web2py

Enterprise Web Framework

VERY VERY DRAFT MANUAL 0.1

by Massimo Di Pierro

Contents

Chapter 1

Introduction

web2py is a free and open source full-stack web framework for agile development of secure database-driven web applications; it is written in Python and programmable in Python. Full-stack means that web2py comprises of multiple libraries that are very well integrated and useful to web developers, thus making web2py more complex and functional than just the sum of its components.

web2py follows the Model View Controller (MVC) software engineering design pattern that forces the developer to separate the data representation (the model) from the data presentation (the view) and the application logic and workflow (the controller). web2py provides libraries to help the developer design, implement, and test each of these three parts.

web2py differs from other web frameworks because it is the only framework to fully embrace the Web 2.0 paradigm where the web is the computer. In fact, web2py does not require installation or configuration; it runs on any architecture (Windows, Mac and Unix/Linux); and the development, deployment, and maintenance phases for the applications can be done via a local or remote web interface.

web2py also provides a ticketing system. If an error occurs, a ticket is issued to the user for tracking and the error is logged for the administrator.

web2py is open source and released under the GPL2.0 license, but web2py developed applications are not subject to any license constraint, as long as they do not explictely contain web2py source code. In fact, web2py allows the developer to byte-code compile applications and distribute them as closed source, although they will require web2py to run.

web2py forces the developer to follow sound and secure software engineer-

ing practices (such as the Model-View-Controller pattern, forms validation and self-submission, secure session cookies) and helps the developer write concise, functional, clean, and portable code.

Here are some examples of web2py statements, taken out of context, to show off its power and simplicity:

```
db.define_table('image',
   SQLField('title','string'),
   SQLField('picture','upload'))
```

creates a database table called "image" with two fields: "title", a string; and "picture", something that needs to be uploaded (the actual image). If this table already exists it is altered appropriately. Given this table

```
form=SQLFORM(db.image)
```

creates an HTML form for this table that allows users to upload pictures. The following statement:

```
if form.accepts(request.vars,session): pass
```

validates a submitted form, renames the uploaded picture in a secure way, stores the picture in a file, inserts the correspoding record in the database, prevents double submission by assigning a unique key to the form, and eventually modifies the form by adding error messages if the data submitted by the user is not validated.

1.1 Name and principles

Python programming typically follows these basic principles:

- do not repeat yourself
- there should be only one way of doing things
- explicit is better than implicit

web2py fully embraces the first two principles by forcing the developer to use sound software engineering practices that discourage repetition of code. web2py guides the developer through almost all the tasks common to any web application development (creating and processing forms, managing sessions, cookies, errors, etc.).

web2py differs from other frameworks with regard to the third principle because it sometimes conflicts with the other two. In particular, web2py automatically imports its own modules and instantiates some global objects (request, response, session, cache, T) and this is "done under the hood". To some this may appear as magic, but it should not. web2py is just trying to avoid the annoying characteristic of other frameworks that force the developer to import the same modules at the top of every model and controller.

web2py, by importing its own modules, just saves the developer time and prevents the developer from making mistakes; thus following the spirit of "do not repeat yourself" and "there should be only one way of doing things".

If the developer wishes to use other Python modules or third party modules, it can be done, but those modules have to be imported explicitly.

1.2 On Web Frameworks

At its most fundamental level, a web application consists of a set of programs (or functions) that are executed when a URL is visited. The output of the program is returned to the visitor and rendered by the browser.

The two classic approaches for developing web applications consists of 1) generating HTML programmatically and embedding HTML as strings into computer code; 2) embedding pieces of code into HTML pages. The second model is the one followed by PHP (where the code is in PHP, a C-like language), ASP (where the code is in Visual Basic), and JSP (where the code is in Java).

Here we present an example of a PHP progam that, once executed, retrieves data from a database and returns an HTML page showing the selected records.

```
1  <html><body><h1>Records</h1><?
2    mysql_connect(localhost,username,password);
3    @mysql_select_db(database) or die( "Unable to select database"
       );
4    $query="SELECT * FROM contacts";
5    $result=mysql_query($query);
6    mysql_close();
7    $i=0;
8    while ($i < mysql_numrows($result)) {
9      $name=mysql_result($result,$i,"name");
10     $phone=mysql_result($result,$i,"phone");
11     echo "<b>$name</b><br>Phone:$phone<br><br><hr><br>";
```

```
12      $i++;
13    }
14  ?></body></html>
```

The problem with this approach is that code is embedded into HTML but also the code needs to generate additional HTML and needs to generate SQL to query the database; hence, multiple layers of the application are entangled and make the application very complicated and difficult to maintain. The situation is even worse for AJAX applications and the complexity of PHP code grows with the number of pages (files) that comprise the application.

The same functionality of the above example can be expressed in web2py with one line of Python code:

```
1  def index():
2      return HTML(BODY(H1('Records'),db().select(db.contacts.ALL))
           )
```

In this simple example, the HTML page structure is represented programmatically by the `HTML`, `BODY`, and `H1` objects; the database `db` is queried by the `select` command; finally everything is serialized into HTML.

This is just one example of the power of web2py and its built-in libraries. web2py does even more for the developer by automatically handling cookies, sessions, creation of database tables, database modifications, form validation, SQL injection prevention, cross-site scripting (XSS) prevention, and many other web application tasks.

Web frameworks are typically categorized as one of two types of frameworks. A "glued" framework is built by assemblying (or gluing together) several third party components. A "full-stack" framework is built by creating components designed specifically to work together and they are tightly integrated.

web2py is a full-stack framework. Almost all its components are built from scratch and designed to work together, but they function just as well outside of the complete web2py framework. For example, the Object Relational Mapper (ORM) or the templating language can be used independely of the web2py framework by importing `gluon.sql` or `gluon.template` in your own Python applications. `gluon` is the name of the web2py folder that contains system libraries. Some of the web2py libraries, such as the building and processing forms from database tables have dependencies on other portions of web2py. web2py can also work with third party Python libraries, including other templating languages and ORMs but, they will not be as tightly integrated as the original components.

1.3 Model-View-Controller

web2py follows the Model-View-Controller (MVC) software engineering design pattern. This means web2py forces the developer to separate the Model (the data representation) from the View (the data presentation) and from the Controller (the application workflow). Let's consider again the above example and see how to build a web2py application around it.

The application would consist of three files:

- **db.py** is the Model:

```
db=SQLDB('sqlite://mydb.db')
db.define_table('contacts',
   SQLField('name','string',length=20),
   SQLField('phone','string',length=12))
```

  It connects to the database (in this example a SQLite database stored in the `mydb.db` file) and defines a table called `contacts`. If the table does not exist, web2py creates it. web2py, transparently and in backrgoud, generates the appropriate SQL dialect for the database backend. The developer can see the generated SQL but does not need to change the code if the database backend is replaced with PostgreSQL, MySQL or Oracle. Once a table is defined and created, web2py also generates a fully functional web based database administration interface to access the database and the tables.

- **default.py** is the Controller:

```
def contacts():
    return dict(records=db().select(db.contacts.ALL))
```

  In web2py URLs are mapped into Python modules and function calls. In this case we declared a single function called `contacts`. A function may return a string (and the string would be the returned web page) or a Python dictionary (i.e. a set of key:value). If the function returns a dictionary, the dictionary is passed to a view with the same name as the controller/function and rendered by the view. In this example the function contacts performs a database select and returns the resulting records as a value associated to the dictionary key `records`.

- **default/contacts.html** is the View:

```
{{extend 'layout.html'}}
<h1>Records</h1>
{{for record in records:}}
{{=record.name}}: {{=record.phone}}<br/>
{{pass}}
```

This view is called automatically by web2py after the associated controller function is executed. The purpose of this view is to render the dictionary `dict(records=...)` into HTML. The view is written into HTML and it embeds Python code enclosed by the special {{ and }} delimiters. Notice that this is very different than the PHP code example because the only code embedded into the HTML is "presentation layer code". The `layout.html` file referenced at the top of the view is provided by web2py and constitutes the basic layout for web2py applications and it can easily be modified or removed.

1.4 Why web2py

web2py is one of many web application frameworks, but it has a few compelling unique features. web2py was originally developed as a teaching tool and the primary motivations for writing web2py were the following:

- Easy for users to learn server-side web development without compromising functionality. For this reason web2py requires no installation, no configuration, has no library dependencies, and exposes most of its functionalities via a web interface.

- Stable from day one by following a top-down design as opposed to other web frameworks that follow a bottom-up design and are typically an ongoing work-in-progress. This usually means the project documentation is out of sync with the web framework, a frustrating dilemma for users.

- Address the most important secuity issues that plague many modern web applications (as determined by OWASP[1]).

[1] http://www.owasp.org/.

1.5 Security

OWASP has listed the top ten security issues that put web applications at risk. Listed here are those issues and a brief comment on how web2py deals with it:

- "Cross Site Scripting (XSS): XSS flaws occur whenever an application takes user supplied data and sends it to a web browser without first validating or encoding that content. XSS allows attackers to execute script in the victim's browser which can hijack user sessions, deface web sites, possibly introduce worms, etc."
  web2py, by default, escapes all variables rendered in the view, thus preventing XSS.

- "Injection Flaws: Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is sent to an interpreter as part of a command or query. The attacker's hostile data tricks the interpreter into executing unintended commands or changing data."
  web2py includes an Object Relational Mapper that makes SQL injection impossible. In fact, SQL is not written by the developer but it is written dynamically by the ORM and ensures all inserted data is properly escaped.

- "Malicious File Execution: Code vulnerable to remote file inclusion (RFI) allows attackers to include hostile code and data, resulting in devastating attacks, such as total server compromise. Malicious file execution attacks affect PHP, XML and any framework which accepts filenames or files from users."
  web2py allows only exposed functions to be executed and thus prohibits malicious file execution. web2py's web based administration interface makes it very easy to keep track of what is exposed and what is not.

- "Insecure Direct Object Reference: A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, database record, or key, as a URL or form parameter. Attackers can manipulate those references to access other objects without authorization."

web2py does not expose any internal objects;moreover, web2py validates all URLs thus preventing directory traversal attacks. web2py also provides a simple mechanism to create forms which automatically validate all input values.

- "Cross Site Request Forgery (CSRF): A CSRF attack forces a logged-on victim's browser to send a pre-authenticated request to a vulnerable web application, which then forces the victim's browser to perform a hostile action to the benefit of the attacker. CSRF can be as powerful as the web application that it attacks."
  web2py stores all session information server side and only stores the session id in a cookie; moreover, web2py prevents double submission of forms by assigning a one-time random token to each form.

- "Information Leakage and Improper Error Handling: Applications can unintentionally leak information about their configuration, internal workings, or violate privacy through a variety of application problems. Attackers use this weakness to steal sensitive data, or conduct more serious attacks."
  web2py is the only web framework to provide a built-in ticketing system. No error can result in code being exposed to users. All errors are logged and a ticket is issued to the user to allow error tracking. Errors and code are only accessible to the administrator.

- "Broken Authentication and Session Management: Account credentials and session tokens are often not properly protected. Attackers compromise passwords, keys, or authentication tokens to assume other users' identities."
  web2py provides a built-in mechanism for authentication and it manages sessions independently per each application. The administrative interface also forces the use of secure session cookies when client is no localhost.

- "Insecure Cryptographic Storage: Web applications rarely use cryptographic functions properly to protect data and credentials. Attackers use weakly protected data to conduct identity theft and other crimes, such as credit card fraud."
  web2py uses the MD5 or the HMAC+SHA-512 hash algorithms to protect stored passwords. Other algorithms are also available.

- "Insecure Communications: Applications frequently fail to encrypt network traffic when it is necessary to protect sensitive communications." web2py supports SSL at multiple levels, in particular it works with Apache or Lighttpd and mod_ssl to provide strong encryption of communications.

- "Failure to Restrict URL Access: Frequently, an application only protects sensitive functionality by preventing the display of links or URLs to unauthorized users. Attackers can use this weakness to access and perform unauthorized operations by accessing those URLs directly." web2py maps URL requests to Python modules and functions. web2py provides a mechanism for declaring which functions are public and which require authentication/authorization.

1.6 What's in the box

web2py comprises of the following components:

- **libraries**: provide the core functionalities of web2py and are accessible programmatically.

- **web server**: the cherrypy WSGI[2] web server.

- the **admin** application: used to create, design, and manage other web2py applications.

- the **examples** application: contains documentation and interactive examples.

- the **welcome** application: the basic template for any other application. Its main task is to greet the developer when web2py starts.

The binary version of web2py comes packaged with the Python interpreter and the SQLite database. Technically, these two are not components of web2py but they are distributed with it.

[2]WSGI is an emerging standard for communication between a web server and Python applications.

1.7 License

1.8 Thanks

- web2py is programmed in Python created by Guido van Rossum.
- This project uses cherrypy.wsgiserver developed by Peter Hunt and Robert Brewer.
- Some aspects of its design are inspired by Ruby on Rails, Django, Turbogears, Pylons, Cherrypy, and web.py frameworks.
- The compilation of the applications would not have been possible without advice provided by Atul Varma
- We use and include EditArea developed by Christophe Dolivet

- ... TinyMCE developed by Moxiecode
- ... simplejson developed by Bob Ippolito
- ... PyRTF developed by Simon Cusack and revised by Grant Edwards
- ... PyRSS2Gen developed by Dalke Scientific Software
- ... feedparser developed by Mark Pilgrim
- ... markdown2 developed by Trent Mick
- ... fcgi.py devloped by Allan Saddi (for production Lighttpd servers)
- ... memcache developed by Evan Martin
- ... jQuery developed by John Resig
- Thanks to Andre Berthiaume for really stress testing web2py under critical conditions.
- Thanks to Peter Wilkinson for providing a great example of simple syntax highlighting code
- Thanks to Markus Gritsch for many useful comments
- Thanks to Niccolo Polo for help with epydoc and streaming.
- Thanks to Marcel Leuthi for help and testing with Oracle.
- Thanks to Limodou for developing shell.py and winservice.py
- Thanks to Michael Willis for help with shell.py.
- Thanks to Younghyun Jo and Pai for help with internationalization.
- Thanks to Sharriff Aina for his tests and PyAMF howto.
- Thanks to Sterling Hankins his helpful comments
- This project also uses css layouts and images downloaded from the web. Their authors are acknowledged inside the respective files, if known.

I also thank all the users, who have submitted many comments and suggestions for improvement.

1.9 About this Book

web2py is based on Python thus programming web2py requires basic Python knowledge. Specifically web2py requires familiarity with

- conditionals: `if...elif...else`
- loops: `for` and `while`
- function calls: `def...return`
- exceptions: `try...excep...else...finally`
- classes: `class: __init__(self,...): ...`
- basic types: `string`, `int`, `float`, `list`, `tuple`, and `dict`.

Chapter 2 of this book covers this basic Python syntax. Programmers already familiar with Python can skip chapter 2.

Chapter 3 [A first application]
Chapter 4 [API]
Chapter 5 [ORM]
Chapter 6 [Templating language]
Chapter 7 [Recipes]

Chapter 2

The Python Language

You need to read this chapter only if you are new to Python.

2.1 Starting up

The binary distributions of web2py for Windows or OSX come packaged with the python interpreter and you can start it on windows with

```
web2py.exe -S admin
```

and on OSX with

```
web2py.app -S admin
```

On linux chance are you have Python. So at prompt type

```
python
```

2.2 Types

Python is a dynamically typed language. It means variables do not have a type and therefore they do not have to be declared. values do have a type and you can query a variable for the type of value it contains

```
>>> a=3
>>> print type(a)
<type 'int'>
>>> a=3.14
>>> print type(a)
<type 'float'>
```

```
>>> a='hello python'
>>> print type(a)
<type 'str'>
```

Python also comes with some complex data structures like the list and the dictionary.

2.2.1 str

Python has two types of strings: ascii strings delimited by '...', "...", "'..."' or """...""", and unicode strings. Unicode strings start with a u followed by a string containing unicode characters. A unicode string can be converted into an ascii string by choosing an encoding for example

```
>>> a=u'This is a unicode string'
>>> b=a.encode('utf8')
```

Here `b` is an ascii string storing UTF88 encoded characters. To avoid confusion, internally web2py always use UTF8 encoded ascii strings.

From now on we will only use ascii strings.

It is also possible to write variables into strings in various ways

```
>>> print 'number is '+str(3)
number is 3
>>> print 'number is %s' % (3)
number is 3
>>> print 'number is %(number)s' % dict(number=3)
number is 3
```

The last notation is to be preferred becaure more explicit and less error prone.

Many Python objects, for example numbers, can be serialized into strings using `str` or `repr`. They are very similar but produce slightly different output. Here are some examples:

```
>>> for i in [3,'hello']: print str(i),repr(i)
3 3
hello 'hello'
```

For user defined objects `str` and `repr` have to be defiend using special attributes as discussed later. `repr` has always a default.

A string, like a list, is an iterable.

2.2.2 list

A Python list which support append, insert, and delete methods:

```
>>> a=[1,2,3]
>>> print type(a)
<type 'list'>
>>> a.append(8)
>>> a.insert(2,7)
>>> del a[0]
>>> print a
[2,7,3,8]
>>> print len(a)
4
```

as well as slicing

```
>>> print a[:3]
[2,7,3]
>>> print a[1:]
[7,3,8]
>>> print a[-2:]
[3,8]
```

and concatenation

```
>>> a=[2,3]
>>> b=[5,6]
>>> print a+b
[2,3,5,6]
```

A list is iterable and we'll see later how to loop over it. The elements of a list do not have to be of the same type and can be any type of Python object.

