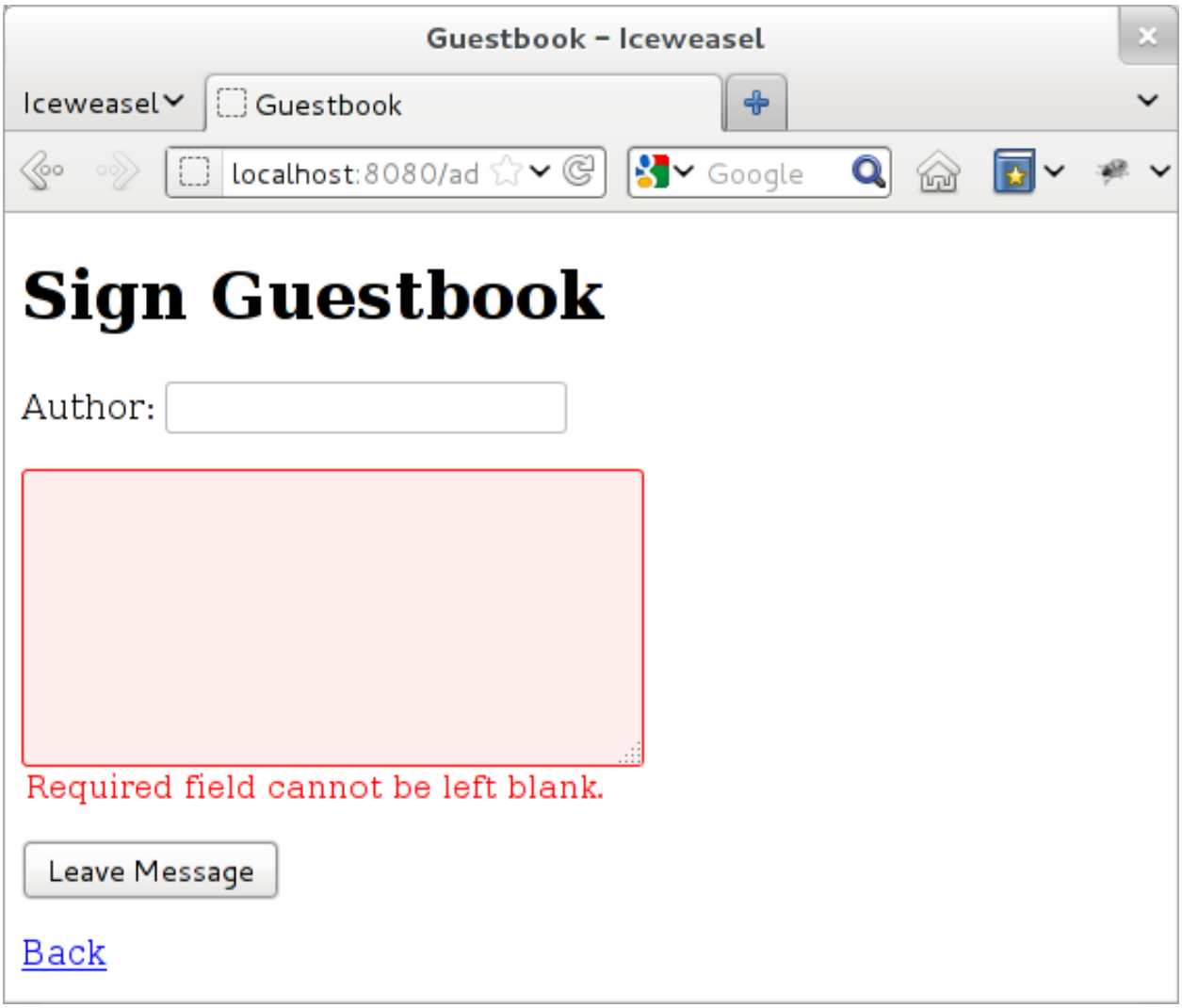
```
$ env/bin/easy_install wheezy.web
```

2.3.2 What You'll Build

You will implement a simple guestbook application where users can see a list of greetings as well as add their own.

List of greetings:

Sign guestbook:



For the purpose of this tutorial we store each of identified software actor in its own file so at the end you will get a project structure with well defined roles.

2.3.3 Domain Model

The domain model represents key concepts of entities within a scope of the application. Our primary entity is a greeting that visitor leave in guestbook, it can be characterized by the following: a time stamp when it was added (current time), an author and a message.

Let's model what we figured so far (file `models.py`):

```
from datetime import datetime

class Greeting(object):

    def __init__(self, id=0, created_on=None, author='', message=''):
        self.id = id
        self.created_on = created_on or datetime.now()
        self.author = author
        self.message = message
```

2.3.4 Validation Rules

Two attributes `author` and `message` are entered by visitor so we need apply some validation rules:

- `author` can be left blank (for anonymous entries) but if it is entered it should not exceed 20 characters in length.
- `message` is required and let take that anything meaningful can be expressed in a text between 5 to 512 characters.

So far so good, let's define our application domain validation constraints (file `validation.py`):

```
from wheezy.validation import Validator
from wheezy.validation.rules import length
from wheezy.validation.rules import required

greeting_validator = Validator({
    'author': [length(max=20)],
    'message': [required, length(min=5, max=512)],
})
```

For the complete list of validation rules available, please refer to wheezy.validation documentation.

2.3.5 Database

For the purpose of this tutorial we have selected SQLite database as persistence layer so let define SQL schema for our domain (file `schema.sql`):

```
CREATE TABLE greeting (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    created_on TIMESTAMP NOT NULL,
    author TEXT,
    message TEXT NOT NULL
);
```

Issue the following command from the terminal:

```
$ cat schema.sql | sqlite3 guestbook.db
```

This creates an SQLite database `guestbook.db` with table `greeting`.

Let's try to add some data from the sqlite3 command prompt:

```
$ sqlite3 guestbook.db
SQLite version 3.7.16.2 2013-04-12 11:52:43
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> INSERT INTO greeting (created_on, author, message)
   ...> VALUES ('2012-03-01 13:50:27', 'John Smith', 'This looks cool!');
sqlite> SELECT * FROM greeting;
1|2012-03-01 13:50|John Smith|This looks cool!
sqlite> .quit
```

We will use these two basic SQL statements (SELECT and INSERT) in repository.

