WEB APPLICATIONS ON PYTHON3 AND PYTHON2 WITH TURBOGEARS

Alessandro Molina
@__amol__
amol@turbogears.org

Who am I

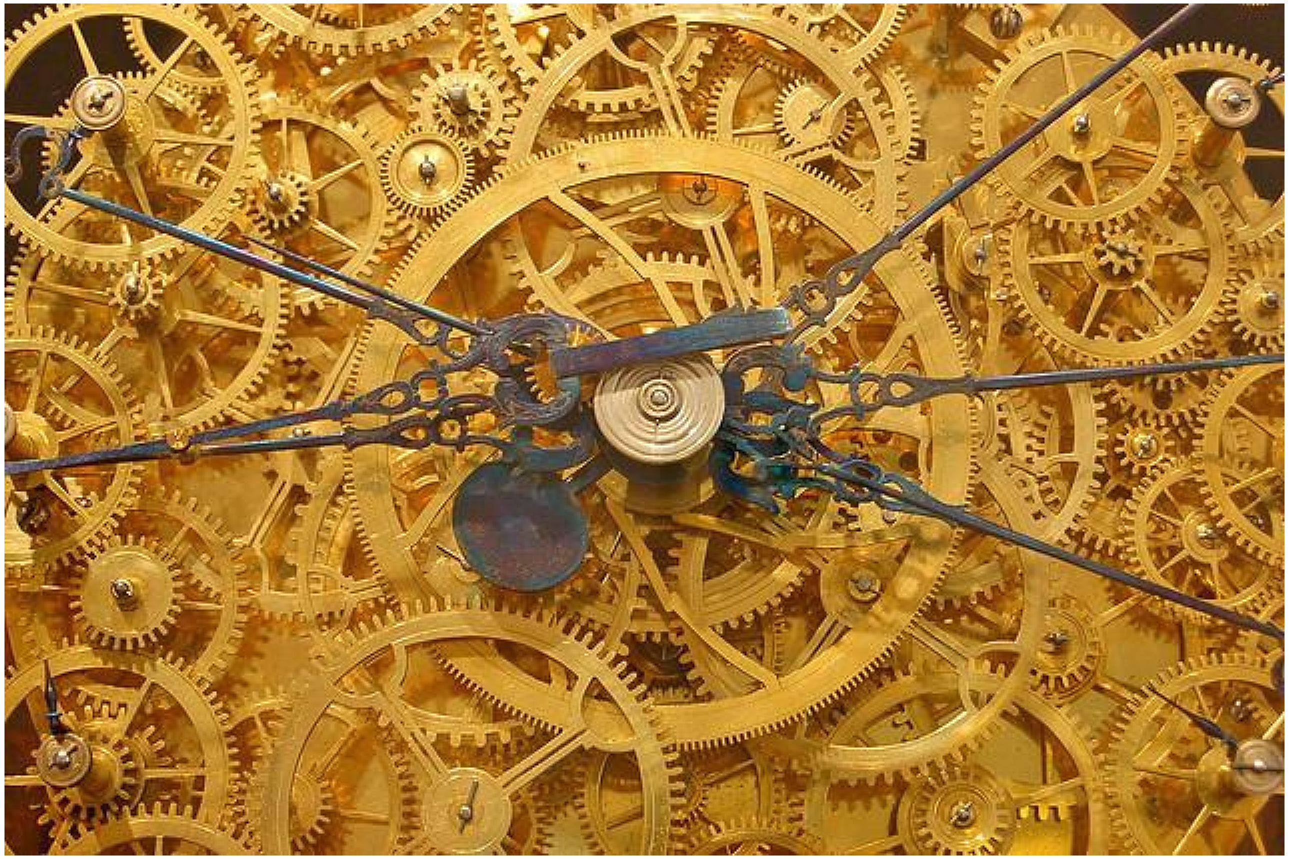
- CTO @ Axant.it mostly Python company (with some iOS and Android)
- TurboGears2 development team member
- MongoDB fan and Ming ODM contributor
- Skeptic developer always looking for a better solution

