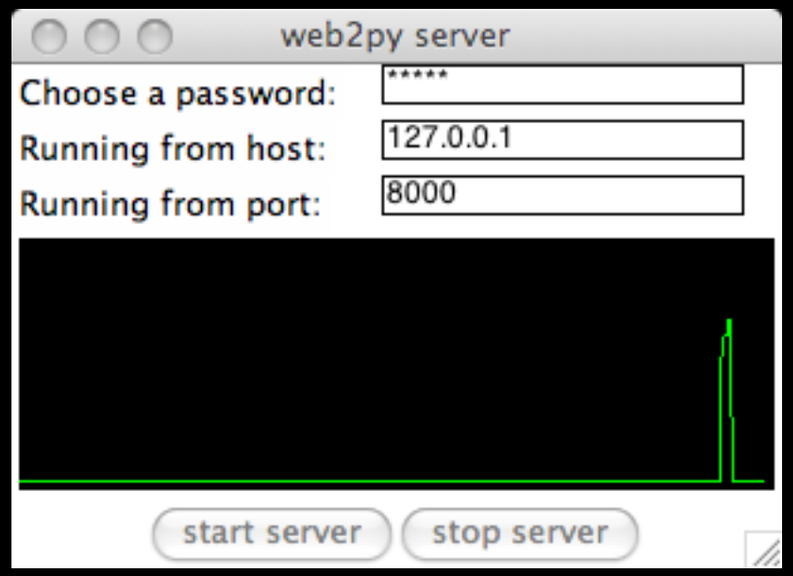
web2py

Enterprise Web Framework

These slides replace the previous tutorials (v1.63)

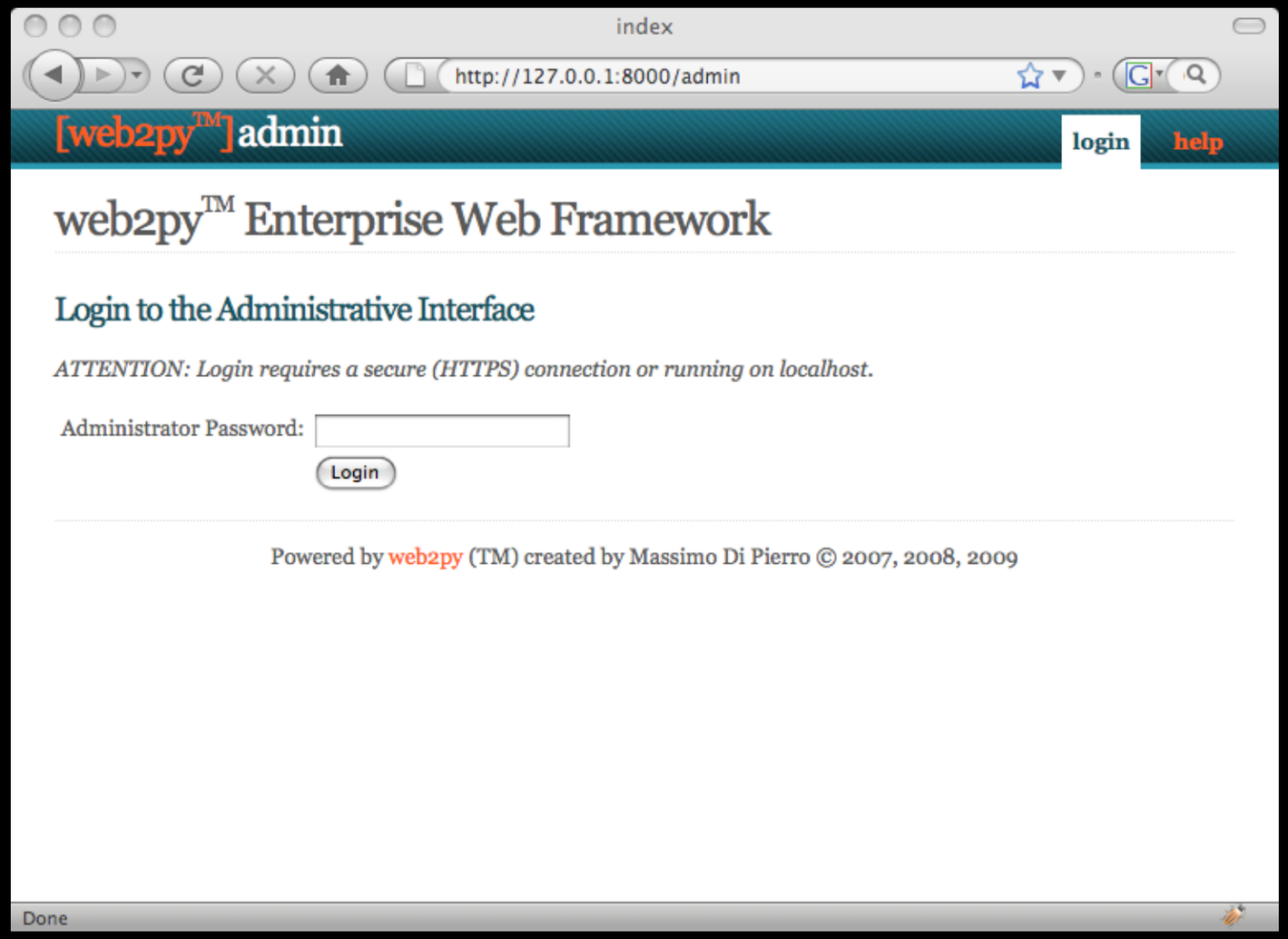
Startup Interface

- Download and click!
- No Installation
- No Dependencies
- No Configuration
- Runs Everywhere
- including Google App Engine



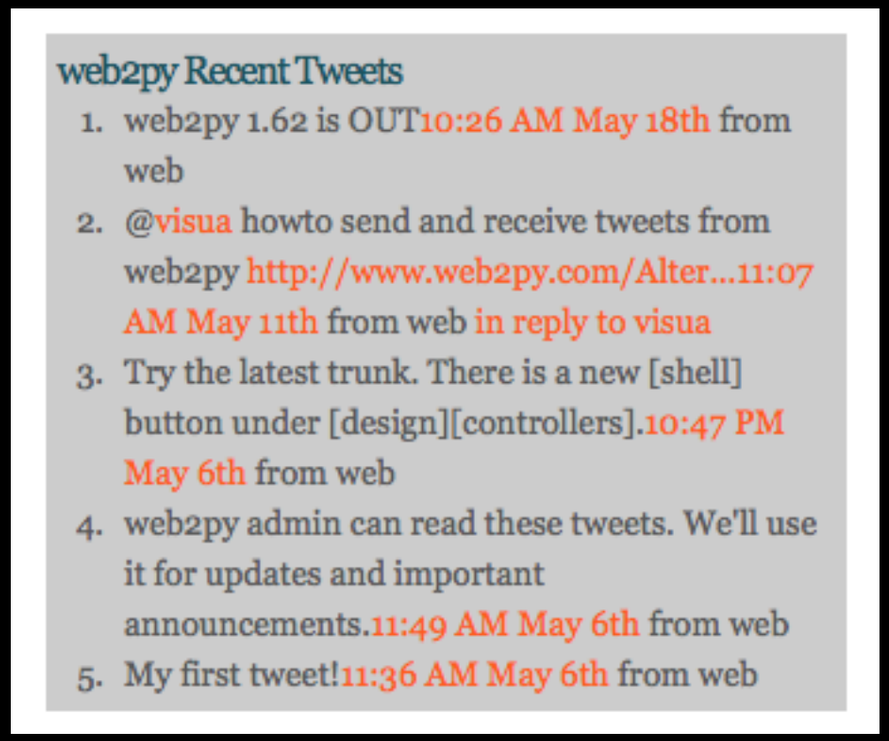
Web Based Admin Interface

- Login
- Manage Apps
- Create Apps
- Design Apps
- Test/Debug
- Mercurial Integration

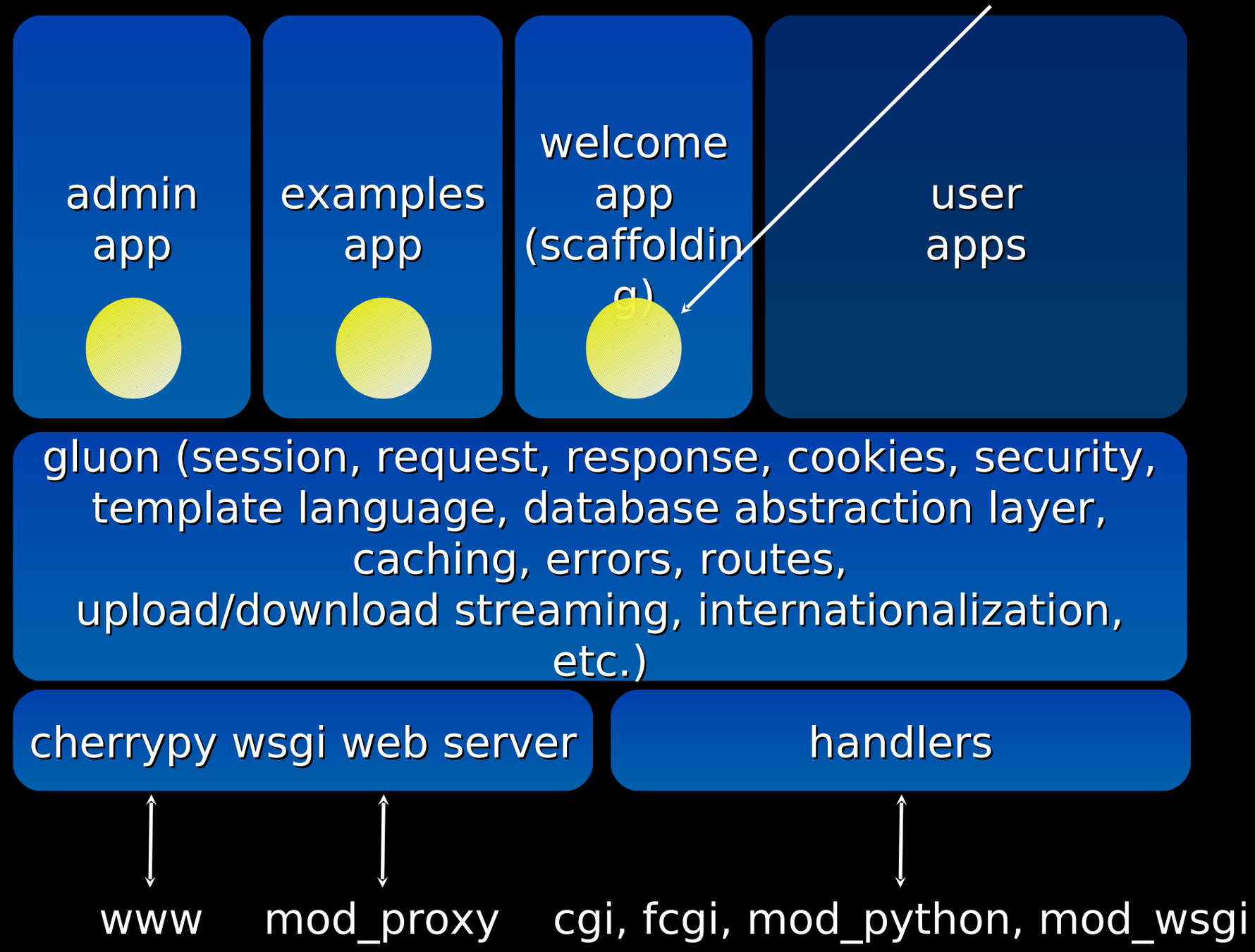


Web Based Admin Interface

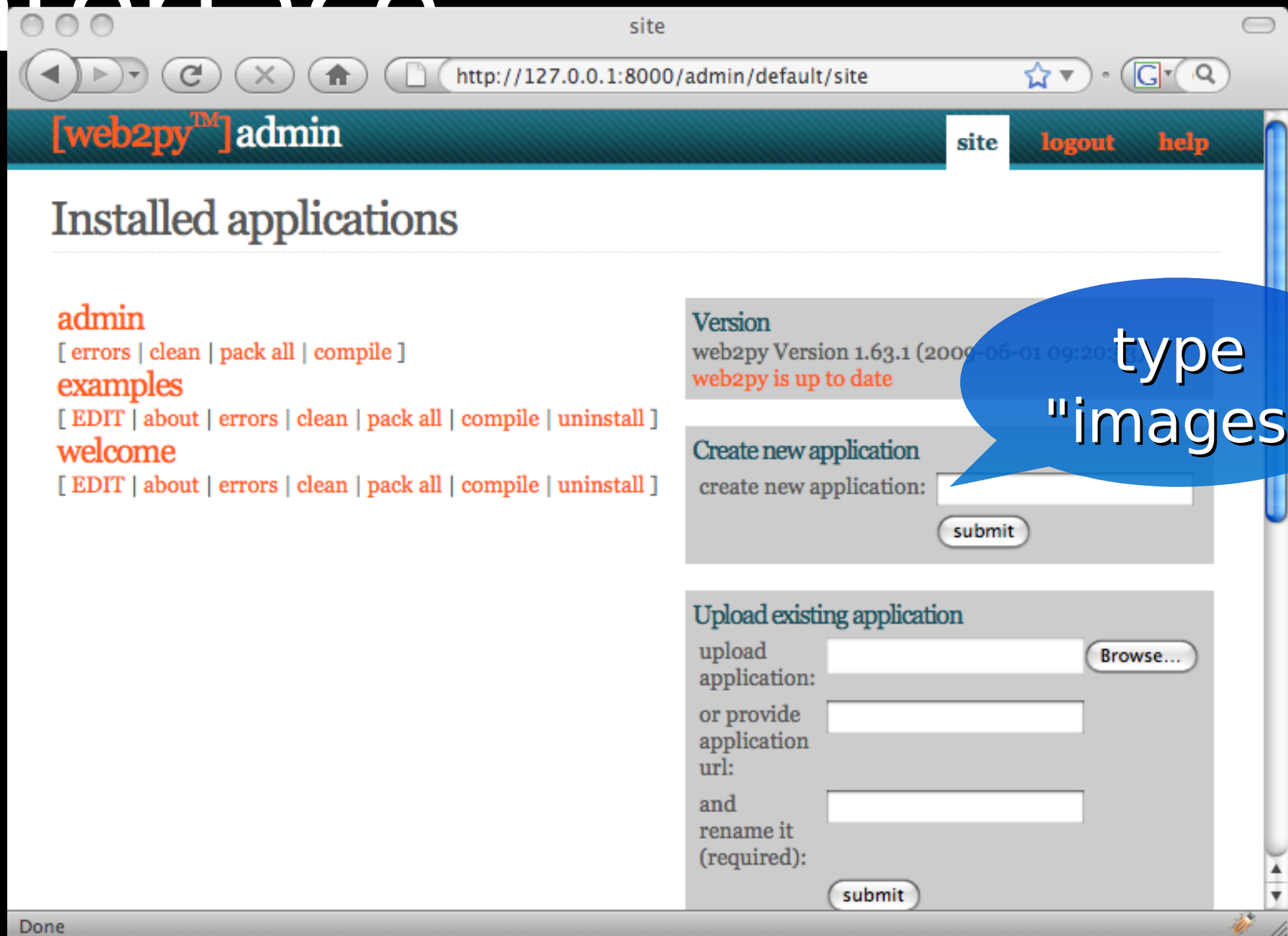
- receive web2py announcements from twitter
- be notified of upgrades