Configuration

Let add configuration file where we can store some settings (file `config.py`):

```
import sqlite3

def session():
    return sqlite3.connect('guestbook.db',
                           detect_types=sqlite3.PARSE_DECLTYPES)
```

We have defined function `session()` that returns an object valid to issue some database related operations including query for data, transaction commit, etc. This object serves the *unit of work* purpose and is suitable to be used with python context manager.

2.3.6 Repository

A Repository mediates between the domain and persistence layers (database, file, in-memory storage, etc.), it encapsulates operations performed and provides object-oriented view of the persistence layer.

Accordingly to the problem statement, we need two things here: a way to get a list of greetings and ability to add a greeting.

Since we have a database and a way to obtain database objects we can add repository (file `repository.py`):

```
from models import Greeting

class Repository(object):

    def __init__(self, db):
        self.db = db

    def list_greetings(self):
        cursor = self.db.execute("""
                SELECT id, created_on, author, message
                FROM greeting
                ORDER BY id DESC
                LIMIT 10
        """)
        return [Greeting(
                id=row[0],
                created_on=row[1],
                author=row[2],
                message=row[3]) for row in cursor.fetchall()]

    def add_greeting(self, greeting):
        self.db.execute("""
                INSERT INTO greeting (created_on, author, message)
                VALUES (?, ?, ?)
        """, (greeting.created_on, greeting.author, greeting.message))
        return True
```

Let's see how it works from python command prompt:

```
$ env/bin/python
Python 2.7.3 (default, Mar  5 2013, 01:19:40)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

(continues on next page)

(continued from previous page)

```
>>> from config import session
>>> from repository import Repository
>>> db = session()
>>> repo = Repository(db)
>>> greetings = repo.list_greetings()
>>> greetings[0]
<models.Greeting object at 0xa023e4c>
>>> greetings[0].created_on
datetime.datetime(2012, 3, 1, 13, 50, 27)
>>> db.close()
>>> exit()
```

2.3.7 View

Handlers

Views contain handlers that respond to requests sent by a browser. We need two handlers: one for list and the other one to add a greeting.

List handler returns a list of greeting stored (file `views.py`):

```
from wheezy.web.handlers import BaseHandler
from config import session
from models import Greeting
from repository import Repository
from validation import greeting_validator

class ListHandler(BaseHandler):

    def get(self):
        with session() as db:
            repo = Repository(db)
            greetings = repo.list_greetings()
        return self.render_response('list.html',
                greetings=greetings)
```

We create a *unit of work* by applying function call to `session` and add it to a scope of python operator `with` (which effectively closes our unit of work when execution leaves this scope). `session` is closed before we pass anything to template render.

Add handler store visitor greeting (file `views.py`):

```
class AddHandler(BaseHandler):

    def get(self, greeting=None):
        greeting = greeting or Greeting()
        return self.render_response('add.html', greeting=greeting)

    def post(self):
        greeting = Greeting()
        if (not self.try_update_model(greeting)
                or not self.validate(greeting, greeting_validator)):
            return self.get(greeting)
        with session() as db:
            repo = Repository(db)
```

(continues on next page)

(continued from previous page)

```
            if not repo.add_greeting(greeting):
                self.error('Sorry, can not add your greeting.')
                return self.get(greeting)
            db.commit()
        return self.see_other_for('list')
```

The respond to browser request to add handler is simply render `add.html` template with some defaults passed with greeting model. However when visitor submits 'add page' we try update model `greeting` with HTML form data. If it fails for any reason we display user error messages (those returned by `try_update_model()`). If update model succeeds it holds data entered by user that we can validate with `greeting_validator`. Note `BaseHandler` keeps a dictionary of all errors reported in `errors` attribute. Again if validation fails we redisplay `add` page with any errors reported.

When input is considered valid per all possible checks we create a unit of work from session and add it to `with` operator scope. Again, operation in repository may fail so we check if fails we add a general error so user can see it, otherwise we commit changes to unit of work and redirect user to list handler.

Configuration

wheezy.web is agnostic to template render. However it integrates with jinja2, mako, tenjin and wheezy.template. For purpose of this tutorial wheezy.template has been selected:

```
$ env/bin/easy_install wheezy.template
```

Let add wheezy.template configuration (file `config.py`):

```
from wheezy.html.ext.template import WidgetExtension
from wheezy.html.utils import html_escape
from wheezy.template.engine import Engine
from wheezy.template.ext.core import CoreExtension
from wheezy.template.loader import FileLoader
from wheezy.web.templates import WheezyTemplate

options = {}

# Template Engine
searchpath = ['templates']
engine = Engine(
    loader=FileLoader(searchpath),
    extensions=[
        CoreExtension(),
        WidgetExtension(),
    ])
engine.global_vars.update({
    'h': html_escape
})
options.update({
    'render_template': WheezyTemplate(engine)
})
```

Above configuration says that templates can be found in `templates` directory and we are using several extensions and helpers from wheezy.html.

Layout

Since templates usually have many things in common let's define common layout used by both pages we are going to create (create directory `templates` and add file `layout.html`):

```
@require(path_for)
<html>
    <head>
        <title>Guestbook</title>
        <link href="@path_for('static', path='site.css')"
            type="text/css" rel="stylesheet" />
    </head>
    <body>
        <div id="main">
            @def content():
            @end
            @content()
        </div>
    </body>
</html>
```

You need to be explicit about any context variable used in the template by specifying them in a `@require` directive.

Templates

Define template for list handler (in directory `templates` add file `list.html`):

```
@extends("layout.html")

@def content():
@require(path_for, greetings)
<h1>Guestbook</h1>
<a href="@path_for('add')">Sign guestbook</a>
@for g in greetings:
<p>
    @g.id!s. On @g.created_on.strftime('%m/%d/%Y %I:%M %p'),
    <b>@str(g.author or 'anonymous')</b> wrote:
    <blockquote>@g.message.replace('\n', '<br/>')</blockquote>
</p>
@end
@end
```

What is interesting here is `path_for()` function that can build reverse path for given route name. So when someone clicks on `Sign guestbook` link the browser navigates to a url that lets add a greeting.

Define template for add handler (in directory `templates` add file `add.html`):

```
@extends("layout.html")