What's going to come

- Side by Side Python2 and Python3
- TurboGears on both
- ObjectDispatch, serving our pages
- Template Engines
- Gearbox toolkit
- What Python2 has that Python3 doesn't: authentication, widgets, i18n, admin

What you think your web app is

What your web app is for real



Some missing pieces

- Not all underlying pieces are available on Python3, yet
- Know when you need to stick to Python2, it will save you a lot of problems
- Think of moving to Python3 sooner than later, it will save you a lot of problems
- Python3 is a better Python, for real!

Why TurboGears

- Full stack framework, most of the features are builtin and available both on Py2 and Py3
- Minimal mode, really fast and simple for API servers and small apps
- Non opinionated, use your favourite template engine or database

Multiple Python Environments

- pythonbrew: a Python installation manager
    - Might want to try pythonz, fork of pythonbrew
- Have your Python2.x and 3.x installations side by side
- Start with Python3.2 at least, most libraries have been ported only to python 3.2 and newer.

Installing PythonBrew

- Download & Install Pythonbrew:
  - curl -kL http://xrl.us/pythonbrewinstall | bash
- Enabled it in your .bashrc
  - source $HOME/.pythonbrew/etc/bashrc
- List installed interpreters:
  - $ pythonbrew list
- Install Python 3.3
  - $ pythonbrew install 3.3.0

Setup Python2 environment

- Create an environment for your Python2 webapp:
  - $ virtualenv --distribute --no-site-packages py2
- Depending on your virtualenv version and system --distribute and --no-site-packages might be the default.

Our Python3 environment

- Switch to Python3
  - $ pythonbrew use Python-3.3.0
- Install virtualenv:
  - $ pip install virtualenv
- Create Python3 environment:
  - $ virtualenv py3
- Recover your standard Python:
  - $ pythonbrew off

Switch env not interpreter

- Work with Python2
  - $ source py2/bin/activate
- Work with Python3
  - $ source py3/bin/activate
- Quit current active environment
  - $ deactivate

Installing TurboGears2

- Enable Python3
    - $ source py3/bin/activate
- Install tg.devtools
    - $ pip install -f http://tg.gy/230 tg.devtools
- You should get TurboGears2-2.3.0b2
- Documentation
    - http://turbogears.readthedocs.org/en/tg2.3.0b2/
    - Don't forget version and trailing slash!

Out first Python3 app

- edit app.py
- TurboGears minimal mode, much like microframeworks

```
from wsgiref.simple_server import make_server
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose()
    def index(self):
        return "<h1>Hello World</h1>"

config = AppConfig(minimal=True, root_controller=RootController())

print("Serving on port 8080...")
httpd = make_server('', 8080, config.make_wsgi_app())
httpd.serve_forever()
```

Object Dispatch

- Routing happens on your controller method names and parameters
- Regular expressions can get messy, never write one anymore
    - unless your need it: tgext.routes
- Easy to get to the controller that handles an url just by looking at the url

Object Dispatch

```
class BlogEntryController
(BaseController):
   @expose()
   def index(self, post):
      return 'HI'

   @expose()
   def edit(self, post):
      return 'HI'

   @expose()
   def update(self, post):
      return 'HI'


class RootController(BaseController):
    blog = BlogEntryController()

    @expose()
    def index(self):
       return 'HI'

   @expose()
    def about(self):
       return 'HI'

   @expose()
    def more(self, *args, **kw):
       return 'HI'
```

URL	CONTROLLER
/index	RootController.index
/	RootController.index
/blog/3	BlogEntryController.index (post = 3)
/blog/update?post=3	BlogEntryController.update (post = 3)
/about	RootController.about
/more/1/2/3	RootController.more (args[0]=1, args[1]=2, args[3]=3)
/more?data=5	RootController.more (kw['data']=5)