2.2.3 tuple

A tuple is like a list but its size and elements are immutable, while in a list they are mutable. If an element is an object, the object attributes are mutable. A tuple is delimited by round brackets

```
>>> a=(2,3)
```

and, like the list, it is iterable.

Tuples are very useful for packing stuff (and the brackets are optional)

```
>>> a=2,3,'hello'
>>> z,y,z=a
>>> print x
2
>>> print z
hello
```

2.2.4 dictionary

A Python dictonary is a hash table. It maps a key into a value.

```
>>> a={'k':'v','k2':3}
>>> a['k']
v
>>> a['k2']
3
>>> a.has_key('k')
True
>>> a.has_key('v')
False
```

Keys can be of any hashable type (int, string, or object implementing the `__hash__` method). Values can be of any type. Different keys and values in the same dictionary do not have to be of the same type. If the keys are alphanumaric characters a dictionary can also be declared with the alternative syntax

```
>>> a=dict(k='v',h2=3)
>>> a['k']
v
>>> print a
{'k':'v','k2',3}
```

Useful methods are

```
>>> a=dict(k='v',k2='3)
>>> print a.keys()
['k','k2']
>>> print a.values()
['v',3]
>>> print a.items()
[('k','v'),('k2',3)]
```

The items method produces a list of tuples each containing a key and the associated value.

2.3 `for` loops

In Python you loop over iterable objects

```
>>> a=[0,1,'hello','python']
>>> for i in a: print i
0
```

```
1
hello
python
```

Two common shortcuts are xrange and enumerate

```
>>> for i in xrange(0,4): print i
0
1
2
4
```

and

```
>>> a=[0,1,'hello','python']
>>> for i,j in enumerate(a): print i,j
0 0
1 1
2 hello
3 python
```

there is also a keyword range(a,b,c) that returns a list of integers ¿=a, ¡b, at increments of c. a defaults to 0 and c defaults to 1. xrange is similar but does not actually generates the list, only an iterator over the list, thus it is usually better.

```
Notice that one can loop over lists of tuples
>>> a=dict(k='v',k2=3)
>>> for i,j in a.items(): print i,j
k v
k2 3
```

2.4 while loops

```
>>> i=0
>>> while i<10: i=i+1
>>> print i
10
```

2.5 Indentation

Python uses indentantion to delimit blocks. Blocks start with a line ending in colon and continues for all lines that have a similar or higher indentation as the next one. For example

```
>>> i=0
>>> while i<3:
>>>     print i
>>>     i=i+1
0
1
2
```

It is common to use 4 spaces for each level of indentation. Do not mix tabs with spaces or you may run into trouble.

2.6 def ... return

Here is a typical Python function

```
>>> def f(a,b=2): return a+b
>>> print f(4)
6
```

Notice that there is no need to specify values of the arguments or the return value(s) and arguments can have default values.

Function can also return multiple objects

```
>>> def f(a,b=2): return a+b,a-b
>>> x,y=f(5)
>>> print x
7
>>> print y
3
```

they can take arguments by name

```
>>> def f(a,b=2): return a+b,a-b
>>> x,y=f(b=5,a=2)
>>> print x
7
>>> print y
-3
```

and they can take an arbitrary number of arguments

```
>>> def f(*a,**b): return a,b
>>> x,y=f(3,'hello',b=4,test='world')
>>> print x
(3,'hello')
>>> print y
{'b':4,'test':'world'}
```

arguments can also be unpacked from a list or tuple

```
>>> def f(a,b): return a+b
>>> c=(1,2)
>>> print f(*a)
3
```

or a dictionary

```
>>> def f(a,b): return a+b
>>> c={'a':1,'b':2)
>>> print f(**a)
3
```

2.7 if ... elif ... else

Here is an example

```
>>> for i in range(3):
>>>     if i==0:
>>>         print 'zero'
>>>     elif i==1:
>>>         print 'one'
>>>     else:
>>>         print 'other'
zero
one
other
```

complex conditions can be created using the `not`, `and` and `or` operators.

2.8 try ... except ... else ... finally

Here is an example

```
>>> try:
>>>     a=1/0
>>> except Exception, e
>>>     print 'error',e,'occurred'
>>> else:
>>>     print 'no problem here'
>>> finally:
>>>     print 'done'
error 3 occurred
done
```

Any object can be raised as an examption but it is good practice to raise object that extend one of the built-in exceptions.

2.9 class

Because Python is dynamically typed, Python classes and objects are a bit odd. In fact one does not need to define its member variables (attributes) when declaring the class. Here is an example:

```
>>> class A(object): pass
>>> myinstance=A()
>>> myinstance.myvariable=3
>>> print myinstance.myvariable
3
```

Notice that `pass` is a do-nothing command. In this case used to define a class `A` that contains nothing. `A()` calls the constructor of the class (in this case the default constructor) and returns an object, instance of the class. The `(object)` in the class definition indicates that out class extends the built-in object. This is not required but it good practice.

Here is a more complex class:

```
>>> class A(object):
>>>     z=2
>>>     def __init__(self,a,b):
>>>         self.x=a,self.y=b
>>>     def add(self):
>>>         return self.x+self.y+self.z
>>> myinstance=A(3,4)
>>> print myinstance.add()
9
```

Functions declared inside the class are methods. Some methods have special reserved names. For example `__init__` is the constuctors. All variables are local variables of the method except variables declared outside methods. For example `z` is *class variable* equivalent to a C++ *static member variable* that holds the same value for all instances of this class.

Notice that `__init__` takes 3 arguments and `add` takes one, and yet we called them with 2 and 0 arguments respectively. The first argument represents, by convention, the local name we want to use inside the method to refer to the current object. Here we use `self` to refer to the current object but we could have used any other name. `self` plays the same role

as `*this` in C++ or `this` in Java but, conversely, `self` is not a reserved keyword.

This syntax is necessary to avoid ambiguiting when declaring nested classes, for example a class that is local of a method inside another class.

In Python we tend to refer to everythins contained in a class and accessed via the . notation as a clas attribute.

2.9.1 Special Attributes

Some important special attributes of classes are the `__len__` attribute and the `__getitem__`, `__setitem_` attributes. This allows for example to create a class that acts as an indexing container

```
>>> class MyList(object)
>>>     def __init__(self,*a): self.a=a
>>>     def __len__(self): return len(self.a)
>>>     def __getitem__(self,i): return self.a[i]
>>>     def __setitem__(self,i,j): self.a[i]=j
>>> b=MyList(3,4,5)
>>> print b[1]
4
>>> a[1]=7
>>> print b.a
[3,7,5]
```

Other special attributes are `__dict__` which is already i any object and contains a dictionary of all the attributes in the object, `__getattr_` and `__setattr_` that allow to get and set attributes of the class `__sum__`, `__sub__`, etc. which allow to overload the aritmetic operators. Anyway for the use of these operators we refer the reader to more advaced books on this topic.

2.10 lambda

There are cases when one may need to dynamically generate an un-named function. This can be done with the `lambda` keyword

```
>>> a=lambda b: b+2
>>> print a(3)
5
```

The expression "`lambda` [a]:[b]" literally reads as "a function that with arguments [a] that returns [b]". Even if the funciton is un-named, it can be

stored into a variable and thus it acquires a name. Technically this is different than using `def` becacuse its is the variable that refers to the function that has a name, not the function itself.

Who needs lambdas? Actually they are very usefule and here is an example. Suppose you have a function `factorize` that finds the prime factors of its argument. This function is time consuming. Also suppose you have a caching function `cache.ram` that takes three arguments, a key, a function and a number of seconds. The first time it is called it calls the function (its second argument), stores the output in a dictionary in memory and returns it. The second time it is called, if the key is in the key is in the dictionary and it is not older than the number of seconds specified, it returns the coresponding value without performing the function call. Who would we cache the output of the `factorize` function for any input?

```
>>> number=3*5*7
>>> seconds=60
>>> print cache.ram(str(number), lambda: factorize(number),
    seconds)
[3,5,7]
>>> print cache.ram(str(number), lambda: factorize(number),
    seconds)
[3,5,7]
```

The output is always the same but the first time `factorize` is called, the second time it is not. The existance of `lambda` allows to create functions (in this case `cache.ram`) that can take any other function as argument and evaluate it lazily. This is exactely how caching in web2py is implemented.

2.11 exec and eval

Python is truly an intepreted language. This means it has the ability to execute Python statements stored in a string. For example

```
>>> a="print 'hello world'"
>>> exec(a)
'hello world'
```

What just happened? The function `exec` tells the interpreter to call itself an interpter/execute the content the string passed as argument. It is also possible to execute the content of a string within a context defined by the symbols in a dictionary

```
>>> a="print b"
>>> c=dict(b=3)
>>> exec(a,{},c)
3
```

Here the interpreter, when executing the string `a` sees the symbols defiend in `c`, i.e. `b` but does not see `c` nor `a` themselves. Notice hat this is different than a restricted environment since `exec` does not limit what the inner code can do, it just defines the set of variables visible to the code.

A related function is `eval`. Eval works very much like `exec` except that it expects the argument string to produce an output value and it returns that value.

```
>>> a="3*4"
>>> b=eval(a)
>>> print b
12
```

2.12 `import` modules

The real power of Python is in its modues which provide a large and consistent set of Application Programming Interface (API) to many system libraries (often in a OS independet way. For example if we need to use a random number generator

```
>>> import random
>>> print random.randint(0,9)
```

will print a random integer between 0 and 9 (including 9). The function `randint` is defined in the module `random`. It is also possible to import an object from a module in the current namespace

```
>>> from random import randint
>>> print randint(0,9)
```

or import all objects from a module in the current namespace

```
>>> from random import *
>>> print randint(0,9)
```

or import everything in a newly defined namespace

```
>>> import random as myrand
>>> print myrand.randint(0,9)
```

In the rest of this book we will only need a few objects defined in modules `os`, `sys`, `datetime`, `time` and we will refer to the module `cPickle`. All of the web2py objects are accessible via a module called `gluon` and that will be the subject of the later chapters.

2.12.1 os

This module provides an interface to the `os` API. For example

```
>>> import os
>>> os.chdir('..')
>>> os.unlink('filename_to_be_deleted')
```

Some of the `os` functions, for example `chdir`, should not be used in web2py because they are not thread safe.

The `os` function we will need the most is

```
>>> import os
>>> a=os.path.join('path','sub_path')
>>> print a
path/sub_path
```

`os.path.join` can join paths in a OS specific way.

2.12.2 sys

[FILL HERE]

2.12.3 datetime

[FILL HERE]

2.12.4 time

[FILL HERE]

2.12.5 cPickle

[FILL HERE]

2.13 Installation and Admin

web2py comes with binary packages for Windows and OSX. There is also a source code package version which runs on Windows, OSX, Linux, and other Unix Systems. The Windows and OSX binary versions include the necessary Python interpreter. The source code package assumes Python 2.5 or greater is already installed on the developer's computer.

web2py requires no installation. To get started, unzip the downloaded zip file for your specific operating system and execute the corresponding `web2py` file.

On **Windows** you can start by clicking on

```
web2py.exe
```

On OSX you can start by clicking on

```
wep2py.app
```

On Unix and Linux you need to run from source. In order to run from source you type:

```
python2.5 web2py.py
```

Both the Python source and the binary version of `webtwopy.py` accept various command line options. You can get a list by typing

```
python2.5 web2py.py -h
```

Here is a more or less complete list:

```
--version             show program's version number and exit

-h, --help            show this help message and exit

-i IP, --ip=IP        the ip address of the server (127.0.0.1)

-p PORT, --port=PORT  the port for of server (8000)

-a PASSWORD, --password=PASSWORD
                      the password to be used for administration
                         (use -a
                      '<recycle>' to reuse the last password)

-u UPGRADE, --upgrade=UPGRADE
```

```
                      upgrade applications

-c SSL_CERTIFICATE, --ssl_certificate=SSL_CERTIFICATE
                      file that contains ssl certificate

-k SSL_PRIVATE_KEY, --ssl_private_key=SSL_PRIVATE_KEY
                      file that contains ssl private key

-d PID_FILENAME, --pid_filename=PID_FILENAME
                      file where to store the pid of the server

-l LOG_FILENAME, --log_filename=LOG_FILENAME
                      file where to log connections

-n NUMTHREADS, --numthreads=NUMTHREADS
                      number of threads

-s SERVER_NAME, --server_name=SERVER_NAME
                      the server name for the web server

-q REQUEST_QUEUE_SIZE, --request_queue_size=REQUEST_QUEUE_SIZE
                      max number of queued requests when server
                         unavailable

-o TIMEOUT, --timeout=TIMEOUT
                      timeout for individual request

-z SHUTDOWN_TIMEOUT, --shutdown_timeout=SHUTDOWN_TIMEOUT
                      timeout on shutdown of server

-f FOLDER, --folder=FOLDER
                      the folder where to run web2py

-S APPNAME, --shell=APPNAME
                      run web2py in interactive shell or IPython
                         (if
                      installed) with specified appname

-P, --plain           only use plain python shell, should be
   used with
                    --shell option

-M, --import_models   auto import model files, default is False,
    should be
                      used with --shell option
```

```
55
56 -R PYTHON_FILE, --run=PYTHON_FILE
57                       run PYTHON_FILE in web2py environment,
                             should be used
58                       with --shell option
59
60 -W WINSERVICE, --winservice=WINSERVICE
61                       -W install|start|stop as windows service
62
63 -L CONFIG, --config=CONFIG
64                       Config file
```

As a general rule the lower case options are used to configure the web server. The -L option tells web2py to read configurtion options from a file, -W installs web2py as a windows service and -S/-P/-M are used to run an interactive Python shell. These are all advanced features that we will discuss later.

At startup web2py goes through a brief presentation

and then shows a small window where it asks the user to chose the administrator password and the IP address of the network interface to be used for the web server and a port number from which to serve requests. By default, web2py runs the web server on 127.0.0.1:8000 but, you can run it on any avaliable IP address and port. You can query the IP address of your ethernet network interface by opening a command line and typing `ipconfig` on Windows or `ifconfig` on OSX and Linux. From now on we will assume web2py is running on localhost (127.0.0.1:8000).

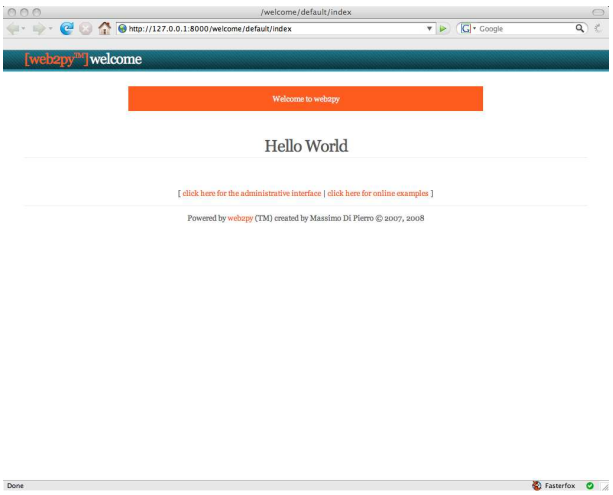
If the user does not provide an administrator password the administration interface will be disabled. This is a security measure to prevent publicly exposing the admin interface.

The administration interface is only accessible from localhost unless web2py runs behind Apache with mod_proxy.

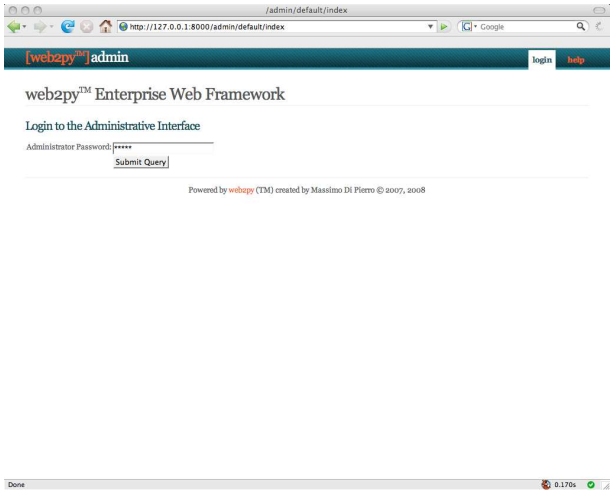
After the administration password has been set web2py starts up the developer's web browser and navigates to

```
1 http://127.0.0.1:8000/welcome/default/index
```

if the computer does not have a default browser setting then the developer will have to open their preferred web browser and navigate to the aforementioned URL.

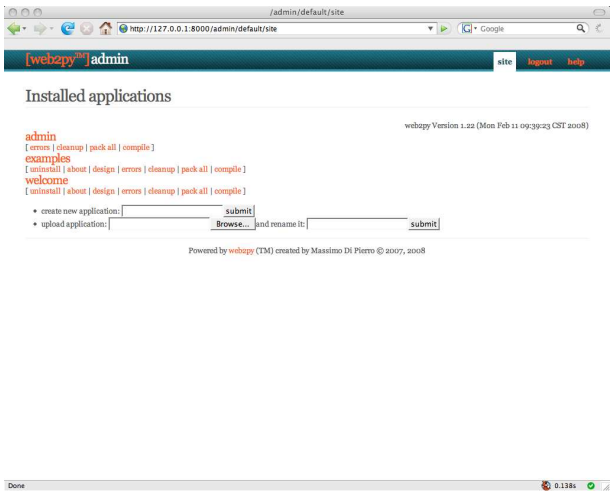


By clicking on "administrative interface" we are redirected to the login page for the administration interface.