web2py Architecture

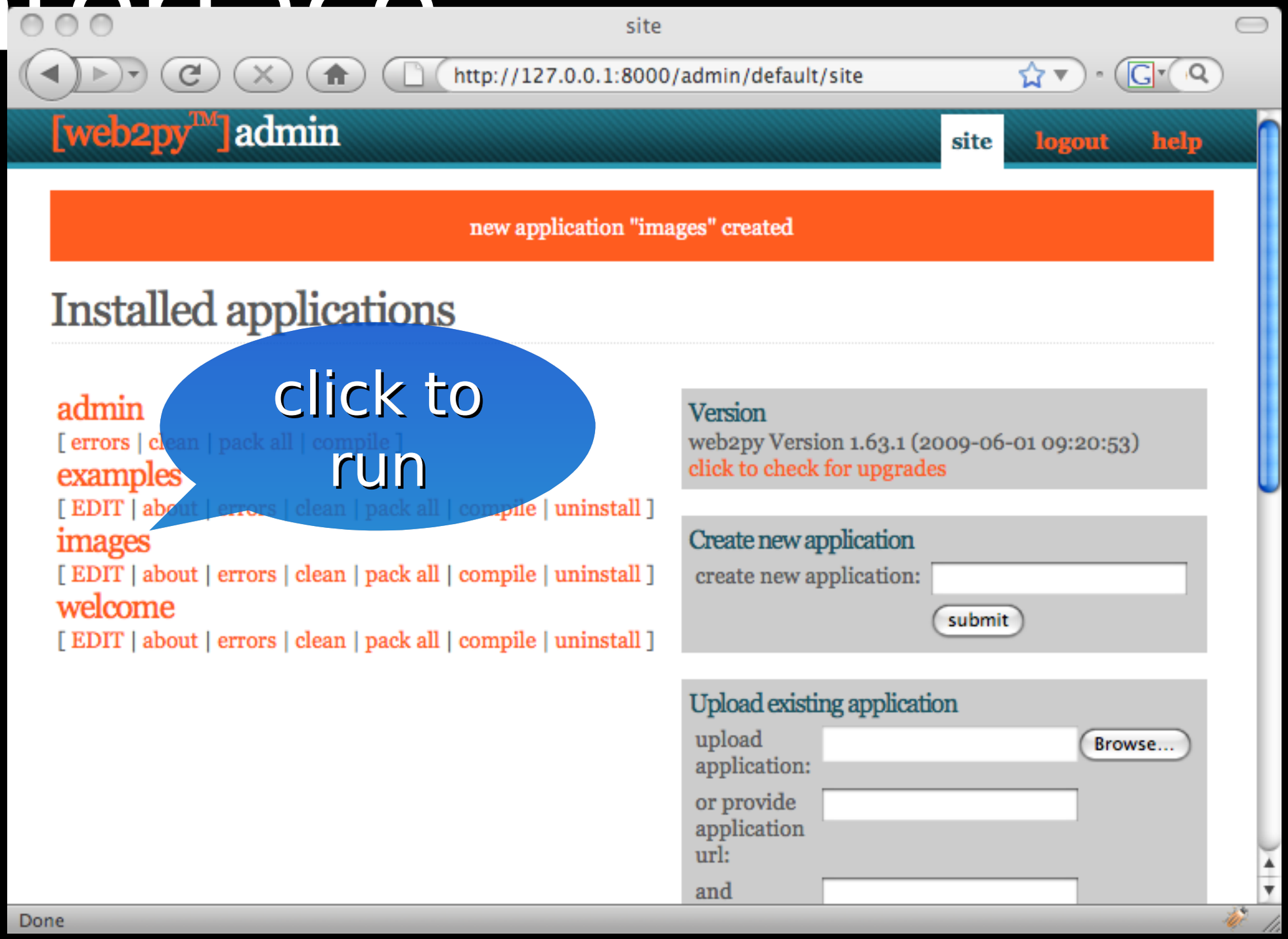


Web Based Admin Interface

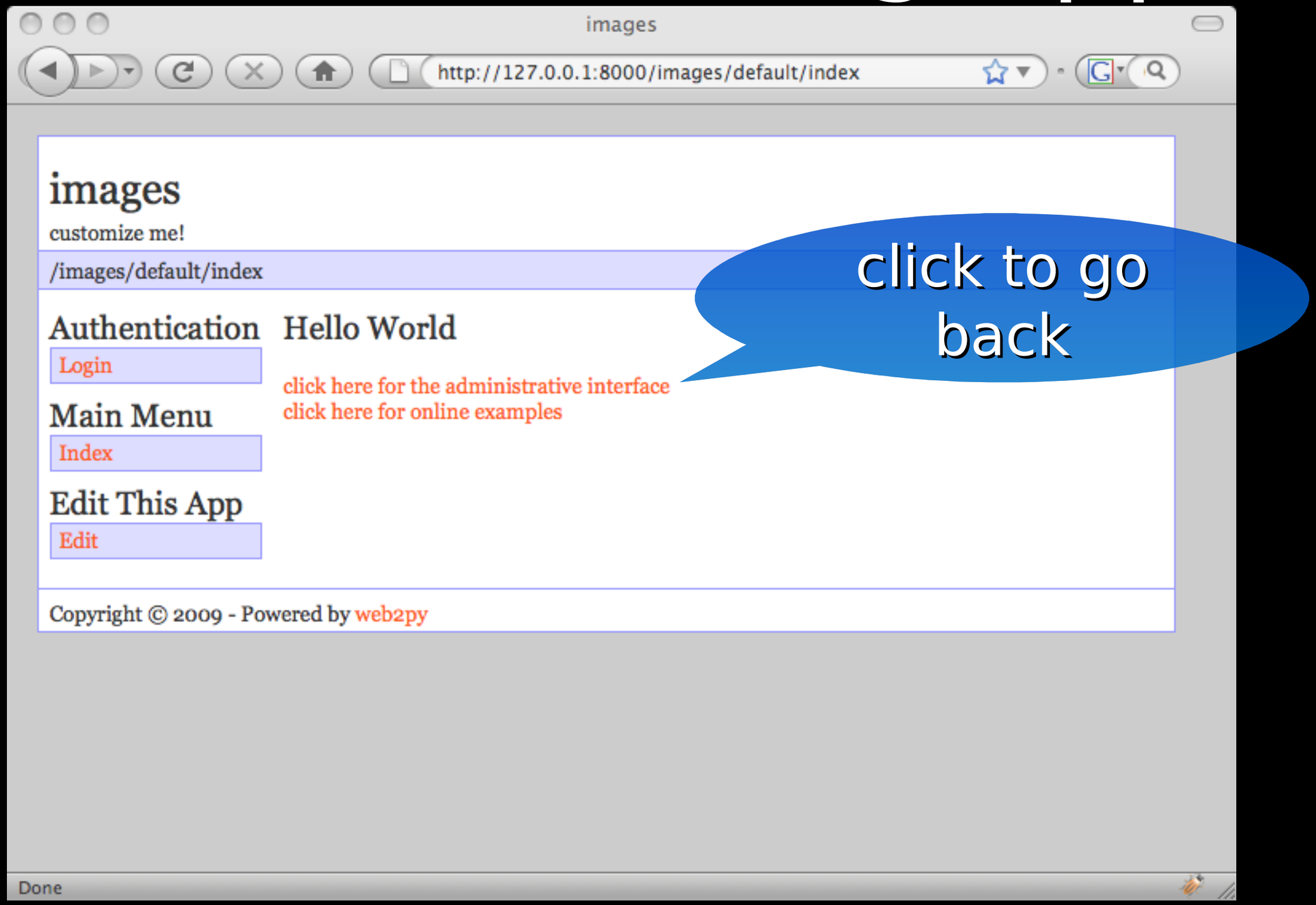


type "images"

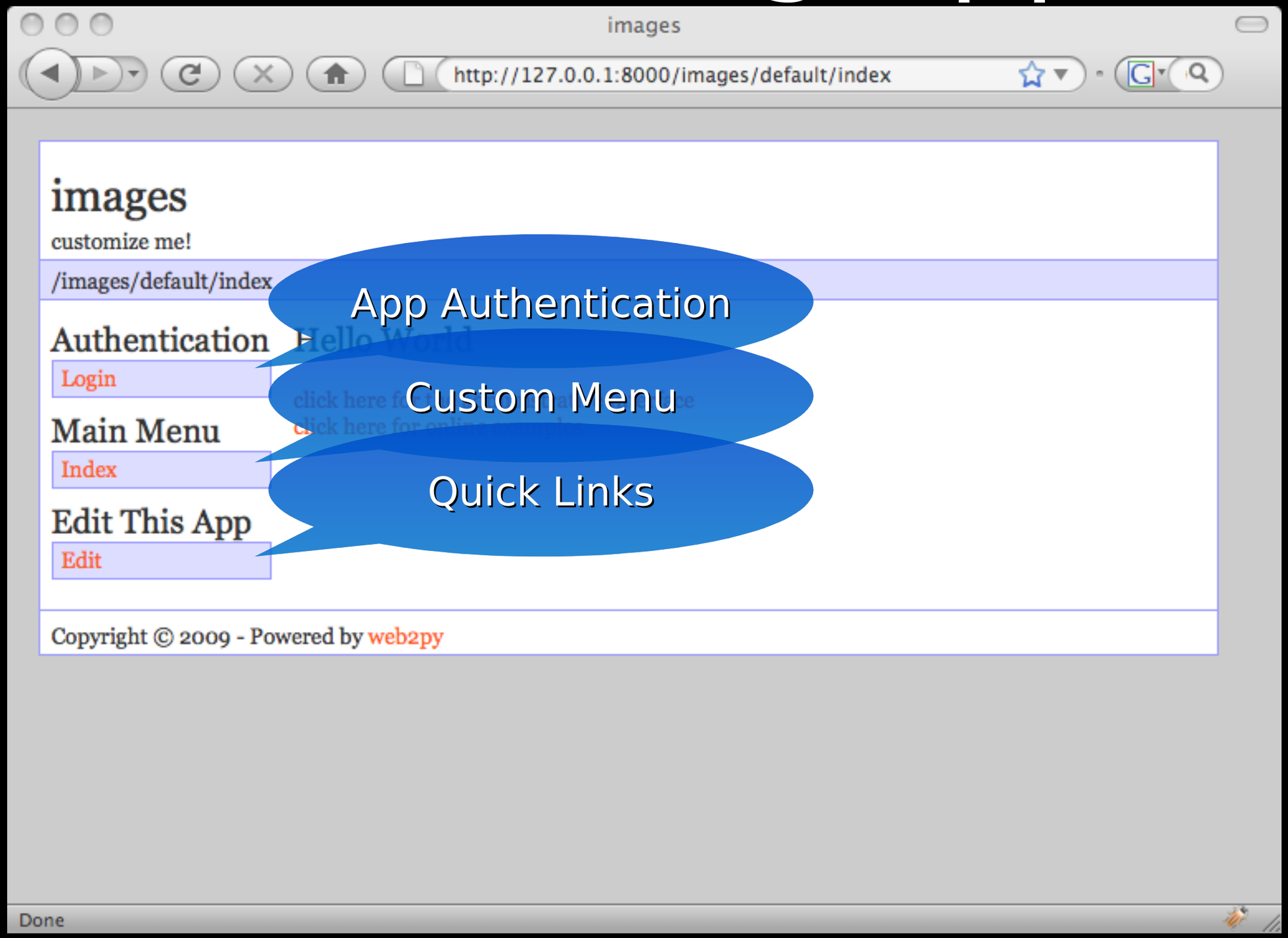
Web Based Admin Interface



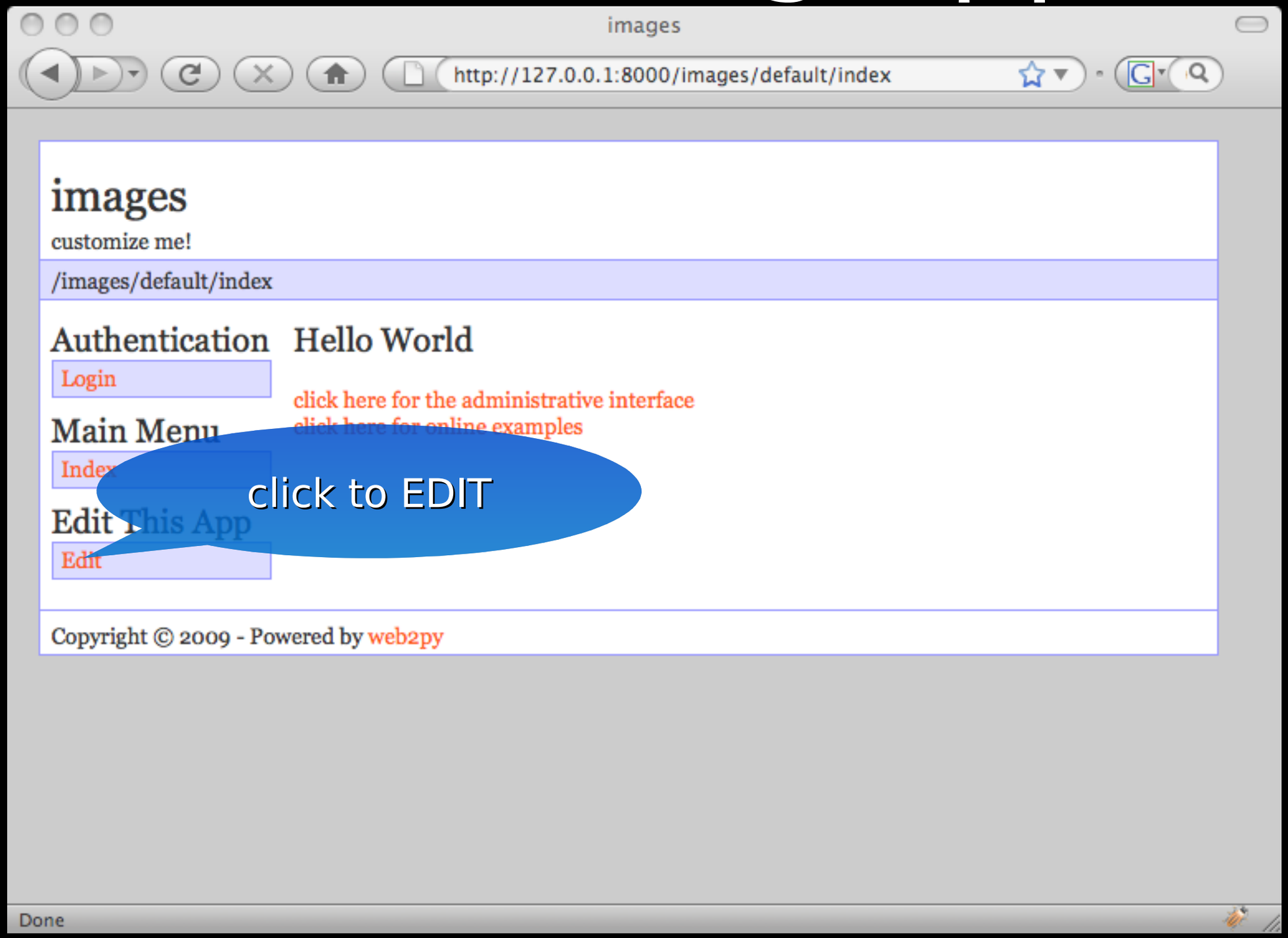
The Scaffolding App



The Scaffolding App



The Scaffolding App



Edit an App Metadata

about images

http://127.0.0.1:8000/admin/default/about/images

[web2py™] admin

site | edit | about | errors | versioning | logout | help

About application "images"

About images
[edit]

what is this app about?

Write something about this app. Developed with web2py.

License for images
[edit]

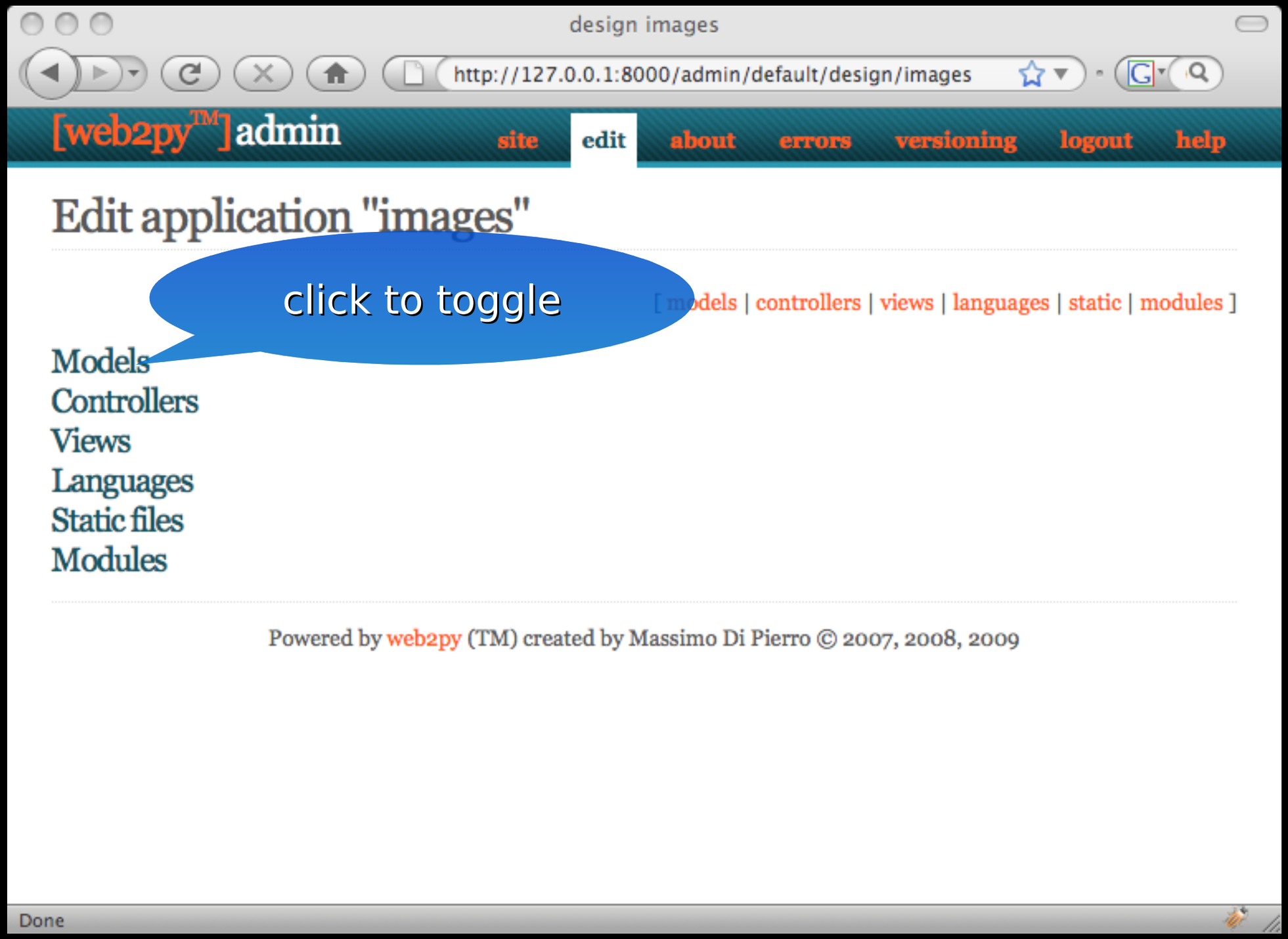
what is its license?

This is a sample license. You can write here anything you want as long as you do not violate web2py copyright, trademark and license.

