# Rolling with web2py

(formerly known as Gluon) created by Massimo Di Pierro

Perhaps you have heard of web2py, the new kid on the block of Web Frameworks. web2py is written in Python so it is more solid and much faster than Ruby on Rails. web2py is also a web application itself so you can do all development, deployment and maintenance of your applications through your web browser and that makes it easier to use than any other framework. Moreover web2py ships in one complete package (for Windows, Mac or Unix/Linux) including everything you need to start development (including Python, SQLite3, and multi-threaded web server).

You can get web2py here: http://www.web2py.com
This document is intentionally designed to mimic
http://onlamp.com/pub/a/onlamp/2005/01/20/rails.html
so that you can compare web2py with Rails.

## *What is Python?*

Python is an object oriented programming language designed to be super easy to teach without any compromise on functionality. Most Java algorithms can be rewritten in Python in one tenth of their original length. Python comes with an extensive set of portable standard libraries including support for many standard internet protocols (http, xml, smtp, pop, and imap, just to mention a few) and APIs to the Operating System.

## *What is web2py?*

web2py is an open source web framework written in Python and programmable in Python for fast development of database-driven web applications. There are many web frameworks today including Ruby on Rails, Django, Pylons and Turbo Gears, so why another one?

I developed web2py with the following goals in mind:

1) As similar as possible to Rails but in Python, so that it is more solid and much faster.

2) All-in-one package with no installation, no configuration and no shell scripting required.

3) Be super easy to teach (my job is to teach). So I made web2py itself as a web application.

4) Top-down design so that the web2py APIs would be stable from day one.

## *Seeing is Believing*

Programming web2py is as easy as programming Rails but, if you do not know Python nor Ruby, web2py is easier to learn than Rails.

What is most important is that web2py requires much less code than J2EE equivalent or PHP equivalent, while enforcing a vary good and safe programming style.

web2py prevents directory traversal, SQL injections, cross site scripting, and reply attack vulnerabilities.

web2py manages session, cookies and application errors for you. All application errors result in a ticket issued to the user and a log entry for the administrator.

web2py writes all the SQL for you. It even creates the tables and decides when to do a migration of the database.

Give it a try.

## *Installing the Software*

Go to http://mdp.cti.depaul.edu/examples and download the Windows, Mac or Unix files.

If you choose to use the Windows or Mac version you do not need anything else: unzip the file and click on `web2py.exe` or `web2py.app` respectively.

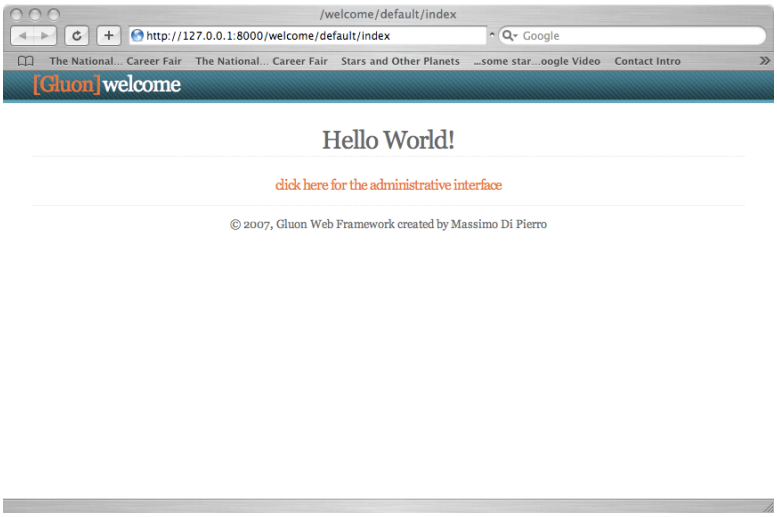
If you choose to use the Unix version you need the Python interpreter (version 2.4 or later) and the SQLite3 database. After you have those, unzip web2py and run

```
python web2py.py
```

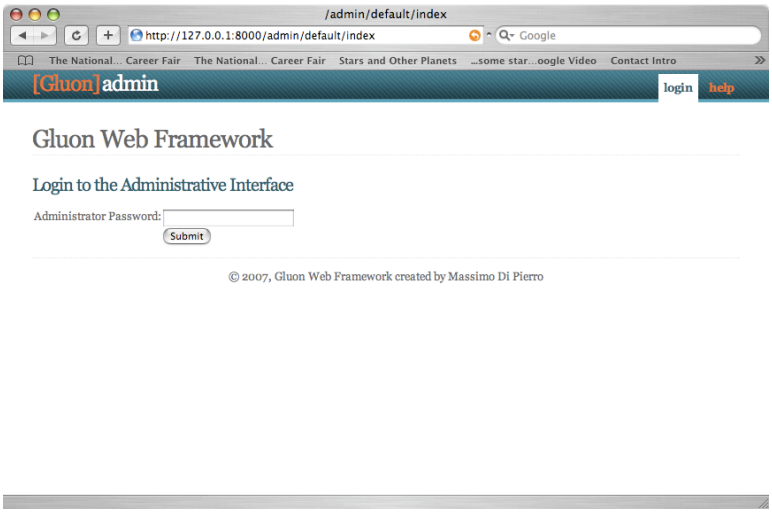
In a production setting you should use PostgreSQL or MySQL and not SQLite3. From the web2py prospective that is as easy as changing one line in the program but that is not discussed here since you do not need it for development.

## *Running web2py*

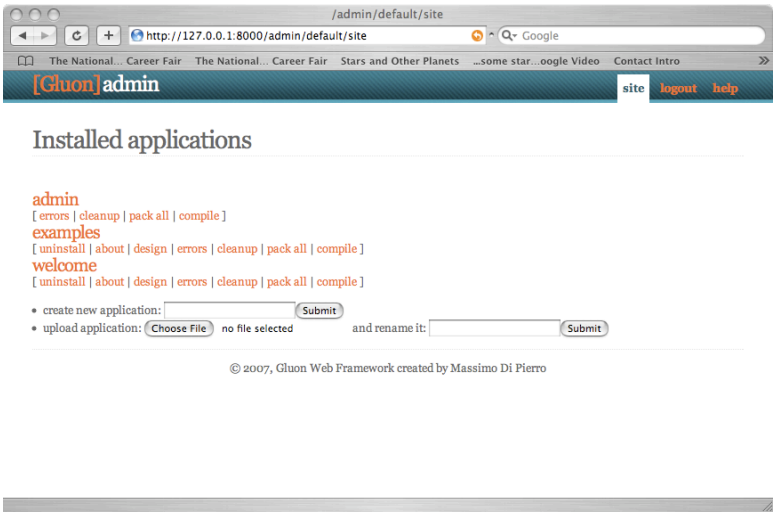
At startup web2py asks one question: “choose the administrator password”. Choose one. After that web2py will open a web browser for you (remember no commands to type ever!) showing this welcome page




Click on “administrative interface”




and type the password that you choose at startup. You will be redirected to the “site” page of the administrative interface:




Here you can:

- install and uninstall applications
- create and design (edit) your applications
- cleanup error logs and sessions
- byte-code compile applications for distribution and faster execution

web2py comes with three applications: **admin** (the administrative interface itself), **examples** (interactive documentation), and **welcome** (a basic template for any other application).

