

TurboGears2

July 9, 2009

Contents

1	First Web Application in Turbogears2	2
1.1	Install TurboGears2	2
1.2	Quick start a project	2
1.2.1	Folder Structure	3
1.3	Start the app	5
1.4	Database	6
1.4.1	DB Connection	6
1.4.2	DB Design	6
1.4.3	Create Database	8
1.5	URL	8
1.5.1	Template	9
1.6	Widget	9
1.6.1	Template	10
1.6.2	Controller	11
1.6.3	DataGrid	12
1.7	Testing/UnitTest	13
1.7.1	Test Index	14
1.7.2	Addresses	14
2	Functionality	15
2.1	Email Functionality	15
2.1.1	Verify Email (Chained validator)	15
2.1.2	Send Confirmation email	15
2.1.3	Send Confirmation link	16
2.2	Authorization	16
2.2.1	Show/Allow based on page name	16
2.2.2	Show/Allow based on authority and permissions	17
3	Rum Admin Interface	18

4	Maps with Turbogears	18
4.1	Install	18
4.2	Create project	19
4.3	layers.ini	19
4.4	Controllers	20
4.5	development.ini	20
4.6	run app	21
4.7	Generate layers and maps	21
4.8	Mylayers	21
4.9	References	25
5	Development Process	25
5.1	SubController/Controller Class	25
6	Others	25
6.1	URLAliasing	25
6.2	script_name	26
6.3	Open ID and Turbogears2	26
7	FAQ	26
7.1	Don't Set the Cookie	26
7.2	mounting test-controllers/getting root-controller instance	27
7.3	General Errors	28

Turbogears2 documentation is located here: <http://lucasmanual.com/mywiki/TurboGears2>
PDF Version: <http://lucasmanual.com/pdf/TurboGears2.pdf>

1 First Web Application in Turbogears2

1.1 Install TurboGears2

Create virtualenv, activate it, and install all the packages.

```
virtualenv --no-site-packages tg2env
cd tg2env/
source bin/activate
easy_install -i http://www.turbogears.org/2.0/downloads/ ↔
current/index tg.devtools
```

1.2 Quick start a project

- Create tg2 package using quickstart:

```
paster quickstart
```

- Enter the Project name, and choose if you want identity to be enabled. You should see something like this.

```
Enter project name: addressbook
Enter package name [addressbook]:
Do you need authentication and authorization in this project? ←
    [yes]
Selected and implied templates:
.....
```

- Done setting it up.

1.2.1 Folder Structure

This will create a skeleton of your project. Your address book project has the following structure::

```
addressbook/
|-- MANIFEST.in
|-- README.txt
|-- addressbook
|   |-- __init__.py
|   |-- config
|   |   |-- __init__.py
|   |   |-- app_cfg.py
|   |   |-- deployment.ini_tmpl
|   |   |-- environment.py
|   |   |-- middleware.py
|   |-- controllers
|   |   |-- __init__.py
|   |   |-- controller.template
|   |   |-- error.py
|   |   |-- root.py
|   |   |-- secure.py
|   |   |-- template.py
|   |-- i18n
|   |   |-- ru
|   |   |   |-- LC_MESSAGES
|   |   |   |-- addressbook.po
|   |-- lib
|   |   |-- __init__.py
|   |   |-- app_globals.py
|   |   |-- base.py
|   |   |-- helpers.py
|   |-- model
|   |   |-- __init__.py
|   |   |-- auth.py
|   |   |-- model.template
|   |-- public
|   |   |-- css
|   |   |   |-- style.css
|   |   |-- favicon.ico
```

```

| | |-- images
| | | |-- contentbg.png
| | | |-- error.png
| | | |-- header_inner2.png
| | | |-- headerbg.png
| | | |-- info.png
| | | |-- inputbg.png
| | | |-- loginbg.png
| | | |-- loginbottombg.png
| | | |-- loginheader-right.png
| | | |-- menu-item-actibg-first.png
| | | |-- menu-item-actibg.png
| | | |-- menu-item-border.png
| | | |-- menubg.png
| | | |-- ok.png
| | | |-- pagebg.png
| | | |-- star.png
| | | |-- strype2.png
| | | |-- under_the_hood_blue.png
| | | |-- warning.png
| |-- templates
| | |-- __init__.py
| | |-- about.html
| | |-- authentication.html
| | |-- debug.html
| | |-- error.html
| | |-- footer.html
| | |-- header.html
| | |-- index.html
| | |-- login.html
| | |-- master.html
| | |-- sidebars.html
| |-- tests
| | |-- __init__.py
| | |-- functional
| | | |-- __init__.py
| | | |-- test_authentication.py
| | | |-- test_root.py
| | |-- models
| | | |-- __init__.py
| | | |-- test_auth.py
| |-- websetup.py
|-- addressbook.egg-info
| |-- PKG-INFO
| |-- SOURCES.txt
| |-- dependency_links.txt
| |-- entry_points.txt
| |-- paster_plugins.txt
| |-- requires.txt
| |-- top_level.txt

```

```
|-- development.ini
|-- ez_setup
|   |-- README.txt
|   `-- __init__.py
|-- setup.cfg
|-- setup.py
|-- setup.pyc
`-- test.ini
```