The login password chosen when web2py was started is the administrator password. Notice there is only one administrator and therefore only one administrator password. For security reasons the developer is asked to choose a new password every time web2py starts. This has nothing to do with the authentication mechanism of web2py applications which will be discussed in a separate chapter.

After the administrator logs into web2py, the browser is redirected to the "site" page.



This page lists all installed web2py applications and allows the administrator to manage them. web2py comes with three applications:

- An **admin** application, the one we are using right now.
- An **examples** application, the online interactive documentation and a replica of the web2py official web page.
- A **welcome** application. This is the basic template for any other web2py application. It is also the application that welcomes a user at startup.

Ready to use web2py applications are referred to as web2py appliances and one can download some freely available ones from

```
1 http://mdp.cti.depaul.edu/appliances/
```

web2py users are encouraged to submit new appliances, either in open source or closed source form.

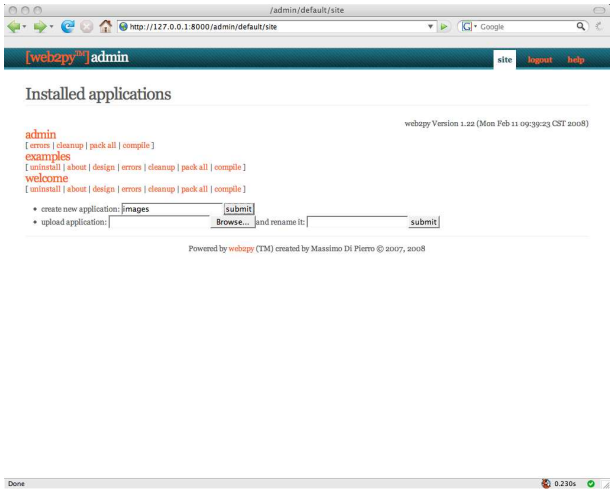
From the **site** page we can perform the following operations:

- **install** an application by completing the form at the bottom of the page (give a name to the application, select the file containing a packaged application, and click "submit").
- **uninstall** an application by clicking the corresponding button (there is a confirmation page)
- **create** a new application by choosing a name and clicking "submit" (more details will be given below).
- **package** an application for distribution by clicking on the corresponding button (the application will be downloaded as a tar file containing everything, including the database).
- **clean up** temporary files such as sessions and errors files.
- **compile** the application and **package the compiled app** (a compiled application runs faster than a non-compiled one and it can be distributed in closed source form).
- **design** an application (this will be discussed in great detail in the rest of this book).

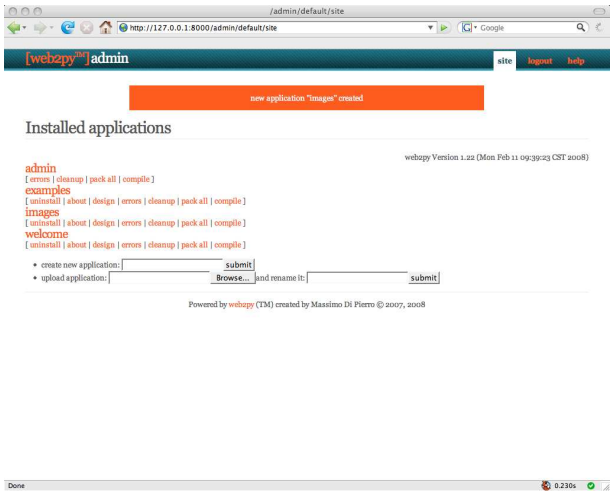
2.14 Creating a New Application

Here we wish to create a web site that allows the administrator to post images and give them a name, and allow the visitors of the web site to see the images and submit comments.

It seems natural to call this new application "images", hence, we proceed by typing the application name in the form at the bottom and press "submit".



The new application has been created. That was easy!

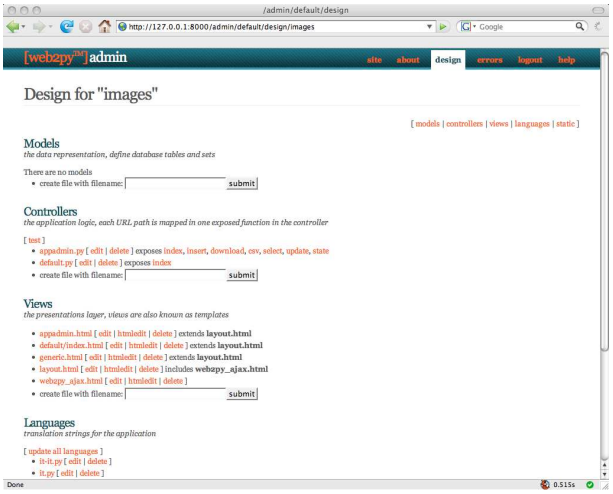


To run the new application just visit

```
1 http://127.0.0.1:8000/images
```

At this point we have not yet designed the application therefore all we have is a copy of the welcome application.

To design the application, we need to click on **design** from the **site** page.



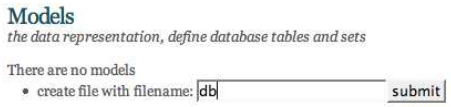
The **design** page tells us what is inside the application. Every web2py application consists of certain files, most of which follow in one of the five categories:

- **models**: describe the data representation
- **controllers**: describe the application logic and workflow
- **views**: describe the data presentation
- **languages**: describe how to translate the application presentation to other languages
- **static files**: static images, css files, javascript files, etc.

Technically, an application contains other files (database, session files, errors files, etc.) but they are not listed on the design page because they are not created or modified by the administrator. They are created and modified by the application itself.

2.15 Models

We will start by creating a model, a representation of the persistent data in our application (the images we want to upload, their names, and the comments). First, we need to create a model file which, for lack of imagination, we will call `db.py`. Models and controllers must have a .py extension since they contain Python code. If the extension is not provided, it is appended by web2py. Views instead have a `.html` extension since they mainly contain HTML code.



After we create the file we click on the corresponding "edit" button and type the following code:

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

db.define_table('image',
   SQLField('title'),
   SQLField('file','upload'))

db.define_table('comment',
   SQLField('image_id',db.image),
   SQLField('author'),
   SQLField('email'),
   SQLField('body','blob'))

db.image.title.requires=[IS_NOT_EMPTY(),\
                         IS_NOT_IN_DB(db,db.image.title)]
db.comment.image_id.requires=IS_IN_DB(db,db.image.id,\
                                      '%(title)s')
db.comment.author.requires=IS_NOT_EMPTY()
db.comment.email.requires=IS_EMAIL()
db.comment.body.requires=IS_NOT_EMPTY()
```

Let's analyze this line by line:

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

It defines a global variable called `db` set to a newly created database connection (represented by the SQLDB object) to a **sqlite** database stored in the

file **db.db**. If the database does not exist it is created. The file is located in the application `databases` folder.

The developer can change the name of the file as well as the name of the global variable `db`, but it is convenient to give them the same name so it is easy to remember.

The rest of the code defines (and creates) two tables, links them to each other, and sets constrains on the values of their fields. For example in

```
db.define_table('image',
   SQLField('title'),
   SQLField('file','upload'))
```

`define_table` is a method of our **db** object. The first argument "image" is the name of the table we are defining. The other arguments are the fields belonging to that table. This table has a field called "title", a field called "file" and a field called "id" that serves as the table primary key (it is defined even if it is not explicitly named). The field "title" is a variable size string (the default), and the field "file" is of type "upload". "upload" is a special type of field of the web2py ORM used to store a file. web2py knows how to store uploaded files via a mechanism that is explained later.

As soon as a table is defined web2py takes one of several possible actions:

- if the table does not exist the table is created
- if the table exists and corresponds to the definition, it proceeds but...
- ...if the table exists and does not correspond to the definition, the table is altered accordingly, and if a field has a different type, web2py tries to convert the values.

This beahviour is called "migrations".

Consider now the second table:

```
db.define_table('comment',
   SQLField('image_id',db.image),
   SQLField('author'),
   SQLField('email'),
   SQLField('body','blob'))
```

This code defines (and creates) another table called "comment". It contains an "author", an "email" (we intend to store the email of the author of the comment), a "body" which is a type "blob" (we intend to use it to store the

actual comment posted by author), and an "image_id" field which is of type reference to a **db.image**.

Notice that since we have created table "image", **db.image** now refers to that table. When we use a table as field type, we mean that the field contains a reference to the **id** field of the referenced table.

```
db.image.title.requires=[IS_NOT_EMPTY(),\
                         IS_NOT_IN_DB(db,db.image.title)]
```

db.image.title is the field "title" of table "image". The attribute `requires` allows us to set requirement constraints. Here we require that the "title" is not empty (`IS_NOT_EMPTY()`) and that it is unique (`IS_NOT_IN_DB(db, db.image.title)`). These validators are applied when processing input forms and are also used to notify the users when fields are not validated. `IS_NOT_IN_DB(a,b)` is a special validator that checks that the value of a field `b` for a new record is not already in the database (or a set) `a`.

```
db.comment.image_id.requires=IS_IN_DB(db,db.image.id,\
                                      '%(title)s')
```

Here we require that the field "image_id" of table "comment" is in **db.image.id**. Didn't we declare this already when we declared `db.comment.image_id` a reference to `db.image.id`? Yes, as was as the database was concerned, but now we are explicitly telling the model that this condition has to be enforced at the FORM processing level, when a new comment is posted. We also requiring that the "image_id" should be represented by the "title", `'%(title)s'`, of the corresponding record.

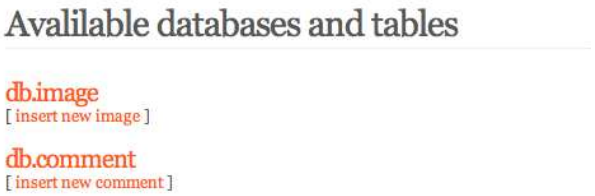
```
db.comment.author.requires=IS_NOT_EMPTY()
db.comment.email.requires=IS_EMAIL()
db.comment.body.requires=IS_NOT_EMPTY()
```

The meaning of these other validators is obvious and does not require explanation.

Once a model is defined, if there are no errors, web2py creates an application administration interface to manage the database. This can be accessed via the "database administration" in the "design" page or directly:

```
http://127.0.0.1:8000/images/appadmin
```

Here is a screenshot of the **appadmin** interface:



This interface is coded in the provided controller called **appadmin.py** and the corresponding view **appadmin.html**. This interface, referred to from now on simply as **appadmin** allows the administrator to insert new database records, edit and delete existing records, browse tables, and perform database joins.

The first time the application administration interface is accessed, the model is executed and the tables are created. The web2py ORM translates Python code into SQL statements that are specific to the selected database backend (SQLite in this example). The developer can see the generated SQL using the design page by clicking on the "sql.log" link under "models". Notice that the link is not there if tables have not yet been created.

If we edit the Model and access appadmin again, web2py generates SQL to alter the existing tables. This process is logged into **sql.log**.

Let's go back to appadmin and try to insert a new image record. Go back to "database administration" and click "insert new image".

web2py has translated the **db.image.file** "upload" field into an upload form for the file. When the form is submitted and an image file is uploaded, the file is renamed in a secure way that preserves the extension, it is saved with the new name under the application **uploads** folder, and a the new name is stored in the **db.image.file** field. This process is designed to prevent directory traversal attacks. The same mechanism can easily be used in user defined controllers.

When the administrator clicks on the table name in appadmin, web2py performs a select of all records on the current table (identified by the query SQL **image.id** $\rangle$ 0) and renders the result.

One can select a different set of records by editing the SQL query and pressing "apply".

Notice that by pressing "apply" the developer turns the query into a more sophisticated "select" page that allows the developer to update and delete those selected recods.

To edit or delete a single record, click on the record id number.
Try to insert a record in the "comment" table.

Notice that because of the `IS_IN_DB` validator, the reference field "image_id" is rendered by a dropdown menu. The items in the dropdown are identified by a key (**db.image.id**) but are represented by the **db.image.title**, as specified by the validator.

Validators are very powerful objects that know how to represent fields, filter field values, generate errors, and format values extracted from the field.

2.16 Controllers and Views

So far our application knows how to store data and we know how to access the database via appadmin. Nevertheless, access to appadmin is restricted to the administrator and it is not intended as a production web interface for the application; hence the next part of this walkthrough. Specifically we want to create:

- an "index" page that allows us to list all available images sorted by title and links to detail pages for the images.
- a "view/[id]" page that shows the visitor the image with id==[id] and allows the visitor to view and post comments.

For this purpose we need two controller functions that we will call **index** and **view**.

Let's write the former by editing the **default.py** controller. From appadmin, we can "edit" **default.py**

```
def index():
    images=db().select(db.image.ALL,orderby=db.image.title)
    return dict(images=images)
```

A controller function (also referred to occasionally as an "action") is a function that takes no arguments (variables are passed via a request object) and returns a string (the body of the corresponding page) or a dictionary. The keys of the items in the dictionary are interpreted as variables passed to the view associated to the action. If there is no view, the action is rendered by the "generic.html" view that is provided with every web2py application.

The index controller performs a select of all fields from table **image** (`db.image.ALL`) ordered by **db.image.title**. The result of the select is a **SQLRows** object (which is an iterable object) containing the records. We assign it to a local variable called `images` that we return to the view.

If we do not write a view, the dictionary is rendered by `views/generic.html`.

We choose to write our own view for the index action. Return to appadmin and "edit" **default/index.html**:

```
{{extend 'layout.html'}}

<h1>Current Images</h1>
<ul>
{{for image in images:}}
{{=LI(A(image.title,_href=URL(r=request,f="view",args=[image.id
    ])))}}
{{pass}}
</ul>
```

which loops over the records

```
{{for image in images:}}
    ....
{{pass}}
```

and for each image record displays

```
LI(A(image.title,_href=URL(r=request,f='view',args=[image.id]))
```

This is a `<li>...<li>` tag that contans a `<a href="...">...</a>` tag that contans the `image.title`. The value of the hypertext reference (href attribute) is

```
URL(r=request,f='view',args=[image.id])
```

i.e. the URL within the same application and controller as the current request, calling the function called "view" and passing a single argument, **im-**

age.id. Notice that `LI`, `A`, etc. are web2py helpers that map to those corresponding HTML tags. Their unnamed arguments are interpreted as objects to be inserted inside the tag. Named arguments starting with a _ (underscore) (for example `_href`) are interpreted as tag attribute values without the _. Named arguments not starting with a _ have a special meaning for some tags. Notice now the underscore notation solves the problem of specifying a "class" tag ("class" is a reserved keywork in Python but "_class" is not).

As an example, the following statement

```
{{=LI(A('something',_href=URL(r=request,f='view',args=[123])))}}
```

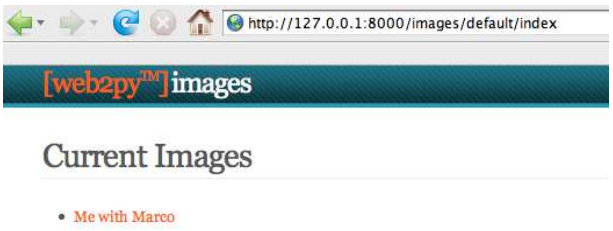
is rendered as

```
<li><a href="/images/default/view/123">something</a></li>
```

If we go back to the "design" interface we see that "default.py exposes index". By clicking on "index" we visit the newly created page:

```
http://127.0.0.1:8000/images/default/index
```

which appears as



If we click on the image name link, we get directed to

```
http://127.0.0.1:8000/images/default/view/1
```

and this results in an error since have not yet created an action called "view" in controller default.py.

Let's edit the default.py controller again:

```
def index():
    images=db().select(db.image.ALL,orderby=db.image.title)
    return dict(images=images)


def view():
    image=db(db.image.id==request.args[0]).select()[0]
```

```
    form=SQLFORM(db.comment,fields=['author','email','body'])
    form.vars.image_id=image.id
    if form.accepts(request.vars,session):
        response.flash='your comment is posted'
    comments=db(db.comment.image_id==image.id).select()
    return dict(image=image,comments=comments,form=form)


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

We have now added two actions: "view" and "download". The "view" action selects the image with the `id` parsed from the request args and all comments related to the image. "View" then passes everything to the view template "default/view.html". The "download" action is a standard action used to retrieve an uploaded file. We copied it from **appadmin.py**.

Notice the following statements:

```
form=SQLFORM(db.comment,fields=['author','email','body'])
```

It creates an entry form for the **db.comment** table using only the specified fields.

```
form.vars.image_id=image.id
```

sets the value for the reference field which is not part of the input form because it is not in the list of fields specified above.

```
if form.accepts(request.vars,session):
```

Process the submitted form (submitted form variables are stored in `request.vars`) within the current session (the session is used to assign a unique key to each form to prevent double submissions and enforce navigation). If the submitted form variables are validated the new comment is inserted in the **db.comment** table, otherwise the form is modified to include error messages (for example, if the author email address is invalid).

We now need to create a view for the "view" action. Return to `appadmin` and create a new view called "view":

```
{{extend 'layout.html'}}
<h1>Image: {{=image.title}}</h1>
```

```
<center>
<img width="200px"
     src="{{=URL(r=request,f='download',args=[image.file])}}"/>
</center>
{{if len(comments):}}
  <h2>Comments</h2><br /><p>
  {{for comment in comments:}}
    {{=comment.author}} says <i>{{=comment.body}}</i>
  {{pass}}</p>
{{pass}}
<h2>Post a comment</h2>
{{=form}}
```

This view displays the **image.file** by calling the "download" action. If there are comments it will loop over them and display each one.