@def content():
@require(greeting, path_for, errors)
<h1>Sign Guestbook</h1>
@greeting.error()
<form action="@path_for('add')" method='post'>
    <p>
        @greeting.author.label('Author:')
        @greeting.author.textbox()
```

(continues on next page)

(continued from previous page)

```
        @greeting.author.error()
    </p>
    <p>
        @greeting.message.textarea()
        @greeting.message.error()
    </p>
    <p>
    <input type='submit' value='Leave Message'>
    </p>
</form>
<a href="@path_for('list')">Back</a>
```

Here you can see syntax provided by wheezy.html for HTML rendering: label, textbox, error, etc. HTML widgets require context variable `errors`. Please refer to the wheezy.html documentation.

Style

Let's add some style (create directory `static` and add file `site.css`):

```
input[type="text"], textarea {
    border: 1px solid #BBB; border-radius: 3px; }
input.error, textarea.error {
    border: 1px solid #FF0000; background-color: #FFEEEE; }
span.error { color: #FF0000; display: block; font-size: 0.95em;
    background: transparent 0px 2px no-repeat; text-indent: 2px; }
span.error-message {
    display: block; padding: 25px 25px 25px 80px; margin: 0 0 15px 0;
    border: 1px solid #DFDFDF; color: #333333; font-size: 13px;
    line-height: 17px; float: none; font-weight: normal;
    width: auto; -moz-border-radius:5px 5px 5px 5px; }
span.error-message { border:1px solid #C44509;
    background: no-repeat scroll 2px 50% #fdcea4; }
```

2.3.8 URLs

URLs tell how browser requests maps to handlers that ultimately process them. Let map the root path to list handler and `add` path to add handler (file `urls.py`):

```
from wheezy.routing import url
from wheezy.web.handlers import file_handler
from views import AddHandler
from views import ListHandler

all_urls = [
    url('', ListHandler, name='list'),
    url('add', AddHandler, name='add'),
    url('static/{path:any}',
        file_handler(root='static/'),
        name='static')
]
```

Note each url mapping has a unique name, so it can be easily referenced by function that build reverse path for given name or perform request redirect.

2.3.9 Application

Let's define an entry point for guestbook application that combines all together (file `app.py`):

```
from wheezy.http import WSGIApplication
from wheezy.web.middleware import bootstrap_defaults
from wheezy.web.middleware import path_routing_middleware_factory

from config import options
from urls import all_urls


main = WSGIApplication([
    bootstrap_defaults(url_mapping=all_urls),
    path_routing_middleware_factory
], options)

if __name__ == '__main__':
    from wsgiref.handlers import BaseHandler
    from wsgiref.simple_server import make_server
    try:
        print('Visit http://localhost:8080/')
        BaseHandler.http_version = '1.1'
        make_server('', 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
    print('\nThanks!')
```

Try to run the application by issuing the following command:

```
$ env/bin/python app.py
```

Visit http://localhost:8080/ to see your site in a browser.

2.3.10 AJAX and JSON

AJAX and JSON significantly minimize HTTP traffic between web browser and server thus allow you save bandwidth and serve more clients.

In this tutorial we will display validation errors using AJAX + JSON and fallback to regular HTML rendering is case browser has JavaScript disabled for some reason.

Add changes to `views.py`:

```
class AddHandler(BaseHandler):

    ...

    def post(self):
        greeting = Greeting()
        if (not self.try_update_model(greeting)
                or not self.validate(greeting, greeting_validator)):
            if self.request.ajax:
                return self.json_response({'errors': self.errors})
            return self.get(greeting)
        ...
```

What we added here is check if the current request is AJAX request and if so we return JSON response with errors reported:

```
if self.request.ajax:
    return self.json_response({'errors': self.errors})
```

Now we need some JavaScript code to:

- submit HTML form via AJAX
- display errors
- correctly handle redirect response

Create a new file `site.js` and place it in `static` directory with the following content (we will be using jQuery):

```
String.prototype.format = function() {
    var args = arguments;
    return this.replace(/\{\d+\}/g, function(capture) {
        return args[capture.match(/\d+/)];
    });
}

function JSONForm(data, form) {
    $(form).prev('span.error-message').remove();
    $('span.error', form).remove();
    $('.error', form).removeClass('error');
    $.each(data.errors, function(key, value) {
        if (key == '__ERROR__') {
            form.before('<span class="error-message">{0}</span>'.format(
                    value.pop()))
        }
        else {
            key = key.replace(/_/g, '-');
            $('label[for="{0}"]'.format(key), form).addClass('error')
            var field = $('#' + key, form);
            field.addClass('error');
            field.after('<span class="error">{0}</span>'.format(
                value.pop()));
        }
    });
}

function ajaxForm(selector, dataType) {
    if (!dataType) dataType = 'json'
    $(selector || 'input[type="submit"]').live('click', function(e) {
        submit = $(this);
        submit.attr('disabled', 'disabled');
        var form = submit.parents('form:first');
        var data = null;
        if (this.name) {
            data = form.serializeArray();
            data.push({name: this.name, value: ''});
            data = $.param(data);
        }
        else
            data = form.serialize();
        $.ajax({
```

(continues on next page)

(continued from previous page)

```
            type: form.attr('method') || 'get',
            url: form.attr('action'),
            data: data,
            dataType: dataType,
            success: function(data, textStatus, jqXHR) {
                if (jqXHR.status == 207) {
                    window.location.replace(jqXHR.getResponseHeader('Location'));
                } else if (data.see_other) {
                    window.location.replace(data.see_other);
                } else if (dataType == 'json'){
                    submit.removeAttr('disabled');
                    JSONForm(data, form);
                }
            }
        });
        return false;
    });
}
```

Open `layout.html` and add link to jQuery library and `site.js` somewhere within head HTML tag:

```
<head>
    ...
    <script type="text/javascript"
    src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js">
    </script>
    <script type="text/javascript"
    src="@path_for('static', path='site.js')">
    </script>
</head>
```

Add the following to `add.html` to create a javascript AJAX form:

```
<script type="text/javascript">
    $(document).ready(function() {
        ajaxForm();
    })
</script>
```

Try to run the application by issuing the following command:

```
$ env/bin/python app.py
```

Visit http://localhost:8080/ to see your site in a browser (try both with JavaScript enabled and disabled).

2.3.11 Content Cache

Why would we be making a call to database every time the list of greetings is displayed to user? What if we can cache that page for some period of time and regenerate it only when someone added another greeting? Let's implement this use case with the wheezy.caching package.

Open `config.py` and add import for MemoryCache and Cached:

```
from wheezy.caching.memory import MemoryCache
```

At the end of `config.py` add initialization logic for cache, cache factory and configuration options for HTTP cache middleware:

```
cache = MemoryCache()

# HTTPCacheMiddleware
options.update({
    'http_cache': cache
})
```

Since we are going to use HTTP cache middleware we need to instruct the application bootstrap process about the middleware we are going to use. Open file `app.py` and import `http_cache_middleware_factory`:

```
from wheezy.http.middleware import http_cache_middleware_factory
```

To the list of `WSGIApplication` middleware, add a HTTP cache middleware factory:

```
main = WSGIApplication([
    bootstrap_defaults(url_mapping=all_urls),
    http_cache_middleware_factory,
    path_routing_middleware_factory
], options)
```

Finally let's apply cache profile to the ListHandler. Add a few imports (`views.py`):

```
from datetime import timedelta

from wheezy.http import CacheProfile
from wheezy.web import handler_cache
```

Use the `handler_cache` decorator to apply cache profile to the handler response:

```
class ListHandler(BaseHandler):

    @handler_cache(CacheProfile('server', duration=timedelta(minutes=15)))
    def get(self):
        ...
```

The `ListHandler` response is cached by server for 15 minutes.

Try to run the application by issuing the following command:

```
$ env/bin/python app.py
```

Visit http://localhost:8080/ to see your site in a browser. Try to add a greeting, and notice that the list page is not updated (it is being cached by server). Next we will use cache dependency to invalidate content cache.

Take a look at wheezy.http for various options available for content caching.

2.3.12 Cache Dependency

Let's add cache invalidation logic, so once user enters a new greeting it causes the list page to be refreshed.

In file `config.py` add import for `Cached`:

```
from wheezy.caching.patterns import Cached
```

Declare cached (right after the created cache instance):

```
cache = MemoryCache()
cached = Cached(cache, time=15 * 60)
```

Modify `ListHandler` so it is aware about the list cache dependency key:

```
class ListHandler(BaseHandler):

    @handler_cache(CacheProfile('server', duration=timedelta(minutes=15)))
    def get(self):
        ...
            greetings = repo.list_greetings()
        response = self.render_response('list.html',
                greetings=greetings)
        response.cache_dependency = ('d_list', )
        #response.cache_dependency.append('d_list')
        return response
```

Finally let's add a trigger, that causes the invalidation to occur in cache. Import cached from config module:

```
from config import cached
```

Modify `AddHandler` so that, on successful commit, the content cache for `ListHandler` response is invalidated:

```
class AddHandler(BaseHandler):
    ...
    def post(self):
        ...
            db.commit()
        cached.dependency.delete('d_list')
        return self.see_other_for('list')
```

Try to run the application by issuing the following command:

```
$ env/bin/python app.py
```

Visit http://localhost:8080/ to see your site in a browser. Try add a greeting and notice that list page is refreshed this time.

Take a look at wheezy.caching for various cache implementations including distributed cache support.

2.3.13 Cache Vary

AJAX + JSON, content caching and cache dependency are a great way to boost application performance. How about content compression? That is another great option to save traffic. What if we were able cache compressed response thus we will save on server CPU as well. Let implement this use case.

Transforms are used to manipulate handler response according to some algorithm. We will use this feature to compress response right before it enters content cache.

Add imports in file `views.py`:

```
from wheezy.http.transforms import gzip_transform
from wheezy.web.transforms import handler_transforms
```

Let's apply compression to `ListHandler`:

```
class ListHandler(BaseHandler):

    @handler_cache(CacheProfile('server', duration=timedelta(minutes=15)))
    @handler_transforms(gzip_transform(compress_level=9, min_length=250))
    def get(self):
        ...
```

Notice `handler_transforms()` decorator is after handler cache, this way it is able to compress response before it goes to the cache.

At this point we have a single version of the cached page - compressed. What about browsers that do not accept gzip content encoding? It would be good somehow to distinguish between web requests that support compression and those that do not. Fortunately browsers send an HTTP header `Accept-Encoding` that serves exactly this purpose. All we need is instruct content cache to *vary* response depending on value in `Accept-Encoding` HTTP header.

Instruct `ListHandler` cache profile to vary response by `Accept-Encoding` HTTP request header:

```
class ListHandler(BaseHandler):

    @handler_cache(CacheProfile('server', duration=timedelta(minutes=15),
        vary_environ=['HTTP_ACCEPT_ENCODING']))
    @handler_transforms(gzip_transform(compress_level=9, min_length=250))
    def get(self):
        ...
```

Notice we added `vary_environ` and used WSGI environment variable `HTTP_ACCEPT_ENCODING` to be included into cache key used by content cache.

We can apply more permissive content caching to `AddHandler`:

```
class AddHandler(BaseHandler):

    @handler_cache(CacheProfile('both', duration=timedelta(hours=1),
            vary_environ=['HTTP_ACCEPT_ENCODING'],
            http_vary=['Accept-Encoding']))
    @handler_transforms(gzip_transform(compress_level=9, min_length=500))
    def get(self, greeting=None):
        ...
```

Notice that for HTTP caching we added `http_vary` directive, so intermediate proxies can properly serve cached content.

Try to run the application by issuing the following command:

```
$ env/bin/python app.py
```

Visit http://localhost:8080/ to see your site in a browser.

Take a look at wheezy.http for various options available for content caching.

2.3.14 Exercises

1. Refactor views by moving the cache profiles definition to a separate file (e.g. profile.py)
2. Refactor repository by enforcing contract with duck typing asserts. See post and example.
3. Refactor repository by introducing caching repository implementation (use factory to provide repository, see caching.py and factory.py).

4. Enhance content caching for list handler by utilizing HTTP ETag browser caching (see membership cache profile in profile.py).
5. Improve templates with preprocessor (see examples for preprocessor and config.py).

2.4 User Guide

wheezy.web is a lightweight WSGI framework that aims take most benefits out of standard python library and serves a sort of glue with other libraries. It can be run from python 2.4 up to the most cutting edge python 3. The framework aims to alleviate the overhead associated with common activities performed in Web application development.

wheezy.web framework follows the model–view–controller (MVC) architectural pattern to separate the data model from the user interface. This is considered a good practice as it modularizes code, promotes code reuse.

wheezy.web framework follows a push-based architecture. Handlers do some processing, and then "push" the data to the template layer to render the results.

2.4.1 Web Handlers

Handler is any callable that accepts an instance of `HTTPRequest` and returns `HTTPResponse`:

```
def handler(request):
    return response
```

wheezy.web comes with the following handlers:

- `MethodHandler` - represents the most generic handler. It serves dispatcher purpose for HTTP request method (GET, POST, etc). Base class for all handlers.
- `BaseHandler` - provides methods that integrates such features as: routing, i18n, model binding, template rendering, authentication, xsrf/resubmission protection.
- `RedirectRouteHandler` - redirects to given route name.
- `FileHandler` - serves static files out of some directory.
- `TemplateHandler` - serves templates that don't require up front data processing.

MethodHandler

wheezy.web routes incoming web request to handler per url mapping (it uses wheezy.routing for this purpose):

```
all_urls = [
    url("", WelcomeHandler, name="default"),
    url("welcome", welcome, name="welcome"),
]
```

You subclass from `MethodHandler` or `BaseHandler` and define methods `get()` or `post()` that handle HTTP request methods `GET` or `POST`.

```
class WelcomeHandler(BaseHandler):
    def get(self):
        response = HTTPResponse()
        response.write("Hello World!")
        return response
```

This method must return an `HTTPResponse` object.

`MethodHandler` has a number of useful attributes:

- `options` - a dictionary of application configuration options.
- `request` - an instance of `wheezy.http.HTTPRequest`.
- `route_args` - a dictionary of arguments matched in url routing.
- `cookies` - a list of cookies that extend `HTTPResponse`.

Please note that this handler automatically responds with HTTP status code 405 (method not allowed) in case the requested HTTP method is not overridden in your handler, e.g. there is incoming POST request but your handler does not provide an implementation.

BaseHandler

`BaseHandler` provides methods that integrates such features as:

1. routing
2. AJAX
3. i18n
4. model binding
5. JSON
6. template rendering
7. authentication
8. authorization
9. xsrf/resubmission protection
10. context sharing

You need to inherit from this class and define get() and/or post() to be able respond to HTTP requests. This class inherits from `MethodHandler`, so everything mentioned for `MethodHandler` applies to `BaseHandler` as well.

Routing

Routing feature is provided via integration with wheezy.routing package. There are the following methods:

- `path_for(name, **kwargs)` - returns url path by route name. Any missing parameters are obtained from that current route.
- `absolute_url_for(name, **kwargs)` - returns absolute url for the given route name by combining current request with route information.
- `redirect_for(name, **kwargs)` - returns redirect found response (HTTP status code 302) by route name.
- `see_other_for(name, **kwargs)` - returns see other redirect response (HTTP status code 303) by route name.

All these methods support the following arguments:

- `name` - a name of the route.
- `kwargs` - extra arguments necessary for routing.

Please refer to wheezy.routing documentation for more information.

AJAX

Both redirects `redirect_for` and `see_other_for` understands AJAX requests and change HTTP status code to 207 while preserving HTTP header `Location`.

Browsers incorrectly handle redirect response to ajax request, so there is used HTTP status code 207 that javascript is capable to receive and process browser redirect. Here is an example for jQuery (see file core.js):

```
$.ajax({
    // ...
    success: function(data, textStatus, jqXHR) {
        if (jqXHR.status == 207) {
            window.location.replace(
                jqXHR.getResponseHeader('Location'));
        } else {
            // ...
        }
    }
});
```

If AJAX response status code is 207, browser navigates to URL specified in HTTP response header `Location`.

Please refer to wheezy.http documentation for more information.

Internationalization

Internationalization feature is provided via integration with wheezy.core package (module `i18n`). There are the following attributes:

- `locale` - default implementation return a value resolved from route arguments, particularly to name `locale`.
- `translations` - returns `TranslationsManager` (wheezy.core feature) for the current locale.
- `translation` - returns translations for the current handler. Default implementation return `NullTranslations` object. Your application handler must override this attribute to provide valid `gettext` translations.

Here is example from template demo application:

```
class SignInHandler(BaseHandler):

    @attribute
    def translation(self):
        return self.translations['membership']
```

This code loads *membership* translations from i18n directory. In order to function properly the following configuration options must be defined:

```
from wheezy.core.i18n import TranslationsManager

options = {}
options['translations_manager'] = TranslationsManager(
            directories=['i18n'],
            default_lang='en')
```

See example in public demo application config.py.

Model Binding

Once the html form is submitted, you need a way to bind these values to some domain model, validate, report errors, etc. This is where integration with wheezy.validation package happens.

There are the following attributes and methods:

- `errors` - a dictionary where each key corresponds to attribute being validated and value to a list of errors reported.
- `try_update_model(model, values=None)` - tries update domain `model` with `values`. If `values` is not specified it is the same as using `self.request.form`. You can pass here `self.request.query` or `self.route_args`.
- `ValidationMixin::validate(model, validator)` - shortcut for domain `model` validation per `validator`.
- `ValidationMixin::error(message)` - adds a general error (this error is added with key *__ERROR__*).

Here is an example from the template demo application (see file membership/web/views.py):

```
class SignInHandler(BaseHandler):

    def get(self, credential=None):
        if self.principal:
            return self.redirect_for('default')
        credential = credential or Credential()
        return self.render_response('membership/signin.html',
                self.widgets(credential=credential))

    def post(self):
        credential = Credential()
        if (not self.try_update_model(credential)
            or not self.validate(credential, credential_validator)):
            return self.get(credential)
        return self.redirect_for('default')
```

On POST this handler updates `credential` with values from the html form submitted. In case `try_update_model` or `valida` fails. we re-display the sign-in page with errors reported.

Here is an example from the template demo application that demonstrates how to use the general error (see file membership/web/views.py):

```
class SignUpHandler(BaseHandler):

    def post(self):
        if not self.validate_resubmission():
            self.error('Your registration request has been queued. '
                    'Please wait while your request will be processed. '
                    'If your request fails please try again.')
            return self.get()
        ...
```

Read more about model binding and validation in wheezy.validation package.

JSON

There is integration with wheezy.http package in JSON object encoding.

- `json_response()` - returns `HTTPResponse` with JSON content.

Here is an example:

```
class SignInHandler(BaseHandler):
    ...
    def post(self):
        ...
        credential = Credential()
        if (not self.try_update_model(credential)
                ...):
            if self.request.ajax:
                return self.json_response({'errors': self.errors})
            return self.get(credential)
        ...
        return self.see_other_for('default')
```

In case of error in ajax requests, the handler returns JSON object with any errors reported, otherwise it renders response the template. This way you are able to serve both: browsers with javascript enabled or disabled.

See file core.js for an example of how errors are processed by browser.

Templates

wheezy.web is not tied to some specific template engine, instead it provides you a convinient contract to add one you prefer (see file config.py). Template contract is any callable of the following form:

```
def render_template(self, template_name, **kwargs):
    return string
```

There are the following attributes and methods:

- `helpers` - a dictionary of context objects to be passed to `render_template` implementation (you need to override this method in case you need more specific context information in template).
  - `_` - `gettext` translations support.
  - `errors` - a dictionary with errors reported during validation. Key corresponds to attribute validated and value to a list of errors.
  - `handler` - an instance of currently executing handler.
  - `route_args`, `absolute_url_for`, `path_for` - relates to routing related methods.
  - `principal` - an instance of `wheezy.security.Principal` for the authenticated request or `None`.
  - `resubmission` - resubmission HTML form widget.
  - `xsrf` - XSRF protection HTML form widget.
- `render_template(template_name, widgets=None, **kwargs)` - renders template with name `template_name` and pass it context information in `**kwargs`.
- `render_response(template_name, widgets=None, **kwargs)` - writes result of `render_template` into `wheezy.http.HTTPResponse` and return it.

`widgets` argument in `render_template` and `render_response` is used to explicitly wrap HTML widgets (see wheezy.html package). Note, if you are using template engine that comes with widgets preprocessing you do not need to explicitly initialize this argument.

Widgets

Widgets are coming from wheezy.html package (see *WidgetExtension* for a template engine). Here is `SignUpHandler` from demo:

```
class SignUpHandler(BaseHandler):

    ...

    @handler_cache(profile=none_cache_profile)
    def get(self, registration=None):
        # ...
        return self.render_response(
            'membership/signup.html',
            model=self.model,
            registration=registration,
            account=registration.account,
            credential=registration.credential,
            questions=questions,
            account_types=tuple((k, self.gettext(v))
                                for k, v in account_types))
```

The benefit of using widgets is a syntax sugar in html template. They are processed by template proprocessor and generate template engine specific code.

Mako example:

```
<p>
    ${account.email.label('Email:')}
    ${account.email.textbox(autocomplete='off')}
    ${account.email.error()}
</p>
```

Wheezy Template example:

```
<p>
    @account.account_type.label('Account Type:')
    @account.account_type.radio(choices=account_types)
    @account.account_type.error()
</p>
```

Please note that wheezy.html package provides optimization of widgets per template engine used. That optimization is provided through use of template specific constructs. Preprocessor for Mako / Jinja2 / Tenjin / Wheezy.Template templates translates widgets to template engine specific operations offering optimal performance.

Read more about available widgets in wheezy.html package.

Authentication

Authentication is a process of confirming the truth of security principal. In a web application it usually relates to creating an encrypted cookie value, which can not easily be compromised by attacker. This is where integration with wheezy.security happens.

The process of creating authentication cookie is as simple as assiging instance of `wheezy.security.Principal` to attribute `principal`. Let's demonstrate this by example:

```
from wheezy.security import Principal

class SignInHandler(BaseHandler):

    def post(self):
        ...
        self.principal = Principal(
            id=credential.username,
            alias=credential.username)
        ...
```

Once we confirmed user has entered valid username and password we create an instance of `Principal` and assign it to `principal` attribute. In `setprincipal` implementation authentication cookie is created with a dump of `Principal` object and it value is protected by `wheezy.security.crypto.Ticket` (read more in wheezy.security).

Here are authentication configuration options (see file config.py):

```
options = {}

options.update({
        'ticket': Ticket(
            max_age=config.getint('crypto', 'ticket-max-age'),
            salt=config.get('crypto', 'ticket-salt'),
            cypher=aes128,
            digestmod=ripemd160 or sha256 or sha1,
            options={
                'CRYPTO_ENCRYPTION_KEY': config.get('crypto', 'encryption-key'),
                'CRYPTO_VALIDATION_KEY': config.get('crypto', 'validation-key')
            }),

        'AUTH_COOKIE': '_a',
        'AUTH_COOKIE_DOMAIN': None,
        'AUTH_COOKIE_PATH': '',
        'AUTH_COOKIE_SECURE': False,
})
```

You can obtain current security `Principal` by requesting `principal` attribute. The example below redirects user to default route in case he or she is already authenticated:

```
class SignInHandler(BaseHandler):

    def get(self, credential=None):
        if self.principal:
            return self.redirect_for('default')
        ...
```

Sign out is even simpler, just delete `principal` attribute:

```
class SignOutHandler(BaseHandler):

    def get(self):
        del self.principal
        return self.redirect_for('default')
```

Authorization

Authorization specify access rights to resources and provide access control in particular to your application.

You are able to request authorization by decorating your handler method with `authorize()`:

```
from wheezy.web import authorize

class MembersOnlyHandler(BaseHandler):

    @authorize
    def get(self, registration=None):
        return response
```

There is also a way to demand specific role:

```
class BusinessOnlyHandler(BaseHandler):

    @authorize(roles=('business',))
    def get(self, registration=None):
        return response
```

In case there are multiple roles specified in `authorize()` decorator than first match grant access. That means user is required to be at least in one role to pass this guard.

`authorize()` decorator may return HTTP response with status code 401 (Unauthorized) or 403 (Forbidden).

It is recommended to use `HTTPErrorMiddleware` to route HTTP status codes to signin or forbidden handlers. Read more in *HTTPErrorMiddleware* section.

@secure

Decorator `secure` accepts only secure requests (those that are communication via SSL) and if incoming request is not secure, issue permanent redirect to HTTPS location:

```
class MyHandler(BaseHandler):
    @secure
    def get(self):
        ...
        return response
```

The behavior can be controlled via `enabled` (in case it is `False` no checks performed, defaults to `True`).

XSRF/Resubmission

Cross-site request forgery (CSRF or XSRF), also known as a one-click attack is a type of malicious exploit of a website whereby unauthorized commands are transmitted from a user that the website trusts. Logging out of sites and avoiding their "remember me" features can mitigate CSRF risk.

Forms that can be accidentally, or maliciously submitted multiple times can cause undesired behavior and/or result in your application. Resubmits can happen for many reasons, mainly through page refresh, browser back button and incident multiple button clicks.

Regardless a source of issue you need to be aware it happening.

wheezy.web has built-in XSRF and resubmission protection. Configuration options let you customize name used:

```
options = {}
options.update({
        'XSRF_NAME': '_x',
        'RESUBMISSION_NAME': '_c'
})
```

You need include XSRF and/or resubmission widget into your form. Each template has context functions `xsrf()` and `resubmission()` for this purpose:

```
<form method="post">
    @xsrf()
    ...
</form>
```

Validation happens in handler, here is how it implemented in membership/web/views.py:

```
class SignInHandler(BaseHandler):

    def post(self):
        if not self.validate_xsrf_token():
            return self.redirect_for(self.route_args.route_name)
        ...
```

If XSRF token is invalid we redisplay the same page. Or we can show user an error message, here is use case for resubmission check:

```
class SignUpHandler(BaseHandler):

    def post(self):
        if not self.validate_resubmission():
            self.error('Your registration request has been queued. '
                    'Please wait while your request will be processed. '
                    'If your request fails please try again.')
            return self.get()
        ...
```

Since there is no simple rule of thumb when to use which protection and how to react in case it happening, it still strongly recommended take into account such situations during application development and provide unified application wide behavior.

Context

`BaseHandler` holds a number of useful features that other application layers (e.g. service layer, business logic) can benefit from.

There `context` attribute is available for this purpose. It is a dictionary that extends `options` with the following information: errors, locale, principal and translations.

Here is example from the template demo application (see membership/web/views.py):

```
class SignInHandler(BaseHandler):

    @attribute
    def factory(self):
        return Factory(self.context)
```

Context is passed to service factory.

Redirect Handler

`RedirectRouteHandler` redirects to a given route name (HTTP status code 302). You can use `redirect_handler()` in url mapping declaration:

```
all_urls = [
    url('', redirect_handler('welcome'), name='default'),
    ...
]
```

The example above always performs a redirect match for route *default* to route *welcome*. It asks browser to redirect it request to another page.

Permanent Redirect

`PermanentRedirectRouteHandler` performs a permanent redirect (HTTP status code 301) to the given route name. You can use `permanent_redirect_handler()` in the url mapping declaration:

```
all_urls = [
    url('', permanent_redirect_handler('welcome'), name='default'),
    ...
]
```

The example above results in a permanent redirect for route *default* to route *welcome*.

FileHandler

`FileHandler` serves static files out of some directory. You can use `file_handler()` in url mapping declaration:

```
all_urls = [
    url('static/{path:any}', file_handler(
            root='content/static/'), name='static'),
    ...
]
```

`file_handler()` accepts the following arguments:

- `root` - a root path of directory that holds static files, e.g. *.css*, *.js*, *.jpg*, etc. It is recommended that this directory be isolated from any other part of the application.

Request Headers

`FileHandler` handles both GET and HEAD browser requests, provides *Last-Modified* and *ETag* HTTP response headers, as well as understands *If-Modified-Since* and *If-None-Match* request headers, as sent by browser for static content.

GZip and Caching

It is recommended to use `file_handler()` together with `gzip_transform` and `response_cache` (requries HTTP cache middleware).

Here is example from template demo application:

```
from wheezy.http import response_cache
from wheezy.http.transforms import gzip_transform
from wheezy.http.transforms import response_transforms
from wheezy.web.handlers import file_handler


static_files = response_cache(static_cache_profile)(
    response_transforms(gzip_transform(compress_level=6))(
        file_handler(
            root='content/static/',
            age=timedelta(hours=1))))

all_urls = [
    url('static/{path:any}', static_files, name='static'),
    ...
]
```

Templates

Path for static files is provided by standard wheezy.routing `path_for(name, **kwargs)` function:

```
path_for('static', path='core.js')
```

TemplateHandler

`TemplateHandler` serves templates that do not require up front data processing. This mostly relates to some static pages, e.g. about, help, error, etc.

You can use `template_handler()` in the url mapping declaration:

```
from wheezy.web.handlers import template_handler

public_urls = [
    url('about', template_handler('public/about.html'), name='about'),
]
```

`template_handler()` supports the following arguments:

- `template_name` - template name used to render response.
- `status_code` - HTTP status code to set in response. Defaults to 200.

2.4.2 Middleware

wheezy.web extends middleware provided by wheezy.http by adding the following:

- bootstrap defaults
- path routing middleware
- http error middleware

Bootstrap Defaults

`bootstrap_defaults()` middleware factory does not provide any middleware, instead it is used to check application options and provide defaults.

The following options are checked:

- `path_router` - if it is not defined already an instance of `wheezy.routing.PathRouter` is created. Argument `url_mapping` is passed to `PathRouter.add_routes` method.
- `render_template` - defaults to an instance of `wheezy.web.templates.MakoTemplate`.
- `translations_manager` - defaults to an instance of `wheezy.core.i18n.TranslationsManager`.
- `ticket` - defaults to an instance of `wheezy.security.crypto.Ticket`.

PathRoutingMiddleware

`PathRoutingMiddleware` provides integration with wheezy.routing package. It is added to `WSGIApplication` via `path_routing_middleware_factory()`.