Inside your addressbook folder you find another addressbook folder and inside of it you will find::

```
Config - Here you configure your application to the requirements you might have.
Controllers - Here you manipulate data, setup your urls, define what gets passed to what, your whole application flow etc.
Model - Here you define all your database models
Public - Here you keep all your static files, your css modules, your images.
Templates - Here you store your genshi template files.
i18n - Here you define your international settings.
lib- This is where you can attach plugins like turbomail, and configure global settings for you application, and setup helpers for templates.
tests - Here you define your nose test that you can write for your app.
```

1.3 Start the app

- If this is your first time running the app you need to run setup.py. You should be in a folder that has the development.ini and setup.py

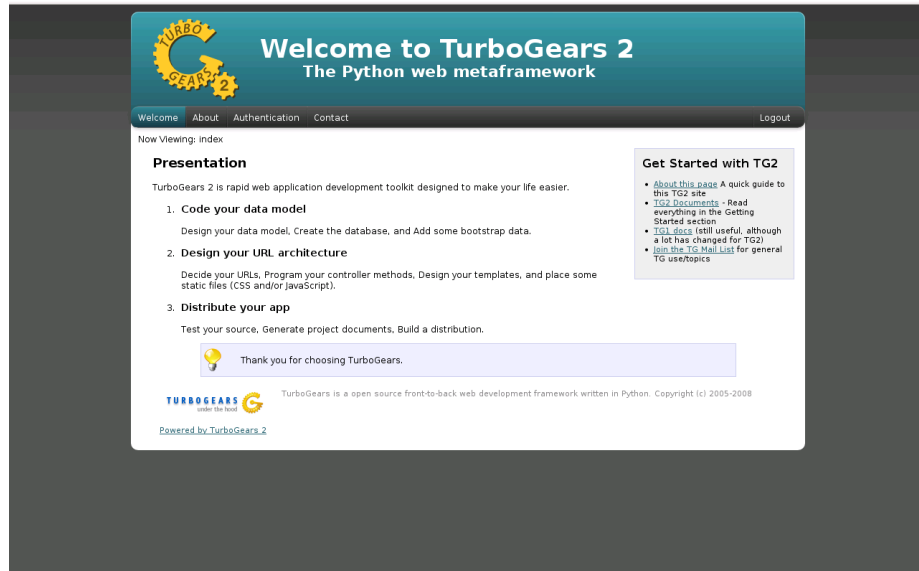
```
cd addressbook
python setup.py develop
```

To start the project we will use the following command.:

```
paster serve --reload development.ini
```

```
- paster provides a simple mechanism for running a TurboGears projects.
- server tells paster to start the webserver
--reload tells paster to reload the pages if one of the files have changed.
-development.ini is a general configuration file for turbogears2
```

You should be able to open your favorite browser and see the initial turobgears website. Visit this site: <http://localhost:8080/> You should see you site running.



1.4 Database

1.4.1 DB Connection

***Your Database configuration is located in development.ini file under**

```
sqlalchemy.url = sqlite:///%(here)s/devdata.db
```

***You can change this url to your version of database**

```
sqlalchemy.url=postgres://username:password:port@hostname/ ↵  
    databasename  
sqlalchemy.url=mysql://username:password@hostname:port/ ↵  
    databasename
```

- When you have your database connection ready we are ready to design simple database.

1.4.2 DB Design

***We need to create our database structure. We do it in model.py First you need to copy the model.template to mo**

```
cd addressbook/model
cp model.template model.py
```

***Inside of it change the foo_table, Class Foo and mapper in a following way**

```
from pylons import config
#from sqlalchemy import *
import sqlalchemy
from sqlalchemy.orm import mapper, relation
from addressbook.model import metadata

# Normal tables may be defined and mapped at module level.
# Our Addressbook table definition
from datetime import datetime
addressbook_table = sqlalchemy.Table("Addressbook", metadata,
    sqlalchemy.Column('Address_Sid', sqlalchemy.Integer, ←
        primary_key=True),
    sqlalchemy.Column('FirstName', sqlalchemy.Unicode(40), ←
        nullable=False),
    sqlalchemy.Column('LastName', sqlalchemy.Unicode(40), ←
        nullable=False),
    sqlalchemy.Column('MaidenLastName', sqlalchemy.Unicode ←
        (40)),
    sqlalchemy.Column('Email', sqlalchemy.Unicode(80), ←
        nullable=False),
    sqlalchemy.Column('Address', sqlalchemy.Unicode(80), ←
        nullable=False),
    sqlalchemy.Column('City', sqlalchemy.Unicode(80), nullable ←
        =False),
    sqlalchemy.Column('State', sqlalchemy.String(2), nullable= ←
        False),
    sqlalchemy.Column('ZipCode', sqlalchemy.Integer, nullable= ←
        False),
    sqlalchemy.Column('DOB', sqlalchemy.Date(), nullable=False ←
        ),
    sqlalchemy.Column('Gender', sqlalchemy.Unicode(6), ←
        nullable=False),
    sqlalchemy.Column('Description', sqlalchemy.Unicode(255), ←
        nullable=False),
    sqlalchemy.Column('Created', sqlalchemy.Integer, default= ←
        int(time.time())),
    sqlalchemy.Column('Last_UpdatedDate', sqlalchemy.Date, ←
        default=datetime.now().date(), onupdate=func.now(). ←
        date()),
    )
#This is an empty class that will become our data class
class Addressbook(object):
    def __init__(self, **kw):
```

```

        """automatically mapping attributes"""
        for key, value in kw.iteritems():
            setattr(self, key, value)
#Mapping of Table to Python Object Class
mapper(Addressbook, addressbook_table)
# Classes for reflected tables may be defined here, but the ↵
# table and
# mapping itself must be done in the init_model function.

