web2py

Enterprise Web Framework

2nd Edition by Massimo Di Pierro

WEB2PY

Enterprise Web Framework / 2nd Ed.

Massimo Di Pierro

Massimo Di Pierro
School of Computing
DePaul University
243 S Wabash Ave
Chicago, IL 60604 (USA)
Email: mdipierro@cs.depaul.edu

Library of Congress Cataloging-in-Publication Data:

WEB2PY: Enterprise Web Framework
Printed in the United States of America.

to my family

CONTENTS

Preface

I am guilty! After publicly complaining about the existence of too many Python based web frameworks, after praising the merits of Django, Pylons, TurboGears, CherryPy, and web.py, after having used them professionally and taught them in University level courses, I could not resist and created one more: WEB2PY.

Why did I commit such a crime? I did it because I felt trapped by existing choices and tempted by the beautiful features of the Python language. It all started with the need to convince my father to move away from Visual Basic and embrace Python as a development language for the Web. At the same time I was teaching a course on Python and Django at DePaul University. These two experiences made me realize how the beautiful features of those systems were hidden behind a steep learning curve. At the University for example we teach introductory programming using languages like Java and C++ but we do not get into networking issues until later courses. In many Universities students can graduate in Computer Science without ever seeing a Unix Bash Shell or editing an Apache configuration file. And yet these days to be an effective web developer you must know shell scripting, Apache, SQL, HTML, CSS, JavaScript, and Ajax. Knowing how to program in one

language is not enough to understand the intricacy and subtleties of the APIs exposed by the existing frameworks. Not to mention security.

WEB2PY started with the goal to drastically reduce the learning curve, incorporating everything needed into a single tool that is accessible via the web browser, collapsing the API to a minimum (only 12 core objects and functions), delegating all the security issues to the framework, and forcing developers to follow modern software engineering practices.

Most of the development work was done in the summer of 2007 while I was on vacation. Since WEB2PY was released many people have contributed by submitting patches to fix bugs and to add features. WEB2PY has evolved steadily since and yet it never broke backward compatibility. In fact, WEB2PY has a top-down design vs the bottom-up design of other frameworks. It is not built by adding layer upon layer. It is built from the user perspective and it has been constantly optimized inside in order to become faster and leaner, while always keeping backward compatibility. I am happy to say that today WEB2PY is one of the fastest web frameworks and also one of the the smallest (the core libraries including the Database Abstraction Layer, the template language, and all the helpers amounts to about 300KB, the entire source code including sample applications and images amounts to less than 2.0MB).

Yes, I am guilty, but so are the growing number of users and contributors. Nevertheless, I feel, I am no more guilty than the creators of the other frameworks I have mentioned.

Finally, I would like to point out, I have already paid a price for my crime, since I have been condemned to spend my 2008 summer vacation writing this book and my 2009 summer vacations revising it.

This second edition describes many features added after the release of the first edition, including CRUD, Access Control, and Services.

I hope you, dear reader, understand I have done it for you: to free you from current web programming difficulties, and to allow you to express yourself more and better on the Web.

CHAPTER 1

INTRODUCTION

WEB2PY [1] is a free, open-source web framework for agile development of secure database-driven web applications; it is written in Python[2] and programmable in Python. WEB2PY is a full-stack framework, meaning that it contains all the components you need to build fully functional web applications.

WEB2PY is designed to guide a web developer to follow good software engineering practices, such as using the Model View Controller (MVC) pattern. WEB2PY separates the data representation (the model) from the data presentation (the view) and also from the application logic and workflow (the controller). WEB2PY provides libraries to help the developer design, implement, and test each of these three parts separately, and makes them work together.

WEB2PY is built for security. This means that it automatically addresses many of the issues that can lead to security vulnerabilities, by following well established practices. For example, it validates all input (to prevent injections), escapes all output (to prevent cross-site scripting), renames uploaded files (to prevent directory traversal attacks), and stores all session information

server side. WEB2PY leaves little choice to application developers in matters related to security.

WEB2PY includes a Database Abstraction Layer (DAL) that writes SQL [3] dynamically so that the developer does not have to. The DAL knows how to generate SQL transparently for SQLite [4], MySQL [6], PostgreSQL [5], MSSQL [7], FireBird [8], Oracle [9], IBM DB2 [10] and Informix [11]. The DAL can also generate function calls for Google BigTable when running on the Google App Engine (GAE) [12]. Once one or more database tables are defined, WEB2PY also generates a fully functional web-based database administration interface to access the database and the tables.

WEB2PY differs from other web frameworks in that 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 that can run Python (Windows, Windows CE, Mac OS X, iPhone, and Unix/Linux), and the development, deployment, and maintenance phases for the applications can be done via a local or remote web interface. WEB2PY runs with CPython (the C implementation) and/or Jython (the Java implementation), versions 2.4, 2.5 and 2.6 although "officially" only support 2.5 else we cannot guarantee backward compatibility for applications.

WEB2PY provides a ticketing system. If an error occurs, a ticket is issued to the user, 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 applications do not explicitly contain WEB2PY source code, they are not considered "derivative works". WEB2PY also allows the developer to bytecode-compile applications and distribute them as closed source, although they will require WEB2PY to run. The WEB2PY license includes an exception that allows web developers to ship their products with original pre-compiled WEB2PY binaries, without the accompanying source code.

Another feature of WEB2PY, is that we, its developers, commit to maintain backward compatibility in future versions. We have done so since the first release of WEB2PY in October, 2007. New features have been added and bugs have been fixed, but if a program worked with WEB2PY 1.0, that program will still work today.

Here are some examples of WEB2PY statements that illustrate its power and simplicity. The following code:

```
db.define_table('person',
   Field('name', 'string'),
   Field('image', 'upload'))
```

creates a database table called "person" with two fields: "name", a string; and "image", something that needs to be uploaded (the actual image). If the table already exists but does not match this definition, it is altered appropriately.

Given the table defined above, the following code:

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

creates an insert form for this table that allows users to upload images.

The following statement:

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

validates a submitted form, renames the uploaded image in a secure way, stores the image in a file, inserts the corresponding record in the database, prevents double submission, and eventually modifies the form itself by adding error messages if the data submitted by the user does not pass validation.

1.1 Principles

Python programming typically follows these basic principles:

- Don't repeat yourself (DRY).
- 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 in web application development (creating and processing forms, managing sessions, cookies, errors, etc.).

WEB2PY differs from other frameworks with regard to the third principle, which sometimes conflicts with the other two. In particular, WEB2PY automatically imports its own modules and instantiates its 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 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, saves time and prevents mistakes, thus following the spirit of "don't 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, those modules must be imported explicitly, as in any other Python program.

1.2 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 are:

- Generating HTML [13, 14] programmatically and embedding HTML as strings into computer code.
- Embedding pieces of code into HTML pages.

The first model is the one followed, for example, by early CGI scripts. The second model is followed, for example, by PHP [15] (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 program that, when executed, retrieves data from a database and returns an HTML page showing the selected records:

```
<html><body><h1>Records</h1><?
  mysql_connect(localhost,username,password);
  @mysql_select_db(database) or die( "Unable to select database");
  $query="SELECT * FROM contacts";
  $result=mysql_query($query);
  mysql_close();
  $i=0;
  while ($i < mysql_numrows($result)) {
    $name=mysql_result($result,$i,"name");
    $phone=mysql_result($result,$i,"phone");
    echo "<b>$name</b><br>Phone:$phone<br /><br /><hr /><br />";
    $i++;
  }
?></body></html>
```

The problem with this approach is that code is embedded into HTML, but this very same code also needs to generate additional HTML and to generate SQL statements to query the database, entangling multiple layers of the application and making it difficult to read and maintain. The situation is even worse for Ajax applications, and the complexity grows with the number of pages (files) that make up the application.

The functionality of the above example can be expressed in WEB2PY with two lines of Python code:

```
def index():
    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`[1] is queried by the `select` command; finally, everything is serialized into HTML.

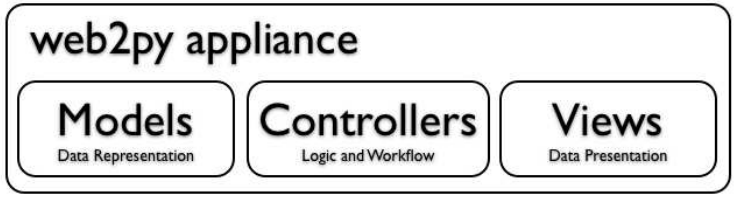
This is just one example of the power of WEB2PY and its built-in libraries. WEB2PY does even more for the developer by automatically handling cookies, sessions, creation of database tables, database modifications, form validation, SQL injection prevention, cross-site scripting (XSS) prevention, and many other indispensable web application tasks.

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

WEB2PY is a full-stack framework. Almost all of 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 Database Abstraction Layer (DAL) or the template language can be used independently of the WEB2PY framework by importing `gluon.sql` or `gluon.template` into your own Python applications. `gluon` is the name of the WEB2PY folder that contains system libraries. Some WEB2PY libraries, such as 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 template languages and DALs, but they will not be as tightly integrated as the original components.

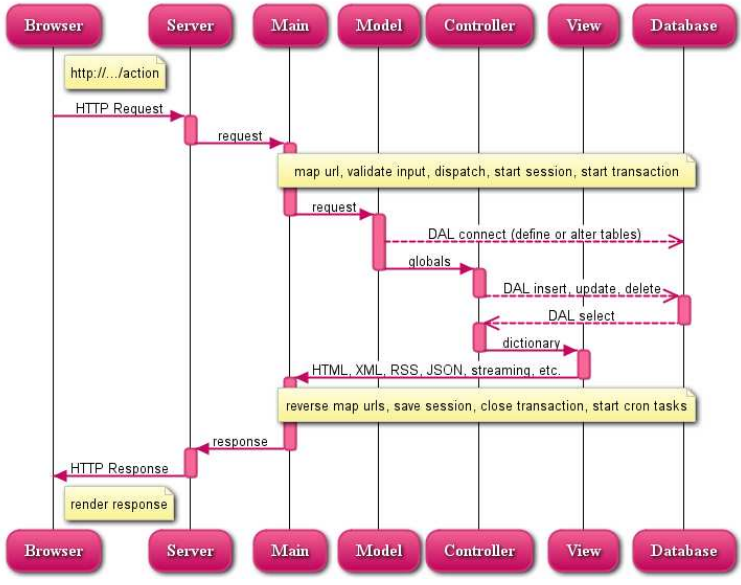
1.3 Model-View-Controller

WEB2PY forces the developer to separate data representation (the model), data presentation (the view) and the application workflow (the controller). Let's consider again the previous example and see how to build a WEB2PY application around it.



[1]There is nothing special about the name `db`; it is just a variable holding your database connection.

The typical workflow of a request in WEB2PY is described in the following diagram:



In the diagram:

- The Server can be the WEB2PY built-in web server or a third-party server, such as Apache. The Server handles multi-threading.
- Main is the main WEB2PY WSGI application. It performs all common tasks and wraps user applications. It deals with cookies, sessions, transactions, url mapping and reverse mapping, dispatching (deciding which function to call based on the URL). It can serve and stream static files if the web server is not doing it already.
- The Models, Views and Controller components make up the user application. There can be multiple applications hosted in the same WEB2PY instance.
- The dashed arrows represent communication with the database engine (or engines). The database queries can be written in raw SQL (discouraged) or by using the WEB2PY Database Abstraction Layer (recommended), so that that WEB2PY application code is not dependent on the specific database engine.
- The dispatcher maps the requested URL into a function call in the controller. The output of the function can be a string or a dictionary

of symbols (a hash table). The data in the dictionary is rendered by a view. If the visitor requests an HTML page (the default), the dictionary is rendered into an HTML page. If the visitor requests the same page in XML, WEB2PY tries to find a view that can render the dictionary in XML. The developer can create views to render pages in any of the already supported protocols (HTML, XML, JSON, RSS, CSV, RTF) or additional custom protocols.

- All calls are wrapped into a transaction, and any uncaught exception causes the transaction to roll back. If the request succeeds, the transaction is committed.

- WEB2PY also handles sessions and session cookies automatically, and when a transaction is committed, the session is also stored.

- It is possible to register recurrent tasks (cron) to run at scheduled times and/or after the completion of certain actions. In this way it is possible to run long and compute-intensive tasks in the background without slowing down navigation.

Here is a minimal and complete MVC application consisting of three files:

- "db.py" is the model:

```
db = DAL('sqlite://storage.sqlite')
db.define_table('contacts',
   Field('name'),
   Field('phone'))
```

It connects to the database (in this example a SQLite database stored in the `storage.sqlite` file) and defines a table called `contacts`. If the table does not exist, WEB2PY creates it and, transparently and in the background, generates SQL code in the appropriate SQL dialect for the specific database engine used. The developer can see the generated SQL but does not need to change the code if the database back-end, which defaults to SQLite, is replaced with MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Informix, or Google Big Tables in the Google App Engine.

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. It is called `appadmin`.

- "default.py" is the controller:

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

In WEB2PY, URLs are mapped to Python modules and function calls. In this case, the controller contains a single function (or "action") called `contacts`. An action may return a string (the returned website) or a Python dictionary (a set of key:value pairs). If the function returns a dictionary, it is passed to a view with the same name as the controller/function, which in turn renders it. In this example, the function `contacts` performs a database `select` and returns the resulting records as a value associated with 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 (action) is executed. The purpose of this view is to render the variables in the returned dictionary `records=...` into HTML. The view file is written in HTML, but it embeds Python code delimited by the special {{ and }} delimiters. This is quite different from 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 all WEB2PY applications. The layout file can easily be modified or replaced.

1.4 Why web2py

WEB2PY is one of many web application frameworks, but it has compelling and unique features. WEB2PY was originally developed as a teaching tool, with the following primary motivations:

- Easy for users to learn server-side web development without compromising on functionality. For this reason WEB2PY requires no installation, no configuration, has no dependencies[2], and exposes most of its functionality via a web interface.
- WEB2PY has been stable from day one because it follows a top-down design; i.e., its API was designed before it was implemented. Even

[2]except for the source code distribution, which requires Python 2.5 and its standard library modules

as new functionality has been added, WEB2PY has never broken backwards compatibility, and it will not break compatibility when additional functionality is added in the future.

- WEB2PY proactively addresses the most important security issues that plague many modern web applications, as determined by OWASP[19] below.
- WEB2PY is light. Its core libraries, including the Database Abstraction Layer, the template language, and all the helpers amount to 300KB. The entire source code including sample applications and images amounts to 2.0MB.
- WEB2PY has a small footprint and is very fast. It uses the CherryPy [16] WSGI-compliant[3] web server that is 30% faster than Apache with mod_proxy and four times faster than the Paste http server. Our tests also indicate that, on an average PC, it serves an average dynamic page without database access in about 10ms. The DAL has very low overhead, typically less than 3%.

1.5 Security

The Open Web Application Security Project[19] (OWASP) is a free and open worldwide community focused on improving the security of application software.

OWASP has listed the top ten security issues that put web applications at risk. That list is reproduced here, along with a description of how each issue is addressed by WEB2PY:

- "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 scripts 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, preventing XSS.
- "Injection Flaws: Injection flaws, particularly SQL injection, are common in web applications. Injection occurs when user-supplied data is

[3]The Web Server Gateway Interface [17, 18] (WSGI) is an emerging Python standard for communication between a web server and Python applications.

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 a Database Abstraction Layer that makes SQL injection impossible. Normally, SQL statements are not written by the developer. Instead, SQL is generated dynamically by the DAL, ensuring that 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."
  WEB2PY allows only exposed functions to be executed, preventing malicious file execution. Imported functions are never exposed; only actions are exposed. 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 that 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 storing only the session id in a browser-side 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 includes a ticketing system. No error can result in code being exposed to the users. All errors are logged and a ticket is issued to the

user that allows error tracking. Errors and source code are accessible only 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 administrator authentication, and it manages sessions independently for each application. The administrative interface also forces the use of secure session cookies when the client is not "localhost". For applications, it includes a powerful Role Based Access Control API.

- "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 includes the SSL-enabled [20] CherryPy WSGI server, but it can also use Apache or Lighttpd and mod_ssl to provide SSL 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 and authorization. The included Role Based Access Control API allow developers to restcrict access to any function based on login, group membership or group based permissions. The permissions are very granular and can be combined with CRUD to allow, for example, to give access to specific tables and/or records.

1.6 In the box

You can download WEB2PY from the official web site:

```
http://www.web2py.com
```

WEB2PY is composed of the following components:

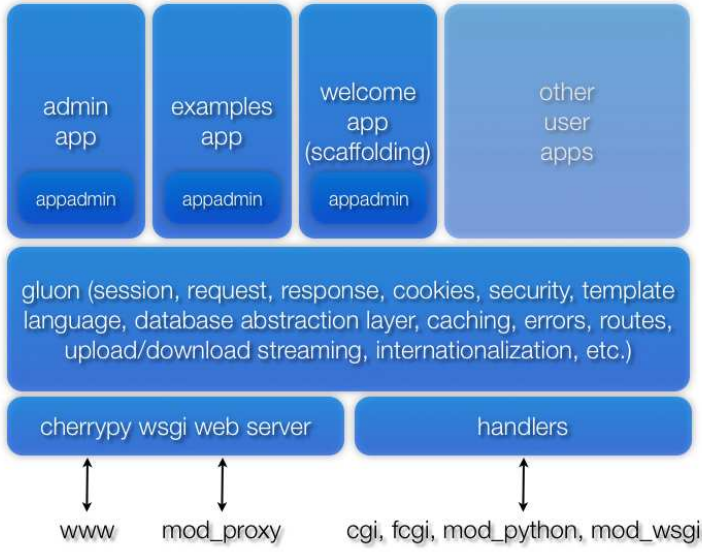
- **libraries**: provide core functionality of WEB2PY and are accessible programmatically.
- **web server**: the CherryPy WSGI web server.
- the **admin** application: used to create, design, and manage other WEB2PY applications. **admin** provide a complete web-based Integrated Development Environment (IDE) for building WEB2PY applications. It also includes other functionality, such as web-based testing and a web-based shell.
- the **examples** application: contains documentation and interactive examples. **examples** is a clone of the official WEB2PY web site, and includes epydoc and Sphinx documentation.
- the **welcome** application: the basic scaffolding template for any other application. By default it includes a pure CSS cascading menu and user authentication (discussed in Chapter 8).

WEB2PY is distributed in source code and binary form for Microsoft Windows and for Mac OS X.

The source code distribution can be used in any platform where Python or Jython run, and includes the above-mentioned components. To run the source code, you need Python 2.5 pre-installed on the system. You also need one of the supported database engines installed. For testing and light-demand applications, you can use the SQLite database, included with Python 2.5.

The binary versions of WEB2PY (for Windows and Mac OS X) include a Python 2.5 interpreter and the SQLite database. Technically, these two are not components of WEB2PY. Including them in the binary distributions enables you to run WEB2PY out of the box.

The following image depicts the overall WEB2PY structure:



1.7 License

WEB2PY is licensed under the GPL version 2 License. The full text of the license if available in ref. [30].

The license includes but it is not limited to the following articles:

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

[...]

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License.

[...]

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER

PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

- WEB2PY includes some third-party code (for example the Python interpreter, the CherryPy web server, and some JavaScript libraries). Their respective authors and licenses are acknowledged in the official website [1] and in the code itself.
- Applications developed with WEB2PY, as long as they do not include WEB2PY source code, are not considered derivative works. This means they are not bound by the GPLv2 license, and you can distribute the applications you developed under any license you choose, including a closed-source and/or commercial license.

1.8 License Commercial Exception

The WEB2PY license also includes a commercial exception:

You may distribute an application you developed with WEB2PY together with an unmodified official binary distribution of WEB2PY, as downloaded from the official website[1], as long as you make it clear in the license of your application which files belong to the application and which files belong to WEB2PY.

1.9 Acknowledgments

WEB2PY was originally developed by and copyrighted by Massimo Di Pierro. The first version (1.0) was released in October, 2007. Since then it has been adopted by many users, some of whom have also contributed bug reports, testing, debugging, patches, and proofreading of this book.

Some of the major contributors are, in alphabetical order by first name:

Alexandre Andrade, Alexey Nezhdanov (GAE and database performance), Alvaro Justen (dynamical translations), Andre Berthiaume, Andre Bossard, Attila Csipa (cron job), Bill Ferrett (modular DAL design), Boris Manojlovic (Ajax edit), Carsten Haese (Informix), Chris Baron, Christopher Smiga (Informix), Clifford John Lazell (tester and JS), David H. Lee (OpenID), Denes Lengyel (validators, DB2 support), Douglas Soares de Andrade (2.4 and 2.6 compliance, docstrings), Felipe Barousse, Fran Boon (authorization and authentication), Francisco Gama (bug fixing), Fred Yankowski (XHTML compliance), Gabriele Carrettoni, Graham Dumpleton, Gregor Jovanovich, Hans Christian v. Stockhausen (OpenID), Hans Donner (GAE support, Google login, widgets, Sphinx documentation), Ivan Valev, Joe Barnhart, Jonathan Benn (validators and tests), Jonathan Lundell, Jose Jachuf (Firebird support), Kacper Krupa, Kyle Smith (JavaScript), Limodou (winservice), Lucas Geiger,Marcel Leuthi (Oracle support), Mark Larsen (taskbar widget), Mark Moore (databases and daemon scripts), Markus Gritsch (bug fixing), Martin Hufsky (expressions in DAL), Mateusz Banach (stickers, validators, content-type), Michael Willis (shell), Milan Andric, Minor Gordon, Nathan Freeze (admin design, validators), Niall Sweeny (MSSQL support), Niccolo Polo (epydoc), Nicolas Bruxer (memcache support), Ondrej Such (MSSQL support), Pai (internationalization), Phyo Arkar Lwin (web hosting and Jython tester), Ricardo Cardenes, Richard Gordon, Richard Baron Penman, Robin Bhattacharyya (Google App Engine support), Roman Goldmann, Ruijun Luo (windows binary), Scott Santarromana, Sergey Podlesnyi (Oracle and migrations tester), Shane McChesney, Sharriff Aina (tester and PyAMF integration), Sterling Hankins (tester), Stuart Rackham (MSSQL support), Telman Yusupov (Oracle support), Tim Farrell, Tim Michelsen (Sphinx documentation), Timothy Farrell (Python 2.6 compliance, windows support), Tito Garrido, Yair Eshel (internationalizaiton), Yarko Tymciurak (design, Sphinx documentation), Ygao, Younghyun Jo (internationalization), Zoom Quiet

I am sure I forgot somebody, so I apologize.

I particularly thank Alvaro, Denes, Felipe, Graham, Jonathan, Hans, Kyle, Mark, Richard, Robin, Roman, Scott, Shane, Sharriff, Sterling, Stuart and Yarko for proofreading various chapters of this book. Their contribution was invaluable. If you find any errors in this book, they are exclusively my fault,

probably introduced by a last-minute edit. I also thank Ryan Steffen of Wiley Custom Learning Solutions for help with publishing the first edition of this book.

WEB2PY contains code from the following authors, whom I would like to thank:

Guido van Rossum for Python [2], Peter Hunt, Richard Gordon, Robert Brewer for the CherryPy [21] web server, Christopher Dolivet for EditArea [22], Brian Kirchoff for nicEdit [23], Bob Ippolito for simplejson [24], Simon Cusack and Grant Edwards for pyRTF [25], Dalke Scientific Software for pyRSS2Gen [26], Mark Pilgrim for feedparser [27], Trent Mick for markdown2 [28], Allan Saddi for fcgi.py, Evan Martin for the Python memcache module [29], John Resig for jQuery [31].

The logo used on the cover of this book was designed by Peter Kirchner at Young Designers.

I thank Helmut Epp (provost of DePaul University), David Miller (Dean of the College of Computing and Digital Media of DePaul University), and Estia Eichten (Member of MetaCryption LLC), for their continuous trust and support.

Finally, I wish to thank my wife, Claudia, and my son, Marco, for putting up with me during the many hours I have spent developing WEB2PY, exchanging emails with users and collaborators, and writing this book. This book is dedicated to them.

1.10 About this Book

This book includes the following chapters, besides this introduction:

- Chapter 2 is a minimalist introduction to Python. It assumes knowledge of both procedural and object-oriented programming concepts such as loops, conditions, function calls and classes, and covers basic Python syntax. It also covers examples of Python modules that are used throughout the book. If you already know Python, you may skip Chapter 2.

- Chapter 3 shows how to start WEB2PY, discusses the administrative interface, and guides the reader through various examples of increasing complexity: an application that returns a string, a counter application, an image blog, and a full blown wiki application that allows image uploads and comments, provides authentication, authorization, web services and an RSS feed. While reading this chapter, you may need

to refer to Chapter 2 for general Python syntax and to the following chapters for a more detailed reference about the functions that are used.

- Chapter 4 covers more systematically the core structure and libraries: URL mapping, request, response, sessions, cacheint, CRON, internationalization and general workflow.
- Chapter 5 is a reference for the template language used to build views. It shows how to embed Python code into HTML, and demonstrates the use of helpers (objects that can generate HTML).
- Chapter 6 covers the Database Abstraction Layer, or DAL. The syntax of the DAL is presented through a series of examples.
- Chapter 7 covers forms, form validation and form processing. FORM is the low level helper for form building. SQLFORM is the high level form builder. In Chapter 7 we also discuss the new Create/Read/Update/Delete (CRUD) API.
- Chapter 8 covers authentication, authorization and the extensible Role-Based Access Control mechanism available in WEB2PY. Mail configuration and CAPTCHA are also discussed here, since they are used by authentication.
- Chapter 9 is about creating web services in WEB2PY. We provide examples of integration with the Google Web Toolkit via Pyjamas, and with Adobe Flash via PyAMF.
- Chapter 10 is about WEB2PY and jQuery recipes. WEB2PY is designed mainly for server-side programming, but it includes jQuery, since we have found it to be the best open-source JavaScript library available for effects and Ajax. In this chapter, we discuss how to effectively use jQuery with WEB2PY.
- Chapter 11 is about production deployment of WEB2PY applications. We mainly address three possible production scenarios: on a Linux web server or a set of servers (which we consider the main deployment alternative), running as a service on a Microsoft Windows environment, and deployment on the Google Applications Engine (GAE). In this chapter, we also discuss security and scalability issues.
- Chapter 12 contains a variety of other recipes to solve specific tasks, inlcuding upgrades, gecoding, pagination, Twitter API, and more.

This book only covers basic WEB2PY functionalities and the API that ships with WEB2PY. This book does not cover WEB2PY appliances, for

example KPAX, the WEB2PY Content Management System. The appliance for Central Authentication Service is briefly discussed in Chapter 8.

You can download WEB2PY appliances from the corresponding web site [33].

You can find additional topics discussed on AlterEgo [34], the interactive WEB2PY FAQ.

1.11 Elements of Style

Ref. [35] contains good style practices when programming with Python. You will find that WEB2PY does not always follow these rules. This is not because of omissions or negligence; it is our belief that the users of WEB2PY should follow these rules and we encourage it. We chose not to follow some of those rules when defining WEB2PY helper objects in order to minimize the probability of name conflict with objects defined by the user.

For example, the class that represents a `<div>` is called `DIV`, while according to the Python style reference it should have been called `Div`. We believe that, for this specific example that using an all-upper-case "DIV" is a more natural choice. Moreover, this approach leaves programmers free to create a class called "Div" if they choose to do so. Our syntax also maps naturally into the DOM notation of most browsers (including, for example, Firefox).

According to the Python style guide, all-upper-case strings should be used for constants and not variables. Continuing with our example, even considering that `DIV` is a class, it is a special class that should never be redefined by the user because doing so would break other WEB2PY applications. Hence, we believe this qualifies the `DIV` class as something that should be treated as a constant, further justifying our choice of notation.

In summary, the following conventions are followed:

- HTML helpers and validators are all upper case for the reasons discussed above (for example `DIV`, `A`, `FORM`, `URL`).

- The translator object `T` is upper case despite the fact that it is an instance of a class and not a class itself. Logically the translator object performs an action similar to the HTML helpers — it affects rendering part of the presentation. Also, `T` needs to be easy to locate in the code and has to have a short name.

- DAL classes follow the Python style guide (first letter capitalized), sometimes with the addition of a clarifying DAL prefix (for example `Table`, `Field`, `DALQuery`, etc.).

In all other cases we believe we have followed, as much as possible, the Python Style Guide (PEP8). For example all instance objects are lower-case (request, response, session, cache), and all internal classes are capitalized.

In all the examples of this book, WEB2PY keywords are shown in bold, while strings and comments are shown in italic.

CHAPTER 2

THE PYTHON LANGUAGE

2.1 About Python

Python is a general-purpose and very high-level programming language. Its design philosophy emphasizes programmer productivity and code readability. It has a minimalist core syntax with very few basic commands and simple semantics, but it also has a large and comprehensive standard library, including an Application Programming Interface (API) to many of the underlying Operating System (OS) functions. The Python code, while minimalist, defines objects such as linked lists (`list`), tuples (`tuple`), hash tables (`dict`), and arbitrarily long integers (`long`).

Python supports multiple programming paradigms. These are object-oriented (`class`), imperative (`def`), and functional (`lambda`) programming. Python has a dynamic type system and automatic memory management using reference counting (similar to Perl, Ruby, and Scheme).

Python was first released by Guido van Rossum in 1991. The language has an open, community-based development model managed by the non-profit Python Software Foundation. There are many interpreters and compilers that

implement the Python language, including one in Java (Jython) but, in this brief review, we refer to the reference C implementation created by Guido.

You can find many tutorials, the official documentation and library references of the language on the official Python website [2]

For additional Python references, we can recommend the books in ref. [36] and ref. [37].

You may skip this chapter if you are already familiar with the Python language.

2.2 Starting up

The binary distributions of WEB2PY for Microsoft Windows or Apple OS X come packaged with the Python interpreter built into the distribution file itself.

You can start it on Windows with the following command (type at the DOS prompt):

```
web2py.exe -S welcome
```

On Apple OS X, enter the following command type in a Terminal window (assuming you're in the same folder as web2py.app):

```
./web2py.app/Contents/MacOS/web2py -S welcome
```

On a Linux or other Unix box, chances are that you have Python already installed. If so, at a shell prompt type:

```
python web2py.py -S welcome
```

If you do not have Python 2.5 already installed, you will have to download and install it before running WEB2PY.

The `-S welcome` command line option instructs WEB2PY to run the interactive shell as if the commands were executed in a controller for the **welcome** application, the WEB2PY scaffolding application. This exposes almost all WEB2PY classes, objects and functions to you. This is the only difference between the WEB2PY interactive command line and the normal Python command line.

The admin interface also provides a web-based shell for each application. You can access the one for the "welcome" application at.

```
http://127.0.0.1:8000/admin/shell/index/welcome
```

You can try all the examples in this chapter using the normal shell or the web-based shell.

2.3 help, dir

The Python language provides two commands to obtain documentation about objects defined in the current scope, both builtins and user defined.

We can ask for `help` about an object, for example “1”:

```
>>> help(1)
Help on int object:

class int(object)
 |  int(x[, base]) -> integer
 |
 |  Convert a string or number to an integer, if possible.  A floating point
 |  argument will be truncated towards zero (this does not include a string
 |  representation of a floating point number!)  When converting a string, use
 |  the optional base.  It is an error to supply a base when converting a
 |  non-string. If the argument is outside the integer range a long object
 |  will be returned instead.
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
...
```

and, since "1" is an integer, we get a description about the `int` class and all its methods. Here the output has been truncated because it is very long and detailed.

Similarly, we can obtain a list of methods of the object "1" with the command `dir`:

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__
    ', '__delattr__', '__div__', '__divmod__', '__doc__', '__float__'
    , '__floordiv__', '__getattribute__', '__getnewargs__', '__hash__
    ', '__hex__', '__index__', '__init__', '__int__', '__invert__', '
    __long__', '__lshift__', '__mod__', '__mul__', '__neg__', '
    __new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow__
    ', '__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__
    ', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '
    __rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__', '
    __rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__
    ', '__str__', '__sub__', '__truediv__', '__xor__']
```

2.4 Types

Python is a dynamically typed language, meaning that variables do not have a type and therefore do not have to be declared. Values, on the other hand, do have a type. 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 includes, natively, data structures such as lists and dictionaries.

str

Python supports the use of two different types of strings: ASCII strings and Unicode strings. ASCII strings are delimited by '...', "..." or by "'.."' or """...""". Triple quotes delimit multiline strings. Unicode strings start with a `u` followed by the string containing Unicode characters. A Unicode string can be converted into an ASCII string by choosing an encoding for example:

```
>>> a = 'this is an ASCII string'
>>> b = u'This is a Unicode string'
>>> a = b.encode('utf8')
```

After executing these three commands, the resulting `a` is an ASCII string storing UTF8 encoded characters. By design, WEB2PY uses UTF8 encoded strings internally.

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 more explicit and less error prone, and is to be preferred.

Many Python objects, for example numbers, can be serialized into strings using `str` or `repr`. These two commands are very similar but produce slightly different output. For example:

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

For user-defined classes, `str` and `repr` can be defined/redefined using the special operators `__str__` and `__repr__`. These are briefly described later on; for more, refer to the official Python documentation [38]. `repr` always has a default value.

Another important characteristic of a Python string is that, like a list, it is an iterable object.

```
>>> for i in 'hello':
        print i
h
e
l
l
o
```

list

The main methods of a Python list are append, insert, and delete:

```
>>> 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
```

Lists can be sliced:

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

and concatenated:

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

A list is iterable; you can loop over it:

```
>>> a = [1, 2, 3]
>>> for i in a:
        print i
1
2
3
```

The elements of a list do not have to be of the same type; they can be any type of Python object.

tuple

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

```
>>> a = (1, 2, 3)
```

So while this works for a list:

```
>>> a = [1, 2, 3]
>>> a[1] = 5
>>> print a
[1, 5, 3]
```

the element assignment does not work for a tuple:

```
>>> a = (1, 2, 3)
>>> print a[1]
2
>>> a[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

The tuple, like the list, is an iterable object. Notice that a tuple consisting of a single element must include a trailing comma, as shown below:

```
>>> a = (1)
>>> print type(a)
<type 'int'>
>>> a = (1,)
>>> print type(a)
<type 'tuple'>
```

Tuples are very useful for efficient packing of objects because of their immutability, and the brackets are often optional:

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

dict

A Python `dict`ionary is a hash table that maps a key object to a value object. For example:

```
>>> 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 any object whose class implements 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 alphanumeric characters, a dictionary can also be declared with the alternative syntax:

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

Useful methods are `has_key`, `keys`, `values` and `items`:

```
>>> 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 its associated value.

Dictionary elements and list elements can be deleted with the command `del`:

```
>>> a = [1, 2, 3]
>>> del a[1]
>>> print a
[1, 3]
>>> a = dict(k='v', h2=3)
>>> del a['h2']
>>> print a
{'k':'v'}
```

Internally, Python uses the `hash` operator to convert objects into integers, and uses that integer to determine where to store the value.

```
>>> hash("hello world")
-1500746465
```

2.5 About Indentation

Python uses indentation to delimit blocks of code. A block starts with a line ending in colon, and continues for all lines that have a similar or higher indentation as the next line. 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. It is a good policy not to mix tabs with spaces, or you may run into trouble.

2.6 for...in

In Python, you can loop over iterable objects:

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

One common shortcut is `xrange`, which generates an iterable range without storing the entire list of elements.

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

This is equivalent to the C/C++/C#/Java syntax:

```
for(int i=0; i<4; i=i+1) { print(i); }
```

Another useful command is `enumerate`, which counts while looping:

```
>>> 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 starting with the value `a`, incrementing by `c`, and ending with the last value smaller than `b`, `a` defaults to 0 and `c` defaults to 1. `xrange` is similar but does not actually generate the list, only an iterator over the list; thus it is better for looping.

You can jump out of a loop using `break`

```
>>> for i in [1, 2, 3]:
        print i
        break
1
```

You can jump to the next loop iteration without executing the entire code block with `continue`

```
>>> for i in [1, 2, 3]:
        print i
        continue
        print 'test'
1
2
3
```

2.7 while

The `while` loop in Python works much as it does in many other programming languages, by looping an indefinite number of times and testing a condition before each iteration. If the condition is `False`, the loop ends.

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

There is no `loop...until` construct in Python.

2.8 def...return

Here is a typical Python function:

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

There is no need (or way) to specify types of the arguments or the return type(s).

Function arguments can have default values and can return multiple objects:

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

Function arguments can be passed explicitly 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
```

Functions can take a variable number of arguments:

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

Here arguments not passed by name (3, 'hello') are stored in list `a`, and arguments passed by name (`c` and `test`) are stored in the dictionary `b`.

In the opposite case, a list or tuple can be passed to a function that requires individual positional arguments by unpacking them:

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

and a dictionary can be unpacked to deliver keyword arguments:

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

2.9 if...elif...else

The use of conditionals in Python is intuitive:

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

"elif" means "else if". Both `elif` and `else` clauses are optional. There can be more than one `elif` but only one `else` statement. Complex conditions can be created using the `not`, `and` and `or` operators.

```
>>> for i in range(3):
>>>     if i == 0 or (i == 1 and i + 1 == 2):
>>>         print '0 or 1'
```

2.10 try... except...else...finally

Python can throw - pardon, raise - Exceptions:

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

If the exception is raised, it is caught by the `except` clause, which is executed, while the `else` clause is not. If no exception is raised, the `except` clause is not executed, but the `else` one is. The `finally` clause is always executed.

There can be multiple `except` clauses for different possible exceptions:

```
>>> try:
>>>     raise SyntaxError
>>> except ValueError:
>>>     print 'value error'
>>> except SyntaxError:
>>>     print 'syntax error'
syntax error
```

The `else` and `finally` clauses are optional.

Here is a list of built-in Python exceptions + HTTP (defined by WEB2PY)

```
BaseException
 +-- HTTP (defined by web2py)
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- Exception
      +-- GeneratorExit
      +-- StopIteration
      +-- StandardError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |    |    +-- UnicodeError
      |    |         +-- UnicodeDecodeError
      |    |         +-- UnicodeEncodeError
      |    |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
       +-- ImportWarning
       +-- UnicodeWarning
```

For a detailed description of each of them, refer to the official Python documentation.

WEB2PY exposes only one new exception, called `HTTP`. When raised, it causes the program to return an HTTP error page (for more on this refer to Chapter 4).

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

2.11 class

Because Python is dynamically typed, Python classes and objects may seem odd. In fact, you do not need to define the member variables (attributes) when declaring a class, and different instances of the same class can have different attributes. Attributes are generally associated with the instance, not the class (except when declared as "class attributes", which is the same as "static member variables" in C++/Java).

Here is an example:

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

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

Here is a more complex class:

```
>>> class MyClass(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 = MyClass(3, 4)
>>> print myinstance.add()
9
```

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

Notice that `__init__` takes 3 arguments and `add` takes one, and yet we call them with 2 and 0 arguments respectively. The first argument represents,

by convention, the local name used 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 `self` is not a reserved keyword.

This syntax is necessary to avoid ambiguity when declaring nested classes, such as a class that is local to a method inside another class.

2.12 Special Attributes, Methods and Operators

Class attributes, methods, and operators starting with a double underscore are usually intended to be private, although this is a convention that is not enforced by the interpreter.

Some of them are reserved keywords and have a special meaning.

Here, as an example, are three of them:

- `__len__`
- `__getitem__`
- `__setitem__`

They can be used, for example, to create a container object that acts like a list:

```
>>> 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 operators include `__getattr__` and `__setattr__`, which define the get and set attributes for the class, and `__sum__` and `__sub__`, which overload arithmetic operators. For the use of these operators we refer the reader to more advanced books on this topic. We have already mentioned the special operators `__str__` and `__repr__`.

2.13 File Input/Output

In Python you can open and write in a file with:

```
>>> file = open('myfile.txt', 'w')
>>> file.write('hello world')
```

Similarly, you can read back from the file with:

```
>>> file = open('myfile.txt', 'r')
>>> print file.read()
hello world
```

Alternatively, you can read in binary mode with "rb", write in binary mode with "wb", and open the file in append mode "a", using standard C notation.

The `read` command takes an optional argument, which is the number of bytes. You can also jump to any location in a file using `seek`.

You can read back from the file with `read`

```
>>> print file.seek(6)
>>> print file.read()
world
```

and you can close the file with:

```
>>> file.close()
```

although often this is not necessary, because a file is closed automatically when the variable that refers to it goes out of scope.

> *When using* WEB2PY*, you do not know where the current directory is, because it depends on how* WEB2PY *is configured. The variable* `request.folder` *contains the path to the current application. Paths can be concatenated with the command* `os.path.join`*, discussed below.*

2.14 lambda

There are cases when you may need to dynamically generate an unnamed 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 with arguments [a] that returns [b]". Even if the function is unnamed, it can be stored into a variable, and thus it acquires a name. Technically this is different than using `def`, because it is the variable referring to the function that has a name, not the function itself.

Who needs lambdas? Actually they are very useful because they allow to refactor a function into another function by setting default arguments, without defining an actual new function but a temporary one. For example:

```
>>> def f(a, b): return a + b
>>> g = lambda a: f(a, 3)
>>> g(2)
5
```

Here is a more complex and more compelling application. Suppose you have a function that checks whether its argument is prime:

```
def isprime(number):
    for p in range(2, number):
        if number % p:
            return False
    return True
```

This function is obviously time consuming.

Suppose you have a caching function `cache.ram` that takes three arguments: a key, a function and a number of seconds.

```
value = cache.ram('key', f, 60)
```

The first time it is called, it calls the function `f()`, stores the output in a dictionary in memory (let's say "d"), and returns it so that value is:

```
value = d['key']=f()
```

The second time it is called, if the key is in the dictionary and not older than the number of seconds specified (60), it returns the corresponding value without performing the function call.

```
value = d['key']
```

How would you cache the output of the function **isprime** for any input? Here is how:

```
>>> number = 7
>>> print cache.ram(str(number), lambda: isprime(number), seconds)
True
>>> print cache.ram(str(number), lambda: isprime(number), seconds)
True
```

The output is always the same, but the first time `cache.ram` is called, `isprime` is called; the second time it is not.

> *The existence of* `lambda` *allows refactoring an existing function in terms of a different set of arguments.*
>
> `cache.ram` *and* `cache.disk` *are* WEB2PY *caching functions.*

2.15 exec, eval

Unlike Java, Python is a truly interpreted language. This means it has the ability to execute Python statements stored in strings. For example:

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

What just happened? The function `exec` tells the interpreter to call itself and execute the content of 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 defined in `c` (`b` in the example), but does not see `c` or `a` themselves. 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`, which works very much like `exec` except that it expects the argument to evaluate to a value, and it returns that value.

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

2.16 import

The real power of Python is in its library modules. They provide a large and consistent set of Application Programming Interfaces (APIs) to many system libraries (often in a way independent of the operating system).

For example, if you need to use a random number generator, you can do:

```
>>> import random
>>> print random.randint(0, 9)
5
```

This prints a random integer between 0 and 9 (including 9), 5 in the example. The function `randint` is defined in the module `random`. It is also possible to import an object from a module into the current namespace:

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

or import all objects from a module into 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 mainly use objects defined in modules `os`, `sys`, `datetime`, `time` and `cPickle`.

> *All of the* WEB2PY *objects are accessible via a module called* `gluon`*, and that is the subject of later chapters. Internally,* WEB2PY *uses many Python modules (for example* `thread`*), but you rarely need to access them directly.*

In the following subsections we consider those modules that are most useful.

os

This module provides an interface to the operating system API. For example:

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

> *Some of the* `os` *functions, such as* `chdir`*, MUST NOT be used in* WEB2PY *because they are not thread-safe.*

`os.path.join` is very useful; it allows the concatenation of paths in an OS-independent way:

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

System environment variables can be accessed via:

```
>>> print os.environ
```

which is a read-only dictionary.

sys

The `sys` module contains many variables and functions, but the one we use the most is `sys.path`. It contains a list of paths where Python searches for modules. When we try to import a module, Python looks for it in all the folders listed in `sys.path`. If you install additional modules in some location and want Python to find them, you need to append the path to that location to `sys.path`.

```
>>> import sys
>>> sys.path.append('path/to/my/modules')
```

When running WEB2PY, Python stays resident in memory, and there is only one `sys.path`, while there are many threads servicing the HTTP requests. To avoid a memory leak, it is best to check if a path is already present before appending:

```
>>> path = 'path/to/my/modules'
>>> if not path in sys.path:
        sys.path.append(path)
```

datetime

The use of the datetime module is best illustrated by some examples:

```
>>> import datetime
>>> print datetime.datetime.today()
2008-07-04 14:03:90
>>> print datetime.date.today()
2008-07-04
```

Occasionally you may need to timestamp data based on the UTC time as opposed to local time. In this case you can use the following function:

```
>>> import datetime
>>> print datetime.datetime.utcnow()
2008-07-04 14:03:90
```

The datetime modules contains various classes: date, datetime, time and timedelta. The difference between two date or two datetime or two time objects is a timedelta:

```
>>> a = datetime.datetime(2008, 1, 1, 20, 30)
>>> b = datetime.datetime(2008, 1, 2, 20, 30)
>>> c = b - a
>>> print c.days
1
```

In WEB2PY, date and datetime are used to store the corresponding SQL types when passed to or returned from the database.

time

The time module differs from date and datetime because it represents time as seconds from the epoch (beginning of 1970).

```
>>> import time
>>> t = time.time()
1215138737.571
```

Refer to the Python documentation for conversion functions between time in seconds and time as a `datetime`.

cPickle

This is a very powerful module. It provides functions that can serialize almost any Python object, including self-referential objects. For example, let's build a weird object:

```
>>> class MyClass(object): pass
>>> myinstance = MyClass()
>>> myinstance.x = 'something'
>>> a = [1 ,2, {'hello':'world'}, [3, 4, [myinstance]]]
```

and now:

```
>>> import cPickle
>>> b = cPickle.dumps(a)
>>> c = cPickle.loads(b)
```

In this example, `b` is a string representation of `a`, and `c` is a copy of `a` generated by deserializing `b`.

cPickle can also serialize to and deserialize from a file:

```
>>> cPickle.dumps(a, open('myfile.pickle', 'wb'))
>>> c = cPickle.loads(open('myfile.pickle', 'rb'))
```

CHAPTER 3

OVERVIEW

3.1 Startup

WEB2PY comes in binary packages for Windows and Mac OS X. There is also a source code version that runs on Windows, Mac, Linux, and other Unix systems. The Windows and OS X binary versions include the necessary Python interpreter. The source code package assumes that Python is already installed on the 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, run:

```
web2py.exe
```

On OS X, run:

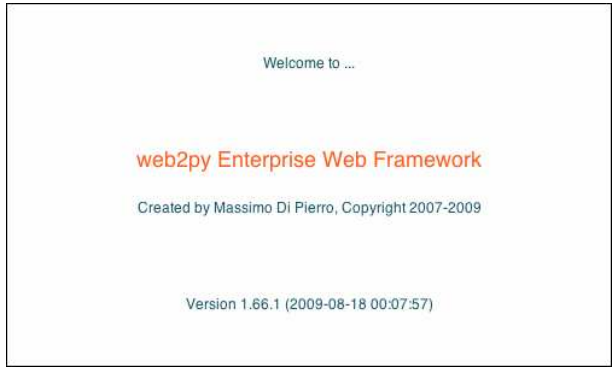
```
web2py.app
```

On Unix and Linux, run from source by typing:

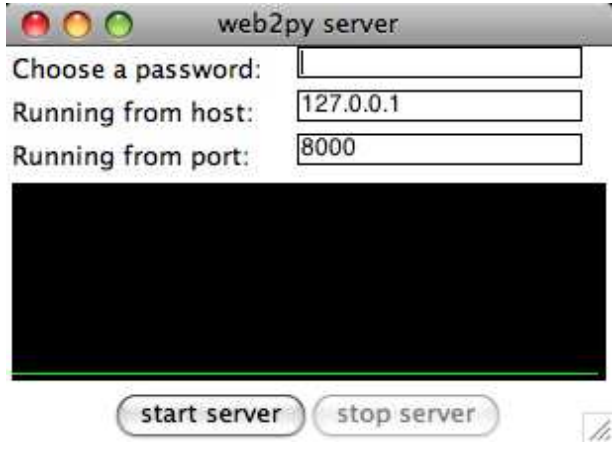
```
python2.5 web2py.py
```

The WEB2PY program accepts various command line options which are discussed later.

By default, at startup, WEB2PY displays a startup window:



and then displays a GUI widget that asks you to choose a one-time administrator password, 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 its web server on 127.0.0.1:8000 (port 8000 on localhost), but you can run it on any available IP address and port. You can query the IP address of your network interface by opening a command line and typing `ipconfig` on Windows or `ifconfig` on OS X and Linux. From now on we assume WEB2PY is running on localhost (127.0.0.1:8000). Use 0.0.0.0:80 to run WEB2PY publicly on any of your network interfaces.



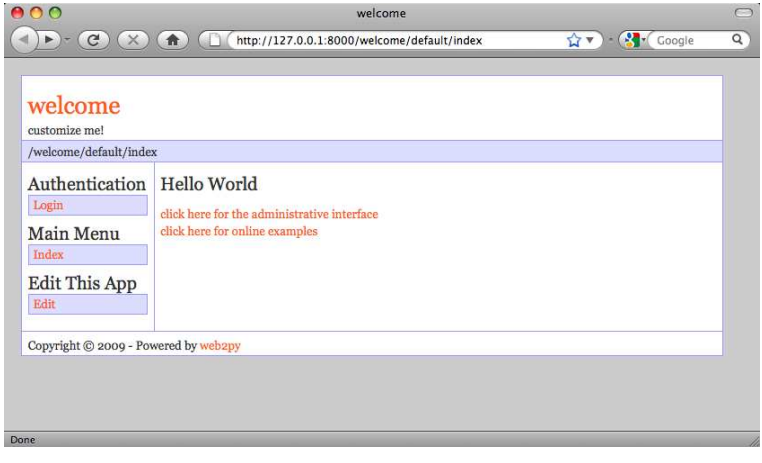
If you do not provide an administrator password, the administration interface is disabled. This is a security measure to prevent publicly exposing the admin interface.