Here is how everything will appear to a visitor.

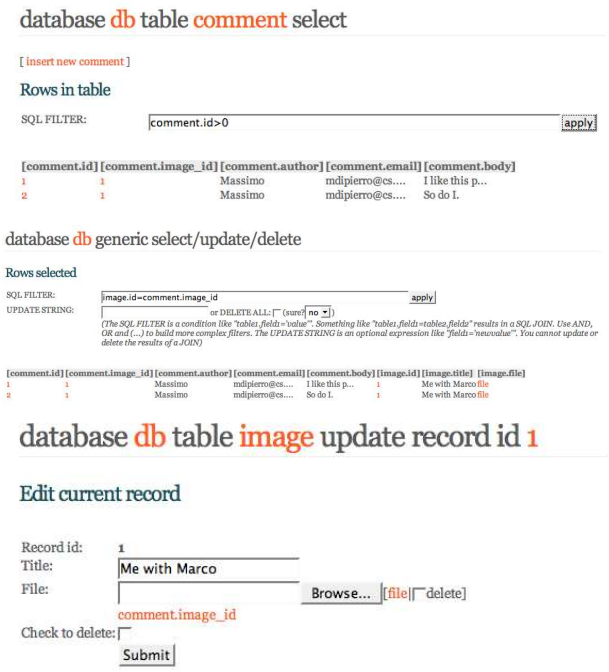
When a visitor submits an invalid form the visitor will see error messages like the following:

2.17 appadmin

One aspect that we have not discussed is the ability of appadmin to perform joins. In fact, if the SQL FILTER contains a condition that involves two or

more tables, for example in

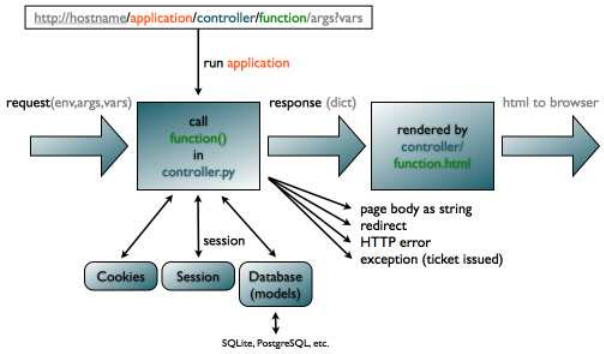
```
1 image.id=comment.image_id
```



Chapter 3

A Deeper Look

3.1 Workflow



web2py maps a URL in the form

```
1 http://127.0.0.1:8000/a/c/f
```

into a call to function `f()` in controller `c.py` in application `a`. If `f` is not present web2py defaults to `index()`. If `c` is not present web2py defaults to `default.py` and if `a` is not present web2py defaults to the `init` application. If there is no `init` app, web2py tries to run the `examples` app.

By default any new request also creates a new session and a session cookie is returned to the client browser to keep track of this session.

A controller canot be named "static" because a URL like this

```
1 http://127.0.0.1:8000/a/static/filename
```

has a special meaning. It returns the static file `filename` from application `a`. When static files are downloaded, web2py does not create a session nor does it issue a cookie. web2py provides a mechanism to override this behavior which is discussed later. web2py supports the HTTP_IF_MODIFIED_SINCE protocol and only serves static files if they have been modified since last time they were requisted by client.

Also controller functions that take arguments or start with a double underscore are not publicly exposed and can only be called by other controller functions.

web2py maps GET/POST requests for the the form

```
http://127.0.0.1:8000/a/c/f/x/y/z?p=1&q=2
```

into a call to function `f` in controller `c.py` into application `a` and it stores the various parameters in the `request` variable as follows:

```
request.args=['x', 'y', 'z']
```

and

```
request.vars={'p':1, 'q':2}
```

and

```
request.application='a'
request.controller='c'
request.function='f'
```

web2py stores environment variables in `request.env`, for example:

```
request.env.path_info='a/c/f'
```

and HTTP headers into environment variables, for example:

```
request.env.http_host='127.0.0.1:8000'
```

Notice that if the URL is not a valid expression, web2py returns a HTTP 400 error message.

If the URL corresponds to a request for a static file, web2py simply reads and returns (streams) the requested file. Notice that web2py validates the URL to prevent directory traversal attacks.

If the URL corresponds to a request for a function call, web2py processes the request in the following order:

- parses cookies
- creates an environment (E) in which to execute the function

- initializes `request`, `response`, `cache`
- reopens an existing `session` or creates a new one
- executes the models in the requested application
- executes the requested controller function
- if the function returns a dictionary, executes the associated view
- on success, commits all open transations
- saves the session
- returns the HTTP response

(Notice that the controller and the view are executed in differed copies of the same environment E, therefore the view does not see the controllers, but it sees the models and it sees the variables returned by the controller functions.)

If an Exception (other than HTTP) is raised web2py does the following:

- stores the traceback in an error file and assigns a ticket number to it
- rolls back all open transactions
- returns an error page reporting the ticket number.

If the Exception is an HTTP Exception this is assumed to be intended behavior (e.g. an HTTP redirect), the open transactions are committed, and the behavior after that is specified by the HTTP exception itself. The HTTP exception class is not a standard Python exception but is defined by web2py.

3.2 Libraries

web2py libraries provide the following functionalities:

- map URLs into function calls
- handle passing and returning parameters via HTTP
- perform validation of those parameters

- protect the applications from most security issues
- handle data persistance (database, session, cache, cookies)
- perform string translations for various supported languages
- generate HTML programmatically (e.g. from database tables)
- generate SQL and add a powerful Python abstraction layer above the specified database (SQLite, MySQL, PostgreSQL, or Oracle). This abstraction layer will be referred to as Object Relational Mapper (ORM)
- generate Rich Text Format RTF output
- generate Comma Separated Value (CSV) output from database tables
- generate Really Simple Syndication (RSS) feeds
- generate Javascript Object Notation (JSON) serialization strings for AJAX
- translate WIKI markup (markdown) to HTML
- expose XMLRPC web services
- upload and download large files via streaming.

Most of the web2py libraries are exposed to user applications as objects (`request`, `response`, `session`, `cache`, and `T`), classes (`exception`, `helpers`, and `validators`), or functions (`redirect`).

3.3 Applications

Applications developed in web2py are comprised of the following parts:

- **models** describe representation of the data as database tables and relations among those tables.
- **controllers** describe the application logic and workflow.
- **views** describe how data should be presented to the user using HTML and javascript.

- **languages** describe how to translate strings in the application into various supported languages.
- **static files** do not require processing like images, css stylesheets, etc.
- **ABOUT** and **README** documents are self explanatory.
- **errors** store error reports generated from the application unexpectedly.
- **sessions** store information related to each particular user.
- **database** stores sqlite databases and additional table information.
- **cache** store cached items at the application scope.
- **modules** are other optional Python modules.
- **private** files are accessed by the controllers but not directly by the developer.
- **uploads** files are accessed by the models but not directly by the developer (e.g. files uploaded by users of the application).

Models, views, controllers, languages, and static files are accessible via the administration [Design] interface. ABOUT, README, and errors are also accessible via the administration interface through the corresponding menu items. Sessions, cache, modules and private files are accessible to the applications but not via the administration interface.

Everything is neatly organized in a clear directory structure that is replicated for every installed web2py application, althought the user never needs to access the filesystem directly:

```
1 ABOUT      controllers   languages   private     uploads
2 LICENSE    databases     models      sessions    views
3 cache      errors        modules     static      __init__.py
```

`__init__.py` is an empty file which is required in order to allow Python (and web2py) to access the modules in `modules`.

3.3.1 Howto Import Third Party Modules

Every application folder has a modules subfolder. You can place python modules in there. You can access them from `models/views/controllers` with

```
import applications.[youapp].modules.[yourmodule] as [yourmodule
    ]
```

3.4 Complete list of exposed API

Global Objects

```
request, response, session, cache
```

Navigation Functions

```
redirect, HTTP
```

Internationalization

```
T
```

HTML Helpers

```
XML, URL, BEAUTIFY

A, B, BODY, BR, CENTER, CODE, DIV, EM, EMBED, FIELDSET, FORM,
H1, H2, H3, H4, H5, H6, HEAD, HR, HTML, IFRAME, IMG, INPUT,
LABEL, LI, LINK, OL, UL, META, OBJECT, ON, OPTION, P, PRE,
SCRIPT, SELECT, SPAN, STYLE,  TABLE,  TD, TAG, TBODY,
TEXTAREA, TFOOT, TH, THEAD, TITLE, TR, TT
```

Validators

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

Database API

```
SQLDB, SQLField
```

3.5 Global Objects

3.5.1 request

The `request` object is a subclass of web2py class `Storage`, of which extends the Python class `dict`. It is basically a dictionary, but its elements can also be used as attributes:

```
request.vars
```

is the same as

```
request['vars']
```

If an attribute (or key) is not in the dictionary, it does not raise an exception but instead returns `None`.

`request` has the following elements/attributes, some of which which are also a subclass of the `Storage` object:

- **request.cookies**: a `Cookie.SimpleCookie()` object containing the cookies passed with the HTTP request. It acts like a dictionary of cookies. Each cookies is a Morsel object.
- **request.env**: a `Storage` object containing the environment variables passed to the controller, including HTTP header variables from the HTTP request and standard WSGI parameters.
- **request.application**: the name of the requested application (parsed from `request.env.path_info`).
- **request.controller**: the name of the requested controller (parsed from the `request.env.path_info`).
- **request.function**: the name of the requested function (parsed from the `request.env.path_info`)
- **request.folder**: the application folder
- **request.args**: `=request.env.path_info.split('/')[3:]`
- **request.vars**: a `Storage` object containing the HTTP GET and HTTP POST variables.

- **request.get_vars**: a `Storage` object containing only the HTTP GET variables.

- **request.post_vars**: a `Storage` object containing only the HTTP POST variables.

As an example the following call on my system

```
1 http://127.0.0.1:8000/examples/default/status/x/y/z?p=1&q=2
```

will show the following values:

variable	value
request.application	examples
request.controller	default
request.view	status
request.folder	applications/examples/
request.args	['x','y','z']
request.vars	⟨ Storage {'p': 1, 'q': 2}⟩
request.get_vars	⟨ Storage {'p': 1, 'q': 2}⟩
request.post_vars	⟨ Storage {}⟩
request.env.content_length	0
request.env.content_type	
request.env.http_accept	text/xml,text/html;
request.env.http_accept_encoding	gzip, deflate
request.env.http_accept_language	en
request.env.http_cookie	session_id_examples=127.0.0.1.119725
request.env.http_host	127.0.0.1:8000
request.env.http_max_forwards	10
request.env.http_referer	http://mdp.cti.depaul.edu/
request.env.http_user_agent	Mozilla/5.0
request.env.http_via	1.1 mdp.cti.depaul.edu
request.env.http_x_forwarded_for	76.224.34.5
request.env.http_x_forwarded_host	mdp.cti.depaul.edu
request.env.http_x_forwarded_server	127.0.0.1
request.env.path_info	/examples/simple_examples/status
request.env.query_string	remote_addr:127.0.0.1
request.env.request_method	GET
request.env.script_name	
request.env.server_name	127.0.0.1
request.env.server_port	8000
request.env.server_protocol	HTTP/1.1
request.env.wsgi_errors	⟨ open file '⟨ stderr⟩ ', mode 'w' at ⟩
request.env.wsgi_input	
request.env.wsgi_multiprocess	False
request.env.wsgi_multithread	True
request.env.wsgi_run_once	False
request.env.wsgi_url_scheme	http
request.env.wsgi_version	10

3.5.2 response

`response` is another a `Storage` object automatically created by web2py and it contains the following:

- **response.write(text)**: a method to write text into the output page body
- **response.headers**: a `dict` into which to write the headers to be passed to the response.
- **response.status**: the HTTP integer status code to be passed to the response. Default is 200 (OK).
- **response.body**: a `StringIO` object into which web2py writes the output page body.
- **response.view**: the name of the view that will render the page. This is set by default to to

```
1 "%s/%s.html" % (request.controller, request.function)
```

- **response.render(vars)**: a method used to call the view explicitly inside the controller. It should be used for caching purpose only (see cache examples).
- **response.xmlrpc(request,methods)**: when a controller returns it, this function exposes the passed methods via XML-RPC.
- **response.stream(file,chunk_size)**: when a controller returns it,web2py streams the file content back to the client in blocks of size `chunk_size`.
- **response.session_id**: the id of the current session. It is determined automatically. If the developer does not wish to save the session, set this to `None`.
- **response.session_id_name**: the name of the session cookie for this application.
- **response.flash**: optional parameters that may or may not be interpreted by the views. Normally used to notify the user about something that happened.

- **response.keywords**: optional parameters that may or may not be interpreted by the views. Normally used to set the meta keywords in the HTML header.
- **response.description**: optional parameters that may or may not be interpreted by the views. Normally used to set the meta decription in the HTML header.
- **response.menu**: optional parameters that may or may not be interpreted by the views. Normally used to pass a tree like representation of a navigation menu to the view.
- **response.cookies**: similar to **request.cookies** but whil the latter contains the cookies sent from the client to the server, the former contains cookies being sent by the server to the client. The session cookie is handled automatically.

File Streaming

Since version 1.22, streaming is the default method used by web2py to serve files from the following directories:

```
[web2py working folder]/application/[your app]/static/
[web2py working folder]/application/[your app]/uploads/
```

For customised file serving function that overrides the default download behavior, web2py will automatically detect whether the returning object is a streaming object (an iterator), or a string object, and serve accordingly.

```
Examples:
    def my_big_file_downloader():
        ...
        ...
        import os
        filename=request.args[0]
        pathfilename=os.path.join(request.folder,'uploads/',
            filename)
        return response.stream(open(pathfilename,'rb'),
            chunk_size=10**6)
```

You call them with:

```
http://[host]/[app]/[controller]/my_big_file_downloader/[
   filename in uploads]
```

or

```
http://[host]/[app]/[controller]/my_big_file_downloader/[
   filename in uploads]/[...]
```

where [...] is the filename you want the downloading user to see.

Streaming a Virtual File

Here is how you jam various types of invalid requests. Requires v1.21 (only available at code.google.com//web2py). Will be officially released within the week.

in applications/myapp/controllers/default.py

```
class Jammer():
   def read(self,n):
      return 'x'*n
def jammer(): return response.stream(Jammer(),40000)
```

3.5.3 session

`session` is an empty `Storage` object. Whatever is stored into `session` for example

```
session.myvariable="hello"
```

can be retrieved at a later time

```
a=session.myvariable
```

if the code is executed within the same session (by the same user if the user has not deleted session cookies and if the session did not expire). Because `session` is a `Storage` object, trying to access an attribute/key that is not set does not raise an exception but returns `None` instead.

Recipe: Secure Session Cookies

To set a secure session cookie use, somewhere in your controller.

```
response.cookies[response.session_id_name]['secure']=True
```

This is done automatically in the admin interface is the client is not no localhost. If an application uses a secure session cookies it has to run over HTTP otherwise the mechanism will break.

3.5.4 cache

The `cache` object has two attributes:

- **cache.ram**: the application cache in main memory.
- **cache.disk**: the application cache on disk.

`cache` is callable, mainly for use in a decorator, in order to cache controller functions and views (see the section on cache).

3.6 Classes

3.6.1 T and Internationalization

This is the language translator. All constant strings (and only constant strings) should be marked by `T` as in:

```
1 a=T("hello world")
```

Strings that are marked with `T` are identified by web2py as needing language translation (see internationalization examples) and they will be translated when the code (in the model, controller, or view) is executed.

The `T` object can also contain variable fields, for example

```
1 a=T("hello %(name)s",dict(name="Massimo"))
```

the first string will be translated according to the requested language file and and name will be replaced independently on the language.

Concatenating translation strings is nota good idea, in fact that is why web2py does not even allow you to do

```
1 T("bla")+T("bla")
```

but it does allow

```
1 T("bla %(name)s",dict(name='Tim'))
```

The requested language is determined by the "Accept-Language" field in the HTTP header but this selection can be overwritten programmatically by requesting a specific file,

```
T.force('it-it')
```

which reads the `languages/it-it.py` language file. Language files can be created and edited via the administrative interface.

Because string translation is eveluated lazily when the view is rendered, the **force** command can be issued anywhere other than in the views.

3.6.2 HTTP Exception and redirect

`HTTP` is a special exception clas defined by web2py. It can be raised anywhere in a model, controller, or view. It forces web2py to stop execution, commit any open database transactions, and return an HTTP response. For example:

```
raise HTTP(400,"something is wrong",a=3)
```

The first argument of `HTTP` is the HTTP error code. The second argument is the string that will be returned as body of the response. Additional optional named arguments (like `a=3`) are used to build the response HTTP header (`a: 3` in the example). If one does not wish to commit the open database transaction, the user should rollback before raising the exception.

Any exception other than `HTTP` will cause web2py to rollback any open database transaction, log the error traceback, issue a ticket to the visitor, and return a standard error page.

This means that only `HTTP` can be used for cross-page control flow. Other exceptions must be caught by the application, otherwise they are interpreted as bugs and recorded by web2py.

While `HTTP` can be used for redirection, web2py provides a function for this purpose:

```
redirect("http://mdp.cti.depaul.edu")
```

The `redirect` function simply raises the appropriate `HTTP` exception to perform a redirection. `redirect` takes an optional second argument which is the HTTP status code for the redirection (303 by default). Change this number to 307 for a temporay redirect and to 301 for a permanent redirect.