## *Let's Write Code*

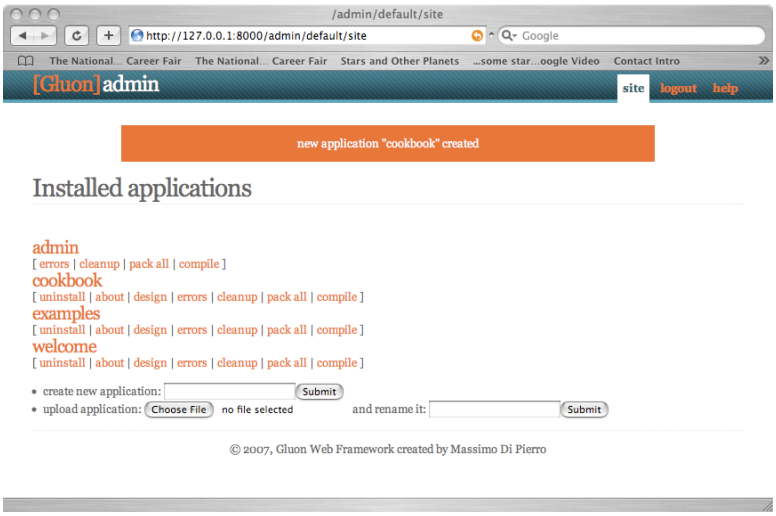
We'll create an online collaborative **cookbook** for holding and sharing recipes. We want our cookbook to:

- Display a list of all **recipe**s.
- Create new recipes and edit existing recipes.
- Assign a recipe to a **category** (like "dessert" or "soup").

If you like, you can download the complete web2py Cookbook example and follow along.

## Creating an Empty web2py Application

To start a new application type a name in the appropriate field, in our case **cookbook**, and press the button *submit*:




A new web2py application is not empty but it is a clone of the **welcome** applicaiton. It contains a single controller, a single view, a base layout, a generic view and its own database administrative interface called **appadmin** (not to be confused with **admin**, the site-wide administrative interface).

## Testing the Empty Web Application

You are already running web2py web server so there is nothing to test really. Anyway, click on **cookbook/design** and you will see

Here is where you can view/create/edit the components of your application. Under **Controllers** there is a file called `default.py` which "exposes `index`". If you click on `index` your newly created application will "welcome you".

## *web2py Model View Controller Design*

Any web2py application is comprised of:

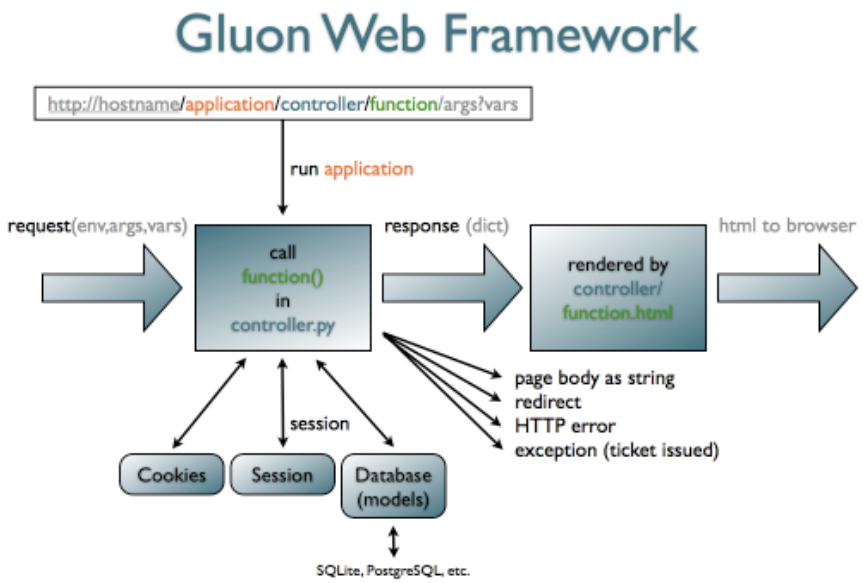
- Models: files that contain a description of the data stored by your application. For example the fields in the tables of your databases, their relations, and requirements. web2py tells you which tables are defined in each model file.
- Controllers: files that contain the logic of your application. Each URL is uniquely mapped into a function in a controller file. That function can generate a page, delegate a view to render a page, redirect to another URL or raise an exception (depending on the exception that may result in a ticket being issued or in a HTTP error page). web2py tells you which functions are exposed by each controller file.
- Views: files that contain HTML and special {{ }} tags which render in HTML variables returned by the controller. This is the presentation layer of your application. web2py tells you when a view extends or imports other views.
- Languages: files that contain translation tables for all strings (those that you explicitly mark as language dependent) for any of the languages you want to support.
- Static files: all other files, including images, CSS, JavaScript, etc.

Notice that you do not need an editor nor you need to know the web2py directory structure since you can create and edit files from the **design** page.

Also notice that while it is good policy to give a view to every controller function (called action in rails), you do not have to since web2py always provides a `generic.html` view that will render any page that is missing a template.

## *URLs and Controllers*

This image represents the general structure of web2py's core functionality




A URL like

http://hostname/cookbook/default/index/bla/bla/bla?variable=value

will result in a call to function `index()` in controller `default.py` in applicaiton cookbook.

"bla", "bla" and "bla" will be passed as `request.args[0:3]` while "value" will be stored in `request.vars.variable`.

Controller functions should return a dictionary like in

```
return dict(name=value, othername=othervalue)
```

and the variables name and othername will be passed to the associated view.

Try now, from **cookbook/design**, to create a `test.py` controller (just type the name and click submit), edit `test.py` and create your own index function

```
1 # try something like
2 def index(): return dict(text="Hello form cookbook application")
```

go back to **cookbook/design** and click on the `index` function exposed by `test.py`.


web2py is using the `generic.html` view, which extends the basic `layout.html`, to render the variable text returned by your `index()` function.

## *The excitement Begins...*

## *Creating the Model*

Go to **cookbook/design** and create a new model called `db.py` (just type `db` in the apposite field and click submit). The definition of a model here is slightly different than in Rails. In web2py a model is a single file that contains a definition of all tables in each database.


Edit the just created `db.py` model and write the following:

```
import datetime; now=datetime.date.today()
db=SQLDB('sqlite://db.db')

db.define_table('category',SQLField('name'))

db.define_table('recipe',
                SQLField('title'),
                SQLField('description',length=256),
                SQLField('category',db.category),
                SQLField('date','date',default=now),
                SQLField('instructions','text'))

db.category.name.requires=[IS_NOT_EMPTY(),IS_NOT_IN_DB(db,'category.name')]
db.recipe.title.requires=[IS_NOT_EMPTY()]
db.recipe.description.requires=IS_NOT_EMPTY()
db.recipe.category.requires=IS_IN_DB(db,'category.id','category.name')
db.recipe.date.requires=IS_DATE()
```

This model defined two tables **category** and **recipe**. **recipe** has a field **category** that is a reference to **db.category** and field **date** that default to today. Each field has some requirements (this is optional), **category.name** requires that a new value IS_NOT_IN_DB (the field must be unique), **recipe.category** requires that the field IS_IN_DB (the reference is valid), **recipe.date** requires that it contains a valid date.

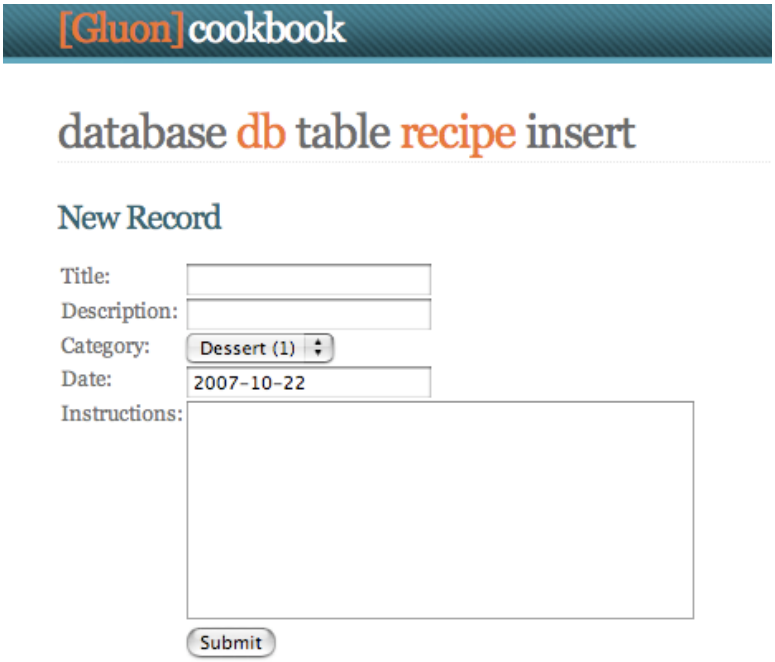
These requirements will be enforced in any entry form, whether part of the administrative interface or user generated.