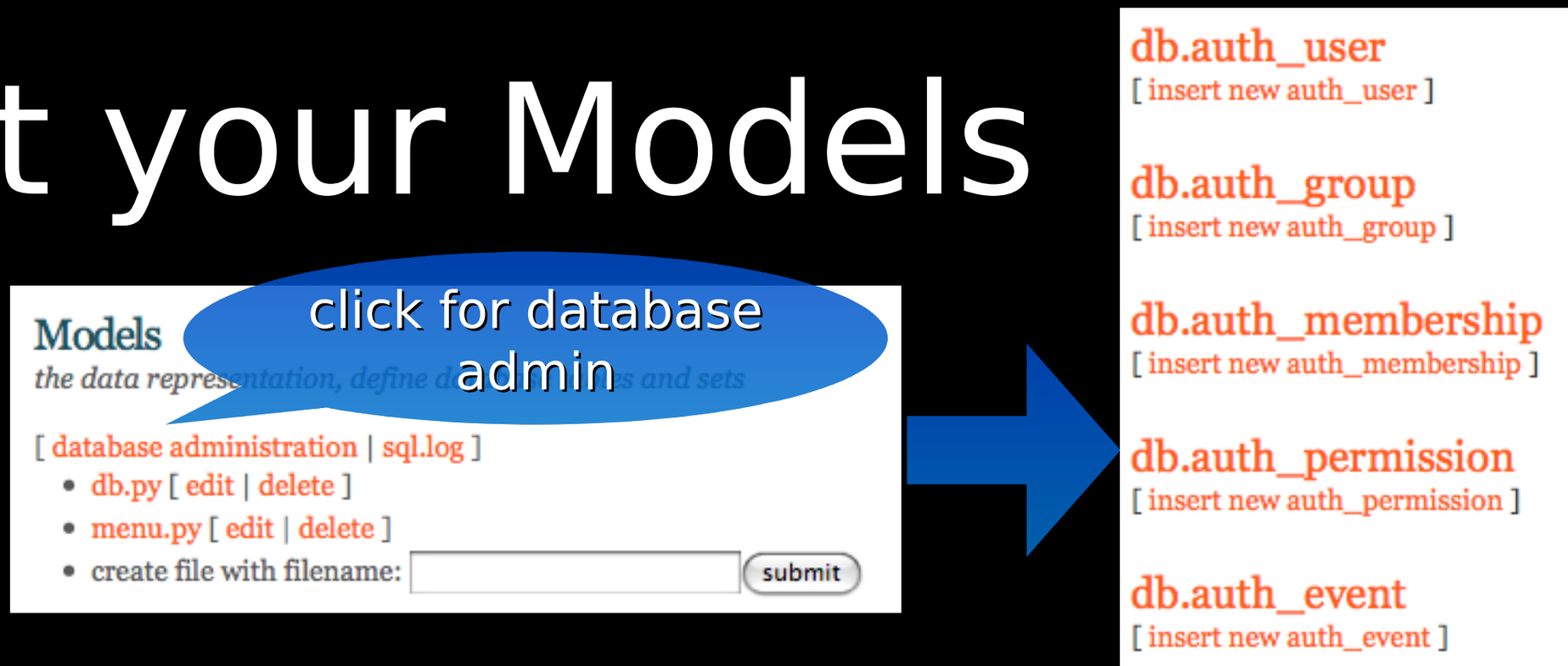
Powered by web2py (TM) created by Massimo Di Pierro © 2007, 2008, 2009

Done

Edit an App


design images
http://127.0.0.1:8000/admin/default/design/images
[web2py™] admin
site
edit
about
errors
versioning
logout
help
Edit application "images"
click to toggle
[models | controllers | views | languages | static | modules]
Models
Controllers
Views
Languages
Static files
Modules
Powered by web2py (TM) created by Massimo Di Pierro © 2007, 2008, 2009
Done

Edit your Models

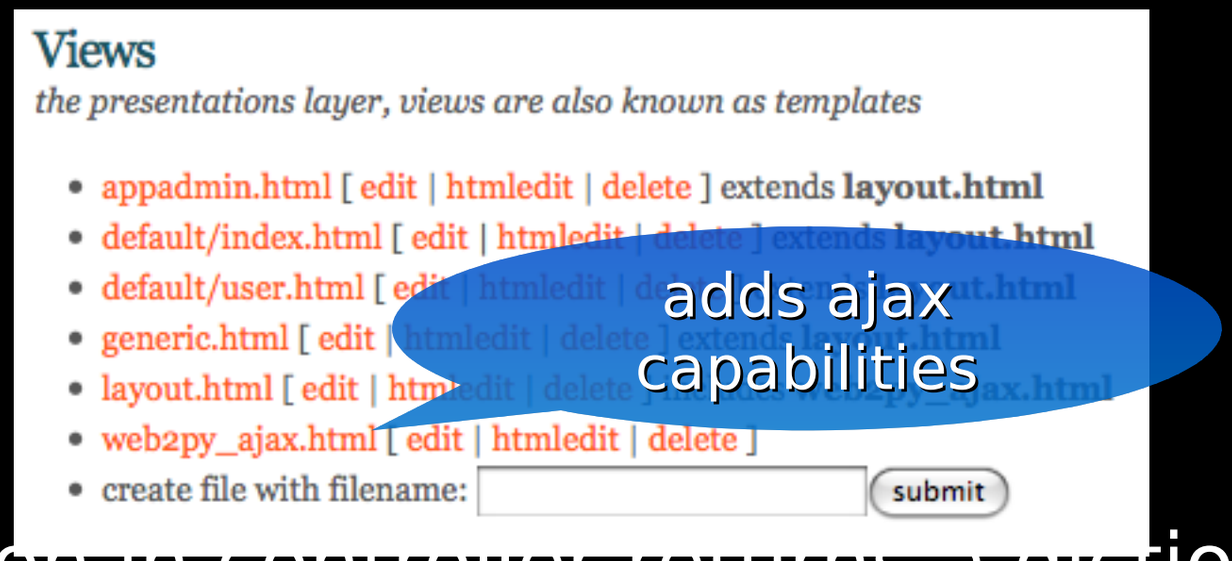


- Models describe the "data representation" of you app
- db.py connects to DB, defines tables, Auth, Crud (edit this file to add tables)
- menu.py defines menus for the scaffoling app (can be removed if not needed)

Edit your Controllers

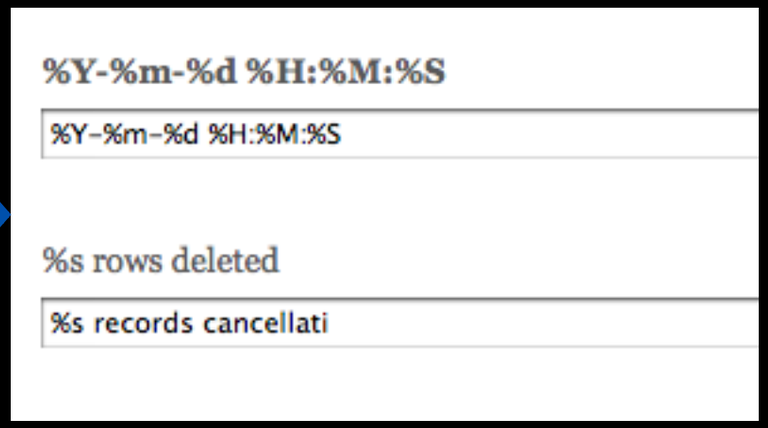
- Controllers describe the workflow of your app
- default.py is the default (entry point) controller of your app
- appadmin.py defines a database administrative interface for your app (appadmin)
- click on [test] to run all doctests online

Edit your Views



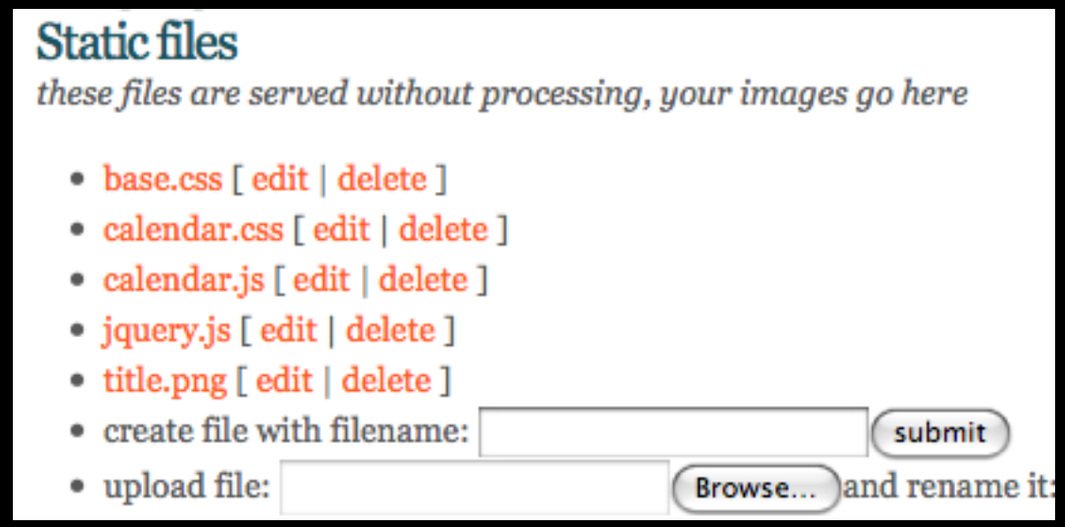
- Each function in controller returns a dictionary that is rendered by a view file. A function can have multiple views with different extensions (html, xml, json, rss, etc.)
- Views can extend other views (for example "layout.html")
- Functions without views use "generic.*" views.

Edit your Languages



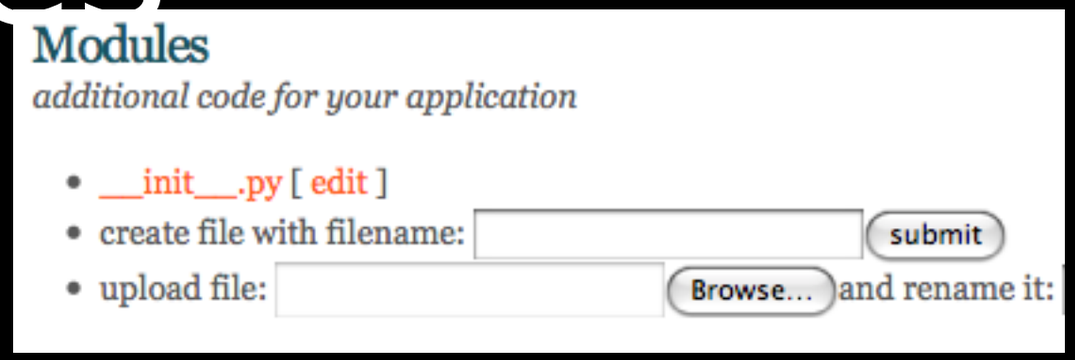
- Apps have language files. They contain translations of strings marked as T("to be translated")

Upload your Static Files



- look into view "layout.html" for example of how to inlude static files, for example
-
- Use of URL function is important for reverse url mapping

Upload/Edit your Modules



- every app can have its own modules. Import them with
- exec("import applications.%s.modules.mymodule as mymodule" % request.application)

Edit Files Online

edit images/controllers/default.py

http://127.0.0.1:8000/admin/default/edit/images/contro

[web2py™] admin

site edit about errors versioning logout help

Editing file "images/controllers/default.py"

exposes index, user, download, call

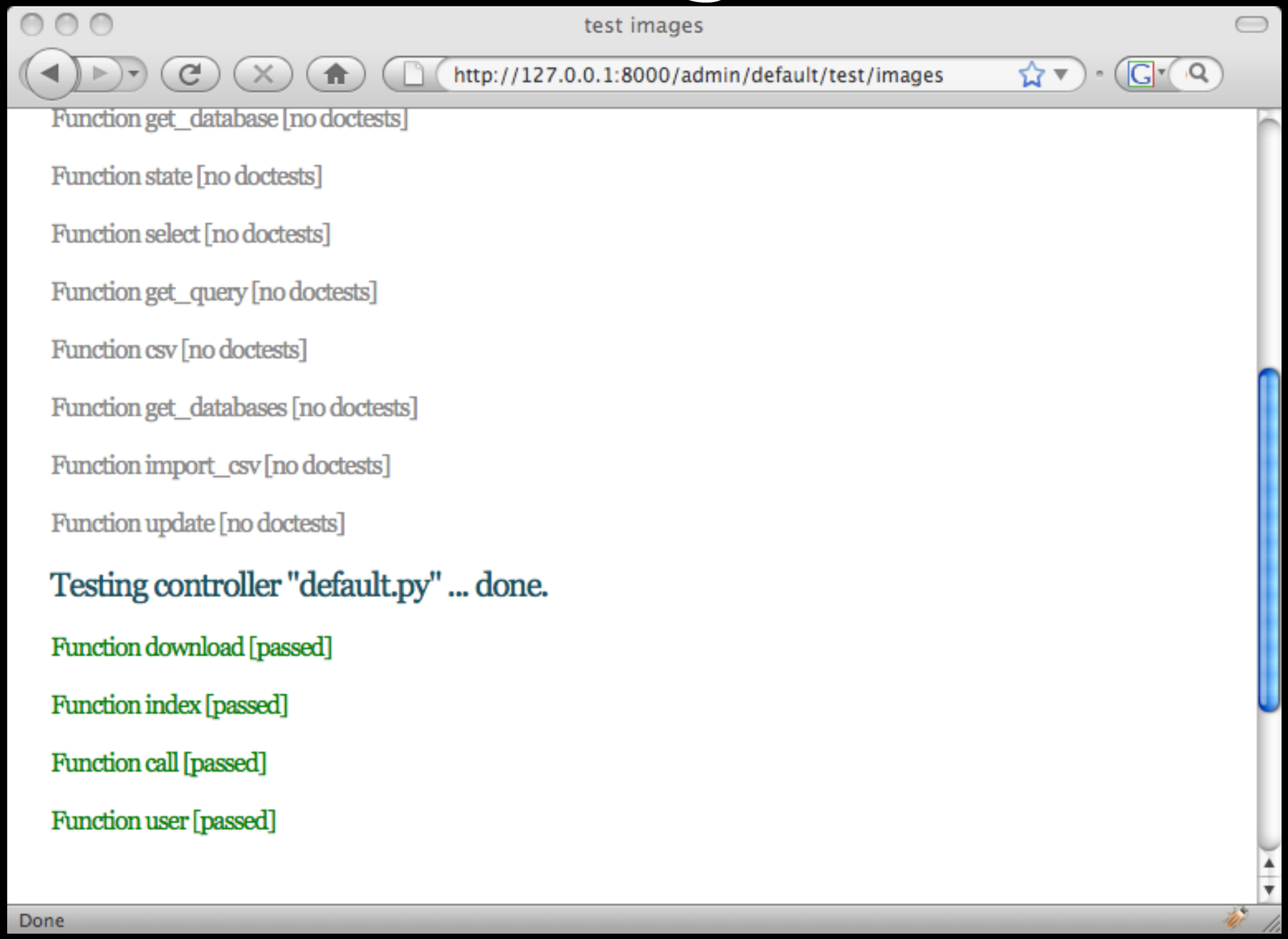
save Saved file hash: 0144abbe38eaab84db58a6 Last saved on: Mon Jun 1 13:42:54 2009

10 pt

```
1  ##########################################################################
2  ## This is a samples controller
3  ## - index is the default action of any application
4  ## - user is required for authentication and authorization
5  ## - download is for downloading files uploaded in the db (does streaming)
6  ## - call exposes all registered services (none by default)
7  ##########################################################################
8
9  def index():
10     """
11     example action using the internationalization operator T and flash
12     rendered by views/default/index.html or views/generic.html
13     """
14     response.flash = T('Welcome to web2py')
15     return dict(message=T('Hello World'))
16
17
18 def user():
19     """
```

Done

Online Testing



Auto File Conflict Resolution

resolve images/views/default/list_images.json

http://127.0.0.1:8000/admin/default/resolve/images/vie

resolve images/views/default/... | ticket images/127.0.0.1.2009-0...

[web2py™] admin

site edit about errors versioning logout help

file changed on disk

Resolve Conflict file "images/views/default/list_images.json"

new old all

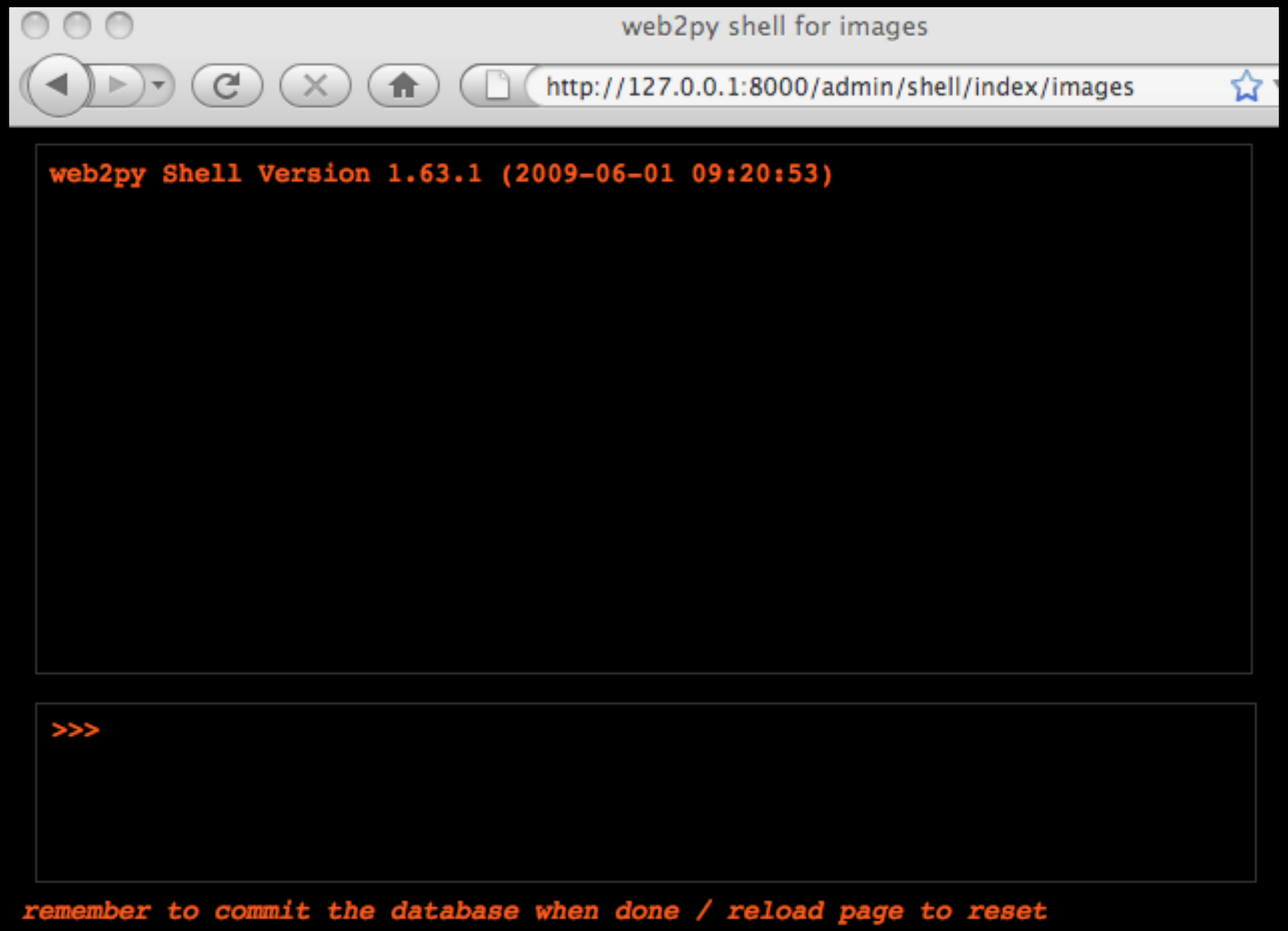
```
{{
from gluon.serializers import json
respone.write(json(response._vars.images.as_list()),escape=False)
}}
```