Template Engine agnostic

- Doesn't enforce any template language bound to your controllers
- Genshi, Jinja, Mako and Kajiki officially supported
- Genshi is strongly suggested due to the need of a lingua franca for puggable applications

Templates out of the box

TYPE	NAME	URL
Markup + Streamed	Genshi	http://genshi.edgewall.org/
Text + Compiled	Mako	http://www.makotemplates.org/
Text + Compiled	Jinja	http://http://jinja.pocoo.org/
Markup + Compiled	Kajiki	http://kajiki.pythonisito.com/

Add a Template

- Install Genshi:
    - `$ pip install genshi`
- Register it as a renderer available to the framework:
    - `base_config.renderers = ['genshi']`
- Expose it in controllers:
    - `@expose('template.html')`

Hello Template

- index should now expose index.html template and return dict()

```html
<html xmlns="http://www.w5.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello World</h1>
  </body>
</html>
```

```python
from wsgiref.simple_server import make_server
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose('index.html')
    def index(self):
        return dict()

config = AppConfig(minimal=True,
                   root_controller=RootController())
config.renderers = ['genshi']

print("Serving on port 8080...")
httpd = make_server('', 8080, config.make_wsgi_app())
httpd.serve_forever()
```

Hello $user

- Every entry in the returned dictionary will be available inside the exposed template as a variable

```
<html xmlns="http://www.w5.org/1999/xhtml"
      xmlns:py="http://genshi.edgewall.org/">
  <head>
    <title>Hello World</title>
  </head>
  <body>
    <h1>Hello ${user}</h1>
  </body>
</html>
```

```
from wsgiref.simple_server import make_server
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose('index.html')
    def index(self):
        return dict(user='World')

config = AppConfig(minimal=True,
                   root_controller=RootController())
config.renderers = ['genshi']

print("Serving on port 8080...")
httpd = make_server('', 8080, config.make_wsgi_app())
httpd.serve_forever()
```

from request import data

- All arguments available in your URL will be passed as method parameters

```python
from wsgiref.simple_server import make_server
from tg import expose, TGController, AppConfig

class RootController(TGController):
    @expose('index.html')
    def index(self, user='World', **kw):
        return dict(user=user)

config = AppConfig(minimal=True, root_controller=RootController())
config.renderers = ['genshi']

print("Serving on port 8080...")
httpd = make_server('', 8080, config.make_wsgi_app())
httpd.serve_forever()
```

Going Full Stack

- TurboGears minimal mode provides a quick way to be productive.
- Full stack mode provides an already configured environment and more features