### *The Database Administrative Interface (appadmin)*

Go to **cookbook/design** and, under model, you will see two new links database **administration** and **sql.log**. Click on the former and if you do not have typos you will see:


This is your application administrative interface. Try to insert a new category record:


and some new recipes:




Wasn't this easier than Rails? Let's not even compare with PHP, JSP, ASP, J2EE, etc.

Who created the tables? web2py did! web2py looked for a database called `db.db`, could not find one so it created the database and the tables you just defined. If you modify a table definition, web2py will alter the table for you (SQLite3 only supports adding fields, Postgresql also supports dropping fields). If you define another table it will be created. You can look at the SQL generated by web2py for this migration by clicking on **sql.log**.

Feel free to explore the administrative interface, insert a few records and try to list them.


The table is sortable by clicking on the header and will paginate if you have more than 100 items. Try a JOIN by typing "recipe.category=category.id" in the SQL FILTER field.


Where did field **id** come form? In web2py every table has a unique integer key called **id**. If you click on the id value in the table you will be able to edit the individual record.

Notice that `appadmin.py` is part of your cookbook application so you can read it and modify it. In this tutorial we choose not to do it and we prefer to take the longer route and write a new controller from scratch. We believe this better serves our didactic purpose.

### *Creating Functions (Actions)*

While in **cookbook/design,** edit the `test.py` controller and add the following:

```
def recipes():
    records=db().select(db.recipe.ALL,orderby=db.recipe.title)
    return dict(records=SQLTABLE(records))
```

Now back in design, click on "recipes" and you should see


Notice that the variable `records` passed to the view is a **SQLTABLE** that knows how to render itself in CSS friendly HTML. The variable `records` is rendered by the `generic.html` view.

Let's customize this more. Change the controller into:

```
1.  def recipes():
2.      records=db(db.recipe.category==request.vars.category)\
3.              .select(orderby=db.recipe.title)
4.      form=SQLFORM(db.recipe,fields=['category'])
5.      return dict(form=form,records=records)
6.
7.  def show():
8.      id=request.vars.id
9.      recipes=db(db.recipe.id==id).select()
10.     if not len(recipes): redirect(URL(r=request,f='recipes'))
11.     return dict(recipe=recipes[0])
12.
13. def new_recipe():
14.     form=SQLFORM(db.recipe,fields=['title','description',\
15.                                    'category','instructions'])
16.     if form.accepts(request.vars,session):
17.         redirect(URL(r=request,f='recipes'))
18.     return dict(form=form)
```

Notice how:

- **recipes** now returns a list of records, not an SQLTABLE, moreover it generates a selection **form** from the **category** field of the table.
- **show** takes the **request.vars.id** and performs select, on failure it redirects to **recipes**
- **new_recipe** returns a SQLFORM object which builds an HTML form from the definition of a table (**db.recipe**). **form.accepts()** performs validation of the form (according to the requirements in the model), updates the form with error messages and, on successful validation, it inserts the new record in the database.

- `URL(r=request,f='function')` generates the url for "function" in the current application and controller as determined by the HTTP request.

This code is already fully working using the generic view but we will perform additional customization at the layout layer below.

Notice that some validators, like IS_DATETIME() for a 'datetime' field, are automatically set by default.

## *Creating Views*

Now create a view for **recipes**. This view is called `test/recipes.html` (type the name with path in the opposite field and click submit).


Edit the newly created file

```
1.  {{extend 'layout.html'}}
2.  <h1>List all recipes</h1>
3.  {{=form}}
4.  <table>
5.  {{for recipe in records:}}
6.  <tr>
7.  <td>{{=A(recipe.title,_href=URL(r=request,f='show?id=%s'%recipe.id))}}</td>
8.  <td>{{=recipe.date}}</td>
9.  </tr>
10. {{pass}}
11. </table>
12.
13. {{=A('create new recipe',_href=URL(r=request,f='new_recipe'))}}
```

Now try the calling **recipes** again


Notice that the code inside {{ }} tags is Python code with some caveats:

- There is no indentation requirement, a block of code starts with a line ending in colon and ends with a line starting with pass (exemptions are def:return, if:elif:else:pass and try:except:pass).
- The view sees everything defined in the model plus the variables returned by the controller.
- `{{=something}}` will render something in HTML after escaping special characters.

Notice that

```
{{=A(message,_href=link)}}
```

is an HTML helper. It simply writes the

```
<a href= "link">message</a>
```

tag for you.

Create a view `test/show.html` that contains:

```
1. {{extend 'layout.html'}}
2. <h1>{{=recipe.title}}</h1>
3. <h2>{{=recipe.description}}</h2>
4. Posted on {{=recipe.date}}<br/><br/>
5. <p>{{=recipe.instructions}}</p>
```

It will look like this:


Finally create a `test/new_recipe.html` that contains:

```
1. {{extend 'layout.html'}}
2. <h1>New recipe</h1>
3. {{=form}}
```

It will look like this:


Notice how web2py capitalized the names of the fields in the form and generated a SELECT/OPTION for the category field based on the specified requirements.

If you do no like the **[web2py]cookbook** banner or the CSS you can edit them both in the `layout.html` file.

## *Some Magic*

If you try to submit a form that does not meet the requirements (for example try to submit an empty recipe), web2py will notify you about that.


## *Conclusions*

We have written a working web2py application with only the browser, a few clicks and a total of 53 lines of code. We also got for free a database administrative interface that allows to insert, select, update and delete individual records or record sets.

web2py also includes easy to use functions to import/export tables in CSV, to generate RSS feeds and RTF files (compatible with MS Word), and to handle JSON for AJAX.

To read more about web2py visit the web page:

http://mdp.cti.depaul.edu

If you have questions, please join our Google group:

http://groups.google.com/group/web2py?hl=en

## *Appendix. The Database API*

Connect to a sqlite3 database in file test.db

```
>>> db=SQLDB("sqlite://test.db")
```

or connect to a MySQL database

```
>>> db=SQLDB("mysql://username:password@host:port/dbname")
```

or connect to a PostgreSQL database

```
>>> db=SQLDB("postgres://username:password@host:port/dbname")
```

Available field types

```
>>> tmp=db.define_table('users',\
       SQLField('stringf','string',length=32,required=True),\
       SQLField('booleanf','boolean',default=False),\
       SQLField('passwordf','password'),\
       SQLField('textf','text'),\
       SQLField('blobf','blob'),\
       SQLField('uploadf','upload'),\
       SQLField('integerf','integer'),\
       SQLField('doublef','double'),\
       SQLField('datef','date',default=datetime.date.today()),\
       SQLField('timef','time'),\
       SQLField('datetimef','datetime'),\
       migrate='test_user.table')
```

A field is an object of type SQLField

```
    >>> SQLField('fieldname','fieldtype',length=32,\
                 default=None,required=False,requires=[])
```