merge

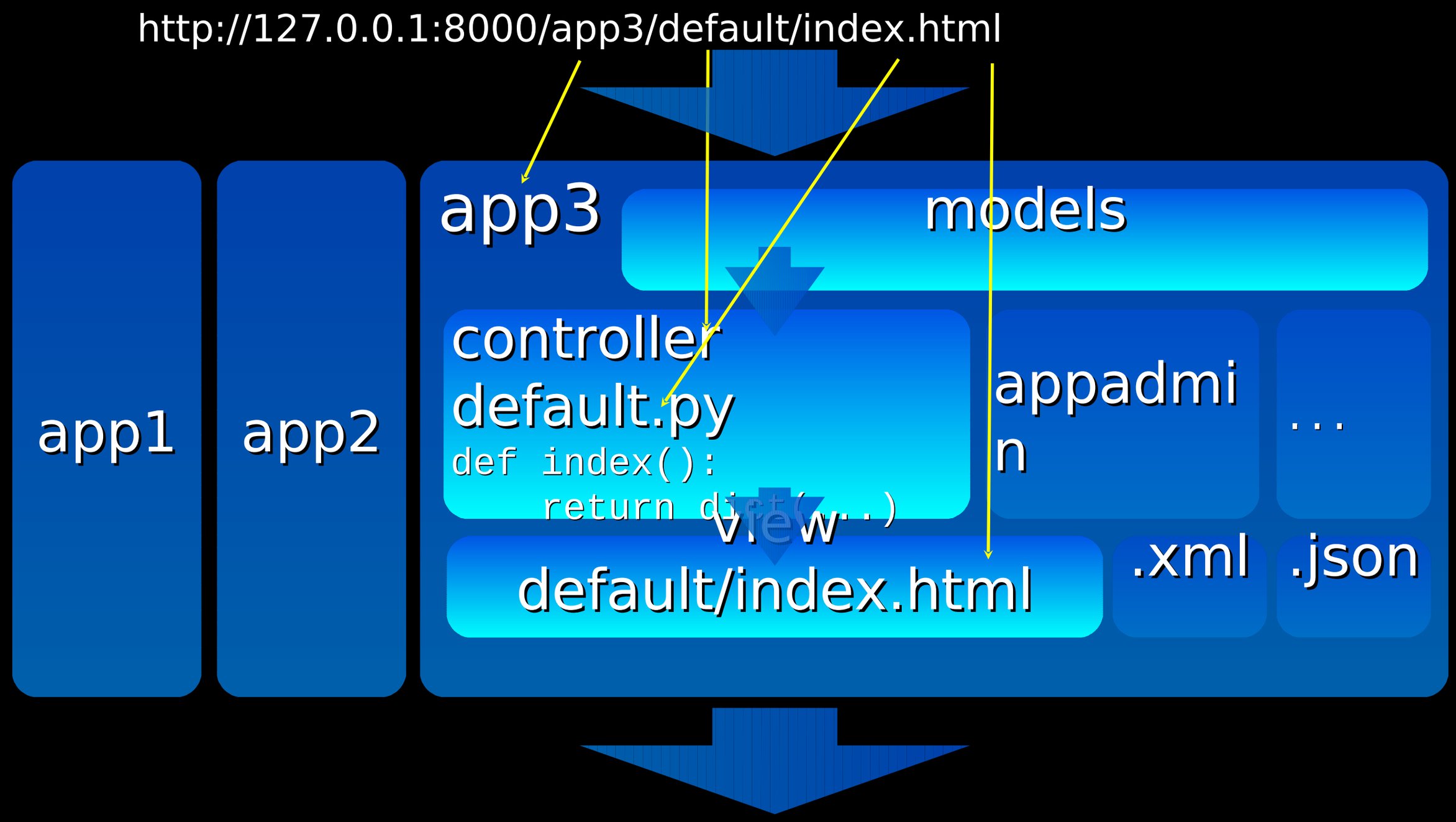
Powered by web2py (TM) created by Massimo Di Pierro © 2007, 2008, 2009

Done

Ajax Shell


web2py shell for images
http://127.0.0.1:8000/admin/shell/index/images
web2py Shell Version 1.63.1 (2009-06-01 09:20:53)
>>>
remember to commit the database when done / reload page to reset

web2py Control Flow



web2py URL Parsing

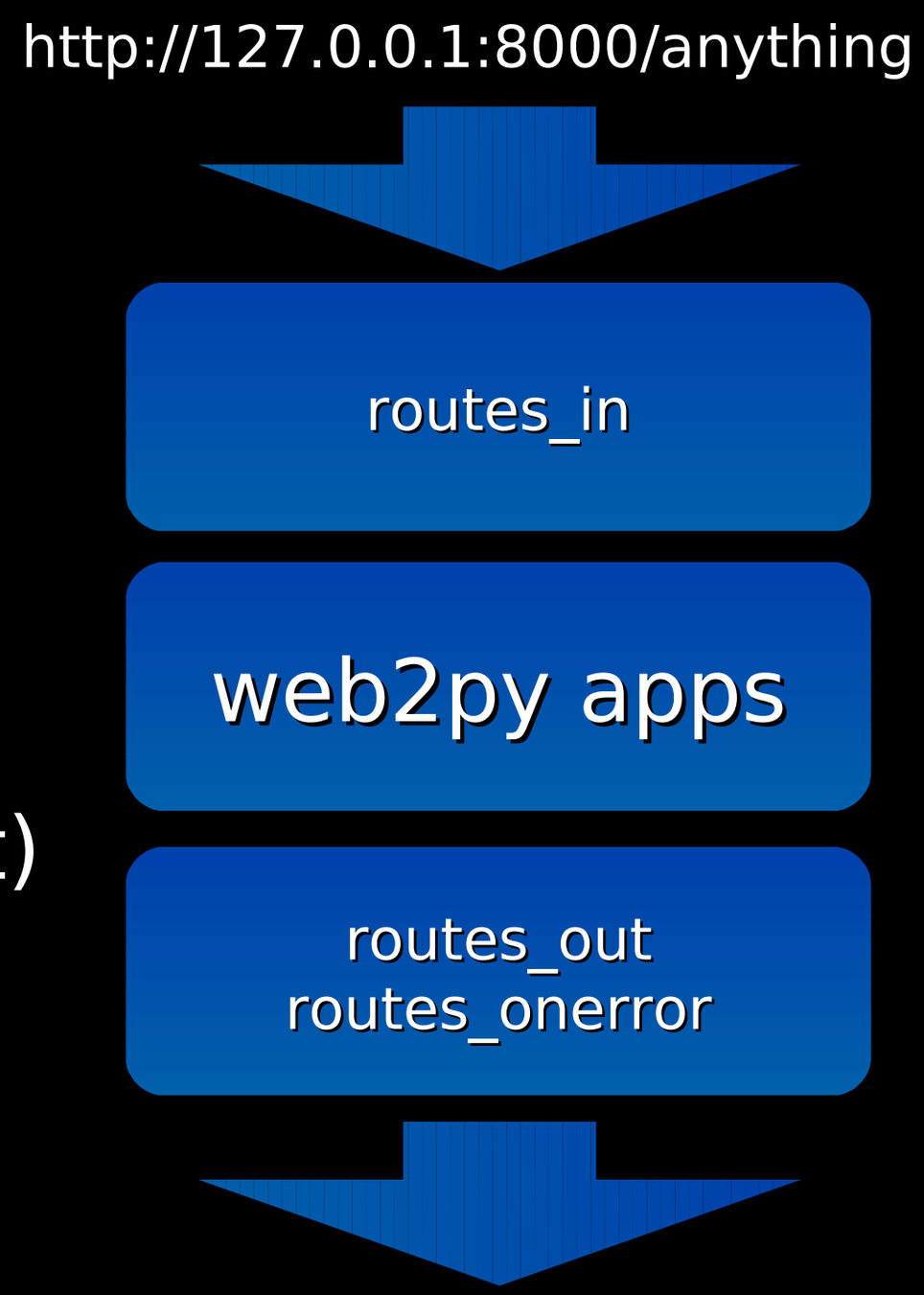
http://127.0.0.1:8000/app3/default/index/a/b/c.html?name=Max

```
request.application == ‘app3’

request.controller == ‘default’

request.function ==’index’

request.args == [’a’,’b’,’c’]

request.extension == ‘html’

request.vars.name == ‘Max’
```

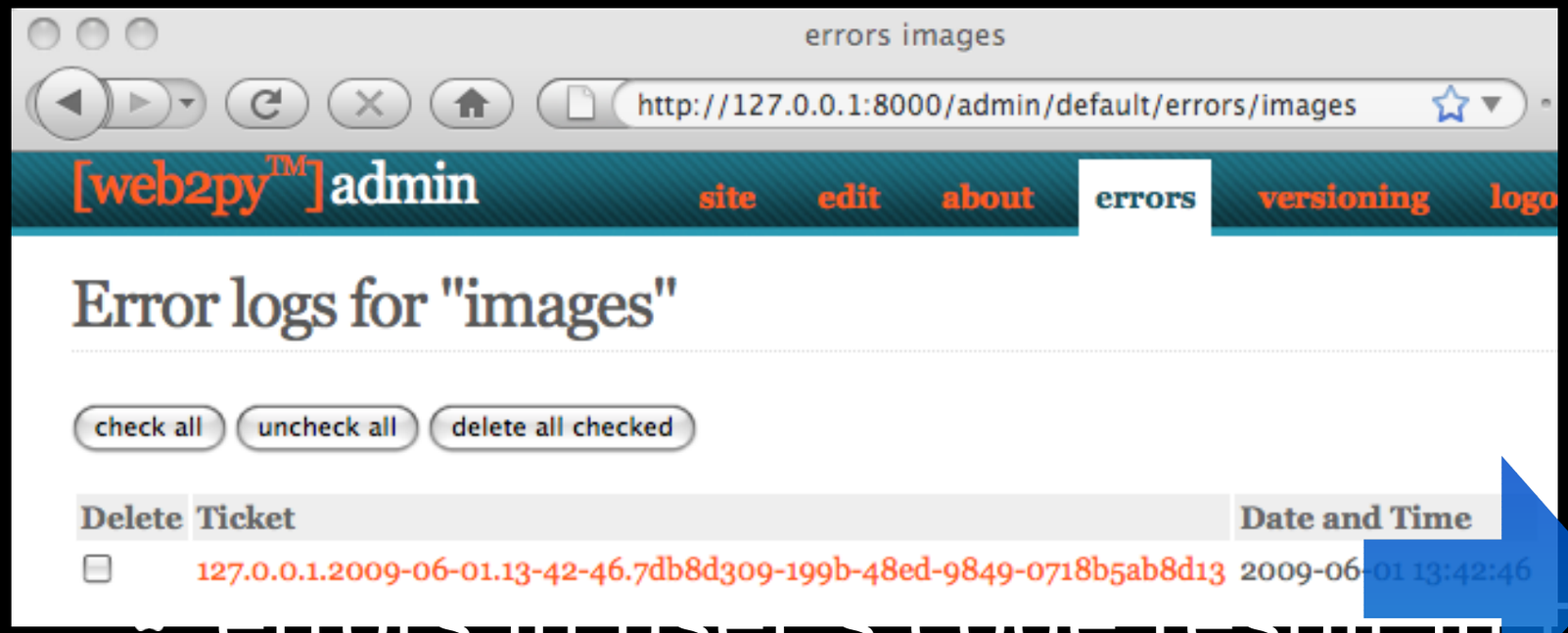
Environment variables are in request.env

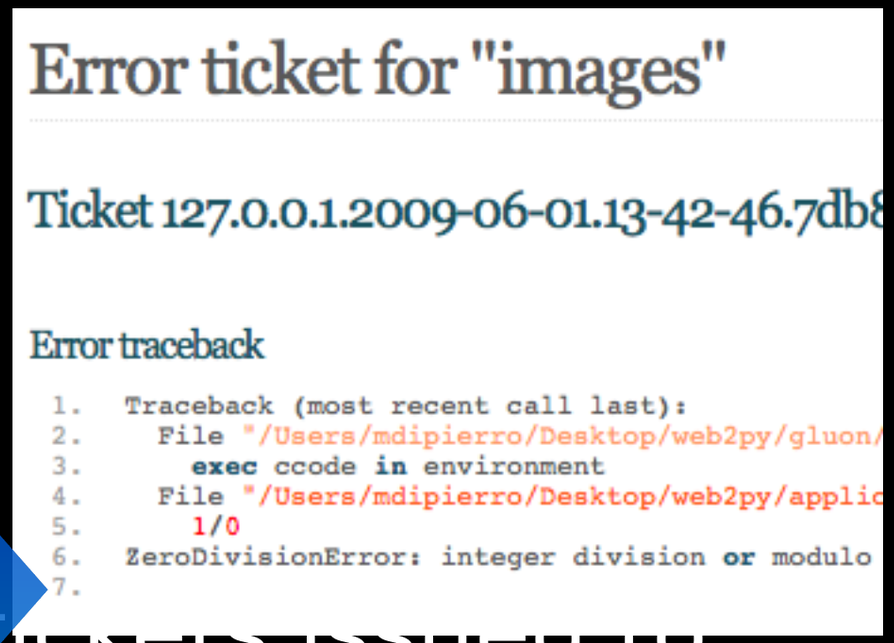
routes.py (optional)

- incoming URLs can be filtered and rewritten (routes_in)
- URLs generated by web2py can also be filtered (routes_onerror) and rewritten (routes_out)
- rename routes.example.py as routes.py and edit.