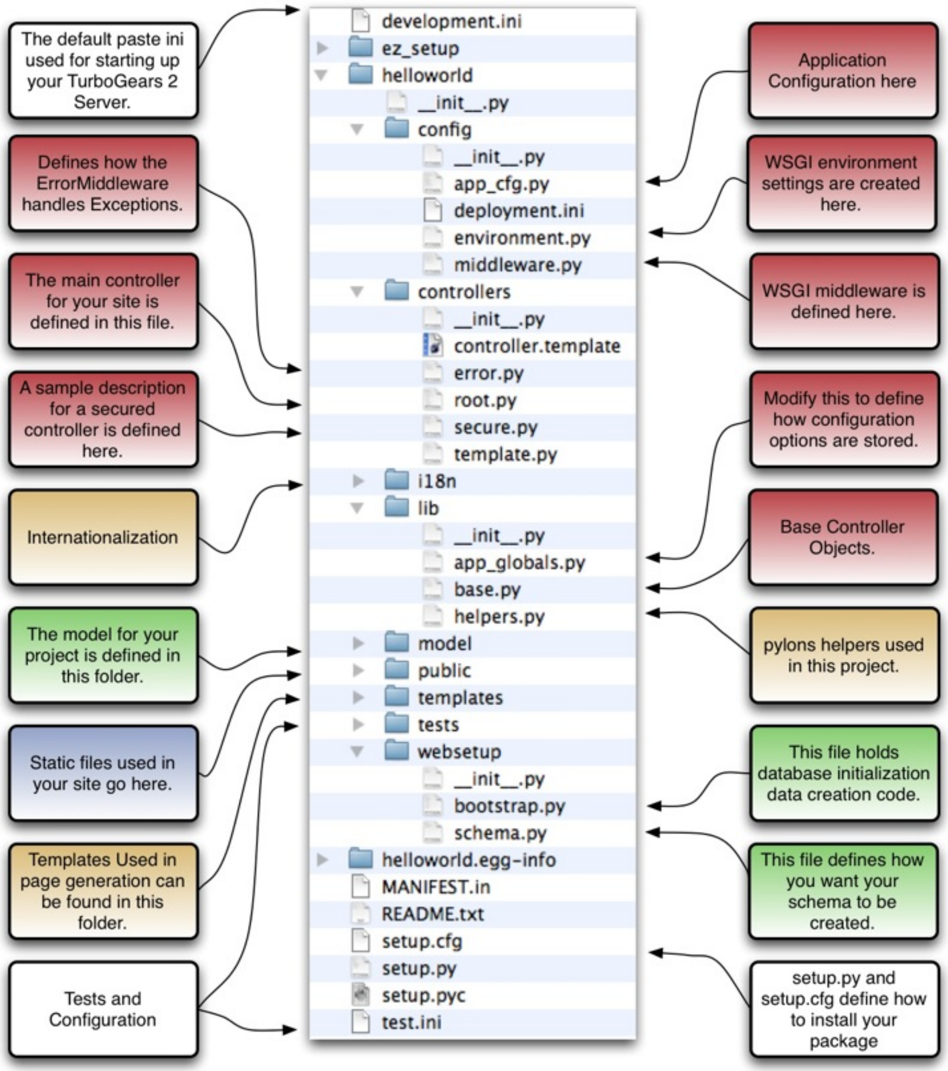
What is Full Stack

- ORM and Transaction Manager
- Authentication and Identification
- Interactive Debugger and Error Reporting
- PasteDeploy compatible configuration
- Static Files
- Session and Caching
- Widgets and Admin (on Python2)

Creating a Full Stack application

- Full stack applications are created through the gearbox toolkit
  - $ gearbox quickstart --skip-tw myapp
  - --skip-tw is required due to forms generation library not being available on Python3 yet.
- Full stack applications are packages: can be installed and updated to deploy

What's inside



The default paste ini used for starting up your TurboGears 2 Server.
Defines how the ErrorMiddleware handles Exceptions.
The main controller for your site is defined in this file.
A sample description for a secured controller is defined here.
Internationalization
The model for your project is defined in this folder.
Static files used in your site go here.
Templates Used in page generation can be found in this folder.
Tests and Configuration
Application Configuration here
WSGI environment settings are created here.
WSGI middleware is defined here.
Modify this to define how configuration options are stored.
Base Controller Objects.
pylons helpers used in this project.
This file holds database initialization data creation code.
This file defines how you want your schema to be created.
setup.py and setup.cfg define how to install your package
development.ini
ez_setup
helloworld
__init__.py
config
__init__.py
app_cfg.py
deployment.ini
environment.py
middleware.py
controllers
__init__.py
controller.template
error.py
root.py
secure.py
template.py
i18n
lib
__init__.py
app_globals.py
base.py
helpers.py
model
public
templates
tests
websetup
__init__.py
bootstrap.py
schema.py
helloworld.egg-info
MANIFEST.in
README.txt
setup.cfg
setup.py
setup.pyc
test.ini

Install the quickstarted app

- To use the app you need to install it:
  - $ pip install -e .
- Installing also brings in dependencies the app requires
- Now run your application
  - $ gearbox serve --reload

A lot is there now

- Point your browser to http://localhost:8080 and see TurboGears in action

- Quite a lot is there now!
- Have a look around
- App pages explain the app itself

Authentication

- Click the login link in the upper-right corner
  - username: manager
  - password: managepass
- Crash!
- Database has not been initialized
  - You now know what the interactive debugger looks like!