Drop the table

```
    >>> db.users.drop()
```

Examples of insert, select, update, delete

```
    >>> tmp=db.define_table('person',\
              SQLField('name'), \
              SQLField('birth','date'),\
              migrate='test_person.table')
    >>> person_id=db.person.insert(name="Marco",birth='2005-06-22')
    >>> person_id=db.person.insert(name="Massimo",birth='1971-12-21')
    >>> rows=db().select(db.person.ALL)
    >>> for row in rows: print row.name
    Marco
    Massimo
    >>> me=db(db.person.id==person_id).select()[0]
    >>> me.name
    'Massimo'
    >>> db(db.person.name=='Massimo').update(name='massimo')
    >>> db(db.person.name=='Marco').delete() # test delete
```

### Update a single record

```
>>> me.update_record(name="Max")
>>> me.name
'Max'
```

### Complex search conditions

```
>>> rows=db((db.person.name=='Max')&\
            (db.person.birth<'2003-01-01')).select()
>>> rows=db((db.person.name=='Max')| \
            (db.person.birth<'2003-01-01')).select()
>>> me=db(db.person.id==person_id).select(db.person.name)[0]
>>> me.name
'Max'
>>> rows=db(db.person.birth.month()==12).select()
>>> rows=db(db.person.birth.year()>1900).select()
>>> rows=db(db.person.birth==None).select()
>>> rows=db(db.person.birth!=None).select()
>>> rows=db(db.person.name.upper()=='MAX').select()
>>> rows=db(db.person.name.like('%ax')).select()
>>> rows=db(db.person.name.upper().like('%AX')).select()
>>> rows=db(~db.person.name.upper().like('%AX')).select()
```

### Usage of orderby, groupby and limitby

```
>>> people=db().select(db.person.name,orderby=db.person.name)
>>> order=db.person.name|~db.person.birth
>>> people=db().select(db.person.name,orderby=order)
>>> people=db().select(db.person.name,orderby=order,\
                       groupby=db.person.name)
>>> people=db().select(db.person.name,orderby=order,limitby=(0,100))
```

### Example of one to many relation

```
>>> tmp=db.define_table('dog', \
          SQLField('name'), \
          SQLField('birth','date'), \
          SQLField('owner',db.person),\
          migrate='test_dog.table')
>>> dog_id=db.dog.insert(name='Snoopy',birth=None,owner=person_id)
```

### A simple JOIN

```
>>> rows=db(db.dog.owner==db.person.id).select()
>>> for row in rows: print row.person.name,row.dog.name
Max Snoopy
```

### Example of many to many relation

```
>>> tmp=db.define_table('author',SQLField('name'),\
                        migrate='test_author.table')
>>> tmp=db.define_table('paper',SQLField('title'),\
                        migrate='test_paper.table')
>>> tmp=db.define_table('authorship',\
        SQLField('author_id',db.author),\
        SQLField('paper_id',db.paper),\
```

```
        migrate='test_authorship.table')
>>> aid=db.author.insert(name='Massimo')
>>> pid=db.paper.insert(title='QCD')
>>> tmp=db.authorship.insert(author_id=aid,paper_id=pid)
```

### SQLSet

```
>>> authored_papers=db((db.author.id==db.authorship.author_id)&\
                       (db.paper.id==db.authorship.paper_id))
>>> rows=authored_papers.select(db.author.name,db.paper.title)
>>> for row in rows: print row.author.name, row.paper.title
Massimo QCD
```

### Search with belongs

```
>>> set=(1,2,3)
>>> rows=db(db.paper.id.belongs(set)).select(db.paper.ALL)
>>> print rows[0].title
QCD
```

### Nested selects

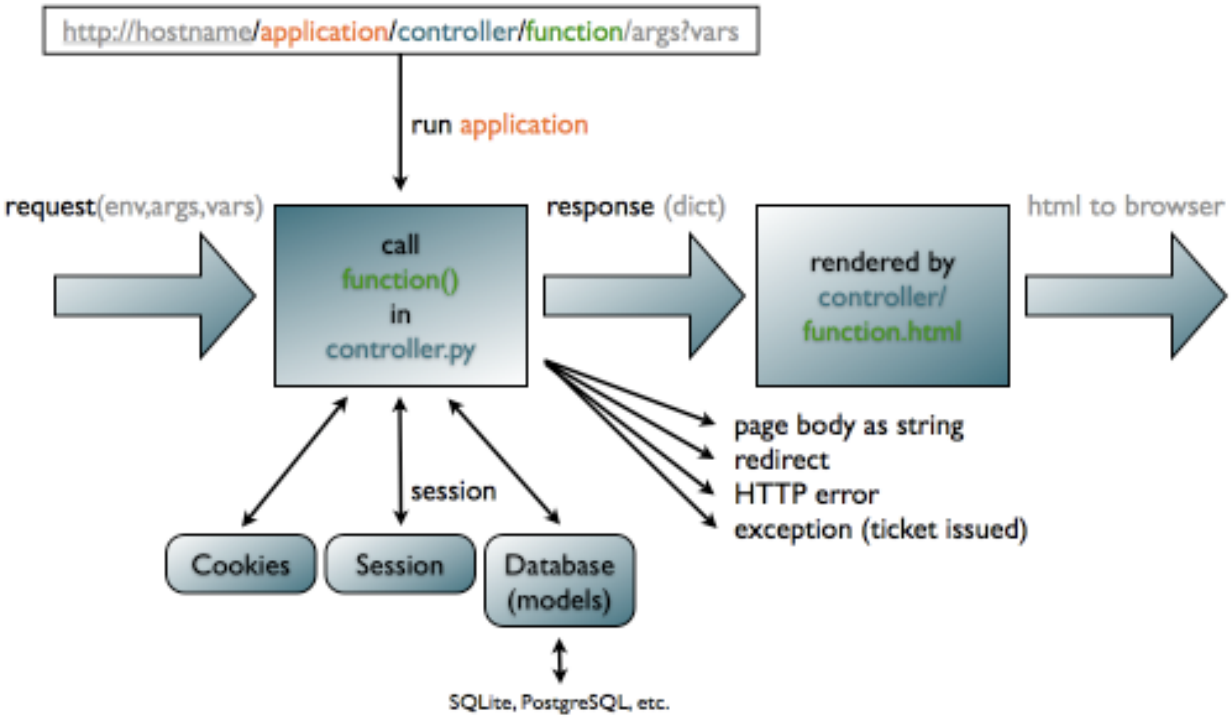
```
>>> nested_select=db()._select(db.authorship.paper_id)
>>> rows=db(db.paper.id.belongs(nested_select)).select(db.paper.ALL)
>>> print rows[0].title
QCD
```

### Output in CSV format

```
>>> str(authored_papers.select(db.author.name,db.paper.title))
'author.name,paper.title\r\nMassimo,QCD\r\n'
```

# web2py™ API

## URL mapping overview



## Container Objects

request, response, session, cache

## Navigation Functions and Objects

redirect, HTTP

## Internationalization

T

## Views Helpers

XML, URL, BEAUTIFY

## HTTP Building Objects

A, B, BODY, BR, CENTER, CODE, DIV, EM, EMBED, FORM,
H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, IMG, INPUT,
LI, LINK, LO, LU, META, OBJECT, ON, OPTION, P, PRE,
SCRIPT, SELECT, SPAN, STYLE, TABLE, TD,
TEXTAREA, TH,TITLE, TR, TT

## Validator Objects

IS_ALPHANUMERIC, IS_DATE, IS_DATETIME, IS_EMAIL,
IS_EXPR, IS_FLOAT_IN_RANGE, IS_INT_IN_RANGE, IS_IN_SET,
IS_LENGTH, IS_MATCH, IS_NOT_EMPTY, IS_TIME, IS_URL,
CLEANUP, CRYPT, IS_IN_DB, IS_NOT_IN_DB

## Database API

SQLDB, SQLField

## Database to HTML

SQLFORM, SQLTABLE

# web2py™ Examples

## Simple Examples

*Here are some working and complete examples that explain the basic syntax of the framework. You can click on the web2py keywords (in the highlighted code!) to get documentation.*

### Example 1

**In controller: simple_examples.py**

```
def hello1():
    return "Hello World"
```

If the controller function returns a string, that is the body of the rendered page.