The administration interface is only accessible from localhost unless you run WEB2PY behind Apache with mod_proxy. If admin detects a proxy, the session cookie is set to secure and **admin** login does not work unless the communication between the client and the proxy goes over HTTPS. This is another security measure. All communications between the client and the admin must always be local or encrypted; otherwise an attacker would be able to perform a man-in-the middle attack or a replay attack and execute arbitrary code on the server.

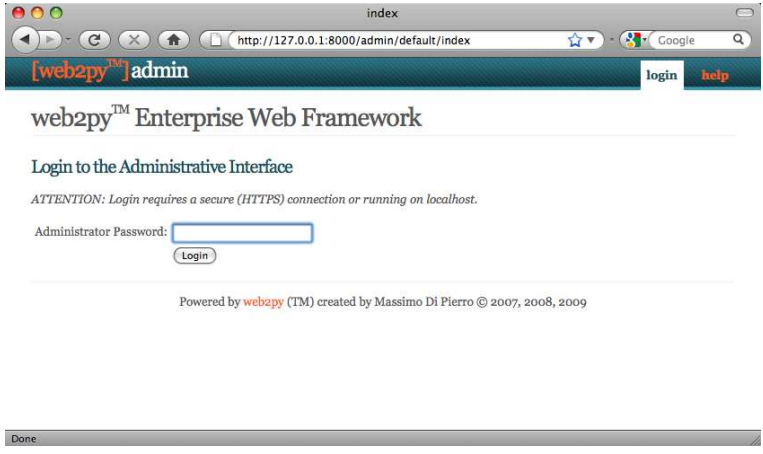
After the administration password has been set, WEB2PY starts up the web browser at the page:

```
http://127.0.0.1:8000/
```

If the computer does not have a default browser, open a web browser and enter the URL.

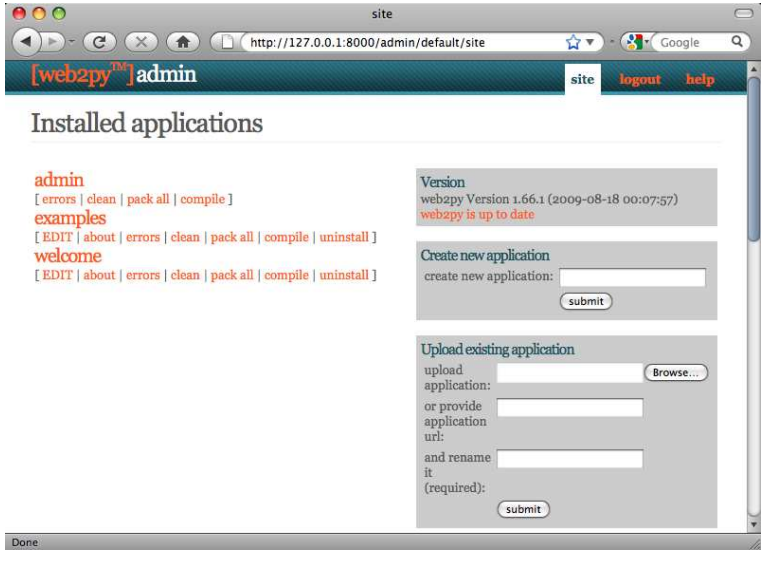


Clicking on "administrative interface" takes you to the login page for the administration interface.



The administrator password is the same as the password you chose at startup. Notice that 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 unless the <recycle> option is specified. This is distinct from the authentication mechanism in WEB2PY applications.

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 you are using right now.
- An **examples** application, with the online interactive documentation and a replica of the WEB2PY official website.
- A **welcome** application. This is the basic template for any other WEB2PY application. It is referred to as the scaffolding application. This is also the application that welcomes a user at startup.

Ready-to-use WEB2PY applications are referred to as WEB2PY *appliances*. You can download many freely available appliances from [33]. WEB2PY users are encouraged to submit new appliances, either in open-source or closed-source (compiled and packed) form.

From the admin application's [site] page, you can perform the following operations:

- **install** an application by completing the form on the bottom right of the page. Give a name to the application, select the file containing a packaged application or the URL where the application is located, 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".
- **package** an application for distribution by clicking on the corresponding button. A downloaded application is a tar file containing everything, including the database. You should never untar this file; it is automatically unpackaged by WEB2PY when one installs it using **admin**.
- **clean up** an application's temporary files, such as sessions, errors and cache files.
- **EDIT** an application.

3.2 Say Hello

Here, as an example, we create a simple web app that displays the message "Hello from MyApp" to the user. We will call this application "myapp". We will also add a counter that counts how many times the same user visits the page.

You can create a new application simply by typing its name in the form on the top right of the **site** page in **admin**.

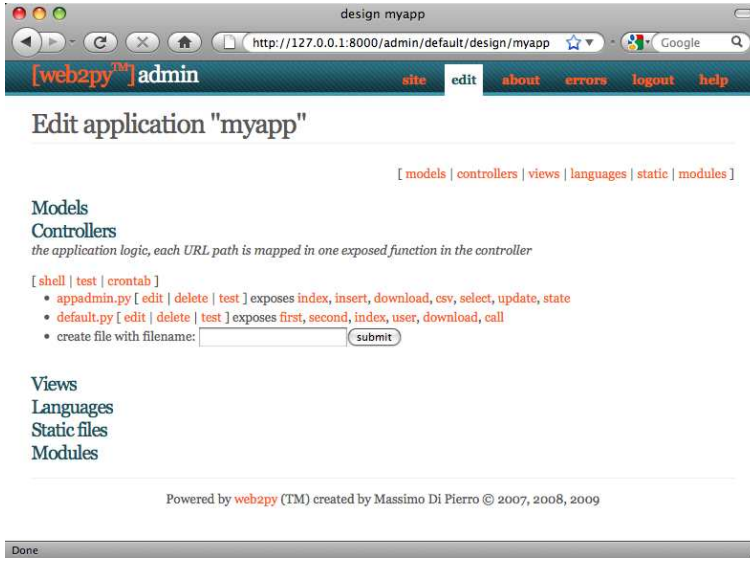
After you press [submit], the application is created as a copy of the built-in welcome application.

To run the new application, visit:

```
http://127.0.0.1:8000/myapp
```

Now you have a copy of the welcome application.

To edit an application, click on the [EDIT] button for the newly created application.



The **EDIT** page tells you what is inside the application. Every WEB2PY application consists of certain files, most of which fall into one of 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.
- **modules**: Python modules that belong to the application.
- **static files**: static images, CSS files [39, 40, 41], JavaScript files [42, 43], etc.

Everything is neatly organized following the Model-View-Controller design pattern. Each section in the [EDIT] page corresponds to a subfolder in the application folder.

Notice that section headings will toggle their content. Folder names under static files are also collapsible.

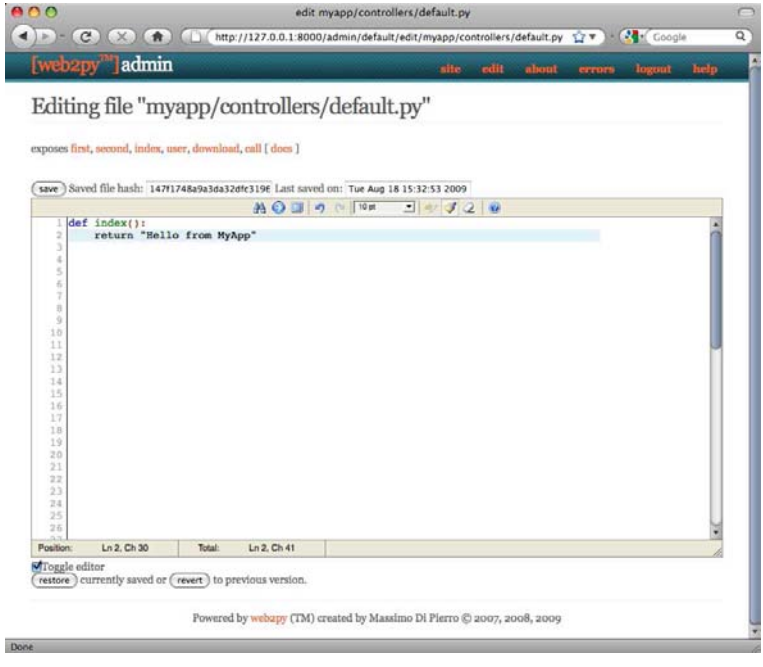
> *Each file listed in the section corresponds to a file physically located in the subfolder. Any operation performed on a file via the admin interface (create, edit, delete) can be performed directly from the shell using your favorite editor.*

The application contains other types of files (database, session files, error files, etc.), but they are not listed on the [EDIT] page because they are not created or modified by the administrator. They are created and modified by the application itself.

The controllers contain the logic and workflow of the application. Every URL gets mapped into a call to one of the functions in the controllers (actions). There are two default controllers: "appadmin.py" and "default.py". **appadmin** provides the database administrative interface; we do not need it now. "default.py" is the controller that you need to edit, the one that is called by default when no controller is specified in the URL. Edit the "index" function as follows:

```
def index():
    return "Hello from MyApp"
```

Here is what the online editor looks like:



Save it and go back to the [EDIT] page. Click on the index link to visit the newly created page.

When you visit the URL

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

the index action in the default controller of the myapp application is called. It returns a string that the browser displays for us. It should look like this:

Now, edit the "index" function as follows:

```
def index():
    return dict(message="Hello from MyApp")
```

Also from the [EDIT] page, edit the view default/index (the new file associated with the action) and, in this file, write:

```
<html>
   <head></head>
   <body>
      <h1>{{=message}}</h1>
   </body>
</html>
```

Now the action returns a dictionary defining a `message`. When an action returns a dictionary, WEB2PY looks for a view with the name "[controller]/[function].[extension]" and executes it. Here [extension] is the requested extension. If no extension is specified, it defaults to "html", and that is what we will assume here. Under this assumption, the view is an HTML file that embeds Python code using special {{ }} tags. In particular, in the example, the {{=message}} instructs WEB2PY to replace the tagged code with the value of the `message` returned by the action. Notice that `message` here is not a WEB2PY keyword but is defined in the action. So far we have not used any WEB2PY keywords.

If WEB2PY does not find the requested view, it uses the "generic.html" view that comes with every application.

> *If an extension other than "html" is specified ("json" for example), and the view file "[controller]/[function].json" is not found,* WEB2PY *looks for the view "generic.json".* WEB2PY *comes with generic.html, generic.json, generic.xml, and generic.rss. These generic views can be modified for each application individually, and additional views can be added easily.*

Read more on this topic in Chapter 9.

If you go back to [EDIT] and click on index, you will now see the following HTML page:

3.3 Let's Count

Let's now add a counter to this page that will count how many times the same visitor displays the page.

WEB2PY automatically and transparently tracks visitors using sessions and cookies. For each new visitor, it creates a session and assigns a unique "session_id". The session is a container for variables that are stored server-side. The unique id is sent to the browser via a cookie. When the visitor requests another page from the same application, the browser sends the cookie back, it is retrieved by WEB2PY, and the corresponding session is restored.

To use the session, modify the default controller:

```
def index():
    if not session.counter:
        session.counter = 1
    else:
        session.counter += 1
    return dict(message="Hello from MyApp", counter=session.counter)
```

Notice that counter is not a WEB2PY keyword but `session` is. We are asking WEB2PY to check whether there is a counter variable in the session and, if not, to create one and set it to 1. If the counter is there, we ask WEB2PY to increase the counter by 1. Finally we pass the value of the counter to the view.

A more compact way to code the same function is this:

```
def index():
    session.counter = (session.counter or 0) + 1
    return dict(message="Hello from MyApp", counter=session.counter)
```

Now modify the view to add a line that displays the value of the counter:

```
<html>
   <head></head>
   <body>
      <h1>{{=message}}</h1>
      <h2>Number of visits: {{=counter}}</h2>
   </body>
</html>
```

When you visit the index page again (and again) you should get the following HTML page:

The counter is associated to each visitor, and is incremented each time the visitor reloads the page. Different visitors see different counters.

3.4 Say My Name

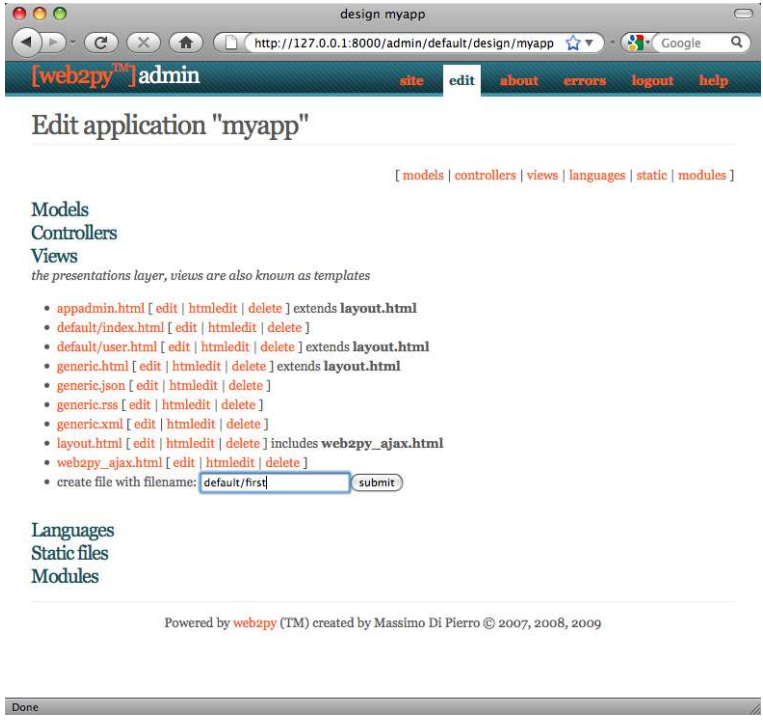
Now create two pages (first and second), where the first page creates a form, asks the visitor's name, and redirects to the second page, which greets the visitor by name.

$$\texttt{first} \xrightarrow{form} \texttt{second}$$

Write the corresponding actions in the default controller:

```
def first():
    return dict()

def second():
    return dict()
```

Then create a view "default/first.html" for the first action:



and enter:

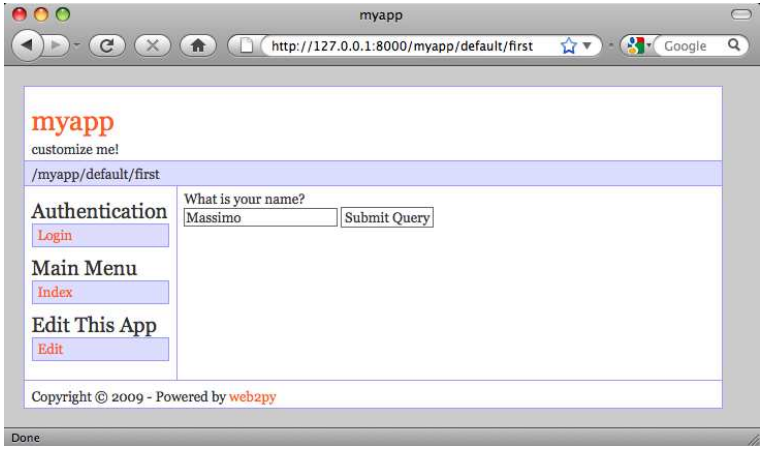
```
{{extend 'layout.html'}}
What is your name?
<form action="second">
  <input name="visitor_name" />
  <input type="submit" />
</form>
```

Finally, create a view "default/second.html" for the second action:

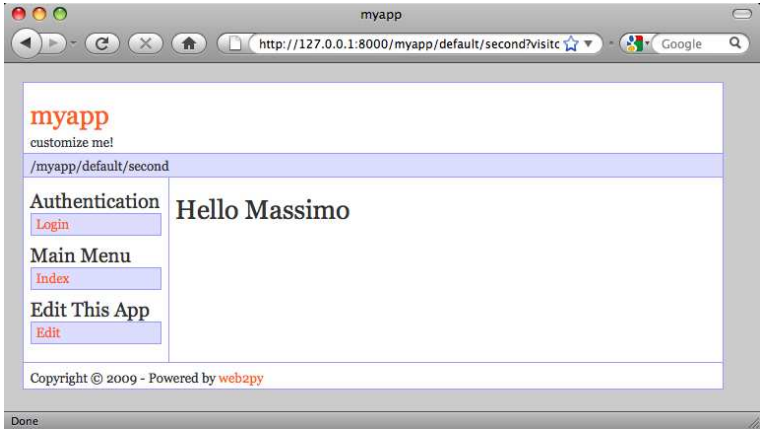
```
{{extend 'layout.html'}}
<h1>Hello {{=request.vars.visitor_name}}</h1>
```

In both views we have extended the basic "layout.html" view that comes with WEB2PY. The layout view keeps the look and feel of the two pages coherent. The layout file can be edited and replaced easily, since it mainly contains HTML code.

If you now visit the first page, type your name:



and submit the form, you will receive a greeting:



3.5 Form self-submission

The above mechanism for form submission is very common, but it is not good programming practice. All input should be validated and, in the above example, the burden of validation would fall on the second action. Thus the action that performs the validation is different from the action that generated the form. This may cause redundancy in the code.

A better pattern for form submission is to submit forms to the same action that generated them, in our example the "first". The "first" action should receive the variables, process them, store them server side, and redirect the visitor to the "second" page, which retrieves the variables.

$$\texttt{first} \xrightarrow{redirect} \texttt{second}$$

You can modify the default controller as follows to implement self-submission:

```
def first():
    if request.vars.visitor_name:
        session.visitor_name = request.vars.visitor_name
        redirect(URL(r=request, f='second'))
    return dict()

def second():
    return dict()
```

Accordingly, you need to modify the "default/first.html" view:

```
{{extend 'layout.html'}}
What is your name?
<form>
  <input name="visitor_name" />
  <input type="submit" />
</form>
```

and the "default/second.html" view needs to retrieve the data from the `session` instead of from the `request.vars`:

```
{{extend 'layout.html'}}
<h1>Hello {{=session.visitor_name or "anonymous"}}</h1>
```

From the point of view of the visitor, the self-submission behaves exactly the same as the previous implementation. We have not added validation yet, but it is now clear that validation should be performed by the first action.

This approach is better also because the name of the visitor stays in the session, and can be accessed by all actions and views in the applications without having to be passed around explicitly.

Note that if the "second" action is ever called before a visitor name is set, it will display "Hello anonymous" because `session.visitor_name` returns `None`. Alternatively we could have added the following code in the controller (inside or outside the `second` function:

```
if not request.function=='first' and not session.visitor_name:
    redirect(URL(r=request, f='first'))
```

This is a general mechanism that you can use to enforce authorization on controllers, although see Chapter 8 for a more powerful method.

With WEB2PY we can move one step further and ask WEB2PY to generate the form for us, including validation. WEB2PY provides helpers (FORM, INPUT, TEXTAREA, and SELECT/OPTION) with the same names as the equivalent HTML tags. They can be used to build forms either in the controller or in the view.

For example, here is one possible way to rewrite the first action:

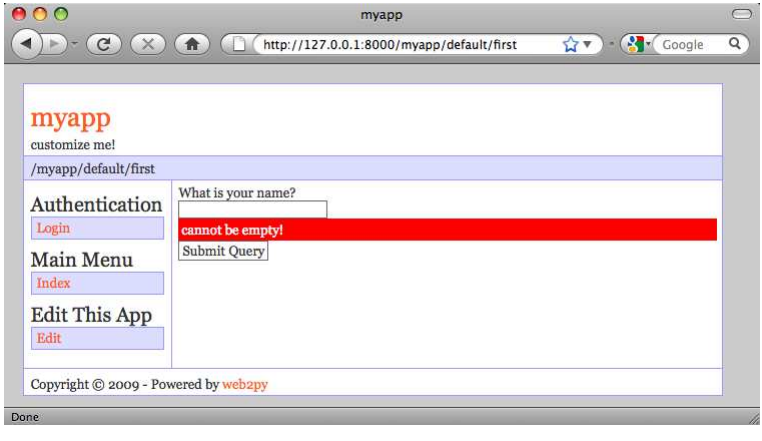
```
def first():
    form = FORM(INPUT(_name='visitor_name', requires=IS_NOT_EMPTY()),
                INPUT(_type='submit'))
    if form.accepts(request.vars, session):
        session.visitor_name = form.vars.visitor_name
        redirect(URL(r=request, f='second'))
    return dict(form=form)
```

where we are saying that the FORM tag contains two INPUT tags. The attributes of the input tags are specified by the named arguments starting with underscore. The `requires` argument is not a tag attribute (because it does not start by underscore) but it sets a validator for the value of visitor_name.

The `form` object can be easily serialized in HTML by embedding it in the "default/first.html" view.

```
{{extend 'layout.html'}}
What is your name?
{{=form}}
```

The `form.accepts` method applies the validators. If the self-submitted form passes validation, it stores the variables in the session and redirects as before. If the form does not pass validation, error messages are inserted in the form and shown to the user, shown below:



In the next section we will show how forms can be generated automatically from a model.

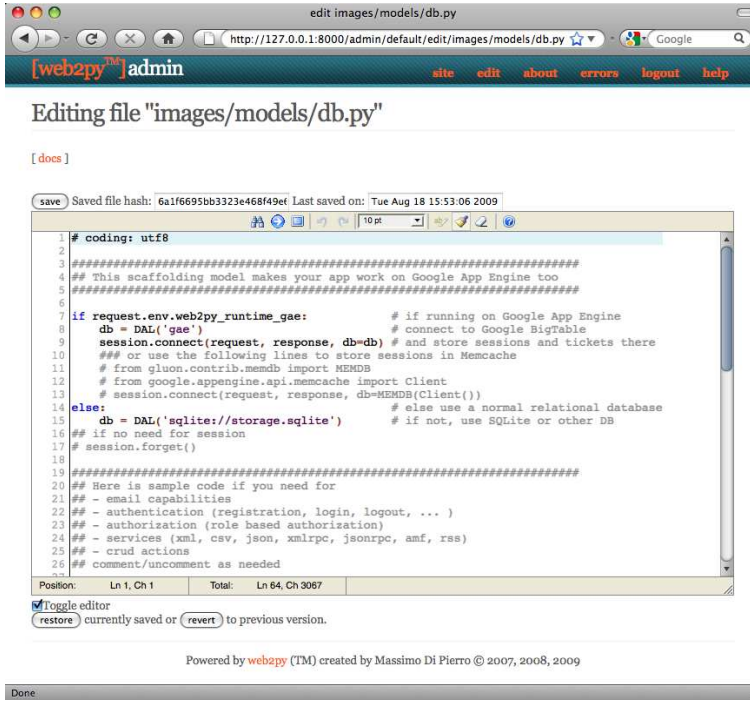
3.6 An Image Blog

Here, as another example, we wish to create a web application that allows the administrator to post images and give them a name, and allows the visitors of the web site to view the images and submit comments.

As before, create the new application from the **site** page in **admin** and navigate to the [EDIT] page:

We start by creating a model, a representation of the persistent data in the application (the images to upload, their names, and the comments). First, you need to create/edit a model file which, for lack of imagination, we call "db.py". Models and controllers must have a `.py` extension since they are 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.

Edit the "db.py" file by clicking the corresponding "edit" button:



and enter the following:

```
db = DAL("sqlite://storage.db")

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

db.define_table('comment',
   Field('image_id', db.image),
   Field('author'),
   Field('email'),
   Field('body', 'text'))

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()

db.comment.image_id.writable = db.comment.image_id.readable = False
```

Let's analyze this line by line.

- Line 1 defines a global variable called `db` that represents the database connection. In this case it is a connection to a SQLite database stored

in the file "applications/images/databases/storage.db". In the SQLite case, if the database does not exist, it is created.

You 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, to make it easy to remember.

- Lines 3-5 define a table "image". `define_table` is a method of the `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 ("id" is not explicitly declared because all tables have an id field by default). The field "title" is a string, and the field "file" is of type "upload". "upload" is a special type of field used by the WEB2PY Data Abstraction Layer (DAL) to store the names of uploaded files. WEB2PY knows how to upload files (via streaming if they are large), rename them safely, and store them.

  When a table is defined, WEB2PY takes one of several possible actions: a) if the table does not exist, the table is created; b) 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 its contents; c) if the table exists and corresponds to the definition, WEB2PY does nothing.

  This behavior is called "migration". In WEB2PY migrations are automatic, but can be disabled for each table by passing `migrate=False` as the last argument of `define_table`.

- Lines 7-11 define another table called "comment". A comment has an "author", an "email" (we intend to store the email address of the author of the comment), a "body" of type "text" (we intend to use it to store the actual comment posted by the author), and an "image_id" field of type reference that points to `db.image` via the "id" field.

- In lines 13-14 `db.image.title` represents the field "title" of table "image". The attribute `requires` allows you to set requirements/constraints that will be enforced by WEB2PY forms. 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)`). The objects representing these constraints are called validators. Multiple validators can be grouped in a list. Validators are executed in the order they appear. `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 `a`.

- Line 16 requires that the field "image_id" of table "comment" is in `db.image.id`. As far as the database is concerned, we had already declared this when we defined the table "comment". Now we are explicitly telling the model that this condition should be enforced by WEB2PY, too, at the form processing level when a new comment is posted, so that invalid values do not propagate from input forms to the database. We also require that the "image_id" be represented by the "title", `'%(title)s'`, of the corresponding record.

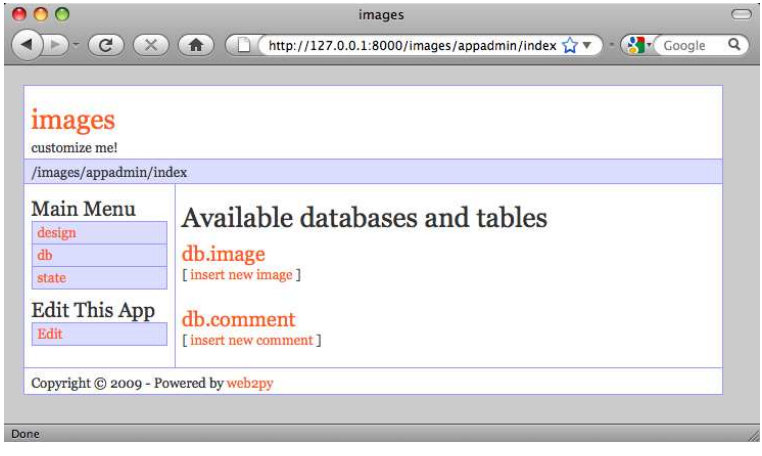
- Line 18 indicates that the field "image_id" of table "comment" should not be shown in forms, `writable=False` and not even in readonly forms, `readable=False`.

The meaning of the validators in lines 17-19 should be obvious.

Once a model is defined, if there are no errors, WEB2PY creates an application administration interface to manage the database. You access it via the "database administration" link in the [EDIT] page or directly:

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

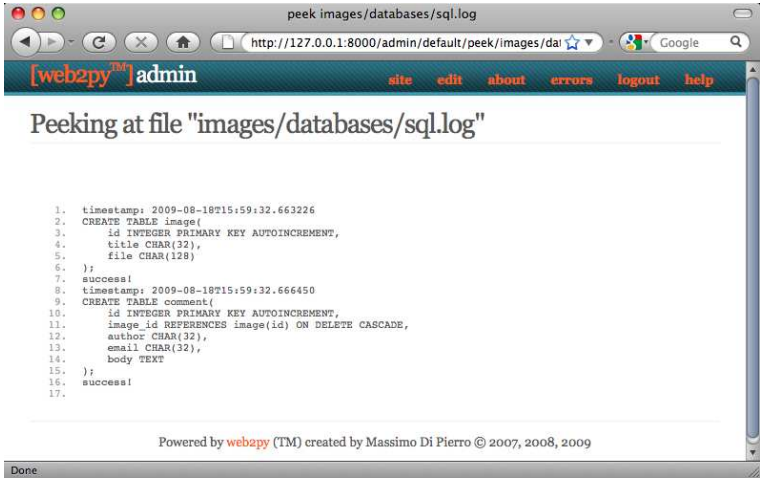
Here is a screenshot of the **appadmin** interface:



This interface is coded in the controller called "appadmin.py" and the corresponding view "appadmin.html". From now on, we will refer to this interface simply as **appadmin**. It allows the administrator to insert new database records, edit and delete existing records, browse tables, and perform database joins.

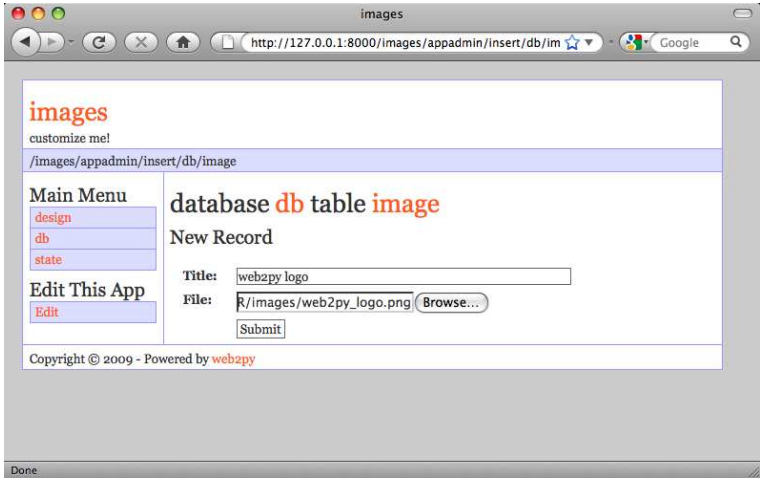
The first time **appadmin** is accessed, the model is executed and the tables are created. The WEB2PY DAL translates Python code into SQL statements that are specific to the selected database back-end (SQLite in this example).

You can see the generated SQL from the [EDIT] page by clicking on the "sql.log" link under "models". Notice that the link is not present until the tables have been created.



If you were to edit the model and access **appadmin** again, WEB2PY would generate SQL to alter the existing tables. The generated SQL is logged into "sql.log".

Now go back to **appadmin** and try to insert a new image record:



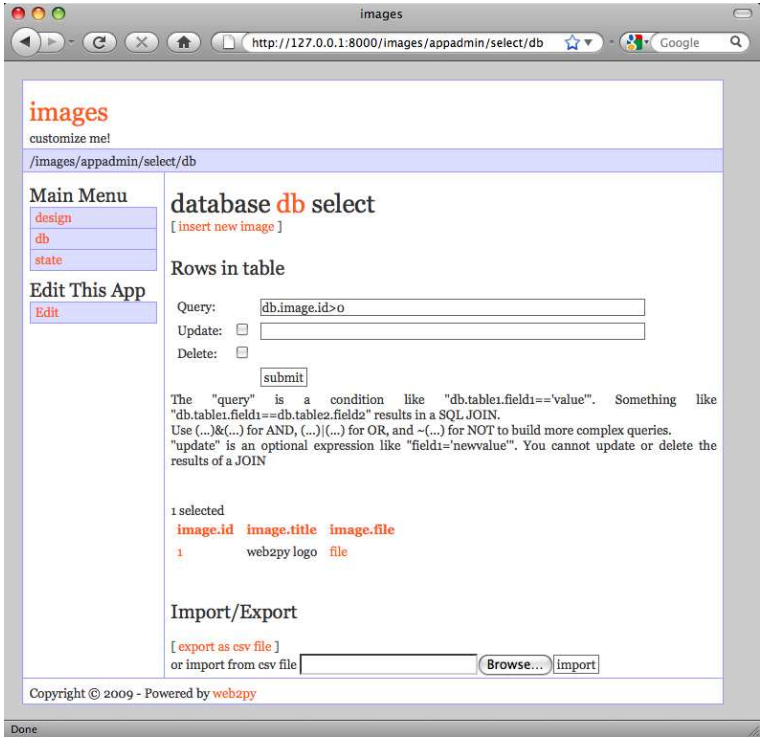
WEB2PY has translated the `db.image.file` "upload" field into an upload form for the file. When the form is submitted and an image file is uploaded, the file is renamed in a secure way that preserves the extension, it is saved with the new name under the application "uploads" folder, and the new name

is stored in the `db.image.file` field. This process is designed to prevent directory traversal attacks.

When you click on a table name in **appadmin**, WEB2PY performs a select of all records on the current table, identified by the DAL query

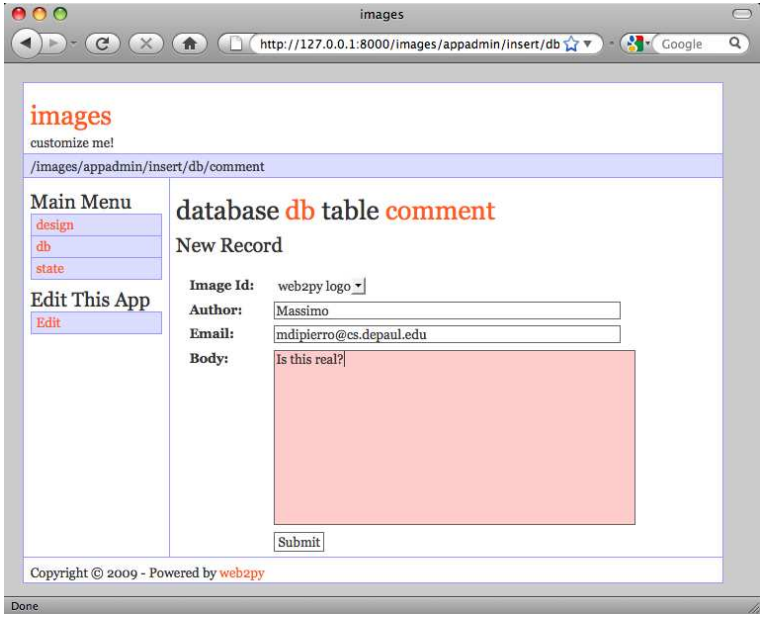
```
db.image.id > 0
```

and renders the result.



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

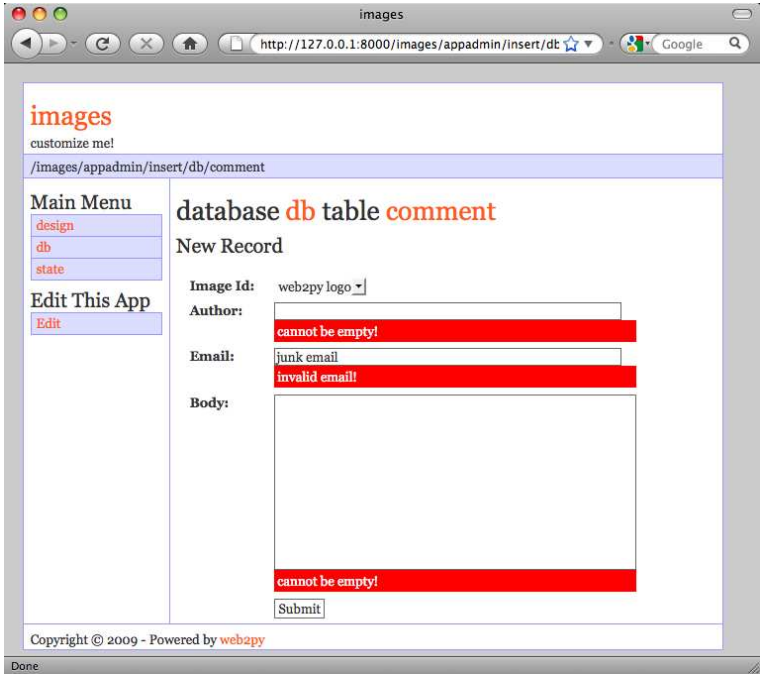
To edit or delete a single record, click on the record id number.



Because of the `IS_IN_DB` validator, the reference field "image_id" is rendered by a drop-down menu. The items in the drop-down are stored as keys (`db.image.id`), but are represented by their `db.image.title`, as specified by the validator.

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

The following figure shows what happens when you submit a form that does not pass validation:



The same forms that are automatically generated by **appadmin** can also be generated programmatically via the `SQLFORM` helper and embedded in user applications. These forms are CSS-friendly, and can be customized.

Every application has its own **appadmin**; therefore, **appadmin** itself can be modified without affecting other applications.

So far, the application knows how to store data, and we have seen how to access the database via appadmin. 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 walk-through. Specifically we want to create:

- An "index" page that lists all available images sorted by title and links to detail pages for the images.
- A "show/[id]" page that shows the visitor the requested image and allows the visitor to view and post comments.
- A "download/[name]" action to download uploaded images.

This is represented schematically here:

$$\texttt{index} \longrightarrow \texttt{show/}[id] \xrightarrow{img} \texttt{download/}[name]$$

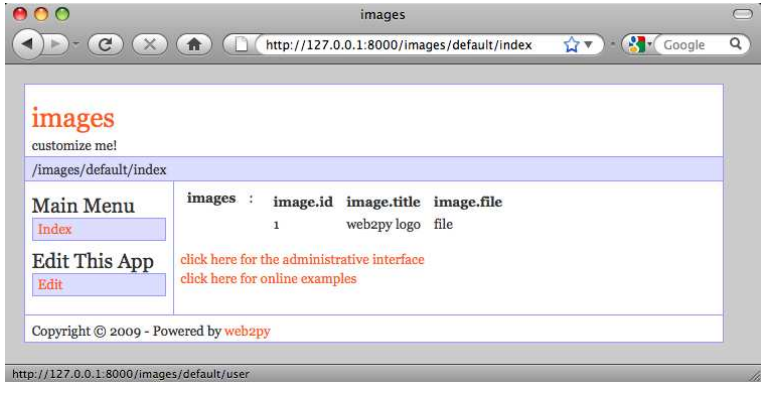
Go back to the [EDIT] page and edit the "default.py" controller, replacing its contents with the following:

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

This action returns 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 action performs a select of all fields (`db.image.ALL`) from table image, ordered by `db.image.title`. The result of the select is a `Rows` object containing the records. Assign it to a local variable called `images` returned by the action to the view. `images` is iterable and its elements are the selected rows. For each row the columns can be accessed as dictionaries: `images[0]['title']` or equivalently as `images[0].title`.

If you do not write a view, the dictionary is rendered by "views/generic.html" and a call to the index action would look like this:



You have not created a view for this action yet, so WEB2PY renders the set of records in plain tabular form.

Proceed to create a view for the index action. Return to admin, edit "default/index.html" and replace its content with the following:

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

The first thing to notice is that a view is pure HTML with special {{...}} tags. The code embedded in {{...}} is pure Python code with one caveat: indentation is irrelevant. Blocks of code start with lines ending in colon (:) and end in lines beginning with the keyword `pass`. In some cases the end of a block is obvious from context and the use of `pass` is not required.

Lines 5-7 loop over the image rows and for each row image display:

```
1 LI(A(image.title, _href=URL(r=request, f='show', args=image.id))
```

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

```
1 URL(r=request, f='show', args=image.id)
```

i.e., the URL within the same application and controller as the current request `r=request`, calling the function called "show", `f="show"`, and passing a single argument to the function, `args=image.id`.

`LI`, `A`, etc. are WEB2PY helpers that map to the corresponding HTML tags. Their unnamed arguments are interpreted as objects to be serialized and inserted in the tag's innerHTML. Named arguments starting with an underscore (for example `_href`) are interpreted as tag attributes but without the underscore. For example `_href` is the `href` attribute, `_class` is the `class` attribute, etc.

As an example, the following statement:

```
1 {{=LI(A('something', _href=URL(r=request, f='show', args=123))}}
```

is rendered as:

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

A handful of helpers (`INPUT`, `TEXTAREA`, `OPTION` and `SELECT`) also support some special named attributes not starting with underscore (`value`, and `requires`). They are important for building custom forms and will be discussed later.

Go back to the [EDIT] page. It now indicates that "default.py exposes index". By clicking on "index", you can visit the newly created page:

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

which looks like:

If you click on the image name link, you are directed to:

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

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

Let's edit the "default.py" controller and replace its content with:

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

def show():
    image = db(db.image.id==request.args(0)).select()[0]
    form = SQLFORM(db.comment)
    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():
    return response.download(request, db)
```

The controller contains two actions: "show" and "download". The "show" action selects the image with the `id` parsed from the request args and all comments related to the image. "show" then passes everything to the view "default/show.html".

The image id referenced by:

```
URL(r=request, f='show', args=image.id)}
```

in "default/index.html", can be accessed as: `request.args(0)` from the "show" action.

The "download" action expects a filename in `request.args(0)`, builds a path to the location where that file is supposed to be, and sends it back to the client. If the file is too large, it streams the file without incurring any memory overhead.

Notice the following statements:

- Line 7 creates an insert form SQLFORM for the `db.comment` table using only the specified fields.

- Line 8 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.

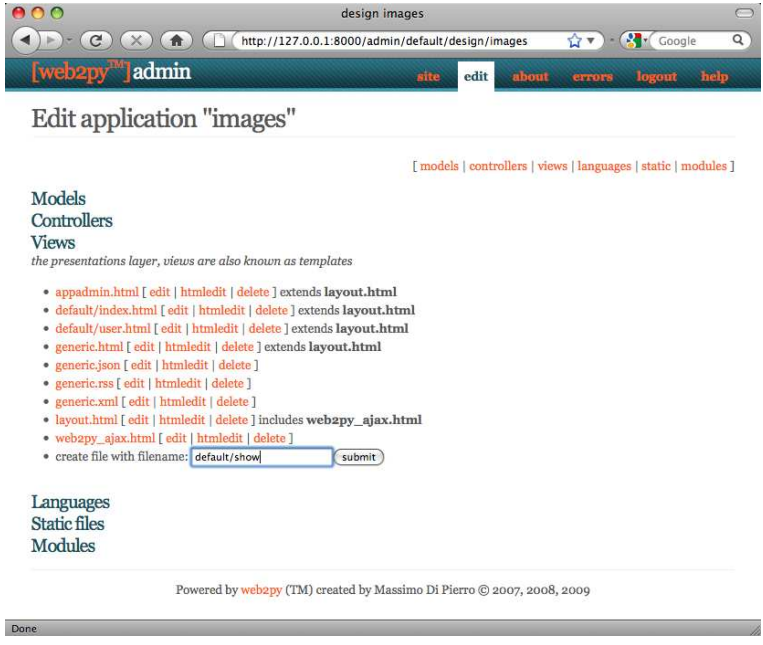
- Line 9 processes the submitted form (the submitted form variables are in `request.vars`) within the current session (the session is used to prevent double submissions, and to enforce navigation). If the submitted form variables are validated, the new comment is inserted in the `db.comment` table; otherwise the form is modified to include error messages (for example, if the author's email address is invalid). This is all done in line 9!.

- Line 10 is only executed if the form is accepted, after the record is inserted into the database table. `response.flash` is a WEB2PY variable that is displayed in the views and used to notify the visitor that something happened.

- Line 11 selects all comments that reference the current image.

The "download" action is already defined in the "default.py" controller of the scaffolding application.

The "download" action does not return a dictionary, so it does not need a view. The "show" action, though, should have a view, so return to **admin** and create a new view called "default/show.html" by typing "default/show" in the create view form:

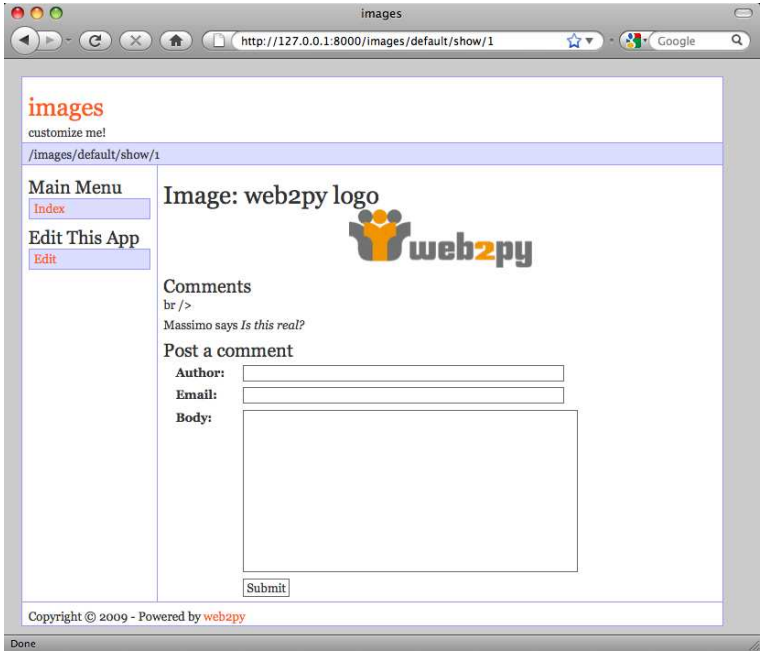


Edit this new file and replace its content with the following:

```
{{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:}}
    <p>{{=comment.author}} says <i>{{=comment.body}}</i></p>
  {{pass}}</p>
{{else:}}
  <h2>No comments posted yet</h2>
{{pass}}
<h2>Post a comment</h2>
{{=form}}
```

This view displays the **image.file** by calling the "download" action inside an `<img ... />` tag. If there are comments, it loops over them and displays each one.

Here is how everything will appear to a visitor.



When a visitor submits a comment via this page, the comment is stored in the database and appended at the bottom of the page.

3.7 Adding CRUD

WEB2PY also provides a CRUD (Create/Read/Update/Delete) API that simplifies forms even more. To use CRUD it is necessary to define it somewhere, such as in module "db.py":

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

These two lines are already in the scaffolding application.

The `crud` object provides high-level methods, for example:

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

that can be used to replace the programming pattern:

```
form = SQLFORM(...)
if form.accepts(...):
    session.flash = ...
    redirect(...)
```

Here, we rewrite the previous "show" action using crud:

```
def show():
    image = db(db.image.id==request.args(0)).select()[0]
    db.comment.image_id.default = image.id
    form = crud.create(db.image, next=URL(r=request, args=image.id),
                       message='your comment is posted')
    comments = db(db.comment.image_id==image.id).select()
    return dict(image=image, comments=comments, form=form)
```

The `next` argument of `crud.create` is the URL to redirect to after the form is accepted. The `message` argument is the one to be displayed upon acceptance. You can read more about CRUD in Chapter 7.

3.8 Adding Authentication

The WEB2PY API for Role-Based Access Control is quite sophisticated, but for now we will limit ourselves to restricting access to the show action to authenticated users, deferring a more detailed discussion to Chapter 8.

To limit access to authenticated users, we need to complete three steps. In a model, for example "db.py", we need to add:

```
from gluon.tools import Auth
auth = Auth(globals(), db)
auth.define_tables()
```

In our controller, we need to add one action:

```
def user():
    return dict(form=auth())
```

Finally, we decorate the functions that we want to restrict, for example:

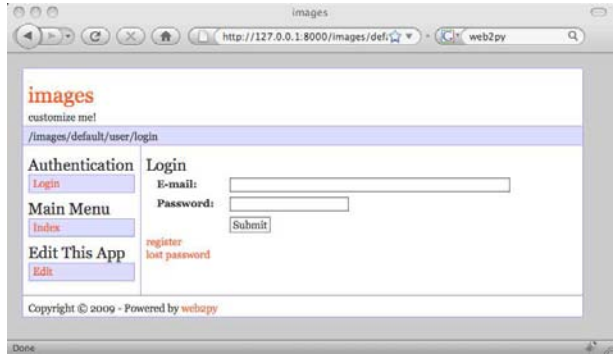
```
@auth.requires_login()
def show():
    image = db(db.image.id==request.args(0)).select()[0]
    db.comment.image_id.default = image.id
    form = crud.create(db.image, next=URL(r=request, args=image.id),
                       message='your comment is posted')
    comments = db(db.comment.image_id==image.id).select()
    return dict(image=image, comments=comments, form=form)
```

Any attempt to access

```
http://127.0.0.1:8000/images/default/show/[image_id]
```

will require login. If the user is not logged it, the user will be redirected to

```
http://127.0.0.1:8000/images/default/user/login
```



The `user` function also exposes, among others, the following actions:

```
http://127.0.0.1:8000/images/default/user/logout
http://127.0.0.1:8000/images/default/user/register
http://127.0.0.1:8000/images/default/user/profile
http://127.0.0.1:8000/images/default/user/change_password
```

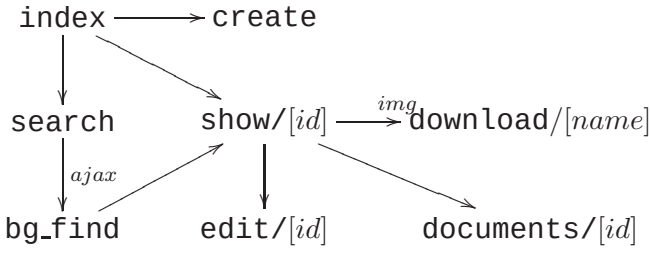
Now, a first time user needs to register in order to be able to login and read/post comments.

> *Both the* `auth` *object and the* `user` *function are already defined in the scaffolding application. The* `auth` *object is highly customizable and can deal with email verification, registration approvals, CAPTCHA, and alternate login methods via plugins.*

3.9 A Wiki

In this section, we build a wiki. The visitor will be able to create pages, search them (by title), and edit them. The visitor will also be able to post comments (exactly as in the previous applications), and also post documents (as attachments to the pages) and link them from the pages. As a convention, we adopt the Markdown syntax for our wiki syntax. We will also implement a search page with Ajax, an RSS feed for the pages, and a handler to search the pages via XML-RPC [44].

The following diagram lists the actions that we need to implement and the links we intend to build among them.



Start by creating a new scaffolding app, naming it "mywiki".

The model must contain three tables: page, comment, and document. Both comment and document reference page because they belong to page. A document contains a file field of type upload as in the previous images application.

Here is the complete model:

```
db = DAL('sqlite://storage.db')

from gluon.tools import *
auth = Auth(globals(),db)
auth.define_tables()
crud = Crud(globals(),db)

if auth.is_logged_in():
    user_id = auth.user .id
else:
    user_id = None

db.define_table('page',
    Field('title'),
    Field('body', 'text'),
    Field('created_on', 'datetime', default=request.now),
    Field('created_by', db.auth_user, default=user_id))

db.define_table('comment',
    Field('page_id', db.page),
    Field('body', 'text'),
    Field('created_on', 'datetime', default=request.now),
    Field('created_by', db.auth_user, default=user_id))

db.define_table('document',
    Field('page_id', db.page),
    Field('name'),
    Field('file', 'upload'),
    Field('created_on', 'datetime', default=request.now),
    Field('created_by', db.auth_user, default=user_id))

db.page.title.requires = [IS_NOT_EMPTY(), IS_NOT_IN_DB(db, 'page.
    title')]
db.page.body.requires = IS_NOT_EMPTY()
db.page.created_by.readable = False
```