3.6.3 URL helper

`URL` is a function used to build web2py absolute URLs. The most common usage is in expressions like:

```
redirect(URL(a=request.application,\
             c=request.controller,\
             f="myfunction"))
```

It asks the client's browser to redirect to a URL within the same application and controller but to a different function, in this case "myfunction". A common shorthand is the following

```
redirect(URL(r=request,f="myfunction"))
```

and the application and controller are determined from the `request` object.

`URL` takes two optional parameters `args` and `vars` and they are better explained by the following examples:

```
URL(a="myapp",\
    c="mycontroller",\
    f="myfunction",\
    args=['hello','world'])
```

outputs

```
"/myapp/mycontroller/myfunction/hello/world"
```

while

```
URL(a="myapp",\
    c="mycontroller",\
    f="myfunction",\
    vars=dict(hello='world'))
```

outputs

```
"/myapp/mycontroller/myfunction?hello=world"
```

`args` and `vars` can be combined. The generated URL is automatically encoded.

The `args` attributes are then automatically parsed, decoded, and stored in `request.args` by web2py. Similarly, the `vars` are parsed, decoded, and stored in `request.vars`.

`args` and `vars` provide the basic mechanism by which web2py exchanges information with the client's browser. [SAY MORE]

3.6.4 XML

Given a string

```
s="<b>hello world</b>"
```

`s` can be displayed in a view like this

```
{{=s}}
```

By default the operator {{=...}} escapes the string as

```
"&lt;b;gt;hello world&lt;/b&gt;"
```

For security reasons `<b>...</b>` are not interepreted as XML tags unless the developer explicitly says so:

```
{{=XML(s)}}
```

`XML` is not a function but a class used to build XML objects. XML objects have a method .xml() that serializes an object as XML. One rarely needs to call this method explicitly since it is called by the {{=...}} operator.

This is useful in nested expressions using helpers, as shown later.

Notice that `XML` marks a string as XML but does not verify that it is valid XML. This choice improves performance and allows the string to contain javascript code as well.

3.6.5 HTML Helpers

Consider the following code in a view

```
{{=DIV('this','is','a','test',_id='123',_class='myclass')}}
```

it is rendered as

```
<div id="123" class="myclass">thisisatest</div>
```

We call `DIV` a helper class, something that can be used to build HTML programmatically. It corresponds to the HTML `<div>` tag.

Unnamed arguments are interpreted as objects contained in between the open and close tags. Named arguments starting with an underscore are interpreted as tag attributes (without the underscore). Some helpers also have named arguments that do not start with underscore and they play a special role.

The following other helpers
`A`, `B`, `BODY`, `BR`, `CENTER`, `DIV`, `EM`, `EMBED`, `FORM`, `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `HEAD`, `HR`, `HTML`, `IMG`, `INPUT`, `LI`, `LINK`, `OL`, `UL`, `META`, `OBJECT`, `ON`, `OPTION`, `P`, `PRE`, `SCRIPT`, `SELECT`, `SPAN`, `STYLE`, `TABLE`, `THEAD`, `TBODY`, `TFOOT`, `TD`, `TEXTAREA`, `TH`, `TITLE`, `TR`, `TT`

can be used to build complex expressions that can then be serialized in XML. For example

```
{{=DIV(B(I("hello ","<world>"))),_class="myclass)}}
```

is rendered

```
<div class="myclass"><b><i>hello &lt;world&gt;</i></b></div>
```

3.6.6 TAG Universal Helper

Sometime one needs to generate XML using custom tags. web2py provides a universal tag generator called TAG.

```
{{=TAG.name('a','b',c='d')}}
```

generates the following XML

```
<name c="d">ab</name>
```

where 'a' and 'b' and 'd' are escaped as appropriate. Using TAG you can can generate any HTML/XML tag you need and is not already provided in the API. TAGs can be nested and are serialized with str()

An equivalent syntax is

```
{{=TAG['name']('a','b',c='d')}}
```

Notice that TAG is a function and TAG.name or TAG['name'] is an object.

3.6.7 BEAUTIFY

BEAUTIFY is similar to XML, but it is used to mark objects instead of strings, for example:

```
{{=BEAUTIFY({a:["hello", XML("world")], b:(1,2)})}}
```

BEAUTIFY returns an XML-like object serializable to XML with a nice looking representation of its constructor argument. In this case the XML representation of

```
{a:["hello", XML("world")], b:(1,2)}
```

will render as

```
<table>
<tr><td>a</td><td>:</td><td>hello<br/>world</td></tr>
<tr><td>b</td><td>:</td><td>1</br>2</td></tr>
</table>
```

Notice that "hello" is escaped but "world" is not because it is marked as XML.

3.6.8 FORM, INPUT, TEXTAREA and SELECT/OPTION

The main purpose of helper is in building forms.

Consider the following example:

```
myform=SPAN(FORM(TABLE(TR('a',INPUT(_name='a')),
                       TR('',INPUT(_type='submit')))))
```

when serialized, by `myform.xml()` or in a view by

```
{{=myform}}
```

reads

```
<span><form enctype="multipart/form-data" method="post"><table>
   <tr><td>a</td><td><input name="a"/></td></tr>
   <tr><td></td><td><input type="submit"/></td></tr>
</table></form></span>
```

Notice a few things:

- The `TR` object knows it must contain `TD` elements. If it does not, the `TR` adds them automatically. The same is true for other types of container tags, for example `OL`, `UL`, `SELECT`.
- `myform` is not a string. It is an object and, as such, it is aware of its own contents. In particular, it knows it contains a form and it knows how to deal with it.

Assuming `myform` is defined in a controller, it can be used to process a request generated by the form itself with the following syntax:

```
if myform.accepts(request.vars,session,'myform'):
    ... the form has been validated and was accepted
elif myform.errors:
    ... the form was not accepted, errors are dict myform.errors
else:
    ... this is the first time the form is displayed, no errors.
```

`accepts` and `xml` are the only methods of any helper. `accepts` processes the request variables, validates them (if validators have been specified, which is not the case in this example) returns `True` on success and stores the form variables in `myform.vars`. If the form fails validation error messages are stored in `myform.errors` and the form is automatically modified to report those errors.

Here is a more complex example:

```
myform=SPAN(FORM(TABLE(TR('a',INPUT(_name='a',
                      requires=IS_INT_IN_RANGE(0,10))),
                      TR('',INPUT(_type='submit')))))
```

And let us pass an empty dictionary as request variables:

```
print myform.accepts(dict())
print myform.errors
print myform.xml()
```

produces the following output:

```
False
<Storage {'a': ''}>
<Storage {'a': 'too small or too large!'}>
<span><form enctype="multipart/form-data" method="post"><table>
  <tr><td>a</td><td><input name="a"/>
    <div class="error">too small or too large!</div></td></tr>
  <tr><td></td><td><input type="submit"/></td></tr>
</table></form></span>
```

Notice the error in the form.

Lets us pass now a valid session variable `a='5'`:

```
print myform.accepts(dict(a='5'))
print myform.vars
print myform.errors
print myform.xml()
```

produces the following output:

```
True
<Storage {'a': 5}>
<Storage {}>
<span><form enctype="multipart/form-data" method="post"><table>
  <tr><td>a</td><td><input name="a"/></td></tr>
  <tr><td></td><td><input type="submit"/></td></tr>
</table></form></span>
```

Notice that there are no errors in the form, the variable `a` has been parsed as an integer, and the form contains no error messages.

The `accepts` method takes three arguments. The first is required and it is a dictionary (or `Storage` object) with the request variables. The second is optional and it is the user's `session`. The third is an optional form name. This is important only if a page contains multiple forms. If a session is given, the form is modified to contain a unique key and prevent double sumission. The key is generated the first time the form is serialized. When the form is

submitted, if the key matches, the form is validated as above. If the key is missing or the key does not match `accepts` returns `True`, but the variables are stored in `myform`. This is the way `accepts` is intended to be used (see the section on forms).

The proper way to do it is have a self submitting forms that redirect to the destination page.

Here are two sample controllers:

```
def page1():
    form=FORM('your name:',INPUT(_name="name"),INPUT(_type="
       submit"))
    if form.accepts(session.vars,session):
         session.name=form.vars.name
         redirect(URL(r=request,f='page2'))
    return dict(form=form)

def page2():
    return dict(message='hello %s'%session.name)
```

3.7 Validators

Validators are classes used to validate input fields (including forms generated from database tables).

Here is an example of using a validator with a FORM:

```
INPUT(_name='a',requires=IS_INT_IN_RANGE(0,10))
```

Here is an example of a validator on a database table:

```
table=db.define_table('users',SQLField('name'))
table.name.requires=IS_NOT_EMPTY()
```

Validators are always assigned using the `requires` attribute. A field can have a single validator or multiple validators. Multiple validators are made part of a list. Here is an example of a validator on a database table:

```
table.name.requires=[IS_NOT_EMPTY(),IS_NOT_IN_DB(db,table)]
```

The second validator requires that the field is unique.

Validators are called by the function `accepts` on a FORM or other HTML helper object that contains a form. They are always called in the order they are listed.

All validators follow the following prototype:

```
class sample_validator:
    def __init__(self,*a,error_message='error'):
        self.a=a
        self.e=error_message
    def __call__(value):
        if validate(value): return (parsed(value),None)
        return (value,self.e)
    def formatter(self,value):
        return format(value)
```

i.e. when called to validate a value, a validator returns a tuple `(x,y)`. If `y` is `None` then the value passed validation and `x` contains a parsed value. For example, if the validator requires the value to be an integer `x` will be `int(value)`. If the value did not pass validation then `x` contains the input value and `y` contains an error message that explains the failed validation. This error message is used to report the error in forms that do not validate. All built-in validators have constructors that take the optional argument `error_message` that allows the developer to change the default error message. Here is an example of a validator on a database table

```
table.name.requires=IS_NOT_EMPTY(error_message=T('fill this!'))
```

where we have used the translation operator `T` to allow internationalization. Notice that default error messages are not translated.

3.7.1 Example: Validate a Phone Number

[FILL HERE]

```
IS_MATCH("^1+\d{3}\-?\d{3}\-?\d{4}$")
```

or for european something like

```
IS_MATCH("^\+?[\d\-]*$")
```

3.8 FORM Processing

When a web page displays a form, the form should not redirect to another page. The form should be submitted to the same page (self submission) and upon validation of the input, the current page should perform a redirection. This programming pattern is designed to make page design clean since the same object that generates a from, validates the form (for web2py this object

is a FORM or a SQLFROM). Moreover the redirection may depend on the input values.

Here is one more example of how this is achieved in web2py

```
def page1():
    form=FORM('your name:',\
              INPUT(_name="name",requires_IS_NOT_EMPTY()),\
              INPUT(_type="submit"))
    if form.accepts(session.vars,session):
         session.name=form.vars.name
         session.flash='you have been redirected'
         redirect(URL(r=request,f='page2'))
    elif form.error:
         response.flash='there are errors in your form'
    return dict(form=form)

def page2():
    return dict(message='hello %s'%session.name)
```

The user visits page1, fills the form and upon successful submission, is greeted by page2. Notice that the user's name is partsed from the `form.vars` and stored in `session` by page1, then it is retrieved from the `session` in page2. Also notice that the flash is set with `response.flash` when it is to be displayed in the current page and it is set with `session.flash` when it is to be displayed after redirection.

3.8.1 keep values in forms with keepvalues

Sometime you want an insert form that, upon submission and after the insert, retains the preceding values to hep the user insert a new record. This can be done:

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

db.define_table('user', SQLField('name','string'))
```

And in controller

```
def test():
    form=SQLFORM(db.user)
    if form.accepts(request.vars,session,keepvalues=True):
        response.flash="record inserted"
    return dict(form=form)
```

Notice the keepvalues=True argument of the accepts method.

3.9 Routes and URL Rewrite

web2py supports URL rerwite although this is not really recommended. You should really use Apache (or lighttpd) + mod_proxy (or mod_rewrite) for this purpose. Moreover rewriting web2py urls can break links in applications. So **do not do it**. Anyway if you really really want to ...

To use this feature just create a new file in web2py/ called routes.py and write something like this in it

```
routes_in=(
  ('/testme','/examples/default/index'),
)

routes_out=(
  ('/examples/default/index','/testme'),
)
```

Now visiting `http://127.0.0.1:8000/testme` will result in a call to `http://127.0.0.1:8000/examples/default/index` and this will only work from 127.0.0.1. To the visitor, all links to the page itself will look like links to `\testme`

Routes_in consist of a list of 2-tuples. The first element of a tuple is a regular expression of the form

```
'^IP_ADDRESS:PATH$'
```

where IP_ADDRESS is a regex pattern that will match the remote address (".*" to catch them all) and PATH is a regex that matches the path in the requested URL. The second part of the tuple tells web2py how to rewrite the path for those requests that match both the remote address and the path patterns. Notice that "." has to be escaped since it has a special regex meaning.

The presence of an IP_ADDRESS pattern allows to rewrite URL differently depending on the remote address IP address and, for example, prevent a certain domain from accessing a certain application.

Rules are executed in order. If a request does not match any route in, the url is not rewritten and the request is sent to web2py.

Routes_out are used to rewrite the output of the URL(...) function.

3.9.1 robots.txt and favicon.ico

Place the two files in the static folder in one of the installed apps. for example in

```
appications/examples/static/
```

Edit or create your routes.py file in the main web2py folder

```
routes_in=(
  ('.*:/favicon.ico','/examples/static/favicon.ico'),
  ('.*:/robots.txt','/examples/static/robots.txt'),
)

routes_out=()
```

3.10 Setting Default (root) Application

There are many ways to do it:

- call you app "init"
- If there is no app "init" the "examples" is the default
- You can create routes.py as discussed here `http://mdp.cti.depaul.edu/AlterEgo/default/show/67`
- If you run web2py behind apache with mod_proxy or mod_rewrite you can configure apache to do so as in `http://mdp.cti.depaul.edu/AlterEgo/default/show/38`

Chapter 4

Object Relational Mapper

web2py comes with an Object Relational Mapper (ORM), i.e. an API that maps Python objects into database objects such as queries, tables, and records.

web2py defines the following classes that comprise the ORM:

- **SQLDB** represents a database connection. For example:

```
db=SQLDB('sqlite://mydb.db')
```

- **SQLTable** represents a database table. It is never instantiated by the developer but is created by `SQLDB.define_table`. Its most important methods are **insert** and **drop**.

```
db.define_table('mytable',SQLField('myfield'))
```

- **SQLField** represents a database field. It can be instantianted and passed as an argument to `SQLDB.define_table`.
- **SQLRows** is the object returned by a database select. It can be thought of as a list of `SQLStorage`(s).

```
rows=db(db.mytable.myfield!=None).select()
```

- **SQLStorage** contains field values.

```
for row in rows: print row.myfield
```

- **SQLQuery** is an object that represents a database "where" clause

```
myquery=(db.mytable.myfield!=None)&(db.mytable.myfield>
    'A')
```

- **SQLSet** is an object that represents a set of records. Its most important methods are **select**, **update**, or **delete**.

```
myset=db(myquery)
rows=myset.select()
rows=myset.update(myfield='somevalue')
rows=myset.delete()
myset.delete()
```

- **SQLXorable** is an object derived from a field. Something that can be used to build expressions (such as queries, orderby, and groupby conditions). For example.