Authentication, #2 try

- Create database and basic entities
  - $ gearbox setup-app
  - By default sqlite: devdata.db
- Click the login link in the upper-right corner
  - username: manager
  - password: managepass
- Woah! Welcome back manager!

Authorization

- Go to http://localhost:8080/secc
    - Secure controller here
- Logout
- Go to http://localhost:8080/secc
    - Only for people with "manage" permission

Users and Permissions

- Default users are created in the application setup script (setup-app)
    - Have a look at websetup/bootstrap.py
- Default models are provided by the quickstart command for User, Group and Permission
    - Have a look at models/auth.py

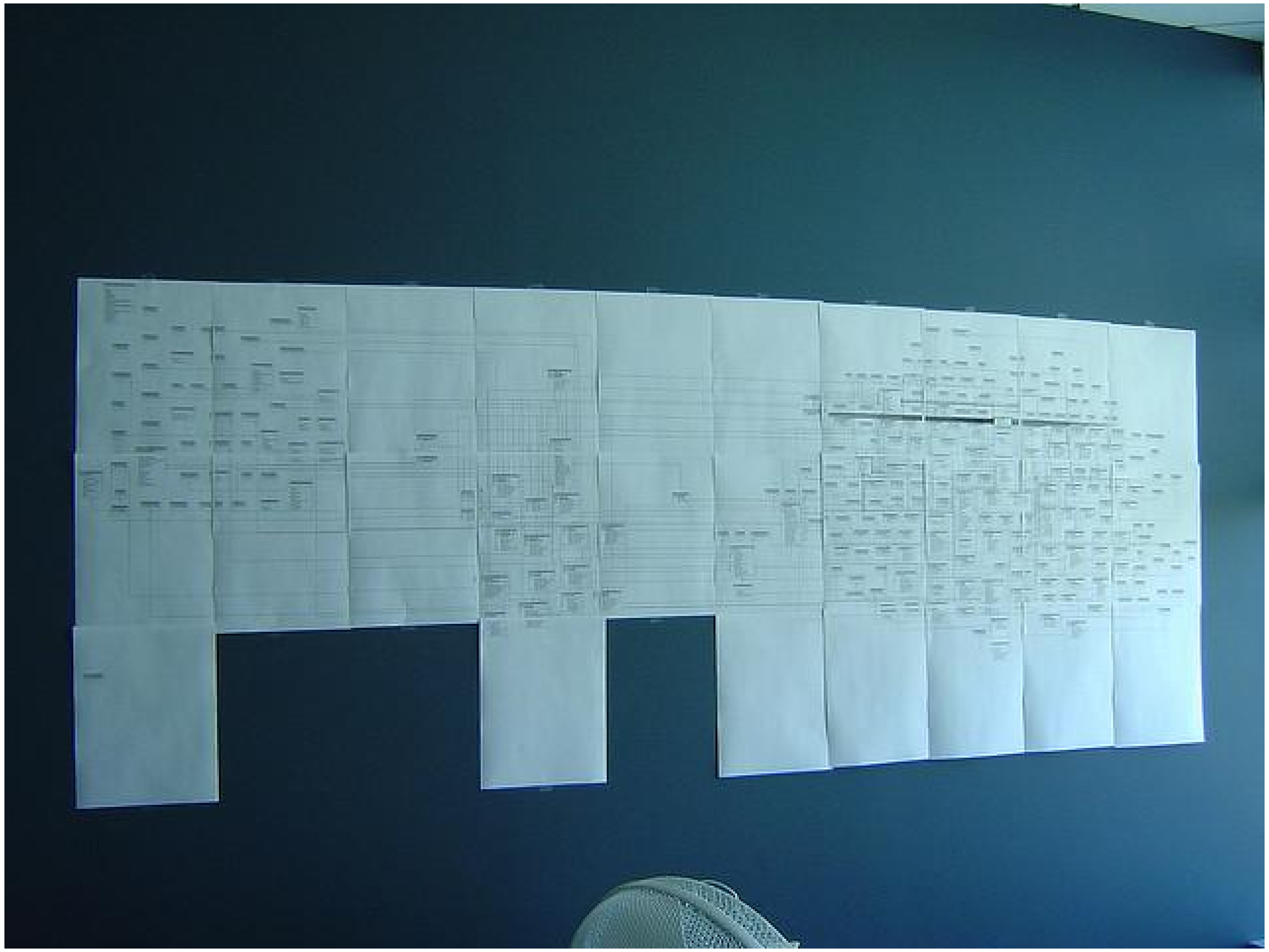
Predicates and Authorization

- Turbogears checks for authorization requiring predicates bound to controllers or methods

```
@expose('prova.templates.index')
@require(predicates.has_permission('manage', msg=l_('Only for
managers')))
def manage_permission_only(self, **kw):
    """Illustrate how a page for managers only works."""
    return dict(page='managers stuff')
```

- http://turbogears.readthedocs.org/en/tg2.3.0b2/turbogears/authorization.html

Database



Accessing Models

- TurboGears relies on SQLAlchemy for SQL based databases and on Ming for MongoDB databases
  - Both are first citizens of the TurboGears Admin
  - Both are supported out of the box
  - Run quickstart --ming to have MongoDB support
  - Run quickstart --nosa to disable database at all
  - gearbox help quickstart

Accessing Models

- TurboGears relies on SQLAlchemy for SQL based databases and on Ming for MongoDB databases
    - Both are first citizens of the TurboGears Admin
    - Both are supported out of the box
    - Run quickstart --ming to have MongoDB support
    - Run quickstart --nosa to disable database at all
    - gearbox help quickstart

Create, Read, Update, Delete

- Create
  - DBSession.add(Page(name='index'))
- Read