```
db.page.created_by.writable = False
db.page.created_on.readable = False
db.page.created_on.writable = False

db.comment.page_id.requires = IS_IN_DB(db, 'page.id', '%(title)s')
db.comment.body.requires = IS_NOT_EMPTY()
db.comment.page_id.readable = False
db.comment.page_id.writable = False
db.comment.created_by.readable = False
db.comment.created_by.writable = False
db.comment.created_on.readable = False
db.comment.created_on.writable = False

db.document.page_id.requires = IS_IN_DB(db, 'page.id', '%(title)s')
db.document.name.requires = [IS_NOT_EMPTY(), IS_NOT_IN_DB(db, '
    document.name')]
db.document.page_id.readable = False
db.document.page_id.writable = False
db.document.created_by.readable = False
db.document.created_by.writable = False
db.document.created_on.readable = False
db.document.created_on.writable = False
```

Edit the controller "default.py" and create the following actions:

- index: list all wiki pages
- create: post another wiki page
- show: show a wiki page and its comments, and append comments
- edit: edit an existing page
- documents: manage the documents attached to a page
- download: download a document (as in the images example)
- search: display a search box and, via an Ajax callback, return all matching titles as the visitor types
- bg_find: the Ajax callback function. It returns the HTML that gets embedded in the search page while the visitor types.

Here is the "default.py" controller:

```
def index():
     """ this controller returns a dictionary rendered by the view
         it lists all wiki pages
     >>> index().has_key('pages')
     True
     """
     pages = db().select(db.page.id, db.page.title,
                         orderby=db.page.title)
     return dict(pages=pages)
```

```
10
11 @auth.requires_login()
12 def create():
13      "creates a new empty wiki page"
14      form = crud.create(db.page, next = URL(r=request, f='index'))
15      return dict(form=form)
16
17 def show():
18      "shows a wiki page"
19      thispage = db.page[request.args(0)]
20      if not thispage:
21         redirect(URL(r=request, f='index'))
22      db.comment.page_id.default = thispage.id
23      if user_id:
24          form = crud.create(db.comment)
25      else:
26          form = None
27      pagecomments = db(db.comment.page_id==thispage.id).select()
28      return dict(page=thispage, comments=pagecomments, form=form)
29
30 @auth.requires_login()
31 def edit():
32      "edit an existing wiki page"
33      thispage = db.page[request.args(0)]
34      if not thispage:
35         redirect(URL(r=request, f='index'))
36      form = crud.update(db.page, thispage,
37           next = URL(r=request, f='show', args=request.args))
38      return dict(form=form)
39
40 @auth.requires_login()
41 def documents():
42      "lists all documents attached to a certain page"
43      thispage = db.page[request.args(0)]
44      if not thispage:
45         redirect(URL(r=request, f='index'))
46      db.document.page_id.default = thispage.id
47      form = crud.create(db.document)
48      pagedocuments = db(db.document.page_id==thispage.id).select()
49      return dict(page=thispage, documents=pagedocuments, form=form)
50
51 def user():
52      return dict(form=auth())
53
54 def download():
55      "allows downloading of documents"
56      return response.download(request, db)
57
58 def search():
59      "an ajax wiki search page"
60      return dict(form=FORM(INPUT(_id='keyword',
61               _onkeyup="ajax('bg_find', ['keyword'], 'target');")),
62               target_div=DIV(_id='target'))
63
64 def bg_find():
65      "an ajax callback that returns a <ul> of links to wiki pages"
```

```
66      pattern = '%' + request.vars.keyword.lower() + '%'
67      pages = db(db.page.title.lower().like(pattern))\
68        .select(orderby=db.page.title)
69      items = [A(row.title, _href=URL(r=request, f=show, args=row.id))
                 \
70              for row in pages]
71      return UL(*items).xml()
```

Lines 2-6 provide a comment for the index action. Lines 4-5 inside the comment are interpreted by python as test code (doctest). Tests can be run via the admin interface. In this case the tests verify that the index action runs without errors.

Lines 19, 33, and 43 try fetch a `page` record with the id in `request.args(0)`.

Line 14, 24 and 47 define and process create forms, for a new page and a new comment and a new document respectively.

Line 36 defines and process an update form for a wiki page.

Some magic happens in line 59. The `onkeyup` attribute of the INPUT tag "keyword" is set. Every time the visitor presses a key or releases a key, the JavaScript code inside the `onkeyup` attribute is executed, client-side. Here is the JavaScript code:

```
1 ajax('bg_find', ['keyword'], 'target');
```

`ajax` is a JavaScript function defined in the file "web2py_ajax.html" which is included by the default "layout.html". It takes three parameters: the URL of the action that performs the synchronous callback ("bg_find"), a list of the IDs of variables to be sent to the callback (["keyword"]), and the ID where the response has to be inserted ("target").

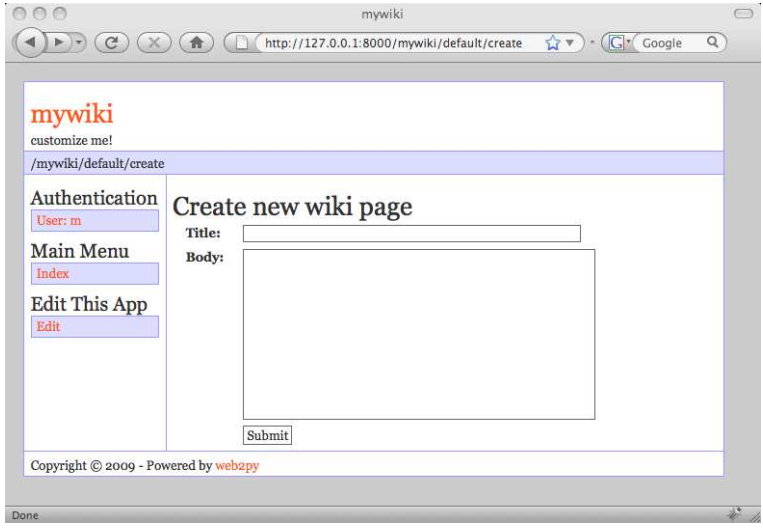
As soon as you type something in the search box and release a key, the client calls the server and sends the content of the 'keyword' field, and, when the sever responds, the response is embedded in the page itself as the innerHTML of the 'target' tag.

The 'target' tag is a DIV defined in line 75. It could have been defined in the view as well.

Here is the code for the view "default/create.html":

```
1 {{extend 'layout.html'}}
2 <h1>Create new wiki page</h1>
3 {{=form}}
```

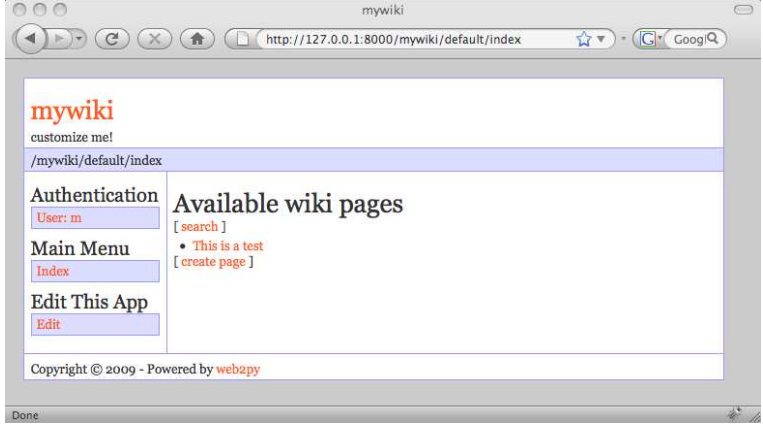
If you visit the **create** page, you see the following:



Here is the code for the view "default/index.html":

```
{{extend 'layout.html'}}
<h1>Available wiki pages</h1>
[ {{=A('search', _href=URL(r=request, f='search'))}} ]<br />
<ul>{{for page in pages:}}
     {{=LI(A(page.title, _href=URL(r=request, f='show', args=page.id)
          ))}}
{{pass}}</ul>
[ {{=A('create page', _href=URL(r=request, f='create'))}} ]
```

It generates the following page:



Here is the code for the view "default/show.html":

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

```
[ {{=A('edit', _href=URL(r=request, f='edit', args=request.args))}}
| {{=A('documents', _href=URL(r=request, f='documents', args=request.
    args))}} ]<br />
{{import gluon.contrib.markdown}}
{{=gluon.contrib.markdown.WIKI(page.body)}}
<h2>Comments</h2>
{{for comment in comments:}}
  <p>{{=db.auth_user[comment.created_by].first_name}} on {{=comment.
      created_on}}
          says <I>{{=comment.body}}</i></p>
{{pass}}
<h2>Post a comment</h2>
{{=form}}
```

WEB2PY includes `gluon.contrib.markdown.WIKI`, which knows how to convert Markdown syntax to HTML. Alternatively, you could have chosen to accept raw HTML instead of Markdown syntax. In this case you would have to replace:

```
{{=gluon.contrib.markdown.WIKI(page.body)}}
```

with:

```
{{=XML(page.body)}}
```

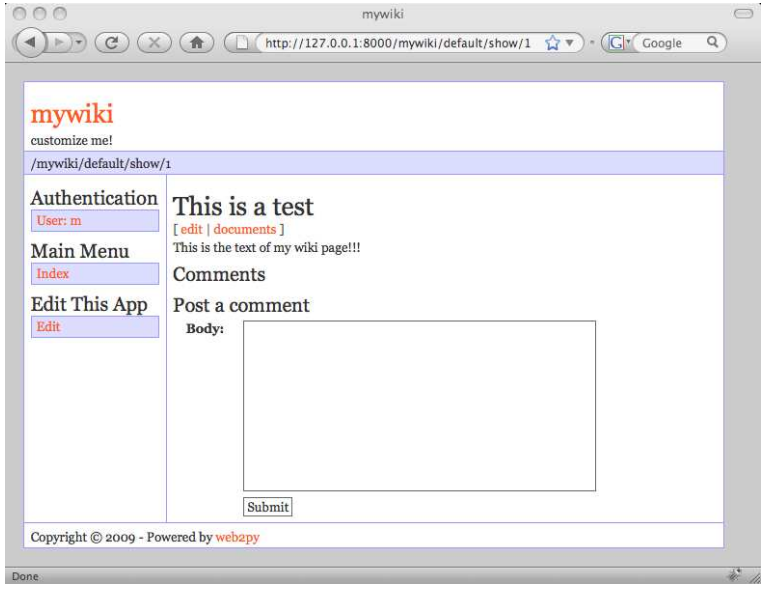
(so that the XML does not get escaped, as by default WEB2PY behavior).

This can be done better with:

```
{{=XML(page.body, sanitize=True)}}
```

By setting `sanitize=True`, you tell WEB2PY to escape unsafe XML tags such as "<script>", and thus prevent XSS vulnerabilities.

Now if, from the index page, you click on a page title, you can see the page that you have created:



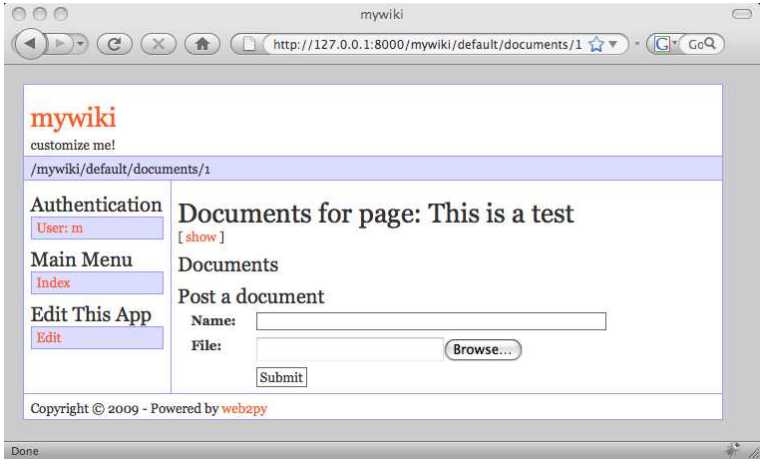
Here is the code for the view "default/edit.html":

```
{{extend 'layout.html'}}
<h1>Edit wiki page</h1>
[ {{=A('show', _href=URL(r=request, f='show', args=request.args))}}
    ]<br />
{{=form}}
```

It generates a page that looks almost identical to the create page.
Here is the code for the view "default/documents.html":

```
{{extend 'layout.html'}}
<h1>Documents for page: {{=page.title}}</h1>
[ {{=A('show', _href=URL(r=request, f='show', args=request.args))}}
    ]<br />
<h2>Documents</h2>
{{for document in documents:}}
   {{=A(document.name, _href=URL(r=request, f='download', args=
       document.file))}}
   <br />
{{pass}}
<h2>Post a document</h2>
{{=form}}
```

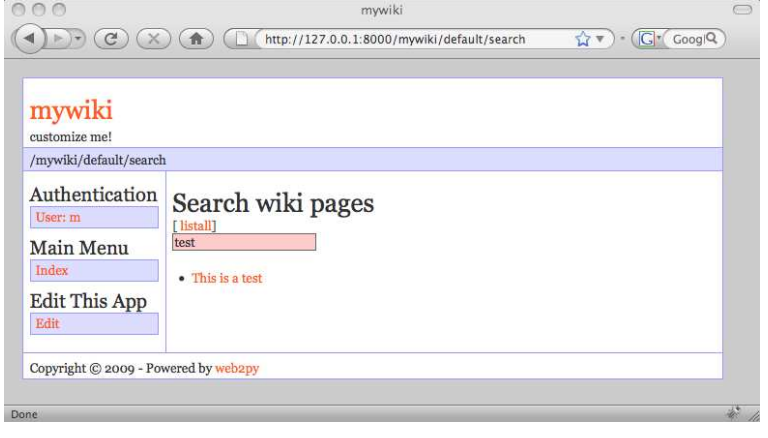
If, from the "show" page, you click on documents, you can now manage the documents attached to the page.



Finally here is the code for the view "default/search.html":

```
{{extend 'layout.html'}}
<h1>Search wiki pages</h1>
[ {{=A('listall', _href=URL(r=request, f='index'))}}]<br />
{{=form}}<br />{{=target_div}}
```

which generates the following Ajax search form:



You can also try to call the callback action directly by visiting, for example, the following URL:

```
http://127.0.0.1:8000/mywiki/default/search/keyword=wiki
```

If you look at the page source you see the HTML returned by the callback:

```
<ul><li><a href="/mywiki/default/show/4">I made a Wiki</a></li></ul>
```

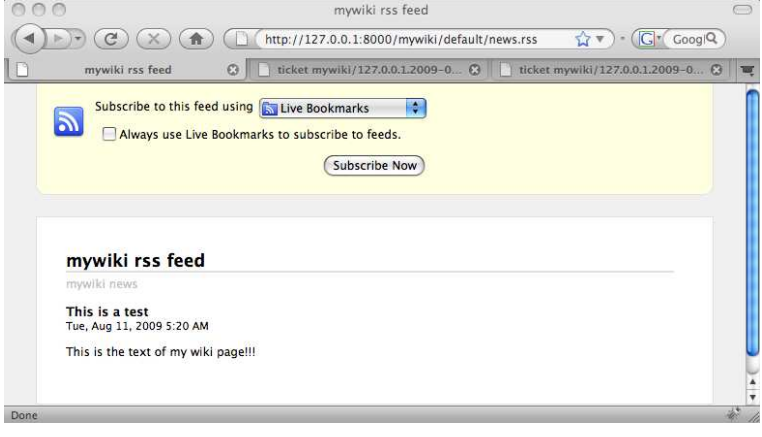
Generating an RSS feed from the stored pages using WEB2PY is easy because WEB2PY includes `gluon.contrib.rss2`. Just append the following action to the default controller:

```
def news():
    "generates rss feed form the wiki pages"
    import gluon.contrib.markdown as md
    pages = db().select(db.page.ALL, orderby=db.page.title)
    return dict(
       title = 'mywiki rss feed',
       link = 'http://127.0.0.1:8000/mywiki/default/index',
       description = 'mywiki news',
       created_on = request.now,
       items = [
          dict(title = row.title,
               link = URL(r=request, f='show', args=row.id),
               description = md.WIKI(row.body).xml(),
               created_on = row.created_on
               ) for row in pages]
       )
```

and when you visit the page

```
http://127.0.0.1:8000/mywiki/default/news.rss
```

you see the feed (the exact output depends on the feed reader). Notice that the dict is automatically converted to RSS, thanks to the .rss extension in the URL.



WEB2PY also includes feedparser to read third-party feeds.

Finally, let's add an XML-RPC handler that allows searching the wiki programmatically:

```
service=Service(globals())

@service.xmlrpc()
def find_by(keyword):
     "finds pages that contain keyword for XML-RPC"
     return db(db.page.title.lower().like('%' + keyword + '%'))\
         .select().as_list()

def call():
```

```
    "exposes all registered services, including XML-RPC"
    return service()
```

Here, the handler action simply publishes (via XML-RPC), the functions specified in the list. In this case, `find_by`. `find_by` is not an action (because it takes an argument). It queries the database with `.select()` and then extracts the records as a list with `.response` and returns the list.

Here is an example of how to access the XML-RPC handler from an external Python program.

```
>>> import xmlrpclib
>>> server = xmlrpclib.ServerProxy(
    'http://127.0.0.1:8000/mywiki/default/call/xmlrpc')
>>> for item in server.find_by('wiki'):
        print item.created_on, item.title
```

The handler can be accessed from many other programming languages that understand XML-RPC, including C, C++, C# and Java.

3.10 More on admin

The administrative interface provides additional functionality that we briefly review here.

[site]

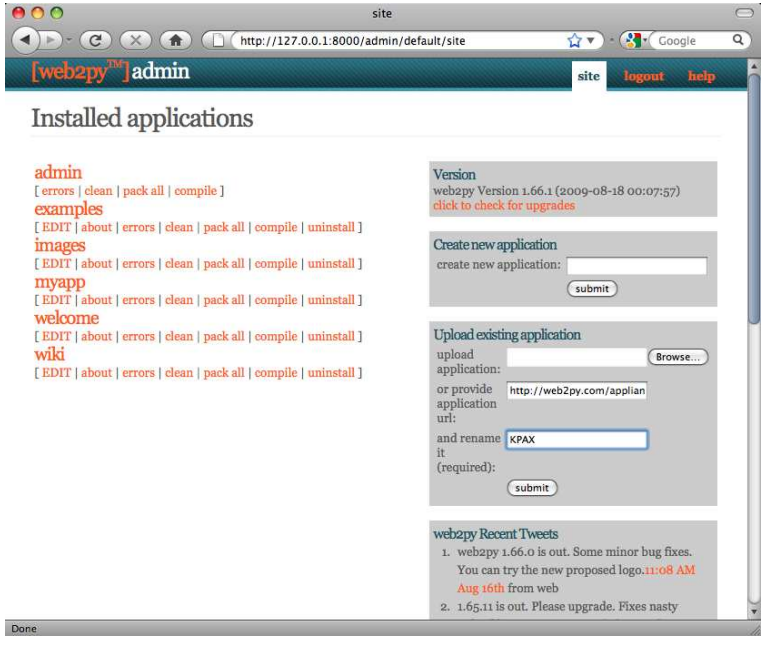
This page lists all installed applications. There are two forms at the bottom.

The first of them allows creating a new application by specifying its name.

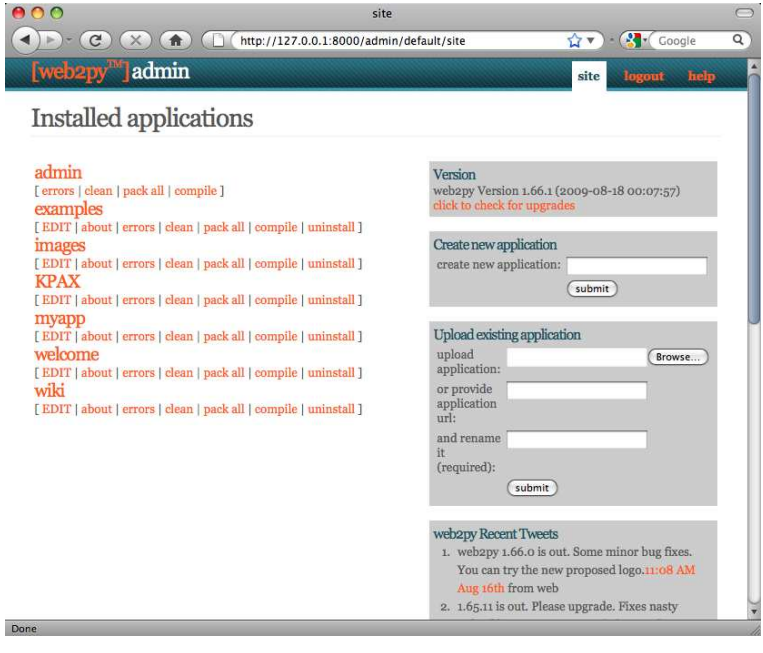
The second form allows uploading an existing application from either a local file or a remote URL. When you upload an application, you need to specify a name for it. This can be its original name, but does not need to be. This allows installing multiple copies of the same application. You can try, for example, to upload the KPAX content management system from:

```
http://web2py.com/appliances/default/download/app.source
    .221663266939.tar
```

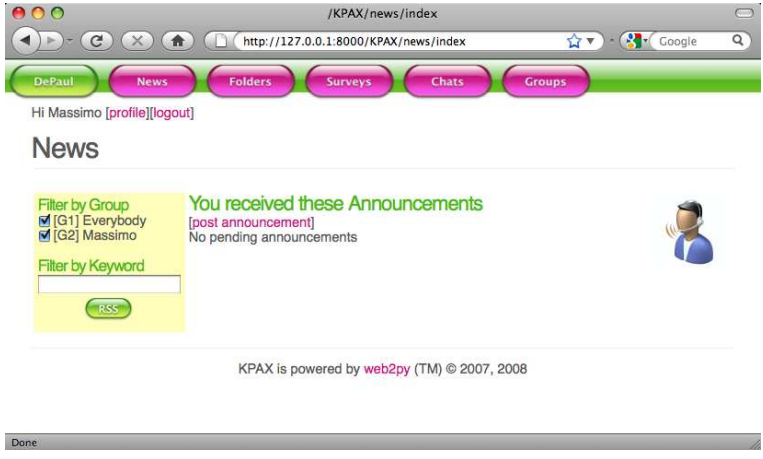
Uploaded applications can be `.tar` files (old convention) and `.w2p` files (new convention). The latter ones are gzipped tar files. They can be uncompressed manually with `tar zxvf [filename]` although this is never necessary.



Upon successful upload, WEB2PY displays the MD5 checksum of the uploaded file. You can use it to verify that the file was not corrupted during upload.



Click on the KPAX name on admin to get it up and running.



Application files are stored as w2p files (tar gzipped), but you are not intended to tar or untar them manually; WEB2PY *does it for you.*

For each application the [site] page allows you to:

- Uninstall the application.
- Jump to the [about] page (read below).
- Jump to the [EDIT] page (read below).
- Jump to the [errors] page (read below).
- Clean up temporary files (sessions, errors, and cache.disk files).
- Pack all. This returns a tar file containing a complete copy of the application. We suggest that you clean up temporary files before packing an application.
- Compile the application. If there are no errors, this option will bytecode-compile all models, controllers and views. Because views can extend and include other views in a tree, before bytecode compilation, the view tree for every controller is collapsed into a single file. The net effect is that a bytecode-compiled application is faster, because there is no more parsing of templates or string substitutions occurring at runtime.
- Pack compiled. This option is only present for bytecode-compiled applications. It allows packing the application without source code for

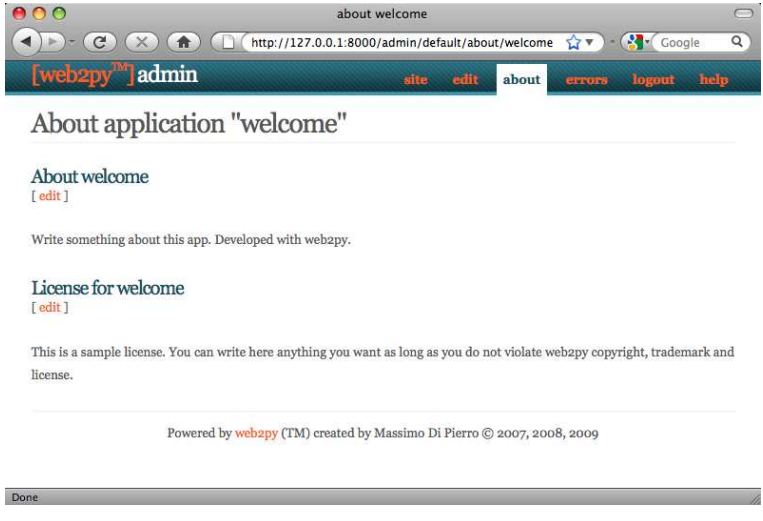
distribution as closed source. Note that Python (as any other programming language) can technically be decompiled; therefore compilation does not provide complete protection of the source code. Nevertheless, decompilation can be difficult and can be illegal.

- Remove compiled. It simply removes the byte-code compiled models, views and controllers from the application. If the application was packaged with source code or designed locally, there is no harm in removing the bytecode-compiled files, and the application will continue to work. If the application was installed form a packed compiled file, then this is not safe, because there is no source code to revert to, and the application will no longer work.

All the functionality available from the WEB2PY *admin site page is also accessible programmatically via the API defined in the module* `gluon/admin.py`. *Simply open a python shell and import this module.*

[about]

The [about] tab allows editing the description of the application and its license. These are written respectively in the ABOUT and LICENSE files in the application folder.



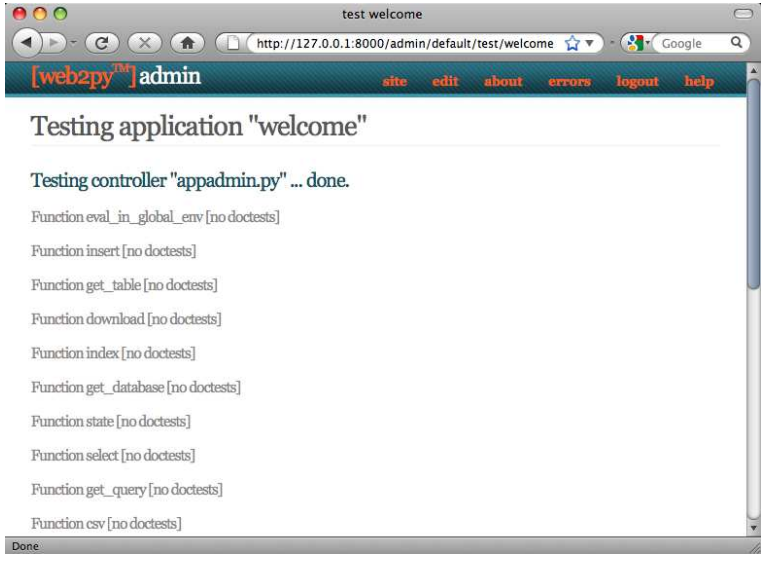
You can use Markdown syntax for these files as described in ref. [28].

[EDIT]

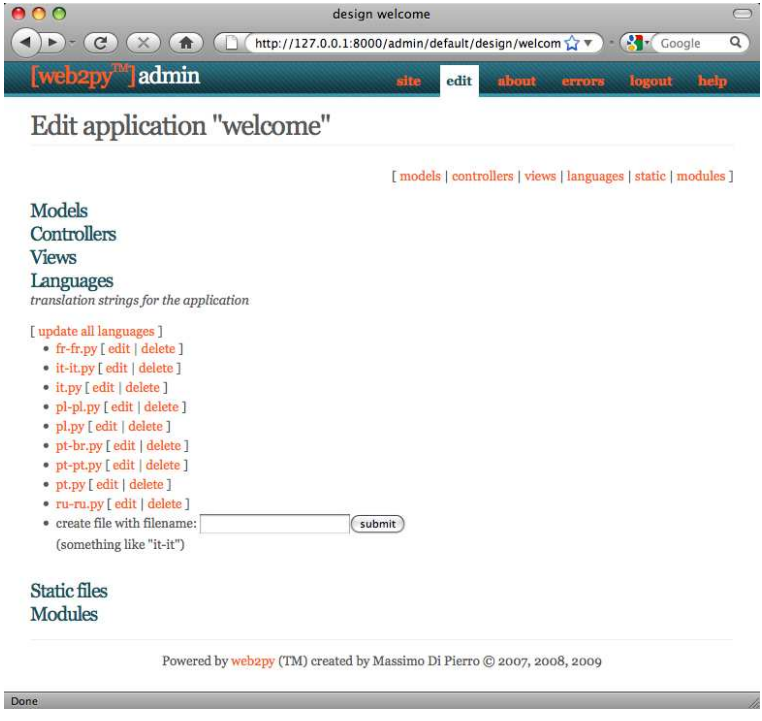
You have used the [EDIT] page already in this chapter. Here we want to point out a few more functionalities of the [EDIT] page.

- If you click on any file name, you can see the content of the file with syntax highlighting.
- If you click on edit, you can edit the file via a web interface.
- If you click on delete, you can delete the file (permanently).
- If you click on test, WEB2PY will run tests. Tests are written by the developer using Python doctests, and each function should have its own tests.
- View files have an htmledit link that allows editing the view using a web-based WYSIWYG editor.
- You can add language files, scan the app to discover all strings, and edit string translations via the web interface.
- If the static files are organized in folders and subfolders, the folder hierarchy can be toggled by clicking on a folder name.

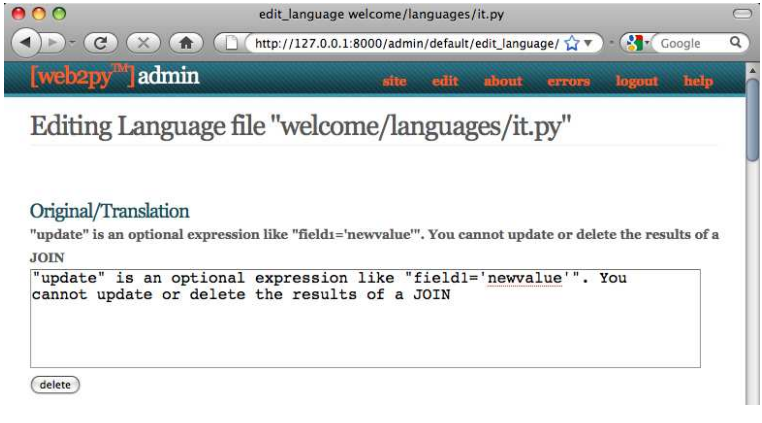
The image below shows the output of the test page for the welcome application.



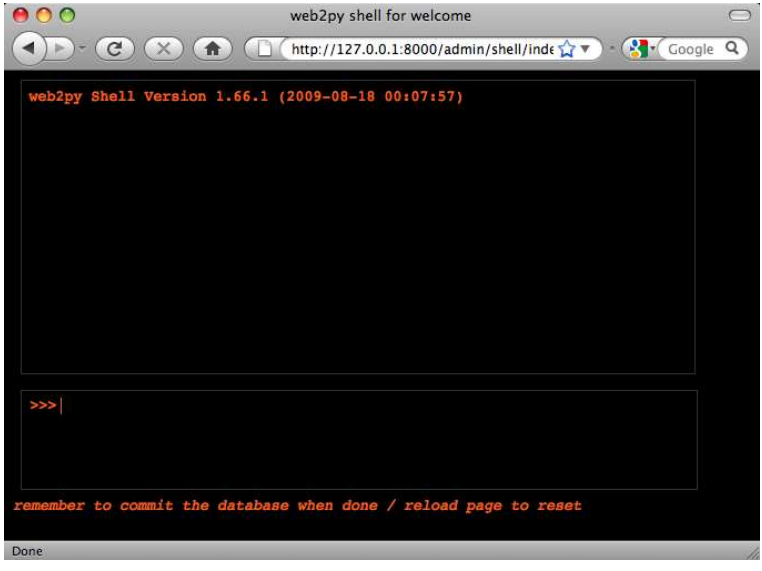
The image below show the languages tab for the welcome application.



The image below shows how to edit a language file, in this case the "it" (Italian) language for the welcome application.



shell If you click on the "shell" link under the controllers tab in [EDIT], WEB2PY will open a web based Python shell and will execute the models for the current application. This allows you to interactively talk to your application.



crontab Also under the controllers tab in [EDIT] there is a "crontab" link. By clicking on this link you will be able to edit the WEB2PY crontab file. This follows the same syntax as the unix crontab but does not rely on unix. In fact, it only requires WEB2PY and it works on Windows too. It allows you to register actions that need to be executed in background as scheduled times. For more information about this we refer to the next chapter.

[errors]

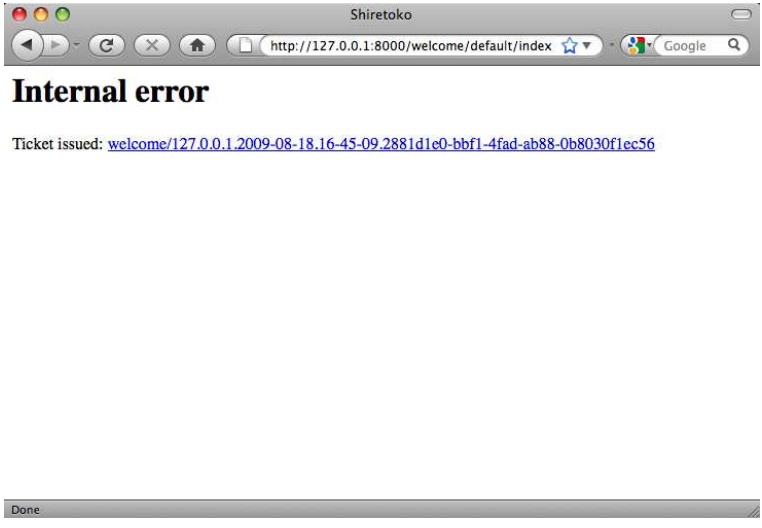
When programming WEB2PY, you will inevitably make mistakes and introduce bugs. WEB2PY helps in two ways: 1) it allows you to create tests for every function that can be run in the browser from the [EDIT] page; and 2) when an error manifests itself, a ticket is issued to the visitor and the error is logged.

Purposely introduce an error in the images application as shown below:

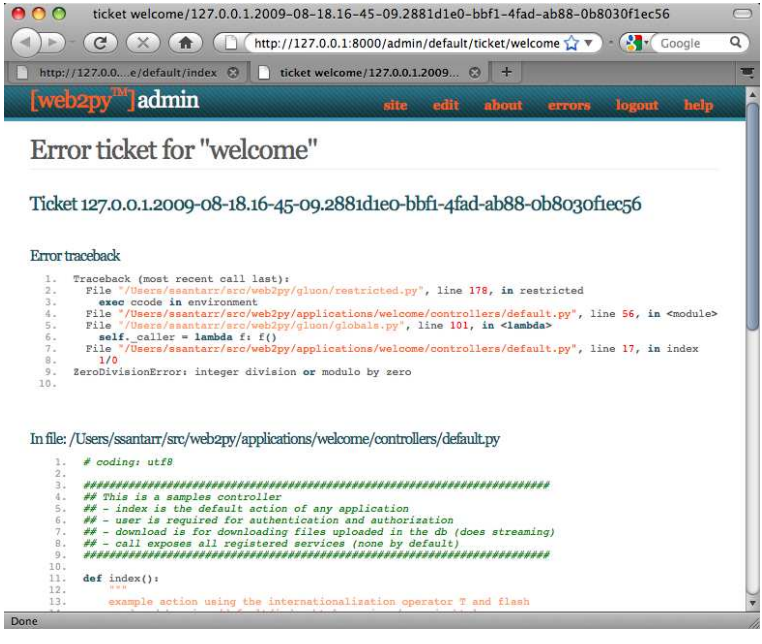
```
1 def index():
2     images = db().select(db.image.ALL,orderby=db.image.title)
```

```
    1/0
    return dict(images=images)
```

When you access the index action, you get the following ticket:



Only the administrator can access the ticket:



The ticket shows the traceback, and the content of the file that caused the problem. If the error occurs in a view, WEB2PY shows the view converted

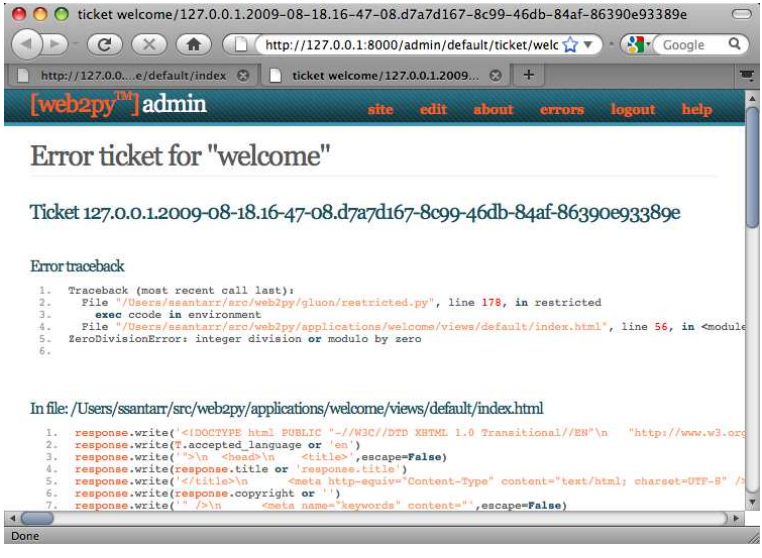
from HTML into Python code. This allows to easily identify the logical structure of the file.

Notice that everywhere **admin** shows syntax-highlighted code (for example, in error reports, WEB2PY keywords are shown in orange). If you click on a WEB2PY keyword, you are redirected to a documentation page about the keyword.

If you fix the 1/0 bug in the index action and introduce one in the index view:

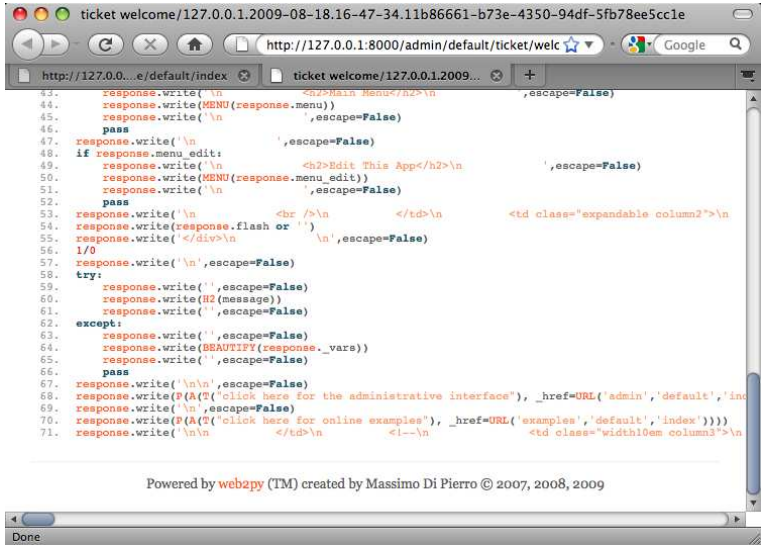
```
{{extend 'layout.html'}}

<h1>Current Images</h1>
<ul>
{{for image in images:}}
{{1/0}}
{{=LI(A(image.title, _href=URL(r=request, f="show", args=image.id)))
    }}
{{pass}}
</ul>
```

you get the following ticket:



Note that WEB2PY has converted the view from HTML into a Python file, and thus, the error described in the ticket refers to the generated Python code and NOT to the original view file.

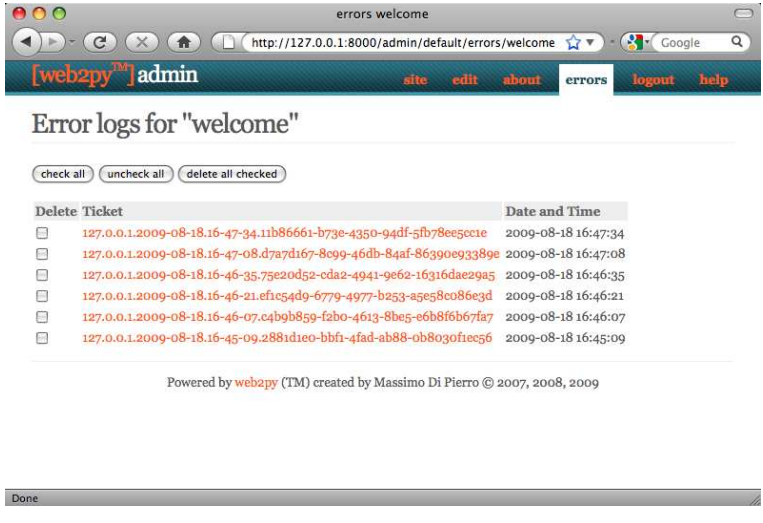
you get the following ticket:



This may seem confusing at first, but in practice it makes debugging easier, because the Python indentation highlights the logical structure of the code that you embedded in the views.

The code is shown at the bottom of the same page.

All tickets are listed under admin in the [errors] page for each application:



[mercurial]

If you are running from source and you have the Mercurial version control libraries installed:

```
easy_install mercurial
```

then the administrative interface shows one more menu item called "mercurial". It automatically creates a local Mercurial repository for the application. Pressing the "commit" button in the page will commit the current application.

This feature is experimental and will be improved in the future.

3.11 More on appadmin

appadmin is not intended to be exposed to the public. It is designed to help you by providing an easy access to the database. It consists of only two files: a controller "appadmin.py" and a view "appadmin.html" which are used by all actions in the controller.

The **appadmin** controller is relatively small and readable; it provides an example on designing a database interface.

appadmin shows which databases are available and which tables exist in each database. You can insert records and list all records for each table individually. **appadmin** paginates output 100 records at a time.

Once a set of records is selected, the header of the pages changes, allowing you to update or delete the selected records.

To update the records, enter an SQL assignment in the Query string field:

```
title = 'test'
```

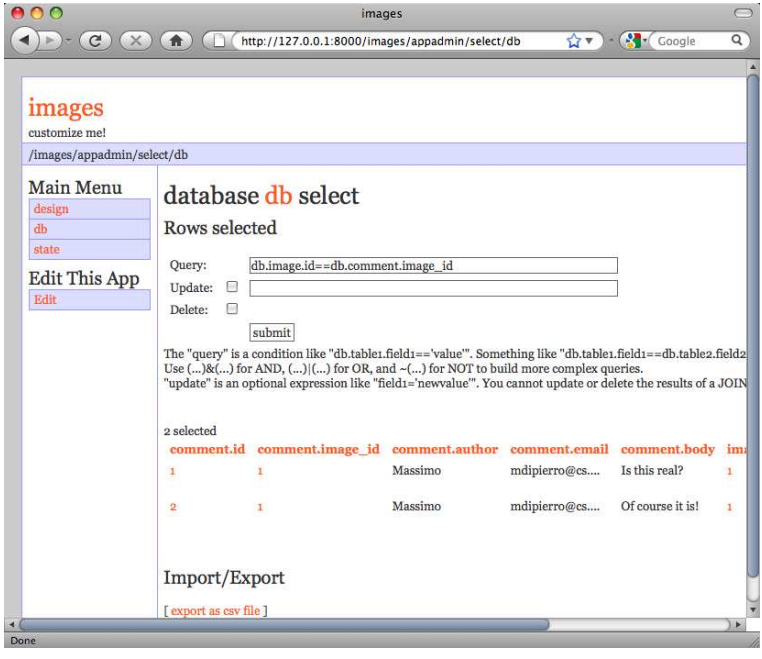
where string values must be enclosed in single quotes. Multiple fields can be separated by commas.

To delete a record, click the corresponding checkbox and confirm that you are sure.

appadmin can also perform joins if the SQL FILTER contains a SQL condition that involves two or more tables. For example, try:

```
db.image.id == db.comment.image_id
```

WEB2PY passes this along to the DAL, and it understands that the query links two tables; hence, both tables are selected with an INNER JOIN. Here is the output:



If you click on the number of an id field, you get an edit page for the record with the corresponding id.

If you click on the number of a reference field, you get an edit page for the referenced record.

You cannot update or delete rows selected by a join because they involve records from multiple tables and this would be ambiguous.

CHAPTER 4

THE CORE

4.1 Command Line Options

It is possible to skip the GUI and start WEB2PY directly from the command line by typing something like:

```
python web2py.py -a 'your password' -i 127.0.0.1 -p 8000
```

When WEB2PY starts, it creates a file called "parameters_8000.py" where it stores the hashed password. If you use "<ask>" as the password, WEB2PY prompts you for it.

For additional security, you can start WEB2PY with:

```
python web2py.py -a '<recycle>' -i 127.0.0.1 -p 8000
```

In this case WEB2PY reuses the previously stored hashed password. If no password is provided, or if the "parameters_8000.py" file is deleted, the web-based administrative interface is disabled.

> *WEB2PY normally runs with CPython (the C implementation of the Python interpreter created by Guido van Rossum), but it can also run with Jython (the Java implementation of the interpreter). The latter possibility allows the use of WEB2PY in the context of a J2EE infrastructure. To use Jython, simply replace "python web2py.py ..." with "jython web2py.py". Details about installing Jython, zxJDBC modules required to access the databases can be found in Chapter 12.*

The "web2py.py" script can take many command-line arguments specifying the maximum number of threads, enabling of SSL, etc. For a complete list type:

```
>>> python web2py.py -h
Usage: python web2py.py

web2py Web Framework startup script. ATTENTION: unless a password
is specified (-a 'passwd'), web2py will attempt to run a GUI.
In this case command line options are ignored.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -i IP, --ip=IP        ip address of the server (127.0.0.1)
  -p PORT, --port=PORT  port of server (8000)
  -a PASSWORD, --password=PASSWORD
                        password to be used for administration
                        use -a "<recycle>" to reuse the last
                        password
  -u UPGRADE, --upgrade=UPGRADE
                        -u yes: upgrade applications and exit
  -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 to store the pid of the server
  -l LOG_FILENAME, --log_filename=LOG_FILENAME
                        file to log connections
  -n NUMTHREADS, --numthreads=NUMTHREADS
                        number of threads
  -s SERVER_NAME, --server_name=SERVER_NAME
                        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 (10 seconds)
  -z SHUTDOWN_TIMEOUT, --shutdown_timeout=SHUTDOWN_TIMEOUT
                        timeout on shutdown of server (5 seconds)
  -f FOLDER, --folder=FOLDER
                        folder from which to run web2py
  -v, --verbose         increase --test verbosity
  -Q, --quiet           disable all output
```

```
  -D DEBUGLEVEL, --debug=DEBUGLEVEL
                        set debug output level (0-100, 0 means all,
                        100 means none; default is 30)
  -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
  -R PYTHON_FILE, --run=PYTHON_FILE
                        run PYTHON_FILE in web2py environment;
                        should be used with --shell option
  -T TEST_PATH, --test=TEST_PATH
                        run doctests in web2py environment;
                        TEST_PATH like a/c/f (c,f optional)
  -W WINSERVICE, --winservice=WINSERVICE
                        -W install|start|stop as Windows service
  -C, --cron            trigger a cron run manually; usually invoked
                        from a system crontab
  -N, --no-cron         do not start cron automatically
  -L CONFIG, --config=CONFIG
                        config file
  -F PROFILER_FILENAME, --profiler=PROFILER_FILENAME
                        profiler filename
  -t, --taskbar         use web2py gui and run in taskbar
                        (system tray)
```

Lower-case options are used to configure the web server. The `-L` option tells WEB2PY to read configuration options from a file, `-W` installs WEB2PY as a windows service, while `-S`, `-P` and `-M` options start an interactive Python shell. The `-T` option finds and runs controller doctests in a WEB2PY execution environment. For example, the following example runs doctests from all controllers in the "welcome" application:

```
python web2py.py -vT welcome
```

In the WEB2PY folder there is a sample "options_std.py" configuration file for the internal web server:

```
import socket, os
ip = '127.0.0.1'
port = 8000
password = '<recycle>' ### <recycle> means use the previous password
pid_filename = 'httpserver.pid'
log_filename = 'httpserver.log'
ssl_certificate = '' ### path to certificate file
ssl_private_key = '' ### path to private key file
numthreads = 10
server_name = socket.gethostname()
request_queue_size = 5
timeout = 10
shutdown_timeout = 5
folder = os.getcwd()
```

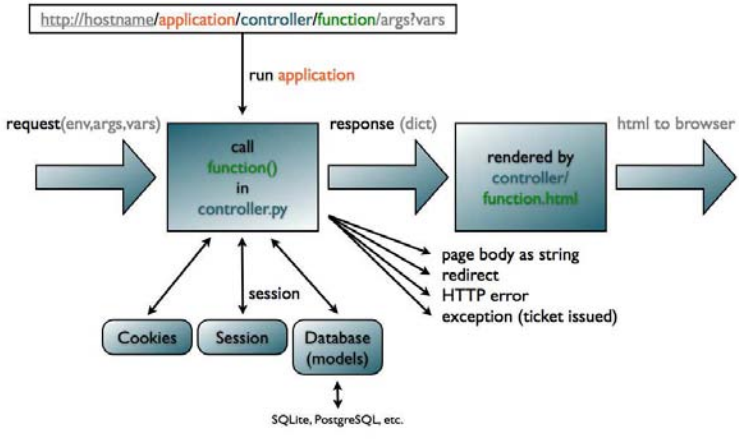
This file contains the WEB2PY defaults. If you edit this file, you need to import it explicitly with the `-L` command-line option.

4.2 URL Mapping

WEB2PY maps a URL of the form:

```
http://127.0.0.1:8000/a/c/f.html
```

to the function `f()` in controller "c.py" in application "a". If `f` is not present, WEB2PY defaults to the `index` controller function. If `c` is not present, WEB2PY defaults to the "default.py" controller, and if `a` is not present, WEB2PY defaults to the `init` application. If there is no `init` application, WEB2PY tries to run the `welcome` application. This is shown schematically in the image below:



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

The extension `.html` is optional; `.html` is assumed as default. The extension determines the extension of the view that renders the output of the controller function `f()`. It allows the same content to be served in multiple formats (html, xml, json, rss, etc.).

There is an exception made for URLs of the form:

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

There is no controller called "static". WEB2PY interprets this as a request for the file called "filename" in the subfolder "static" of the application "a".

When static files are downloaded, WEB2PY does not create a session, nor does it issue a cookie or execute the models. WEB2PY always streams static files in chunks of 1MB, and sends PARTIAL CONTENT when the client sends a RANGE request for a subset of the file. WEB2PY also supports the IF_MODIFIED_SINCE protocol, and does not send the file if it is already stored in the browser's cache and if the file has not changed since that version.

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

WEB2PY maps GET/POST requests of the form:

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

to function `f` in controller "c.py" in application `a`, and it stores the URL 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'
```

In the above example, both `request.args[i]` and `request.args(i)` can be used to retrieve the i-th element of the `request.args`, but while the former raises an exception if the list does not have such an index, the latter returns None in this case.

```
request.url
```

stores the full URL of the current request (not including GET variables). It is the same as:

```
URL(r=request,args=request.args)
```

If the HTTP request is a GET, then `request.env.request_method` is set to "GET"; if it is a POST, `request.env.request_method` is set to "POST". URL query variables are stored in the `request.vars` Storage dictionary; they are also stored in `request.get_vars` (following a GET request) or `request.post_vars` (following a POST request).