```
myorder=db.mytable.myfield.upper()|db.mytable.id
db().select(db.table.ALL,orderby=myorder)
```

4.1 Estabilishing a Connection with SQLDB

The proper way to estabilish a connection is in a model file. A connection is established with the following command:

```
db=SQLDB('sqlite://mydb.db')
```

There is nothing special about `db`, it is a local variable representing the connection object `SQLDB`. The constructor of `SQLDB` takes a single argument, the connection string. The connection string is the only web2py code that depends on a specific backend database. Here are examples of connection strings for specific types of supported back-end databases:

- **SQLite**. The binary versions of web2py come with the SQLite backend. This is a lighweight SQL database. All tables are stored in a single file. SQLite does not support concurrency and the database file is locked every time it is accessed. If the file does not exist it is created.
- **MySQL**. web2py can connect to a MySQL database via the MySQLdb driver. To connect to an existing MySQL database called 'test'

via the MySQL server program running on localhost through port 3306 (default), use

```
db=SQLDB('mysql://username:password@localhost:3306/test
    ')
```

– **PostgreSQL**. web2py can connect to a PostgreSQL database via the psycopg2 driver. To connect to an existing PostgreSQL database called 'test' via the PostgreSQL server program running on localhost through port 5432 (default), use

```
db=SQLDB('postgres://username:password@localhost:3306/
    test')
```

– **Oracle**. web2py can connect to a Oracle database via the cx_Oracle driver. To connect to an existing PostgreSQL database called 'test' use

```
db=SQLDB('oracle://username:password@test')
```

Notice that in the case of MySQL, PostgreSQL and Oracle the database "test" must be created outside web2py. Once the connection is established, web2py will create, alter, and drop tables appropriately.

4.2 SQLDB.define_table

Given an established connection and SQLDB object `db` it is easy to create table using the command:

```
db.define_table('users',
    SQLField('name'),
    SQLField('email'),
    migrate=True)
```

`define_table`'s first argument is the table name; a variable number of SQLField(...) arguments that represent the fields to be stored in the table with their respective attributes; and an optional third argument `migrate` which by default is `True`.

If `migrate` is `True`, web2py will create the table if not there, or alter the table if the existing table does not match the current definition. In order to determine whether the table exists or not, web2py uses a

`.table` file placed in the current folder. It is possible to specify the name of this file with `migrate='filename'`. If the developer does not wish web2py to create/alter the database because the database was created outside web2py or because the developer wants web2py to ignore the `.table` file, this can be done by setting `migrate=False`.

4.3 SQLField

The constuctor of an `SQLField` takes the following arguments, of which only 'fieldname' is required:

argument	default value	comment
fieldname		the field name
type	'string'	the field type
length	32	applies only to string
default	None	(read below)
required	False	whether the field is required for insertion
requires	(read below)	validators for the field
notnull	False	if True translates to SQL NOT NULL
unique	False	if True translates to SQL UNIQUE
ondelete	'CASCADE'	applies only to reference fields

The "default" value is utilized in two places: a) in forms generated with `SQLFORM(db.table)`; b) in INSERT statements if a value for the field is not specified. A value of `None` is translated to a SQL `NULL` value.

`required` indicates whether or not an insert is allowed without a specified value or a default value for the field.

`requires` points to a validator object or a list of validators. Validators are used only to validate entry/edit forms generated by `SQLFORM(db.table)`. Notice that a validator can also format the input forms used to insert data into the database. The following table shows default validator for each type

field type	default field attributes	default field validators
'string'	length=32	IS_LENGTH(length)
'blob'		
'boolean'		
'integer'		IS_INT_IN_RANGE(-1e100,1e100)
'double'		IS_FLOAT_IN_RANGE(-1e100,1e100)
'date'		IS_DATE()
'time'		IS_TIME()
'datetime'		IS_DATETIME()
'password'		CRYPT()
'upload'		
'reference'		

4.3.1 date, time, and datetime

[REWRITE]

`date`, `time`, and `datetime` objects, by default, are serialized following the ISO standard:

```
YYYY-MM-DD hh:mm:ss
```

here is an example:

```
>>> db.define_table('instant',SQLField('timestamp','datetime'))
>>> db.instant.insert(timestamp=datetime.datetime(2007,12,21,9,30,00))
>>> db.instant.insert(timestamp='2007-12-21 9:30:00')
>>> rows=db().select(db.instant.timestamp)
>>> t0=rows[0].timestamp
```

Here `t0` is a `Datetime` object. If the DATETIME format is not specified the default formatting is ISO.

A DATETIME format is specified as an argument of the IS_DATETIME validator:

```
db.instant.timestamp.requires=IS_DATETIME('%m/%d/%Y %H:%M:%S')
```

4.3.2 password

[REWRITE]

A password field is just a string with two added caveats. By default is has a CRYPT validator which encrypts the string using the MD5 hash algorithm. The encryption is performed by `SQLFORM(...).accepts(...)`. Moreover, edit forms treat passwords differently from string fields by not showing the text value in the HTML. User input is formatted as `*******`.

4.3.3 upload

An upload field is just a string but it is handled differently by forms since it is rendered as an upload form. Moreover, `SQLFORM(...).accepts(...)` receives the uploaded file, renames it in a safe way while preserving its file extension, saves the file in the [appname]/uploads folder, and stores the filename in the database table's `upload` field.

[WRITE MORE]

4.3.4 references

`reference` fields requered (Tim: Not sure what "requered" should be since it's not obvious what this section should be about. Make sure you split this code listing up into sections, otherwise people may just ignore it and this section is very important since it shows many of the different aspects of the ORM and people will probably refer to it often.)

```
1     >>> if len(sys.argv)<2: db=SQLDB("sqlite://test.db")
2     >>> if len(sys.argv)>1: db=SQLDB(sys.argv[1])
3     >>> tmp=db.define_table('users',\
4               SQLField('stringf','string',length=32,required
                   =True),\
5               SQLField('booleanf','boolean',default=False),\
6               SQLField('passwordf','password'),\
7               SQLField('blobf','blob'),\
8               SQLField('uploadf','upload'),\
9               SQLField('integerf','integer'),\
10              SQLField('doublef','double'),\
```

```
            SQLField('datef','date',default=datetime.date.
                today()),\
            SQLField('timef','time'),\
            SQLField('datetimef','datetime'),\
            migrate='test_user.table')

Insert a field

 >>> db.users.insert(stringf='a',booleanf=True,passwordf=
    'p',blobf='x',\
                     uploadf=None, integerf=5,doublef
                        =3.14,\
                     datef=datetime.date(2001,1,1),\
                     timef=datetime.time(12,30,15),\
                     datetimef=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]
 >>> me.name
 'Massimo'
 >>> db(db.person.name=='Massimo').update(name='massimo')
 >>> db(db.person.name=='Marco').delete()

 Update a single record

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

```
50
51      Examples of complex search conditions
52
53      >>> len(db((db.person.name=='Max')&(db.person.birth<'
            2003-01-01')).select())
54      1
55      >>> len(db((db.person.name=='Max')|(db.person.birth<'
            2003-01-01')).select())
56      1
57      >>> me=db(db.person.id==person_id).select(db.person.name
            )[0]
58      >>> me.name
59      'Max'
60
61      Examples of search conditions using extract from date/
            datetime/time
62
63      >>> len(db(db.person.birth.month()==12).select())
64      1
65      >>> len(db(db.person.birth.year()>1900).select())
66      1
67
68      Example of usage of NULL
69
70      >>> len(db(db.person.birth==None).select())
71      0
72      >>> len(db(db.person.birth!=None).select())
73      1
74
75      Examples of search consitions using lower, upper, and
            like
76
77      >>> len(db(db.person.name.upper()=='MAX').select())
78      1
79      >>> len(db(db.person.name.like('%ax')).select())
80      1
81      >>> len(db(db.person.name.upper().like('%AX')).select())
82      1
83      >>> len(db(~db.person.name.upper().like('%AX')).select()
            )
84      0
85
86      orderby, groupby and limitby
87
88      >>> 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

An INNER JOIN

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

A LEFT OUTER JOIN

>>> len(db(db.dog.owner==db.person.id).select(left=db.
    dog))
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')
```

```
129     >>> aid=db.author.insert(name='Massimo')
130     >>> pid=db.paper.insert(title='QCD')
131     >>> tmp=db.authorship.insert(author_id=aid,paper_id=pid)
132
133     Define a SQLSet
134
135     >>> authored_papers=db((db.author.id==db.authorship.
            author_id)&\
136                             (db.paper.id==db.authorship.
                                paper_id))
137     >>> rows=authored_papers.select(db.author.name,db.paper.
            title)
138     >>> for row in rows: print row.author.name, row.paper.
            title
139     Massimo QCD
140
141     Example of search condition using  belongs
142
143     >>> set=(1,2,3)
144     >>> rows=db(db.paper.id.belongs(set)).select(db.paper.
            ALL)
145     >>> print rows[0].title
146     QCD
147
148     Example of search condition using nested select
149
150     >>> nested_select=db()._select(db.authorship.paper_id)
151     >>> rows=db(db.paper.id.belongs(nested_select)).select(
            db.paper.ALL)
152     >>> print rows[0].title
153     QCD
154
155     Output in csv
156
157     >>> str(authored_papers.select(db.author.name,db.paper.
            title))
158     'author.name,paper.title\\r\\nMassimo,QCD\\r\\n'
159
160     Delete all leftover tables
161
162     >>> db.authorship.drop()
163     >>> db.author.drop()
164     >>> db.paper.drop()
```

4.4 SQLTable

4.4.1 insert

4.4.2 drop

4.5 SQLQuery and SQLSet

Yes it is correct

```
query1=(db.table.field==value)|(db.table.field==other_value)
db(query1)
db(query1)(query2)
db((query1)&(query2))
```

They are equivalent notations.

In a SQLSet you can:

```
db(query1).select(db.table.field)
db(query1).delete()
db(query1).update(field=value)
```

4.5.1 delete

4.5.2 select

```
db=SQLDB(....)
db.define_table('mytable',SQLField('myfield'),SQLField('
    datefield','date'))
```

You can order ascending:

```
db().select(orderby=db.mytable.myfield)
```

Order descending:

```
db().select(orderby=~db.mytable.myfield)
```

Combine ordering rules:

```
db().select(orderby=~db.mytable.myfield|db.mytable.
    datefield)
```

You can also do

```
db().select(orderby="mytable.myfield DESC")
```

where "..." is an SQL orderby option, and things like

```
orderby=db.mytable.myfield.upper()
orderby=db.mytable.datefield.month()
```

or combinations of any of the above.

pagination example

```
def show_records():
    if len(request.args):
       page=int(request.args[0])
    else:
       page=0
    query=db.table.id>0
    rows=db(query).select(limitby=(page,page+100))
    backward=A('backward',_href=URL(r=request,args=[page
       -100]) if page else "
    forward=A('forward',_href=URL(r=request,args=[page+100])
        if len(rows) else "
    return dict(rows=rows,backward=backward,forward=forward)
```

4.5.3 update

4.5.4 operator like

The SQL LIKE operator is accessed by the like() method of a field:

```
db(db.category.name.like('d%'))
```

considers only categories starting with d.

4.5.5 operator belongs

4.5.6 date and datetime operators

4.6 SQLRows

4.7 SQLFORM and SQLTABLE

[FILL HERE]

4.7.1 About file upload and renaming

Example.

in models/db.py

```
db.define_table('image',SQLField('file','upload'),SQLField('
    filename'))
```

in controllers/db.py

```
def test():
    form=SQLFORM(db.image,fields=['file'])
    if request.vars.file!=None:
        form.vars.filename=strip_path_and_sanitize(request.
            vars.file.filename)
    if form.accepts(request.vars,session):
        response.flash='file uploaded'
    return dict(form=form)

def download():
    return response.stream(open(os.path.join(request.folder,
        'uploads',request.args[0]),'rb'))
```

you can link the file as

```
/app/controller/download/{{=image.file}}/{{=image.filename}}
```

You need your own strip_path_and_sanitize function

4.8 IS_IN_DB and IS_NOT_IN_DB

Consider:

```
db.recipe.category.requires=IS_IN_DB(db,'category.id','
    category.name')
```

The parameters are:

1. a set of records
2. `category.id` describes how the field will be represented in the database
3. `category.name` describes how the field will be represented to the use

For example you can do things like:

```
requires=IS_IN_DB(db(db.category.name.like('d%')),
                  'category.id',
                  'Name: %(name)s, id: %(id)s any text your
                     like')
```

4.9 Commit and Rollback Transaction

In web2py, if you do multiple inserts/update/select in a controller function, web2py will commit all of them when the function returns unless there is an uncaught exception. In this case it will rollback all of them before issuing a ticket.

You have the option to

```
db.commit()
```

and

```
db.rollback()
```

anywhere you please.

4.10 Distributed Transactions

Assuming you have two (or more) connections to distinct postgresql databases, let's say:

```
dba=SQLDB('postgres://...')
dbb=SQLDB('postgres://...')
```

in your models or controllers, you can commit them both explicitly with

```
SQLDB.distributed_transaction_commit(dba,dbb)
```

rollback and raise Exception on failure. Connections are not closed either way. So that you can still use dba and dbb after the distributed transaction.

4.11 SQL Logs

[BREAK THIS IN TWO SECTIONS]

`sql.log` logs only CREATE, DROP, and ALTER table.

INSERTs, UPDATEs and SELECTs are logged one at the time in the `db._lastsql` string.

To check whether caching is used you can do

```
db._lastsql=None
rows=db(...).select(...,cache=(cache.ram,60))
if db._lastsql:
    print 'SQL WAS EXECUTED AND RESULT CACHED:',db._lastsql
else:
    print 'SQL WAS NOT EXECUTED, DATA RETRIEVED FROM CACHE'
```

4.12 Exporting data in XML using TAG

Now that it is needed since web2py exports already in CSV but you can do define

```
def export_xml(rows):
    idx=range(len(rows.colnames))
    colnames=[item.replace('.','_') for item in rows.
        colnames]
    records=[]
    for row in rows.response: records.append(TAG['record'
        ](*[TAG[colnames[i]](row[i]) for i in idx]))

    return str(TAG['records'](*records))
```

Now if you have a model like

```
db=SQLDB('sqlite://test.db')
db.define_table('mytable',SQLField('myfield'))
for i in range(100): db.mytable.insert(myfield=i)
```

you can get your data in XML by doing

```
print export_xml(db().select(db.mytable.ALL))
```

4.13 Running on Google App Engine using GQLDB

Chapter 5

Views and the Template Language

web2py uses Python for the models, the controllers, and the views, but it uses a slightly modified Python syntax for views with the intention of allowing more functionality while not posing any restrictions on proper Python usage.

The purpose of a view is to embed code (Python) in an HTML document. This poses some problems:

- how should one escape code?
- should one indent based on Python or HTML rules?

web2py uses {{ ... }} to escape Python code embedded in HTML. The advantage of using curly brackets instead of angle brackets is that it's transparent to all common HTML editors and thus allows the developer to use those editors to create web2py views.

We also believe that since the developer is embedding Python code into HTML, the document should be indented according to HTML rules and not Python rules, hence we allow unindented Python inside the {{...}} tags. Since Python uses indentation to delimit blocks of code we need a different way to delimit blocks of code. For this purpose the web2py templating language uses the `pass` keyword. A block starts with a line ending with a semicolon and ends with a line starting with

`pass`. The keyword `pass` is not necessary when the end of the block is obvious from the context. For example in

```
{{
if i==0:
document.write('i is 0')
else:
document.write('i is not 0')
pass
}}
```

Notice that web2py did not invent `pass`. It is a reserved word in Python. Some Python editors, such as emacs, use the keyword `pass` to understand the division of blocks in the same way as web2py does and can use it to re-indent code automatically.

web2py templating language does exactly the same. When it finds something like

```
<html><body>
{{for i in range(10):}}{{=i}}hello<br/>{{pass}}
</body></html>
```

it translates it into a program

```
response.write("""<html><body>""",escape=False)
for i in range(10):
    response.write(i)
    response.write("""hello<br/>""",escape=False)
response.write("""</body></html>""",escape=False)
```

where the HTML is written to the `response.body` and the indentation is determined by the control flow and the block division.

When there is an error in a web2py view, the error report shows the generated view code, not the actual view as written by the developer. This helps the developer debug the code by highlighting its logical structure as opposed to its aesthetical structure (which is something that can be debugged with an HTML editor or the DOM inspector of the browser).

Also note that

```
{{=i}}
```

is translated into

```
1 response.write(i)
```

and without `escape=False`. Any variable displayed into the HTML is serialized or escape by default.

Here is another example that introduces the `H1` helper

```
1 {{=H1(i)}}
```

which is translated into

```
1 response.write(H1(i))
```

upon execution the H1 object is serialized and its components are printed recursively, serialized, or escaped as needed. This mechanism guarantees that all content displayed into the web page is always escaped thus preventing XSS vulnerabilities. At the same time, the code remains simple and easy to debug.

The method `response.write(obj,escape=True)` takes two arguments, the object to be printed and whether it has to be escaped (set to `True` by default). If `obj` has a `.xml()` attribute it simply calls it and displays the output (escape value is ignored), otherwise it uses the object's `__str__` method to serialize it and, if requested, escapes it. All built-in helper objects (`H1` in the example) are objects that know how to serialize themselves via the `.xml()` attribute.

This is done transparently for the developer who, although he or she can, never needs to call the method `response.write` explicitly.

5.1 Basic Syntax

The web2py templating language supports all Python control structures. Here we provide some examples of each of them. They can be nested according to usual programming practice.

5.1.1 for...in

In templates you can loop over any iterable object

```
{{items=['a','b','c']}}
<lu>
{{for item in items:}}<li>{{=item}}</li>{{pass}}
</ul>
```

which produces

```
<lu>
<li>a</li>
<li>b</li>
<li>c</li>
</ul>
```

Here `item` is any iterable object such as a Python list, Python tuple,or SQLRows object, or any object that is implemented as an iterator. Notice that the elements displayed are first serialized or escaped by a call to `response.write`.

5.1.2 while

You can loop uwing a while loop

```
{{k=3}}
<lu>
{{while k>0:}}<li>{{=k}}{{k=k-1}}</li>{{pass}}
</ul>
```

which produces

```
<lu>
<li>3</li>
<li>2</li>
<li>1</li>
</ul>
```

5.1.3 if...elif...else

You can express conditional clauses

```
{{
import random
k=random.randint(0,100)
}}
```

```
<h2>
{{=k}}
{{if k%2:}}is odd{{else:}}is even{{pass}}
</h2>
```

produces

```
<h2>
45 is odd
</h2>
```

Notice that since it is obvous that the `else` closes the first `if` block there is no need for a `pass` statement and using one would be incorrect. There is instead a need to close the `else` block with `pass`.

We remind the reader that in Python "else if" is written "elif" as in the following example:

```
{{
import random
k=random.randint(0,100)
}}
<h2>
{{=k}}
{{if k%4==0:}}is disible by 4
{{elif k2==0:}}is even
{{else:}}is odd
{{pass}}
</h2>
```

It produces

```
<h2>
64 is divisible by 4
</h2>
```

5.1.4 try...except

It is also possible to use `try...except` statements in views with one caveat. Consider the following example:

```
{{try:}}
Hello {{=1/0}}
{{except:}}
division by zero
{{pass}}
```

It will produce the following output

```
Hello
division by zero
```

In fact, execution stops at 1/0 (when the division by zero occurs). This example indicates that all output generated before an exception occurs in the view, including the output generated inside the try block, will be rendered in the view.