### Example 2

**In controller: simple_examples.py**

```
def hello2():
    return T("Hello World")
```

The function T() marks strings that need to be translated. Translation dictionaries can be created at /admin/default/design

### Example 3

**In controller: simple_examples.py**

```
def hello3():
    return dict(message=T("Hello World"))
```

**and view: simple_examples/hello3.html**

```
{{extend 'layout.html'}}
<h1>{{=message}}</h1>
```

If you return a dictionary, the variables defined in the dictionery are visible to the view (template).

## Example 4

**In controller: simple_examples.py**

```
def hello4():
    response.view='simple_examples/hello3.html'
    return dict(message=T("Hello World"))
```

You can change the view, but the default is /[controller]/[function].html. If the default is not found web2py tries to render the page using the generic.html view.

## Example 5

**In controller: simple_examples.py**

```
def hello5():
    return HTML(BODY(H1(T('Hello World'),_style="color: red;"))).xml()
```

You can also generate HTML using helper objects HTML, BODY, H1, etc. Each of these tags is an class and the views know how to render the corresponding objects. The method .xml() serializes them and produce html/xml code for the page. Each tag, DIV for example, takes three types of arguments:

- unnamed arguments, they correspond to nested tags
- named arguments and name starts with '_'. These are mapped blindly into tag attributes and the '_' is removed. attributes without value like "READONLY" can be created with the argument "_readonly=ON".
- named arguments and name does not start with '_'. They have a special meaning. See "value=" for INPUT, TEXTAREA, SELECT tags later.

## Example 6
**In controller: simple_examples.py**

```
def status():
    return dict(request=request,session=session,response=response)
```

Here we are showing the request, session ad response objects using the generic.html template.

## Example 7
**In controller: simple_examples.py**

```
def redirectme():
    redirect(URL(r=request,f='hello3'))
```

You can do redirect.

## Example 8
**In controller: simple_examples.py**

```
def raisehttp():
    raise HTTP(400,"internal error")
```

You can raise HTTP exceptions to return an error page.

## Example 9
**In controller: simple_examples.py**

```
1
def raiseexception():
    1/0
    return 'oops'
```

If an exception occurs (other than HTTP) a ticket is generated and the event is logged for the administrator. These tickets and logs can be accessed, reviewed and deleted and any later time.

## Example 10
**In controller: simple_examples.py**

```
def servejs():
    import gluon.contenttype
    response.headers['Content-Type']=\
       gluon.contenttype.contenttype('.js')
    return 'alert("This is a Javascript document");'
```

You can serve other than HTML pages by changing the contenttype via the response.headers. The gluon.contenttype module can help you figure the type of the file to be server. NOTICE: this is not necessary for static files unless you want to require authorization.

## Example 11
**In controller: simple_examples.py**

```
def makejson():
    import gluon.contrib.simplejson as sj
    return sj.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
```

If you are into Ajax, web2py includes gluon.contrib.simplejson, developed by Bob Ippolito. This module provides a fast and easy way to serve asynchronous content to your Ajax page. gluon.simplesjson.dumps(...) can serialize most Python types into JSON. gluon.contrib.simplejson.loads(...) performs the reverse operation.

## Example 12
**In controller: simple_examples.py**

```
def makertf():
    import gluon.contrib.pyrtf as q
    doc=q.Document()
    section=q.Section()
    doc.Sections.append(section)
    section.append('Section Title')
    section.append('web2py is great. '*100)
    response.headers['Content-Type']='text/rtf'
    return q.dumps(doc)
```

web2py also includes gluon.contrib.pyrtf, developed by Simon Cusack and revised by Grant Edwards. This module allows you to generate Rich Text Format documents including colored formatted text and pictures.

### Example 13

**In controller: simple_examples.py**

```
def makerss():
    import datetime
    import gluon.contrib.rss2 as rss2
    rss = rss2.RSS2(
       title = "web2py feed",
       link = "http://mdp.cti.depaul.edu",
       description = "About web2py",
       lastBuildDate = datetime.datetime.now(),
       items = [
          rss2.RSSItem(
            title = "web2py and PyRSS2Gen-0.0",
            link = "http://mdp.cti.depaul.edu/",
            description = "web2py can now make rss feeds!",
            guid = rss2.Guid("http://mdp.cti.depaul.edu/"),
            pubDate = datetime.datetime(2007, 11, 14, 10, 30)),
        ])
    response.headers['Content-Type']='application/rss+xml'
    return rss2.dumps(rss)
```

web2py also includes gluon.contrib.rss2, developed by Dalke Scientific Software. It generates RSS2 feeds.

## Session Examples

### Example 14

**In controller: session_examples.py**

```
def counter():
    if not session.counter: session.counter=0
    session.counter+=1
    return dict(counter=session.counter)
```

**and view: session_examples/counter.html**

```
{{extend 'layout.html'}}
<h1>session counter</h1>
<h2>{{for i in range(counter):}}{{=i}}...{{pass}}</h2>
<a href="{{=URL(r=request)}}">click me to count</a>
```

Click to count. The session.counter is persistent for this user and application. Every applicaiton within the system has its own separate session management.

# Template Examples

## Example 15

**In controller: template_examples.py**

```
def variables(): return dict(a=10, b=20)
```

**and view: template_examples/variables.html**

```
{{extend 'layout.html'}}
<h1>Your variables</h1>
<h2>a={{=a}}</h2>
<h2>a={{=b}}</h2>
```

A view (also known as template) is just an HTML file with {{...}} tags. You can put ANY python code into the tags, no need to indent but you must use pass to close blocks. The view is transformed into a python code and then executed. {{=a}} prints a.xml() or escape(str(a)).

## Example 16

**In controller: template_examples.py**

```
def test_for(): return dict()
```

**and view: template_examples/test_for.html**

```
<h1>For loop</h1>
{{for number in ['one','two','three']:}}
  <h2>{{=number.capitalize()}}<h2>
```

```
{{pass}}
```

You can do for and while loops.

## Example 17

**In controller: template_examples.py**

```
def test_if(): return dict()
```

**and view: template_examples/test_if.html**

```
{{extend 'layout.html'}}
<h1>If statement</h1>
{{a=10}}
{{if a%2==0:}}
<h2>{{=a}} is even</h2>
{{else:}}
<h2>{{=a}} is odd</h2>
{{pass}}
```

You can do if, elif, else.

## Example 18

**In controller: template_examples.py**

```
def test_try(): return dict()
```

**and view: template_examples/test_try.html**

```
{{extend 'layout.html'}}
<h1>Try... except</h1>
{{try:}}
   <h2>a={{=1/0}}</h2>
{{except:}}
   infinity</h2>
{{pass}}
```

You can do try, except, finally.

## Example 19

**In controller: template_examples.py**

```
def test_def(): return dict()
```

**and view: template_examples/test_def.html**

```
{{extend 'layout.html'}}
{{def itemlink(name):}}<li>{{=A(name,_href=name)}}</li>{{return}}
<ul>
{{itemlink('http://www.google.com')}}
{{itemlink('http://www.yahoo.com')}}
{{itemlink('http://www.nyt.com')}}
</ul>
```

You can write functions in HTML too.

## Example 20

**In controller: template_examples.py**

```
def escape(): return dict(message='<h1>text is scaped</h1>')
```

**and view: template_examples/escape.html**

```
{{extend 'layout.html'}}
<h1>Strings are automatically escaped</h1>

<h2>Message is</h2>
{{=message}}
```

The argument of {{=...}} is always escaped unless it is an object with a .xml() method such as link, A(...), a FORM(...), a XML(...) block, etc.