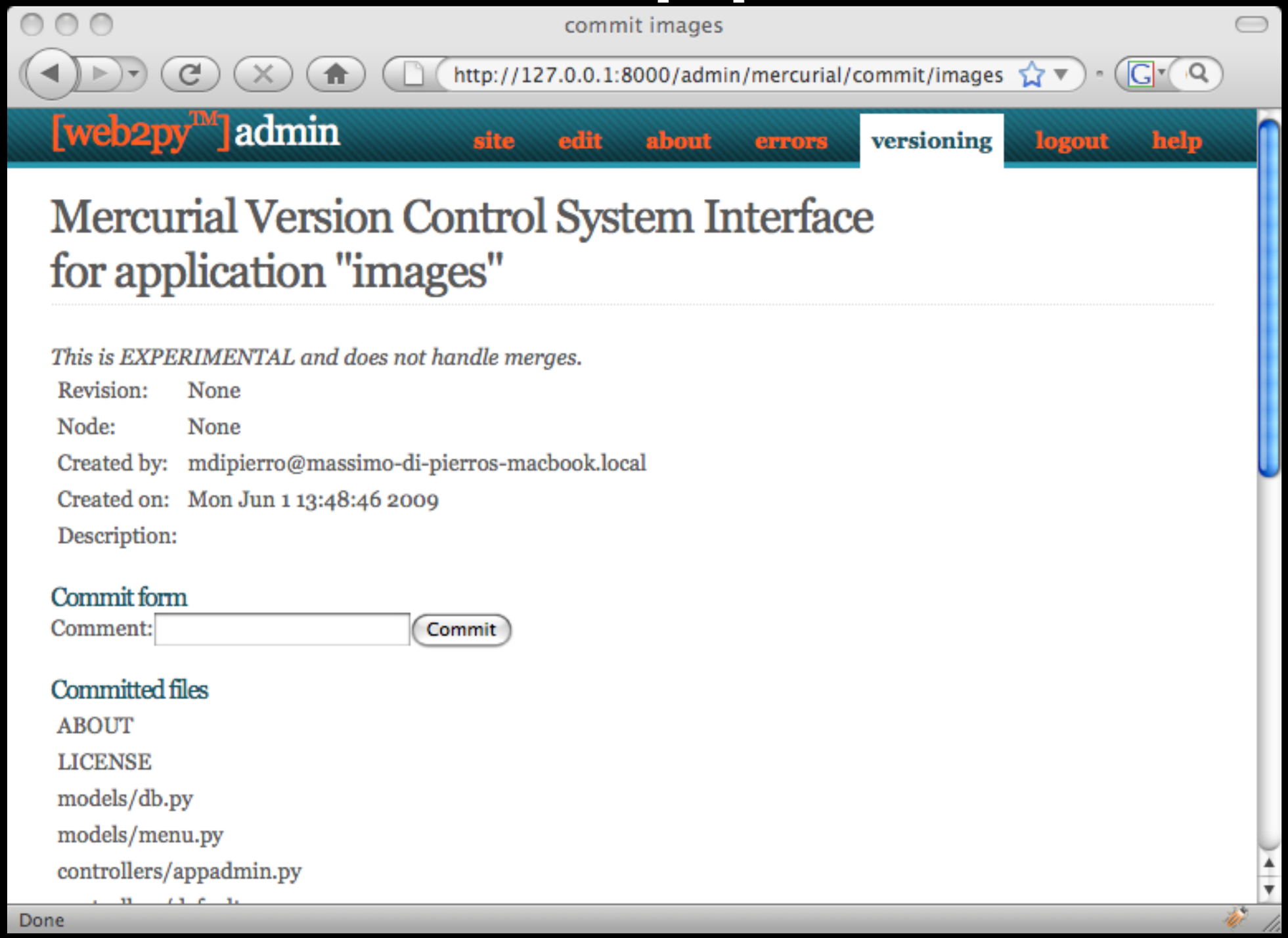
Errors and Tickets



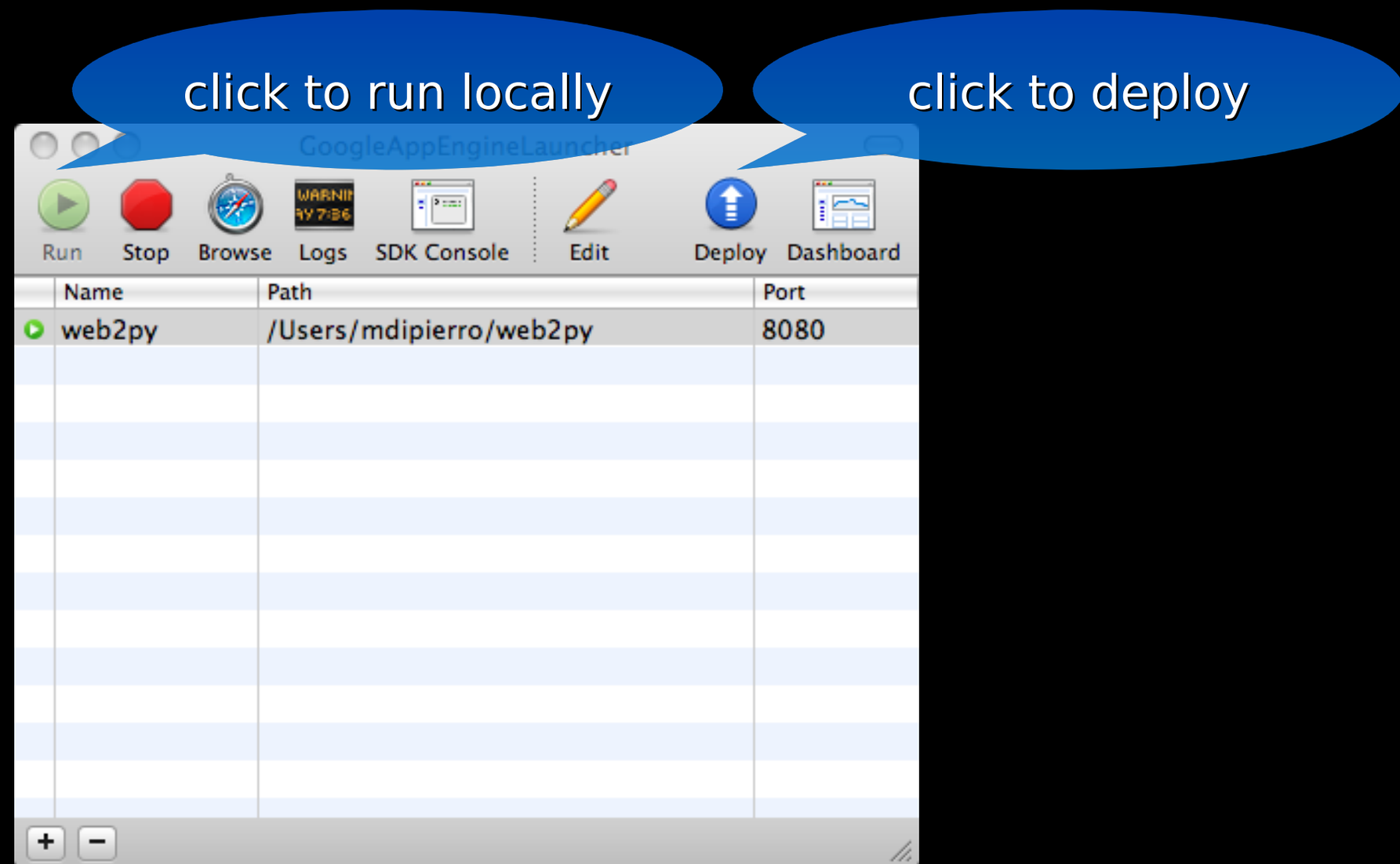


- errors in user's code result in tickets issued to visitors
- administrator can retrieve and read tickets online
- click on any web2py keyword to get documentation

Mercurial Support (beta)



Deploy on Google App Engine



Edit App "images"

What do we want?

- Visitor can post images
- Images have category
- Visitors can post comments

How do we do it?

- edit "images" model "db.py" file and append

```
db.define_table(‘category’,
    SQLField(‘name’))
db.define_table(‘image’,
    SQLField(‘category_id’,db.category),   # reference field
    SQLField(‘title’, ‘string’),
    SQLField(‘description’, ‘text’),
    SQLField(‘file’,’upload’),             # something to be uploaded
    SQLField(‘posted_by’,db.auth_user))    # reference field
db.define_table(‘comment’,
    SQLField(‘image_id’,db.image),         # reference field
    SQLField(‘body’,’text’),
    SQLField(‘posted_by’,db.auth_user),    # reference field
    SQLField(‘posted_on’,’datetime’))
```

Create Categories Using "appadmin"

Some Details

- edit "images" model "db.py" and append

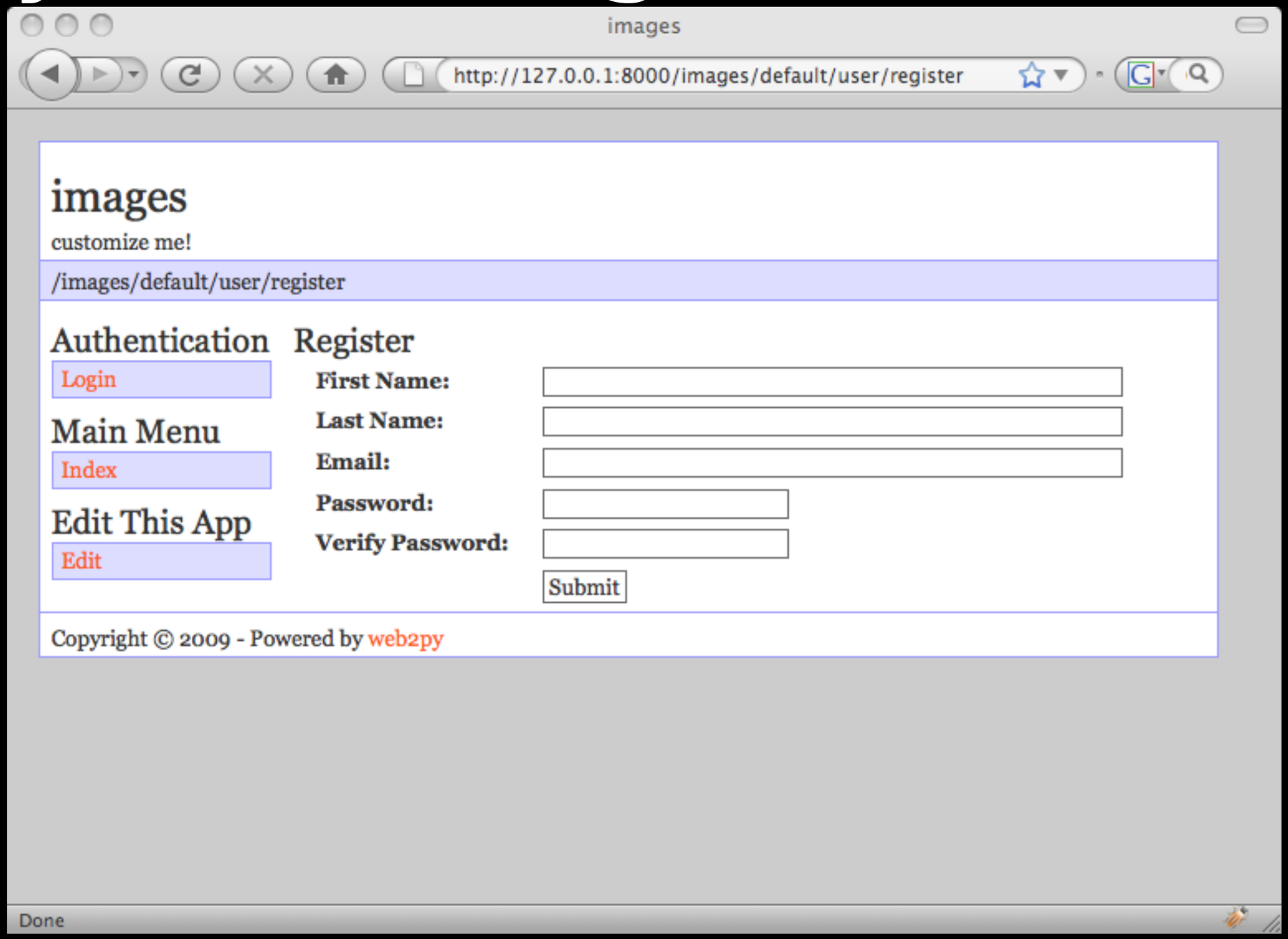
```
# make categories unique
db.category.name.requires = IS_NOT_IN_DB(db,’category.name’)

# category_id to be represented by category name
db.image.category_id.requires = IS_IN_DB(db,’category.id’,’%(name)s’)
db.image.title.requires = IS_NOT_EMPTY()

# auto fill posted_by fields and do not show them in forms
db.image.posted_by.default = auth.user.id if auth.user else 0
db.image.posted_by.writable = db.image.posted_by.readable=False
db.comment.posted_by.default = auth.user.id if auth.user else 0
db.comment.posted_by.writable = db.comment.posted_by.readable = False

# auto fill posted_on field and make it readonly
db.comment.posted_on.default = request.now
db.comment.posted_on.writable = False
```

Some Actions

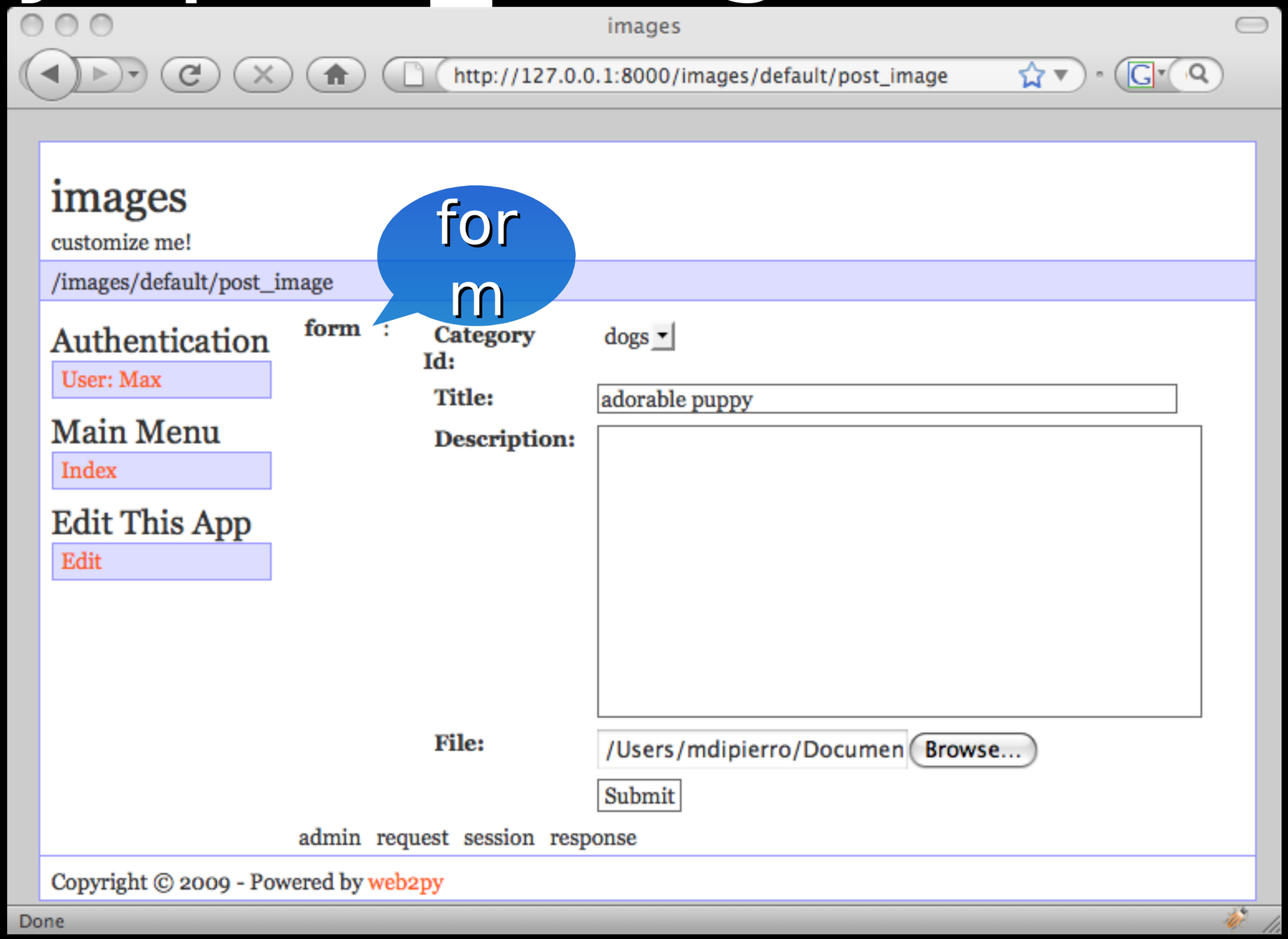
- edit "images" controller "default.py" and insert

```
def list_images():
    return dict(images=db(db.image.id>0).select())

@auth.requires_login()
def post_image():
    return dict(form=crud.create(db.image))

@auth.requires_login()
def view_image():
    image_id = request.args(0) or redirect(URL(r=request,f='index'))
    db.comment.image_id.default = image_id
    db.comment.image_id.writable = db.comment.image_id.readable = False
    return dict(form1=crud.read(db.image, image_id),
                comments=db(db.comment.image_id==image_id)\
                         .select(orderby=db.comment.posted_on),
                form2=crud.create(db.comment))
```

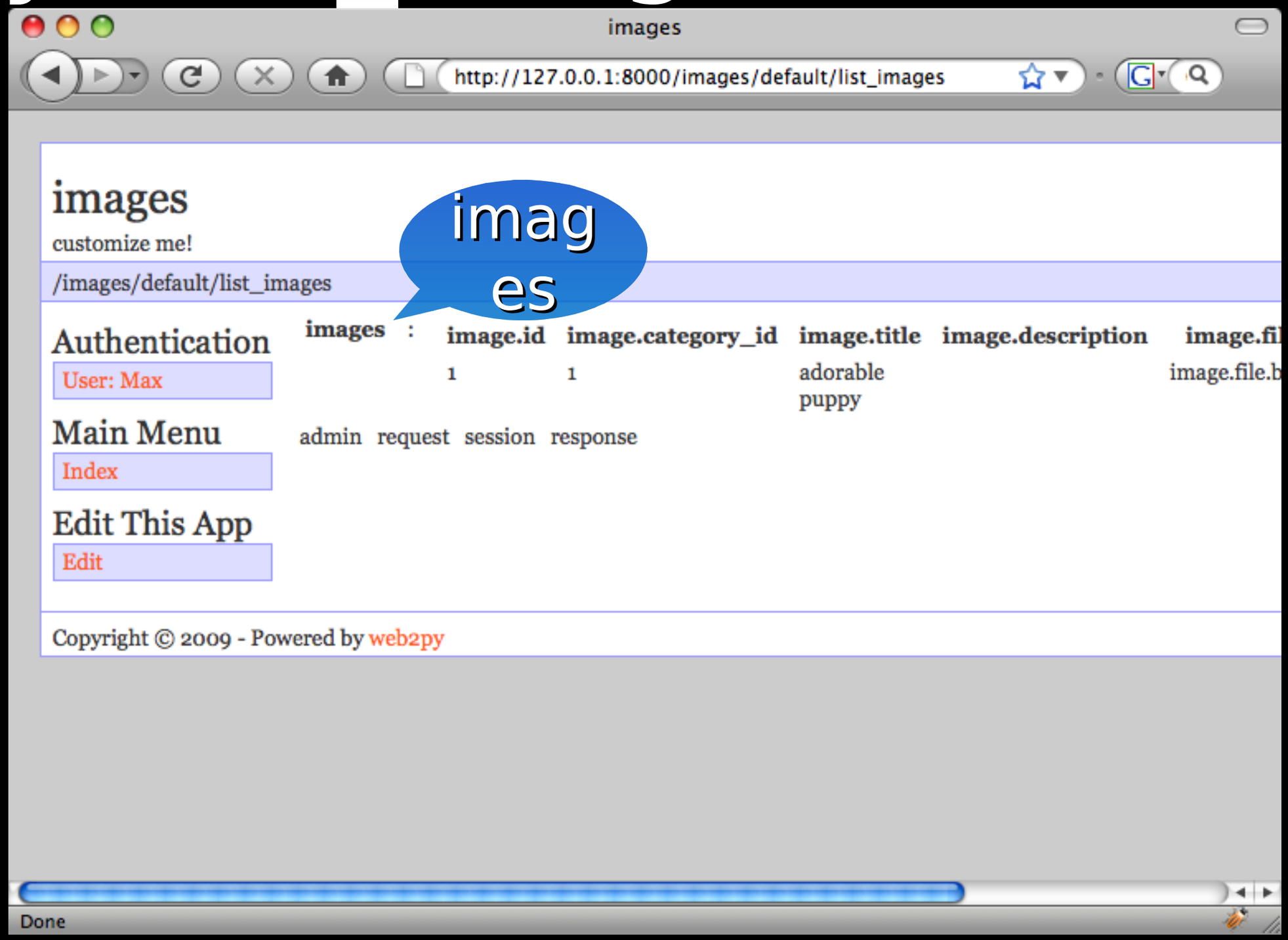
Try "user/register"