  - page = DBSession.query(Page).filter_by(name='index').one()
- Update
  - page.data = 'This is an empty page'
- Delete
  - DBSession.delete(page)

Wiki20 Tutorial

- TurboGears documentation provides a great Wiki in 20 minutes Tutorial
  - http://turbogears.readthedocs.org/en/tg2.3.0b2/turbogears/wiki20.html#wiki-model
- Just skip up to the Wiki Model section, we already know the previous parts

Let's play with it!

Python3 no more

Back to Python2

- This is as far as you can get using Python3
- Some features not available there
    - i18n utilities
    - Widgets
    - MongoDB
    - TurboGears Admin

Moving to Widgets and Forms

- Doing forms is a tedious task
- Validating data users write is a mess
- Let's use widgets and forms
  - Generates HTML for us
  - Validates input
  - Keeps track of values in case of errors
  - Reports errors to users

Enable forms

- Switch back to Python2
  - $ source py2/bin/activate
- Enable forms in your project
  - edit config/app_cfg.py
  - base_config.use_toscawidgets2 = True
- Install
  - Add tw2.forms to setup.py install_requires
  - pip install -e .

Writing Widgets and Validation

```
from tg import validate
import tw2.core as twc
import tw2.forms as twf


class PageForm(twf.TableForm):
    pagename = twf.HiddenField(validator=twc.Validator(required=True))
    data = twf.TextArea(validator=twc.Validator(required=True))

    action = lurl('/save')


class RootController(BaseController):
    @expose()
    @validate(PageForm, error_handler=edit)
    def save(self, pagename, data):
        page = DBSession.query(Page).filter_by(pagename=pagename).one()
        page.data = data
        flash("Page successfully updated!")
        return redirect("/" + pagename)
```

Let's translate

- TurboGears detects language of the user and translates text in templates and controllers accordigly
- Translation itself is available on both Python2 and Python3
- String collection is only available on Python2