WEB2PY stores WSGI and WEB2PY 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* WEB2PY *validates all URLs to prevent directory traversal attacks.*

URLs are only allowed to contain alphanumeric characters, underscores, slashes; the `args` may contain non-consecutive dots. Speces are replaced by underscores before validation. If the URL syntax is invalid, WEB2PY returns an HTTP 400 error message [45, 46].

If the URL corresponds to a request for a static file, WEB2PY simply reads and returns (streams) the requested file.

If the URL does not request a static file WEB2PY processes the request in the following order:

- Parses cookies.
- Creates an environment in which to execute the function.
- Initializes `request`, `response`, `cache`.
- Opens the existing `session` or creates a new one.
- Executes the models belonging to the requested application.
- Executes the requested controller action function.
- If the function returns a dictionary, executes the associated view.
- On success, commits all open transactions.
- Saves the session.
- Returns an HTTP response.

Notice that the controller and the view are executed in different copies of the same environment; therefore, the view does not see the controller, but it sees the models and it sees the variables returned by the controller action function.

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 the intended behavior (for example, an HTTP redirect), and all open database transactions are committed. The behavior after that is specified by the HTTP exception itself. The HTTP exception class is not a standard Python exception; it is defined by WEB2PY.

4.3 Libraries

The WEB2PY libraries are exposed to the user applications as global objects. For example (`request`, `response`, `session`, `cache`), classes (helpers, validators, DAL API), and functions (`T` and `redirect`).

These objects are defined in the following core files:

```
web2py.py
gluon/__init__.py
gluon/admin.py
gluon/cache.py
gluon/compileapp.py
gluon/contenttype.py
gluon/fileutils.py
gluon/globals.py
gluon/highlight.py
gluon/html.py
gluon/http.py
gluon/import_all.py
gluon/languages.py
gluon/main.py
gluon/myregex.py
gluon/portalocker.py
gluon/restricted.py
gluon/rewrite.py
gluon/sanitizer.py
gluon/serializers.py
gluon/settings.py
gluon/shell.py
gluon/sql.py
gluon/sqlhtml.py
gluon/storage.py
gluon/streamer.py
gluon/template.py
gluon/tools.py
gluon/utils.py
gluon/validators.py
gluon/widget.py
gluon/winservice.py
gluon/wsgiserver.py
gluon/xmlrpc.py
```

The tar gzipped apps that ship with WEB2PY are in

```
admin.w2p
```

```
examples.w2p
welcome.w2p
```

The first time you start WEB2PY*, two new folders are created: deposit and applications. The three w2p files above are unzipped in the applications folder. The deposit folder is used as temporary storage for installing and uninstalling applications.*

WEB2PY unittests are in

```
gluon/tests/
```

Handlers for connecting with various web servers:

```
cgihandler.py
gaehandler.py
fcgihandler.py
wsgihandler.py
modpythonhandler.py
gluon/contrib/gateways/__init__.py
gluon/contrib/gateways/fcgi.py
```

(fcgi.py was developed by Allan Saddi)

Two example files:

```
options_std.py
routes.example.py
```

The former is an optional configuration file that can be passed to web2py.py with the `-L` option. The second is an example of a URL mapping file. It is loaded automatically when renamed "routes.py".

The files

```
app.yaml
index.yaml
```

are configuration files necessary for deployment on the Google App Engine. You probably do not need to modify them, but you can read more about them on the Google Documentation pages.

There are also additional libraries, usually developed by a third party:

feedparser [27] by Mark Pilgrim for reading RSS and Atom feeds:

```
gluon/contrib/__init__.py
gluon/contrib/feedparser.py
```

markdown2 [28] by Trent Mick for wiki markup:

```
gluon/contrib/markdown/__init__.py
gluon/contrib/markdown/markdown2.py
```

memcache [29] Python API by Evan Martin:

```
gluon/contrib/memcache/__init__.py
gluon/contrib/memcache/memcache.py
```

gql, a port of the DAL to the Google App Engine:

```
gluon/contrib/gql.py
```

memdb, a port of the DAL on top of memcache:

```
gluon/contrib/memdb.py
```

gae_memcache is an API to use memcache on the Google App Engine:

```
gluon/contrib/gae_memcache.py
```

pyrtf [25] for generating Rich Text Format (RTF) documents, developed by Simon Cusack and revised by Grant Edwards:

```
gluon/contrib/pyrtf
gluon/contrib/pyrtf/__init__.py
gluon/contrib/pyrtf/Constants.py
gluon/contrib/pyrtf/Elements.py
gluon/contrib/pyrtf/PropertySets.py
gluon/contrib/pyrtf/README
gluon/contrib/pyrtf/Renderer.py
gluon/contrib/pyrtf/Styles.py
```

PyRSS2Gen [26] developed by Dalke Scientific Software, to generate RSS feeds:

```
gluon/contrib/rss2.py
```

simplejson [24] by Bob Ippolito, the standard library for parsing and writing JSON objects:

```
gluon/contrib/simplejson/__init__.py
gluon/contrib/simplejson/decoder.py
gluon/contrib/simplejson/encoder.py
gluon/contrib/simplejson/jsonfilter.py
gluon/contrib/simplejson/scanner.py
```

cron and **wsgihooks** are required for executing cron jobs and tasks that must be executed after a page is served.

```
gluon/contrib/cron.py
gluon/contrib/wsgihooks.py
```

A file that allows interaction with the taskbar in windows, when WEB2PY is running as a service:

```
gluon/contrib/taskbar_widget.py
```

Optional **login_methods** to be used for authentication:

```
gluon/contrib/login_methods/__init__.py
gluon/contrib/login_methods/basic_auth.py
gluon/contrib/login_methods/cas_auth.py
gluon/contrib/login_methods/email_auth.py
gluon/contrib/login_methods/gae_google_account.py
gluon/contrib/login_methods/ldap_auth.py
```

WEB2PY also contains a folder with useful scripts:

```
scripts/cleancss.py
scripts/cleanhtml.py
scripts/contentparser.py
scripts/repair.py
scripts/sessions2trash.py
scripts/sync_languages.py
scripts/tickets2db.py
scripts/web2py.archlinux.sh
scripts/web2py.fedora.sh
scripts/web2py.ubuntu.sh
scripts/web2py-wsgi.conf
```

These are discussed in Chapter 12, but they are more or less self-documenting.

Finally WEB2PY includes these files required to build the binary distributions.

```
Makefile
setup_exe.py
setup_app.py
```

These are setup scripts for **py2exe** and **py2app** respectively and they are only required to build the binary distributions of WEB2PY.

In summary, WEB2PY libraries provide the following functionality:

- 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 persistence (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, MS SQL, Firebird, PostgreSQL, or Oracle). This abstraction layer is referred to as the Database Abstraction Layer (DAL).
- 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 XML-RPC web services.
- Upload and download large files via streaming.

WEB2PY applications contain additional files, particularly third-party JavaScript libraries, such as jQuery, calendar, EditArea and nicEdit. Their authors are acknowledged in the files themselves.

4.4 Applications

Applications developed in WEB2PY are composed of the following parts:

- **models** describe a representation of the data as database tables and relations between 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 (e.g. images, CSS stylesheets, etc).
- **ABOUT** and **README** documents are self-explanatory.
- **errors** store error reports generated by the application.
- **sessions** store information related to each particular user.
- **databases** store SQLite databases and additional table information.
- **cache** store cached application items.
- **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).
- **tests** is a directory for storing test scripts, fixtures and mocks.

Models, views, controllers, languages, and static files are accessible via the web 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, although the user never needs to access the filesystem directly:

```
ABOUT        databases    languages  modules   static   views
cache        errors       LICENSE    private   tests    cron
controllers  __init__.py  models     sessions  uploads
```

"__init__.py" is an empty file which is required in order to allow Python (and WEB2PY) to import the modules in the `modules` directory.

Notice that the **admin** application simply provides a web interface to WEB2PY applications on the server file system. WEB2PY applications can also be created and developed from the command-line; you don't have to use the browser **admin** interface. A new application can be created manually by replicating the above directory structure under ,e.g., "applications/newapp/" (or simply untar the `welcome.w2p` file into your new application directory). Application files can also be created and edited from the command-line without having to use the web **admin** interface.

4.5 API

Models, controllers, and views are executed in an environment where the following objects are already imported for us:

Global Objects

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

Navigation

```
redirect, HTTP
```

Internationalization

```
T
```

Helpers

```
XML, URL, BEAUTIFY

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

Validators

```
IS_ALPHANUMERIC, IS_DATE, IS_DATETIME, IS_EMAIL,
IS_EXPR, IS_FLOAT_IN_RANGE, IS_IMAGE, IS_INT_IN_RANGE, IS_IN_SET,
IS_IPV4, IS_LENGTH, IS_LOWER, IS_MATCH, IS_NULL_OR, IS_NOT_EMPTY,
IS_TIME, IS_URL, IS_UPLOAD_FILENAME, IS_LIST_OF, IS_UPPER,
IS_STRONG, CLEANUP, CRYPT, IS_IN_DB, IS_NOT_IN_DB
```

Database

```
DAL, Field
```

For backward compatibility `SQLDB=DAL` and `SQLField=Field`. We encourage you to use the new syntax `DAL` and `Field`, instead of the old syntax.

Other objects and modules are defined in the libraries, but they are not automatically imported since they are not used as often.

The core API entities in the WEB2PY execution environment are `request`, `response`, `session`, `cache`, `URL`, `HTTP`, `redirect` and `T` and are discussed below.

A few objects and functions, including **Auth**, **Crud** and **Service**, are defined in "gluon/tools.py" and they need to be imported is necessary:

```
from gluon.tools import Auth, Crud, Service
```

4.6 request

The `request` object is an instance of the ubiquitous WEB2PY class that is called `gluon.storage.Storage`, which extends the Python `dict` class. It is basically a dictionary, but the item values can also be accessed as attributes:

```
request.vars
```

is the same as:

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

Unlike a dictionary, if an attribute (or key) does not exist, it does not raise an exception. Instead, it returns `None`.

`request` has the following items/attributes, some of which are also an instance of the `Storage` class:

- **request.cookies**: a `Cookie.SimpleCookie()` object containing the cookies passed with the HTTP request. It acts like a dictionary of cookies. Each cookie 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. The environment variables are all converted to lower case, and dots are converted to underscores for easier memorization.
- **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.extension**: the extension of the requested action. It defaults to "html". If the controller function returns a dictionary and does not specify a view, this is used to determine the extension of the view file that will render the dictionary (parsed from the `request.env.path_info`).
- **request.folder**: the application directory. For example if the application is "welcome", `request.folder` is set to the absolute path "/path/-to/welcome". In your programs, you should always use this variable and the `os.path.join` function to build paths to the files you need to access. Although WEB2PY always uses absolute paths, it is a good rule never to explicitly change the current working folder (whatever that is) since this is not a thread-safe practice.
- **request.now**: a `datetime.datetime` object storing the timestamp of the current request.
- **request.args**: A list of the URL path components following the controller function name; equivalent to
  `request.env.path_info.split('/')[3:]`
- **request.vars**: a `gluon.storage.Storage` object containing the HTTP GET and HTTP POST query variables.
- **request.get_vars**: a `gluon.storage.Storage` object containing only the HTTP GET query variables.

- **request.post_vars**: a `gluon.storage.Storage` object containing only the HTTP POST query variables.
- **request.client**: The ip address of the client as determined by `request.env.remote_addr` or `request.env.http_x_forwarded_for` if present. While this is useful it should not be trusted because the `http_x_forwarded_for` can be spoofed.
- **request.body**: a readonly file stream that contains the body of the HTTP request. This is automatically parsed to get the `request.post_vars` and then rewinded. It can be read with `request.body.read()`.

As an example, the following call on a typical system:

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

results in table 4.1

Which environment variables are actually defined depends on the web server. Here we are assuming the built-in cherrypy wsgi server. The set of variables is not much different when using the Apache web server.

The `request.env.http_*` variables are parsed from the request HTTP header.

The `request.env.web2py_*` variables. These are not parsed from the web server environment, but are created by WEB2PY in case your applications need to know about the WEB2PY location and version, and whether it is running on the Google App Engine (because specific optimizations may be necessary).

Also notice the `request.env.wsgi_*` variables. They are specific to the wsgi adaptor.

4.7 response

`response` is another instance of the `Storage` class. It contains the following:

- **response.author**: optional parameter that may be included in the views. It should contain the name of the author of the page being displayed and should be rendered by the HTML meta tag.
- **response.body**: a `StringIO` object into which WEB2PY writes the output page body. NEVER CHANGE THIS VARIABLE.
- **response.cookies**: similar to **request.cookies**, but while the latter contains the cookies sent from the client to the server, the former contains cookies sent by the server to the client. The session cookie is handled automatically.

variable	value
request.application	examples
request.controller	default
request.function	index
request.extension	html
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://web2py.com/
request.env.http_user_agent	Mozilla/5.0
request.env.http_via	1.1 web2py.com
request.env.http_x_forwarded_for	76.224.34.5
request.env.http_x_forwarded_host	web2py.com
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.web2py_path	/Users/mdipierro/web2py
request.env.we2bpy_version	Version 1.65.1 (2009-07-05 10:19:29)
request.env.web2py_runtime_gae	(optional, defined only if GAE detected)
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

Figure 4.1 Example of system variables stored in `request`

- **response.description**: optional parameter that may be included in the views, normally used to set the meta description in the HTML header. It should be rendered by the corresponding meta tag.
- **response.download(request, db)**: a method used to implement the controller function that allows downloading of uploaded files.
- **response.flash**: optional parameter that may be included in the views. Normally used to notify the user about something that happened.
- **response.headers**: a `dict` for HTTP response headers.
- **response.keywords**: optional parameter that may be included in the views. It should be rendered by the corresponding HTML meta tag.
- **response.menu**: optional parameter that may be included in the views, normally used to pass a navigation menu tree to the view. It can be rendered by the MENU helper.
- **response.postprocessing**: this is a list of functions, empty by default. These functions are used to filter the response object at the output of an action, before the output is rendered by the view. It can be used to implement support for other template languages.
- **response.render(view, vars)**: a method used to call the view explicitly inside the controller. `view` is an optional parameter which is the name of the view file, `vars` is a dictionary of named values passed to the view.
- **response.session_file**: file stream containing the session.
- **response.session_file_name**: name of the file where the session will be saved.
- **response.session_id**: the id of the current session. It is determined automatically. NEVER CHANGE THIS VARIABLE.
- **response.session_id_name**: the name of the session cookie for this application. NEVER CHANGE THIS VARIABLE.
- **response.status**: the HTTP status code integer to be passed to the response. Default is 200 (OK).
- **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.subtitle**: optional parameter that may be included in the views. It should contain the subtitle of the page.

- **response.title**: optional parameter that may be included in the views. It should contain the title of the page and should be rendered by the HTML title TAG in the header.
- **response._vars**: this variable is accessible only in a view, not in the action. It contains the value returned by the action to the view.
- **response.view**: the name of the view template that must render the page. This is set by default to:

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

  or, if the above file cannot be located, to

```
"generic.%s" % (request.extension)
```

  Change the value of this variable to modify the view file associated with a particular action.
- **response.xmlrpc(request, methods)**: when a controller returns it, this function exposes the methods via XML-RPC [44]. This function is deprecated since a better mechanism is available and described in Chapter 9.
- **response.write(text)**: a method to write text into the output page body.

Since **response** is a `gluon.storage.Storage` object it can be used to store other attributes that you may want to pass to the view. While there is no technical restriction our recommendation is to store only variables that are to be rendered by all pages in the overall layout ("layout.html").

Anyway, we strongly suggest to stick to the variables listed here:

```
response.title
response.subtitle
response.author
response.keywords
response.description
response.flash
response.menu
```

because this will make it easier to replace the standard "layout.html" file that comes with WEB2PY with another layout file, one that uses the same set of variables.

4.8 session

`session` is another instance of the `Storage` class. Whatever is stored into `session` for example:

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

can be retrieved at a later time:

```
a = session.myvariable
```

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

The session object has two important methods. One is **forget**:

```
session.forget()
```

It tells WEB2PY not to save the session. This should be used in those controllers whose actions are called often and do not need to track user activity.

The other method is **connect**:

```
session.connect(request, response, db, masterapp=None)
```

where `db` is the name of an open database connection (as returned by the DAL). It tells WEB2PY that you want to store the sessions in the database and not on the filesystem. WEB2PY creates a table:

```
db.define_table('web2py_session',
                Field('locked', 'boolean', default=False),
                Field('client_ip'),
                Field('created_datetime', 'datetime', default=now),
                Field('modified_datetime', 'datetime'),
                Field('unique_key'),
                Field('session_data', 'text'))
```

and stores cPickled sessions in the `session_data` field.

The option `masterapp=None`, by default, tells WEB2PY to try to retrieve an existing session for the application with name in `request.application`, in the running application.

If you want two or more applications to share sessions, set `masterapp` to the name of the master application.

You can check the state of your application at any time by printing the `request`, `session` and `response` system variables. One way to do it is to create a dedicated action:

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

4.9 cache

`cache` a global object also available in the WEB2PY execution environment. It has two attributes:

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

`cache` is callable, this allows it to be used as a decorator for caching actions and views.

The following example caches the `time.ctime()` function in RAM:

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

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

The following example caches the `time.ctime()` function on disk:

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

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

The next example caches the `time.ctime()` function to both RAM and disk:

```
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 me', _href=request.url))
```

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 in RAM or on disk, `lambda: time.ctime()` is executed and the cache is updated. This technique is useful in a multiprocess environment. The two times do not have to be the same.

The following example is caching in RAM the output of the controller function (but not the view):

```
@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 me', _href=request.url))
```

The dictionary returned by `cache_controller_in_ram` is cached in RAM for 5 seconds. Note that the result of a database select cannot be cached without

first being serialized. A better way is to cache the database directly using the `select` method `cache` argument.

The following example is caching the output of the controller function on disk (but not the view):

```
@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=request.url))
```

The dictionary returned by `cache_controller_on_disk` is cached on disk for 5 seconds. Remember that WEB2PY cannot cache a dictionary that contains unpickleable objects.

It is also possible to cache the view. The trick is to render the view in the controller function, so that the controller returns a string. This is done by returning `response.render(d)` where `d` is the dictionary we intended to pass to the view:

The following example caches the output of the controller function in RAM (including the rendered view):

```
@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=request.url))
    return response.render(d)
```

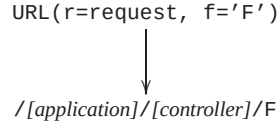
`response.render(d)` returns the rendered view as a string which is now cached for 5 seconds. This is the best and fastest way of caching.

It is also possible to define other caching mechanisms such as memcache. Memcache is available via `gluon.contrib.memcache` and is discussed in more details in Chapter 11.

4.10 URL

The `URL` function is one of the most important functions in WEB2PY. It generates internal URL paths for the actions and the static files.

Here is an example:



Notice that the output of the URL function depends on the name of the current application, the calling controller and other parameters. WEB2PY supports URL mapping and reverse URL mapping. URL mapping allows to redefine the format of external URLs. If you use the URL function to generate all the internal URLs, then additions or changes to URL mappings will prevent broken links within the WEB2PY application.

You can pass additional parameters to the URL function, i.e., extra terms in the URL path (args) and URL query variables (vars):

```
URL(r=request, f='F', args=['x', 'y'], vars=dict(z='t'))
```


/[application]/[controller]/F/x/y?z=t

The args attributes are automatically parsed, decoded, and finally stored in request.args by WEB2PY. Similarly, the vars are parsed, decoded, and then stored in request.vars.

args and vars provide the basic mechanism by which WEB2PY exchanges information with the client's browser.

If args contains only one element, there is no need to pass it in a list.

You can also use the URL function to generate URLs to actions in other controllers and other applications:

```
URL('a', 'c', 'f', args=['x', 'y'], vars=dict(z='t'))
```



```
/a/c/f/x/y?z=t
```

For the reasons mentioned above, you should always use the URL function to generate URLs of static files for your applications. Static files are stored in the application's static subfolder (that's where they go when uploaded using the administrative interface). WEB2PY provides a virtual 'static' controller whose job is to retrieve files from the static subfolder, determine their content-type, and stream the file to the client. The following example generates the URL for the static file "image.png":

```
URL(r=request, c='static', f='image.png')
```


/[application]/static/image.png

You do not need to encode/escape the args *and* vars *arguments; this is done automatically for you.*

4.11 HTTP and redirect

WEB2PY defines only one new exception called HTTP. This exception can be raised anywhere in a model, a controller, or a view with the command:

```
raise HTTP(400, "my message")
```

It causes the control flow to jump away from the user's code, back to WEB2PY, and return an HTTP response like:

```
HTTP/1.1 400 BAD REQUEST
Date: Sat, 05 Jul 2008 19:36:22 GMT
Server: CherryPy/3.1.0beta3 WSGI Server
Content-Type: text/html
Via: 1.1 127.0.0.1:8000
Connection: close
Transfer-Encoding: chunked

my message
```

The first argument of HTTP is the HTTP status code. The second argument is the string that will be returned as the body of the response. Additional optional named arguments are used to build the response HTTP header. For example:

```
raise HTTP(400, 'my message', test='hello')
```

generates:

```
HTTP/1.1 400 BAD REQUEST
Date: Sat, 05 Jul 2008 19:36:22 GMT
Server: CherryPy/3.1.0beta3 WSGI Server
Content-Type: text/html
Via: 1.1 127.0.0.1:8000
Connection: close
Transfer-Encoding: chunked
test: hello

my message
```

If you do not want to commit the open database transaction, rollback before raising the exception.

Any exception other than HTTP causes WEB2PY to roll back 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 ticketed by WEB2PY.

The command:

```
redirect('http://www.web2py.com')
```

is simply a shortcut for:

```
raise HTTP(303,
           'You are being redirected <a href="%s">here</a>' %
               location,
           Location='http://www.web2py.com')
```

The named arguments of the `HTTP` initializer method are translated into HTTP header directives, in this case, the redirection target location. `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 temporary redirect or to 301 for a permanent redirect.

4.12 T and Internationalization

The object `T` is the language translator. It constitutes a single global instance of the WEB2PY class `gluon.language.translator`. All string constants (and only string constants) should be marked by `T`, for example:

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

Strings that are marked with `T` are identified by WEB2PY as needing language translation and they will be translated when the code (in the model, controller, or view) is executed. If the string to be translated is not a constant but a variable, it will be added to the translation file at runtime (except on GAE) to be translated later.

The `T` object can also contain interpolated variables, for example:

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

The first string is translated according to the requested language file and the `name` variable is replaced independently of the language.

Concatenating translation strings is not a good idea; this is why WEB2PY does not allow you to do:

```
T("blah ") + name + T(" blah")   # invalid!
```

but it does allow:

```
T("blah %(name)s blah", 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, for example:

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

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

Normally, string translation is evaluated lazily when the view is rendered; hence, the translator `force` method should not be called inside a view.

It is possible to disable lazy evaluation via

```
T.lazy = False
```

In this way, strings are translated immediately by the T operator based on the currently accepted or forced language.

A common issue is the following. The original application is in English. Suppose that there is a translation file (for example Italian, "it-it.py") and the HTTP client declares that it accepts both English (en) and Italian (it-it) in that order. The following unwanted situation occurs: WEB2PY does not know the default is written in English (en). Therefore, it prefers translating everything into Italian (it-it) because it only found the Italian translation file. If it had not found the "it-it.py" file, it would have used the default language strings (English).

There are two solutions for this problem: create a translation language for English, which would be redundant and unnecessary, or better, tell WEB2PY which languages should use the default language strings (the strings coded into the application). This can be done with:

```
T.current_languages = ['en', 'en-en']
```

`T.current_languages` is a list of languages that do not require translation.

Notice that 'it' and 'it-it' are different languages from the point of view of WEB2PY. To support both of them, one would need two translation files, always lower case. The same is true for all other languages.

The currently accepted language is stored in

```
T.accepted_language
```

4.13 Cookies

WEB2PY uses the Python cookies modules for handling cookies. Cookies from the browser are in `request.cookies` and cookies sent by the server are in `response.cookies`. You can set a cookie as follows:

```
response.cookies['mycookie'] = 'somevalue'
response.cookies['mycookie']['expires'] = 24 * 3600
response.cookies['mycookie']['path'] = '/'
```

The second line tells the browser to keep the cookie for 24 hours. The third line tells the browser to send the cookie back to any application (URL path) at the current domain.

The cookie can be made secure with:

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

A secure cookie is only sent back over HTTPS and not over HTTP.

The cookie can be retrieved with:

```
if request.cookies.has_key('mycookie'):
    value = request.cookies['mycookie'].value
```

Unless sessions are disabled, WEB2PY, under the hood, sets the following cookie and uses it to handle sessions:

```
response.cookies[response.session_id_name] = response.session_id
response.cookies[response.session_id_name]['path'] = "/"
```

4.14 init Application

When you deploy WEB2PY, you will want to set a default application, i.e., the application that starts when there is an empty path in the URL, as in:

```
http://127.0.0.1:8000
```

By default, when confronted with an empty path, WEB2PY looks for an application called **init**. If there is no init application it looks for an application called **welcome**.

Here are three ways to set the default application:

- Call your default application "init".
- Make a symbolic link from "applications/init" to your application's folder.
- Use URL rewrite as discussed in the next section.

4.15 URL Rewrite

WEB2PY has the ability to rewrite the URL path of incoming requests prior to calling the controller action (URL mapping), and conversely, WEB2PY can rewrite the URL path generated by the `URL` function (reverse URL mapping). One reason to do this is for handling legacy URLs, another is to simplify paths and make them shorter.

To use this feature, create a new file in the "web2py" folder called "routes.py" and define two lists (or tuples) of 2-tuples `routes_in` and `routes_out`. Each tuple contains two elements: the pattern to be replaced and the string that replaces it. For example:

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

With these routes, the URL:

```
http://127.0.0.1:8000/testme
```

is mapped into:

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

To the visitor, all links to the page URL looks like `/testme`.

The patterns have the same syntax as Python regular expressions. For example:

```
  ('.*\.php', '/init/default/index'),
```

maps all URLs ending into ".php" to the index page.

Sometimes you want to get rid of the application prefix from the URLs because you plan to expose only one application. This can be achieved with:

```
routes_in = (
  ('/(?P<any>.*)', '/init/\g<any>'),
)
routes_out = (
  ('/init/(?P<any>.*)', '/\g<any>'),
)
```

There is also an alternative syntax that can be mixed with the regular expression notation above. It consists of using `$name` instead of `(?P<name>[\w_]+)` or `\g<name>`. For example:

```
routes_in = (
  ('/$c/$f', '/init/$c/$f'),
)

routes_out = (
  ('/init/$c/$f', '/$c/$f'),
)
```

would also eliminate the "/example" application prefix in all URLs.

Using the `$` notation, you can automatically map `routes_in` to `routes_out`, provided you don't use any regular expressions. For example:

```
routes_in = (
  ('/$c/$f', '/init/$c/$f'),
)

routes_out = [(x, y) for (y, x) in routes_in]
```

If there are multiple routes, the first to match the URL is executed. If no pattern matches, the path is left unchanged.

Here is a minimal "routes.py" for handling favicon and robots requests:

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

The general syntax for routes is more complex than the simple examples we have seen so far. Here is a more general and representative example:

```
routes_in = (
 ('140\.191\.\d+\.\d+:https://www.web2py.com:POST /(?P<any>.*)\.php',
  '/test/default/index?vars=\g<any>'),
)
```

It maps `https` `POST` requests to host `www.web2py.com` from a remote IP matching the regular expression

```
140\.191\.\d+\.\d+
```

requesting a page matching the regular expression

```
/(?P<any>.*)\.php!
```

into

```
/test/default/index?vars=\g<any>
```

where `\g<any>` is replaced by the matching regular expression.

The general syntax is

```
[remote address]:[protocol]://[host]:[method] [path]
```

The entire expression is matched as a regular expression, so "." should always be escaped and any matching subexpression can be captured using "(?P<...>...)" according to Python regex syntax.

This allows to reroute requests based on the client IP address or domain, based on the type of the request, on the method, and the path. It also allows to map different virtual hosts into different applications. Any matched subexpression can be used to built the target URL and, eventually, passed as a GET variable.

All major web servers, such as Apache and lighttpd, also have the ability to rewrite URLs. In a production environment we suggest having the web server perform URL rewriting.

4.16 Routes on Error

You can also use "routes.py" to redirect the visitor to special actions in case there is an error on server. You can specify this mapping globally, for each app, for each error code, for each app and error code. Here is an example:

```
routes_onerror = [
  ('init/400', '/init/default/login'),
  ('init/*', '/init/static/fail.html'),
  ('*/404', '/init/static/cantfind.html'),
  ('*/*', '/init/error/index')
]
```

For each tuple the first string is matched against "[appname]/[error code]". If a match is found the user is redirected to the URL in the second string of the matching tuple. In case a ticket was issued, the ticket is passed to the new URL as a GET variable called ticket.

Unmatched errors display a default error page. This default error page can also be customized here:

```
error_message = '<html><body><h1>Invalid request</h1></body></html>'
error_message_ticket = '<html><body><h1>Internal error</h1>Ticket
    issued: <a href="/admin/default/ticket/%(ticket)s" target="_blank
    ">%(ticket)s</a></body></html>'
```

The first variable contains the error message when an invalid application is requested. The second variable contains the error message when a ticket is issued.

4.17 Cron

The WEB2PY cron provides the ability for applications to execute tasks at preset times, in a platform independent manner.

For each application, cron functionality is defined by a crontab file "app/cron/crontab", following the syntax defined here (with some extensions that was WEB2PY specific):

```
http://en.wikipedia.org/wiki/Cron#crontab_syntax
```

This means that every application can have a separate cron configuration and that cron config can be changed from within WEB2PY without affecting the host OS itself.

Here is an example:

```
0-59/1  *  *  *  *  root python /path/to/python/script.py
30      3  *  *  *  root *applications/admin/cron/db_vacuum.py
*/30    *  *  *  *  root **applications/admin/cron/something.py
@reboot root    *mycontroller/myfunction
@hourly root    *applications/admin/cron/expire_sessions.py
```

The last two lines in this example, use extensions to regular cron syntax to provide additional WEB2PY functionality.

Web2py cron has a some extra syntax to support WEB2PY application specifics.

If the task/script is prefixed with an asterisk (*) and ends with ".py", it will be executed in the WEB2PY environment. This means you will have all the controllers and models at your disposal. If you use two asterisks (**), the MODELs will not be executed. This is the recommended way of calling as it has less overhead and avoids potential locking problems.

Notice that scripts/functions executed in the WEB2PY environment require a manual `db.commit()` at the end of the function or the transaction will be reverted.

WEB2PY does not generate tickets or meaningful tracebacks in shell mode (in which cron is run). Make sure that your WEB2PY code runs without errors before you set it up as a cron task, as you will likely not be able to see them when run from cron.

Moreover, be careful how you use models. While the execution happens in a separate process, database locks have to be taken into account in order to avoid pages waiting for cron tasks that be blocking the database. Use the ** syntax if you don't need to use the database in your cron task.

You can also call a controller function. There is no need to specify a path. The controller and function will be that of the invoking application. Take special care about the caveats listed above. Example:

```
1 */30  *  *  *  *  root *mycontroller/myfunction
```

If you specify @reboot in the first field in the crontab file, the given task will be executed only once, on WEB2PY startup. You can use this feature if you want to precache, check or initialize data for an application on WEB2PY startup. Note that cron tasks are executed in parallel with the application — if the application is not ready to serve requests until the cron task is finished, you should implement checks to reflect this. Example:

```
1 @reboot  *  *  *  *  root *mycontroller/myfunction
```

Depending on how you are invoking WEB2PY, there are four modes of operation for WEB2PY cron.

- Soft cron: available under all execution modes
- Hard cron: available if using the built-in web server (either directly or via Apache mod_proxy)
- External cron: available if you have access to the system's own cron service
- No cron

The default is hard cron if you are using the built-in web server; in all other cases the default is soft cron.

Soft cron is the default if you are using CGI, FASTCGI or WSGI. Your tasks will be executed in the first call (page load) to WEB2PY after the time specified in crontab (but after processing the page, so no delay to the user is visible). Obviously, there is some uncertainty exactly when the task will be executed depending on the traffic the site receives. Also, the cron task may get interrupted if the web server has a page load timeout set. If these limitations are not acceptable, see "external cron". Soft cron is a reasonable last resort, but if your web server allows other cron methods, they should be preferred over soft cron.

Hard cron is the default if you are using the built-in web server (either directly or via Apache mod_proxy). Hard cron is executed in a parallel thread, so unlike soft cron there are no limitations with regard to run time or execution time precision.

External cron is not default in any scenario, but requires you to have access to the system cron facilities. It runs in a parallel process, so none of the limitations of soft cron apply. This is the recommended way of using cron under WSGI or FASTCGI.

Example of line to add to the system crontab, (usually /etc/crontab):

```
0-59/1 * * * * web2py cd /var/www/web2py/ && python web2py.py -C -D 1
    >> /tmp/cron.output 2>&1
```

If you are running external cron, make sure you add the -N command line parameter to your WEB2PY startup script or config so there is no collision of multiple types of cron.

In case you do not need any cron functionality within a particular process, you can use the -N command line parameter to disable it. Note that this might disable some maintenance tasks (like the automatic cleaning of session dirs). The most common use of this function:

- You already have set up external cron triggered from the system (most common with WSGI setups)
- If you want to debug your application without cron interfering either with actions or with output

4.18 Import Other Modules

WEB2PY is written in Python, so it can import and use any Python module, including third party modules. It just needs to be able to find them.

Modules can be installed in the official Python "site-packages" directory or anywhere your application can find them. Modules in "site-packages" directory are, as the name suggests, site-level packages. Applications requiring site-packages are not portable unless these modules are installed separately. The advantage of having modules in "site-packages" is that multiple applications can share them. Let's consider, for example, the plotting package called "matplotlib". You can install it from the shell using the PEAK `easy_install` command:

```
easy_install py-matplotlib
```

and then you can import it into any model/controller/view with:

```
import matplotlib
```

You can also install packages manually in the application "modules" folder. The advantage is that the module will be automatically copied and distributed with the application. If the application is called "test", you can import "mymodule" with:

```
import applications.test.modules.mymodule as mymodule
```

Since the application "test" may be renamed, we suggest the following two approaches:

```
exec('import applications.%s.modules.mymodule as mymodule' % \
     request.application)
```

or:

```
import sys, os
path = os.path.join(request.folder, 'modules')
if not path in sys.path:
    sys.path.append(path)
import mymodule
```

The first approach using `exec` is slower than the second, but it avoids conflicts. The second approach is faster but it may import the wrong modules if different applications contain modules with the same name.

4.19 Execution Environment

WEB2PY model and controller files are not Python modules in that they cannot be imported using the Python `import` statement. The reason for this is that

models and controllers are designed to be executed in a prepared environment that has been prepopulated with WEB2PY global objects (request, response, session, cache and T) and helper functions. This is necessary because Python is a statically (lexically) scoped language, whereas the WEB2PY environment is created dynamically.

WEB2PY provides the `exec_environment` function to allow you to access models and controllers directly. `exec_environment` creates a WEB2PY execution environment, loads the file into it and then returns a Storage object containing the environment. The Storage object also serves as a namespacing mechanism. Any Python file designed to be executed in the execution environment can be loaded using `exec_environment`. Uses for `exec_environment` include:

- Accessing data (models) from other applications.
- Accessing global objects from other models or controllers.
- Executing controller functions from other controllers.
- Loading site-wide helper libraries.

This example reads rows from the `user` table in the `cas` application:

```
1 from gluon.shell import exec_environment
2 cas = exec_environment('applications/cas/models/db.py')
3 rows = cas.db().select(cas.db.user.ALL)
```

Another example: suppose you have a controller "other.py" that contains:

```
1 def some_action():
2     return dict(remote_addr=request.env.remote_addr)
```

Here is how you can call this action from another controller (or from the WEB2PY shell):

```
1 from gluon.shell import exec_environment
2 other = exec_environment('applications/app/controllers/other.py',
      request=request)
3 result = other.some_action()
```

In line 2, `request=request` is optional. It has the effect of passing the current request to the environment of "other". Without this argument, the environment would contain a new and empty (apart from `request.folder`) request object. It is also possible to pass a response and a session object to `exec_environment`. Be careful when passing request, response and session objects — modification by the called action or coding dependencies in the called action could lead to unexpected side effects.

The function call in line 3 does not execute the view; it simply returns the dictionary unless `response.render` is called explicitly by "some_action".

One final caution: don't use `exec_environment` inappropriately. If you want the results of actions in another application, you probably should implement an XML-RPC API (implementing an XML-RPC API with WEB2PY is almost trivial). Don't use `exec_environment` as a redirection mechanism; use the `redirect` helper.

4.20 Cooperation

There are many ways applications can cooperate:

- Applications can connect to the same database and thus share tables. It is not necessary that all tables in the database are defined by all applications, but they must be defined by those applications that use them. All applications that use the same table, bar one, must define the table with `migrate=False`.
- Applications can share sessions with the command:

```
session.connect(request, response, masterapp='appname', db=db)
```

  Here "appname" is the name of the master application, the one that sets the initial session_id in the cookie. `db` is a database connection to the database that contains the session table (`web2py_session`). All apps that share sessions must use the same database for session storage.

- Applications can call each other's actions remotely via XML-RPC.
- Applications can access each other's files via the filesystem (assuming they share the same filesystem).
- Applications can call each other's actions locally using `exec_environment` as discussed above.
- Applications can import each other's modules using the syntax:

```
import applications.otherapp.modules.othermodule as mymodule.
```

- Applications can import any module in the `PYTHONPATH` search path, `sys.path`.

  If a module function needs access to one of the core objects (request, response, session, cache, and T), the objects must be passed explicitly to the function. Do not let the module create another instance of the core objects. Otherwise, the function will not behave as expected.

CHAPTER 5

THE VIEWS

WEB2PY uses Python for its models, controllers, and views, although it uses a slightly modified Python syntax in the views to allow more readable code without imposing any restrictions on proper Python usage.

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

- How should embedded code be escaped?
- Should indenting be 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. This allows the developer to use those editors to create WEB2PY views.

Since the developer is embedding Python code into HTML, the document should be indented according to HTML rules, and not Python rules. Therefore, we allow unindented Python inside the {{...}} tags. Since Python normally uses indentation to delimit blocks of code, we need a different way

to delimit them; this is why the WEB2PY template language makes use of the Python keyword `pass`.

> *A code block starts with a line ending with a colon and ends with a line beginning with* `pass`*. The keyword* `pass` *is not necessary when the end of the block is obvious from the context.*

Here is an example:

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

Note that `pass` is a Python keyword, not a WEB2PY keyword. Some Python editors, such as Emacs, use the keyword `pass` to signify the division of blocks and use it to re-indent code automatically.

The WEB2PY template language does exactly the same. When it finds something like:

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

it translates it into a program:

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

`response.write` writes to the `response.body`.

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 the actual code that is executed (which is something that can be debugged with an HTML editor or the DOM inspector of the browser).

Also note that:

```
{{=x}}
```

generates

```
response.write(x)
```

Variables injected into the HTML in this way are escaped by default. The escaping is ignored if `x` is an `XML` object, even if escape is set to `True`.

Here is an example that introduces the `H1` helper:

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

which is translated to:

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

upon evaluation, the `H1` object and its components are recursively serialized, escaped and written to the response body. The tags generated by `H1` and inner HTML are not escaped. This mechanism guarantees that all text — and only text — displayed on the web page is always escaped, thus preventing XSS vulnerabilities. At the same time, the code is simple and easy to debug.

The method `response.write(obj, escape=True)` takes two arguments, the object to be written and whether it has to be escaped (set to `True` by default). If `obj` has an `.xml()` method, it is called and the result written to the response body (the `escape` argument is ignored). Otherwise it uses the object's `__str__` method to serialize it and, if the escape argument is `True`, escapes it. All built-in helper objects (`H1` in the example) are objects that know how to serialize themselves via the `.xml()` method.

This is all done transparently. You never need to (and never should) call the `response.write` method explicitly.

5.1 Basic Syntax

The WEB2PY template language supports all Python control structures. Here we provide some examples of each of them. They can be nested according to usual programming practice.

for...in

In templates you can loop over any iterable object:

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

which produces:

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

Here `item` is any iterable object such as a Python list, Python tuple, or Rows object, or any object that is implemented as an iterator. The elements displayed are first serialized and escaped.

while

You can create a loop using the while keyword:

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

which produces:

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

if...elif...else

You can use conditional clauses:

```
{{
import random
k = random.randint(0, 100)
}}
<h2>
{{=k}}
{{if k % 2:}}is odd{{else:}}is even{{pass}}
</h2>
```

which produces:

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

Since it is obvious that `else` closes the first `if` block, there is no need for a `pass` statement, and using one would be incorrect. However, you must explicitly close the `else` block with a `pass`.

Recall 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 divisible by 4
{{elif k % 2 == 0:}}is even
{{else:}}is odd
{{pass}}
</h2>
```

It produces:

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

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

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
{{else:}}
no division by zero
{{finally}}
<br />
{{pass}}
```

It will produce the following output:

```
Hello
division by zero
<br />
```

This example illustrates that all output generated before an exception occurs is rendered (including output that preceded the exception) inside the try block. "Hello" is written because it precedes the exception.

def...return

The WEB2PY template language allows the developer to define and implement functions that can return any Python object or a text/html string. Here we consider two examples:

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

produces the following output:

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

The function `itemize1` returns a helper object that is 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}}
<ul>
{{itemize2('www.google.com')}}
</ul>
```

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`, since the function does not return the text, but it writes it directly into the response.

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

5.2 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>
```

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

Positional arguments are interpreted as objects contained between the open and close tags. Named arguments that start with an underscore are interpreted as HTML tag attributes (without the underscore). Some helpers also have named arguments that do not start with underscore; these arguments are tag-specific.

The following set of helpers

5mm `A`, `B`, `BODY`, `BR`, `CENTER`, `DIV`, `EM`, `EMBED`, `FORM`, `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `HEAD`, `HR`, `HTML`, `IMG`, `INPUT`, `LABEL`, `LI`, `LINK`, `OL`, `UL`, `META`, `MENU`, `OBJECT`, `ON`, `OPTION`, `P`, `PRE`, `SCRIPT`, `SELECT`, `SPAN`, `STYLE`, `TABLE`, `THEAD`, `TBODY`, `TFOOT`, `TD`, `TEXTAREA`, `TH`, `TITLE`, `TR`, `TT`

5mm can be used to build complex expressions that can then be serialized to XML [47, 48]. For example:

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

is rendered:

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

The helpers mechanism in WEB2PY is more than a system to generate HTML without concatenating strings. It provides a server-side representation of the Document Object Model (DOM).

Components' objects can be referenced via their position, and helpers act as lists with respect to their components:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> print a
<div><span>ab</span>c</div>
>>> del a[1]
>>> a.append(B('x'))
>>> a[0][0] = 'y'
>>> print a
<div><span>yb</span><b>x</b></div>
```

Attributes of helpers can be referenced by name, and helpers act as dictionaries with respect to their attributes:

```
>>> a = DIV(SPAN('a', 'b'), 'c')
>>> a['_class'] = 's'
>>> a[0]['_class'] = 't'
>>> print a
<div class="s"><span class="t">ab</span>c</div>
```

Helpers can be located and updated:

```
>>> a = DIV(DIV(DIV('a', _id='target')))
>>> a.element(_id='target')[0] = 'changed'
>>> print a
<div><div><div>changed</div></div></div>
```

Any attribute can be used to locate an element (not just `_id`), including multiple attributes (the function element can take multiple named arguments) but only the first matching element will be returned.

XML

`XML` is an object used to encapsulate text that should not be escaped. The text may or may not contain valid XML. For example, it could contain JavaScript.

The text in this example is escaped:

```
>>> print DIV("<b>hello</b>")
&lt;b&gt;hello&lt;/b&gt;
```

by using `XML` you can prevent escaping:

```
>>> print DIV(XML("<b>hello</b>"))
<b>hello</b>
```

Sometimes you want to render HTML stored in a variable, but the HTML may contain unsafe tags such as scripts:

```
>>> print XML('<script>alert("unsafe!")</script>')
<script>alert("unsafe!")</script>
```

Unescaped executable input such as this (for example, entered in the body of a comment in a blog) is unsafe, because it can be used to generate Cross Site Scripting (XSS) attacks against other visitors to the page.

The WEB2PY XML helper can sanitize our text to prevent injections and escape all tags except those that you explicitly allow. Here is an example:

```
>>> print XML('<script>alert("unsafe!")</script>', sanitize=True)
&lt;script&gt;alert(&quot;unsafe!&quot;)&lt;/script&gt;
```

The XML constructors, by default, consider the content of some tags and some of their attributes safe. You can override the defaults using the optional `permitted_tags` and `allowed_attributes` arguments. Here are the default values of the optional arguments of the XML helper.

```
XML(text, sanitize=False,
    permitted_tags=['a', 'b', 'blockquote', 'br/', 'i', 'li',
       'ol', 'ul', 'p', 'cite', 'code', 'pre', 'img/'],
    allowed_attributes={'a':['href', 'title'],
       'img':['src', 'alt'], 'blockquote':['type']})
```

Built-in Helpers

A This helper is used to build links.

```
>>> print A('<click>', XML('<b>me</b>'),
            _href='http://www.web2py.com')
<a href='http://www.web2py.com'>&lt;click&gt;<b>me/b></a>
```

B This helper makes its contents bold.

```
>>> print B('<hello>', XML('<i>world</i>'), _class='test', _id=0)
<b id="0" class="test">&lt;hello&gt;<i>world</i></b>
```

BODY This helper makes the body of a page.

```
>>> print BODY('<hello>', XML('<b>world</b>'), _bgcolor='red')
<body bgcolor="red">&lt;hello&gt;<b>world</b></body>
```

CENTER This helper centers its content.

```
>>> print CENTER('<hello>', XML('<b>world</b>'),
>>>               _class='test', _id=0)
<center id="0" class="test">&lt;hello&gt;<b>world</b></center>
```

CODE This helper performs syntax highlighting for Python, C, C++, HTML and WEB2PY code, and is preferable to PRE for code listings. CODE also has the ability to create links to the WEB2PY API documentation.

Here is an example of highlighting sections of Python code.

```
>>> print CODE('print "hello"', language='python').xml()
<table><tr valign="top"><td style="width:40px; text-align: right;"><
    pre style="
        font-size: 11px;
        font-family: Bitstream Vera Sans Mono,monospace;
        background-color: transparent;
            margin: 0;
            padding: 5px;
            border: none;
        background-color: #E0E0E0;
        color: #A0A0A0;
    ">1.</pre></td><td><pre style="
        font-size: 11px;
        font-family: Bitstream Vera Sans Mono,monospace;
        background-color: transparent;
            margin: 0;
            padding: 5px;
            border: none;
            overflow: auto;
    "><span style="color:#185369; font-weight: bold">print </span><
        span style="color: #FF9966">"hello"</span></pre></td></tr></
        table>
```

Here is a similar example for HTML

```
>>> print CODE(
>>>   '<html><body>{{=request.env.remote_add}}</body></html>',
>>>   language='html')
<table><tr valign="top"><td style="width:40px; text-align: right;"><
    pre style="
....
    "><span style="font-weight: bold">&lt;</span>html<span style="
        font-weight: bold">&gt;&lt;</span>body<span style="font-
        weight: bold">&gt;{{=</span><span style="text-decoration:None
        ;color:#FF5C1F;">request</span><span style="font-weight: bold
        ">.</span>env<span style="font-weight: bold">.</span>
        remote_add<span style="font-weight: bold">}}&lt;/</span>body<
        span style="font-weight: bold">&gt;&lt;/</span>html<span
        style="font-weight: bold">&gt;</span></pre></td></tr></table>
```

These are the default arguments for the CODE helper:

```
CODE("print 'hello world'", language='python', link=None, counter=1,
    styles={})
```

Supported values for the `language` argument are "python", "html_plain", "c", "cpp", "web2py", and "html". The "html" language interprets {{ and }} tags as "web2py" code, while "html_plain" doesn't.

If a `link` value is specified, for example "/examples/global/vars/", WEB2PY API references in the code are linked to documentation at the link URL. For

example "request" would be linked to "/examples/global/vars/request". In the above example, the link URL is handled by the "var" action in the "global.py" controller that is distributed as part of the WEB2PY "examples" application.

The `counter` argument is used for line numbering. It can be set to any of three different values. It can be `None` for no line numbers, a numerical value specifying the start number, or a string. If the counter is set to a string, it is interpreted as a prompt, and there are no line numbers.

DIV All helpers apart from `XML` are derived from `DIV` and inherit its basic methods.

```
>>> print DIV('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<div id="0" class="test">&lt;hello&gt;<b>world</b></div>
```

EM Emphasizes its content.

```
>>> print EM('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<em id="0" class="test">&lt;hello&gt;<b>world</b></em>
```

FIELDSET This is used to create an input field together with its label.