```
options = {}
main = WSGIApplication(
    middleware=[
        bootstrap_defaults(url_mapping=all_urls),
        path_routing_middleware_factory,
    ],
```

This factory requires *path_router* to be available in application options.

HTTPErrorMiddleware

`HTTPErrorMiddleware` provides a custom error page in case http status code is above 400 (HTTP status codes from 400 and up relates to client error, 500 and up - server error). This middleware is initialized with `error_mapping` dictionary, where key corresponds to HTTP status code and value to route name. In case of a status code match it redirects incoming request to route per `error_mapping`.

`HTTPErrorMiddleware` can be added to `WSGIApplication` via `http_error_middleware_factory()`:

```
main = WSGIApplication(
    middleware=[
        bootstrap_defaults(url_mapping=all_urls),
        http_cache_middleware_factory,
        http_error_middleware_factory,
        path_routing_middleware_factory
    ],
    options=options)
```

The following configuration options available:

```
from wheezy.core.collections import defaultdict

options = {}
options['http_errors'] = defaultdict(lambda: 'http500', {
            # HTTP status code: route name
            400: 'http400',
```

(continues on next page)

(continued from previous page)

```
            401: 'signin',
            403: 'http403',
            404: 'http404',
            500: 'http500',
        }),
})
```

`defaultdict` is used to provide default route name if there is no match in `http_errors` dictionary. All routes defined in `http_errors` must exist. These checks occur in `http_error_middleware_factory()`.

2.4.3 Transforms

Transforms are a way to manipulate handler response accordingly to some algorithm. *wheezy.web* provides decorator `handler_transforms()` to adapt transforms available in wheezy.http to web handlers sub-classed from `BaseHandler`:

```
from wheezy.http.transforms import gzip_transform
from wheezy.web.handlers import BaseHandler
from wheezy.web.transforms import handler_transforms

class MyHandler(BaseHandler):

    @handler_transforms(gzip_transform(compress_level=9))
    def get(self):
        return response
```

Please refer to wheezy.http documentation for more information.

2.4.4 Templates

wheezy.web does not provide its own implementation for template rendering instead it offers integration with the following packages:

- Jinja2 Templates
- Mako Templates
- Tenjin Templates
- Wheezy.Template

Contract

Template contract is any callable of the following form:

```
def render_template(self, template_name, **kwargs):
    return 'unicode string'
```

Jinja2 Templates

Here is the configuration option to define that Jinja2 templates are rendered within the application (see config.py for details):

```
from jinja2 import Environment
from jinja2 import FileSystemLoader
from wheezy.html.ext.jinja2 import WidgetExtension
from wheezy.html.ext.jinja2 import WhitespaceExtension
from wheezy.html.utils import format_value
from wheezy.web.templates import Jinja2Template

env = Environment(
    loader=FileSystemLoader('content/templates'),
    auto_reload=False,
    extensions=[
        WidgetExtension,
        WhitespaceExtension
    ])
env.globals.update({
    'format_value': format_value,
})
render_template = Jinja2Template(env)
```

The arguments passed to `Environment` are specific to Jinja2 templates and not explained here. Please refer to Jinja2 documentation.

Mako Templates

Here is the configuration option to define that Mako templates are rendered within application (see config.py for details):

```
from wheezy.html.ext.mako import whitespace_preprocessor
from wheezy.html.ext.mako import widget_preprocessor
from wheezy.web.templates import MakoTemplate


render_template = MakoTemplate(
        module_directory='/tmp/mako_modules',
        filesystem_checks=False,
        directories=['content/templates'],
        cache_factory=cache_factory,
        preprocessor=[
            widget_preprocessor,
            whitespace_preprocessor,
        ])
```

The arguments passed to `MakoTemplate` are specific to Mako templates and not explained here. Please refer to Mako documentation.

Tenjin Templates

Here is configuration option to define that Tenjin templates are rendered within application (see config.py for details):

```
from wheezy.html.ext.tenjin import whitespace_preprocessor
from wheezy.html.ext.tenjin import widget_preprocessor
from wheezy.html.utils import format_value
from wheezy.web.templates import TenjinTemplate

render_template = TenjinTemplate(
```

(continues on next page)

(continued from previous page)

```
        path=['content/templates'],
        pp=[
            widget_preprocessor,
            whitespace_preprocessor,
        ],
        helpers={
            'format_value': format_value
        })
```

The arguments passed to `TenjinTemplate` are specific to Tenjin templates and not explained here. Please refer to Tenjin documentation.

Wheezy Template

Here is configuration option to define that Wheezy.Template templates are rendered within application (see config.py for details):

```
from wheezy.html.ext.template import WhitespaceExtension
from wheezy.html.ext.template import WidgetExtension
from wheezy.html.utils import format_value
from wheezy.html.utils import html_escape
from wheezy.template.engine import Engine
from wheezy.template.ext.core import CoreExtension
from wheezy.template.loader import FileLoader
from wheezy.web.templates import WheezyTemplate

searchpath = ['content/templates-wheezy']
engine = Engine(
        loader=FileLoader(searchpath),
        extensions=[
            CoreExtension(),
            WidgetExtension(),
            WhitespaceExtension()
        ])
engine.global_vars.update({
    'format_value': format_value,
    'h': html_escape,
})
render_template = WheezyTemplate(engine)
```

The arguments passed to `Engine` are specific to Wheezy.Template and not explained here. Please refer to Wheezy.Template documentation.

2.4.5 Caching

wheezy.web provides decorator `handler_cache()` to adapt cache interface available in wheezy.http to web handlers sub-classed from `BaseHandler`:

```
from wheezy.http import CacheProfile
from wheezy.web.handlers import BaseHandler
from wheezy.web.caching import handler_cache

none_cache_profile = CacheProfile(
        'none',
```

(continues on next page)

(continued from previous page)

```
        no_store=True,
        enabled=True)

class MyHandler(BaseHandler):

    @handler_cache(profile=none_cache_profile)
    def get(self, credential=None):
        return response
```

Please refer to wheezy.http documentation for more information. All features available in wheezy.http caching are applicable.

Content caching plus cache dependency is the most advanced boost of your application performance. Regardless of template engine this can give up to 8-10 times better performance.

2.5 Modules

2.5.1 wheezy.web.authorization

2.5.2 wheezy.web.caching

2.5.3 wheezy.web.templates

2.5.4 wheezy.web.transforms

2.5.5 wheezy.web.handlers

2.5.6 wheezy.web.handlers.base

2.5.7 wheezy.web.handlers.file

2.5.8 wheezy.web.handlers.method

2.5.9 wheezy.web.handlers.template

2.5.10 wheezy.web.middleware

2.5.11 wheezy.web.middleware.bootstrap

2.5.12 wheezy.web.middleware.errors

2.5.13 wheezy.web.middleware.routing