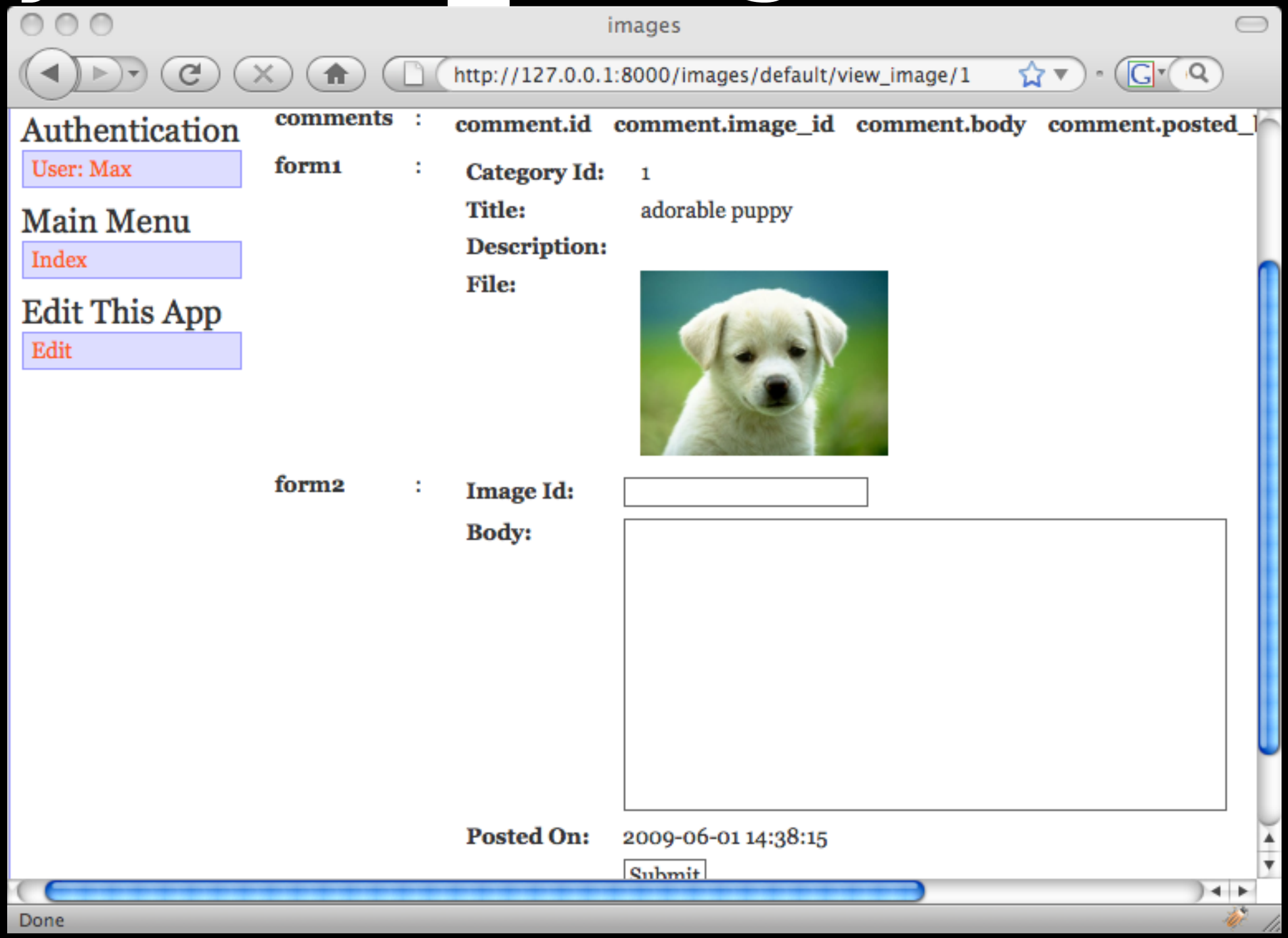
Try "post_image"



Try "list_images"



Try "view_image/1"



Things to Notice...

- all form processing is already there
- all functions (actions) have default views
- crud.update/crud.read have image previews
- uploaded files (image.file) are served by the "download" action in the scaffolding app
- You need to register and login to post and comment

Further Details

- Fields can be customized further in models or actions

```
·# change label
·db.image.posted_by.label = 'Posted by'

·# change representation
·db.image.posted_by.represent = lambda value: \
·   '%(first_name)s %(last_name)s' % db.auth_user[value]

·# custom comments
·db.image.posted_by.comment = 'set automatically'

·# automatically set when records are updated
·db.image.posted_by.update = auth.user.id if auth.user else 0
```

- (readable/writable=False does not affect appadmin)

Add view default/list_images.html

- create "images" – view "default/list_images.html" and edit

```
{{extend ‘layout.html’}}


<h1>Posted Images</h1>

<ul>

  {{for image in images:}}

  <li>{{=A(image.title,

           _href=URL(r=request,f=’view_image’,args=image.id))}}</li>

  {{pass}}

</ul>


{{=A(‘post image’,_href=URL(r=request,f=’post_image’))}}
```

Add view "default/post_image.html"

- create "images" view "default/post_image.html" and edit

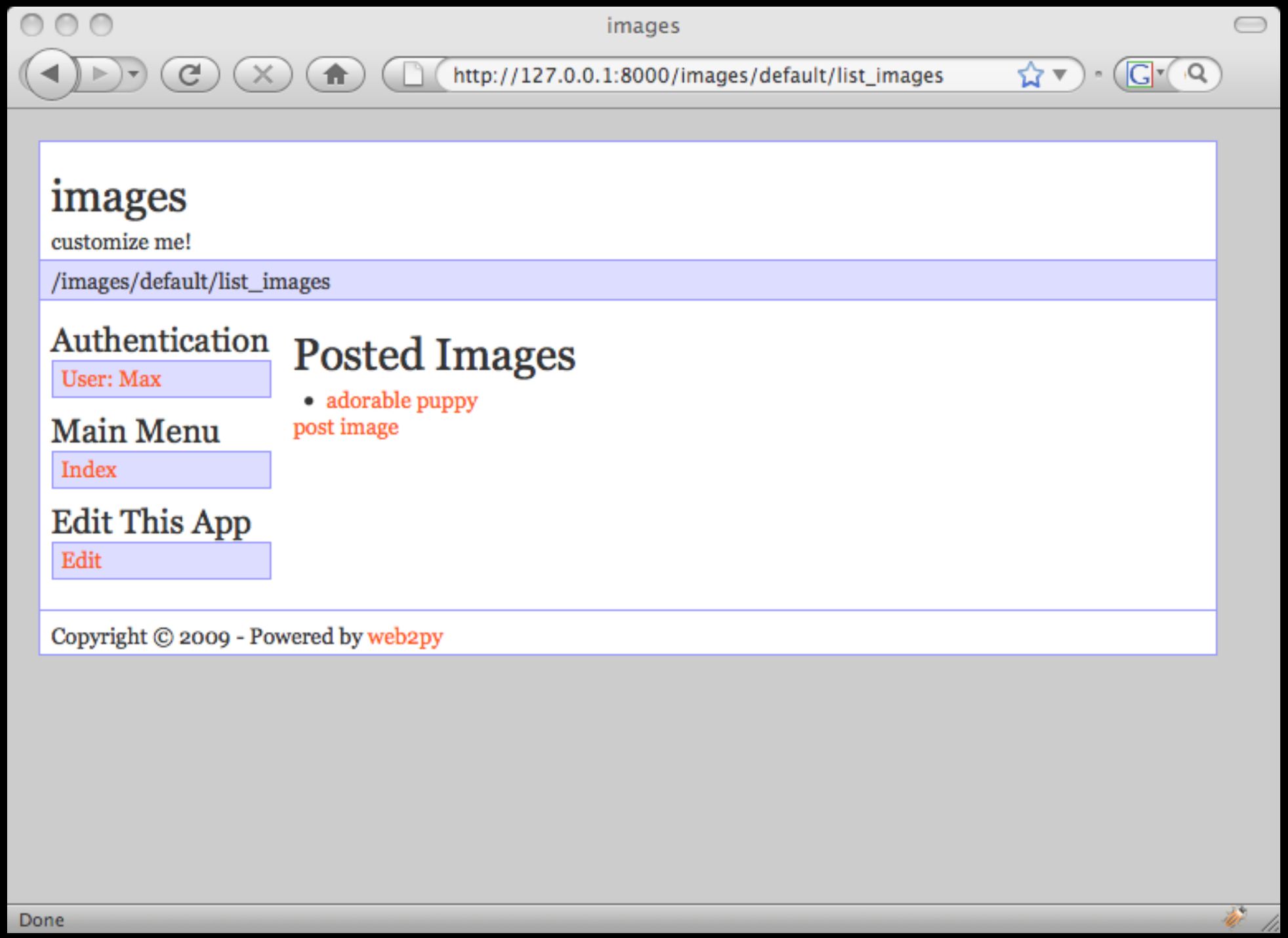
```
{{extend ‘layout.html’}}

<h1>Post an Image</h1>

{{=form}}
```

Add view "default/view_image.html"

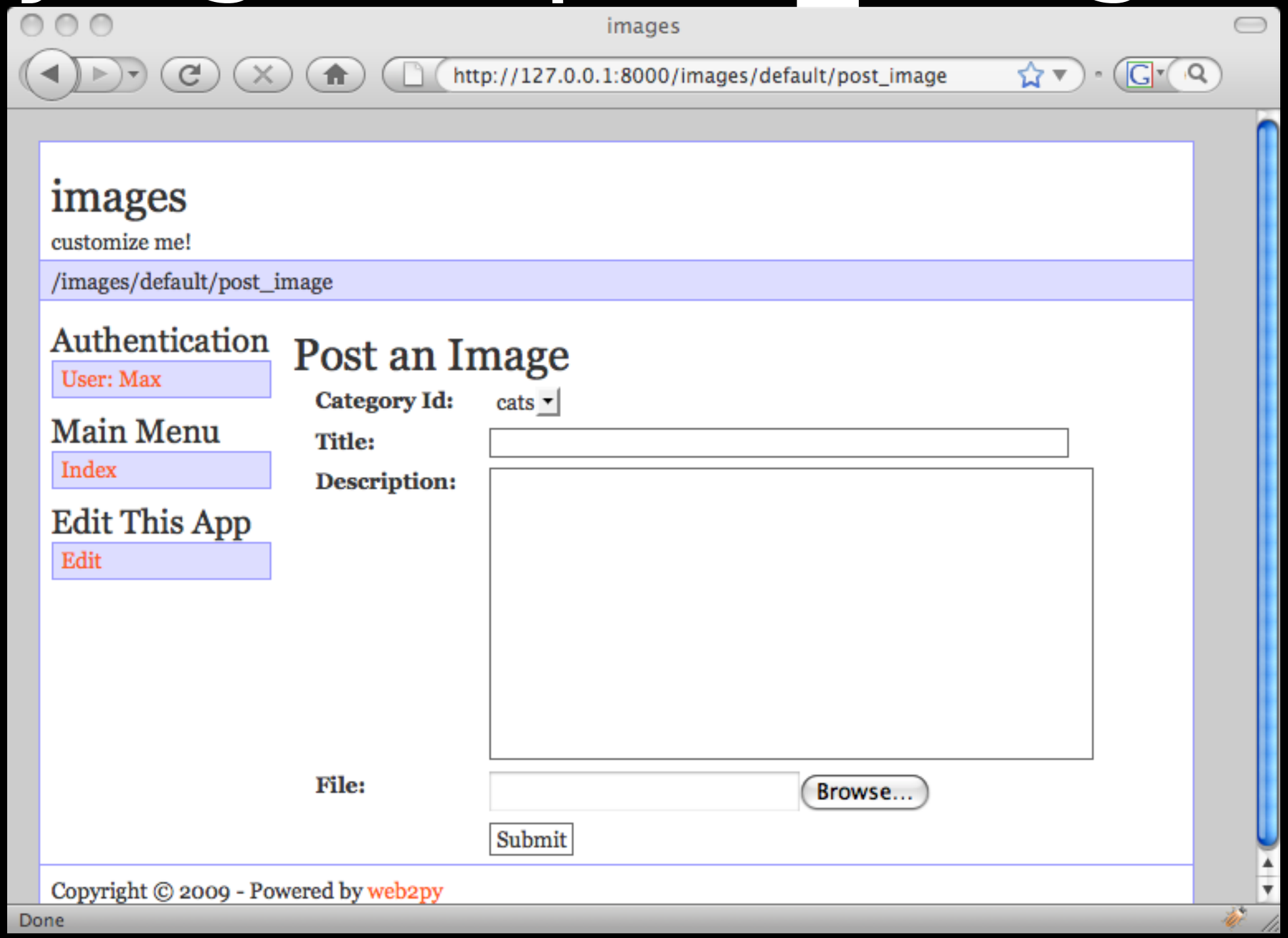
- create "images" view "default/view_image.html" and edit

```
{{extend ‘layout.html’}}

<h1>Image {{=form1.record.title}}</h1>

{{=IMG(_src=URL(r=request,f=’download’,args=form1.record.file))}}

<i>{{=form1.record.description}}</i>

<h2>Comments</h2>

{{for comment in comments:}}{{=db.auth_user[comment.posted_by].first_name}}
   said "{{=comment.body}}" on {{=comment.posted_on}}<br/>

{{pass}}

<h2>Add Comment</h2>

{{=form2}}
```

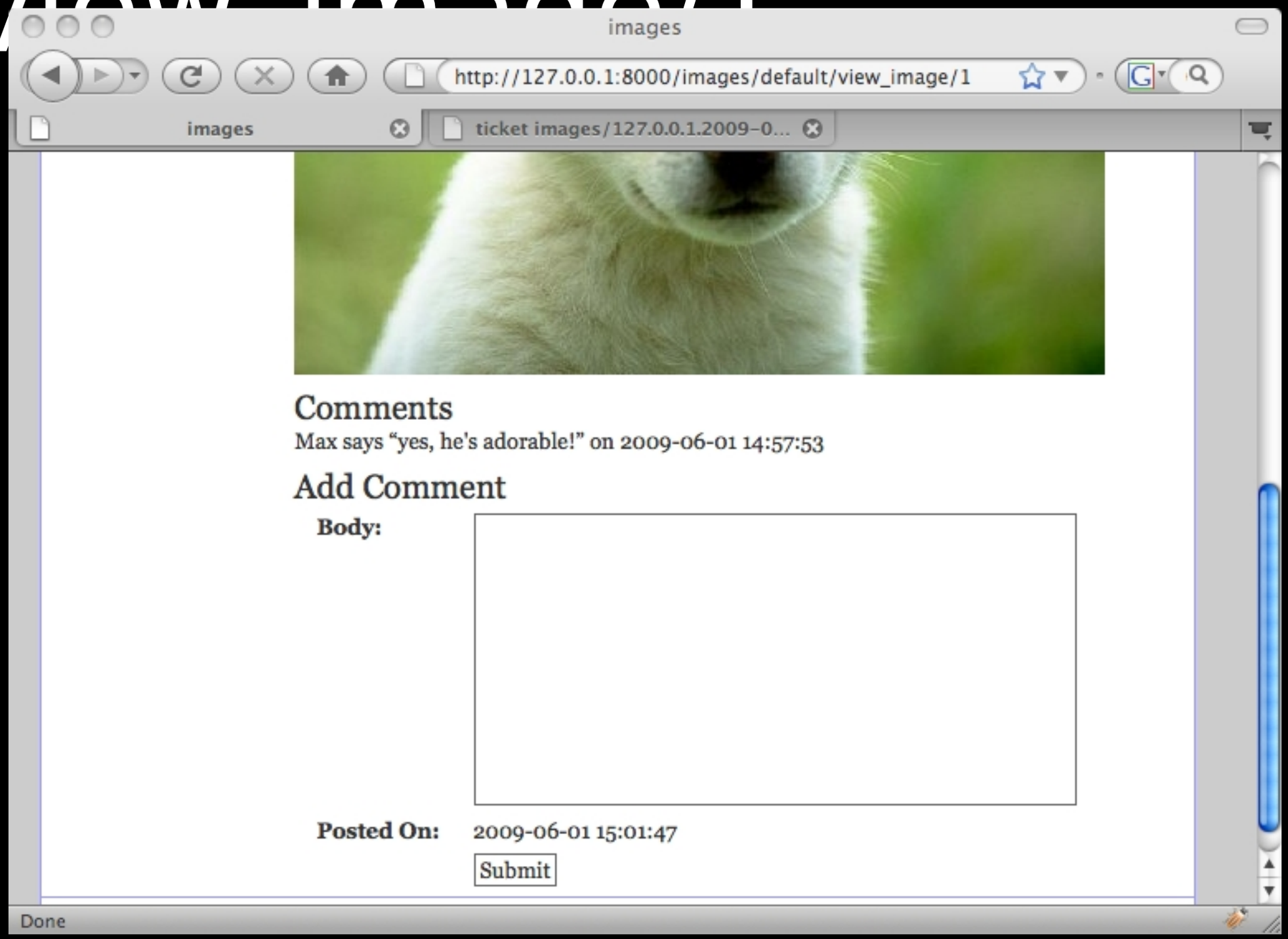
Try "list_images"



Try again "post_image"



Try again "view_image/1"



Menus

- The menu for the scaffolding application is in file models/menu.py
- Menus are rendered by {{=MENU(menu)}} where
- menu is a list of list of the form
  - menu = [[‘item’,False,URL(...),[ ]],]
- False indicates if the link is the current link
- [ ] is an optional submenu

Other "system variables"

- response.menu = [ ] # the official main manu
- response.title = "write your own"
- response.subtitle "best app ever"
- response.author = "you@example.com"
- response.keywords = "puppies, kittens, mice"
- response.description = "silly app"

Changing Layout

- A layout is just a regular html file but it must contain:
  - <head>....{{include ‘web2py_ajax.html’}}</head>
  - {{=response.flash or ""}}
  - {{include}}
- Optionally it can contain:
  - {{=MENU(response.menu or [])}}
  - meta tags (see default layout.html)

Advanced: list_images.xml

- create "images/views/default/list_images.xml" and edit

```
<images>

  {{for image in images:}}

  <image>

     <title>{{=image.title}}</title>

     <description>{{=image.description}}</description>

     <link>{{=URL(r=request,f='download',args=image.file)}}</link>

  </image>

  {{pass}}

</images>
```

- visit http://.../list_images.xml to get XML output

Advanced: list_images.json

- create "images" view "default/list_images.json" and edit

```
{{
from gluon.serializers import json

response.headers[‘Content-Type’]=’text/json’

response.write(json(images.as_list()),escape=False)

}}
```