```

1.4.3 Create Database

*Lets create the tables in our database by running the following command::

```
paster setup-app development.ini
```

1.5 URL

Lets create a url where we will add addresses to our addressbook. Lets make it <http://localhost:8080/-addaddress> We need to create a function in a root.py in controllers folder. Our new function will look just like index but with a different name. See below code.

```

class RootController(BaseController):
    """
    The root controller for the addressbook application.

    All the other controllers and WSGI applications should be ↵
    mounted on this
    controller. For example::

        panel = ControlPanelController()
        another_app = AnotherWSGIApplication()

    Keep in mind that WSGI applications shouldn't be mounted ↵
    directly: They
    must be wrapped around with :class:`tg.controllers. ↵
    WSGIAppController`.

    """

    secc = SecureController()
    admin = Catwalk(model, DBSession)
    error = ErrorController()

    @expose('addressbook.templates.index')
    def index(self):
        return dict(page='index')

```

```
@expose('addressbook.templates.about')
def about(self):
    return dict(page='about')
@expose('addressbook.templates.addaddress')
def addaddress(self, **kw):
    return dict(page='addaddress')
```

```
``@expose``
  This line defines which template we will use for this url. ←
  The template we use: addressbook/templates/addaddress. ←
  html
``def addaddress(self):``
  This adds our url http://localhost:8080/addaddress
``return dict(page='addaddress')``
  The dict returns data to the template in a python ←
  dictionary structure 'key=value'. We returned variable ←
  called page.
```

- Above will set your url. Now you need to create/modify template html to show what you want and do some code in address to do what you want it to do with the data.

1.5.1 Template

- Now that we have defined which template we will use, lets create the actual html file and start our app to make sure everything is working.
- Lets create a template real quick:

```
cd addressbook/templates/
cp index.html addaddress.html
```

- Lets start the app and see if our url works:

```
paster serve --reload development.ini
```

Ok. It worked. You should see the "Now Viewing:addaddress" This is using the page variable that we passed to the template. Lets create a widget aka the form where we will provide information and submit it.

1.6 Widget

- Lets create a widget form that will be asking for our addresses. Add the following to the top of our controller/root.py. For now we will be adding it to root.py but as you get more familiar you will be putting these in a seperate file:

```

from tw.forms import TableForm, TextField, CalendarDatePicker ←
    , SingleSelectField, TextArea
from tw.api import WidgetsList
#Validator
from formencode.validators import Int, NotEmpty, ←
    DateConverter, DateValidator,PostalCode,String,Email

class AddressForm(TableForm):
    # This WidgetsList is just a container
    class fields(WidgetsList):
        FirstName = TextField(validator=NotEmpty)
        LastName = TextField(validator=NotEmpty)
        MaidenLastName = TextField(validator=String)
        Email = TextField(validator=Email)
        Address = TextField(validator=NotEmpty)
        City = TextField(validator=NotEmpty)
        State = TextField(validator=NotEmpty)
        #Or you could do:
        #StateChoices = ("IL",
            #              "IN",
            #              "MS",
            #              )
        #State = SingleSelectField(options=StateChoices, ←
            validator=NotEmpty)
        ZipCode = TextField(size=6, validator=PostalCode())
        DOB = CalendarDatePicker(validator=DateConverter())
        GenderChoices = ("Female"),
            ("Male"),
            )
        Gender = SingleSelectField(options=GenderChoices)
        Description = TextArea(attrs=dict(rows=3, cols=25))

#then, we create an instance of this form
address_form = AddressForm("address_form", action=' ←
    saveaddress')

```

It will send the results to "saveaddress" which we have to create in root.py. But for now lets create our template.

1.6.1 Template

- Lets edit our template and add our form in there. We don't have to return the form as dictionary we can set pylons.c.myformname and use template_context.myformname in a template. Lets do just that. Edit our addressbook/templates/addaddress.html and add the following line:

```

${tmpl_context.address_form()}

```

Above will display our form in the template. The `address_form` is equivalent to `pylons.c.address_form` in a `root.py`. The template will look like:

First Name	<input type="text"/>
Last Name	<input type="text"/>
Maiden Last Name	<input type="text"/>
Email	<input type="text"/>
Address	<input type="text"/>
City	<input type="text"/>
State	<input type="text"/>
Zip Code	<input type="text"/>
Dob	<input type="text" value="09/01/2008"/> <input type="button" value="Choose"/>
Gender	<input type="text" value="Female"/>
Description	<input type="text"/>
<input type="button" value="Submit"/>	