## Example 21

**In controller: template_examples.py**

```
def xml():
    return dict(message=XML('<h1>text is not escaped</h1>'))
```

**and view: template_examples/xml.html**

```
{{extend 'layout.html'}}
<h1>XML</h1>
<h2>Message is</h2>
{{=message}}
```

If you do not want to esacpe the argument of {{=...}} mark it as XML.

## Example 22

**In controller: template_examples.py**

```
def beautify(): return dict(message=BEAUTIFY(request))
```

**and view: template_examples/beautify.html**

```
{{extend 'layout.html'}}
<h1>BEAUTIFY</h1>
<h2>Message is</h2>
{{=message}}
```

You can use BEUTIFY to turn lists and dictionaries into organized HTML.

# Layout Examples

## Example 23

**In controller: layout_examples.py**

```
def civilized():
    response.menu=[['civilized',True,URL(r=request,f='civilized')],
                   ['slick',False,URL(r=request,f='slick')],
                   ['basic',False,URL(r=request,f='basic')]]
    response.flash='you clicked on civilized'
    return dict(message="you clicked on civilized")
```

**and view: layout_examples/civilized.html**

```
{{extend 'layout_examples/layout_civilized.html'}}
   <h2>{{=message}}</h2>
<p>{{for i in range(1000):}}bla {{pass}} </p>
```

You can specify the layout file at the top of your view. civilized Layout file is a view that somewhere in the body contains {{include}}.

## Example 24

**In controller: layout_examples.py**

```
def slick():
    response.menu=[['civilized',False,URL(r=request,f='civilized')],
                   ['slick',True,URL(r=request,f='slick')],
                   ['basic',False,URL(r=request,f='basic')]]
    response.flash='you clicked on slick'
    return dict(message="you clicked on slick")
```

**and view: layout_examples/slick.html**

```
{{extend 'layout_examples/layout_sleek.html'}}
<h2>{{=message}}</h2>
{{for i in range(1000):}}bla {{pass}}
```

Same here, but using a different template.

## Example 25

**In controller: layout_examples.py**

```
def basic():
    response.menu=[['civilized',False,URL(r=request,f='civilized')],
                   ['slick',False,URL(r=request,f='slick')],
                   ['basic',True,URL(r=request,f='basic')]]
    response.flash='you clicked on basic'
    return dict(message="you clicked on basic")
```

**and view: layout_examples/basic.html**

```
{{extend 'layout.html'}}
<h2>{{=message}}</h2>
{{for i in range(1000):}}bla {{pass}}
```

'layout.html' is the default template, every applicaiton has a copy of it.

# Form Examples

## Example 26

**In controller: form_examples.py**

```
def form():
    form=FORM(TABLE(TR("Your name:",INPUT(_type="text",_name="name",
              requires=IS_NOT_EMPTY())),
           TR("Your email:",INPUT(_type="text",_name="email",
              requires=IS_EMAIL())),
           TR("Admin",INPUT(_type="checkbox",_name="admin")),
           TR("Sure?",SELECT('yes','no',_name="sure",
              requires=IS_IN_SET(['yes','no']))),
           TR("Profile",TEXTAREA(_name="profile",
              value="write something here")),
           TR("",INPUT(_type="submit",_value="SUBMIT"))))
    if form.accepts(request.vars,session):
        response.flash="form accepted!"
    else:
        response.flash="form is invalid!"
    return dict(form=form,vars=form.vars)
```

You can use HTML helpers like FORM, INPUT, TEXTAREA, OPTION, SELECT to build forms. the "value=" attribute sets the initial value of the field (works for TEXTAREA and OPTION/SELECT too) and the requires attribute sets the validators. FORM.accepts(..) trys to validate the form and, on success, stores vars into form.vars. On failure the error messages are stored into form.errors and shown in the form.

# Database Examples

You can find more examples of the web2py ORM here

Let's create a simple model with users, dogs, products and purchases (the database of an animal store). Users can have many dogs (ONE TO MANY), can buy many producs and every product can have many buyers (MANY TO MANY).

## Example 27
**in model: dba.py**

```
dba=SQLDB('sqlite://tests.db')

dba.define_table('users',
                 SQLField('name'),
                 SQLField('email'))

# ONE (users) TO MANY (dogs)
dba.define_table('dogs',
                 SQLField('owner_id',dba.users),
                 SQLField('name'),
                 SQLField('type'),
                 SQLField('vaccinated','boolean',default=False),
                 SQLField('picture','upload',default=''))

dba.define_table('products',
                 SQLField('name'),
                 SQLField('description','blob'))

# MANY (users) TO MANY (products)
dba.define_table('purchases',
                 SQLField('buyer_id',dba.users),
                 SQLField('product_id',dba.products),
                 SQLField('quantity','integer'))

purchased=((dba.users.id==dba.purchases.buyer_id)&(dba.products.id==db
a.purchases.product_id))

dba.users.name.requires=IS_NOT_EMPTY()
dba.users.email.requires=[IS_EMAIL(), IS_NOT_IN_DB(dba,'users.email')]
dba.dogs.owner_id.requires=IS_IN_DB(dba,'users.id','users.name')
dba.dogs.name.requires=IS_NOT_EMPTY()
dba.dogs.type.requires=IS_IN_SET(['small','medium','large'])
dba.purchases.buyer_id.requires=IS_IN_DB(dba,'users.id','users.name')
dba.purchases.product_id.requires=IS_IN_DB(dba,'products.id','products
.name')
dba.purchases.quantity.requires=IS_INT_IN_RANGE(0,10)
```

Tables are created if they do not exist (try... except). Here "purchased" is an SQLQuery object, "dba(purchased)" would be a SQLSet obejcts. A SQLSet object can be selected, updated, deleted. SQLSets can also be intersected. Allowed field types are string, integer, password, text, blob, upload, date, time, datetime, references(*), and id(*). The id field is there by default and must not be declared. references are for one to many and many to many as in the example above. For strings you should specify a length or you get length=32.

You can use dba.tablename.fieldname.requires= to set restrictions on the field values. These restrictions are automatically converted into widgets when generating forms from the table with SQLFORM(dba.tablename).

define_tables creates the table and attempts a migration if table has changed or if database name has changed since last time. If you know you already have the table in the database and you do not want to attemt a migration add one last argument to define_table `migrate=False`.

## Example 28

**In controller: database_examples.py**

```
response.menu=[['Register User',False,URL(r=request,f='regis-
ter_user')],
               ['Register Dog',False,URL(r=request,f='register_dog')],
               ['Register
Product',False,URL(r=request,f='register_product')],
               ['Buy product',False,URL(r=request,f='buy')]]

def register_user():
    ### create an insert form from the table
    form=SQLFORM(dba.users)
    ### if form correct perform the insert
    if form.accepts(request.vars,session):
        response.flash='new record inserted'
    ### and get a list of all users
```

```
    records=SQLTABLE(dba().select(dba.users.ALL))
    return dict(form=form,records=records)
```

**and view: database_examples/register_user.html**

```
{{extend 'layout_examples/layout_civilized.html'}}
<h1>User registration form</h1>
{{=form}}
<h2>Current users</h2>
{{=records}}
```

This is a simple user registration form. SQLFORM takes a table and returns the corresponding entry form with validators, etc. SQLFORM.accepts is similar to FORM.accepts but, if form is validated, the corresponding insert is also performed. SQLFORM can also do update and edit if a record is passed as its second argument. SQLTABLE instead turns a set of records (result of a select) into an HTML table with links as specified by its optional parameters. The response.menu on top is just a variable used by the layout to make the navigation menu for all functions in this controller.