```
>>> print FIELDSET('Height:', INPUT(_name='height'), _class='test')
<fieldset class="test">Height:<input name="height" /></fieldset>
```

FORM This is one of the most important helpers. In its simple form, it just makes a `<form>...</form>` tag, but because helpers are objects and have knowledge of what they contain, they can process submitted forms (for example, perform validation of the fields). This will be discussed in detail in Chapter 7.

```
>>> print FORM(INPUT(_type='submit'), _action='', _method='post')
<form enctype="multipart/form-data" action="" method="post">
<input type="submit" /></form>
```

The "enctype" is "multipart/form-data" by default.

The constructor of a `FORM`, and of `SQLFORM`, can also take a special argument called `hidden`. When a dictionary is passed as `hidden`, its items are translated into "hidden" INPUT fields. For example:

```
>>> print FORM(hidden=dict(a='b'))
<form enctype="multipart/form-data" action="" method="post">
<input value="b" type="hidden" name="a" /></form>
```

H1, H2, H3, H4, H5, H6 These helpers are for paragraph headings and subheadings:

```
>>> print H1('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<h1 id="0" class="test">&lt;hello&gt;<b>world</b></h1>
```

HEAD For tagging the HEAD of an HTML page.

```
>>> print HEAD(TITLE('<hello>', XML('<b>world</b>')))
<head><title>&lt;hello&gt;<b>world</b></title></head>
```

HTML This helper is a little different. In addition to making the `<html>` tags, it prepends the tag with a doctype string [49, 50, 51].

```
>>> print HTML(BODY('<hello>', XML('<b>world</b>')))
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http
    ://www.w3.org/TR/html4/loose.dtd">
<html><body>&lt;hello&gt;<b>world</b></body></html>
```

The HTML helper also takes some additional optional arguments that have the following default:

```
HTML(..., lang='en', doctype='transitional')
```

where doctype can be 'strict', 'transitional', 'frameset', 'html5', or a full doctype string.

XHTML XHTML is similar to HTML but it creates an XHTML doctype instead.

```
XHTML(..., lang='en', doctype='transitional', xmlns='http://www.w3.
    org/1999/xhtml')
```

where doctype can be 'strict', 'transitional', 'frameset', or a full doctype string.

INPUT Creates an `<input.../>` tag. An input tag may not contain other tags, and is closed by `/>` instead of `>`. The input tag has an optional attribute `_type` that can be set to "text" (the default), "submit", "checkbox", or "radio".

```
>>> print INPUT(_name='test', _value='a')
<input value="a" name="test" />
```

It also takes an optional special argument called "value", distinct from "_value". The latter sets the default value for the input field; the former sets its current value. For an input of type "text", the former overrides the latter:

```
>>> print INPUT(_name='test', _value='a', value='b')
<input value="b" name="test" />
```

For radio buttons `INPUT` selectively sets the "checked" attribute:

```
>>> for v in ['a', 'b', 'c']:
>>>     print INPUT(_type='radio', _name='test', _value=v, value='b')
    , v
<input value="a" type="radio" name="test" /> a
<input value="b" type="radio" checked="checked" name="test" /> b
<input value="c" type="radio" name="test" /> c
```

and similarly for checkboxes:

```
>>> print INPUT(_type='checkbox', _name='test', _value='a', value=
    True)
<input value="a" type="checkbox" checked="checked" name="test" />
>>> print INPUT(_type='checkbox', _name='test', _value='a', value=
    False)
<input value="a" type="checkbox" name="test" />
```

IFRAME This helper includes another web page in the current page. The url of the other page is specified via the "_src" attribute.

```
>>> print IFRAME(_src='http://www.web2py.com')
<iframe src="http://www.web2py.com"></iframe>
```

LABEL It is used to create a LABEL tag for an INPUT field.

```
>>> print LABEL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<label id="0" class="test">&lt;hello&gt;<b>world</b></label>
```

LI It makes a list item and should be contained in a UL or OL tag.

```
>>> print LI('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<li id="0" class="test">&lt;hello&gt;<b>world</b></li>
```

LEGEND It is used to create a legend tag for a field in a form.

```
>>> print LEGEND('Name', _for='somefield')
<legend for="somefield">Name</legend>
```

META To be used for building META tags in the HTML head. For example:

```
>>> print META(_name='security', _content='high')
<meta name="security" content="high" />
```

OBJECT Used to embed objects (for example, a flash player) in the HTML.

```
>>> print OBJECT('<hello>', XML('<b>world</b>'),
>>>               _src='http://www.web2py.com')
<object src="http://www.web2py.com">&lt;hello&gt;<b>world</b></object
    >
```

OL It stands for Ordered List. The list should contain LI tags. `OL` arguments that are not `LI` objects are automatically enclosed in `<li>...</li>` tags.

```
>>> print OL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<ol id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></
    ol>
```

ON This is here for backward compatibility and it is simply an alias for `True`. It is used exclusively for checkboxes and deprecated since `True` is more Pythonic.

```
>>> print INPUT(_type='checkbox', _name='test', _checked=ON)
<input checked="checked" type="checkbox" name="test" />
```

OPTION This should only be used as part of a SELECT/OPTION combination.

```
>>> print OPTION('<hello>', XML('<b>world</b>'), _value='a')
<option value="a">&lt;hello&gt;<b>world</b></option>
```

As in the case of INPUT, WEB2PY make a distinction between "_value" (the value of the OPTION), and "value" (the current value of the enclosing select). If they are equal, the option is "selected".

```
>>> print SELECT('a', 'b', value='b'):
<select>
<option value="a">a</option>
<option value="b" selected="selected">b</option>
</select>
```

P This is for tagging a paragraph.

```
>>> print P('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<p id="0" class="test">&lt;hello&gt;<b>world</b></p>
```

PRE Generates a `<pre>...</pre>` tag for displaying preformatted text. The CODE helper is generally preferable for code listings.

```
>>> print PRE('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<pre id="0" class="test">&lt;hello&gt;<b>world</b></pre>
```

SCRIPT This is include or link a script, such as JavaScript. The content between the tags is rendered as an HTML comment, for the benefit of really old browsers.

```
>>> print SCRIPT('alert("hello world");', _language='javascript')
<script language="javascript"><!--
alert("hello world");
//--></script>
```

SELECT Makes a `<select>...</select>` tag. This is used with the OPTION helper. Those SELECT arguments that are not OPTION objects are automatically converted to options.

```
>>> print SELECT('<hello>', XML('<b>world</b>'), _class='test', _id
    =0)
<select id="0" class="test"><option value="&lt;hello&gt;">&lt;hello&
    gt;</option><option value="&lt;b&gt;world&lt;/b&gt;"><b>world</b
    ></option></select>
```

SPAN Similar to DIV but used to tag inline (rather than block) content.

```
>>> print SPAN('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<span id="0" class="test">&lt;hello&gt;<b>world</b></span>
```

STYLE Similar to script, but used to either include or link CSS code. Here the CSS is included:

```
>>> print STYLE(XML('body {color: white}'))
<style><!--
body { color: white }
//--></style>
```

and here it is linked:

```
>>> print STYLE(_src='style.css')
<style src="style.css"><!--
//--></style>
```

TABLE, TR, TD These tags (along with the optional THEAD, TBODY and TFOOTER helpers) are used to build HTML tables.

```
>>> print TABLE(TR(TD('a'), TD('b')), TR(TD('c'), TD('d')))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>
```

TR expects TD content; arguments that are not TD objects are converted automatically.

```
>>> print TABLE(TR('a', 'b'), TR('c', 'd'))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>
```

It is easy to convert a Python array into an HTML table using Python's * function arguments notation, which maps list elements to positional function arguments.

Here, we will do it line by line:

```
>>> table = [['a', 'b'], ['c', 'd']]
>>> print TABLE(TR(*table[0]), TR(*table[1]))
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>
```

Here we do all lines at once:

```
>>> table = [['a', 'b'], ['c', 'd']]
>>> print TABLE(*[TR(*rows) for rows in table])
<table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>
```

TBODY This is used to tag rows contained in the table body, as opposed to header or footer rows. It is optional.

```
>>> print TBODY(TR('<hello>'), _class='test', _id=0)
<tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>
```

TEXTAREA This helper makes a `<textarea>...</textarea>` tag.

```
>>> print TEXTAREA('<hello>', XML('<b>world</b>'), _class='test')
<textarea class="test" cols="40" rows="10">&lt;hello&gt;<b>world</b
    ></textarea>
```

The only caveat is that its optional "value" overrides its content (inner HTML)

```
>>> print TEXTAREA(value="<hello world>", _class="test")
<textarea class="test" cols="40" rows="10">&lt;hello world&gt;</
    textarea>
```

TFOOT This is used to tag table footer rows.

```
>>> print TFOOT(TR(TD('<hello>')), _class='test', _id=0)
<tfoot id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tfoot>
```

TH This is used instead of TD in table headers.

```
>>> print TH('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<th id="0" class="test">&lt;hello&gt;<b>world</b></th>
```

THEAD This is used to tag table header rows.

```
>>> print THEAD(TR(TD('<hello>')), _class='test', _id=0)
<thead id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></thead>
```

TITLE This is used to tag the title of a page in an HTML header.

```
>>> print TITLE('<hello>', XML('<b>world</b>'))
<title>&lt;hello&gt;<b>world</b></title>
```

TR Tags a table row. It should be rendered inside a table and contain `<td>...</td>` tags. TR arguments that are not TD objects will be automatically converted.

```
>>> print TR('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tr id="0" class="test"><td>&lt;hello&gt;</td><td><b>world</b></td></
    tr>
```

TT Tags text as typewriter (monospaced) text.

```
>>> print TT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tt id="0" class="test">&lt;hello&gt;<b>world</b></tt>
```

UL Signifies an Unordered List and should contain LI items. If its content is not tagged as LI, UL does it automatically.

```
>>> print UL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<ul id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></
    ul>
```

Custom Helpers

Sometimes you need to generate custom XML tags. WEB2PY provides TAG, a universal tag generator.

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

generates the following XML

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

Arguments "a" and "b" and "d" are automatically escaped; use the XML helper to suppress this behavior. Using TAG you can generate HTML/XML tags not already provided by 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 an object, and `TAG.name` or `TAG['name']` is a function that returns a temporary helper class.

MENU The MENU helper takes a list of lists of the form of `response.menu` (as described in Chapter 4) and generates a tree-like structure using unordered lists representing the menu. For example:

```
>>> print MENU([['One', False, 'link1'], ['Two', False, 'link2']])
<ul class="web2py-menu web2py-menu-vertical"><li><a href="link1">One
    </a></li><li><a href="link2">Two</a></li></ul>
```

Each menu item can have a fourth argument that is a nested submenu (and so on recursively):

```
>>> print MENU([['One', False, 'link1', [['Two', False, 'link2']]]])
<ul class="web2py-menu web2py-menu-vertical"><li class="web2py-menu-
    expand"><a href="link1">One</a><ul class="web2py-menu-vertical"><
    li><a href="link2">Two</a></li></ul></li></ul>
```

The MENU helper takes the following optional arguments:

- `_class`: defaults to "web2py-menu web2py-menu-vertical" and sets the class of the outer UL elements.
- `ul_class`: defaults to "web2py-menu-vertical" and sets the class of the inner UL elements.
- `li_class`: defaults to "web2py-menu-expand" and sets the class of the inner LI elements.

The "base.css" of the scaffolding application understands the following basic types of menus: "web2py-menu web2py-menu-vertical" and "web2py-menu web2py-menu-horizontal".

5.3 BEAUTIFY

BEAUTIFY is used to build HTML representations of compound objects, including lists, tuples and dictionaries:

```
{{=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:

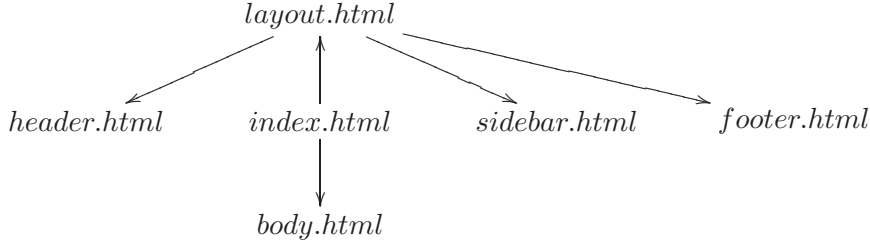
```
{"a":["hello", XML("world")], "b":(1, 2)}
```

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>
```

5.4 Page Layout

Views can extend and include other views in a tree-like structure, as in the following example (an upward arrow means extend, while a downward arrow means include):



In this example, the view "index.html" extends "layout.html" and includes "body.html". "layout.html" includes "header.html", "sidebar.html" and "footer.html".

The root of the tree is what we call a layout view. Just like any other HTML template file, you can edit it using the WEB2PY administrative interface. The file name "layout.html" is just a convention.

Here is a minimalist page that extends the "layout.html" view and includes the "page.html" view:

```
{{extend 'layout.html'}}
<h1>Hello World</h1>
{{include 'page.html'}}
```

The extended layout file must contain an `{{include}}` directive, something like:

```
<html><head><title>Page Title</title></head>
  <body>
    {{include}}
  </body>
</head>
```

When the view is called, the extended (layout) view is loaded, and the calling view replaces the `{{include}}` directive inside the layout. Processing continues recursively until all extend and include directives have been processed. The resulting template is then translated into Python code.

> `extend` *and* `include` *are special template directives, not Python commands.*

Layouts are used to encapsulate page commonality (headers, footers, menus), and though they are not mandatory, they will make your application easier to write and maintain. In particular, we suggest writing layouts that take advantage of the following variables that can be set in the controller. Using these well known variables will help make your layouts interchangeable:

```
response.title
response.subtitle
response.author
response.keywords
response.description
response.flash
response.menu
```

These are all strings and their meaning should be obvious, except for `response.menu`. The `response.menu` menu is a list of three-element tuples. The three elements are: the link name, a boolean representing whether the link is active (is the current link), and the URL of the linked page. For example:

```
response.menu = [['Google', False', 'http://www.google.com'],
                 ['Index', True, URL(r=request, f='index')]]
```

We also recommend that you use:

```
{{include 'web2py_ajax.html'}}
```

in the HTML head, since this will include the jQuery libraries and define some backward-compatible JavaScript functions for special effects and Ajax.

Here is a minimal "layout.html" page based on the preceding recommendations:

```
<!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>
  <!-- define the meta tags -->
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <meta name="keywords" content="{{=response.keywords}}" />
  <meta name="description" content="{{=response.description}}" />
  <meta name="author" content="{{=response.author}}" />

  <!-- choose a title or use the application name -->
  <title>{{=response.title or request.application)}}</title>

  <!-- include jQuery and other ajax functions -->
  {{include 'web2py_ajax.html'}}

  <!-- include a style.css file and optional js files -->
  <link href="{{=URL(r=request, c='static', f='style.css')}}"
        rel="stylesheet" type="text/css"/>
</head>
<body>

<!-- build your header -->
<div class="header">[Here goes the header]</div>

<!-- here is the menu -->
{{if response.menu:}}
<div id="menu">
<ul>
  <!-- loop over menu items -->
  {{=for _name, _active, _link in response.menu:}}
    <li><a href="{{=_link}}" class="{{='active' if _active else '
        inactive'}}">{{=_name}}</a></li>
  {{pass}}
</ul>
</div>
{{pass}}

<!-- here is the flash message -->
<div id="flash">{{=response.flash or ''}}</div>

<!-- here the extending view is included -->
{{include}}

<!-- here is the footer -->
<div class="footer">[created by {{=response.author}} with web2py]</
    div>
</body>
</html>
```

In the layout, it may sometimes be necessary to display variables that are defined in the extending view. This will not be a problem as long as the variables are defined before the "extend" directive. This behavior can be used to extend a layout in more than one place (a standard layout is extended at the point where the `{{include}}` directive occurs). The idea is to define view functions that generate separate portions of the page (for example: sidebar,

maincontent) and render them in different parts of the layout. The view functions are called in the layout at the points we want them rendered.

For example in the following layout:

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

The functions "maincontent" and "sidebar" are defined in the extending view, although in this example we allowed for the possibility that view does not define "sidebar" function. Here is the corresponding view:

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

Notice that the functions are defined in HTML (although they can also contain Python code) so that `response.write` is used to write their content (the functions do not return the content). This is why the layout calls the view function using `{{maincontent()}}` rather than `{{=maincontent()}}`.

5.5 Using the Template System to Generate Emails

It is possible to use the template system to generate emails. For example, consider the database table

```
db.define_table('person', Field('name'))
```

where you want to send to every person in the database the following message, stored in a view file "message.html":

```
Dear {{=person.name}},
You have won the second prize, a set of steak knives.
```

You can achieve this in the following way

```
>>> from gluon.tool import Mail
>>> mail = Mail(globals())
>>> mail.settings.server = 'smtp.gmail.com:587'
>>> mail.settings.sender = '...@somewhere.com'
>>> mail.settings.login = None or 'username:password'
>>> for person in db(db.person.id>0).select():
>>>     context = dict(person=person)
```

```
>>>     message = response.render('message.html', context)
>>>     mail.send(to=['who@example.com'],
>>>               subject='None',
>>>               message=message)
```

Most of the work is done in the statement

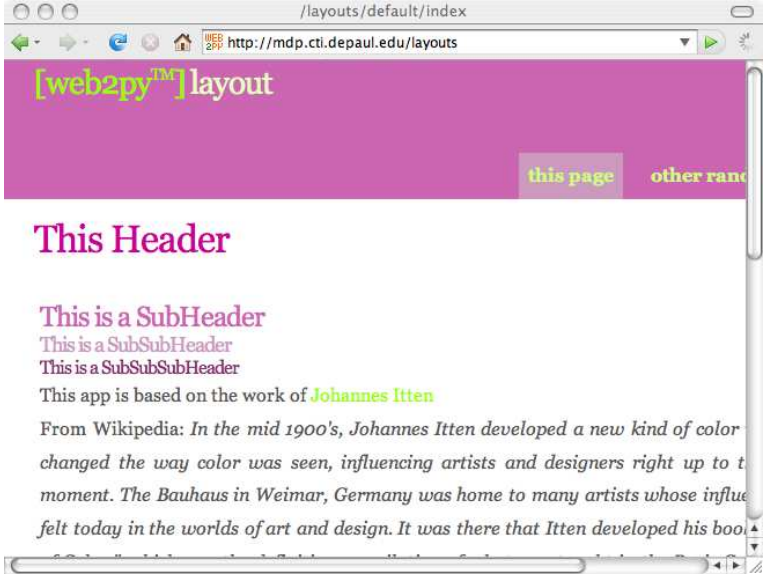
```
response.render('message.html', context)
```

It renders the view "file.html" with the variables defined in the dictionary "context", and it returns a string with the rendered email text. The context is a dictionary that contains variables that will be visible to the template file.

The same mechanism that is used to generate email text can also be used to generate SMS or any other type of message based on a template.

5.6 Layout Builder

The WEB2PY web site provides a layout builder to help us design new layout pages. Here is a screenshot:



This service is in a beta stage and has limited functionality. It is based on the work of Johannes Itten, an exponent of the Bauhaus, and creator of the modern "theory of color".

The website lets you select a base color and a few parameters of your layout, such as the height of the header, and it generates a sample layout (in

HTML with embedded CSS) with matching colors and a coherent look and feel. To use the layout, simply download it, and save it over the existing layout.html of your application.

CHAPTER 6

THE DATABASE ABSTRACTION LAYER

6.1 Dependencies

WEB2PY comes with a Database Abstraction Layer (DAL), an API that maps Python objects into database objects such as queries, tables, and records. The DAL dynamically generates the SQL in real time using the specified dialect for the database back end, so that you do not have to write SQL code or learn different SQL dialects (the term SQL is used generically), and the application will be portable among different types of databases. At the time of this writing, the supported databases are SQLite (which comes with Python and thus WEB2PY), PostgreSQL, MySQL, Oracle, MSSQL, FireBird, DB2, Informix and (partially) the Google App Engine (GAE). GAE is treated as a particular case in Chapter 11.

The Windows binary distribution works out of the box with SQLite and MySQL. The Mac binary distribution works out of the box with SQLite. To

use any other database back-end, run from the source distribution and install the appropriate driver for the required back end.

Once the proper driver is installed, start WEB2PY from source, and it will find the driver. Here is a list of drivers:

database	driver (source)
SQLite	sqlite3 or pysqlite2 or zxJDBC [53] (on Jython)
PostgreSQL	psycopg2 [54] or zxJDBC [53] (on Jython)
MySQL	MySQLdb [55]
Oracle	cx_Oracle [56]
MSSQL	pyodbc [57]
FireBird	kinterbasdb [58]
DB2	pyodbc [57]
Informix	informixdb [59]

WEB2PY defines the following classes that make up the DAL:

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

```
db = DAL('sqlite://storage.db')
```

- **Table** represents a database table. You do not directly instantiate Table; instead, `DAL.define_table` instantiates it.

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

  The most important methods of a Table are `insert`, `truncate`, `drop`, and `import_from_csv_file`.

- **DAL Field** represents a database field. It can be instantiated and passed as an argument to `DAL.define_table`.

- **DAL Rows** is the object returned by a database select. It can be thought of as a list of `DALStorage` rows:

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

- **DAL Storage** contains field values.

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

- **DAL Query** is an object that represents an SQL "where" clause:

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

- **DAL Set** is an object that represents a set of records. Its most important methods are `count`, `select`, `update`, and `delete`.

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

- **DAL Expression** is something that can be ORed, for example in `orderby` and `groupby` expressions. The Field class is derived from Expression. Here is an example.

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

6.2 Connection Strings

A connection with the database is established by creating an instance of the DAL object:

```
>>> db = DAL('sqlite://storage.db', pool_size=0)
```

`db` is not a keyword; it is a local variable that stores the connection object `DAL`. You are free to give it a different name. The constructor of `DAL` requires a single argument, the connection string. The connection string is the only WEB2PY code that depends on a specific back-end database. Here are examples of connection strings for specific types of supported back-end databases (in all cases, we assume the database is running from localhost on its default port and is named "test"):

- **SQLite**

```
'sqlite://storage.db'
```

- **MySQL**

```
'mysql://username:password@localhost/test'
```

- **PostgreSQL**

```
'postgres://username:password@localhost/test'
```

- **MSSQL**

```
'mssql://username:password@localhost/test'
```

- **FireBird**

```
'firebird://username:password@localhost/test'
```

- **Oracle**

```
'oracle://username:password@test'
```

- **DB2**

```
'db2://username:password@test'
```

- **Informix**

```
'informix://username:password@test'
```

- **Google BigTable on Google App Engine**

```
'gae'
```

Notice that in SQLite the database consists of a single file. If it does not exist, it is created. This file is locked every time it is accessed. In the case of MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Informix the database "test" must be created outside WEB2PY. Once the connection is established, WEB2PY will create, alter, and drop tables appropriately.

It is also possible to set the connection string to `None`. In this case DAL will not connect to any back-end database, but the API can still be accessed for testing. Examples of this will be discussed in Chapter 7.

Connection Pooling

The second argument of the DAL constructor is the `pool_size`; it defaults to 0.

For databases other than SQLite and GAE, it is slow to establish a new database connection for each request. To avoid this, WEB2PY implements a mechanism of connection pooling. When a connection is established, after the page has been served and the transaction completed, the connection is not closed, but it goes into a pool. When the next http request arrives, WEB2PY tries to pick a connection from the pool and use that one for a new transaction. If there are no available connections from the pool, a new connection is established.

Connections in the pools are shared sequentially among threads, in the sense that they may be used by two different but not simultaneous threads. There is only one pool for each WEB2PY process.

When WEB2PY starts, the pool is always empty. The pool grows up to the minimum between the value of `pool_size` and the max number of concurrent

requests. This means that if `pool_size=10` but our server never receives more than 5 concurrent requests, then the actual pool size will only grow to 5. If `pool_size=0` then connection pooling is not used.

Connection pooling is ignored for SQLite, since it would not yield any benefit.

6.3 DAL, Table, Field

The best way to understand the DAL API is to try each function yourself. This can be done interactively via the WEB2PY shell, although ultimately, DAL code goes in the models and controllers.

Start by creating a connection. For the sake of example, you can use SQLite. Nothing in this discussion changes when you change the back-end engine.

```
>>> db = DAL('sqlite://storage.db')
```

The database is now connected and the connection is stored in the global variable `db`.

At any time you can retrieve the connection string.

```
>>> print db._uri
sqlite://storage.db
```

and the database name

```
>>> print db._dbname
sqlite
```

The connection string is called a `_uri` because it is an instance of a Uniform Resource Identifier.

The DAL allows multiple connections with the same database or with different databases, even databases of different types. For now, we will assume the presence of a single database since this is the most common situation.

The most important method of a DAL is `define_table`:

```
>>> db.define_table('person', Field('name'))
```

It defines, stores and returns a `Table` object called "person" containing a field (column) "name". This object can also be accessed via `db.person`, so you do not need to catch the return value. `define_table` checks whether or not the corresponding table exists. If it does not, it generates the SQL to create it and executes the SQL. If the table does exist but differs from the one being defined, it generates the SQL to alter the table and executes it. If a field has

changed type but not name, it will try to convert the data[4]. If the table exists and matches the current definition, it will leave it alone. In all cases it will create the `db.person` object that represents the table.

6.4 Migrations

We refer to this behavior as a "migration". WEB2PY logs all migrations and migration attempts in the file "databases/sql.log".

The first argument of `define_table` is always the table name. The other unnamed arguments are the fields (Field). The function also takes an optional last argument called "migrate" which must be referred to explicitly by name as in:

```
>>> db.define_table('person', Field('name'), migrate='person.table')
```

The value of migrate is the filename (in the "databases" folder for the application) where WEB2PY stores internal migration information for this table. These files are very important and should never be removed except when the entire database is dropped. In this case, the ".table" files have to be removed manually. By default, migrate is set to True. This causes WEB2PY to generate the filename from a hash of the connection string. If migrate is set to False, the migration is not performed, and WEB2PY assumes that the table exists in the datastore and it contains (at least) the fields listed in `define_table`. The best practice is to give an explicit name to the migrate table.

There may not be two tables in the same application with the same migrate filename.

These are the default values of a Field constructor:

```
Field(name, 'string', length=None, default=None,
      required=False, requires='<default>',
      ondelete='CASCADE', notnull=False, unique=False,
      uploadfield=True, widget=None, label=None, comment=None,
      writable=True, readable=True, update=None, authorize=None,
      autodelete=False, represent=None)
```

Not all of them are relevant for every field. "length" is relevant only for fields of type "string". "uploadfield" and "authorize" are relevant only for fields of type "upload". "ondelete" is relevant only for fields of type "reference" and "upload".

- `length` sets the maximum length of a "string", "password" or "upload" field. If `length` is not specified a default value is used but the default

[4]If you do not want this, you need to redefine the table twice, the first time, letting WEB2PY drop the field by removing it, and the second time adding the newly defined field so that WEB2PY can create it.

value is not guanarteed to be backward compatible. *To avoind unwanted migrations on upgrades, we reccommend that you always specify the length for string, password and upload fields.*

- `default` sets the default value for the field. The default value is used when performing an insert if a value is not explicitly specified. It is also used to prepopulate forms built from the table using SQLFORM.
- `required` tells the DAL that no insert should be allowed on this table if a value for this field is not explicitly specified.
- `requires` is a validator or a list of validators. This is not used by the DAL, but it is used by SQLFORM. The default validators for the given types are shown in the following table:

field type	default field validators
string	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	
upload	
reference	

> *Notice that* `requires=...` *is enforced at the level of forms,* `required=True` *is enforced at the level of the DAL (insert), while* `notnull`, `unique` *and* `ondelete` *are enforced at the level of the database. While they sometimes may seem redundant, it is important to maintain the distinction when programming with the DAL.*

- `ondelete` translates into the "ON DELETE" SQL statement. By default "CASCADE" tells the database that when it deletes a record, it should also delete all records that refer to it.
- `notnull=True` translates into the "NOT NULL" SQL statement. It asks the database to prevent null values for the field.
- `unique=True` translates into the "UNIQUE" SQL statement. It asks the database to make sure that values of this field are unique within the table.

- `uploadfield` applies only to fields of type "upload". A field of type "upload" stores the name of a file saved somewhere else, by default on the filesystem under the application "uploads/" folder. If `uploadfield` is set, then the file is stored in a blob field within the same table and the value of `uploadfield` is the name of the blob field. This will be discussed in more detail later in the context of SQLFORM.

- `widget` must be one of the available widget objects, including custom widgets, for example

```
db.mytable.myfield.widget = SQLFORM.widgets.string.widget
```

  A list of available widgets will be discussed later. Each field type has a default widget.

- `label` is a string (or something that can be serialized to a string) that contains the label to be used for this field in autogenerated forms.

- `comment` is a string (or something that can be serialized to a string) that contains a comment associated with this field, and will be displayed to the right of the input field in the autogenerated forms.

- `writable` if a field is writable, it can be edited in autogenerated create and update forms.

- `readable` if a field is readable, it will be visible in readonly forms. If a field is neither readable nor writable, it will not be displayed in create and update forms.

- `update` contains the default value for this field when the record is updated.

- `authorize` can be used to require access control on the corresponding field, for "upload" fields only. It will be discussed more in detail in the context of Authentication and Authorization.

- `autodelete` determines if the corresponding uploaded file should be deleted when the record referencing the file is deleted. For "upload" fields only.

- `represent` can be None or can point to a function that takes a field value and returns an alternate representation for the field value. Examples:

```
db.mytable.name.represent = lambda name: name.capitalize()
db.mytable.other_id.represent = lambda id:
    db.other[id].somefield
db.mytable.some_uploadfield.represent = lambda value: \
    A('get it', _href=URL(r=request, f='download', args=value))
```

"blob" fields are also special. By default, binary data is encoded in base64 before being stored into the actual database field, and it is decoded when extracted. This has the negative effect of using 25% more storage space than necessary in blob fields, but has two advantages. On average it reduces the amount of data communicated between WEB2PY and the database server, and it makes the communication independent of back-end-specific escaping conventions.

You can query the database for existing tables:

```
>>> print db.tables
['person']
```

You can also query a table for existing fields:

```
>>> print db.person.fields
['id', 'name']
```

Do not declare a field called "id", because one is created by WEB2PY anyway. Every table has a field called "id" by default. It is an auto-increment integer field (starting at 1) used for cross-reference and for making every record unique, so "id" is a primary key. (Note: the id's starting at 1 is back-end specific. For example, this does not apply to the Google App Engine (GAE).)

You can query for the type of a table:

```
>>> print type(db.person)
<class 'gluon.sql.Table'>
```

and you can access a table from the DAL connection using:

```
>>> print type(db['person'])
<class 'gluon.sql.Table'>
```

Similarly you can access fields from their name in multiple equivalent ways:

```
>>> print type(db.person.name)
<class 'gluon.sql.Field'>
>>> print type(db.person['name'])
<class 'gluon.sql.Field'>
>>> print type(db['person']['name'])
<class 'gluon.sql.Field'>
```

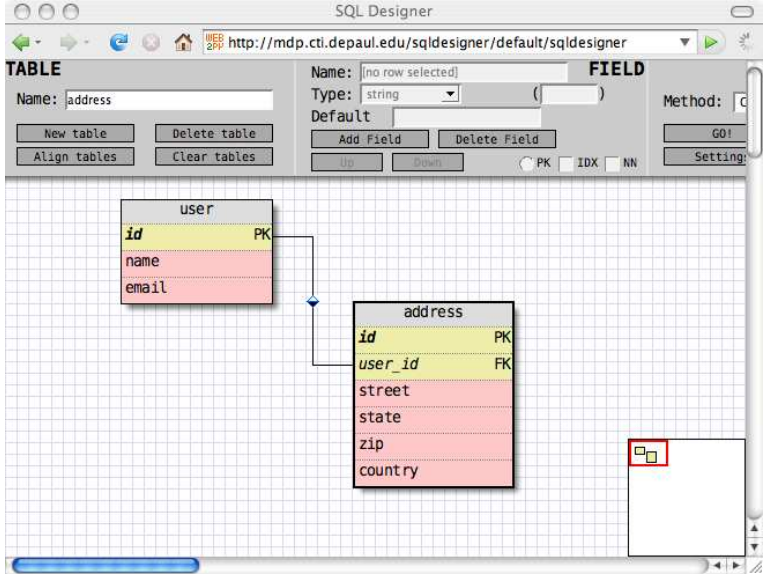
Given a field, you can access the attributes set in its definition:

```
>>> print db.person.name.type
string
>>> print db.person.name.unique
False
>>> print db.person.name.notnull
False
>>> print db.person.name.length
32
```

including its parent table, tablename, and parent connection:

```
>>> db.person.name._table == db.person
True
>>> db.person.name._tablename == 'person'
True
>>> db.person.name._db == db
True
```

The WEB2PY web site provides resources to help in the early stages of development via an online SQL designer that allows you to design a model visually and download the corresponding WEB2PY model [60]. Here is a screenshot:



This service is currently a beta version. It only works with Firefox, does not allow re-loading of models, and does not always define tables in the right order.

Once you define a table with references, it can only reference tables previously defined.

insert

Given a table, you can insert records

```
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
```

Insert returns the unique "id" value of each record inserted.

You can truncate the table, i.e., delete all records and reset the counter of the id.

```
>>> db.person.truncate()
```

Now, if you insert a record again, the counter starts again at 1 (this is back-end specific and does not apply to GAE):

```
>>> db.person.insert(name="Alex")
1
```

commit and rollback

No create, drop, insert, truncate, delete, or update operation is actually committed until you issue the commit command

```
>>> db.commit()
```

To check it let's insert a new record:

```
>>> db.person.insert(name="Bob")
2
```

and roll back, i.e., ignore all operations since the last commit:

```
>>> db.rollback()
```

If you now insert again, the counter will again be set to 2, since the previous insert was rolled back.

```
>>> db.person.insert(name="Bob")
2
```

Code in models, views and controllers is enclosed in WEB2PY code that looks like this:

```
try:
     execute models, controller function and view
except:
     rollback all connections
     log the traceback
     send a ticket to the visitor
else:
     commit all connections
     save cookies, sessions and return the page
```

There is no need to ever call `commit` or `rollback` explicitly in WEB2PY unless one needs more granular control.

executesql

The DAL allows you to explicitly issue SQL statements.

```
>>> print db.executesql('SELECT * FROM person;')
[(1, u'Massimo'), (2, u'Massimo')]
```

In this case, the return values are not parsed or transformed by the DAL, and the format depends on the specific database driver. This usage with selects is normally not needed, but it is more common with indexes.

_lastsql

Whether SQL was executed manually using executesql or was SQL generated by the DAL, you can always find the SQL code in `db._lastsql`. This is useful for debugging purposes:

```
>>> rows = db().select(db.person.ALL)
>>> print db._lastsql
SELECT person.id, person.name FROM person;
```

> WEB2PY *never generates queries using the "*" operator.* WEB2PY *is always explicit when selecting fields.*

drop

Finally, you can drop tables and all data will be lost:

```
>>> db.person.drop()
```

Indexes

Currently the DAL API does not provide a command to create indexes on tables, but this can be done using the **executesql** command. This is because the existence of indexes can make migrations complex, and it is better to deal with them explicitly. Indexes may be needed for those fields that are used in recurrent queries.

Here is an example of how to create an index using SQL in SQLite:

```
>>> db = DAL('sqlite://storage.db')
>>> db.define_table('person', Field('name'))
>>> db.executesql('CREATE INDEX IF NOT EXISTS myidx ON person name;)
```

Other database dialects have very similar syntaxes but may not support the optional "IF NOT EXISTS" directive.

Legacy Databases

WEB2PY can connect to legacy databases under some conditions:

- Each table must have a unique auto-increment integer field called "id"
- Records must be referenced exclusively using the "id" field.

If these conditions are not met, it is necessary to manually ALTER TABLE to conform them to these requirements, or they cannot be accessed by WEB2PY.

This should not be thought of as a limitation, but rather, as one of the many ways WEB2PY encourages you to follow good practices.

> *When accessing an existing table, i.e., a table not created by* WEB2PY *in the current application, always set* `migrate=False`.

Distributed Transaction

This feature is only supported with PostgreSQL, because it provides an API for two-phase commits.

Assuming you have two (or more) connections to distinct PostgreSQL databases, for example:

```
db_a = DAL('postgres://...')
db_b = DAL('postgres://...')
```

In your models or controllers, you can commit them concurrently with:

```
DAL.distributed_transaction_commit(db_a, db_b)
```

On failure, this function rolls back and raises an `Exception`.

In controllers, when one action returns, if you have two distinct connections and you do not call the above function, WEB2PY commits them separately. This means there is a possibility that one of the commits succeeds and one fails. The distributed transaction prevents this from happening.

6.5 Query, Set, Rows

Let's consider again the table defined (and dropped) previously and insert three records:

```
>>> db.define_table('person', Field('name'))
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
>>> db.person.insert(name="Carl")
3
```

You can store the table in a variable. For example, with variable `person`, you could do:

```
>>> person = db.person
```

You can also store a field in a variable such as `name`. For example, you could also do:

```
>>> name = person.name
```

You can even build a query (using operators like ==, !=, <, >, <=, >=, like, belongs) and store the query in a variable `q` such as in:

```
>>> q = name=='Alex'
```

When you call `db` with a query, you define a set of records. You can store it in a variable `s` and write:

```
>>> s = db(q)
```

Notice that no database query has been performed so far. DAL + Query simply define a set of records in this db that match the query. WEB2PY determines from the query which table (or tables) are involved and, in fact, there is no need to specify that.

select

Given a Set, `s`, you can fetch the records with the command `select`:

```
>>> rows = s.select()
```

It returns an iterable object of class `gluon.sql.Rows` whose elements are `gluon.sql.DALStorage`. DALStorage objects act like dictionaries, but their elements can also be accessed as attributes, like `gluon.storage.Storage`.The former differ from the latter vecause its values are readonly.

The Rows object allows looping over the result of the select and printing the selected field values for each row:

```
>>> for row in rows:
        print row.id, row.name
1 Alex
```

You can do all the steps in one statement:

```
>>> for row in db(db.person.name=='Alex').select():
        print row.name
Alex
```

The select command can take arguments. All unnamed arguments are interpreted as the names of the fields that you want to fetch. For example, you can be explicit on fetching field "id" and field "name":

```
>>> for row in db().select(db.person.id, db.person.name):
        print row.name
Alex
Bob
Carl
```

The table attribute ALL allows you to specify all fields:

```
>>> for row in db().select(db.person.ALL):
        print row.name
Alex
Bob
Carl
```

Notice that there is no query string passed to db. WEB2PY understands that if you want all fields of the table person without additional information then you want all records of the table person.

An equivalent alternative syntax is the following:

```
>>> for row in db(db.person.id > 0).select():
        print row.name
Alex
Bob
Carl
```

and WEB2PY understands that if you ask for all records of the table person (id > 0) without additional information, then you want all the fields of table person.

A Rows object is a container for:

```
rows.colnames
rows.response
```

`colnames` is a list of the column names returned by the raw select. `response` is a list of tuples which contains the raw response of select, before being parsed and converted to the proper WEB2PY format.

While a Rows object cannot be pickled nor serialized by XML-RPC, `colnames` and `response` can.

Again, many of these options are back-end-specific. In this case, field selection works differently on the Google App Engine.

Serializing Rows in Views

The result of a select can be displayed in a view with the following syntax:

```
{{extend 'layout.html'}}
<h1>Records</h2>
{{=db().select(db.person.ALL)}}
```

and it is automatically converted into an HTML table with a header containing the column names and one row per record. The rows are marked as alternating class "even" and class "odd". Under the hood, the Rows is first converted into a SQLTABLE object (not to be confused with Table) and then serialized. The values extracted from the database are also formatted by the validators associated to the field and then escaped. (Note: Using a db in this way in a view is usually not considered good MVC practice.)

orderby, groupby, limitby, distinct

The `select` command takes five optional arguments: orderby, groupby, limitby, left and cache. Here we discuss the first three.

You can fetch the records sorted by name:

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name):
        print row.name
Alex
Bob
Carl
```

You can fetch the records sorted by name in reverse order (notice the ~):

```
>>> for row in db().select(db.person.ALL, orderby=~db.person.name):
        print row.name
Carl
Bob
Alex
```

And you can sort the records according to multiple fields by concatenating them with a "|":

```
>>> for row in db().select(db.person.ALL, orderby=~db.person.name|db.
    person.id):
        print row.name
Carl
Bob
Alex
```

Using groupby together with orderby, you can group records with the same value for the specified field (this is backend specific, and is not on the GAE):

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name,
                           groupby=db.person.name):
        print row.name
```

```
Alex
Bob
Carl
```

With the argument `distinct=True`, you can specify that you only want to select distinct records. This has the same effect as grouping using all specified fields except that it does not require sorting. When using distinct it is important not to select ALL fields, and in particular not to select the "id" field, else all records will always be distinct.

Here is an example:

```
>>> for row in db().select(db.person.name, distinct=True):
        print row.name
Alex
Bob
Carl
```

With limitby, you can select a subset of the records (in this case, the first two starting at zero):

```
>>> for row in db().select(db.person.ALL, limitby=(0, 2)):
        print row.name
Alex
Bob
```

Currently, "limitby" is only partially supported on MSSQL since the Microsoft database does not provide a mechanism to fetch a subset of records not starting at 0.

Logical Operators

Queries can be combined using the binary AND operator "&":

```
>>> rows = db((db.person.name=='Alex') & (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
4 Alex
```

and the binary OR operator "|":

```
>>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
1 Alex
```

You can negate a query (or sub-query) with the `!=` binary operator:

```
>>> rows = db((db.person.name!='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

or by explicit negation with the `~` unary operator:

```
>>> rows = db(~(db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

Due to Python restrictions in overloading "AND" and "OR" operators, these cannot be used in forming queries. The binary operators must be used instead.

count, delete, update

You can count records in a set:

```
>>> print db(db.person.id > 0).count()
3
```

You can delete records in a set:

```
>>> db(db.person.id > 3).delete()
```

And you can update all records in a set by passing named arguments corresponding to the fields that need to be updated:

```
>>> db(db.person.id > 3).update(name='Ken')
```

Expressions

The value assigned an update statement can be an expression. For example consider this model

```
>>> db.define_table('person',
        Field('name'),
        Field('visits', 'integer', default=0))
>>> db(db.person.name == 'Massimo').update(
        visits = db.person.visits + 1)
```

The values used in queries can also be expressions

```
>>> db.define_table('person',
        Field('name'),
        Field('visits', 'integer', default=0),
        Field('clicks', 'integer', default=0))
>>> db(db.person.visits == db.person.clicks + 1).delete()
```

update_record

WEB2PY also allows updating a single record that is already in memory using `update_record`

```
>>> rows = db(db.person.id > 2).select()
>>> row = rows[0]
>>> row.update_record(name='Curt')
```

6.6 One to Many Relation

To illustrate how to implement one to many relations with the WEB2PY DAL, define another table "dog" that refers to the table "person" which we redefine here:

```
>>> db.define_table('person',
                    Field('name'))
>>> db.define_table('dog',
                    Field('name'),
                    Field('owner', db.person))
```

Table "dog" has two fields, the name of the dog and the owner of the dog. When a field type is another table, it is intended that the field reference the other table by its id. In fact, you can print the actual type value and get:

```
>>> print db.dog.owner.type
reference person
```

Now, insert three dogs, two owned by Alex and one by Bob:

```
>>> db.dog.insert(name='Skipper', owner=1)
1
>>> db.dog.insert(name='Snoopy', owner=1)
2
>>> db.dog.insert(name='Puppy', owner=2)
3
```

You can select as you did for any other table:

```
>>> for row in db(db.dog.owner==1).select():
        print row.name
Skipper
Snoopy
```

Because a dog has a reference to a person, a person can have many dogs, so a record of table person now acquires a new attribute dog, which is a Set, that defines the dogs of that person. This allows looping over all persons and fetching their dogs easily:

```
>>> for person in db().select(db.person.ALL):
        print person.name
        for dog in person.dog.select():
            print '    ', dog.name
Alex
     Skipper
```

```
     Snoopy
Bob
     Puppy
Carl
```

Inner Joins

Another way to achieve a similar result is by using a join, specifically an INNER JOIN. WEB2PY performs joins automatically and transparently when the query links two or more tables as in the following example:

```
>>> rows = db(db.person.id==db.dog.owner).select()
>>> for row in rows:
        print row.person.name, 'has', row.dog.name
Alex has Skipper
Alex has Snoopy
Bob has Puppy
```

Observe that WEB2PY did a join, so the rows now contain two records, one from each table, linked together. Because the two records may have fields with conflicting names, you need to specify the table when extracting a field value from a row. This means that while before you could do:

```
row.name
```

and it was obvious whether this was the name of a person or a dog, in the result of a join you have to be more explicit and say:

```
row.person.name
```

or:

```
row.dog.name
```

Left Outer Join

Notice that Carl did not appear in the list above because he has no dogs. If you intend to select on persons (whether they have dogs or not) and their dogs (if they have any), then you need to perform a LEFT OUTER JOIN. This is done using the argument "left" of the select command. Here is an example:

```
>>> rows=db().select(db.person.ALL, db.dog.ALL, left=db.dog.on(db.
    person.id==db.dog.owner))
>>> for row in rows:
        print row.person.name, 'has', row.dog.name
Alex has Skipper
Alex has Snoopy
Bob has Puppy
Carl has None
```

where:

```
left = db.dog.on(...)
```

does the left join query. Here the argument of `db.dog.on` is the condition required for the join (the same used above for the inner join). In the case of a left join, it is necessary to be explicit about which fields to select.

Grouping and Counting

When doing joins, sometimes you want to group rows according to certain criteria and count them. For example, count the number of dogs owned by every person. WEB2PY allows this as well. First, you need a count operator. Second, you want to join the person table with the dog table by owner. Third, you want to select all rows (person + dog), group them by person, and count them while grouping:

```
>>> count = db.person.id.count()
>>> for row in db(db.person.id==db.dog.owner).select(db.person.name,
    count, groupby=db.person.id):
        print row.person.name, row._extra[count]
Alex 2
Bob 1
```

Notice the count operator (which is built-in) is used as a field. The only issue here is in how to retrieve the information. Each row clearly contains a person and the count, but the count is not a field of a person nor is it a table. So where does it go? It goes into a dictionary called `_extra`. This dictionary exists for every row returned by a select when you fetch special objects from the database that are not table fields.

6.7 How to see SQL

Sometimes you need to generate the SQL but not execute it. This is easy to do with WEB2PY since every command that performs database IO has an equivalent command that does not, and simply returns the SQL that would have been executed. These commands have the same names and syntax as the functional ones, but they start with an underscore:

Here is `_insert`

```
>>> print db.person._insert(name='Alex')
INSERT INTO person(name) VALUES ('Alex');
```

Here is `_count`

```
>>> print db(db.person.name=='Alex')._count()
SELECT count(*) FROM person WHERE person.name='Alex';
```

Here is `_select`

```
>>> print db(db.person.name=='Alex')._select()
SELECT person.id, person.name FROM person WHERE person.name='Alex';
```

Here is `_delete`

```
>>> print db(db.person.name=='Alex')._delete()
DELETE FROM person WHERE person.name='Alex';
```

And finally, here is `_update`

```
>>> print db(db.person.name=='Alex')._update()
UPDATE person SET  WHERE person.name='Alex';
```

6.8 Exporting and Importing Data

CSV (one table at a time)

When a DALRows object is converted to a string it is automatically serialized in CSV:

```
>>> rows = db(db.person.id==db.dog.owner).select()
>>> print rows
person.id,person.name,dog.id,dog.name,dog.owner
1,Alex,1,Skipper,1
1,Alex,2,Snoopy,1
2,Bob,3,Puppy,2
```

You can serialize a single table in CSV and store it in a file "test.csv":

```
>>> open('test.csv', 'w').write(str(db(db.person.id).select()))
```

and you can easily read it back with:

```
>>> db.person.import_from_csv_file(open('test.csv', 'r'))
```

When importing, WEB2PY looks for the field names in the CSV header. In this example, it finds two columns: "person.id" and "person.name". It ignores the "person." prefix, and it ignores the "id" fields. Then all records are appended and assigned new ids. Both of these operations can be performed via the appadmin web interface.

CSV (all tables at once)

In WEB2PY, you can backup/restore an entire database with two commands:
To export:

```
>>> db.export_to_csv_file(open('somefile.csv', 'wb'))
```

To import:

```
>>> db.import_from_csv_file(open('somefile.csv', 'rb'))
```

This mechanism can be used even if the importing database is of a different type than the exporting database. The data is stored in "somefile.csv" as a CSV file where each table starts with one line that indicates the tablename, and another line with the fieldnames:

```
TABLE tablename
field1, field2, field3, ...
```

Two tables are separated by [5]`\r\n\r\n`". The file ends with the line

```
END
```

The file does not include uploaded files if these are not stored in the database. In any case it is easy enough to zip the "uploads" folder separately.

When importing, the new records will be appended to the database if it is not empty. In general the new imported records will not have the same record id as the original (saved) records but WEB2PY will restore references so they are not broken, even if the id values may change.

If a table contains a field called "uuid", this field will be used to identify duplicates. Also, if an imported record has the same "uuid" as an existing record, the previous record will be updated.

CSV and remote Database Synchronization

Consider the following model:

```
db = DAL('sqlite:memory:')
db.define_table('person',
    Field('name'))
db.define_table('dog',
    Field('owner', db.person),
    Field('name'))
db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s')

if not db(db.person.id>0).count():
    id = db.person.insert(name="Massimo")
    db.dog.insert(owner=id, name="Snoopy")
```

Each record is identified by an ID and referenced by that ID. If you have two copies of the database used by distinct WEB2PY installations, the ID is unique only within each database and not across the databases. This is a problem when merging records from different databases.

5"

In order to make a record uniquely identifiable across databases, they must:

- have a unique id (UUID),
- have a timestamp (to figure out which one is more recent if multiple copies),
- reference the UUID instead of the id.

This can be achieved without modifying WEB2PY. Here is what to do:

- Change the above model into:

```
db.define_table('person',
    Field('uuid', length=64, default=uuid.uuid4()),
    Field('modified_on', 'datetime', default=now),
    Field('name'))
db.define_table('dog',
    Field('uuid', length=64, default=uuid.uuid4()),
    Field('modified_on', 'datetime', default=now),
    Field('owner', length=64),
    Field('name'))

db.dog.owner.requires = IS_IN_DB(db,'person.uuid','%(name)s')

if not db(db.person.id).count():
    id = uuid.uuid4()
    db.person.insert(name="Massimo", uuid=id)
    db.dog.insert(owner=id, name="Snoopy")
```

- Create a controller action to export the database:

```
def export():
    s = StringIO.StringIO()
    db.export_to_csv_file(s)
    response.headers['Content-Type'] = 'text/csv'
    return s.getvalue()
```

- Create a controller action to import a saved copy of the other database and sync records:

```
def import_and_sync():
    form = FORM(INPUT(_type='file', _name='data'),
                INPUT(_type='submit'))
    if form.accepts(request.vars):
        db.import_from_csv_file(form.vars.data.file,unique=False
            )
        # for every table
        for table in db.tables:
            # for every uuid, delete all but the latest
            items = db(db[table].id>0).select(db[table].id,
                       db[table].uuid,
                       orderby=~db[table].modified_on,
                       groupby=db[table].uuid)
```

```
            for item in items:
                db((db[table].uuid==item.uuid)&\
                   (db[table].id!=item.id)).delete()
    return dict(form=form)
```

- Create an index manually to make the search by uuid faster.

Notice that steps 2 and 3 work for every database model; they are not specific for this example.

Alternatively, you can use XML-RPC to export/import the file.

If the records reference uploaded files, you also need to export/import the content of the uploads folder. Notice that files therein are already labeled by UUIDs so you do not need to worry about naming conflicts and references.

HTML/XML (one table at a time)

DALRows objects also have an `xml` method (like helpers) that serializes it to XML/HTML:

```
>>> rows = db(db.person.id > 0).select()
>>> print rows.xml()
<table><thead><tr><th>person.id</th><th>person.name</th><th>dog.id</
    th><th>dog.name</th><th>dog.owner</th></tr></thead><tbody><tr
    class="even"><td>1</td><td>Alex</td><td>1</td><td>Skipper</td><td
    >1</td></tr><tr class="odd"><td>1</td><td>Alex</td><td>2</td><td>
    Snoopy</td><td>1</td></tr><tr class="even"><td>2</td><td>Bob</td
    ><td>3</td><td>Puppy</td><td>2</td></tr></tbody></table>
```

If you need to serialize the DALRows in any other XML format with custom tags, you can easily do that using the universal TAG helper and the * notation:

```
>>> rows = db(db.person.id > 0).select()
>>> print TAG.result(*[TAG.row(*[TAG.field(r[f], _name=f) for f in db
    .person.fields]) for r in rows])
<result><row><field name="id">1</field><field name="name">Alex</field
    ></row><row><field name="id">2</field><field name="name">Bob</
    field></row><row><field name="id">3</field><field name="name">
    Carl</field></row></result>
```

6.9 Many to Many

In the previous examples, we allowed a dog to have one owner but one person could have many dogs. What if Skipper was owned by Alex and Curt? This requires a many-to-many relation, and it is realized via an intermediate table that links a person to a dog via an ownership relation.

Here is how to do it:

```
>>> db.define_table('person',
                    Field('name'))
>>> db.define_table('dog',
                    Field('name'))
>>> db.define_table('ownership',
                    Field('person', db.person),
                    Field('dog', db.dog))
```

the existing ownership relationship can now be rewritten as:

```
>>> db.ownership.insert(person=1, dog=1) # Alex owns Skipper
>>> db.ownership.insert(person=1, dog=2) # Alex owns Snoopy
>>> db.ownership.insert(person=2, dog=3) # Bob owns Puppy
```

Now you can add the new relation that Curt co-owns Skipper:

```
>>> db.ownership.insert(person=3, dog=1) # Curt owns Skipper too
```

Because you now have a three-way relation between tables, it may be convenient to define a new set on which to perform operations:

```
>>> persons_and_dogs = db((db.person.id==db.ownership.person) & (db.
    dog.id==db.ownership.dog))
```

Now it is easy to select all persons and their dogs from the new Set:

```
>>> for row in persons_and_dogs.select():
        print row.person.name, row.dog.name
Alex Skipper
Alex Snoopy
Bob Puppy
Curt Skipper
```

Similarly, you can search for all dogs owned by Alex:

```
>>> for row in persons_and_dogs(db.person.name=='Alex').select():
        print row.dog.name
Skipper
Snoopy
```

and all owners of Skipper:

```
>>> for row in persons_and_dogs(db.dog.name=='Skipper').select():
        print row.owner.name
Alex
Curt
```

A lighter alternative to Many 2 Many relations is a tagging. Tagging is discussed in the context of the IS_IN_DB validator. Tagging works even on database backends that does not support JOINs like the Google App Engine.

6.10 Other Operators

WEB2PY has other operators that provide an API to access equivalent SQL operators. Let's define another table "log" to store security events, their timestamp and severity, where the severity is an integer number.