Collect text

- Install Babel
  - $ pip install babel
- Template content is automatically collected and translated
- Text in controllers must be wrapped with l_() or _() to make them translatable
- Wrap your flash messages

Perform collection

- Walkthrough on i18n
    - http://turbogears.readthedocs.org/en/tg2.3.0b2/turbogears/i18n.html
- Utility commands
    - python setup.py extract_messages
    - python setup.py init_catalog -l it
    - poedit i18n/it/LC_MESSAGES/myproj.po
    - python setup.py compile_catalog

TurboGears Admin

- A lot of effort has been spent in
  - writing forms
  - validating data
  - editing pages
- Still a lot to do
  - How do I delete a page?
  - How do I search for a page?

Admin does that for you

Enabling the Admin

- Automatically done if you quickstarted without --skip-tw option
- Enable manually as we started on Py3
  - $ pip install tgext.admin
  - Add admin controller

```
from tgext.admin.tgadminconfig import TGAdminConfig
from tgext.admin.controller import AdminController

class RootController(BaseController):
    admin = AdminController(model, DBSession, config_type=TGAdminConfig)
```

Rapid Prototyping

- TurboGears admin is based on tgext.crud, a powerfull rapid prototyping tool
- Have a look at the admin tutorial
  - http://turbogears.readthedocs.org/en/tg2.3.0b2/turbogears/wikier/index.html
- Avoid pushing the admin too far
  - Custom solutions are cleaner than a too much customized admin

Admin is great for REST

- REST api for free
- For real, try to put .json after your pages list
    - /admin/pages.json
- Supports a full featured REST api
    - Meaningful error codes
    - Conditional PUT
- You might want to use tgext.crud directly to build rest services