1.6.2 Controller

Lets modify our `root.py` again and change the `def addaddress` and add `def saveaddress` functions. First make sure you have the following imported at the top of `root.py`:

```
from addressbook.lib.base import BaseController
from tg import expose, flash
from pylons.i18n import ugettext as _
from tg import redirect, validate
import pylons
from addressbook.model import DBSession, metadata
from addressbook.model import Addressbook
```

Our new change modifies `addaddress` and adds the `save` function which should look like this:

```
@expose('addressbook.templates.addaddress')
def addaddress(self, **kw):
    """Form to add new record"""
    # Flash updates the status on the page. Try modifying ↵
    it to something you want.
    flash("Hello Addressbook!")
```

```

# This is where we attach the form to pylons.c. ←
# something and whatever you attach will be ←
# available in the template as templ_context. ←
# something
# Passing the form in the return dict is no longer ←
# required, you can
# set pylons.c.form instead and use c.form in your ←
# template
pylons.c.address_form = address_form
return dict(page='addaddress')
@validate(address_form, error_handler=addaddress)
@expose()
def saveaddress(self, **kw):
    """Save it to the database"""
    address = Addressbook()
    address.FirstName = kw['FirstName']
    address.LastName = kw['LastName']
    address.MaidenLastName = kw['MaidenLastName']
    address.Email = kw['Email']
    address.Address = kw['Address']
    address.City = kw['City']
    address.State = kw['State']
    address.ZipCode = kw['ZipCode']
    address.DOB = kw['DOB']
    address.Description = kw['Description']
    address.Gender = kw['Gender']
    DBSession.add(address)
    DBSession.commit()
    flash("Successfully saved.")
    raise redirect("addaddress")
@expose()
def addresses(self, **kw):
    """List our addressbook"""
    return dict()

```

The addaddress will now display our widget in our template and when you click on Save it will send the data to **saveaddress**. Saveaddress will save the data into database. The only thing left now is to show the data that is in our addressbook.

1.6.3 DataGrid

Now we need to display our data. We will do that with datagrid from toscawidget. First thing we need to do is to import proper programs. Then we will define what will be displayed. Add the above lines above our root class:

```

from tw.forms.datagrid import DataGrid

address_grid = [('First Name', 'FirstName'),
                ('Last Name', 'LastName'),
                ('Address', 'Address'),

```

```

('City', 'City'),
('State', 'State'),
('Zip Code', 'ZipCode'),
('Gender', 'Gender')]

```

Then Lets modify our **addresses** function to be something like this:

```

@expose('addressbook.templates.addresses')
def addresses(self, **kw):
    """This function will display our data in a grid"""
    all=DBSession.query(Addressbook).all()
    pylons.c.address_grid=DataGrid(fields = address_grid)
    pylons.c.address_data=DBSession.query(Addressbook).
        all()
    return dict(page='addresses')

```

The `address_grid` will render the data, while `address_data` contains actual data. Now lets create `addresses.html` that will display our data.:

```

cd addressbook/templates
cp index.html addresses.html

```

Add the following somewhere in a code:

```

<p>${tmpl_context.address_grid(tmpl_context.address_data)} </ ←
p>

```

Now start the app again and go to <http://localhost:8080/addresses>

The data grid looks like this:

First Name	Last Name	Address	City	State	Zip Code	Gender
Lucas	S	123 main street	chicago	il	60600	Male
Kate	S	234 n main	chicago	il	60630	Female
Lukasz	Szybalski	123 main street	chicago	il	60630	Male

1.7 Testing/UnitTest

Testing in `turbogears2` is done by `nosetests`. You already have a folder designed for what you need to do. You can test your application for errors in few simple steps. The test files are located in `addressbook/tests/`

Run the tests using the following command:

```
python setup.py nosetests
```

You should see tests being run and at the end you will find:

```
----- ↵  
Ran 0 tests in 0.045s  
OK
```

1.7.1 Test Index

You can add more tests to file `yourapp/tests/functional/`. We already have a test that checks for `/index` page. Lets create a another test for our application:

```
cp addressbook/tests/functional/test_root.py addressbook/ ↵  
tests/functional/test_addressbook.py
```

Inside change the first test and delete the rest:

```
class TestPageData(TestController):  
  
    def test_addressbook(self):  
        resp = self.app.get('/addaddress')  
        ns = resp.namespace  
        assert 'page' in ns  
        assert ns['page'] == 'addaddress'
```

Save and run the tests again:

```
python setup.py nosetests
```

You will see:

```
----- ↵  
Ran 18 tests in 1.981s  
OK
```

Done. You have created your first test. With the above code you can make sure each page is returning before you upload to production server.