```
>>> db.define_table('log', Field('event'),
                           Field('timestamp', 'datetime'),
                           Field('severity', 'integer'))
```

As before, insert a few events, a "port scan", an "xss injection" and an "unauthorized login". For the sake of the example, you can log events with the same timestamp but with different severities (1, 2, 3 respectively).

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> print db.log.insert(event='port scan', timestamp=now, severity=1)
1
>>> print db.log.insert(event='xss injection', timestamp=now,
    severity=2)
2
>>> print db.log.insert(event='unauthorized login', timestamp=now,
    severity=3)
3
```

like, upper, lower

Fields have a like operator that you can use to match strings:

```
>>> for row in db(db.log.event.like('port%')).select():
        print row.event
port scan
```

Here "port%" indicates a string starting with "port". The percent sign character, "%", is a wild-card character that means "any sequence of characters".

Similarly, you can use `upper` and `lower` methods to convert the value of the field to upper or lower case, and you can also combine them with the like operator.

```
>>> for row in db(db.log.event.upper().like('PORT%')).select():
        print row.event
port scan
```

year, month, day, hour, minutes, seconds

The date and datetime fields have day, month and year methods. The datetime and time fields have hour, minutes and seconds methods. Here is an example:

```
>>> for row in db(db.log.timestamp.year()==2009).select():
        print row.event
port scan
xss injection
unauthorized login
```

belongs

The SQL IN operator is realized via the belongs method which returns true when the field value belongs to the specified set (list of tuples):

```
>>> for row in db(db.log.severity.belongs((1, 2))).select():
        print row.event
port scan
xss injection
```

The DAL also allows a nested select as the argument of the belongs operator. The only caveat is that the nested select has to be a `_select`, not a `select`, and only one field has to be selected explicitly, the one that defines the set.

```
>>> bad_days = db(db.log.severity==3)._select(db.log.timestamp)
>>> for row in db(db.log.timestamp.belongs(bad_days)).select():
        print row.event
port scan
xss injection
unauthorized login
```

Previously, you have used the count operator to count records. Similarly, you can use the sum operator to add (sum) the values of a specific field from a group of records. As in the case of count, the result of a sum is retrieved via the `_extra` dictionary.

```
>>> sum = db.log.severity.sum()
>>> print db().select(sum)[0]._extra[sum]
6
```

6.11 Caching Selects

The select method also takes a cache argument, which defaults to None. For caching purposes, it should be set to a tuple where the first element is the cache model (cache.ram, cache.disk, etc.), and the second element is the expiration time in seconds.

In the following example, you see a controller that caches a select on the previously defined db.log table. The actual select fetches data from the backend database no more frequently than once every 60 seconds and stores the

result in cache.ram. If the next call to this controller occurs in less than 60 seconds since the last database IO, it simply fetches the previous data from cache.ram.

```
def cache_db_select():
    logs = db().select(db.log.ALL, cache=(cache.ram, 60))
    return dict(logs=logs)
```

> *The results of a select are complex, unpickleable objects; they cannot be stored in a session and cannot be cached in any other way than the one explained here.*

6.12 Shortcuts

The DAL supports various code-simplifying shortcuts. In particular:

```
db.mytable[id]
```

returns the record with the given `id` if it exists. If the `id` does not exist, it returns `None`.

```
del db.mytable[id]
```

deletes the record with the given `id`, if it exists.

```
db.mytable[0] = dict(myfield='somevalue')
```

creates a new record with field values specified by the dictionary on the right hand side.

```
db.mytable[id] = dict(myfield='somevalue')
```

updates an existing record with field values specified by the dictionary on the right hand side.

6.13 Self-Reference and Aliases

It is possible to define tables with fields that refer to themselves although the usual notation may fail. The following code would be wrong because it uses a variable `db.person` before it is defined:

```
db.define_table('person',
    Field('name'),
    Field('father_id', db.person),
    Field('mother_id', db.person))
```

The solution consists of using an alternate notation

```
db.define_table('person',
    Field('name'),
    Field('father_id', 'reference person'),
    Field('mother_id', 'reference person'))
```

In fact `db.tablename` and `"reference tablename"` are equivalent field types.

If the table refers to itself, then it is not possible to perform a JOIN to select a person and its parents without use of the SQL "AS" keyword. This is achieved in WEB2PY using the `with_alias`. Here is an example:

```
>>> Father = db.person.with_alias('father')
>>> Mother = db.person.with_alias('mother')
>>> db.person.insert(name='Massimo')
1
>>> db.person.insert(name='Claudia')
2
>>> db.person.insert(name='Marco', father_id=1, mother_id=2)
3
>>> rows = db().select(db.person.name, Father.name, Mother.name,
      left=(Father.on(Father.id==db.person.father_id),
            Mother.on(Mother.id==db.person.mother_id)))
>>> for row in rows:
        print row.person.name, row.father.name, row.mother.name
Massimo None None
Claudia None None
Marco Massimo Claudia
```

Notice that we have chosen to make a distinction between:

- "father_id": the field name used in the table "person";
- "father": the alias we want to use for the table referenced by the above field; this is communicated to the database;
- "Father": the variable used by WEB2PY to refer to that alias.

The difference is subtle, and there is nothing wrong in using the same name for the three of them:

```
db.define_table('person',
    Field('name'),
    Field('father', 'reference person'),
    Field('mother', 'reference person'))
>>> father = db.person.with_alias('father')
>>> mother = db.person.with_alias('mother')
>>> db.person.insert(name='Massimo')
1
>>> db.person.insert(name='Claudia')
2
>>> db.person.insert(name='Marco', father=1, mother=2)
3
>>> rows = db().select(db.person.name, father.name, mother.name,
      left=(father.on(father.id==db.person.father),
            mother.on(mother.id==db.person.mother)))
>>> for row in rows:
```

```
        print row.person.name, row.father.name, row.mother.name
Massimo None None
Claudia None None
Marco Massimo Claudia
```

But it is important to have the distinction clear in order to build correct queries.

6.14 Table Inheritance

It is possible to create a table that contains all the fields from another table. It is sufficient to pass the other table in place of a field to `define_table`. For example

```
db.define_table('person', Field('name'))

db.define_table('doctor', db.person, Field('specialization'))
```

It is also possible to define a dummy table that is not stored in a database in order to reuse it in multiple other places. For example:

```
current_user_id = (auth.user and auth.user.id) or 0

timestamp = db.Table(db, 'timestamp_table',
    Field('created_on', 'datetime', default=request.now),
    Field('created_by', db.auth_user, default=current_user_id),
    Field('updated_on', 'datetime', default=request.now),
    Field('updated_by', db.auth_user, update=current_user_id))

db.define_table('payment', timestamp, Field('amount', 'double'))
```

This example assumes that standard WEB2PY authentication is enabled.

CHAPTER 7

FORMS AND VALIDATORS

There are four distinct ways to build forms in WEB2PY:

- `FORM` provides a low-level implementation in terms of HTML helpers. A `FORM` object can be serialized into HTML and is aware of the fields it contains. A `FORM` object knows how to validate submitted form values.
- `SQLFORM` provides a high-level API for building create, update and delete forms from an existing database table.
- `SQLFORM.factory` is an abstraction layer on top of `SQLFORM` in order to take advantage of the form generation features even if there is no database present. It generates a form very similar to `SQLFORM` from the description of a table but without the need to create the database table.
- `CRUD` methods. These are functionally equivalent to SQLFORM and are based on SQLFORM, but provide a simpler notation.

All these forms are self-aware and, if the input does not pass validation, they can modify themselves and add error messages. The forms can be

queried for the validated variables and for error messages that have been generated by validation.

Arbitrary HTML code can be inserted into or extracted from the form using helpers.

7.1 FORM

Consider as an example a **test** application with the following "default.py" controller:

```
def display_form():
    return dict()
```

and the associated "default/display_form.html" view:

```
{{extend 'layout.html'}}
<h2>Input form</h2>
<form enctype="multipart/form-data"
      action="{{=request.url}}" method="post">
Your name:
<input name="name" />
<input type="submit" />
</form>
<h2>Submitted variables</h2>
{{=BEAUTIFY(request.vars)}}
```

This is a regular HTML form that asks for the user's name. When you fill the form and click the submit button, the form self-submits, and the variable `request.vars.name` and its value is displayed at the bottom.

You can generate the same form using helpers. This can be done in the view or in the action. Since WEB2PY processed the form in the action, it is OK to define the form in the action.

Here is the new controller:

```
def display_form():
   form=FORM('Your name:',
             INPUT(_name='name'),
             INPUT(_type='submit'))
   return dict(form=form)
```

and the associated "default/display_form.html" view:

```
{{extend 'layout.html'}}
<h2>Input form</h2>
{{=form}}
<h2>Submitted variables</h2>
{{=BEAUTIFY(request.vars)}}
```

The code so far is equivalent to the previous code, but the form is generated by the statement `{{=form}}` which serializes the `FORM` object.

Now we add one level of complexity by adding form validation and processing.

Change the controller as follows:

```
def display_form():
    form=FORM('Your name:',
              INPUT(_name='name', requires=IS_NOT_EMPTY()),
              INPUT(_type='submit'))
    if form.accepts(request.vars, session):
        response.flash = 'form accepted'
    elif form.errors:
        response.flash = 'form has errors'
    else:
        response.flash = 'please fill the form'
    return dict(form=form)
```

and the associated "default/display_form.html" view:

```
{{extend 'layout.html'}}
<h2>Input form</h2>
{{=form}}
<h2>Submitted variables</h2>
{{=BEAUTIFY(request.vars)}}
<h2>Accepted variables</h2>
{{=BEAUTIFY(form.vars)}}
<h2>Errors in form</h2>
{{=BEAUTIFY(form.errors)}}
```

Notice that:

- In the action, we added the `requires=IS_NOT_EMPTY()` validator for the input field "name".
- In the action, we added a call to `form.accepts(...)`
- In the view, we are printing `form.vars` and `form.errors` as well as the form and `request.vars`.

All the work is done by the `accepts` method of the `form` object. It filters the `request.vars` according to the declared requirements (expressed by validators). `accepts` stores those variables that pass validation into `form.vars`. If a field value does not meet a requirement, the failing validator returns an error and the error is stored in `form.errors`. Both `form.vars` and `form.errors` are `gluon.storage.Storage` objects similar to `request.vars`. The former contains the values that passed validation, for example:

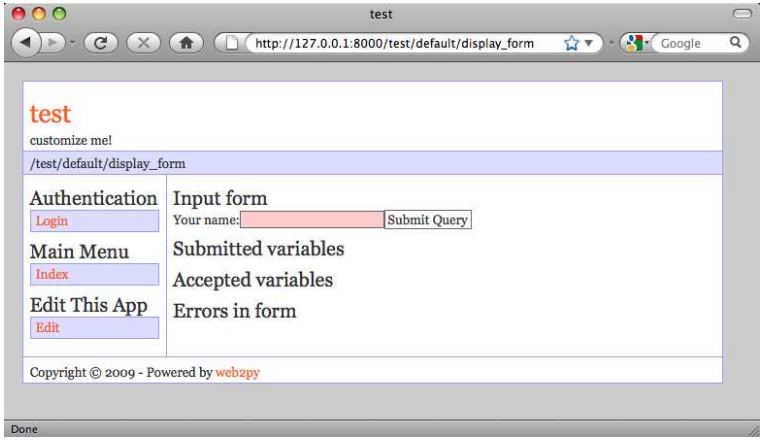
```
form.vars.name = "Max"
```

The latter contains the errors, for example:

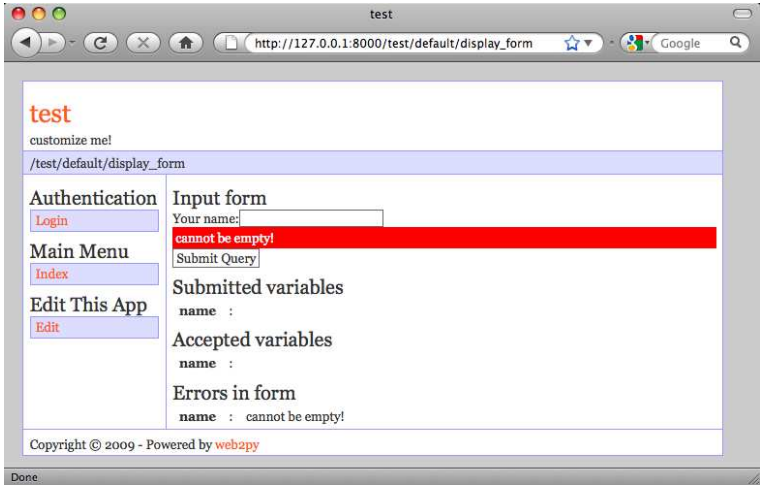
```
form.errors.name = "Cannot be empty!"
```

The `accepts` function returns `True` if the form is accepted and `False` otherwise. A form is not accepted if it has errors or when it has not been submitted (for example, the first time it is shown).

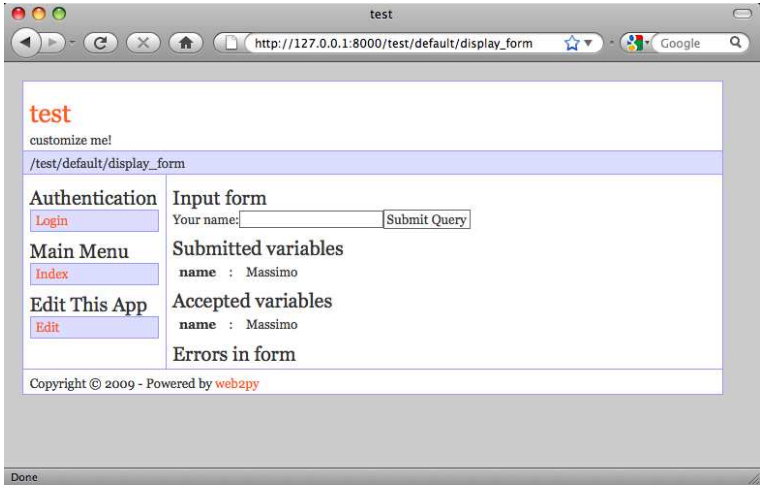
Here is how this page looks the first time it is displayed:



Here is how it looks upon invalid submission:



Here is how it looks upon a valid submission:



Hidden fields

When the above form object is serialized by `{{=form}}`, and because of the previous call to the `accepts` method, it now looks like this:

```
<form enctype="multipart/form-data" action="" method="post">
your name:
<input name="name" />
<input type="submit" />
<input value="783531473471" type="hidden" name="_formkey" />
<input value="default" type="hidden" name="_formname" />
</form>
```

Notice the presence of two hidden fields: "_formkey" and "_formname". Their presence is triggered by the call to `accepts` and they play two different and important roles:

- The hidden field called "_formkey" is a one-time token that WEB2PY uses to prevent double submission of forms. The value of this key is generated when the form is serialized and stored in the `session`. When the form is submitted this value must match, or else `accepts` returns `False` without errors as if the form was not submitted at all. This is because WEB2PY cannot determine whether the form was submitted correctly.
- The hidden field called "_formname" is generated by WEB2PY as a name for the form, but the name can be overridden. This field is necessary to allow pages that contain and process multiple forms. WEB2PY distinguishes the different submitted forms by their names.

The role of these hidden fields and their usage in custom forms and pages with multiple forms is discussed in more detail later in the chapter.

If the form above is submitted with an empty "name" field, the form does not pass validation. When the form is serialized again it appears as:

```
<form enctype="multipart/form-data" action="" method="post">
your name:
<input value="" name="name" />
<div class="error">cannot be empty!</div>
<input type="submit" />
<input value="783531473471" type="hidden" name="_formkey" />
<input value="default" type="hidden" name="_formname" />
</form>
```

Notice the presence of a DIV of class "error" in the serialized form. WEB2PY inserts this error message in the form to notify the visitor about the field that did not pass validation. The `accepts` method, upon submission, determines that the form is submitted, checks whether the field "name" is empty and whether it is required, and eventually inserts the error message from the validator into the form.

The base "layout.html" view is expected to handle DIVs of class "error". The default layout uses jQuery effects to make errors appear and slide down with a red background. See Chapter 10 for more details.

keepvalues

The full signature of the `accepts` method is the following:

```
form.accepts(vars, session=None, formname='default',
             keepvalues=False, onvalidation=None):
```

The optional argument `keepvalues` tells WEB2PY what to do when a form is accepted and there is no redirection, so the same form is displayed again. By default the form is cleared. If `keepvalues` is set to `True`, the form is prepopulated with the previously inserted values. This is useful when you have a form that is supposed to be used repeatedly to insert multiple similar records.

onvalidation

The `onvalidation` argument can be `None` or can be a function that takes the form and returns nothing. Such a function would be called and passed the form, immediately after validation (if validation passes) and before anything else happens. The purpose of this function is multifold. It can be used, for example, to perform additional checks on the form and eventually add errors

to the form. It can also be used to compute the values of some fields based on the values of other fields. It can be used to trigger some action (like sending an email) before a record is created/updated.

Here is an example:

```
db.define_table('numbers',
    Field('a', 'integer'),
    Field('b', 'integer'),
    Field('d', 'integer', readable=False, writable=False))

def my_form_processing(form):
    c = form.vars.a * form.vars.b
    if c < 0:
       form.errors.b = 'a*b cannot be negative'
    else:
       form.vars.c = c

def insert_numbers():
   form = SQLFORM(db.numbers)
   if form.accepts(request.vars, session,
                   onvalidation=my_form_processing)
       session.flash = 'record inserted'
       redirect(request.url)
   return dict(form=form)
```

Forms and redirection

The most common way to use forms is via self-submission, so that the submitted field variables are processed by the same action that generated the form. Once the form is accepted, it is unusual to display the current page again (something we are doing here only to keep things simple). It is more common to redirect the visitor to a "next" page.

Here is the new example controller:

```
def display_form():
    form = FORM('Your name:',
               INPUT(_name='name', requires=IS_NOT_EMPTY()),
               INPUT(_type='submit'))
    if form.accepts(request.vars, session):
        session.flash = 'form accepted'
        redirect(URL(r=request, f='next'))
    elif form.errors:
        response.flash = 'form has errors'
    else:
        response.flash = 'please fill the form'
    return dict(form=form)

def next():
    return dict()
```

In order to set a flash on the next page instead of the current page you must use `session.flash` instead of `response.flash`. WEB2PY moves the former into the latter after redirection. Note that using `session.flash` requires that you do not `session.forget()`.

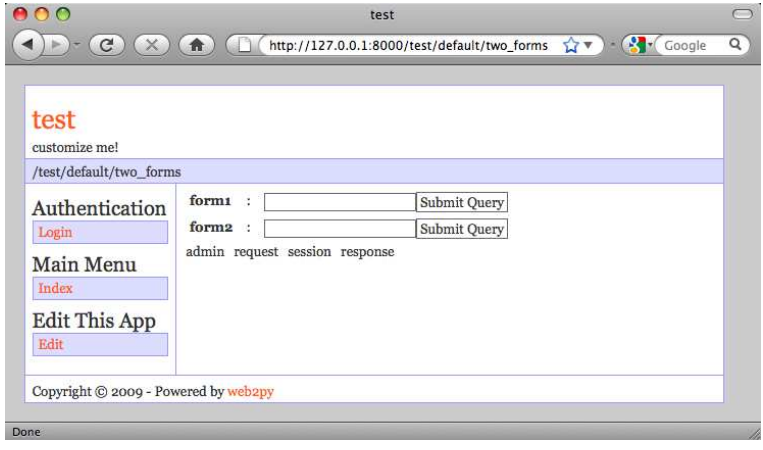
Multiple forms per page

The content of this section applies to both `FORM` and `SQLFORM` objects.

It is possible to have multiple forms per page, but you must allow WEB2PY to distinguish them. If these are derived by `SQLFORM` from different tables, then WEB2PY gives them different names automatically; otherwise you need to explicitly give them different form names. Moreover, when multiple forms are present on the same page, the mechanism for preventing double submission breaks, and you must omit the `session` argument when calling the `accepts` method. Here is an example:

```
def two_forms():
    form1 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
                 INPUT(_type='submit'))
    form2 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
                 INPUT(_type='submit'))
    if form1.accepts(request.vars, formname='form_one'):
        response.flash = 'form one accepted'
    if form2.accepts(request.vars, formname='form_two'):
        response.flash = 'form two accepted'
    return dict(form1=form1, form2=form2)
```

and here is the output it produces:



When the visitor submits an empty form1, only form1 displays an error; if the visitor submits an empty form2, only form2 displays an error message.

No self-submission

The content of this section applies to both `FORM` and `SQLFORM` objects. What we discuss here is possible but not recommended, since it is always good practice to have forms that self-submit. Sometimes, though, you don't have a choice, because the action that sends the form and the action that receives it belong to different applications.

It is possible to generate a form that submits to a different action. This is done by specifying the URL of the processing action in the attributes of the `FORM` or `SQLFORM` object. For example:

```
form = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit'), _action=URL(r=request, f='page_two'))

def page_one():
    return dict(form=form)

def page_two():
    if form.accepts(request.vars, formname=None):
         response.flash = 'form accepted'
    else:
         response.flash = 'there was an error in the form'
    return dict()
```

Notice that since both "page_one" and "page_two" use the same `form`, we have defined it only once by placing it outside of all the actions, in order not to repeat ourselves. The common portion of code at the beginning of a controller gets executed every time before giving control to the called action.

Since "page_one" does not call `accepts`, the form has no name and no key, so you must not pass the `session` and set `formname=None` in `accepts`, or the form will not validate when "page_two" receives it.

7.2 SQLFORM

We now move to the next level by providing the application with a model file:

```
db = DAL('sqlite://db.db')
db.define_table('person',
                Field('name', requires=IS_NOT_EMPTY()))
```

Modify the controller as follows:

```
def display_form():
   form = SQLFORM(db.person)
   if form.accepts(request.vars, session):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   else:
```

```
        response.flash = 'please fill out the form'
    return dict(form=form)
```

The view does not need to be changed.

In the new controller, you do not need to build a FORM, since the SQLFORM constructor built one from the table db.person defined in the model. This new form, when serialized, appears as:

```
<form enctype="multipart/form-data" action="" method="post">
  <table>
    <tr id="person_name__row">
       <td><label id="person_name__label"
                  for="person_name">Your name: </label></td>
       <td><input type="text" class="string"
                  name="name" value="" id="person_name" /></td>
       <td></td>
    </tr>
    <tr id="submit_record__row">
       <td></td>
       <td><input value="Submit" type="submit" /></td>
       <td></td>
    </tr>
  </table>
  <input value="9038845529" type="hidden" name="_formkey" />
  <input value="person" type="hidden" name="_formname" />
</form>
```

The automatically generated form is more complex than the previous low-level form. First of all, it contains a table of rows, and each row has three columns. The first column contains the field labels (as determined from the db.person), the second column contains the input fields (and eventually error messages), and the third column is optional and therefore empty (it can be populated with the fields in the SQLFORM constructor).

All tags in the form have names derived from the table and field name. This allows easy customization of the form using CSS and JavaScript. This capability is discussed in more detail in Chapter 10.

More important is that now the accepts method does a lot more work for you. As in the previous case, it performs validation of the input, but additionally, if the input passes validation, it also performs a database insert of the new record and stores in form.vars.id the unique "id" of the new record.

A SQLFORM object also deals automatically with "upload" fields by saving uploaded files in the "uploads" folder (after having them renamed safely to avoid conflicts and prevent directory traversal attacks) and stores their names (their new names) into the appropriate field in the database.

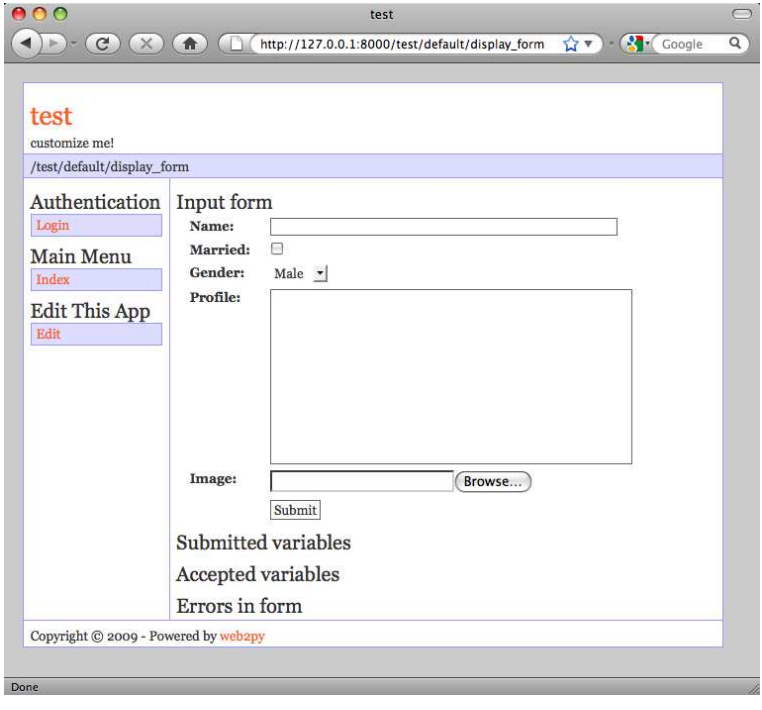
A SQLFORM displays "boolean" values with checkboxes, "text" values with textareas, values required to be in a definite set or a database with dropboxes, and "upload" fields with links that allow users to download the uploaded files.

It hides "blob" fields, since they are supposed to be handled differently, as discussed later.

For example, consider the following model:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('married', 'boolean'),
    Field('gender', requires=IS_IN_SET(['Male', 'Female', 'Other'])),
    Field('profile', 'text'),
    Field('image', 'upload'))
```

In this case, `SQLFORM(db.person)` generates the form shown below:



The `SQLFORM` constructor allows various customizations, such as displaying only a subset of the fields, changing the labels, adding values to the optional third column, or creating UPDATE and DELETE forms, as opposed to INSERT forms like the current one.

`SQLFORM` is the single biggest time-saver object in WEB2PY.

The class `SQLFORM` is defined in "gluon/sqlhtml.py". It can be easily extended by overloading its `xml` method, the method that serializes the objects, to change its output.

The signature for the `SQLFORM` constructor is the following:

```
SQLFORM(table, record=None, deletable=False,
        linkto=None, upload=None, fields=None, labels=None, col3={},
```

```
        submit_button='Submit', delete_label='Check to delete:',
        id_label='Record id: ', showid=True,
        readonly=False, comments=True, keepopts=[],
        ignore_rw=False, **attributes)
```

- The optional second argument turns the INSERT form into an UPDATE form for the specified record (see next subsection).
- If `deletable` is set to `True`, the UPDATE form displays a "Check to delete" checkbox. The value of the label if this field is set via the `delete_label` argument.
- `submit_button` sets the value of the submit button.
- `id_label` sets the label of the record "id"
- The "id" of the record is not shown if `showid` is set to `False`.
- `fields` is an optional list of field names that you want to display. If a list is provided, only fields in the list are displayed. For example:

```
fields = ['name']
```

- `labels` is a dictionary of field labels. The dictionary key is a field name and the corresponding value is what gets displayed as its label. If a label is not provided, WEB2PY derives the label from the field name (it capitalizes the field name and replaces underscores with spaces). For example:

```
labels = {'name':'Your Full Name:'}
```

- `col3` is a dictionary of values for the third column. For example:

```
col3 = {'name':A('what is this?',
      _href='http://www.google.com/search?q=define:name')}
```

- `linkto` and `upload` are optional URLs to user-defined controllers that allow the form to deal with reference fields. This is discussed in more detail later in the section.
- `readonly`. If set to True, displays the form as readonly
- `comments`. If set to False, does not display the col3 comments
- `ignore_rw`. Normally, for a create/update form, only fields marked as writable=True are shown, and for readonly forms, only fields marked as readable=True are shown. Setting `ignore_rw=True` causes those constraints to be ignored, and all fields are displayed. This is mostly used

in the appadmin interface to display all fields for each table, overriding what the model indicates.

- Optional `attributes` are arguments starting with underscore that you want to pass to the `FORM` tag that renders the `SQLFORM` object. Examples are:

```
_action = '.'
_method = 'POST'
```

There is a special `hidden` attribute. When a dictionary is passed as `hidden`, its items are translated into "hidden" INPUT fields (see the example for the `FORM` helper in Chapter 5).

Insert/Update/Delete SQLFORM

If you pass a record as optional second argument to the `SQLFORM` constructor, the form becomes an UPDATE form for that record. This means that when the form is submitted the existing record is updated and no new record is inserted. If you set the argument `deletable=True`, the UPDATE form displays a "check to delete" checkbox. If checked, the record is deleted.

You can, for example, modify the controller of the previous example so that when we pass an additional integer argument in the URL path, as in:

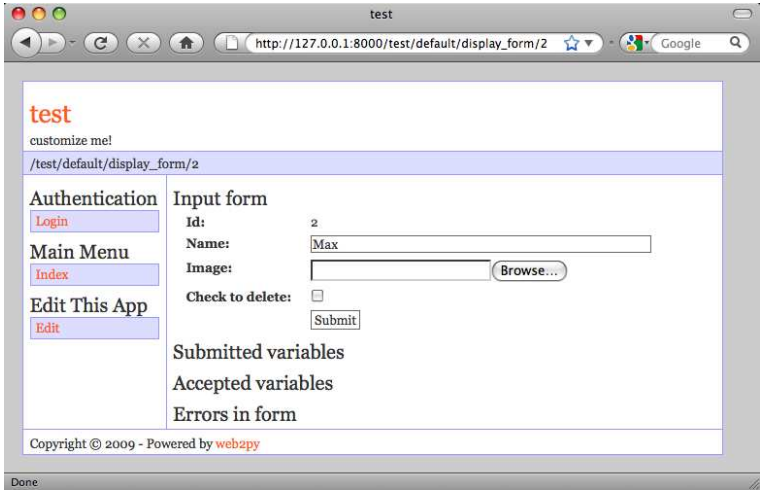
```
/test/default/display_form/2
```

and if there is a record with the corresponding id, the `SQLFORM` generates an UPDATE/DELETE form for the record:

```
def display_form():
   if len(request.args):
       records = db(db.person.id==request.args[0]).select()
   if len(request.args) and len(records):
       form = SQLFORM(db.person, records[0], deletable=True)
   else:
       form = SQLFORM(db.person)
   if form.accepts(request.vars, session):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   return dict(form=form)
```

Line 3 finds the record, line 5 makes an UPDATE/DELETE form, and line 7 makes an INSERT form. Line 8 does all the corresponding form processing.

Here is the final page:



By default `deletable=False`.

Edit forms also contain a hidden INPUT field with `name="id"` which is used to identify the record. This id is also stored server-side for additional security and, if the visitor tampers with the value of this field, the UPDATE is not performed and WEB2PY raises a SyntaxError, "user is tampering with form".

When a Field is marked with `writable=False`, the field is not shown in create forms, and it is is shown readonly in update forms. If a field is marked as `writable=False` and `readable=False`, then the field is not shown at all, not even in update forms.

Forms created with

```
form = SQLFORM(...,ignore_rw=True)
```

ignore the `readable` and `writable` attributes and always show all fields. Forms in `appadmin` ignore them by default.

Forms created with

```
form = SQLFORM(table,record_id,readonly=True)
```

always show all fields in readonly mode, and they cannot be accepted.

SQLFORM in HTML

There are times when you want to use `SQLFORM` to benefit from its form generation and processing, but you need a level of customization of the form in HTML that you cannot achieve with the parameters of the `SQLFORM` object, so you have to design the form using HTML.

Now, edit the previous controller and add a new action:

```
def display_manual_form():
   form = SQLFORM(db.person)
   if form.accepts(request.vars, formname='test'):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   else:
       response.flash = 'please fill the form'
   return dict()
```

and insert the form in the associated "default/display_manual_form.html" view:

```
{{extend 'layout.html'}}
<form>
<ul>
  <li>Your name is <input name="name" /></li>
</ul>
  <input type="submit" />
  <input type="hidden" name="_formname" value="test" />
</form>
```

Notice that the action does not return the form because it does not need to pass it to the view. The view contains a form created manually in HTML. The form contains a hidden field "_formname" that must be the same formname specified as an argument of `accepts` in the action. WEB2PY uses the form name in case there are multiple forms on the same page, to determine which one was submitted. If the page contains a single form, you can set `formname=None` and omit the hidden field in the view.

SQLFORM and uploads

Fields of type "upload" are special. They are rendered as INPUT fields of `type="file"`. Unless otherwise specified, the uploaded file is streamed in using a buffer, and stored under the "uploads" folder of the application using a new safe name, assigned automatically. The name of this file is then saved into the field of type uploads.

As an example, consider the following model:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('image', 'upload'))
```

You can use the same controller action "display_form" shown above.

When you insert a new record, the form allows you to browse for a file. Choose, for example, a jpg image. The file is uploaded and stored as:

```
applications/test/uploads/person.image.XXXXX.jpg
```

"XXXXXX" is a random identifier for the file assigned by WEB2PY.

Notice that, by default, the original filename of an uploaded file is b16encoded and used to build the new name for the file. This name is retrieved by the default "download" action and used to set the content disposition header to the original filename.

Only its extension is preserved. This is a security requirement since the filename may contain special characters that could allow a visitor to perform directory traversal attacks or other malicious operations.

The new filename is also stored in `form.vars.image_newfilename`.

When editing the record using an UPDATE form, it would be nice to display a link to the existing uploaded file, and WEB2PY provides a way to do it.

If you pass a URL to the `SQLFORM` constructor via the upload argument, WEB2PY uses the action at that URL to download the file. Consider the following actions:

```
def display_form():
   if len(request.args):
       records = db(db.person.id==request.args[0]).select()
   if len(request.args) and len(records):
       url = URL(r=request, f='download')
       form = SQLFORM(db.person, records[0], deletable=True, upload=
           url)
   else:
       form = SQLFORM(db.person)
   if form.accepts(request.vars, session):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   return dict(form=form)

def download():
    return response.download(request, db)
```

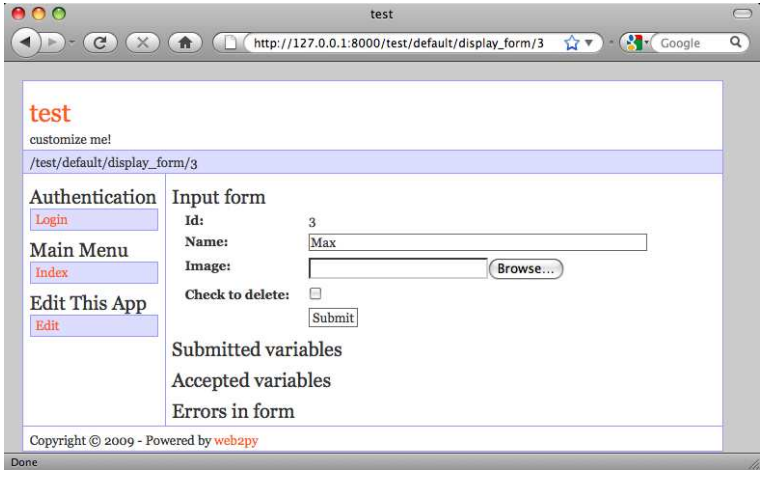
Now, insert a new record at the URL:

```
http://127.0.0.1:8000/test/default/display_form
```

Upload an image, submit the form, and then edit the newly created record by visiting:

```
http://127.0.0.1:8000/test/default/display_form/3
```

(here we assume the latest record has id=3). The form looks like the following:



This form, when serialized, generates the following HTML:

```
<td><label id="person_image__label" for="person_image">Image: </label
    ></td><td><div><input type="file" id="person_image" class="upload
    " name="image" />[<a href="/test/default/download/person.image
    .0246683463831.jpg">file</a>|<input type="checkbox" name="
    image__delete" />delete]</div></td><td></td></tr><tr id="
    delete_record__row"><td><label id="delete_record__label" for="
    delete_record">Check to delete:</label></td><td><input type="
    checkbox" id="delete_record" class="delete" name="
    delete_this_record" /></td>
```

which contains a link to allow downloading of the uploaded file, and a checkbox to remove the file from the database record, thus storing NULL in the "image" field.

Why is this mechanism exposed? Why do you need to write the download function? Because you may want to enforce some authorization mechanism in the download function. See Chapter 8 for an example.

Storing the original filename

WEB2PY automatically stores the original filename inside the new UUID filename and retrieves it when the file is downloaded. Upon download, the original filename is stored in the content-disposition header of the HTTP response. This is all done transparently without the need for programming.

Occasionally you may want to store the original filename in a database field. In this case, you need to modify the model and add a field to store it in:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('image_filename'),
    Field('image', 'upload'))
```

then you need to modify the controller to handle it:

```
def display_form():
    if len(request.args):
        records = db(db.person.id==request.args[0]).select()
    if len(request.args) and len(records):
        url = URL(r=request, f='download')
        form = SQLFORM(db.person, records[0], deletable=True,
                       upload=url, fields=['name', 'image'])
    else:
        form = SQLFORM(db.person, fields=['name', 'image'])
    if request.vars.image:
        form.vars.image_filename = request.vars.image.filename
    if form.accepts(request.vars, session):
        response.flash = 'form accepted'
    elif form.errors:
        response.flash = 'form has errors'
    return dict(form=form)
```

Notice that the `SQLFORM` does not display the "image_filename" field. The "display_form" action moves the filename of the `request.vars.image` into the `form.vars.image_filename`, so that it gets processed by `accepts` and stored in the database. The download function, before serving the file, checks in the database for the original filename and uses it in the content-disposition header.

Removing the action file

The `SQLFORM`, upon deleting a record, does not delete the physical uploaded file(s) referenced by the record. The reason is that WEB2PY does not know whether the same file is used/linked by other tables or used for other purpose. If you know it is safe to delete the actual file when the corresponding record is deleted, you can do the following:

```
db.define_table('image',
    Field('name'),
    Field('file','upload',autodelete=True))
```

The `autodelete` attribute is `False` by default. When set to `True` is makes sure the file is deleted when the record is deleted.

Links to referencing records

Now consider the case of two tables linked by a reference field. For example:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()))
db.define_table('dog',
    Field('owner', db.person),
```

```
    Field('name', requires=IS_NOT_EMPTY()))
db.dog.owner.requires = IS_IN_DB(db,db.person.id,'%(name)s')
```

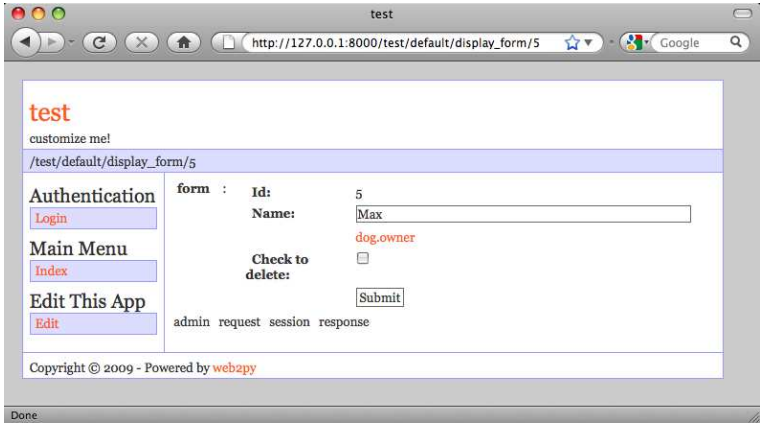
A person has dogs, and each dog belongs to an owner, which is a person. The dog owner is required to reference a valid `db.person.id` by `'%(name)s'`.

Let's use the **appadmin** interface for this application to add a few persons and their dogs.

When editing an existing person, the **appadmin** UPDATE form shows a link to a page that lists the dogs that belong to the person. This behavior can be replicated using the `linkto` argument of the `SQLFORM`. `linkto` has to point to the URL of a new action that receives a query string from the `SQLFORM` and lists the corresponding records. Here is an example:

```
def display_form():
   if len(request.args):
       records = db(db.person.id==request.args[0]).select()
   if len(request.args) and len(records):
       url = URL(r=request, f='download')
       link = URL(r=request, f='list_records')
       form = SQLFORM(db.person, records[0], deletable=True,
                      upload=url, linkto=link)
   else:
       form = SQLFORM(db.person)
   if form.accepts(request.vars, session):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   return dict(form=form)
```

Here is the page:



There is a link called "dog.owner". The name of this link can be changed via the `labels` argument of the `SQLFORM`, for example:

```
labels = {'dog.owner':"This person's dogs"}
```

If you click on the link you get directed to:

```
/test/default/list_records/dog?query=dog.owner%3D5
```

"list_records" is the specified action, with `request.args[0]` set to the name of the referencing table and `request.vars.query` set to the SQL query string. The query string in the URL contains the value "dog.owner=5" appropriately url-encoded (WEB2PY decodes this automatically when the URL is parsed).

You can easily implement a very general "list_records" action as follows:

```
def list_records():
    table = request.args[0]
    query = request.vars.query
    records = db(query).select(db[table].ALL)
    return dict(records=records)
```

with the associated "default/list_records.html" view:

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

When a set of records is returned by a select and serialized in a view, it is first converted into a SQLTABLE object (not the same as a Table) and then serialized into an HTML table, where each field corresponds to a table column.

Prepopulating the form

It is always possible to prepopulate a form using the syntax:

```
form.vars.name = 'fieldvalue'
```

Statements like the one above must be inserted after the form declaration and before the form is accepted, whether or not the field ("name" in the example) is explicitly visualized in the form.

SQLFORM without database IO

There are times when you want to generate a form from a database table using `SQLFORM` and you want to validate a submitted form accordingly, but you do not want any automatic INSERT/UPDATE/DELETE in the database. This is the case, for example, when one of the fields needs to be computed from the value of other input fields. This is also the case when you need to perform additional validation on the inserted data that cannot be achieved via standard validators.

This can be done easily by breaking:

```
form = SQLFORM(db.person)
if form.accepts(request.vars, session):
    response.flash = 'record inserted'
```

into:

```
form = SQLFORM(db.person)
if form.accepts(request.vars, session, dbio=False):
    ### deal with uploads explicitly
    form.vars.id = db.person.insert(**dict(form.vars))
    response.flash = 'record inserted'
```

The same can be done for UPDATE/DELETE forms by breaking:

```
form = SQLFORM(db.person,record)
if form.accepts(request.vars, session):
    response.flash = 'record updated'
```

into:

```
form = SQLFORM(db.person,record)
if form.accepts(request.vars, session, dbio=False):
    if form.vars.get('delete_this_record', False):
        db(db.person.id==record.id).delete()
    else:
        record.update_record(**dict(form.vars))
    response.flash = 'record updated'
```

In both cases web2py deals with the storage and renaming of the uploaded file as if `dbio=True`, the defaul scenario. The uploaded filename is in:

```
form.vars['%s_newfilename' % fieldname]
```

For more details, refer to the source code in "gluon/sqlhtml.py".

7.3 SQLFORM.factory

There are cases when you want to generate forms *as if* you had a database table but you do not want the database table. You simply want to take advantage of the `SQLFORM` capability to generate a nice looking CSS-friendly form and perhaps perform file upload and renaming.

This can be done via a `form_factory`. Here is an example where you generate the form, perform validation, upload a file and store everything in the `session` :

```
def form_from_factory()
    form = SQLFORM.factory(
        Field('your_name', requires=IS_NOT_EMPTY()),
        Field('your_image'))
    if form.accepts(request.vars, session):
        response.flash = 'form accepted'
        session.your_name = form.vars.your_name
```

```
        session.filename = form.vars.your_image
    elif form.errors:
        response.flash = 'form has errors'
    return dict(form=form)
```

Here is the "default/form_from_factory.html" view:

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

You need to use an underscore instead of a space for field labels, or explicitly pass a dictionary of `labels` to `form_factory`, as you would for a `SQLFORM`.

7.4 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 how to require a validator for a table field:

```
db.define_table('person', Field('name'))
db.person.name.requires = IS_NOT_EMPTY()
```

Validators are always assigned using the `requires` attribute of a field. A field can have a single validator or multiple validators. Multiple validators are made part of a list:

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

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

Built-in validators have constructors that take the optional argument `error_message`, which allows you to override the default error message.

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

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

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

Basic Validators

IS_ALPHANUMERIC This validator checks that a field value contains only characters in the ranges a-z, A-Z, or 0-9.

```
requires = IS_ALPHANUMERIC(error_message=T('must be alphanumeric!'))
```

IS_DATE This validator checks that a field value contains a valid date in the specified format. It is good practice to specify the format using the translation operator, in order to support different formats in different locales.

```
requires = IS_DATE(format=T('%Y-%m-%d'),
                   error_message=T('must be YYYY-MM-DD!'))
```

For the full description on % directives look under the IS_DATETIME validator.