- visit http://.../list_images.json to get JSON output

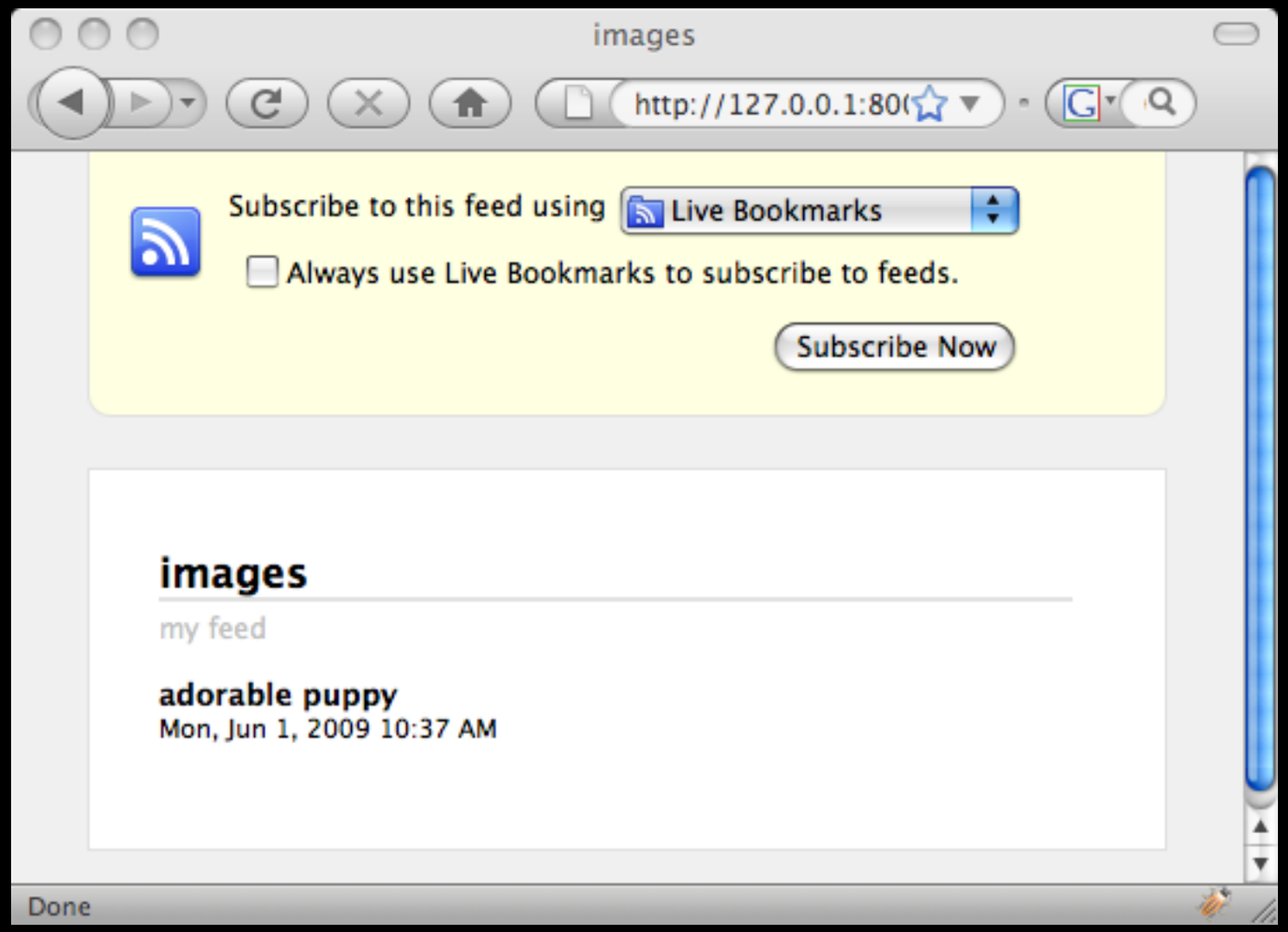
Advanced: list_images.rss

- create "images" view "default/list_images.rss" and edit

```
{{
from gluon.serializers import rss
feed={‘title’:’images’,
      ’link’:URL(r=request),
      ’description’:’my feed’,
      ‘items’:[{‘title’:item.title,
                ’link’:URL(r=request,f=’download’,args=item.file),
                ’description’:item.description} for item in images]}
response.headers[‘Content-Type’]=’application/rss+xml’
response.write(rss(feed),escape=False)
}}
```

- visit http://.../list_images.rss to get RSS output

Try "list_images.rss"



Add Services

- Edit controller "default" and append

```
@service.xmlrpc

@service.jsonrpc   # for Pyjamas for example

@service.amfrpc    # for Flash for example

def image_by_id(id)

   image = db.image[id]

   if image:
      return dict(title=image.title,description=image.description)
   else: return 0
```

- Try the XMLRPC service

```
>>> from xmlrpclib import ServerProxy

>>> s = ServerProxy('http://127.0.0.1:8000/images/default/call/xmlrpc')

>>> print s.image_by_id(1)
```

Services and Authentication

- Any action or service requiring authentication will redirect to login uness it finds basic authentication parameters in the header. For example
- `curl -u username:password http://hostname.../function`
- `curl http://username:password@hostname.../function`

Uploads and Authentication

- Uploaded files are served by the "download" action. To enforce authorization on uploaded files do:
  - `db.image.file.authorize = lambda record: True`
- where the lambda is a function, given the record, decides whether the user is or is not authorized to download the file stored in the record.

Template Language

What for?

- The template language allows to embed code into HTML
- Views should contain code to control presentation (layout, aesthetics, look and feel)
- In web2py the template is used in views that are execute to render the output of a controller.
- Use {{ ... }} to tag code into HTML

Examples

- {{...any python code you like in here...}}
- {{=1+3}} to insert in the HTML the output
- Use pass to close blocks if not obvious
  - {{for item in ‘iterable’:}}...{{=item}}... {{pass}}
  - {{if True:}}...true...{{else:}}...false... {{pass}}

Views can extend and include other files

- {{extend ‘layout.html’}}
- {{include ‘otherview.html’}}

Helpers

- A(...,_href=....) in the previous example is a helper
- Web has helpers for the most common HTML tags and a universal helper (TAG) for custom defined ones.
- Helpers provide a server-side representation of the Document Object Model (DOM)
- They can be used in models, views and controllers

Helpers

- Most helpers are derived from DIV and have the same syntax: DIV(*components,**attributes)
- components are escaped, serialized, concatenated and used for the innerHTML of the DIV. Attributes that start by '_' are used for tag attributes. Example:

```
>>> print DIV('click ',A('here',_href='link'),_class='1')
<div class="1">click <a href="link">here</a></div>
```

- web2py does not check for valid attributes

Helpers

- Strings that appear in the components of a helper are always escaped and strings that appear as attributed are always encoded.
- Use the special helper XML(...) to markup strings that shuould not be escaped.

```
>>> print DIV('<hello/>',XML('<world/>'))
<div>&lt;hello/&gt;<world/></div>
```

Helpers

- Helpers can be used to change the representation of a field for example:

```
db.image.posted_by.represent = lambda value: A(value,_href='...')
```

- The above line represents the field "create_by" as a link in tables and forms (when the field is readonly)

FORM, SQLFORM, etc.

Overview

- There are many ways to generate forms in web2py
  - FORM
  - SQLFORM (*)
  - form_factory
  - crud.create, crud.update, crud.read (*)
  - (*) create forms from models
- Based on helpers, CSS friendly, Customizable in HTML

FORM

- Example:

```
def index():
    form = FORM(‘your name:’,
                INPUT(_name=’yourname’,requires=IS_NOT_EMPTY()),
                INPUT(_type=’submit’))
    if form.accepts(request.vars,session):
        response.flash = ‘hello ‘+form.vars.yourname
    return dict(form=form)
```

SQLFORM

- Example:

```
db.define_table(‘dog’,SQLField(‘name’,requires=IS_NOT_EMPTY()))

def index():
    form = SQLFORM(db.dog)
    if form.accept(request.vars,session):
        response.flash = ‘record %i created‘ % form.vars.id
    return dict(form=form)
```

- Use SQLFORM(table, record) for update forms
- crud.create, crud.update, crud.read discussed later

form_factory

- Example:

```
def index():
    from gluon.sqlhtml import form_factory
    form = form_factory(SQLField(‘name’,requires=IS_NOT_EMPTY()))
    if form.accept(request.vars,session):
        response.flash = ‘dog %s created‘ % form.vars.name
    return dict(form=form)
```

Database Abstraction Layer

Supported backends

- sqlite
- mysql
- postgresql
- oracle
- mssql
- firebird
- db2
- Google App Engine (but no joins and no transactions)
- informix (experimental)

Basic objects

- SQLDB is a database connection
- (GQLDB is a connection to the Google App Engine)
- SQLTable is a database table
- SQLField is a field
- SQLField is a query
- SQLRows is a set of rows returned by a query

tables and field as variables

- db.tablename is db[‘tablename’]
- db.tablename.fieldname is db.tablename[‘fieldname’]
- db.tablename.fieldname is db[‘tablename’][‘fieldname’]
- db.tables is list of available tablenames
- db.tablename.fields is list of available fieldnames

SQLDB

- Connect to local SQLite database

```
db=SQLDB(‘sqlite://storage.sqlite’,pool_size=0)

db=SQLDB(‘mysql://username:password@hostname/database’,pool_size=0)

db=SQLDB(‘postgres://username:password@hostname/database’,pool_size=0)

db=SQLDB(‘oracle://username:password@hostname/database’,pool_size=0)

db=SQLDB(‘mssql://username:password@hostname/database’,pool_size=0)

db=SQLDB(‘firebird://username:password@hostname/database’,pool_size=0)

db=SQLDB(‘db2://username:password@hostname/database’,pool_size=0)

db=GQLDB() # for Google App Engine
```

- Set pool_size>0 for connection pooling (not for SQLite)

SQLTable & SQLField

- Create a table

```
mytable = db.define_table(‘dog’,
   SQLField(‘name’,’string’,length=128),
   SQLField(‘birthdate’,’date’))
```

- Allowed fields are:

```
string   ### length (in bytes) defaults to 64
text     ### unlimited length except on MySQL and Oracle
password
blob     ### blobs are stored base64 encoded
boolean  ### they are stores as T or F when not natively available
integer
double
time
date
datetime
upload   ### store a file name
```

‘upload’ fields

- To upload in filesystem

```
SQLField(‘fieldname’,’upload’)
```

- To upload in database

```
SQLField(‘fieldname’,’upload’,uploadfield = ’fieldname_blob’)
```

- In both cases when a file is uploaded it is safely renamed and both file contents and original filename are stored. in the latter case the file content is stored in a hidden blob field ‘fieldname_blob’ in the same table.
- The original file name is encoded in the new name and used to set the content-disposition header when file is downloaded. Uploaded files are always streamed.

INSERT

- The normal way

```
db.dog.insert(name='Snoopy', birthdate=datetime.date(2009,1,1))
```

```
(returns the id of the newly created dog record)
```

- The shortcut

```
db.dog[0] = dict(name='Snoopy', birthdate=datetime.date(2009,1,1))
```

UPDATE

- The normal way

```
db(db.dog.id==1).update(name='Skipper')
```

```
(can allow more complex queries)
```

- The shortcut

```
db.dog[1] = dict(name='Skipper')
```

```
(the shortcut can only find the record by id)
```

DELETE

- The normal way

```
db(db.dog.id==1).delete()
```

```
(can allow more complex queries)
```

- The shortcut

```
del db.dog[1]
```

```
(the shortcut can only find the record by id)
```

COUNT

```
db(db.dog.name.upper()<‘M’).count()
```

SELECT

- The normal way

```
rows = db(db.dog.id==1).select()

for row in rows:

     print row.id, row.name, row.birthdate


(not rows is True if no records returned)
```

- The shortcut

```
row = db.dog[1]


(row is None if no record with current id)
```

Queries

- Given

```
query1 =  db.dog.birthdate.year()>2007

query2 =  db.dog.birthdate.month()<8
```

- Queries can be combined with and(&), or(|) and not(~)

```
rows=db(query1 & query2).select()

rows=db(query1 | query2).select()

rows=db(query1 & (~query2)).select()
```

- Queries that involve multiple tables indicate an INNER JOIN.

SELECT ... ORDER BY ...

- Alphabetically

```
rows = db(db.dog.id>0).select(orderby = db.dog.name)
```

- Reversed Alphabetically

```
rows = db(db.dog.id>0).select(orderby = ~db.dog.name)
```

- Alphabetically and by year reversed

```
rows = db(db.dog.id>0).select(
        orderby = db.dog.name | ~db.dog.birthdate.year() )
```

SELECT ... DISTINCT ...

- List all dog names from database

```
rows = db(db.dog.id>0).select(db.dog.name, distinct=True)
```

SELECT ... some field ...

- Only name

```
rows = db(db.dog.id>0).select(db.dog.name)
```

- Name and id

```
rows = db(db.dog.id>0).select(db.dog.id, db.dog.name)
```

- All fields (default)

```
rows = db(db.dog.id>0).select(db.dog.ALL)
```

Aggregates

- db.table.field.count()
- db.table.field.max()
- db.table.field.min()
- db.table.field.sum() # for integer and double fields

Extract from date/datetime

- db.table.field.year()
- db.table.field.month()
- db.table.field.day()
- db.table.field.hour()
- db.table.field.minutes()
- db.table.field.seconds()

Operator like

- All dogs with name starting with ‘s’

```
rows = db(db.dog.name.like(‘s%’)).select()
```

- All dogs with name ending with ‘s’

```
rows = db(db.dog.name.like(‘%s’)).select()
```

- All dogs with name containing ‘s’

```
rows = db(db.dog.name.like(‘%s%’)).select()
```

- All dogs with name containing keyword

```
rows = db(db.dog.name.like(‘%%%s%%’ % keyword)).select()
```

Operator belongs and nested SELECTs

- All dogs with name in a list

```
rows = db(db.dog.name.belongs(('Snoopy','Skipper'))).select()
```

- All dogs with name ID from a nested select

```
rows = db(db.dog.id.belongs( db()._select(db.dog.id) ).select()
```

- The argument of the id can be any nested select (involving other tables as well, as long as only one specific field is requested). Notice _select, not select for the nested one.

Expressions

- Consider

```
db.define_table(‘product’,
    SQLField(‘price’,’double’),
    SQLField(‘discounted_price’,‘double’)
```

- the values in queries and updates can be expressions:

```
rows = db(db.product.price==db.product.discounted_price+10).select()

db(db.product.id>10).update(price=db.product.price+20)
```

INNER JOINS

- Define a reference between a dog.owner and a person

```
db.define_table(‘person’,SQLField(‘name’)

db.define_table(‘dog’,SQLField(‘name’),SQLField(‘owner’,db.person)
```

- Select all dogs and their owners

```
rows = db(db.dog.owner==db.person.id).select()

for row in rows:

    print row.dog.name, ’belongs to’, row.person.name
```

LEFT OUTER JOINS

- Define a reference between a dog.owner and a person

```
db.define_table(‘person’,SQLField(‘name’)

db.define_table(‘dog’,SQLField(‘name’),SQLField(‘owner’,db.person)
```

- Select all dogs and their owners

```
rows = db().select(db.person.ALL, db.dog.ALL,

                   left=db.person.on(db.dog.owner==db.person.id))

for row in rows:

    print row.dog.name, ’belongs to’, row.person.name or ‘nobody’
```

INNER JOINS

many2many

- Define a reference between a dogs and people

```
db.define_table(‘person’,SQLField(‘name’)

db.define_table(‘dog’,SQLField(‘name’)

db.define_table(‘friendship’,SQLField(‘dog_id’,db.dog),

                                SQLField(‘person_id’,db.person)
```

- Select all dogs and their owners

```
friends=(db.dog.id==db.friendship.dog_id) &

        (db.person.id==db.friendship.person_id)

rows = db(friends).select(db.dog.ALL, db.person.ALL)

for row in rows:

    print row.dog.name, ’is friend with’, row.person.name
```

OUTER JOINS selfreference

- Consider a self-referential table

```
db.define_table(‘dog’,
    SQLField(‘name’),
    SQLField(‘father_id’,’reference dog’),
    SQLField(‘mother_id’,’reference dog’))
```

- Select all dogs and their parents

```
father=db.dog.with_alias(‘father’)
mother=db.dog.with_alias(‘mother’)
rows = db().select(db.dog.name, db.father.name, db.mother.name,
                   left=(db.father.on(db.father.id==db.dog.father_id),
                         db.mother.on(db.mother.id==db.dog.mother_id)))
for row in rows:
    print row.dog.name, row.father.name, row.mother.name
```

Transactions

- In web2py all actions are executed in a transaction that is committed on success and rolled back on error (and ticket is issued)
- Transactions can also be closed manually

```
db.commit()

db.rollback()
```

- When using the shell or batch web2py scripts, transactions much be committed or rolled back explicitly.