5.1.5 def...return

The web2py template language allows the developer to define and implement functions which can return any Python object or an ordinary text/html string. Here we consider two examples:

```
{{def itemize1(link): return LI(A(link,_href="http://"+link)
    )}}
<lu>
{{=itemize1('www.google.com')}}
</lu>
```

produces the following output

```
<lu>
<li><a href="http:/www.google.com">www.google.com</a></li>
</lu>
```

Notice that the function `itemize1` returns a helper object that is then inserted at the location where the function is called.

Consider now the following code

```
{{def itemize2(link):}}
<li><a href="http://{{=link}}">{{=link}}</a></li>
{{return}}
<lu>
{{itemize2('www.google.com')}}
</lu>
```

it produces exactly the same output as above. In this case the function `itemize2` represents a piece of HTML that is going to replace the web2py tag where the function is called. Notice that there is no '=' in front of the call to `itemize2`.

There is one caveat: Functions defined inside a view must terminate with a return statement or the automatic indentation will fail.

5.2 Page Layout

Yes. Views can extend and include other views in a tree like structure. The root of the tree is what we call a layout view. It is just an HTML file that you can edit as any other view using the administrative interface.

Example of view:

```
{{extend 'layout.html}}
<h1>Title</h1>
{{include 'footer.html'}}
```

A layout file must be something like

```
<html><head>....</head>
<body>
{{include}}
</body>
</head>
```

web2py controller automatically export some variables to the layout (reponse.title, response.keywords, response.description, response.flash, response.flash, and you can define your own). Here is how you can create your own layout.html file.

- find a page layout that you like
- make sure it released under some open source artistic license
- give a name to it ('layoutsleek001')
- create a folder under static called layoutsleek001 and move all the static files for the layout in there
- save the layout under views as layoutsleel001.html
- edit layoutsleek001.html and use =URL(r=request,c='static',f='layoutsleek001/whatever.css to include the static files
- edit layoutsleek001.html and make sure it has the following structure:

Example below: layoutsleek001.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "
    http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;
     charset=utf-8" />
  <meta name="keywords" content="{{=response.keywords}}" />
  <meta name="description" content="{{=response.description
     }}" />
  <title>{{=response.title if response.title else request.
     application)}}</title>
  <link href="{{=URL(r=request,c='static',f='style.css')}}"
     rel="stylesheet" type="text/css"/>
  {{include 'web2py_ajax.html'}}
</head>
<body>

<h1>Here goes the header, followed by the menu</h1>

{{if response.menu:}}
<div id="menu">
<ul>
  {{for _name,_active,_link in response.menu:}}
    <li><a href="{{=_link}}" class="{{='active' if _active
       else 'inactive'}}">{{=_name}}</a></li>
  {{pass}}
</ul>
</div>
{{pass}}

<h1>Here goes the flash message</h1>

{{if response.flash:}}<div id="flash">{{=response.flash}}</
    div>{{pass}}

<h1>Here goes the main (the view that extends this one)</h1>

{{include}}

<h1>... and here the footer</h1>

</body>
</html>
```

Web2py can do this:

layout.html

```
<html><body>
{{include}} <!-- must come before the two blocks below -->
whatever html
{{sidebar()}}
whatever html
{{maincontent()}}
whatever html
</body></html>
```

myview.html

```
{{extend 'layout.html'}}
{{def sidebar():}}
<h1>This is the sidebar</h1>
{{return}}
{{def maincontent():}}
<h1>This is the maincontent</h1>
{{return}}
```

5.3 Ajax

Since version 1.28 web2py ships with jQuery core libraries.

We find jquery to be small, efficient and very intuitive to use.

According to this `http://google-code-updates.blogspot.com/2007/11/my-how-weve-grown.html` Google uses jQuery too.

Here are a list of jQuery plugins that you may find useful:

- for datepicker read `http://mdp.cti.depaul.edu/AlterEgo/default/show/79`
- for plotting `http://code.google.com/p/flot/`
- for grid `http://webplicity.net/flexigrid/`
- for menu `http://www.dynamicdrive.com/dynamicindex17/ddaccordionmenu.htm`
- for sortable lists `http://interface.eyecon.ro/demos/sort.html`

5.3.1 Effects: fade example

```
<div id="message">Hello World</div>
<script>fade("message",+0.3):</script>
```

make the message "Hello World" appear. Use `-0.3` to make it disappear.

5.3.2 Effects: scroller

Here is a sample javascript scroller with jQuery:

```
<ul id="ticker">
  <li>News Item 1</li>
  <li>News Item 2</li>
  <li>News Item 3</li>
</ul>

<script language="javascript">
  var newsitems, curritem=0;
  $(document).ready(function(){
     newsitems = $("#ticker li").hide().size();
     $("#ticker li:eq("+curritem+")").slideDown();
     setInterval(function() {
       $("#ticker li:eq("+curritem+")").slideUp();
       curritem = ++curritem%newsitems;
       $("#ticker li:eq("+curritem+")").slideDown();
     },2000); //time in milliseconds
  });
</script>
```

Additional readings: `http://15daysofjquery.com/`

5.3.3 Asynchronous calls

```
<form><input id="source"/></form>
<button onclick="ajax('test',['source'],'destination');"/>
<div id="destination">...</div>
```

When the button is cliched an http request to `test?source={value of source}` is sent to sever and response is placed in destination tag.

5.3.4 Computation in background

This function does the computation, here it just sleeps for 5 secs

```
1 def background():
2     import time
3     time.sleep(5)
4     return "computation result"
```

This is the page accessed by the user. The computation is called by an ajax call and executed in background.

```
1 def foreground():
2     return dict(somescript=SCRIPT("ajax('background',[],'
          target');"),
3                 somediv=DIV('working...',_id='target'))
```

5.3.5 Date Picker

One of these is the built-in configuration for jquery datepicker but it does ship with datepicker.

That means that if you just copy these two files

- `http://mdp.cti.depaul.edu/examples/static/ui.datepicker.css`
- `http://mdp.cti.depaul.edu/examples/static/ui.datepicker.js`

in the static folder of your application, all INPUT fields of class "date", in particular all those automatically generated by SQLFORM for database fields of type "date", will display a popup ajax datepicker.

As long as you use the defaul layout.html or a custom layout that {{include "web2py_ajax.html"}} in .. , there is no need to configure anything. If the two files are in static, web2py will find them and use them.

You can try the datepicker here: `http://mdp.cti.depaul.edu/demo_app/appadmin/insert/db/event`

5.3.6 Confirmation on Delete

Just add this to your layout after the include 'web2py_ajax.html'

```
<script><!--
$(document).ready(function(){
    $('.delete').attr('onclick','if(this.checked)
    if(!confirm("Sure you want to delete this object?"))
       this.checked=false;');
});
//--></script>
```

Will work on all your forms.

Chapter 6

Production Deployment

6.1 Working with Apache

In a production environment you should use web2py with Apache or lighttpd. With Apache there are two ways.

6.1.1 mod_proxy

install mod_proxy and send all your requests to the web2py web server.
You would put something like this in your `/etc/apache2/sites-available/default`

```
NameVirtualHost *:80
<VirtualHost *:80>
   <Location "/">
     Order deny,allow
     Allow from all
     ProxyPass http://127.0.0.1:8000/
     ProxyPassReverse http://127.0.0.1:8000/
   </Location>
   LogFormat "%h %l %u %t "%r" %>s %b" common
   CustomLog /var/log/apache2/access.log common
</VirtualHost>
```

or you can use mod_swgi. web2py is wsgi compliant. This requires that you use the source version of web2py and you read the mod_wsgi documentation.

6.1.2 mod_wsgi

The handler is in wsgihandler.py in the main web2py folder.

6.1.3 mod_ssl (https and ssl)

Here is a sample apache config file:

```
NameVirtualHost *:80
NameVirtualHost *:443
<VirtualHost *:80>
   <Location "/admin">
     SSLRequireSSL
   </Location>
   <Location "/examples">
     Order deny,allow
     Allow from all
     ProxyPass http://127.0.0.1:8000/examples
     ProxyPassReverse http://127.0.0.1:8000/
   </Location>
   <Location "/">
     Order deny,allow
     Allow from all
     ProxyPass http://127.0.0.1:8000/examples/default/index
     ProxyPassReverse http://127.0.0.1:8000/
   </Location>
   LogFormat "%h %l %u %t \"%r\" %>s %b" common
   CustomLog /var/log/apache2/access.log common
</VirtualHost>

<VirtualHost *:443>
  SSLEngine On
  SSLCertificateFile /etc/apache2/ssl/server.crt
  SSLCertificateKeyFile /etc/apache2/ssl/server.key
  <Location "/">
     Order deny,allow
     Allow from all
     ProxyPass http://127.0.0.1:8000/
     ProxyPassReverse http://127.0.0.1:8000/
  </Location>
  LogFormat "%h %l %u %t \"%r\" %>s %b" common
  CustomLog /var/log/apache2/access.log common
</VirtualHost>
```

6.1.4 Changing default error pages

When requesting an invalid page web2py generates a default 400 error page.

This can be overwritten. There are at least three ways:

1. If you use mod_proxy you can set in the apache config file

```
ProxyErrorOverride On
```

Then you can filter error messages using mod_include SSI and render them as you like.

2. You create a routes.py file in web2py/ as discussed in a previous post.

You need to explicitly list all the allowed routes and add a catch all invalid routes at the end, for example:

```
1 ('^.*:.*$','/examples/default/index')
```

will render all invalid urls as '/examples/default/index'

3. On error web2py also adds a header web2py_error to the http response. You can easily create wsgi app that wraps web2py wsgibase and filters response accodingly.

6.2 Working with lighttpd and FastCGI

The handler is in fcgihandler.py in the main web2py folder.

6.3 web2py as a CGI Script

[WRITE MORE]

6.4 Deployment without built-in apps

It is possible to remove the built-in apps although they are very small (∼1MB) and they are useful. Anyway...

The first time you need to run web2py with all its files. After that you can remove any installed application you want including admin, examples and welcome. Nothing breaks except that you can no longer visit pages that you do not have. You only get an internal error if you try to do that. welcome.tar is the scaffold application. That's one file you should not remove.

Notice that if you remove admin you can no longer login as administrator and the database administration (appadmin) will be disabled for all you apps, unless you adit the appadmin.py files and change the authentication mechanism.

6.5 Load balancing

You can deploy multiple installations of web2py behind a NAT server that provides load balancing. The applications/ folder has to be NFS mounted and you need to backup that as well as your database. Do not use sqllite but connect to a postgresql database. Your bottleneck will be the database, as with any other web application. To avoid complications turn the administrative interface OFF when in production (do not give it a password) and set migrate=False.

6.6 Concurrency Issues and Network File Systems

In order to make web2py work with a load balancing and multiple installation, they all need to share the same folder. For example via NFS.

Notice that you also eed NFS file locking to work, hence you need the

```
nlockmgr Daemon
```

"Nlockmgr is an NFS server daemon that enables file locking and sharing. However, it is not supported by all hosts. If this daemon is not available on your host, enable it or disable file locking and file sharing in the Reflection NFS client. If you want your users to be able to share files, you need the nlockmgr daemon to prevent file corruption."

6.6.1 How to create Flash RPC applications using web2py and PyAMF

by Sharriff Aina

I would explain with a very simple example how you can create Flash RPC applications using web2py as a RPC service provider. I expect you to be familiar with the following topics as they are beyond the scope of this HowTo: Actionscript programming

- RPC Web services
- AMF
- web2py programming

Start off by getting the latest web2py from the site. Next, retrieve the latest version of PyAMF from the Python CheeseShop or directly from the PyAMF site `http://pyamf.org/wiki/Download`

Lets create a simple service that takes 2 numerical values, adds them together and returns the sum. We would call the service `addNumbers`. Create a flash file using Adobe Flash (all versions as from MX 2004), in the first frame of the file, add these lines:

```
1  import mx.remoting.Service;
2  import mx.rpc.RelayResponder;
3  import mx.rpc.FaultEvent;
4  import mx.rpc.ResultEvent;
5  import mx.remoting.PendingCall;
6
7  val1 = 23;
8  val2 = 86;
9
10 service = new Service("http://localhost:8000/pyamf_test/
       default/
```

```
gateway", null, "addNumbers", null, null);
var pc:PendingCall = service.addNumbers(val1, val2);
pc.responder = new RelayResponder(this, "onResult", "onFault
    ");

function onResult(re:ResultEvent):Void {
    trace("Result : " + re.result);
    txt_result.text = re.result;
}

function onFault(fault:FaultEvent):Void {
    trace("Fault: " + fault.fault.faultstring);
}

stop();
```

The last thing to do would be to create a dynamic text field called `txt_result` and place it on the stage.

As you have noticed, we have imported the mx remoting classes to enable Remoting in Flash, you can get them here. I would be using the Actionscript version 2 version of the classes. Add the path of the classes to your class path in the Adobe Flash IDE or just place the `mx` folder next to the newly created file. Compile and export(publish) the swf flash file as `pyamf_test.swf`.

Lets set up the gateway in web2py as we defined in our Flash file, create a new appliance, `pyamf_test`, in the `default` controller, create a function called `gateway` and add these lines to it:

```
import pyamf
import pyamf.remoting.gateway

def addNumbers(val1, val2):
    return val1 + val2

services={'addNumbers.addNumbers':addNumbers}
def gateway():
    base_gateway = pyamf.remoting.gateway.BaseGateway(
        services)
    context = pyamf.get_context(pyamf.AMF0)
    pyamf_request = pyamf.remoting.decode(request.body.read
        (), context)
    pyamf_response = pyamf.remoting.Envelope(pyamf_request.
        amfVersion,
```

```
13                                              pyamf_request.
                                                   clientType)
14      for name, message in pyamf_request:
15          pyamf_response[name] = base_gateway.getProcessor(
               message)(message)
16      response.headers['Content-Type'] = pyamf.remoting.
           CONTENT_TYPE
17      return pyamf.remoting.encode(pyamf_response, context).
           getvalue()
```

Next, place the pyamf_test.amf, pyamf_test.html, AC_RunActiveContent.js, crossdomain.xml files in the `static` folder of the pyamf_test controller,fireup web2py using 8000 as the server port, point your browser to

```
http://localhost:8000/pyamf_test/static/pyamf_test.html
```

and see the result of the `addNumbers` function displayed in the textfield.

6.6.2 response.myvar=value vs return dict(myvar=value)

My recommendation is to use response.my_variable only for those variables that are currently used by the layout.html:

```
1 response.title
2 response.keywords
3 response.description
4 response.flash
5 response.menu
```

This makes easy to build new layout.html files and swap them (I have a student working on it).

Moreover passing all other variables via return dict(...) makes it easier to build doctests.

6.6.3 Eclipse, web2py, imports, code hints and code completion

This document explains how to use Eclipse with web2py.

It was tested on the cookbook example.

The only notable things so far are:

- make a new project, unchecking the default, and choosing the root

folder as the location (where web2py.py is located)

- add `__init__.py` modules where you want to be able to import things

from your own code. Only models have been tested so far.

- add the web2py root folder as a project source folder for the

python path. On the mac version I am using this is the Resources folder, again where the web2py.py file is located

- download the web2py source and add the web2py source root as an

external source folder

- (OPTIONAL) Go to Preferences > Pydev > PyDev Extensions > Auto Imports, and uncheck "Do auto imports?"

One gotcha is make sure you choose the correct case:

```
1 return dict(records=SQLTABLE(records))
```

There are two SQLTABLE: SQLTABLE and SQLTable. The lower case one does not need to exposed since it is not intended to be instantiated by the user. The upper case one is being used to output html, and is the one we want. As Python is case sensitive, you would not get the expected outcome if you choose the wrong item.

To let Eclipse know about variables being passed into the controller at runtime, you can do the following

```
1 global db
2 global request
3 global session
4 global response
```

This will remove the warnings about them being undefined.

Typing `redirect` will get the correct import and you can use it normally.

To use `session`, `request` and (theoretically but untried) `response` with hints and code completion, you can use this code:

```
req=Request()
req=request
ses=Session()
ses=session
resp=Response()
resp=response
```

as you type those you will get the imports for these objects as well:

```
from gluon.globals import Request
from gluon.globals import Session
from gluon.globals import Response
```

Then you can use `req` as you would with `request` but with code hinting, etc., and `ses` for `session`, and `resp` for `response`. If anyone knows a better way to cast `request`, `session`, and `response` as their correct types, please do leave a comment.

Code hints and completion in Eclipse is very nice and provide a valid alternative to the web2py built-in editor. Unfortunately you do not get the code hints unless you also import the statement. If you choose to keep "Do auto imports?" checked, imports from your own models, controllers, etc., may throw errors. You can review those and delete the undesired imports. Or perhaps simply use it as a memory aid and keep typing without selecting the object so as to not trigger the auto import.

Please note that this is still a work in progress!

6.6.4 httpserver.log and the log file format

by Damian Gadek

The web2py web server logs all requests to a file called

```
httpserver.log
```

in the root web2py directory. An alternative filename and location can be specified via web2py command-line options.

New entries are appended to the end of the file each time a request is made.

Each line looks like this:

```
127.0.0.1, 2008-01-12 10:41:20, GET, /admin/default/site,
    HTTP/1.1, 200, 0.270000
```

The format is:

```
ip, timestamp, method, path, protocol, status, time_taken
```

Where

- **ip** is the IP address of the client who made the request
- **timestamp** is the date and time of the request in ISO 8601 format, YYYY-MM-DDT HH:MM:SS
- **method** is either GET or POST
- **path** is the path requested by the client
- **protocol** is the HTTP protocol used to send to the client, usually HTTP/1.1
- **status** is the `http://en.wikipedia.org/wiki/List_of_HTTP_status_codes`
- **time_taken** is the amount of time the server took to process the request, in seconds, not including upload/download time.

6.6.5 License Misconceptions

Some people have criticized web2py for the two reasons below and I would like to address them here because they are wrong and perhaps you can help me debunk these myths:

1. web2py is GPL2 therefore web2py apps must be GPL2. **This is NOT TRUE.**

You can distribute your apps under any license you like, even in closed source as long they are distinct from web2py itself, even if they need web2py to run. If you build an application that incorporates some of the web2py modules or a modification of the web2py modules in such a way that it becomes one with the framework itself or in such a way that it does not require the framework anymore than, yes, your app must be GPL2.

2. web2py requires that you use the web interface for development. **This is NOT TRUE**

All apps are under the web2py/applications folder which has the same tree structure as the admin interface. If you prefer to use a shell or have a favorite editor, you can develop web2py apps using those.

6.6.6 Authentication and single sign-on via CAS

Central Authentication Service (CAS) is a protocol for single sign-on. web2py includes libraries for both Conumers of the service (clients who need authentication) and Providers (server who provide the authentication service).

Read more: `https://mdp.cti.depaul.edu/cas`

6.6.7 sending SMS from web2py

from `http://www.aspsms.com/examples/python/`

```
def send_sms(recipient,text,userkey,password,host="xml1.
    aspsms.com",port=5061,action="/xmlsvr.asp"):
     import socket, cgi
     content="""<?xml version="1.0" encoding="ISO-8859-1"?>
<aspsms>
<Userkey>%s</Userkey>
<Password>%s</Password>
<Originator>"%s</Originator>
<Recipient>
```

```
<PhoneNumber>%s</PhoneNumber>
</Recipient>
<MessageData>%s</MessageData>
<Action>SendTextSMS</Action>
</aspsms>""" % (userkey,password,originator,recipient,cgi.
    escape(text))
     length=len(content)
     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
     s.connect((host.port))
     s.send("POST %s HTTP/1.0\r\n",action)
     s.send("Content-Type: text/xml\r\n")
     s.send("Content-Length: "+str(length)+"\r\n\r\n")
     s.send(CONTENT)
     datarecv=s.recv(1024)
     reply=str(datarecv)
     s.close()
     return reply
```

The aspsms service requires registration and it is not free. It is not secure either since transmission in not encrypted.

6.6.8 Using IPython

Since version 1.26, in web2py/ run

```
python2.5 web2py -S [application]
```

If ipython is installed you will have an ipython shell in the web2py [application]. If [application] does not exist, it is created.

You may also want to read Michael's blog `http://michaelangela.wordpress.com/2008/03/08/interactive-web2py-testing/`

6.6.9 upgrade core applications

Question:

When running from source, is the best way to update web2py by downloading, extracting, executing it in the source directory to unpack all the apps, etc., then copying the unpacked files over the web2py directory? Or unpacking the applications, etc., is not necessary?

Answer:

You do not need to unpack the applications manually. you just do

```
python web2py.y --upgrade yes
```

6.6.10 Authentication

Try CAS: `https://mdp.cti.depaul.edu/cas`

6.6.11 Web Hosting

if you have any experience with this please expand this page

web2py run on any hosting service that provides python2.5, even if they say they do not support it.

there are many possible configurations and you have to choose the most appropriate:

- apache + mod_proxy (easiest way)
- apache + mod_rewrite
- apache + mod_wsgi (best way if you understand wsgi)
- lighttpd + fcgi (fastest desployment configuration)

apache + mod_proxy

- Start web2py from its own folder with (nohup will avoid that it does when you logout)
  nohup python2.5 web2py.py -p 8000 -a yourpassword &
- configure apache to proxy port 8000. As an example here is my apache configuration `http://mdp.cti.depaul.edu/AlterEgo/default/show/38`.

apache + mod_rewite

- Start web2py from its own folder with (nohup will avoid that it does when you logout)
  nohup python2.5 web2py.py -i yourip -p 8000 -a yourpassword &
- configure apache to rewrite the request and use port 8000

apache + mod_wsgi

Read the wsgihandler.py file in the web2py folder

lightpd + fcgi

Read the fcgihandler.py file in the web2py folder

webfaction exception:

webfaction does not allow editing the apache config file but it requires going through their web based GUI. The easiest way to proceed it to use their GUI an empty cherrypy 3.0.0 project. After you create the project, use the shell to locate the newly created application folder. You will find a site.py that starts an empty cherry app (note that it contains an assigned port number). Edit this file as follows:

```
from gluon.main import HttpServer
PORT=3221
PASSWORD='whatever' server=HttpServer(ip='127.0.0.1',port=
    PORT,password=PASSWORD,pid_filename='pid')
server.start()
```

then copy web2py/* into the cherrypy app folder.

You may also want to read the experimental web2py installer for webfaction: `http://forum.webfaction.com/viewtopic.php?id=1346`

Let us know if you have trouble.

6.6.12 security and admin

In general it is not a good idea to expose publicly admin and yourapp/appadmin unless they go over HTTPS and you enable secure cookies with

```
response.cookies[response.session_id_name]['secure']=True
```

This is true for web2py and any other web application: **If you do not want your passwords to transmit unencrypted, your sesion cookies should not either!**

In fact, by default, for security, web2py admin does not work if the client is not localhost.

An easy way to setup a secure production environment on a server (@serveraddress) is to:

- start two instances of web2py:
  ```
  nohup python2.5 web2py -p 8000 -i 127.0.0.1 -a " &
  nohup python2.5 web2py -p 8001 -i 127.0.0.1 -a password &
  ```
- use apache mod_proxy to redirect port 80 to port 8000 (there will be no admin because no password) this is the public site
- from your client machine connect to the second using a ssh tunnel:
  ```
  ssh -L 8001:127.0.0.1:8001 username@serveraddress
  ```
- connect to 127.0.0.1:8001 on the local computer to access the admin of the remote (serveraddress) computer.

All communication via port 8001 will be accessible to you only and encrypted.

6.6.13 Using memcache

web2py has a built-in caching mechanism but, since version 1.26, it also includes the memcache client `ftp://ftp.tummy.com/pub/python-memcached/`

Assuming memcache is running on locahost on port 11211, somewhere in your code, before you use it, do

```
from gluon.contrib.memcache import MemcacheClient

memcache_servers=['127.0.0.1:11211']

cache.mem=MemcacheClient(request,memcache_servers)
```

then use cache.mem instead of cache.ram and cache.disk according to the caching examples.

6.6.14 use an editor of your choice via the admin interface

yes you can if you use firefox:

`https://addons.mozilla.org/en-US/firefox/addon/4125`

thanks to Zoom.Quiet for pointing us to this add-on.

6.6.15 custom error pages

The built-in ticketing mechanism is a last resort error. You should catch exceptions in your own app. Anyway, the following decorator may be helpful:

```
def onerror(function):
  def __onerror__(*a,**b):
    try:
        return function(*a,**b)
    except HTTP, e:
        import os
        status=int(e.status.split(' ')[0])
        filename=os.path.join(request.folder,'views/onerror%i.
            html'%status)
        if os.access(filename,os.R_OK):
            e.body=open(filename,'r').read()
        raise e
    except Exception:
        import os, gluon.restricted
        e=gluon.restricted.RestrictedError(function.__name__)
        SQLDB.close_all_instances(SQLDB.rollback)
        ticket=e.log(request)
        filename=os.path.join(request.folder,'views/onerror.html
            ')
        if os.access(filename,os.R_OK):
            body=open(filename,'r').read()
        else:
            body="""<html><body><h1>Internal error</h1>Ticket
                issued:
            <a href="/admin/default/ticket/%(ticket)s" target="
                _blank">%(ticket)s</a></body></html>"""
        body=body % dict(ticket=ticket)
        raise HTTP(200,body=body)
  return __onerror__
```

There is how you use:

Put the function above in a place where it is accessible (a model file for example)

Consider any function in a model a controller or a view that you where you want to use custom error pages and do

```
@onerror
def index():
     ...
```

now if the index controller function raises an HTTP(400,...) it will be rendered by views/onerror400.html

if the index controller function raises any other exception, it will be rendered by views/onerror.html

Notice that onerrorXXX.html and onerror.html must be static HTML files and are not processed as templates.

onerror.HTML can contain "%(ticket)s" in order to refer to the ticket being issued.

Here is an example of an error400.html page

```
<html><body><h1>Internal Error</h1></body></html>
```

Here is an example of an error.html page

```
<html><body><h1>Hey dude, here is your ticket: %(ticket)s</h1
   ></body></html>
```

6.6.16 web2py as a windows service

Thank you Limodou for all your work on this

web2py supports windows service install/start/stop now in source version. If you want to use it, you should create an options.py (for default usage) with startup parameters or specify an option file with -L parameter in the command line.

The options.py could be like this:

```
import socket, os
ip = socket.gethostname()
port = 80
password = '<recycle>'
pid_filename = 'httpserver.pid'
log_filename = 'httpserver.log'
ssl_certificate = ''
ssl_private_key = ''
numthreads = 10
```

```
server_name = socket.gethostname()
request_queue_size = 5
timeout = 10
shutdown_timeout = 5
folder = os.getcwd()
```

And you don't need to create options.py manually, there is already an options_std.py in web2py folder, so you can copy it and name it options.py or just specify it in command line via -L parameter.

then run:

```
python2.5 web2py.py -W install
python2.5 web2py.py -W start -L options.py
python2.5 web2py.py -W stop
```

6.6.17 web server benchmarks

web2py uses the cherrypy wsgiserver. We did not run benchmarks ourselves but you can find some benchmarks here:

`http://www.aminus.org/blogs/index.php/fumanchu/2006/12/23/cherrypy_3_has_fastest_wsgi_server_yet`

Even if the author does not say which system he run the benchmark on, the results do look impressive and justify our choice.

6.6.18 reading existing sql tables and CRUD

If you have an existing model in the form of SQL CREATE statements you may find this file useful:

`http://mdp.cti.depaul.edu/AlterEgo/default/download/document.file.075074495458.py/auto.py`

Given a file somefile.sql that contains some table definitions in SQL, run

```
python auto.py somefile.sql
```

and it will make web2py models and Create Read Update Delete (CRUD) controllers for all your tables.

You will have to manually cut and paste the output in you app, edit the models to fix reference fields(incorrectly rendered as integer) and add validators.

Appendix

6.7 Quick Examples

6.7.1 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. Try it here: hello1

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 Try it here: hello2

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). Try it here: hello3

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. Try it here: hello4

Example 5

In controller: simple_examples.py

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

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.

Try it here: hello5

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. Try it here: status

Example 7

In controller: simple_examples.py

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

You can do redirect. Try it here: redirectme

Example 8

In controller: simple_examples.py

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

You can raise HTTP exceptions to return an error page. Try it here: raisehttp

Example 9

In controller: simple_examples.py

```
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. Try it here: raiseexception

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, it is not
       supposed to run!");'
```

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. Try it here: servejs

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. Try it here: makejson

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. Try it here: makertf

Example 13

In controller: simple_examples.py

```
def rss_aggregator():
    import datetime
    import gluon.contrib.rss2 as rss2
    import gluon.contrib.feedparser as feedparser
    d = feedparser.parse("http://rss.slashdot.org/Slashdot/
       slashdot/to")

    rss = rss2.RSS2(title=d.channel.title,
       link = d.channel.link,
       description = d.channel.description,
       lastBuildDate = datetime.datetime.now(),
       items = [
          rss2.RSSItem(
            title = entry.title,
            link = entry.link,
            description = entry.description,
            guid = rss2.Guid('unkown'),
            pubDate = datetime.datetime.now()) for entry in d.
               entries]
       )
    response.headers['Content-Type']='application/rss+xml'
    return rss2.dumps(rss)
```

web2py includes gluon.contrib.rss2, developed by Dalke Scientific Software, which generates RSS2 feeds, and gluon.contrib.feedparser, developed by Mark Pilgrim, which collects RSS and ATOM feeds. The above controller collects a slashdot feed and makes new one. Try it here: rss_aggregator

Example 14

In controller: simple_examples.py

```
from gluon.contrib.markdown import WIKI

def ajaxwiki():
    form=FORM(TEXTAREA(_id='text'),INPUT(_type='button',_value='
       markdown',
              _onclick="ajax('ajaxwiki_onclick',['text'],'html')
                 "))
    return dict(form=form,html=DIV(_id='html'))

def ajaxwiki_onclick():
```

```
9     return WIKI(request.vars.text).xml()
```

web2py also includes gluon.contrib.markdown (markdown2) which converts WIKI markup to HTML following this syntax. In this example we added a fancy ajax effect. Try it here: ajaxwiki

6.7.2 Session Examples

Example 15

In controller: session_examples.py

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

and view: session_examples/counter.html

```
1 {{extend 'layout.html'}}
2 <h1>session counter</h1>
3
4 <h2>{{for i in range(counter):}}{{=i}}...{{pass}}</h2>
5
6 <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. Try it here: counter

6.7.3 Template Examples

Example 16

In controller: template_examples.py

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

and view: template_examples/variables.html

```
1 {{extend 'layout.html'}}
2 <h1>Your variables</h1>
3 <h2>a={{=a}}</h2>
4 <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)). Try it here: variables

Example 17

In controller: template_examples.py

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

and view: template_examples/test_for.html

```
{{extend 'layout.html'}}
<h1>For loop</h1>

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

You can do for and while loops. Try it here: test_for

Example 18

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. Try it here: test_if

Example 19

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. Try it here: test_try

Example 20

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. Try it here: test_def

Example 21

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. Try it here: escape

Example 22

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. Try it here: xml

Example 23

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. Try it here: beautify

6.7.4 Layout Examples

Example 24

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. Try it here: civilized

Example 25

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. Try it here: slick

Example 26

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. Try it here: basic

6.7.5 Form Examples

Example 27

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"
    elif form.errors:
        response.flash="form is invalid"
    else:
        response.flash="please fill the form"
    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. Try it here: form

6.7.6 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 28

in model: dba.py

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

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

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'))

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==dba.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 29

In controller: database_examples.py

```
response.menu=[['Register User',False,URL(r=request,f='
   register_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():
    form=SQLFORM(dba.users)
    if form.accepts(request.vars,session):
        response.flash='new record inserted'
    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. Try it here: register_user

Example 30

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')
    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. Try it here: register_dog

Example 31

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. Try it here: register_product

Example 32

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):
        if len(dba(dba.users.id==form.vars.buyer_id).select())
           ==0:
            form.errors.buyer_id="buyer not in database"
        if len(dba(dba.products.id==form.vars.product_id).select
           ())==0:
            form.errors.product_id="product not in database"
        if len(form.errors)==0:
            purchases=dba((dba.purchases.buyer_id==form.vars.
               buyer_id)&
                          (dba.purchases.product_id==form.vars.
                             product_id)).select()
            if len(purchases)>0:
                purchases[0].update_record(quantity=purchases
                   [0].quantity+form.vars.quantity)
            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!"
```

```
22      records=dba(purchased).select(dba.users.name,dba.purchases.
            quantity,dba.products.name)
23      return dict(form=form,records=SQLTABLE(records),vars=form.
            vars,vars2=request.vars)
```

and view: database_examples/buy.html

```
1  {{extend 'layout_examples/layout_civilized.html'}}
2    <h1>Purchase form</h1>
3    {{=form}}
4    [ {{=A('reset purchased',_href=URL(r=request,f='
         reset_purchased'))}} |
5      {{=A('delete purchased',_href=URL(r=request,f='
           delete_purchased'))}} ]<br/>
6    <h2>Current purchases (SQL JOIN!)</h2>
7  <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. Try it here: buy

Example 33

In controller: database_examples.py

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

Try it here: delete_purchased

Example 34

In controller: database_examples.py

```
1  def reset_purchased():
2      dba(dba.purchases.id>0).update(quantity=0)
3      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.) Try it here: reset_purchased

Example 35

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.

6.7.7 Cache Examples

Example 36

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. Try it here: cache_in_ram

Example 37

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. Try it here: cache_on_disk

Example 38

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. Try it here: cache_in_ram_and_disk

Example 39

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. Try it here: cache_controller_in_ram

Example 40

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. Try it here: cache_controller_on_disk

Example 41

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! Try it here: cache_controller_and_view

Example 42

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. Try it here: cache_db_select

Ajax Examples

Example 43

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
    appear 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. Try it here: index

Example 44

In controller: ajax_examples.py

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

Try it here: flash

Example 45

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>
```

Try it here: fade

Testing Examples

Example 46

Using the Python doctest notation it is possible to write tests for all controller functions. Tests are then run via the administrative interface which generates a report. Here is an example of a test in the code:

```
def index():
    '''
    This is a docstring. The following 3 lines are a doctest:
    >>> request.vars.name='Max'
    >>> index()
    {'name': 'Max'}
    '''
    return dict(name=request.vars.name)
```

Large Files Example

Example 47

It is very easy in web2py to stream large files. Here is an example of a controller that does so:

```
def streamer():
    return response.stream(open('largefile.mpg4','rb'),
       chunk_size=4096)
```

XML-RPC Example

Example 48

Web2py has native support for the XMLRPC protocol. Below is a controller function "handler" that exposes two functions, "add" and "sub" via XML-RPC. The controller "tester" executes the two function remotely via xmlrpc.

```
1 def add(a,b): return a+b
2 def sub(a,b): return a-b
3
4 def handler(): return response.xmlrpc(request,[add,sub])
5
6 def tester():
7     import xmlrpclib
8     server=xmlrpclib.ServerProxy('http://hostname:port/app/
          controller/handler')
9     return str(server.add(3,4)+server.sub(3,4))
```