IS_DATETIME This validator checks that a field value contains a valid datetime in the specified format. It is good practice to specify the format using the translation operator, in order to support different formats in different locales.

```
requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),
                       error_message=T('must be YYYY-MM-DD HH:MM:SS!'
                           ))
```

The following symbols can be used for the format string:

```
%a  Locale's abbreviated weekday name.
%A  Locale's full weekday name.
%b  Locale's abbreviated month name.
%B  Locale's full month name.
%c  Locale's appropriate date and time representation.
%d  Day of the month as a decimal number [01,31].
%H  Hour (24-hour clock) as a decimal number [00,23].
%I  Hour (12-hour clock) as a decimal number [01,12].
%j  Day of the year as a decimal number [001,366].
%m  Month as a decimal number [01,12].
%M  Minute as a decimal number [00,59].
%p  Locale's equivalent of either AM or PM.
%S  Second as a decimal number [00,61].
%U  Week number of the year (Sunday as the first day of the week)
        as a decimal number [00,53]. All days in a new year preceding
        the first Sunday are considered to be in week 0.
%w  Weekday as a decimal number [0(Sunday),6].
%W  Week number of the year (Monday as the first day of the week)
        as a decimal number [00,53]. All days in a new year preceding
        the first Monday are considered to be in week 0.
%x  Locale's appropriate date representation.
%X  Locale's appropriate time representation.
%y  Year without century as a decimal number [00,99].
%Y  Year with century as a decimal number.
%Z  Time zone name (no characters if no time zone exists).
%%  A literal "%" character.
```

IS_EMAIL It checks that the field value looks like an email address. It does not try to send email to confirm.

```
requires = IS_EMAIL(error_message=T('invalid email!'))
```

IS_EXPR Its first argument is a string containing a logical expression in terms of a variable value. It validates a field value if the expression evaluates to `True`. For example:

```
requires = IS_EXPR('int(value)%3==0',
                   error_message=T('not divisible by 3'))
```

One should first check that the value is an integer so that an exception will not occur.

```
requires = [IS_INT_IN_RANGE(0, 100), IS_EXPR('value%3==0')]
```

IS_FLOAT_IN_RANGE Checks that the field value is a floating point number within a definite range, $0 \leq value < 100$ in the following example:

```
requires = IS_FLOAT_IN_RANGE(0, 100,
         error_message=T('too small or too large!'))
```

IS_INT_IN_RANGE Checks that the field value is an integer number within a definite range, $0 \leq value < 100$ in the following example:

```
requires = IS_INT_IN_RANGE(0, 100,
         error_message=T('too small or too large!'))
```

IS_IN_SET Checks that the field values are in a set:

```
requires = IS_IN_SET(['a', 'b', 'c'],
         error_message=T('must be a or b or c'))
```

The elements of the set must always be strings unless this validator is preceded by `IS_INT_IN_RANGE` (which converts the value to int) or `IS_FLOAT_IN_RANGE` (which converts the value to float). For example:

```
requires = [IS_INT_IN_RANGE(0, 8), IS_IN_SET([2, 3, 5, 7],
          error_message=T('must be prime and less than 10'))]
```

IS_IN_SET and Tagging The `IS_IN_SET` validator has an optional attribute `multiple=False`. If set to True, multiple values can be stored in a field. The field in this case must be a string field. The multiple values are stored separated by a "|".

`multiple` references are handled automatically in create and update forms, but they are transparent to the DAL. We strongly suggest using the jQuery multiselect plugin to render multiple fields.

IS_LENGTH Checks if length of field's value fits between given boundaries. Works for both text and file inputs.

Its arguments are:

- maxsize: the maximum allowed length / size
- minsize: the minimum allowed length / size

Examples: Check if text string is shorter than 33 characters:

```
INPUT(_type='text', _name='name', requires=IS_LENGTH(32))
```

Check if password string is longer than 5 characters:

```
INPUT(_type='password', _name='name', requires=IS_LENGTH(minsize=6))
```

Check if uploaded file has size between 1KB and 1MB:

```
INPUT(_type='file', _name='name', requires=IS_LENGTH(1048576, 1024))
```

For all field types except for files, it checks the length of the value. In the case of files, the value is a `cookie.FieldStorage`, so it validates the length of the data in the file, which is the behavior one might intuitively expect.

IS_LIST_OF This is not properly a validator. Its intended use is to allow validations of fields that return multiple values. It is used in those rare cases when a form contains multiple fields with the same name or a multiple selection box. Its only argument is another validator, and all it does is to apply the other validator to each element of the list. For example, the following expression checks that every item in a list is an integer in the range 0-10:

```
requires = IS_LIST_OF(IS_INT_IN_RANGE(0, 10))
```

It never returns an error and does not contain an error message. The inner validator controls the error generation.

IS_LOWER This validator never returns an error. It just converts the value to lower case.

```
requires = IS_LOWER()
```

IS_MATCH This validator matches the value against a regular expression and returns an error if it does not match. Here is an example of usage to validate a US zip code:

```
requires = IS_MATCH('^\d{5}(-\d{4})?$',
         error_message='not a zip code')
```

Here is an example of usage to validate an IPv4 address:

```
requires = IS_MATCH('^\d{1,3}(\.\d{1,3}){3}$',
         error_message='not an IP address')
```

Here is an example of usage to validate a US phone number:

```
requires = IS_MATCH('^1?((-)\d{3}-?|\(\d{3}\))\d{3}-?\d{4}$',
         error_message='not a phone number')
```

For more information on Python regular expressions, refer to the official Python documentation.

IS_NOT_EMPTY This validator checks that the content of the field value is not an empty string.

```
requires = IS_NOT_EMPTY(error_message='cannot be empty!')
```

IS_TIME This validator checks that a field value contains a valid time in the specified format.

```
requires = IS_TIME(error_message=T('must be HH:MM:SS!'))
```

IS_URL Rejects a URL string if any of the following is true:

- The string is empty or None
- The string uses characters that are not allowed in a URL
- The string breaks any of the HTTP syntactic rules
- The URL scheme specified (if one is specified) is not 'http' or 'https'
- The top-level domain (if a host name is specified) does not exist

(These rules are based on RFC 2616[61])

This function only checks the URL's syntax. It does not check that the URL points to a real document, for example, or that it otherwise makes semantic sense. This function does automatically prepend 'http://' in front of a URL in the case of an abbreviated URL (e.g. 'google.ca').

If the parameter mode='generic' is used, then this function's behavior changes. It then rejects a URL string if any of the following is true:

- The string is empty or None
- The string uses characters that are not allowed in a URL
- The URL scheme specified (if one is specified) is not valid

(These rules are based on RFC 2396[62])

The list of allowed schemes is customizable with the allowed_schemes parameter. If you exclude None from the list, then abbreviated URLs (lacking a scheme such as 'http') will be rejected.

The default prepended scheme is customizable with the prepend_scheme parameter. If you set prepend_scheme to None, then prepending will be disabled. URLs that require prepending to parse will still be accepted, but the return value will not be modified.

IS_URL is compatible with the Internationalized Domain Name (IDN) standard specified in RFC 3490[63]). As a result, URLs can be regular strings or unicode strings. If the URL's domain component (e.g. google.ca) contains non-US-ASCII letters, then the domain will be converted into Punycode (defined in RFC 3492[64]). IS_URL goes a bit beyond the standards, and allows non-US-ASCII characters to be present in the path and query components of the URL as well. These non-US-ASCII characters will be encoded. For example, space will be encoded as'%20'. The unicode character with hex code 0x4e86 will become '%4e%86'.

Examples:

```
requires = IS_URL())
requires = IS_URL(mode='generic')
requires = IS_URL(allowed_schemes=['https'])
requires = IS_URL(prepend_scheme='https')
requires = IS_URL(mode='generic', allowed_schemes=['ftps', 'https'],
    prepend_scheme='https')
```

IS_STRONG Enforces complexity requirements on a field (usually a password field)

Example:

```
requires = IS_STRONG(min=10, special=2, upper=2)
```

where

- min is minimum length of the value
- special is the minimum number of required special characters
- is the minimum number of upper case characters

IS_IMAGE This validator checks if file uploaded through file input was saved in one of selected image formats and has dimensions (width and height) within given limits.

It does not check for maximum file size (use IS_LENGTH for that). It returns a validation failure if no data was uploaded. It supports the file formats BMP, GIF, JPEG, PNG, and it does not requires the Python Imaging Library.

Code parts taken from http://mail.python.org/pipermail/python-list/2007-June/617126.html

It takes the following arguments:

- extensions: iterable containing allowed image file extensions in lowercase ('jpg' extension of uploaded file counts as 'jpeg')
- maxsize: iterable containing maximum width and height of the image
- minsize: iterable containing minimum width and height of the image

Use (-1, -1) as minsize to bypass the image-size check.
Here are some Examples:

- Check if uploaded file is in any of supported image formats:

```
requires = IS_IMAGE()
```

- Check if uploaded file is either JPEG or PNG:

```
requires = IS_IMAGE(extensions=('jpeg', 'png'))
```

- Check if uploaded file is PNG with maximum size of 200x200 pixels:

```
requires = IS_IMAGE(extensions=('png'), maxsize=(200, 200))
```

IS_UPLOAD_FILENAME This validator checks if name and extension of file uploaded through file input matches given criteria.

It does not ensure the file type in any way. Returns validation failure if no data was uploaded.

Its arguments are:

- filename: filename (before dot) regex
- extension: extension (after dot) regex
- lastdot: which dot should be used as a filename / extension separator: `True` means last dot, e.g., file.png -> file / png `False` means first dot, e.g., file.tar.gz -> file / tar.gz
- case: 0 - keep the case, 1 - transform the string into lowercase (default), 2 - transform the string into uppercase

If there is no dot present, extension checks will be done against empty string and filename checks against whole value.

Examples:

Check if file has a pdf extension (case insensitive):

```
requires = IS_UPLOAD_FILENAME(extension='pdf')
```

Check if file has a tar.gz extension and name starting with backup:

```
requires = IS_UPLOAD_FILENAME(filename='backup.*', extension='tar.gz'
    , lastdot=False)
```

Check if file has no extension and name matching README (case sensitive):

```
requires = IS_UPLOAD_FILENAME(filename='^README$', extension='^$',
    case=0)
```

IS_IPV4 This validator checks if a field's value is an IP version 4 address in decimal form. Can be set to force addresses from a certain range.

IPv4 regex taken from: `http://regexlib.com/REDetails.aspx?regexp_id=1411`

Its arguments are

- minip: lowest allowed address; accepts: str, e.g., 192.168.0.1; iterable of numbers, e.g., [192, 168, 0, 1]; int, e.g., 3232235521
- maxip: highest allowed address; same as above

All three example values are equal, since addresses are converted to integers for inclusion check with following function:

```
number = 16777216 * IP[0] + 65536 * IP[1] + 256 * IP[2] + IP[3]
```

Examples:

Check for valid IPv4 address:

```
requires = IS_IPV4()
```

Check for valid private network IPv4 address:

```
requires = IS_IPV4(minip='192.168.0.1', maxip='192.168.255.255')
```

IS_LOWER This validator never returns an error. It converts the value to lower case.

IS_UPPER This validator never returns an error. It converts the value to upper case.

```
requires = IS_UPPER()
```

IS_NULL_OR Sometimes you need to allow empty values on a field along with other requirements. For example a field may be a date but it can also be empty. The `IS_NULL_OR` validator allows this:

```
requires = IS_NULL_OR(IS_DATE())
```

CLEANUP This is a filter. It never fails. It just removes all characters whose decimal ASCII codes are not in the list [10, 13, 32-127].

```
requires = CLEANUP()
```

CRYPT This is also a filter. It performs a secure hash on the input and it is used to prevent passwords from being passed in the clear to the database.

```
requires = CRYPT(key=None)
```

If the key is None, it uses the MD5 algorithm. If a key is specified it uses the HMAC+SHA512 with the provided key. The key has to be a unique string associated to the database used. The key can never be changed. If you lose the key the previously hashed values become useless.

Database Validators

IS_NOT_IN_DB Consider the following example:

```
db.define_table('person', Field('name'))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

It requires that when you insert a new person, his/her name is not already in the database, `db`, in the field `person.name`. As with all other validators this requirement is enforced at the form processing level, not at the database level. This means that there is a small probability that, if two visitors try to concurrently insert records with the same person.name, this results in a race condition and both records are accepted. It is therefore safer to also inform the database that this field should have a unique value:

```
db.define_table('person', Field('name', unique=True))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

Now if a race condition occurs, the database raises an OperationalError and one of the two inserts is rejected.

The first argument of `IS_NOT_IN_DB` can be a database connection or a DAL SSet. In the latter case, you would be checking only the set defined by the Set.

The following code, for example, does not allow registration of two persons with the same name within 10 days of each other:

```
import datetime
now = datetime.datetime.today()
db.define_table('person',
    Field('name'),
    Field('registration_stamp', 'datetime', default=now))
recent = db(db.person.registration_stamp>now-datetime.timedelta(10))
db.person.name.requires = IS_NOT_IN_DB(recent, 'person.name')
```

IS_IN_DB Consider the following tables and requirement:

```
db.define_table('person', Field('name', unique=True))
db.define_table('dog', Field('name'), Field('owner', db.person)
db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s')
```

It is enforced at the level of dog INSERT/UPDATE/DELETE forms. It requires that a `dog.owner` be a valid id in the field `person.id` in the database `db`. Because of this validator, the `dog.owner` field is represented as a dropbox. The third argument of the validator is a string that describes the elements in the dropbox. In the example you want to see the person `%(name)s` instead of the person `%(id)s`. `%(...)s` is replaced by the value of the field in brackets for each record.

If you want the field validated, but you do not want a dropbox, you must put the validator in a list.

```
db.dog.owner.requires = [IS_IN_DB(db, 'person.id', '%(name)s')]
```

The first argument of the validator can be a database connection or a DAL Set, as in `IS_NOT_IN_DB`.

IS_IN_DB and Tagging The `IS_IN_DB` validator has an optional attribute `multiple=False`. If set to true multiple values can be stored in a field. The field in this case cannot be a reference but it must be a string field. The multiple values are stored separated by a "|".

`multiple` references are handled automatically in create and update forms, but they are transparent to the DAL. We strongly suggest using the jQuery multiselect plugin to render multiple fields.

Custom Validators

All validators follow the prototype below:

```
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` is converted to `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.

The validator may also contain a `formatter` method. It must perform the opposite conversion to the one the `__call__` does. For example, consider the source code for `IS_DATE`:

```
class IS_DATE(object):
    def __init__(self, format='%Y-%m-%d', error_message='must be YYYY
        -MM-DD!'):
        self.format = format
        self.error_message = error_message
    def __call__(self, value):
        try:
            y, m, d, hh, mm, ss, t0, t1, t2 = time.strptime(value,
                str(self.format))
            value = datetime.date(y, m, d)
            return (value, None)
        except:
            return (value, self.error_message)
    def formatter(self, value):
        return value.strftime(str(self.format))
```

On success, the `__call__` method reads a date string from the form and converts it into a datetime.date object using the format string specified in the constructor. The `formatter` object takes a datetime.date object and converts it to a string representation using the same format. The `formatter` is called automatically in forms, but you can also call it explicitly to convert objects into their proper representation. For example:

```
>>> db = DAL()
>>> db.define_table('atable',
       Field('birth', 'date', requires=IS_DATE('%m/%d/%Y')))
>>> id = db.atable.insert(birth=datetime.date(2008, 1, 1))
>>> rows = db(db.atable.id==id).select()
>>> print db.atable.formatter(rows[0].birth)
01/01/2008
```

When multiple validators are required (and stored in a list), they are executed in order and the output of one is passed as input to the next. The chain breaks when one of the validators fails.

Conversely, when we call the `formatter` method of a field, the formatters of the associated validators are also chained, but in reverse order.

Validators with Dependencies

Occasionally, you need to validate a field and the validator depends on the value of another field. This can be done, but it requires setting the validator in the controller, when the value of the other field is known. For example, here is a page that generates a registration form that asks for username and password twice. None of the fields can be empty, and both passwords must match:

```
def index():
    match_it = IS_EXPR('value==%s' % repr(request.vars.password),
                       error_message='passwords do not match')
    form = SQLFORM.factory(
```

```
        Field('username', requires=IS_NOT_EMPTY()),
        Field('password', requires=IS_NOT_EMPTY()),
        Field('password_again', requires=match_it))
    if form.accepts(request.vars, session):
        pass # or take some action
    return dict(form=form)
```

The same mechanism can be applied to FORM and SQLFORM objects.

7.5 Widgets

Here is a list of available WEB2PY widgets:

```
SQLFORM.widgets.string.widget
SQLFORM.widgets.text.widget
SQLFORM.widgets.password.widget
SQLFORM.widgets.integer.widget
SQLFORM.widgets.double.widget
SQLFORM.widgets.time.widget
SQLFORM.widgets.date.widget
SQLFORM.widgets.datetime.widget
SQLFORM.widgets.upload.widget
SQLFORM.widgets.boolean.widget
SQLFORM.widgets.options.widget
SQLFORM.widgets.multiple.widget
SQLFORM.widgets.radio.widget
SQLFORM.widgets.checkboxes.widget
```

The first ten of them are the defaults for the corresponding field types. The "options" widget is used when a field's requires is `IS_IN_SET` or `IS_IN_DB` with `multiple=False` (default behavior). The "multiple" widget is used when a field's requires is `IS_IN_SET` or `IS_IN_DB` with `multiple=True`. The "radio" and "checkboxes" widgets are never used by default, but can be set manually.

For example, to have a "string" field represented by a textarea:

```
Field('comment', 'string', widget=SQLFORM.widgets.text.widget)
```

You can create new widgets or extend existing widgets; in fact, `SQLFORM.widgets[type]` is a class and `SQLFORM.widgets[type].widget` is a static member function of the corresponding class. Each widget function takes two arguments: the field object, and the current value of that field. It returns a representation of the widget. As an example, the string widget could be recoded as follows:

```
def my_string_widget(field, value):
    return INPUT(_name=field.name,
                 _id="%s_%s\" % (field._tablename, field.name),
                 _class=field.type,
                 _value=value,
                 requires=field.requires)
```

```

Field('comment', 'string', widget=my_string_widget)
```

The id and class values must follow the convention described later in this chapter. A widget may contain its own validators, but it is good practice to associate the validators to the "requires" attribute of the field and have the widget get them from there.

7.6 CRUD

One of the recent additions to WEB2PY is the Create/Read/Update/Delete (CRUD) API on top of SQLFORM. CRUD creates an SQLFORM, but it simplifies the coding because it incorporates the creation of the form, the processing of the form, the notification, and the redirection, all in one single function. What that function is depends on what you want to do.

The first thing to notice is that CRUD differs from the other WEB2PY APIs we have used so far because it is not already exposed. It must be imported. It also must be linked to a specific database. For example:

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

The first argument of the constructor is the current context `globals()` so that CRUD can access the local request, response, and session. The second argument is a database connection object, `db`.

The `crud` object defined above provides the following API:

.

- `crud.tables()` returns a list of tables defined in the database.
- `crud.create(db.tablename)` returns a create form for table tablename.
- `crud.read(db.tablename, id)` returns a readonly form for tablename and record id.
- `crud.update(db.tablename, id)` returns an update form for tablename and record id.
- `crud.delete(db.tablename, id)` deletes the record.
- `crud.select(db.tablename, query)` returns a list of records selected from the table.
- `crud()` returns one of the above based on the `request.args()`.

For example, the following action:

```
def data: return dict(form=crud())
```

would expose the following URLs:

```
http://.../[app]/[controller]/data/tables
http://.../[app]/[controller]/data/create/[tablename]
http://.../[app]/[controller]/data/read/[tablename]/[id]
http://.../[app]/[controller]/data/delete/[tablename]
http://.../[app]/[controller]/data/select/[tablename]
```

However, the following action:

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

would only expose the create method

```
http://.../[app]/[controller]/create_tablename
```

While the following action:

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

would only expose the update method

```
http://.../[app]/[controller]/update_tablename
```

and so on.

The behavior of CRUD can be customized in two ways: by setting some attributes of the `crud` object or by passing extra parameters to each of its methods.

Attributes

Here is a complete list of current CRUD attributes, their default values, and meaning:

```
crud.settings.create_next = request.url
```

specifies the URL to redirect to after a successful "create" record.

```
crud.settings.update_next = request.url
```

specifies the URL to redirect to after a successful "update" record.

```
crud.settings.delete_next = request.url
```

specifies the URL to redirect to after a successful "delete" record.

```
crud.settings.download_url = URL(r=request, f='download')
```

specifies the URL to be used for linking uploaded files.

```
crud.settings.create_onvalidation = lambda form: None
```

is an optional function to be called `onvalidation` of "create" forms (see SQLFORM onvalidation)

```
crud.settings.update_onvalidation = lambda form: None
```

is an optional function to be called `onvalidation` of "update" forms (see SQLFORM onvalidation)

```
crud.settings.create_onaccept = lambda form: None
```

is an optional function to be called before redirect after successful "create" record. This function takes the form as its only argument.

```
crud.settings.update_onaccept = lambda form: None
```

is an optional function to be called before redirect after successful "update" record. This function takes the form as its only argument.

```
crud.settings.update_ondelete = lambda form: None
```

is an optional function to be called before redirect after successfully deleting a record using an "update" form. This function takes the form as its only argument.

```
crud.settings.delete_onaccept = lambda record: None
```

is an optional function to be called before redirect after successfully deleting a record using the "delete" method. This function takes the form as its only argument.

```
crud.settings.update_deletable = True
```

determines whether the "update" forms should have a "delete" button.

```
crud.settings.showid = False
```

determines whether the "update" forms should show the id of the edited record.

```
crud.settings.keepvalues = False
```

determines whether forms should keep the previously inserted values or reset to default after successful submission.

Messages

Here is a list of customizable messages:

```
crud.messages.submit_button = 'Submit'
```

sets the text of the "submit" button for both create and update forms.

```
crud.messages.delete_label = 'Check to delete:'
```

sets the label of the "delete" button in "update" forms.

```
crud.messages.record_created = 'Record Created'
```

sets the flash message on successful record creation.

```
crud.messages.record_updated = 'Record Updated'
```

sets the flash message on successful record update.

```
crud.messages.record_deleted = 'Record Deleted'
```

sets the flash message on successful record deletion.

```
crud.messages.update_log = 'Record %(id)s updated'
```

sets the log message on successful record update.

```
crud.messages.create_log = 'Record %(id)s created'
```

sets the log message on successful record creation.

```
crud.messages.read_log = 'Record %(id)s read'
```

sets the log message on successful record read access.

```
crud.messages.delete_log = 'Record %(id)s deleted'
```

sets the log message on successful record deletion.

Notice that `crud.messages` belongs to the class `gluon.storage.Message` which is similar to `gluon.storage.Storage` but it automatically translates its values, without need for the `T` operator.

Log messages are used if and only if CRUD is connected to Auth as discussed in Chapter 8. The events are logged in the Auth table "auth_events".

Methods

The behavior of CRUD methods can also be customized on a per call basis. Here are their signatures:

```
crud.tables()
crud.create(table, next, onvalidation, onaccept, log, message)
crud.read(table, record)
crud.update(table, record, next, onvalidation, onaccept, ondelete,
    log, message, deletable)
crud.delete(table, record_id, next, message)
crud.select(table, query, fields, orderby, limitby, headers, **attr)
```

- `table` is a DAL table or a tablename the method should act on.
- `record` and `record_id` are the id of the record the method should act on.
- `next` is the URL to redirect to after success. If the URL contains the substring "[id]" this will be replaced by the id of the record currently created/updated.

- `onvalidation` has the same function as SQLFORM(..., onvalidation)
- `onaccept` is a function to be called after the form submission is accepted and acted upon, but before redirection.
- `log` is the log message. Log messages in CRUD see variables in the `form.vars` dictionary such as "%(id)s".
- `message` is the flash message upon form acceptance.
- `ondelete` is called in place of `onaccept` when a record is deleted via an "update" form.
- `deletable` determines whether the "update" form should have a delete option.
- `query` is the query to be used to select records.
- `fields` is a list of fields to be selected.
- `orderby` determines the order in which records should be selected (see Chapter 6).
- `limitby` determines the range of selected records that should be displayed (see Chapter 6).
- `headers` is a dictionary with the table header names.

Here is an example of usage in a single controller function:

```
# assuming db.define_table('person', Field('name'))
def people():
    form = crud.create(db.person, next=request.url,
           message=T("record created"))
    persons = crud.select(db.person, fields=['name'],
           headers={'person.name', 'Name'})
    return dict(form=form, persons=persons)
```

7.7 Custom form

If a form is created with SQLFORM, SQLFORM.factory or CRUD, there are multiple ways it can be embedded in a view allowing multiple degrees of customization. Consider for example the following model:

```
db.define_table('image',
    Field('name'),
    Field('file', 'upload'))
```

and upload action

```
def upload_image():
    return dict(form=crud.create(db.image))
```

The simplest way to embed the form in the view for `upload_image` is

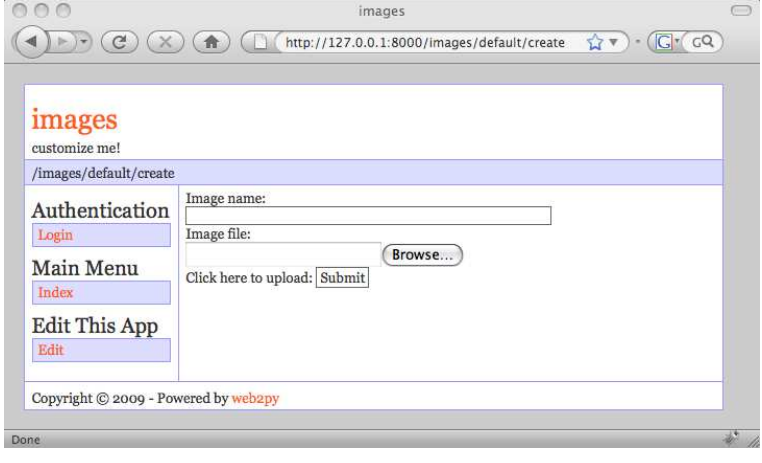
```
{{=form}}
```

This results in a standard table layout. If you wish to use a different layout, you can break the form into components

```
{{=form.custom.begin}}
Image name: <div>{{=form.custom.widget.name}}</div>
Image file: <div>{{=form.custom.widget.file}}</div>
Click here to upload: {{=form.custom.submit}}
{{=form.custom.end}}
```

Where `form.custom.widget[fieldname]` gets serialized into the proper widget for the field. If the form is submitted and it contains errors, they are appended below the widgets, as usual.

The above sample form is show in the image below.



If you do not wish to use the widgets serialized by WEB2PY, you can replace them with HTML. There are some variables that will be useful for this:

- `form.custom.labels[fieldname]` contains the label for the field.
- `form.custom.dspval[fieldname]` form-type and field-type dependent display representation of the field.
- `form.custom.inpval[fieldname]` form-type and field-type dependent values to be used in field code.

It is important to follow the conventions described below.

CSS Conventions

Tags in forms generated by SQLFORM, SQLFORM.factory and CRUD follow a strict CSS naming convention that can be used to further customize the forms.

Given a table "mytable", a field "myfield" of type "string", it is rendered by default by a

```
SQLFORM.widgets.string.widget
```

that looks like this:

```
<input type="text" name="myfield" id="mytable_myfield"
       class="string" />
```

Notice that:

- the class of the INPUT tag is the same as the type of the field. This is very important for the jQuery code in "web2py_ajax.html" to work. It makes sure that you can only have numbers in "integer" and "double" fields, and that "time", "date" and "datetime" fields display the popup calendar.
- the id is the name of the class plus the name of the field, joined by one underscore. This allows you to uniquely refer to the field via `jQuery('#mytable_myfield')` and manipulate, for example, the stylesheet of the field or bind actions associated to the field events (focus, blur, keyup, etc.).
- the name is, as you would expect, the field name.

Switch off errors

Occasionally, you may want to disable the automatic error placement and display form error messages in some place other than the default. That can be done in two steps:

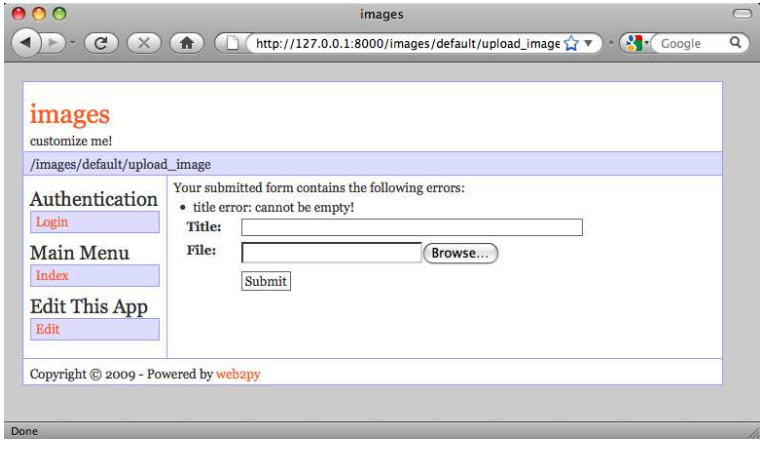
- display the error messages where desired
- `form.error.clear()` before the form is rendered so that error messages are not displayed in the default locations.

Here is an example where the errors are displayed above the form and not in the form.

```
{{if form.errors:}}
  Your submitted form contains the following errors:
```

```
  <ul>
  {{for fieldname in form.errors:}}
    <li>{{=fieldname}} error: {{=form.errors[fieldname]}}</li>
  {{pass}}
  </ul>
  {{form.errors.clear()}}
{{pass}}
{{=form}}
```

The errors will displayed as in the image shown below.



CHAPTER 8

ACCESS CONTROL

WEB2PY includes a powerful and customizable Role-Based Access Control (RBAC) mechanism.

Here is a definition from Wikipedia:

"Role-Based Access Control (RBAC) is an approach to restricting system access to authorized users. It is a newer alternative approach to mandatory access control (MAC) and discretionary access control (DAC). RBAC is sometimes referred to as role-based security.

RBAC is a policy neutral and flexible access control technology sufficiently powerful to simulate DAC and MAC. Conversely, MAC can simulate RBAC if the role graph is restricted to a tree rather than a partially ordered set.

Prior to the development of RBAC, MAC and DAC were considered to be the only known models for access control: if a model was not MAC, it was considered to be a DAC model, and vice versa. Research in the late 1990s demonstrated that RBAC falls in neither category.

Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members of staff (or other system users) are assigned particular roles, and through

those role assignments acquire the permissions to perform particular system functions. Unlike context-based access control (CBAC), RBAC does not look at the message context (such as a connection's source).

Since users are not assigned permissions directly, but only acquire them through their role (or roles), management of individual user rights becomes a matter of simply assigning appropriate roles to the user; this simplifies common operations, such as adding a user, or changing a user's department.

RBAC differs from access control lists (ACLs) used in traditional discretionary access control systems in that it assigns permissions to specific operations with meaning in the organization, rather than to low level data objects. For example, an access control list could be used to grant or deny write access to a particular system file, but it would not dictate how that file could be changed."

The WEB2PY class that implements RBAC is called **Auth**.

Auth needs (and defines) the following tables:

- `auth_user` stores users' name, email address, password, and status (registration pending, accepted, blocked)
- `auth_group` stores groups or roles for users in a many-to-many structure. By default, each user is in its own group, but a user can be in multiple groups, and each group can contain multiple users. A group is identified by a role and a description.
- `auth_membership` links users and groups in a many-to-many structure.
- `auth_permission` links groups and permissions. A permission is identified by a name and, optionally, a table and a record. For example, members of a certain group can have "update" permissions on a specific record of a specific table.
- `auth_event` logs changes in the other tables and successful access via CRUD to objects controlled by the RBAC.

In principle, there is no restriction on the names of the roles and the names of the permissions; the developer can create them to fix the roles and permissions in the organization. Once they have been created, WEB2PY provides an API to check if a user is logged in, if a user is a member of a given group, and/or if the user is a member of any group that has a given required permission.

WEB2PY also provides decorators to restrict access to any function based on login, membership and permissions.

WEB2PY also understands some specific permissions, i.e., those that have a name that correspond to the CRUD methods (create, read, update, delete) and can enforce them automatically without the need to use decorators.

In this chapter, we are going to discuss different parts of RBAC one by one.

8.1 Authentication

In order to use RBAC, users need to be identified. This means that they need to register (or be registered) and log in.

Auth provides multiple login methods. The default one consists of identifying users based on the local `auth_user` table. Alternatively, it can log in users against third-party basic authentication systems (for example a Twitter account), SMTP servers (for example Gmail), or LDAP (your corporate account). It can also use third-party single-sign-on systems, for example Google. This is achieved via plugins, and new plugins are added all the time.

To start using `Auth`, you need at least this code in a model file, which is also provided with the WEB2PY "welcome" application and assumes a `db` connection object:

```
from gluon.tools import Auth
auth = Auth(globals(), db)
auth.define_tables()
```

To expose **Auth**, you also need the following function in a controller (for example in "default.py"):

```
def user(): return dict(form=auth())
```

> *The* `auth` *object and the* `user` *action are already defined in the scaffolding application.*

WEB2PY also includes a sample view "default/user.html" to render this function properly that looks like this:

```
{{extend 'layout.html'}}
<h2>{{=request.args(0)}}</h2>
{{=form}}
{{if request.args(0)=='login':}}
<a href="{{=URL(r=request, args='register')}}" >register</a><br />
<a href="{{=URL(r=request, args='retrieve_password')}}" >lost
    password</a><br />
{{pass}}
```

The controller above exposes multiple actions:

```
http://.../[app]/default/user/register
http://.../[app]/default/user/login
http://.../[app]/default/user/logout
http://.../[app]/default/user/profile
http://.../[app]/default/user/change_password
```

```
http://.../[app]/default/user/verify_email
http://.../[app]/default/user/retrieve_username
http://.../[app]/default/user/retrieve_password
http://.../[app]/default/user/impersonate
http://.../[app]/default/user/groups
http://.../[app]/default/user/not_authorized
```

- **register** allows users to register. It is integrated with CAPTCHA, although this is disabled by default.
- **login** allows users who are registered to log in (if the registration is verified or does not require verification, if it has been approved or does not require approval, and if it has not been blocked).
- **logout** does what you would expect but also, as the other methods, logs the event and can be used to trigger some event.
- **profile** allows users to edit their profile, i.e. the content of the `auth_user` table. Notice that this table does not have a fixed structure and can be customized.
- **change_password** allows users to change their password in a fail-safe way.
- **verify_email**. If email verification is turned on, then visitors, upon registration, receive an email with a link to verify their email information. The link points to this action.
- **retrieve_username**. By default, **Auth** uses email and password for login, but it can, optionally, use username instead of email. In this latter case, if a user forgets his/her username, the `retrieve_username` method allows the user to type the email address and retrieve the username by email.
- **retrieve_password**. Allows users who forgot their password to receive a new one by email. The name here can be misleading because this function does not retrieve the current password (that would be impossible since the password is only stored encrypted/hashed) but generates a new one.
- **impersonate** allows a user to "impersonate" another user. This is important for debugging and for support purposes. `request.args[0]` is the id of the user to be impersonated. This is only allowed if the logged in user `has_permission('impersonate', db.auth_user, user_id)`.
- **groups** lists the groups the current logged in user is a member of.

- **not_authorized** displays an error message when the visitor tried to do something that he/she is not authorized to do.

Logout, profile, change_password, impersonate, and groups require login.

By default they are all exposed, but it is possible to restrict access to only some of these actions.

All of the methods above can be extended or replaced by subclassing **Auth**.

To restrict access to functions to only logged in visitors, decorate the function as in the following example

```
@auth.requires_login()
def hello():
    return dict(message='hello logged in visitor')
```

Any function can be decorated, not just exposed actions. Of course this is still only a very simple example of access control. More complex examples will be discussed later.

Email verification

By default, email verification is disabled. To enable email, append the following lines in the model where `auth` is defined:

```
from gluon.tools import Mail
mail = Mail(globals())
mail.settings.server = 'smtp.example.com:25'
mail.settings.sender = 'you@example.com'
mail.settings.login = 'username:password'
auth.settings.mailer = mail
auth.settings.registration_requires_verification = False
auth.messages.verify_email_subject = 'Email verification'
auth.messages.verify_email = \
  'Click on the link http://...verify_email/%(key)s to verify your
      email'
```

You need to replace the mail.settings with the proper parameters for your SMTP server. Set `mail.settings.login=False` if the SMTP server does not require authentication.

You also need to replace the string

```
'Click on the link ...'
```

in `auth.messages.verify_email` with the proper complete URL of the action `verify_email`. This is necessary because WEB2PY may be installed behind a proxy, and it cannot determine its own public URLs with absolute certainty.

Once `mail` is defined, it can also be used to send email explicitly via

```
mail.send(to=['somebody@example.com'],
          subject='hello', message='hi there')
```

Restrictions on registration

If you want to allow visitors to register but not to log in until registration has been approved by the administrator:

```
auth.settings.registration_requires_approval = True
```

You can approve a registration via the appadmin interface. Look into the table `auth_user`. Pending registrations have a `registration_key` field set to "pending". A registration is approved when this field is set to blank.

Via the appadmin interface, you can also block a user from logging in. Locate the user in the table `auth_user` and set the `registration_key` to "blocked". "blocked" users are not allowed to log in. Notice that this will prevent a visitor from logging in but it will not force a visitor who is already logged in to log out.

You can also block access to the "register" page completely with this statement:

```
auth.settings.actions_disabled.append('register')
```

Other methods of **Auth** can be restricted in the same way.

CAPTCHA and reCAPTCHA

To prevent spammers and bots registering on your site, you may require a registration CAPTCHA. WEB2PY supports reCAPTCHA [65] out of the box. This is because reCAPTCHA is very well designed, free, accessible (it can read the words to the visitors), easy to set up, and does not require installing any third-party libraries.

This is what you need to do to use reCAPTCHA:

- Register with reCAPTCHA [65] and obtain a (PUBLIC_KEY, PRIVATE_KEY) couple for your account. These are just two strings.
- Append the following code to your model after the `auth` object is defined:

```
from gluon.tools import Recaptcha
auth.settings.captcha = Recaptcha(request,
    'PUBLIC_KEY', 'PRIVATE_KEY')
```

reCAPTCHA may not work if you access the web site as 'localhost' or '127.0.0.1', because it is registered to work with publicly visible web sites only.

The `Recaptcha` constructor takes some optional arguments:

```
Recaptcha(..., use_ssl=True, error_message='invalid')
```

Notice that `use_ssl=False` by default.

If you do not want to use reCAPTCHA, look into the definition of the `Recaptcha` class in "gluon/tools.py", since it is easy to use other CAPTCHA systems.

Customizing Auth

The call to

```
auth.define_tables()
```

defines all **Auth** tables that have not been defined already. This means that if you wish to do so, you can define your own `auth_user` table. Using a similar syntax to the one show below, you can customize any other **Auth** table.

Here is the proper way to define a user table:

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

auth_table = db.define_table(
    auth.settings.table_user_name,
    Field('first_name', length=128, default=''),
    Field('last_name', length=128, default=''),
    Field('email', length=128, default='', unique=True),
    Field('password', 'password', length=256,
          readable=False, label='Password'),
    Field('registration_key', length=128, default= '',
          writable=False, readable=False))

auth_table.first_name.requires = \
  IS_NOT_EMPTY(error_message=auth.messages.is_empty)
auth_table.last_name.requires = \
  IS_NOT_EMPTY(error_message=auth.messages.is_empty)
auth_table.password.requires = [IS_STRONG(), CRYPT()]
auth_table.email.requires = [
  IS_EMAIL(error_message=auth.messages.invalid_email),
  IS_NOT_IN_DB(db, auth_table.email)]
auth.settings.table_user = auth_table

# before
# auth.define_tables()
```

You can add any field you wish, but you cannot remove the required fields shown in this example.

It is important to make "password" and "registration_key" fields `readable=False` and make the "registration_key" field `writable=False`, since a visitor must not be allowed to tamper with them.

If you add a field called "username", it will be used in place of "email" for login. If you do, you will need to add a validator as well:

```
auth_table.username.requires = IS_NOT_IN_DB(db, auth_table.username)
```

Renaming Auth tables

The actual names of the `Auth` tables are stored in

```
auth.settings.table_user_name = 'auth_user'
auth.settings.table_group_name = 'auth_group'
auth.settings.table_membership_name = 'auth_membership'
auth.settings.table_permission_name = 'auth_permission'
auth.settings.table_event_name = 'auth_event'
```

The names of the table can be changed by reassigning the above variables after the `auth` object is defined and before the Auth tables are defined. For example:

```
auth = Auth(globals(),db)
auth.settings.table_user_name = 'person'
#...
auth.define_tables()
```

The actual tables can also be referenced, independently of their actual names, by

```
auth.settings.table_user
auth.settings.table_group
auth.settings.table_membership
auth.settings.table_permission
auth.settings.table_event
```

Alternate Login Methods

Auth provides multiple login methods and hooks to create new login methods. Each supported login method corresponds to a file in the folder

```
gluon/contrib/login_methods/
```

Refer to the documentation in the files themselves for each login method, but here we provide some examples.

First of all we need to make a distinction between two types of alternate login methods:

- login methods that use a WEB2PY form (although the credentials are verified outside WEB2PY). An example is LDAP.
- login methods that require an external sign-on (WEB2PY never gets to see the credentials).

Let's consider examples of the first case:

Basic Let's say you have an authentication service, for example at the url `https://basic.example.com`, that accepts basic access authentication. That means the server accepts HTTP requests with a header of the form:

```
GET /index.html HTTP/1.0
Host: basic.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

where the latter string is the base64 encoding of the string username:password. The service responds 200 OK if the user is authorized and 400, 401, 402, 403 or 404 otherwise.

You want to enter username and password using the standard `Auth` login form and verify the credentials against such a service. All you need to do is add the following code to your application

```
from gluon.contrib.login_methods.basic_auth import basic_auth
auth.settings.login_methods.append(
    basic_auth('https://basic.example.com'))
```

Notice that `auth.settings.login_methods` is a list of authentication methods that are executed sequentially. By default it is set to

```
auth.settings.login_methods = [auth]
```

When an alternate method is appended, for example `basic_auth`, **Auth** first tries to log in the visitor based on the content of `auth_user`, and when this fails, it tries the next method in the list. If a method succeeds in logging in the visitor, and if `auth.settings.login_methods[0]==auth`, `Auth` takes the following actions:

- if the user does not exist in `auth_user`, a new user is created and the username/email and passwords are stored.
- if the user does exist in `auth_user` but the new accepted password does not match the old stored password, the old password is replaced with the new one (notice that passwords are always stored hashed unless specified otherwise).

If you do not wish to store the new password in `auth_user`, then it is sufficient to change the order of login methods, or remove `auth` from the list. For example:

```
from gluon.contrib.login_methods.basic_auth import basic_auth
auth.settings.login_methods = \
    [basic_auth('https://basic.example.com')]
```

The same applies for any other login method described here.

SMTP and Gmail You can verify the login credentials using a remote SMTP server, for example Gmail; i.e., you log the user in if the email and password they provide are valid credentials to access the Gmail SMTP server (`smtp.gmail.com:587`). All that is needed is the following code:

```
from gluon.contrib.login_methods.email_auth import email_auth
auth.settings.login_methods.append(
    email_auth("smtp.gmail.com:587", "@gmail.com"))
```

The first argument of `email_auth` is the address:port of the SMTP server. The second argument is the email domain.

This works with any SMTP server that requires TLS authentication.

LDAP Authentication using LDAP works very much as in the previous cases.

To use LDAP login with MS Active Directory:

```
from gluon.contrib.login_methods.ldap_auth import ldap_auth
auth.settings.login_methods.append(ldap_auth(mode='ad',
   server='my.domain.controller',
   base_dn='ou=Users,dc=domain,dc=com'))
```

To use LDAP login with Lotus Notes and Domino:

```
auth.settings.login_methods.append(ldap_auth(mode='domino',
   server='my.domino.server'))
```

To use LDAP login with OpenLDAP (with UID):

```
auth.settings.login_methods.append(ldap_auth(server='my.ldap.server',
   base_dn='ou=Users,dc=domain,dc=com'))
```

To use LDAP login with OpenLDAP (with CN):

```
auth.settings.login_methods.append(ldap_auth(mode='cn',
   server='my.ldap.server', base_dn='ou=Users,dc=domain,dc=com'))
```

Google on GAE Authentication using Google when running on Google App Engine requires skipping the WEB2PY login form, being redirected to the Google login page, and back upon success. Because the behavior is different than in the previous examples, the API is a little different.

```
from gluon.contrib.login_methods.gae_google_login import
    GaeGoogleAccount
auth.settings.login_form = GaeGoogleAccount()
```

8.2 Authorization

Once a new user is registered, a new group is created to contain the user. The role of the new user is conventionally "user_[id]" where [id] is the id of the newly created id. The creation of the group can be disabled with

```
auth.settings.create_user_groups = False
```

although we do not suggest doing so.

Users have membership in groups. Each group is identified by a name/role. Groups have permissions. Users have permissions because of the groups they belong to.

You can create groups, give membership and permissions via **appadmin** or programmatically using the following methods:

```
auth.add_group('role', 'description')
```

returns the id of the newly created group.

```
auth.del_group(group_id)
```

deletes the group with `group_id`.

```
auth.del_group(auth.id_group('user_7'))
```

deletes the group with role "user_7", i.e., the group uniquely associated to user number 7.

```
auth.user_group(user_id)
```

returns the id of the group uniquely associated to the user identified by `user_id`.

```
auth.add_membership(group_id, user_id)
```

gives `user_id` membership of the group `group_id`. If the `user_id` is not specified, then WEB2PY assumes the current logged-in user.

```
auth.del_membership(group_id, user_id)
```

revokes `user_id` membership of the group `group_id`. If the `user_id` is not specified, then WEB2PY assumes the current logged-in user.

```
auth.has_membership(group_id, user_id)
```

checks whether `user_id` has membership of the group `group_id`. If the `user_id` is not specified, then WEB2PY assumes the current logged-in user.

```
auth.add_permission(group_id, 'name', 'object', record_id)
```

gives permission "name" (user defined) on the object "object" (also user defined) to members of the group `group_id`. If "object" is a tablename then the permission can refer to the entire table (`record_id==0`) or to a specific record (`record_id>0`). When giving permissions on tables, it is common to use a permission name in the set ('create', 'read', 'update', 'delete', 'select') since these permissions are understood and can be enforced by CRUD.

```
auth.del_permission(group_id, 'name', 'object', record_id)
```

revokes the permission.

```
auth.has_permission('name', 'object', record_id, user_id)
```

checks whether the user identified by `user_id` has membership in a group with the requested permission.

```
rows = db(accessible_query('read', db.sometable, user_id))\
    .select(db.mytable.ALL)
```

returns all rows of table "sometable" that user `user_id` has "read" permission on. If the `user_id` is not specified, then WEB2PY assumes the current logged-in user. The `accessible_query(...)` can be combined with other queries to make more complex ones. `accessible_query(...)` is the only **Auth** method to require a JOIN, so it does not work on the Google App Engine.

Assuming the following definitions:

```
>>> from gluon.tools import Auth
>>> auth = Auth(globals(), db)
>>> auth.define_tables()
>>> secrets = db.define_table('document', Field('body'))
>>> james_bond = db.auth_user.insert(first_name='James',
                                     last_name='Bond')
```

Here is an example:

```
>>> doc_id = db.document.insert(body = 'top secret')
>>> agents = auth.add_group(role = 'Secret Agent')
>>> auth.add_membership(agents, james_bond)
>>> auth.add_permission(agents, 'read', secrets)
>>> print auth.has_permission('read', secrets, doc_id, james_bond)
True
>>> print auth.has_permission('update', secrets, doc_id, james_bond)
False
```

Decorators

The most common way to check permission is not by explicit calls to the above methods, but by decorating functions so that permissions are checked relative to the logged-in visitor. Here are some examples:

```
def function_one():
    return 'this is a public function'

@auth.requires_login()
def function_two():
    return 'this requires login'

@auth.requires_membership('agents')
def function_three():
```

```
    return 'you are a secret agent'

@auth.requires_permission('read', secrets)
def function_four():
    return 'you can read secret documents'

@auth.requires_permission('delete', 'any file')
def function_five():
    import os
    for file in os.listdir('./'):
        os.unlink(file)
    return 'all files deleted'

@auth.requires_permission('add', 'number')
def add(a, b):
    return a + b

def function_six():
    return add(3, 4)
```

Note that access to all functions apart from the first one is restricted based on permissions that the visitor may or may not have.

If the visitor is not logged in, then the permission cannot be checked; the visitor is redirected to the login page and then back to the page that requires permissions.

If the visitor does not have permission to access a given function, the visitor is redirect to the URL defined by

```
auth.settings.on_failed_authorization = \
    URL(r=request, f='user/on_failed_authorization')
```

You can change this variable and redirect the user elsewhere.

Combining requirements

Occasionally, it is necessary to combine requirements. This can be done via a generic `requires` decorator which takes a single argument, a true or false condition. For example, to give access to agents, but only on Tuesday:

```
@auth.requires(auth.has_membership(agents) \
               and request.now.weekday()==1)
def function_seven():
    return 'Hello agent, it must be Tuesday!'
```

Authorization and CRUD

Using decorators and/or explicit checks provides one way to implement access control.

Another way to implement access control is to always use CRUD (as opposed to SQLFORM) to access the database and to ask CRUD to enforce access control on database tables and records. This is done by linking `Auth` and CRUD with the following statement:

```
crud.settings.auth = auth
```

This will prevent the visitor from accessing any of the CRUD functions unless the visitor is logged in and has explicit access. For example, to allow a visitor to post comments, but only update their own comments (assuming crud, auth and db.comment are defined):

```
def give_create_permission(form):
    group_id = auth.id_group('user_%s' % auth.user.id)
    auth.add_permission(group_id, 'read', db.comment)
    auth.add_permission(group_id, 'create', db.comment)
    auth.add_permission(group_id, 'select', db.comment)

def give_update_permission(form):
    comment_id = form.vars.id
    group_id = auth.id_group('user_%s' % auth.user.id)
    auth.add_permission(group_id, 'update', db.comment, comment_id)
    auth.add_permission(group_id, 'delete', db.comment, comment_id)

auth.settings.register_onaccept = give_create_permission
crud.settings.auth = auth

def post_comment():
   form = crud.create(db.comment, onaccept=give_update_permission)
   comments = db(db.comment.id>0).select()
   return dict(form=form, comments=comments)

def update_comment():
   form = crud.update(db.comment, request.args(0))
   return dict(form=form)
```

You can also select specific records (those you have 'read' access to):

```
def post_comment():
   form = crud.create(db.comment, onaccept=give_update_permission)
   query = auth.accessible_query('read', db.comment, auth.user.id)
   comments = db(query).select(db.comment.ALL)
   return dict(form=form, comments=comments)
```

Authorization and Downloads

The use of decorators and the use of `crud.settings.auth` do not enforce authorization on files downloaded by the usual download function

```
def download(): return response.download(request, db)
```

If one wishes to do so, one must declare explicitly which "upload" fields contain files that need access control upon download. For example:

```
db.define_table('dog',
   Field('small_image', 'upload')
   Field('large_image', 'upload'))

db.dog.large_image.authorization = lambda record: \
   auth.is_logged_in() and \
   auth.has_permission('read', db.dog, record.id, auth.user.id)
```

The attribute `authorization` of upload field can be None (the default) or a function that decides whether the user is logged in and has permission to 'read' the current record. In this example, there is no restriction on downloading images linked by the "small_image" field, but we require access control on images linked by the "large_image" field.

Access control and Basic authentication

Occasionally, it may be necessary to expose actions that have decorators that require access control as services; i.e., to call them from a program or script and still be able to use authentication to check for authorization.

Auth enables login via basic authentication:

```
auth.settings.allow_basic_authentication = True
```

With this set, an action like

```
@auth.requires_login()
def give_me_time():
    import time
    return time.ctime()
```

can be called, for example, from a shell command:

```
wget --user=[username] --password=[password]
    http://.../[app]/[controller]/give_me_time
```

Basic login is often the only option for services (described in the next chapter), but it is disabled by default.

Settings and Messages

Here is a list of all parameters that can be customized for **Auth**

```
auth.settings.actions_disabled = []
```

The actions that should be disabled, for example ['register'].