1.7.2 Addresses

Now add the second test case to our `TestPageData` class. The tests case start with `test_`:

```
def test_addresses_page_data(self):  
    resp = self.app.get('/addresses')  
    ns = resp.namespace  
    #See if there is a page return
```

```

assert 'page' in ns
#See if the page return is called addresses.
assert ns['page'] == 'addresses'
#Test if the template is there.
assert resp.template_name == 'addressbook.templates. ←
addresses'

```

2 Functionality

2.1 Email Functionality

2.1.1 Verify Email (Chained validator)

In order to verify email address, we will create 2 fields where we ask for email, and we will verify they are the same::

```

Email = TextField(validator=Email)
verify_email = TextField(validator=Email)

```

Then before you initialte the widget do::

```

#Chained Validator
from formencode import Schema
from formencode.validators import FieldsMatch

emailValidator = Schema(chained_validators=(FieldsMatch(' ←
    Email',
                                                         'verify_email',
                                                         messages={'invalidNoMatch':"Emails do not ←
                                                         match"}),))

```

Then call your instance and pass the extra validator::

```

#then, we create an instance of this form
address_form = AddressForm("address_form", action=' ←
    saveaddress', validator=emailValidator)

```

You can easily do the same with password validation

2.1.2 Send Confirmation email

1. Install turbomail 3.0dev or higher. 2. Add this to yourpackage.lib.app_globals.py::

```

def __init__(self):
    # ...
    from turbomail.adapters import tm_pylons
    tm_pylons.start_extension()

```

3. In the [default] section of development/deployment.ini add::

```
mail.on = true
mail.manager = immediate
mail.brand =
mail.transport = smtp
mail.smtp.server = your.mail.server
mail.smtp.debug =
mail.encoding = utf-8
mail.utf8qp.on = true
```

4. In your root.py do::

```
import turbomail
message=turbomail.Message("from@example.com,'to@example. ←
    com','Thank you for Registering')
message.plain="Hello this is a test"
turbomail.send(message)
```

2.1.3 Send Confirmation link

If you want to send a confirmation link you need to add a field to your model table, generate the confirmation number, and email it to registering user.

To generate confirmation number you can use::

```
import uuid
address.email_confirmation_uuid = str(uuid.uuid4())
```

To create a confirmation function that will accept that url do::

```
@expose()
def confirmaddress(self, email_confirmation_uuid,**kw):
    #print kw
    #print email_confirmation_uuid
    record=DBSession.query(model.Addressbook).filter(model ←
        .Addressbook.email_confirmation_uuid== ←
        email_confirmation_uuid).one()
    record.email_confirmed=True
    flash("Email Address Confirmed Successfully.")
    raise redirect("/addaddress")
```

2.2 Authorization

2.2.1 Show/Allow based on page name

- If you want to show a menu for example only on certain pages you could check the page name in the following way:

```

<span py:choose="">
<span py:when="defined('page') and 'someword' in page"> Put ↔
    some code here </span>
<span py:otherwise=""><a href="{tg.url('/')}">Home</a> Back ↔
    to your Quickstart Home page </span>
</span>

```

- Or

```

<span py:choose="">
<span py:when="defined('page') and page=='IT'"> Put some code ↔
    here </span>
<span py:otherwise=""><a href="{tg.url('/')}">Home</a> Back ↔
    to your Quickstart Home page </span>
</span>

```

2.2.2 Show/Allow based on authority and permissions

- First read the Getting started with Repoze.what. This will explain the idea of user, group, role. <http://what.repoze.org/docs/1.x/Manual/GettingStarted.html>

With groups and roles, we get:

```

USER:
- username
- password
- belongs to group A
- also belongs to groups B, C
- has roles 'author', 'editor', 'admin'

GROUP or ROLE:
- group/role ID
- can_view_x?
- can_view_y?
- can_modify_y?

```

- See websetup.py file for details on how to add permissions. The file contains code like:

```

manager = model.User()
manager.user_name = u'manager'
manager.display_name = u'Example manager'
manager.email_address = u'manager@somedomain.com'
manager.password = u'managepass'

model.DBSession.add(manager)

```

- After you create proper users, groups, roles/permissions you can look how they are used in a code. <http://www.turbogears.org/2.0/docs/main/Auth/Authorization.html>

3 Rum Admin Interface

- You can replace catwalk (default tg2 admin interface with rum interface. You do that by installing a helper app called tgrum in your virtual enviroment:

```
easy_install tgrum
```

- Now go to root.py and add the following before root controller and replace admin= with the new line.:

```
from tgrum import RumAlchemyController
from tg import config

# Create a prediacte to protect the RumAlchemy admin
is_manager = predicates.has_permission(
    'manage',
    msg=_('Only for people with the "manage" permission')
)
```

And change the admin to:

```
class RootController(BaseController):
    admin = RumAlchemyController(model,
        is_manager,
        template_path=config['paths']['templates'][0],
        render_flash=False,
    )
```

Done. Now go to admin interface and you should have rum admin interface running. You can add users, change users password, etc. See <http://python-rum.org/wiki/TgRum> and <http://docs.python-rum.org/tip/user/deploy.html#running-rum-inside-turbogears-2> for more details.