## Example 29

**In controller: database_examples.py**

```
def register_dog():
    form=SQLFORM(dba.dogs)
    if form.accepts(request.vars,session):
        response.flash='new record inserted'
    download=URL(r=request,f='download') # to see the picture
    records=SQLTABLE(dba().select(dba.dogs.ALL),upload=download)
    return dict(form=form,records=records)
```

**and view: database_examples/register_dog.html**

```
{{extend 'layout_examples/layout_civilized.html'}}
<h1>Dog registration form</h1>
{{=form}}
<h2>Current dogs</h2>
{{=records}}
```

Here is a dog registration form. Notice that the "image" (type "upload") field is rendered into a <INPUT type="file"> html tag. SQLFORM.accepts(...) handles the upload of the file into the uploads/ folder.

## Example 30

**In controller: database_examples.py**

```
def register_product():
    form=SQLFORM(dba.products)
    if form.accepts(request.vars,session):
        response.flash='new record inserted'
    records=SQLTABLE(dba().select(dba.products.ALL))
    return dict(form=form,records=records)
```

**and view: database_examples/register_product.html**

```
{{extend 'layout_examples/layout_civilized.html'}}
<h1>Product registration form</h1>
{{=form}}
<h2>Current products</h2>
{{=records}}
```

Nothing new here.

## Example 31

**In controller: database_examples.py**

```
def buy():
    form=FORM(TABLE(TR("Buyer id:",INPUT(_type="text",
              _name="buyer_id",requires=IS_NOT_EMPTY())),
                    TR("Product id:",INPUT(_type="text",
              _name="product_id",requires=IS_NOT_EMPTY())),
                    TR("Quantity:",INPUT(_type="text",
              _name="quantity",requires=IS_INT_IN_RANGE(1,100))),
                    TR("",INPUT(_type="submit",_value="Order"))))
    if form.accepts(request.vars,session):
        ### check if user is in the database
        if len(dba(dba.users.id==form.vars.buyer_id).select())==0:
            form.errors.buyer_id="buyer not in database"
        ### check if product is in the database
        if len(dba(dba.products.id==form.vars.product_id)\
           .select())==0:
            form.errors.product_id="product not in database"
        ### if no errors
        if len(form.errors)==0:
```

```
            ### get a list of same purchases by same user
            purchases=dba(
              (dba.purchases.buyer_id==form.vars.buyer_id)&
              (dba.purchases.product_id==form.vars.product_id)\
            ).select()
            ### if list contains a record, update that record
            if len(purchases)>0:
                purchases[0].update_record(quantity=\
                           purchases[0].quantity+form.vars.quantity)
            ### or insert a new record in table
            else:
                dba.purchases.insert(buyer_id=form.vars.buyer_id,
                                     product_id=form.vars.product_id,
                                     quantity=form.vars.quantity)
            response.flash="product purchased!"
    if len(form.errors): response.flash="invalid valus in form!"
    ### now get a list of all purchases
    records=dba(purchased).select(dba.users.name, \
            dba.purchases.quantity,dba.products.name)
    return dict(form=form,records=SQLTABLE(records),
                vars=form.vars,vars2=request.vars)
```

**and view: database_examples/buy.html**

```
{{extend 'layout_examples/layout_civilized.html'}}
  <h1>Purchase form</h1>
  {{=form}}
  [ {{=A('reset purchased',_href=URL(r=request,f='reset_purchased'))}}
  | {{=A('delete purchased',
       _href=URL(r=request,f='delete_purchased'))}} ]<br/>
  <h2>Current purchases (SQL JOIN!)</h2>
<p>{{=records}}</p>
```

Here is a rather sophisticated buy form. It checks that the buyer and the product are in the database and updates the corresponding record or inserts a new purchase. It also does a JOIN to list all purchases.

## Example 32

**In controller: database_examples.py**

```
def delete_purchased():
    dba(dba.purchases.id>0).delete()
    redirect(URL(r=request,f='buy'))
```

### Example 33
**In controller: database_examples.py**

```
def reset_purchased():
    dba(dba.purchases.id>0).update(quantity=0)
    redirect(URL(r=request,f='buy'))
```

This is an update on an SQLSet. (dba.purchase.id>0 identifies the set containing only table dba.purchases.)

### Example 34
**In controller: database_examples.py**

```
def download():
    import gluon.contenttype
    filename=request.args[0]
    response.headers['Content-Type']=\
         gluon.contenttype.contenttype(filename)
    return open('applications/%s/uploads/%s' %
         (request.application,filename),'rb').read()
```

This controller allows users to download the uploaded pictures of the dogs. Remember the upload=URL(...'download'...) statement in the register_dog function. Notice that in the URL path /application/controller/function/a/b/etc a, b, etc are passed to the controller as request.args[0], request.args[1], etc. Since the URL is validated request.args[] always contain valid filenames and no '~' or '..' etc. This is usefult to allow visitors to link uploaded files.

## Cache Examples

### Example 35
**In controller: cache_examples.py**

```
def cache_in_ram():
    import time
    t=cache.ram('time',lambda:time.ctime(),time_expire=5)
    return dict(time=t,link=A('click to reload',_href=URL(r=request)))
```

The output of `lambda:time.ctime()` is cached in ram for 5 seconds. The string 'time' is used as cache key.

## Example 36

**In controller: cache_examples.py**

```
def cache_on_disk():
    import time
    t=cache.disk('time',lambda:time.ctime(),time_expire=5)
    return dict(time=t,link=A('click to reload',_href=URL(r=request)))
```

The output of `lambda:time.ctime()` is cached on disk (using the shelve module) for 5 seconds.

## Example 37

**In controller: cache_examples.py**

```
def cache_in_ram_and_disk():
    import time
    t=cache.ram('time',lambda:cache.disk('time',\
      lambda:time.ctime(),time_expire=5),time_expire=5)
    return dict(time=t,link=A('click to reload',_href=URL(r=request)))
```

The output of `lambda:time.ctime()` is cached on disk (using the shelve module) and then in ram for 5 seconds. web2py looks in ram first and if not there it looks on disk. If it is not on disk it calls the function. This is useful in a multiprocess type of environment. The two times do not have to be the same.

## Example 38

**In controller: cache_examples.py**

```
@cache(request.env.path_info,time_expire=5,cache_model=cache.ram)
def cache_controller_in_ram():
    import time
    t=time.ctime()
    return dict(time=t,link=A('click to reload',_href=URL(r=request)))
```

Here the entire controller (dictionary) is cached in ram for 5 seconds. The result of a select cannot be cached unless it is first serialized into a table `lambda:SQLTABLE(dba().select(dba.users.ALL)).xml()`. You can read below for an even better way to do it.

## Example 39

**In controller: cache_examples.py**

```
@cache(request.env.path_info,time_expire=5,cache_model=cache.disk)
def cache_controller_on_disk():
    import time
    t=time.ctime()
    return dict(time=t,link=A('click to reload',_href=URL(r=request)))
```

Here the entire controller (dictionary) is cached on disk for 5 seconds. This will not work if the dictionary contains unpickleble objects.

## Example 40

**In controller: cache_examples.py**

```
@cache(request.env.path_info,time_expire=5,cache_model=cache.ram)
def cache_controller_and_view():
    import time
    t=time.ctime()
    d=dict(time=t,link=A('click to reload',_href=URL(r=request)))
    return response.render(d)
```

`response.render(d)` renders the dictionary inside the controller, so everything is cached now for 5 seconds. This is best and fastest way of caching!

## Example 41

**In controller: cache_examples.py**

```
def cache_db_select():
    import time
    dba.users.insert(name='somebody',email='gluon@mdp.cti.depaul.edu')
    records=dba().select(dba.users.ALL,cache=(cache.ram,5))
    if len(records)>20: dba(dba.users.id>0).delete()
    return dict(records=records)
```

The results of a select are complex unpickable objects that cannot be cached using the previous method, but the select command takes an argument `cache=(cache_model,time_expire)` and will cache the result of the query accordingly. Notice that the key is not necessary since key is generated based on the database name and the select string.