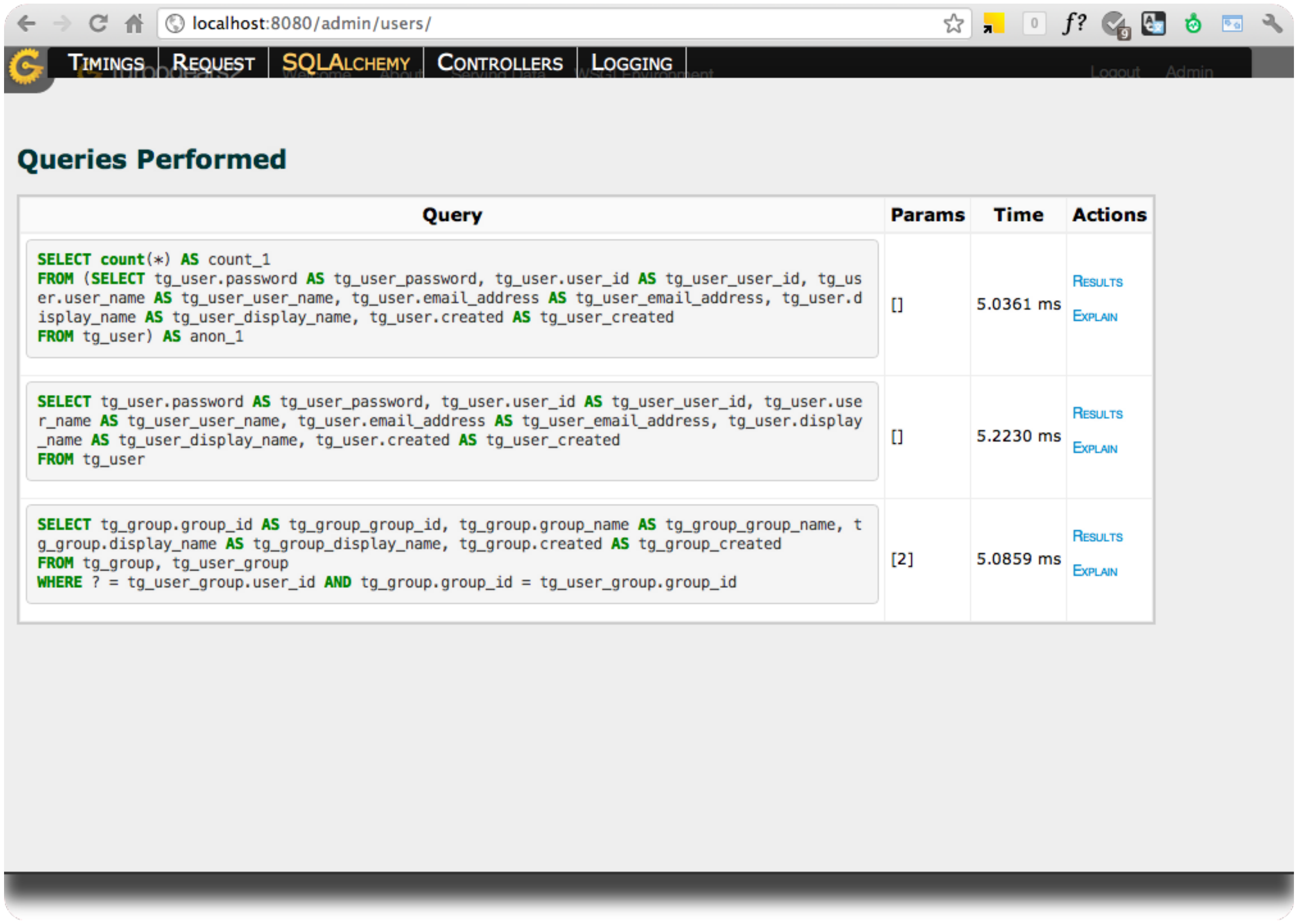
Look for ready made plugins

- tgext.pluggable enables pluggable applications
- The cogbin is a collection of existing extensions and pluggable apps
    - http://turbogears.org/cogbin.html
- Features like Facebook auth, blogging, registration and so on available on cogbin

The DebugBar

- Great plugin available is the DebugBar
  - pip install tgext.pluggable
  - pip install tgext.debugbar
- Enable debugbar
  - config/app_cfg.py
  - from tgext.pluggable import plug
  - plug(base_config, 'tgext.debugbar')

DebugBar in action



Queries Performed

Query	Params	Time	Actions
`SELECT count(*) AS count_1 FROM (SELECT tg_user.password AS tg_user_password, tg_user.user_id AS tg_user_user_id, tg_user.user_name AS tg_user_user_name, tg_user.email_address AS tg_user_email_address, tg_user.display_name AS tg_user_display_name, tg_user.created AS tg_user_created FROM tg_user) AS anon_1`	[]	5.0361 ms	Results Explain
`SELECT tg_user.password AS tg_user_password, tg_user.user_id AS tg_user_user_id, tg_user.user_name AS tg_user_user_name, tg_user.email_address AS tg_user_email_address, tg_user.display_name AS tg_user_display_name, tg_user.created AS tg_user_created FROM tg_user`	[]	5.2230 ms	Results Explain
`SELECT tg_group.group_id AS tg_group_group_id, tg_group.group_name AS tg_group_group_name, tg_group.display_name AS tg_group_display_name, tg_group.created AS tg_group_created FROM tg_group, tg_user_group WHERE ? = tg_user_group.user_id AND tg_group.group_id = tg_user_group.group_id`	[2]	5.0859 ms	Results Explain

Going Mongo with Ming

3 **In Mars,** Evil Emperor Ming of the mythical planet Mongo, who has joined with Mars' Queen Azura to capture the universe, watches the earth from a giant observatory.

Want to support MongoDB

- Try passing --ming to gearbox quickstart
- Full featured admin and tgext.crud as on SQLAlchemy
- Ming ODM has similar syntax
- Ming provides Unit of Work pattern like SQLAlchemy
  - transaction manager missing, pay attention

Want to support MongoDB

- Full featured admin and tgext.crud as on SQLAlchemy
- DebugBar works also with Ming
    - now with cool hightlighting of map-reduce javascript code too!

Questions?