Keywords: Geo, [OpenLayers](#), turbogears, turbogears2, Openstreetmap, mapnik, postgis, TMS, WMS, GML, layers, maps, googlemaps,

4 Maps with Turbogears

4.1 Install

```
virtualenv --no-site-packages tg2env
cd tg2env/
source bin/activate
```

```
easy_install -i http://www.turbogears.org/2.0/downloads/ ↔  
    current/index tg.devtools  
easy_install -i http://www.turbogears.org/2.0/downloads/ ↔  
    current/index tg.ext.geo  
easy_install tw.openlayers  
easy_install psycpg2
```

4.2 Create project

```
paster quickstart geotry
```

- Run development install

```
python setup.py develop
```

- Add tg.ext.geo to

```
vi geotry.egg-info/paster_plugins.txt
```

4.3 layers.ini

- Add layers.ini. This file contains information about your geographic table that you have in your database.

```
[zipcodes]                #Name of the page/section  
singular=zipcode          #Singular name  
plural=zipcodes          #Plural name  
db=gis                   #Name of the database in postgreGIS  
table=zipcodes           #Table name that holds geographic ↔  
    infomration  
epsg=4326                #Projection, "European petroleum survey ↔  
    group" Standard projection on most maps  
units=degrees            #Units  
geomcolumn=the_geom      #The geographic column that holds ↔  
    shapes of geographic data. Usually its called the_geom  
idcolumn=Integer:gid     #ID
```

- Run command that creates geo functions. These function will create python code that will let you edit, view and delete geographic data.

```
paster geo-layer zipcodearea
```

- This creates files `zipcodearea.py` in controller and model. This file would contain the controller code for retrieving the zipcodes, posting new / edit zipcodes and deleting zipcodes

```
-- controllers
| | |-- __init__.py
| | |-- error.py
| | |-- root.py
| | |-- secc.py
| | |-- template.py
| | `-- zipcodearea.py
```

- What this does is to create a code for you that will output one layer that should be GML with `url=zipcodesarea`. These zipcodes would be fetched by GML by calling <http://localhost:8080/zipcodearea>. You can try pointing your browser to this url and you should get a GeoJSON formatted zipcode data.
- GML is Geography Markup language it is useful in getting geographic data in vector form.

4.4 Controllers

- Add this line to a `root.py`

```
vi geotry/controllers/root.py
```

- Below all import statements:

```
from geotry.controllers.zipcodearea import ←
    ZipcodeareaController

class RootController(BaseController):
    #admin = DBMechanic(SAProvider(metadata), '/admin')
    zipcodearea=ZipcodeareaController()
```

4.5 development.ini

- Add the postgresql username and pass to your `development.ini`

```
sqlalchemy.url=postgres://myuser:mypass@localhost/gis
```

4.6 run app

```
paster serve --reload development.ini
```

4.7 Generate layers and maps

- Remember the terms. These are needed for the opnlayers widget.
- GML is Geography Markup language it is useful in getting geographic data in vector form.
- WMS is web map service it is useful for getting geographic maps in raster form (form of an image like gif or png) You need to run WMS server to get data in this type of format.
- TMS is Tile Map Service provides access to cartographic maps of geo-referenced data, not direct access to the data itself. This document standardizes the way in which map tiles are requested by clients, and the ways that servers describe their holdings.
- URL Scheme. mapnik can generate png files for each area. Depending on the zoom level the final tiles are stored in folders `http://tah.openstreetmap.org/Tiles/tile/10/262/380.png`. This file is in `10/262/380.png`. Mapnik uses google like url scheme and therefore if you want to use it with openlayers you need to create custom `get_url` function, but that was done already. [OpenStreetMap.js](#). If you include this file you can add a layer of javascript that looks like: `OpenLayers.Layer.OSM.Mapnik("Mapnik")` This will display data from `openstreetmap.org`. If you want to modify the data source you will have to edit [OpenStreetMap.js](#) and point to your new url.
- See [available layers in tw.openlayers](#)

4.8 Mylayers

- Import necessary files.
- Add the following code right below all the import statements:

```
from tw.api import WidgetsList, js_symbol
from tw.openlayers import Map, GML, WMS, OSMMapnik, ←
    OMSRenderer, LayerSwitcher, OverviewMap, \
        MouseToolbar, MousePosition, PanZoomBar, ←
        \
        Permalink, SelectFeature
```

- Then add this class which will define your layers.

```

class MyLayers(WidgetsList):
    ol = WMS(name="OpenLayers WMS",
            url=["http://labs.metacarta.com/wms/vmap0"],
            options = {'layers':'basic'})
    nasa = WMS(name="NASA Global Mosaic",
              url=['http://t1.hypercube.telascience.org/cgi-bin/ ↵
                  landsat7'],
              options={'layers': 'landsat7'})
    mapnik = OSMapnik(name='MAPNIK')
    transportation = GML(name="Transportation", url=" ↵
                        countries",
                        options = {
                            "format": js_symbol(" OpenLayers.Format.GeoJSON") ↵
                            ,
                            "isBaseLayer": False,
                            "projection": js_symbol(' new OpenLayers. ↵
                                                    Projection("EPSG:4326")')
                        })

```

- Add the options for display:

```

class MyControls(WidgetsList):
    ls = LayerSwitcher()
    ovm = OverviewMap()
    mtb = MouseToolbar()
    mp = MousePosition()
    pzb = PanZoomBar()
    pl = Permalink()
    sf = SelectFeature(layer_name="Transportation", options={
        "hover": True,
        "onSelect": js_symbol("show_info"),
        "onUnselect": js_symbol("erase_info")})

```

- Generate the map

```

mymap = Map(id="map", layers=MyLayers(), controls=MyControls ↵
           (),
           center=(15,0), zoom=3)