# Ajax Examples

## Example 42

**In controller: ajax_examples.py**

```
def index():
    return dict()

def data():
    if not session.m or len(session.m)==10: session.m=[]
    if request.vars.q: session.m.append(request.vars.q)
    session.m.sort()
    return TABLE(*[TR(v) for v in session.m]).xml()
```

**In view: ajax_examples/index.html**

```
{{extend 'layout.html'}}
<p>Type something and press the button. The last 10 entries will ap-
pear sorted in a table below.</p>
<form>
<INPUT type="text" id='q' value="web2py"/>
<INPUT type="button" value="submit"
   onclick="ajax('{{=URL(r=request,f='data')}}',['q'],'target');"/>
</form>
<br/>
<div id="target"></div>
```

The javascript function "ajax" is provided in "web2py_ajax.html" and included by "layout.html". It takes three arguments, a url, a list of ids and a target it. When called it send to the url (via a get) the values of the ids and display the respose in the value (of innerHTML) of the target id.

## Example 43

### In controller: ajax_examples.py

```
def flash():
    response.flash='this text should appear!'
    return dict()
```

## Example 44

### In controller: ajax_examples.py

```
def fade():
    return dict()
```

### In view: ajax_examples/fade.html

```
{{extend 'layout.html'}}
<form>
<input type="button" onclick="fade('test',-0.2);" value="fade down"/>
<input type="button" onclick="fade('test',+0.2);" value="fade up"/>
</form>
<div id="test">{{='Hello World '*100}}</div>
```

# web2py™ Object Relational Mapper API

## Examples

```
>>> db=SQLDB("sqlite://test.db")
>>> #OR db=SQLDB("mysql://username:password@host:port/dbname")
>>> #OR db=SQLDB("postgres://username:password@host:port/dbname")

# syntax: SQLField('fieldname','fieldtype',length=32,
#            required=False, default=None,
#            requires=[IS_EMAIL(error_message='invalid email')])

>>> tmp=db.define_table('users',\
   SQLField('stringfield','string',length=32,required=True),\
   SQLField('booleanfield','boolean',default=False),\
   SQLField('passwordfield','password'),\
   SQLField('textfield','text'),\
   SQLField('blobfield','blob'),\
   SQLField('uploadfield','upload'),\
   SQLField('integerfield','integer'),\
   SQLField('doublefield','double'),\
   SQLField('datefield','date',default=datetime.date.today()),\
   SQLField('timefield','time'),\
   SQLField('datetimefield','datetime'),\
   migrate='test_user.table')

# Insert a field

>>> db.users.insert(stringfield='a',booleanfield=True,\
             passwordfield='p',textfield='x',blobfield='x',\
             uploadfield=None,
             integerfield=5,doublefield=3.14,\
             datefield=datetime.date(2001,1,1),\
             timefield=datetime.time(12,30,15),\
             datetimefield=datetime.datetime(2002,2,2,12,30,15))
1

# Drop the table

>>> db.users.drop()

# Examples of insert, select, update, delete

>>> tmp=db.define_table('person',\
          SQLField('name'), \
          SQLField('birth','date'),\
          migrate='test_person.table')
```

```
    >>> person_id=db.person.insert(name="Marco",birth='2005-06-22')
    >>> person_id=db.person.insert(name="Massimo",birth='1971-12-21')
    >>> len(db().select(db.person.ALL))
    2
    >>> me=db(db.person.id==person_id).select()[0] # test select
    >>> me.name
    'Massimo'
    >>> db(db.person.name=='Massimo').update(name='massimo')
    >>> db(db.person.name=='Marco').delete() # test delete

    Update a single record

    >>> me.update_record(name="Max")
    >>> me.name
    'Max'

    Examples of complex search conditions

    >>> len(db((db.person.name=='Max')&\
        (db.person.birth<'2003-01-01')).select())
    1
    >>> len(db((db.person.name=='Max')|\
        (db.person.birth<'2003-01-01')).select())
    1
    >>> me=db(db.person.id==person_id).select(db.person.name)[0]
    >>> me.name
    'Max'

    # Examples of search conditions using extract from date/datetime/
time

    >>> len(db(db.person.birth.month()==12).select())
    1
    >>> len(db(db.person.birth.year()>1900).select())
    1

    Example of usage of NULL

    >>> len(db(db.person.birth==None).select()) ### test NULL
    0
    >>> len(db(db.person.birth!=None).select()) ### test NULL
    1

    # Examples of search consitions using lower, upper, and like

    >>> len(db(db.person.name.upper()=='MAX').select())
    1
    >>> len(db(db.person.name.like('%ax')).select())
    1
```

```
    >>> len(db(db.person.name.upper().like('%AX')).select())
    1
    >>> len(db(~db.person.name.upper().like('%AX')).select())
    0

    # orderby, groupby and limitby

    >>> people=db().select(db.person.name,orderby=db.person.name)
    >>> order=db.person.name|~db.person.birth
    >>> people=db().select(db.person.name,orderby=order)
    >>> people=db().select(db.person.name,orderby=order,\
                           groupby=db.person.name)
    >>>
people=db().select(db.person.name,orderby=order,limitby=(0,100))

    # Example of one 2 many relation

    >>> tmp=db.define_table('dog', \
              SQLField('name'), \
              SQLField('birth','date'), \
              SQLField('owner',db.person),\
              migrate='test_dog.table')
    >>> db.dog.insert(name='Snoopy',birth=None,owner=person_id)
    1

    # A simple JOIN

    >>> len(db(db.dog.owner==db.person.id).select())
    1

    # Drop tables

    >>> db.dog.drop()
    >>> db.person.drop()

    # Example of many 2 many relation and SQLSet

    >>> tmp=db.define_table('author',SQLField('name'),\
                            migrate='test_author.table')
    >>> tmp=db.define_table('paper',SQLField('title'),\
                            migrate='test_paper.table')
    >>> tmp=db.define_table('authorship',\
            SQLField('author_id',db.author),\
            SQLField('paper_id',db.paper),\
            migrate='test_authorship.table')
    >>> aid=db.author.insert(name='Massimo')
    >>> pid=db.paper.insert(title='QCD')
    >>> tmp=db.authorship.insert(author_id=aid,paper_id=pid)
```

```
# Define a SQLSet

>>> authored_papers=db((db.author.id==db.authorship.author_id)&\
                       (db.paper.id==db.authorship.paper_id))
>>> rows=authored_papers.select(db.author.name,db.paper.title)
>>> for row in rows: print row.author.name, row.paper.title
Massimo QCD

# Example of search condition using  belongs

>>> set=(1,2,3)
>>> rows=db(db.paper.id.belongs(set)).select(db.paper.ALL)
>>> print rows[0].title
QCD

# Example of search condition using nested select

>>> nested_select=db()._select(db.authorship.paper_id)
>>> rows=db(db.paper.id.belongs(nested_select))\
         .select(db.paper.ALL)
>>> print rows[0].title
QCD

# Output in csv

>>> str(authored_papers.select(db.author.name,db.paper.title))
author.name,paper.title
Massimo,QCD

# Delete all leftover tables

>>> db.authorship.drop()
>>> db.author.drop()
>>> db.paper.drop()

# Commit or rollback your work

>>> db.commit() # or db.rollback()
```

`migrate` can be False (do not create/alter tables), True (create/alter tables) or a filename (create/alter tables and store migration information in the file).

Mind there are little idiosyncrasies like the fact that "user" is not a valid field name in PostgreSQL, or the fact that sqlite3 will ignore the type of a field and al-

low you to put anything in it despite the declared type. Every database backend has its own keywords that may conflict with your table names.