Notes on GAE

- GAE does not support
- JOINS (user IS_IN_DB(....,multiple=True) instead
- TRANSACTIONS
- aggregates, extract end like operators
- expressions
- OR (|)
- everything else works in GAE

Create/Read/Update/Delete

crud

- To use CRUD you must instantiate it for each db

```
from gluon.tools import Crud
crud = Crud(globals(),db)
```

- The crud object provides methods for forms

```
form = crud.create(db.tablename)

form = crud.read(db.tablename, record_id)

form = crud.update(db.tablename, record_id)

crud.delete(db.tablename, record_id) # not a form
```

crud.create

- Example of usage in controller

```
def index():
    return dict(form=crud.create(db.tablename))
```

- Example of usage in view

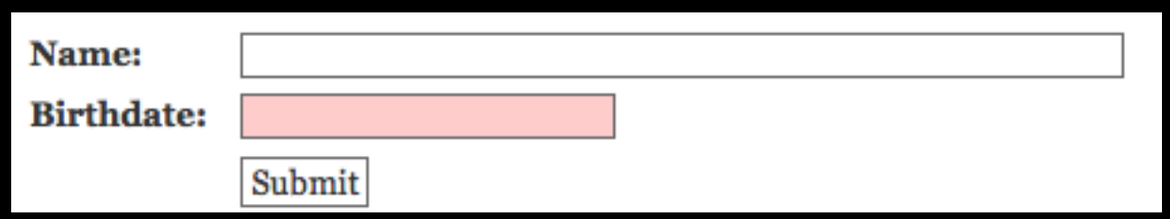
```
{{=crud.create(db.tablename)}}
```

- All form processing is done inside the method

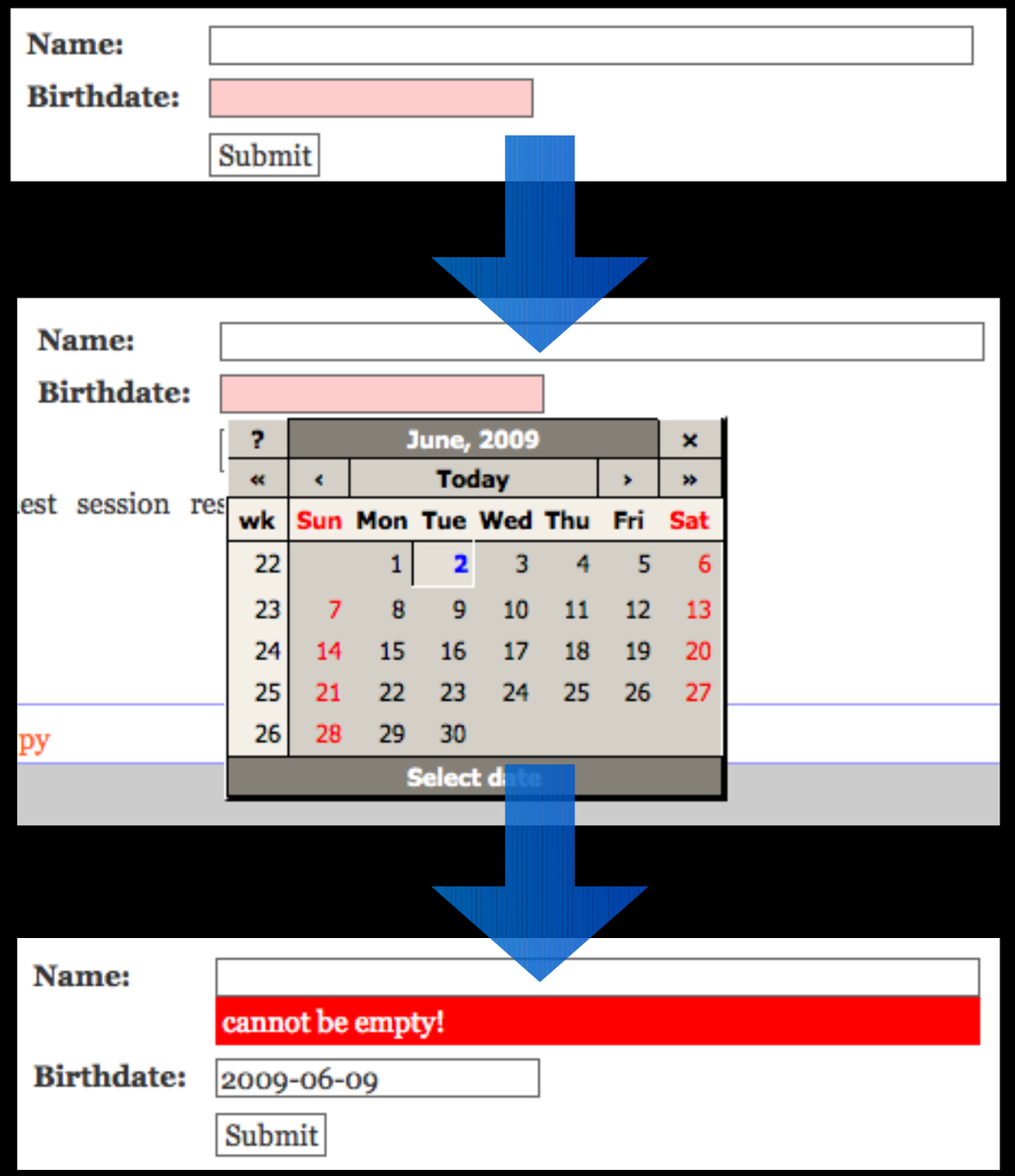
crud.create

- Example:

```
db.define_table('person',SQLField('name',requires=IS_NOT_EMPTY()),
                         SQLField('birthdate','date'))
def create_person():
    return dict(form=crud.create(db.person))
```



crud.create


Name:
Birthdate:
Submit
Name:
Birthdate:
June, 2009
Today
wk Sun Mon Tue Wed Thu Fri Sat
Select date
Name:
cannot be empty!
Birthdate:
2009-06-09
Submit

crud.update

- Example of usage in controller

```
def update_person():
    record_id = request.args(0)
    return dict(form=crud.update(db.tablename,record_id))
```

- The second argument can be a record or a record id
- update forms that contain ‘upload’ fields with an image will show an image preview

crud.read

- crud.read works like crud.update but it is read only

```
def read_person():
    record_id = request.args(0)
    return dict(form=crud.read(db.tablename,record_id))
```

crud.delete

- crud.delete does not generate forms, it just deletes the record and redirects

```
def delete_person():
    record_id = request.args(0)
    crud.delete(db.tablename,record_id,
                next=URL(r=request,f='create_person'))
```

Optional Arguments

- crud.create, crud.update and crud.delete take the following self-explicative arguments

```
crud.update(table,
            record,
            onvalidation = lambda form: None,
            onaccept = lambda form: None,
            message = ‘record updated’,
            next = ‘url to go in case of success’)
```

- onvalidation is called after validation, before db-io
- onaccept is called after validaiton, after db-io
- message is the message to be flashed after redirection
- next is the URL to redirect to after success.

crud()

- Occasionally one may want a do-it-all crud
  - `def data(): return crud()`
- It can be accessed as
  - `http://.../app/default/data/tables`
  - `http://.../app/default/data/create/[tablename]`
  - `http://.../app/default/data/update/[tablename]/[record_id]`
  - `http://.../app/default/data/delete/[tablename]/[record_id]`
  - `http://.../app/default/data/select/[tablename]`

Customizing crud

- readonly, hidden and widgets

```
db.define_table('person',SQLField('name',requires=IS_NOT_EMPTY()),
                         SQLField('birthdate','date'))

db.person.name.writable = False # field is readonly
db.person.name.readable = False # field is hidden
from gluon.sqlhtml import StringWidget
db.person.name.widget = StringWidget.widget # use a widget

def update_person():
    record_id = request.args(0)
    return dict(form=crud.update(db.person,record_id))
```

Available Widgets

- defined in gluon/sqlhtml.py and used as default cases

```
class StringWidget
class IntegerWidget(StringWidget)
class DoubleWidget(StringWidget)
class TimeWidget(StringWidget)
class DateWidget(StringWidget)
class DatetimeWidget(StringWidget)
class TextWidget
class BooleanWidget
class OptionsWidget
class MultipleOptionsWidget
class PasswordWidget
class UploadWidget
```

- they can be extended

More Customization

- Crud behavior can also be customized by setting
- crud.settings.xxx
- crud.messages.xxx
- for a list of xxx consult file gluon.tools.py

Crud and Authorization

- All cruds enforce "Auth" permissions without need for decorators. Details later.

Validators

Validators

- Fields can have validators and they are used in forms
- Some fields (date, time, datetime) have default validators
- All validators are objects and used as in

```
db.tablename.fieldname.requires = IS_NOT_EMPTY()
db.tablename.fieldname.requires = [IS_NOT_EMPTY(), IS_LENGTH(30)]
```

```
db.dog.name.requires = IS_MATCH('\w+',error_message='...')
db.dog.name.requires = IS_EXPR('int(value)==4',error_message='...')
db.dog.name.requires = IS_LENGTH(32,error_message='...')
db.dog.name.requires = IS_IN_SET(['1','2','3'],error_message='...')
db.dog.name.requires = IS_INT_IN_RANGE(-10,10,error_message='...')
db.dog.name.requires = IS_FLOAT_IN_RANGE(-10,10,error_message='...')
db.dog.name.requires = IS_NOT_EMPTY(error_message='...')
db.dog.name.requires = IS_ALPHANUMERIC(error_message='...')
db.dog.name.requires = IS_EMAIL(error_message='...')
db.dog.name.requires = IS_GENERIC_URL(error_message='...')
db.dog.name.requires = IS_HTTP_URL(error_message='...')
db.dog.name.requires = IS_URL(error_message='...')
db.dog.name.requires = IS_TIME("%H:%M:%S",error_message='...')
db.dog.name.requires = IS_DATE("%Y-%m-%d",error_message='...')
db.dog.name.requires = IS_DATETIME("%Y-%m-%d %H:%M:%S",...)
db.dog.name.requires = IS_LIST_OF(IS_INT_IN_RANGE(0,10))
db.dog.name.requires = IS_NULL_OR(IS_INT_IN_RANGE(0,10))
db.dog.name.requires = IS_IN_SUBSET(['1','2','3'],...)
db.dog.name.requires = IS_LOWER() # filter to lower case
db.dog.name.requires = IS_UPPER() # filter to upper case
db.dog.name.requires = CRYPT()    # filder to hash value
db.dog.name.requires = CLEANUP()  # remove non \w+ chars
```

Role Based Access Control

Authentication

- Set it up in a model file (requires db object)

```
from gluon.tools import *
mail=Mail()
mail.settings.server='smtp.example.com:25'
mail.settings.sender='you@example.com'
mail.settings.login='you@example.com:password' or None # if no TLS
auth=Auth(globals(),db)
auth.mailer=mail
auth.settings.registration_requires_verification = False
auth.define_tables()
auth.captcha=Recaptcha('public_key', 'private_key')
auth.messages.verify_email = """Click on the link
    http://127.0.0.1:8000/app/default/user/verify_email/%(key)s
to verify your email, thanks for registering."""  #### IMPORTANT!
```

Authentication

- auth.define_tables() defines:
- auth_user
- auth_group
- auth_membership (users are members of groups)
- auth_permission (groups have permissions)
- auth_event (where auth events are logged)

Custom Auth tables

- All tables can be replaced by a custom defined one

```
auth=Auth(globals(),db)

# the fields below are requires you can add more

auth.settings.table_user = db.define_table(‘auth_user’,
   db.Field(’first_name’, length=128,default=’’),     db.Field(’last_name’,
   length=128,default=’’),     db.Field(’email’, length=128,default=’’),
    db.Field(‘password’, ‘password’,readable=False, label=’Password’),
   db.Field(’registration_key’, length=128,             writable=False,
   readable=False, default=’’))t =
   self.settings.table_usert.first_name.requires =
   IS_NOT_EMPTY()t.last_name.requires = IS_NOT_EMPTY()t.password.requires =
   CRYPT() # password will be stored hashedt.email.requires = [IS_EMAIL(),
   IS_NOT_IN_DB(db, ‘auth_user.email’)]

auth.define_tables() ### auth_user will not be redefined!
```

Exposing Auth

- Normally a single controller function

```
def user(): return auth()
```

- exposes all actions

```
http://..../app/default/user/register
http://..../app/default/user/login
http://..../app/default/user/logout
http://..../app/default/user/retrieve_username
http://..../app/default/user/retrieve_password
http://..../app/default/user/verify_email
http://..../app/default/user/profile
http://..../app/default/user/change_password
```

- and you only need one view: default/user.html (provided)

Exposing Auth Components

- You can also expose auth components separately

```
def login(): return dict(form=auth.login())

def register(): return dict(form=auth.register())

...

def logout(): return auth.logout(next=URL(r=request,f='index'))
```

- And you can have one view each

```
default/login.html

default/register.html

...

(logout does not need a view)
```

Exposing Auth Components

- Individual Auth components take all the same optional extra parameters

```
def login():

  return dict(form=auth.login(

      next = url_for_redirection_after_login,        onvalidation =
lambda form: what_to_do_after_validation(form),        onaccept = lambda
form: what_to_do_after_accepting_login(form),        log="user %
(first_name)s %(last_name)s logged in",    ))
```

More Customization

- Auth behavior can also be customized by setting
- auth.settings.xxx
- auth.messages.xxx
- for a list of xxx consult file gluon.tools.py

Login Plugins

- To authenticate using gmail look into:
  gluon/contrib/login_methods/email_auth.py
- To authenticate using LDAP look into:
  gluon/contrib/login_methods/ldap_auth.py
- To authenticate using a third party BASIC authentication provider look into:
  gluon/contrib/login_methods/basic_auth.py
- In all cases a record will be added to the local auth_user table if user not already in local system

Giving Permissions

```
# create a group
group_id = auth.add_group(role="Manager",
                          description="can update table images")
# give the logged in user membership of this group
auth.add_membership(auth.user.id, group_id)
# give this group permission to "update" all records (0) of table image
auth.add_permissions(group_id,’update’,db.image,0)
```

- You can ask web2py to enforce authorization

```
crud.authorization = auth
```

- Or be explicit for each function

```
@auth.requires_permission(‘update’,db.image)

def do_something(): return dict()
```

Checking Permissions

- You can force authorization on ANY function

```
@auth.requires_login()

def f(): return dict()


@auth.requires_membership(’Manager’)

def g(): return dict()


@auth.requires_permission(‘update’,db.image,0)

def h(): return dict()
```

About Permissions

- In the following code

```
auth.add_permissions(group_id,access_type,resource,record_id)
```

- access_type and resource can be arbitrary strings but web2py if checked explicitly (has_permission, requires_permission) but CRUD only understands access_type in [‘read’,’create’,’update’,’delete’] if resource if a table name and record_id is a specific record or 0 (all records of the table)

Authentication for Services

- If a function is exposed as a service

```
@service.xmlrpc

@auth.requires_login()

def f(): return dict()
```

- or simply called remotely from a script it accepts basic authentication

```
curl -u username:password http://..../f
```

Authentication for Download

- It may be required to enforce authorization on the download function. This is done with

```
db.image.file.authorization=lambda row: \
                                auth.has_permission(‘read’,db.image,row.id)
```

- This has to be done for every ‘upload’ field

Caching

in Ram, on Disk, with Memcache

Caching in RAM

- You can cache any function

```
@cache(‘key’,cache.ram,5000)
def f(): return dict()
```

- You can cache any value or expression

```
a = cache.ram(‘key’,lambda: ‘value’, 5000)
```

- You can cache any action and its views

```
@cache(request.env.path_info,5000, cache.ram)
def action():
    return response.render(response.view,dict())
```

- You can choose any "key" as long as unique. 5000 is the cache expiration time.

Clear and Reset Cache

- To delete a cache entry
  - `cache.ram('key',None)`
- To force a cache reset choose a zero time
  - `cache.ram('key',lambda: 'value', 0)`
- You can use the cache to increment a counter
  - `cache.ram('counter',lambda: 0, 0) # create "counter"`
  - `value = cache.ram.increment('counter',+1) # increment it of +1`

Caching Database select

- Select results can be cached (but not on GAE)

```
rows = db(...).select(....,cache=(cache.ram,5000))
```

cache.ram .disk .memcache

- Cache in ram
  - `cache.ram`
- Cache on disk
  - `cache.disk`
- Cache with memcache
  - `from gluon.contrib import Memcache`
  - `cache.memcache=Memcache([...list of memcache servers...])`
- On GAE cache.ram is mapped into memcache automatically

AJAX

web2py comes with jQuery base

CSS Friendly Forms

- All forms generated by web2py have CSS friendly ids and classes.
- Look at the generated HTML
- You can reference them using jQuery and change their attributes:

```
<script>
jQuery(document).ready(function(){
  jQuery('#tablename_fieldname').attr('size',30);
});
</script>
```

jQuery?

- jQuery base and calendar.js are already included
- together with some other custom functions in web2py_ajax.html
- jquery provides functions for AJAX and Effects
- for more info: http://www.jquery.com

Effects Examples

- Hide a div

```
<div id="one">This is hidden<div>
<script>jQuery(‘#one’).hide();</script>
```

- Make the div appear on click

```
<button onclick="jQuery(‘#one’).slideDown();">click</button>
```

- Effects: hide(), show(), slideDown(), slideToggle(), slideUp(), fadeIn(), fadeOut()
- Attribute set: jQuery(...).attr(‘width’,’30%’)
- Attribute get: var x = jQuery(...).attr(‘width’)

Recommended Plugins

- jQuery datatable
  - http://plugins.jquery.com/project/DataTables
- jQuery multiselect
  - http://abeautifulsite.net/notebook/62
- jQuery autocomplete
  - http://bassistance.de/jquery-plugins/jquery-plugin-autocomplete/
- jPolite port to web2py (experimental)
  - http://www.web2py.com/jPolite

Scalability

App ByteCode Compilation

- web2py apps can be bytecode compiled
- you can distribute bytecode compiled apps
- bytecode compiled apps are faster because there is no template parsing at runtime

Single Server Tricks

- Use mod_wsgi instead of the provided wsgiserver
- Let the web server (apache) deal with files in static/ folder
- Add if statements in models so that only those parts you need (based on request.controller and request.function) are executed
- Do not store sessions and images in database
- Cache as much as you can in run

Multiple Server Tricks

- Cross mount the upload folder
- Leave sessions and tickets on local file system
- Use the Pound load balancer so that sessions are not broken (http://www.apsis.ch/pound/)
- Use web2py/scripts/tickets2db.py to collect tickets and store them in DB
- Use web2py/scripts/sessions2trash.py to periodically cleanup session files

License

License Caveats

- web2py code is GPL2 with exceptions
- You can redistribute unmodified web2py binaries with your app
- web2py executes your code thus its license does not extend to your app. You can distribute your app under any license you wish (including closed source), as long as you do not include web2py code (from gluon/*.py folder) else you are bound by the two points above.
- JS modules in web2py have their own licenses, usually BSD or MIT type licenses.

Further references

References

- Users Group: http://group.google.com/groups/web2py
- Book: http://www.amazon.com/Web2Py-Manual-Massimo-
- Book (pdf): http://www.lulu.com/content/4968879
- Free Apps: http://www.web2py.com/appliances