```

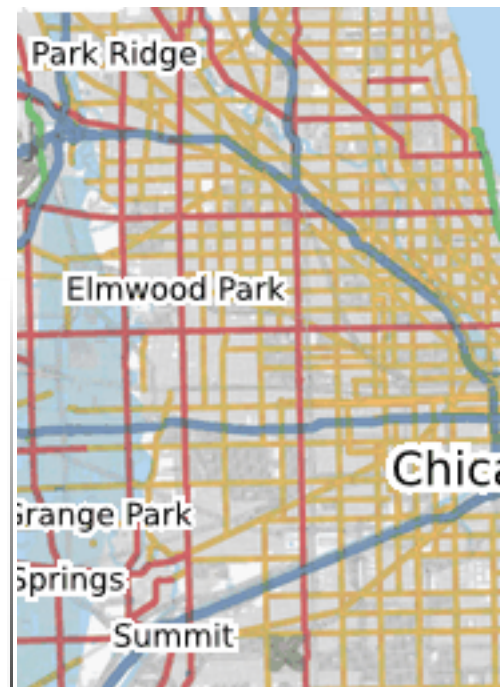
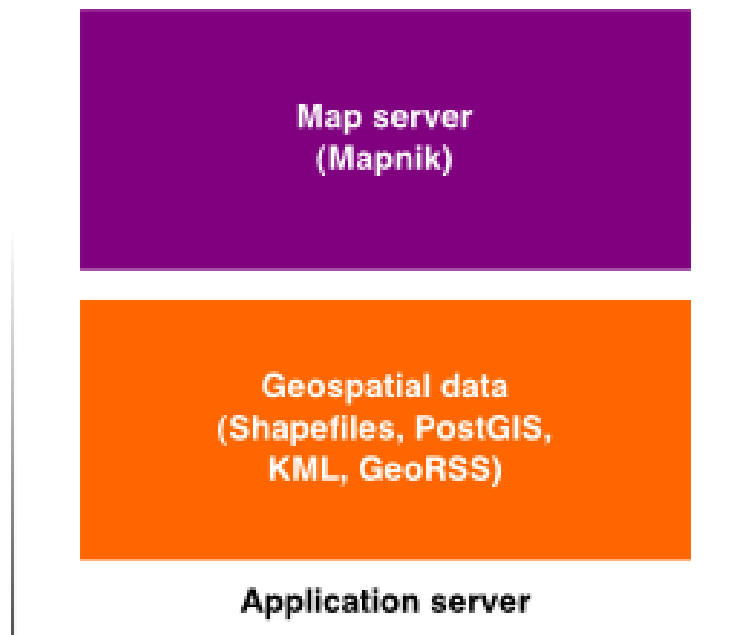
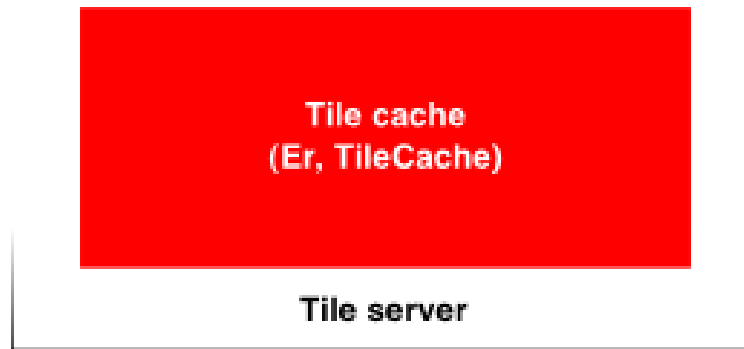
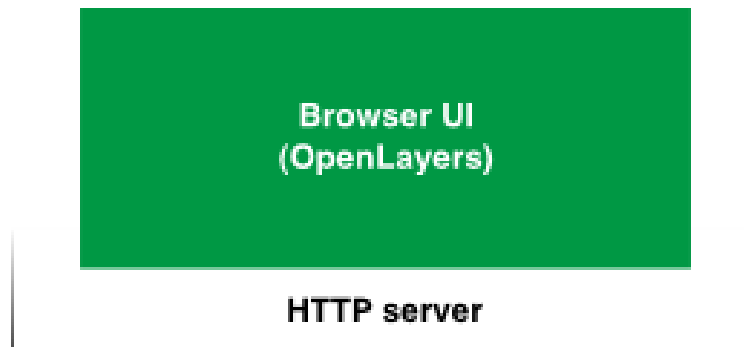
- Add the map to a index function

```

@expose('geogrid.templates.index')
def index(self):
    pylons.c.map = mymap
    return dict(page='index')

```

- Layer in wms is usually a url part `http://localhost/mymaps/ ↵
basic/someWMSservice`
`layers:basic` should get you a basic map using VMAP0 data ↵
refer to WMS way of serving maps (`http://localhost/mymaps ↵
/basic/someWMSservice`)
but `layers:landsat7` should show up landsat imagery (`http:// ↵
localhost/mymaps/landsat7/someWMSservice`)



4.9 References

1. <http://www.turbogears.org/2.0/docs/main/Extensions/Geo/MapFishTutorial.html> (Main source)
2. http://wiki.openstreetmap.org/index.php/OpenLayers_Simple_Example (Stand alone java script. Your final TG could should have similar characteristics)
3. <http://geo.turbogears.org/> (Examples of maps in turbogears2 apps)
4. http://www2.computer.org/portal/c/document_library/get_file?folderId=144965&name=DLFE-4309.pdf (IEEE Review)
5. This document was made in Chicago IL.

5 Development Process

5.1 SubController/Controller Class

- If you know it will take a lot of code you can start working in a seperate file outside of root.py. We will create IT page. :

```
cd myapp/controllers/  
cp controller.tempalate it.py
```

- Edit it.py and see what is in it.
- Now Edit root.py and add the following inside your Base controller.

```
from myapp.controllers.it import SampleController  
it = SampleController()
```

- Now you can access localhost:8080/it which is pulling code from it.py
- You can find more information here: <http://turbogears.org/2.0/docs/main/TGControllers.html>

6 Others

6.1 URLAliasing

1. **Url Aliases** "Assign some nice, user- and seo-friendly URLs (like /company/about) to not so nice ones (like /node/13) - and, well, have an ability to access sub-urls in a nice way, too (like /company/about/edit in the example above). An additional behaviour is to redirect the user to a 'nice' url (/company/about) when he accesses the original one (/node/13)"

6.2 script_name

1. `script_name` "The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location". This may be an empty string, if the application corresponds to the "root" of the server."

6.3 Open ID and Turbogears2

1. [Open ID and Turbogears2](#)

7 FAQ

7.1 Don't Set the Cookie

- Details: http://groups.google.com/group/turbogears/browse_thread/thread/2bd382f7a34cacf

Not directly via config, no.

Your post pointed me in the right direction. I tried your idea, but the SessionMiddleware is needed (they call it core middleware for a reason). Other code expects it to be there, so it can't be just removed. Instead, I went one level deeper and modified the SessionMiddleware itself to NOT set a cookie. I didn't want to modify the beaker package itself of course (bad practice). I could in theory subclass the beaker SessionMiddleware and then subclass AppConfig to add my custom SessionMiddleware, but it becomes a little cumbersome. So, I ended up just replacing the `__call__` method of the original beaker SessionMiddleware in my AppCfg.py file. The commented lines below are the lines that set the cookie:

```
### Start code #####
from beaker.middleware import SessionMiddleware
from beaker.session import SessionObject
def custom_session_middleware__call__(self, environ, start_response):
    session = SessionObject(environ, **self.options)
    if environ.get('paste.registry'):
        if environ['paste.registry'].reglist:
```

```

        environ['paste.registry'].register(self.session, ↵
            session)
    environ[self.environ_key] = session
    environ['beaker.get_session'] = self._get_session

    def session_start_response(status, headers, exc_info = ↵
        None):
        #if session.accessed():
        #    session.persist()
        #    if session.__dict__['_headers']['set_cookie']:
        #        cookie = session.__dict__['_headers'][' ↵
        #            cookie_out']
        #        if cookie:
        #            headers.append(('Set-cookie', cookie))
        return start_response(status, headers, exc_info)
    return self.wrap_app(environ, session_start_response)

SessionMiddleware.__call__ = ↵
    custom_session_middleware__call__
### End code #####

```

7.2 mounting test-controllers/getting root-controller instance

- Details: http://groups.google.com/group/turbogears/browse_thread/thread/3ba7ca9d35fd9d75

Assuming the paster-stuff is bootstrapped through code like

```

    here_dir = os.path.dirname(os.path.abspath(ableton.__file__ ↵
        ))
    conf_dir = os.path.dirname(here_dir)
    wsgiapp = loadapp('config:test.ini', relative_to=conf_dir)

```

you then can do it simply like this (inside a function/method ↵

```

    !!)

    import myproject.controllers.root as root
    root.RootController.mountpoint = TestController()

```

Then you can access the controller through the usual

```

    self.app.get("/mountpoint/action")

```

Of course mounting of whole controller hierarchies is ↵

```

    perfectly fine.

```

So I create a function in our base-test-class that allows to ↵

```

    register a passed
    controller for a given mountpoint. Voila, greatness ensues.

```

7.3 General Errors

Addition modules required

```
ImportError: No module named MySQLdb
```

Fixed with:

```
easy_install MySQL-python
```

ToscaWidget conflict

```
Installed /home/lek/work/mine/tg2env/lib/python2.6/site- ↵  
packages/tw.forms-0.9.7.2-py2.6.egg  
error: Installed distribution ToscaWidgets 0.9.7.1 ↵  
conflicts with requirement ToscaWidgets>=0.9.7.2
```

• Fixed with:

```
easy_install -U tw.forms
```

Keywords: Tutorial, [TurboGears](#), [SqlObject](#), [SqlAlchemy](#), web, python, features, Linux, windows, database, mysql, postgres, mssql, examples, howto, web apps, cherrypy to paste, manual, easy, first time to turbogears, documentation, solutions, fixed, solved, working, pylons, python web framework, Tosca TW widget Forms, [TurboGears](#) controller, [TurboGears](#) model, [TurboGears](#) and sqlalchemy, [TurboGears](#) and AJAX, Turbogears Documentation, Turbogears Manual, Turbogears Howto, Turbogears how to, Turbogears faq, trubogears, DOJO, plugin.