```
auth.settings.registration_requires_verification = False
```

Set to `True` so that registrants receive a verification email and are required to click a link to complete registration.

```
auth.settings.registration_requires_approval = False
```

Set to `True` to prevent login of newly registered users until they are approved (this is done by setting `registration_key==''` via appadmin or programmatically).

```
auth.settings.create_user_groups = True
```

Set to False if you do not want to automatically create a group for each newly registered user.

```
auth.settings.login_url = URL(r=request, f='user', args='login')
```

Tells WEB2PY the URL of the login page

```
auth.settings.logged_url = URL(r=request, f='user', args='profile')
```

If the user tried to access the register page but is already logged in, he is redirected to this URL.

```
auth.settings.download_url = URL(r=request, f='download')
```

Tells WEB2PY the URL to download uploaded documents. It is necessary to create the profile page in case it contains uploaded files, such as the user image.

```
auth.settings.mailer = None
```

Must point to an object with a send method with the same signature as `gluon.tools.Mail.send`.

```
auth.settings.captcha = None
```

Must point to an object with signature similar to `gluon.tools.Recaptcha`.

```
auth.settings.expiration = 3600  # seconds
```

The expiration time of a login session, in seconds.

```
auth.settings.on_failed_authorization = \
    URL(r=request,f='user/on_failed_authorization')
```

The URL to redirect to after a failed authorization.

```
auth.settings.password_field = 'password'
```

The name of the password field as stored in the db. The only reason to change this is when 'password' is a reserved keyword for the db and so cannot be used as a field name. This is the case, for example, for FireBird.

```
auth.settings.showid = False
```

Determines whether the profile page should show the id of the user.

```
auth.settings.login_next = URL(r=request, f='index')
```

By default, the login page, after successful login, redirects the visitor to the referrer page (if and only if the referrer required login). If there is no referrer, it redirects the visitor to the page pointed to by this variable.

```
auth.settings.login_onvalidation = None
```

Function to be called after login validation, but before actual login. The function must take a single argument, the form object.

```
auth.settings.login_onaccept = None
```

Function to be called after login, but before redirection. The function must take a single argument, the form object.

```
auth.settings.login_methods = [auth]
```

Determines alternative login methods, as discussed previously.

```
auth.settings.login_form = auth
```

Sets an alternative login form for single sign-on as discussed previously.

```
auth.settings.allows_basic_auth = False
```

If set to True allows calling actions that have access control enforced through decorators using basic access authentication.

```
auth.settings.logout_next = URL(r=request, f='index')
```

The URL redirected to after logout.

```
auth.settings.register_next = URL(r=request, f='user', args='login')
```

The URL redirected to after registration.

```
auth.settings.register_onvalidation = None
```

Function to be called after registration form validation, but before actual registration, and before any email verification email is sent. The function must take a single argument, the form object.

```
auth.settings.register_onaccept = None
```

Function to be called after registration, but before redirection. The function must take a single argument, the form object.

```
auth.settings.verify_email_next = \
    URL(r=request, f='user', args='login')
```

The URL to redirect a visitor to after email address verification.

```
auth.settings.verify_email_onaccept = None
```

Function to be called after completed email verification, but before redirection. The function must take a single argument, the form object.

```
auth.settings.profile_next = URL(r=request, f='index')
```

The URL to redirect visitors to after they edit their profile.

```
auth.settings.retrieve_username_next = URL(r=request, f='index')
```

The URL to redirect visitors to after they request to retrieve their username.

```
auth.settings.retrieve_password_next = URL(r=request, f='index')
```

The URL to redirect visitors to after they request to retrieve their password.

```
auth.settings.change_password_next = URL(r=request, f='index')
```

The URL to redirect visitors to after they request a new password by email.

You can also customize the following messages whose use and context should be obvious:

```
auth.messages.submit_button = 'Submit'
auth.messages.verify_password = 'Verify Password'
auth.messages.delete_label = 'Check to delete:'
auth.messages.function_disabled = 'Function disabled'
auth.messages.access_denied = 'Insufficient privileges'
auth.messages.registration_verifying = 'Registration needs
    verification'
auth.messages.registration_pending = 'Registration is pending
    approval'
auth.messages.login_disabled = 'Login disabled by administrator'
auth.messages.logged_in = 'Logged in'
auth.messages.email_sent = 'Email sent'
auth.messages.unable_to_send_email = 'Unable to send email'
auth.messages.email_verified = 'Email verified'
auth.messages.logged_out = 'Logged out'
auth.messages.registration_successful = 'Registration successful'
auth.messages.invalid_email = 'Invalid email'
auth.messages.invalid_login = 'Invalid login'
auth.messages.invalid_user = 'Invalid user'
auth.messages.is_empty = "Cannot be empty"
auth.messages.mismatched_password = "Password fields don't match"
auth.messages.verify_email = ...
auth.messages.verify_email_subject = 'Password verify'
auth.messages.username_sent = 'Your username was emailed to you'
auth.messages.new_password_sent = ...
auth.messages.password_changed = 'Password changed'
auth.messages.retrieve_username = ...
auth.messages.retrieve_username_subject = 'Username retrieve'
auth.messages.retrieve_password = ...
auth.messages.retrieve_password_subject = 'Password retrieve'
auth.messages.profile_updated = 'Profile updated'
auth.messages.new_password = 'New password'
auth.messages.old_password = 'Old password'
auth.messages.register_log = 'User %(id)s Registered'
auth.messages.login_log = 'User %(id)s Logged-in'
auth.messages.logout_log = 'User %(id)s Logged-out'
auth.messages.profile_log = 'User %(id)s Profile updated'
auth.messages.verify_email_log = ...
auth.messages.retrieve_username_log = ...
auth.messages.retrieve_password_log = ...
auth.messages.change_password_log = ..
auth.messages.add_group_log = 'Group %(group_id)s created'
auth.messages.del_group_log = 'Group %(group_id)s deleted'
auth.messages.add_membership_log = None
auth.messages.del_membership_log = None
auth.messages.has_membership_log = None
auth.messages.add_permission_log = None
```

```
auth.messages.del_permission_log = None
auth.messages.has_permission_log = None
```

`add|del|has` membership logs allow the use of "%(user_id)s" and "%(group_id)s".

`add|del|has` permission logs allow the use of "%(user_id)s", "%(name)s", "%(table_name)s", and "%(record_id)s".

8.3 Central Authentication Service

WEB2PY provides support for authentication and authorization via appliances. Here we discuss the **cas** appliance for Central Authentication Service (CAS). Notice that at the time of writing CAS is distict and does not work with **Auth**. This will change in the future.

CAS is an open protocol for distributed authentication and it works in the following way: When a visitor arrives at our web site, our application check in the session if the user is already authenticated (for example via a `session.token` object). If the user is not authenticated, the controller redirects the visitor from the CAS appliance, where the user can log in, register, and manage his credentials (name, email and password). If the user registers, he receives an email, and registration is not complete until he responds to the email. Once the user has successfully registered and logged in, the CAS appliance redirects the user to our application together with a key. Our application uses the key to get the credentials of the user via an HTTP request in the background to the CAS server.

Using this mechanism, multiple applications can use the a single sign-on via a single CAS server. The server providing authentication is called a service provider. Applications seeking to authenticate visitors are called service consumers.

CAS is similar to OpenID, with one main difference. In the the case of OpenID, the visitor chooses the service provider. In the case of CAS, our application makes this choice, making CAS more secure.

You can run only the consumer, only the provider, or both (in a single or separate applications).

To run CAS as consumer you must download the file:

```
https://www.web2py.com/cas/static/cas.py
```

and store it as a model file called "cas.py". Then you must edit the controllers that need authentication (for example "default.py") and, at the top, add the following code:

```
CAS.login_url='https://www.web2py.com/cas/cas/login'
CAS.check_url='https://www.web2py.com/cas/cas/check'
```

```
3  CAS.logout_url='https://www.web2py.com/cas/cas/logout'
4  CAS.my_url='http://127.0.0.1:8000/myapp/default/login'
5
6  if not session.token and not request.function=='login':
7      redirect(URL(r=request,f='login'))
8  def login():
9      session.token=CAS.login(request)
10     id,email,name=session.token
11     return dict()
12 def logout():
13     session.token=None
14     CAS.logout()
```

You must edit the attributes of the CAS object above. By default, they point to the CAS provider that runs on "https://mdp.cti.depaul.edu". We provide this service mainly for testing purposes. The `CAS.my_url` has to be the full URL to the login action defined in your application and shown in the code. The CAS provider needs to redirect your browser to this action.

Our CAS provider returns a token containing a tuple (id, email, name), where id is the unique record id of the visitor (as assigned by the provider's database), email is the email address of the visitor (as declared by the visitor to the provider and verified by the provider), and name is the name of the visitor (it is chosen by the visitor and there is no guarantee this is a real name).

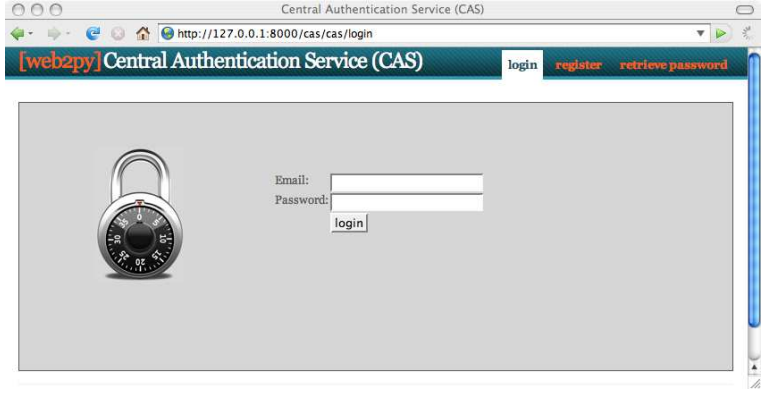
If you visit the local url:

```
1  /myapp/default/login
```

you get redirected to the CAS login page:

```
1  https://mdp.cti.depaul.edu/cas/cas/login
```

which looks like this:



You may also use third-party CAS services, but you may need to edit line 10 above, since different CAS providers may return tokens containing different values. Check the documentation of the CAS service you need to access for details. Most services only return (id, username).

After a successful login, you are redirected to the local login action. The view of the local login action is executed only after a successful CAS login.

You can download the CAS provider appliance from ref. [32] and run it yourself. If you choose to do so, you must also edit the first lines of the "email.py" model in the appliance, so that it points to your SMPT server.

You can also merge the files of the CAS provider appliance provider with those of your application (models under models, etc.) as long there is no filename conflict.

CHAPTER 9

SERVICES

The W3C defines a web service as "a software system designed to support interoperable machine-to-machine interaction over a network". This is a broad definition, and it encompass a large number of protocols not designed for machine-to-human communication, but for machine-to-machine communication such as XML, JSON, RSS, etc.

WEB2PY provides, out of the box, support for the many protocols, including XML, JSON, RSS, CSV, XMLRPC, JSONRPC, AMFRPC. WEB2PY can also be extended to support additional protocols.

Each of those protocols is supported in multiple ways, and we make a distinction between:

- Rendering the output of a function in a given format (for example XML, JSON, RSS, CSV)
- Remote Procedure Calls (for example XMLRPC, JSONRPC, AMFRPC)

9.1 Rendering a dictionary

HTML, XML, and JSON

Consider the following action:

```
def count():
    session.counter = (session.counter or 0) + 1
    return dict(counter=session.counter, now=request.now)
```

This action returns a counter that is increased by one when a visitor reloads the page, and the timestamp of the current page request.

Normally this page would be requested via:

```
http://127.0.0.1:8000/app/default/count
```

and rendered in HTML. Without writing one line of code, we can ask WEB2PY to render this page using a different protocols by adding an extension to the URL:

```
http://127.0.0.1:8000/app/default/count.html
http://127.0.0.1:8000/app/default/count.xml
http://127.0.0.1:8000/app/default/count.json
```

The dictionary returned by the action will be rendered in HTML, XML and JSON, respectively.

Here is the XML output:

```
<document>
   <counter>3</counter>
   <now>2009-08-01 13:00:00</now>
</document>
```

Here is the JSON output:

```
{ 'counter':3, 'now':'2009-08-01 13:00:00' }
```

Notice that date, time, and datetime objects are rendered as strings in ISO format. This is not part of the JSON standard but a WEB2PY convention.

How it works

When, for example, the ".xml" extension is called, WEB2PY looks for a template file called "default/count.xml", and if it does not find it, WEB2PY looks for a template called "generic.xml". The files "generic.html, "generic.xml", "generic.json" are provided with the current scaffolding application.

Other extensions can be easily defined by the user.

Nothing needs to be done to enable this in a WEB2PY app. To use it in an older WEB2PY app, you may need to copy the "generic.*" files from a later scaffolding app (after version 1.60).

Here is the code for "generic.html"

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

{{=BEAUTIFY(response._vars)}}

<button onclick="document.location='{{=URL("admin","default","design"
    ,args=request.application)}}'">admin</button>
<button onclick="jQuery('#request').slideToggle()">request</button>
<div class="hidden" id="request"><h2>request</h2>{{=BEAUTIFY(request)
    }}</div>
<button onclick="jQuery('#session').slideToggle()">session</button>
<div class="hidden" id="session"><h2>session</h2>{{=BEAUTIFY(session)
    }}</div>
<button onclick="jQuery('#response').slideToggle()">response</button>
<div class="hidden" id="response"><h2>response</h2>{{=BEAUTIFY(
    response)}}</div>
<script>jQuery('.hidden').hide();</script>
```

Here is the code for "generic.xml"

```
{{
try:
   from gluon.serializers import xml
   response.write(xml(response._vars),escape=False)
   response.headers['Content-Type']='text/xml'
except:
   raise HTTP(405,'no xml')
}}
```

And here is the code for "generic.json"

```
{{
try:
   from gluon.serializers import json
   response.write(json(response._vars),escape=False)
   response.headers['Content-Type']='text/json'
except:
   raise HTTP(405,'no json')
}}
```

Every dictionary can be rendered in HTML, XML and JSON as long as it only contains python primitive types (int, float, string, list, tuple, dictionary). `response._vars` contains the dictionary returned by the action.

If the dictionary contains other user-defined or WEB2PY-specific objects, they must be rendered by a custom view.

Rendering Rows

If you need to render a set of Rows as returned by a select in XML or JSON or another format, first transform the Rows object into a list of dictionaries using the `as_list()` method.

Consider for example the following mode:

```
db.define_table('person', Field('name'))
```

The following action can be rendered in HTML but not in XML or JSON:

```
def everybody():
    people = db().select(db.person.ALL)
    return dict(people=people)
```

While the following action can rendered in XML and JSON.

```
def everybody():
    people = db().select(db.person.ALL).as_list()
    return dict(people=people)
```

Custom Formats

If, for example, you want to render an action as a Python pickle:

```
http://127.0.0.1:8000/app/default/count.pickle
```

you just need to create a new view file "default/count.pickle" that contains:

```
{{
import cPickle
response.headers['Content-Type'] = 'application/python.pickle'
response.write(cPickle.dumps(response._vars),escape=False)
}}
```

If you want to be able to render as a picked file any action, you only need to save the above file with the name "generic.pickle".

Not all objects are pickleable, and not all pickled objects can be unpickled. It is safe to stick to primitive Python files and combinations of them. Objects that do not contain references to file streams or database connections are are usually pickleable, but they can only be unpickled in an environment where the classes of all pickled objects are already defined.

RSS

WEB2PY includes a "generic.rss" view that can render the dictionary returned by the action as an RSS feed.

Because the RSS feeds have a fixed structure (title, link, description, items, etc.) then for this to work, the dictionary returned by the action must have the proper structure:

```
{'title'      : '',
 'link'       : '',
 'description': '',
 'created_on' : '',
 'entries'    : []}
```

end each entry in entries must have the same similar structure:

```
{'title'      : '',
 'link'       : '',
 'description': '',
 'created_on' : ''}
```

For example the following action can be rendered as an RSS feed:

```
def feed():
    return dict(title="my feed",
                link="http://feed.example.com",
                description="my first feed",
                entries=[
                  dict(title="my feed",
                  link="http://feed.example.com",
                  description="my first feed")
                ])
```

by simply visiting the URL:

```
http://127.0.0.1:8000/app/default/feed.rss
```

Alternatively, assuming the following model:

```
db.define_table('rss_entry',
    Field('title'),
    Field('link'),
    Field('created_on','datetime'),
    Field('description'))
```

the following action can also be rendered as an RSS feed:

```
def feed():
    return dict(title=''my feed'',
                link=''http://feed.example.com'',
                description=''my first feed'',
                entries=db().select(db.rss_entry.ALL).as_list())
```

The `as_list()` method of a Rows object converts the rows into a list of dictionaries.

If additional dictionary items are found with key names not explicitly listed here, they are ignored.

Here is the "generic.rss" view provided by WEB2PY:

```
{{
try:
   from gluon.serializers import rss
   response.write(rss(response._vars),escape=False)
   response.headers['Content-Type']='application/rss+xml'
except:
   raise HTTP(405,'no rss')
}}
```

As one more example of an RSS application, we consider an RSS aggregator that collects data from the "slashdot" feed and returns a new WEB2PY feed.

```
def aggregator():
    import gluon.contrib.feedparser as feedparser
    d = feedparser.parse(
        "http://rss.slashdot.org/Slashdot/slashdot/to")
    return dict(title=d.channel.title,
                link = d.channel.link,
                description = d.channel.description,
                created_on = request.now,
                entries = [
                  dict(title = entry.title,
                  link = entry.link,
                  description = entry.description,
                  created_on = request.now) for entry in d.entries])
```

It can be accessed at:

```
http://127.0.0.1:8000/app/default/aggregator.rss
```

CSV

The Comma Separated Values (CSV) format is a protocol to represent tabular data.

Consider the following model:

```
db.define_model('animal',
    Field('species'),
    Field('genus'),
    Field('family'))
```

and the following action:

```
def animals():
    animals = db().select(db.animal.ALL)
    return dict(animals=animals)
```

WEB2PY does not provide a "generic.csv"; you must define a custom view "default/animals.csv" that serializes the animals into CSV. Here is a possible implementation:

```
{{
import cStringIO
stream=cStringIO.StringIO()
animals.export_to_csv_file(stream)
response.headers['Content-Type']='application/vnd.ms-excel'
response.write(stream.getvalue(), escape=False)
}}
```

Notice that for CSV one could also define a "generic.csv" file, but one would have to specify the name of the object to be serialized ("animals" in the example). This is why we do not provide a "generic.csv" file.

9.2 Remote Procedure Calls

WEB2PY provides a mechanism to turn any function into a web service. The mechanism described here differs from the mechanism described before because:

- The function may take arguments
- The function may be defined in a model or a module instead of controller
- You may want to specify in detail which RPC method should be supported
- It enforces a more strict URL naming convention
- It is smarter then the previous methods because it works for a fixed set of protocols. For the same reason it is not as easily extensible.

To use this feature:
First, you must import and instantiate a service object.

```
from gluon.tools import Service
service = Service(globals())
```

This is already done in the "db.py" model file in the scaffolding application.

Second, you must expose the service handler in the controller:

```
def call():
    session.forget()
    return service()
```

This already done in the "default.py" controller of the scaffolding application. Remove `session.forget()` *is you plan to use session cookies with the services.*

Third, you must decorate those functions you want to expose as a service. Here is a list of currently supported decorators:

```
@service.run
@service.xml
@service.json
@service.rss
@service.csv
@service.xmlrpc
@service.jsonrpc
@service.amfrpc3('domain')
```

As an example consider the following decorated function:

```
@service.run
def concat(a,b):
    return a+b
```

This function can be defined in a model or in a controller. This function can now be called remotely in two ways:

```
http://127.0.0.1:8000/app/default/call/run/concat?a=hello&b=world
http://127.0.0.1:8000/app/default/call/run/concat/hello/world
```

In both cases the http request returns:

```
helloworld
```

If the `@service.xml` decorator is used, the function can be called via

```
http://127.0.0.1:8000/app/default/call/xml/concat?a=hello&b=world
http://127.0.0.1:8000/app/default/call/xml/concat/hello/world
```

and the output is returned as XML:

```
<document>
   <result>helloworld</result>
</document>
```

It can serialize the output of the function even if this is a DAL Rows object. In this case, in fact, it will call `as_list()` automatically.

If the `@service.json` decorator is used, the function can be called via

```
http://127.0.0.1:8000/app/default/call/json/concat?a=hello&b=world
http://127.0.0.1:8000/app/default/call/json/concat/hello/world
```

and the output returned as JSON.

If the `@service.csv` decorator is used, the service handler requires, as return value, an iterable object of iterable objects, such as a list of lists. Here is an example:

```
@service.csv
def table1(a,b):
    return [[a,b],[1,2]]
```

This service can be called by visiting one of the following URLs:

```
http://127.0.0.1:8000/app/default/call/csv/table1?a=hello&b=world
http://127.0.0.1:8000/app/default/call/csv/table1/hello/world
```

and it returns:

```
hello,world
1,2
```

The `@service.rss` decorator expects a return value in the same format as the "generic.rss" view discussed in the previous section.

Multiple decorators are allowed for each function.

So far, everything discussed in this section is simply an alternative to the method described in the previous section. The real power of the service object comes with XMLRPC, JSONRPC and AMFRPC, as discussed below.

XMLRPC

Consider the following code, for example, in the "default.py" controller:

```
@service.xmlrpc
def add(a,b):
    return a+b

@service.xmlrpc
def div(a,b):
    return a+b
```

Now in a python shell you can do

```
>>> from xmlrpclib import ServerProxy
>>> server = ServerProxy(
       'http://127.0.0.1:8000/app/default/call/xmlrpc')
>>> print server.add(3,4)
7
>>> print server.add('hello','world')
'helloworld'
>>> print server.div(12,4)
3
>>> print server.div(1,0)
ZeroDivisionError: integer division or modulo by zero
```

The Python xmlrpclib module provides a client for the XMLRPC protocol. WEB2PY acts as the server.

The client connects to the server via ServerProxy and can remotely call decorated functions in the server. The data (a,b) is passed to the function(s), not via GET/POST variables, but properly encoded in the request body using the XMLPRC protocol, and thus it carries with itself type information (int or string or other). The same is true for the return value(s). Moreover, any exception that happens on the server propagates back to the client.

There are XMLRPC libraries for many programming languages (including C, C++, Java, C#, Ruby, and Perl), and they can interoperate with each other. This is one the best methods to create applications that talk to each other, independent of the programming language.

The XMLRPC client can also be implemented inside a WEB2PY action so that one action can talk to another WEB2PY application (even within the same installation) using XMLRPC. Beware of session deadlocks in this case. If an action calls via XMLRPC a function in the same app, the caller must release the session lock before the call:

```
session.forget()
session._unlock(response)
```

JSONRPC

JSONRPC is very similar to XMLRPC, but uses the JSON based protocol to encode the data instead of XML. As an example of application here, we discuss its usage with Pyjamas. Pyjamas is a Python port of the Google Web Toolkit (originally written in Java). Pyjamas allows to write a client application in Python. Pyjamas translates this code into JavaScript. WEB2PY serves the javascript and communicates with it via AJAX requests originating from the client and triggered by user actions.

Here we describe how to make Pyjamas work with WEB2PY. It does not require any additional libraries other than WEB2PY and Pyjamas.

We are going to build a simple "todo" application with a Pyjamas client (all JavaScript) that talks to the server exclusively via JSONRPC.

Here is how to do it:

First, create a new application called "todo".

Second, in "models/db.py", enter the following code:

```
db=SQLDB('sqlite://storage.sqlite')
db.define_table('todo', Field('task'))

from gluon.tools import Service     # import rpc services
service = Service(globals())
```

Third, in "controllers/default.py", enter the following code:

```
def index():
redirect(URL(r=request,f='todoApp'))

@service.jsonrpc
def getTasks():
    todos = db(db.todo.id>0).select()
    return [(todo.task,todo.id) for todo in todos]

@service.jsonrpc
def addTask(taskFromJson):
    db.todo.insert(task= taskFromJson)
    return getTasks()

@service.jsonrpc
def deleteTask (idFromJson):
    del db.todo[idFromJson]
    return getTasks()

def call():
    session.forget()
    return service()

def todoApp():
    return dict()
```

The purpose of each function should be obvious.

Fourth, in "views/default/todoApp.html", enter the following code:

```
<html>
  <head>
```

```
    <meta name="pygwt:module"
     content="{{=URL(r=request,c='static',f='output/todoapp')}}" />
    <title>
      simple todo application
    </title>
  </head>
  <body bgcolor="white">
    <h1>
      simple todo application
    </h1>
    <i>
      type a new task to insert in db,
      click on existing task to delete it
    </i>
    <script language="javascript"
     src="{{=URL(r=request,c='static',f='output/pygwt.js')}}">
    </script>
  </body>
</html>
```

This view just executes the Pyjamas code in "static/output/todoapp". Code that we have not yet created.

Fifth, in "static/TodoApp.py" (notice it is TodoApp, not todoApp!), enter the following client code:

```
from pyjamas.ui.RootPanel import RootPanel
from pyjamas.ui.Label import Label
from pyjamas.ui.VerticalPanel import VerticalPanel
from pyjamas.ui.TextBox import TextBox
import pyjamas.ui.KeyboardListener
from pyjamas.ui.ListBox import ListBox
from pyjamas.ui.HTML import HTML
from pyjamas.JSONService import JSONProxy

class TodoApp:
    def onModuleLoad(self):
        self.remote = DataService()
        panel = VerticalPanel()

        self.todoTextBox = TextBox()
        self.todoTextBox.addKeyboardListener(self)

        self.todoList = ListBox()
        self.todoList.setVisibleItemCount(7)
        self.todoList.setWidth("200px")
        self.todoList.addClickListener(self)
        self.Status = Label("")

        panel.add(Label("Add New Todo:"))
        panel.add(self.todoTextBox)
        panel.add(Label("Click to Remove:"))
        panel.add(self.todoList)
        panel.add(self.Status)
        self.remote.getTasks(self)

        RootPanel().add(panel)
```

```
    def onKeyUp(self, sender, keyCode, modifiers):
        pass

    def onKeyDown(self, sender, keyCode, modifiers):
        pass

    def onKeyPress(self, sender, keyCode, modifiers):
        """
        This function handles the onKeyPress event, and will add the
        item in the text box to the list when the user presses the
        enter key. In the future, this method will also handle the
        auto complete feature.
        """
        if keyCode == KeyboardListener.KEY_ENTER and \
           sender == self.todoTextBox:
            id = self.remote.addTask(sender.getText(),self)
            sender.setText("")
            if id<0:
                RootPanel().add(HTML("Server Error or Invalid
                    Response"))

    def onClick(self, sender):
        id = self.remote.deleteTask(
                sender.getValue(sender.getSelectedIndex()),self)
        if id<0:
            RootPanel().add(
                HTML("Server Error or Invalid Response"))

    def onRemoteResponse(self, response, request_info):
        self.todoList.clear()
        for task in response:
            self.todoList.addItem(task[0])
            self.todoList.setValue(self.todoList.getItemCount()-1,
                                   task[1])

    def onRemoteError(self, code, message, request_info):
        self.Status.setText("Server Error or Invalid Response: " \
                            + "ERROR " + code + " - " + message)

class DataService(JSONProxy):
    def __init__(self):
        JSONProxy.__init__(self, "../../default/call/jsonrpc",
                           ["getTasks", "addTask","deleteTask"])

if __name__ == '__main__':
    app = TodoApp()
    app.onModuleLoad()
```

Sixth, run Pyjamas before serving the application:

```
cd /path/to/todo/static/
python ~/python/pyjamas-0.5p1/bin/pyjsbuild TodoApp.py
```

This will translate the Python code into JavaScript so that it can be executed in the browser.

To access this application, visit the URL

```
http://127.0.0.1:8000/todo/default/todoApp
```

Credits This subsection was created by Chris Prinos with help form Luke Kenneth Casson Leighton (creators of Pyjamas) and updated by Alexei Vinidiktov. It has been tested by Pyjamas 0.5p1. The example was inspired by this Django page:

```
http://gdwarner.blogspot.com/2008/10/brief-pyjamas-django-tutorial.
    html
```

AMFRPC

AMFRPC is the Remote Procedure Call protocol used by Flash clients to communicate with a server. WEB2PY supports AMFRPC but it requires that you run WEB2PY from source and that you preinstall the PyAMF library. This can be installed from the Linux or Windows shell by typing

```
easy_install pyamf
```

(please consult the PyAMF documentation for more details).

In this subsection we assume that you are already familiar with ActionScript programming.

We will create a simple service that takes two numerical values, adds them together, and returns the sum. We will call our WEB2PY application "pyamf_test", and we will call the service `addNumbers`.

First, using Adobe Flash (any version starting from MX 2004), create the Flash client application by starting with a new Flash FLA file. In the first frame of the file, add these lines:

```
import mx.remoting.Service;
import mx.rpc.RelayResponder;
import mx.rpc.FaultEvent;
import mx.rpc.ResultEvent;
import mx.remoting.PendingCall;

var val1 = 23;
var val2 = 86;

service = new Service(
    "http://127.0.0.1:8000/pyamf_test/default/call/amfrpc3",
    null, "mydomain", 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();
```

This code allows the Flash client to connect to a service that corresponds to a function called "addNumbers" in the file "/pyamf_test/default/gateway". You must also import ActionScript version 2 MX remoting classes to enable Remoting in Flash. Add the path to these classes to the classpath settings in the Adobe Flash IDE, or just place the "mx" folder next to the newly created file.

Notice the arguments of the Service constructor. The first argument is the URL corresponding to the service that we want will create. The third argument is the domain of the service. We choose to call this domain "mydomain".

Second, create a dynamic text field called "txt_result" and place it on the stage.

Third, you need to set up a WEB2PY gateway that can communicate with the Flash client defined above.

Proceed by creating a new WEB2PY app called `pyamf_test` that will host the new service and the AMF gateway for the flash client. Edit the "default.py" controller and make sure it contains

```
@service.amfrpc3('mydomain')
def addNumbers(val1, val2):
    return val1 + val2

def call(): return service()
```

Fourth, compile and export/publish the SWF flash client as `pyamf_test.swf`, place the "pyamf_test.amf", "pyamf_test.html", "AC_RunActiveContent.js", and "crossdomain.xml" files in the "static" folder of the newly created appliance that is hosting the gateway, "pyamf_test".

You can now test the client by visiting:

```
http://127.0.0.1:8000/pyamf_test/static/pyamf_test.html
```

The gateway is called in the background when the client connects to addNumbers.

If you are suing AMF0 instead of AMF3 you can also use the decorator:

```
@service.amfrpc
```

instead of:

```
@service.amfrpc3('mydomain')
```

In this case you also need to change the service URL to:

```
http://127.0.0.1:8000/pyamf_test/default/call/amfrpc
```

9.3 Low Level API and Other Recipes

simplejson

WEB2PY includes gluon.contrib.simplejson, developed by Bob Ippolito. This module provides the most standard Python-JSON encoder-decoder.

SimpleJSON consists of two functions:

- `gluon.contrib.simplesjson.dumps(a)` encodes a Python object `a` into JSON.
- `gluon.contrib.simplejson.loads(b)` decodes a JavaScript object `b` into a Python object.

Object types that can be serialized include primitive types, lists, and dictionaries. Compound objects can be serialized with the exception of user defined classes.

Here is a sample action (for example in controller "default.py") that serializes the Python list containing weekdays using this low level API:

```
def weekdays():
    names=['Sunday','Monday','Tuesday','Wednesday',
           'Thursday','Friday','Saturday']
    import gluon.contrib.simplejson
    return gluon.contrib.simplejson.dumps(names)
```

Below is a sample HTML page that sends an Ajax request to the above action, receives the JSON message, and stores the list in a corresponding JavaScript variable:

```
{{extend 'layout.html'}}
<script>
$.getJSON('/application/default/weekdays',
          function(data){ alert(data); });
</script>
```

The code uses the jQuery function `$.getJSON`, which performs the Ajax call and, on response, stores the weekdays names in a local JavaScript variable `data` and passes the variable to the callback function. In the example the callback function simply alerts the visitor that the data has been received.

PyRTF

Another common need of web sites is that of generating Word-readable text documents. The simplest way to do so is using the Rich Text Format (RTF) document format. This format was invented by Microsoft and it has since become a standard.

WEB2PY includes gluon.contrib.pyrtf, developed by Simon Cusack and revised by Grant Edwards. This module allows you to generate RTF documents programmatically including colored formatted text and pictures.

In the following example we instantiate two basic RTF classes, Document and Section, append the latter to the former and insert some dummy text in the latter:

```
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)
```

In the end the Document is serialized by `q.dumps(doc)`. Notice that before returning an RTF document it is necessary to specify the content-type in the header else the browser does not know how to handle the file.

Depending on the configuration, the browser may ask you whether to save this file or open it using a text editor.

ReportLab and PDF

WEB2PY can also generate PDF documents, with an additional library called "ReportLab"[66].

If you are running WEB2PY from source, it is sufficient to have ReportLab installed. If you are running the Windows binary distribution, you need to unzip ReportLab in the "web2py/" folder. If you are running the Mac binary distribution, you need to unzip ReportLab in the folder:

```
web2py.app/Contents/Resources/
```

From now on we assume ReportLab is installed and that WEB2PY can find it. We will create a simple action called "get_me_a_pdf" that generates a PDF document.

```
from reportlab.platypus import *
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.rl_config import defaultPageSize
from reportlab.lib.units import inch, mm
```

```
from reportlab.lib.enums import TA_LEFT, TA_RIGHT, TA_CENTER,
    TA_JUSTIFY
from reportlab.lib import colors
from uuid import uuid4
from cgi import escape
import os

def get_me_a_pdf():
    title = "This The Doc Title"
    heading = "First Paragraph"
    text = 'bla '* 10000

    styles = getSampleStyleSheet()
    tmpfilename=os.path.join(request.folder,'private',str(uuid4()))
    doc = SimpleDocTemplate(tmpfilename)
    story = []
    story.append(Paragraph(escape(title),styles["Title"]))
    story.append(Paragraph(escape(heading),styles["Heading2"]))
    story.append(Paragraph(escape(text),styles["Normal"]))
    story.append(Spacer(1,2*inch))
    doc.build(story)
    data = open(tmpfilename,"rb").read()
    os.unlink(tmpfilename)
    response.headers['Content-Type']='application/pdf'
    return data
```

Notice how we generate the PDF into a unique temporaty file, `tmpfilename`, we read the generated PDF from the file, then we deletedthe file.

For more information about the ReportLab API, refer to the ReportLab documentation. We strongly recomment using the Platypus API of ReportLab, such as `Paragraph`, `Spacer`, etc.

9.4 Services and Authentication

In the previous chapter we have discussed the use of the following decorators:

```
@auth.requires_login()
@auth.requires_memebrship(...)
@auth.requires_permission(...)
```

For normal actions (not decorated as services), these decorators can be used even if the output is rendered in a format other than HTML.

For functions defined as services and decorated using the `@service...` decorators, the `@auth...` decorators should not be used. The two types of decorators cannot be mixed. If authenticaiton is to be performed, it is the `call` actions that needs to be decorated:

```
@auth.requires_login()
def call(): return service()
```

Notice that it also possible to instantiate multiple service objects, register the same different functions with them, and expose some of them with authentication and some not:

```
public_services=Service(globals())
private_services=Service(globals())

@public_service.jsonrpc
@private_service.jsonrpc
def f(): return 'public'

@private_service.jsonrpc
def g(): return 'private'

def public_call(): return public_service()

@auth.requires_login()
def private_call(): return private_service()
```

This assumes that the caller is passing credentials in the HTTP header (a valid session cookie or using basic authentication, as discussed in the previous section). The client must support it; not all clients do.

CHAPTER 10

AJAX RECIPES

While WEB2PY is mainly for server-side development, it comes with the base jQuery library [31], jQuery calendars (date picker, datetime picker and clock) and some additional JavaScript functions based on jQuery.

Nothing in WEB2PY prevents you from using other Ajax [67] libraries such as Prototype, Scriptaculous or ExtJS but we decided to package jQuery because we find it to be easier to use and more powerful than any other equivalent libraries. We also find it captures the WEB2PY spirit of being functional and concise.

10.1 web2py_ajax.html

The scaffolding WEB2PY application "welcome" includes a file called

```
1 views/web2py_ajax.html
```

This file is included in the HEAD of the default "layout.html" and it provides the following services:

- Includes `static/jquery.js`.
- Includes `static/calendar.js` and `static/calendar.css`, if they exist.
- Defines a `popup` function.
- Defines a `collapse` function (based on jQuery slideToggle).
- Defines a `fade` function (based on jQuery fade).
- Defines an `ajax` function (based on jQuery $.ajax).
- Makes any DIV of class "error" or any tag object of class "flash" slide down.
- Prevents typing invalid integers in INPUT fields of class "integer".
- Prevents typing invalid floats in INPUT fields of class "double".
- Connects INPUT fields of type "date" with a popup date picker.
- Connects INPUT fields of type "datetime" with a popup datetime picker.
- Connects INPUT fields of type "time" with a popup time picker.

`popup`, `collapse`, and `fade` are included only for backward compatibility, and are not discussed here.

Here is an an example of how the other effects play well together.

Consider a **test** app with the following model:

```
db = DAL("sqlite://db.db")
db.define_table('mytable',
     Field('field_integer', 'integer'),
     Field('field_date', 'date'),
     Field('field_datetime', 'datetime'),
     Field('field_time', 'time'))
```

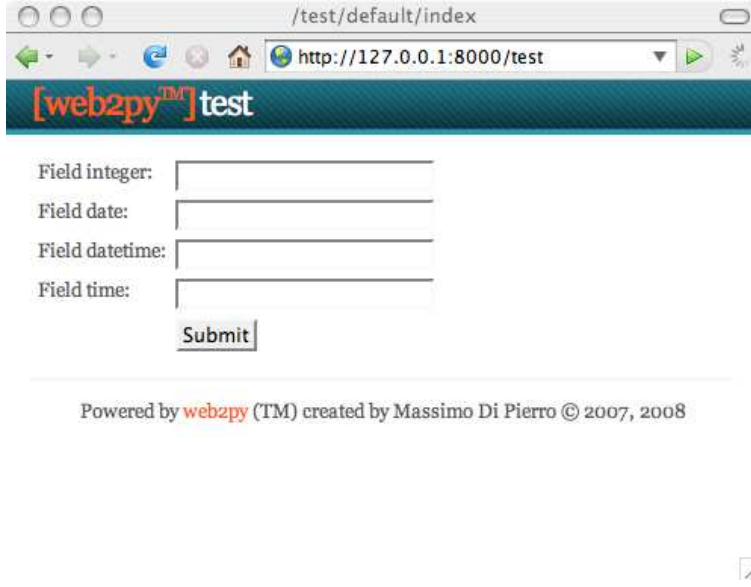
with this "default.py" controller:

```
def index():
    form = SQLFORM(db.mytable)
    if form.accepts(request.vars, session):
        response.flash = 'record inserted'
    return dict(form=form)
```

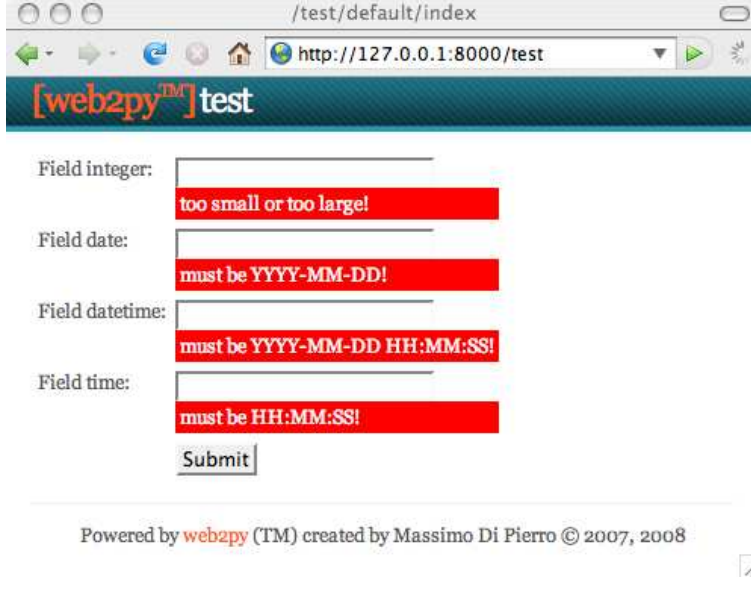
and the following "default/index.html" view:

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

The "index" action generates the following form:



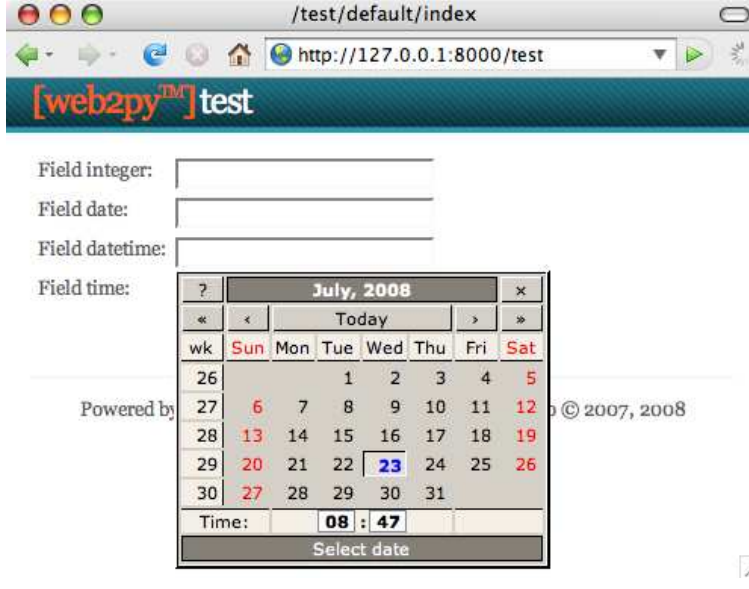
If an invalid form is submitted, the server returns the page with a modified form containing error messages. The error messages are DIVs of class "error", and because of the above web2py_ajax code, the errors appears with a slide-down effect:



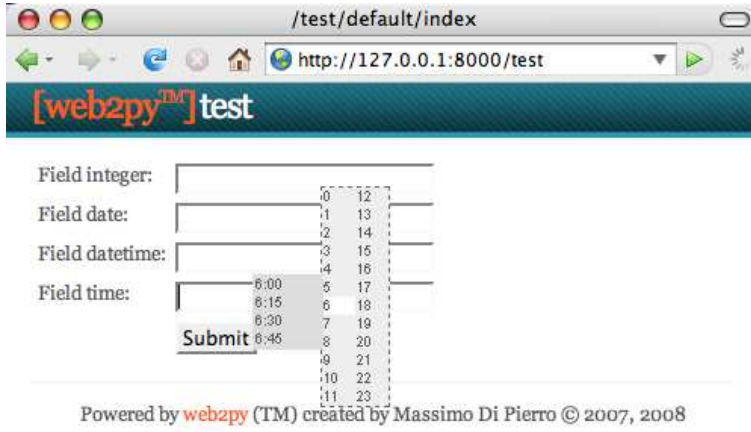
The color of the errors is given in the CSS code in "layout.html".

The web2py_ajax code prevents you from typing an invalid value in the input field. This is done before and in addition to, not as a substitute for, the server-side validation.

The web2py_ajax code displays a date picker when you enter an INPUT field of class "date", and it displays a datetime picker when you enter an INPUT field of class "datetime". Here is an example:



The web2py_ajax code also displays the following time picker when you try to edit an INPUT field of class "time":



Upon submission, the controller action sets the response flash to the message "record inserted". The default layout renders this message in a DIV with id="flash". The web2py_ajax code is responsible for making this DIV slide down and making it disappear when you click on it:

These and other effects are accessible programmatically in the views and via helpers in controllers.

10.2 jQuery Effects

Using jQuery effects is very easy. Here we describe how to do it.

The basic effects described here do not require any additional files; everything you need is already included for you by web2py_ajax.html.

HTML/XHTML objects can be identified by their type (for example a DIV), their classes, or their id. For example:

```
<div class="one" id="a">Hello</div>
<div class="two" id="b">World</div>
```

They belong to class "one" and "two" respectively. They have ids equal to "a" and "b" respectively.

In jQuery you can refer to the former with the following a CSS-like equivalent notations

```
jQuery('.one')    // address object by class "one"
jQuery('#a')      // address object by id "a"
jQuery('DIV.one') // address by object of type "DIV" with class "one"
jQuery('DIV #a')  // address by object of type "DIV" with id "a"
```

and to the latter with

```
jQuery('.two')
jQuery('#b')
jQuery('DIV.two')
jQuery('DIV #b')
```

or you can refer to both with

```
jQuery('DIV')
```

Tag objects are associated to events, such as "onclick". jQuery allows linking these events to effects, for example "slideToggle":

```
<div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello
    </div>
<div class="two" id="b">World</div>
```

Now if you click on "Hello", "World" disappears. If you click again, "World" reappears.

You can also link actions to events outside the tag itself. The previous code can be rewritten as follows:

```
<div class="one" id="a">Hello</div>
<div class="two" id="b">World</div>
<script>
jQuery('.one').click(function(){jQuery('.two').slideToggle()});
</script>
```

Effects return the calling object, so they can be chained.

When the `click` sets the callback function to be called on click. Similarly for `change`, `keyup`, `keydown`, `mouseover`, etc.

A common situation is the need to execute some JavaScript code only after the entire document has been loaded. This is usually done by the onload attribute of BODY but jQuery provides an alternative way that does not require editing the layout:

```
<div class="one" id="a">Hello</div>
<div class="two" id="b">World</div>
<script>
jQuery(document).ready(function(){
   jQuery('.one').click(function(){jQuery('.two').slideToggle()});
});
</script>
```

The body of the unnamed function is executed only when the document is ready, after it has been fully loaded.

Here is a list of useful event names:

Form Events

- onchange: Script to be run when the element changes
- onsubmit: Script to be run when the form is submitted
- onreset: Script to be run when the form is reset
- onselect: Script to be run when the element is selected
- onblur: Script to be run when the element loses focus
- onfocus: Script to be run when the element gets focus

Keyboard Events

- onkeydown: Script to be run when key is pressed
- onkeypress: Script to be run when key is pressed and released
- onkeyup: Script to be run when key is released

Mouse Events

- onclick: Script to be run on a mouse click
- ondblclick: Script to be run on a mouse double-click
- onmousedown: Script to be run when mouse button is pressed
- onmousemove: Script to be run when mouse pointer moves
- onmouseout: Script to be run when mouse pointer moves out of an element

- onmouseover: Script to be run when mouse pointer moves over an element
- onmouseup: Script to be run when mouse button is released

Here is a list of useful effects defined by jQuery:

Effects

- jQuery(...).attr(name): Returns the name of the attribute value
- jQuery(...).attr(name, value): Sets the attribute name to value
- jQuery(...).show(): Makes the object visible
- jQuery(...).hide(): Makes the object hidden
- jQuery(...).slideToggle(speed, callback): Makes the object slide up or down
- jQuery(...).slideUp(speed, callback): Makes the object slide up
- jQuery(...).slideDown(speed, callback): Makes the object slide down
- jQuery(...).fadeIn(speed, callback): Makes the object fade in
- jQuery(...).fadeOut(speed, callback): Makes the object fade out

The speed argument is usually "slow", "fast" or omitted (the default). The callback is an optional function that is called when the effect is completed.

jQuery effects can also easily be embedded in helpers, for example, in a view:

```
{{=DIV('click me!', _onclick="jQuery(this).fadeOut()")}}
```

jQuery is a very compact and concise Ajax library; therefore WEB2PY does not need an additional abstraction layer on top of jQuery (except for the `ajax` function discussed below). The jQuery APIs are accessible and readily available in their native form when needed.

Consult the documentation for more information about these effects and other jQuery APIs.

The jQuery library can also be extended using plugins and User Interface Widgets. This topic is not covered here; see ref. [69] for details.

Conditional Fields in Forms

A typical application of jQuery effects is a form that changes its appearance based on the value of its fields.

This is easy in WEB2PY because the SQLFORM helper generates forms that are "CSS friendly". The form contains a table with rows. Each row contains a label, an input field, and an optional third column. The items have ids derived strictly from the name of the table and names of the fields.

The convention is that every INPUT field has a name equal to *tablename_fieldname* and is contained in a row called *tablename_fieldname__row*.

As an example, create an input form that asks for a taxpayer's name and for the name of the taxpayer's spouse, but only if he/she is married.

Create a test application with the following model:

```
db = DAL('sqlite://db.db')
db.define_table('taxpayer',
    Field('name'),
    Field('married', 'boolean'),
    Field('spouse_name'))
```

the following "default.py" controller:

```
def index():
    form = SQLFORM(db.taxpayer)
    if form.accepts(request.vars, session):
        response.flash = 'record inserted'
    return dict(form=form)
```

and the following "default/index.html" view:

```
{{extend 'layout.html'}}
{{=form}}
<script>
jQuery(document).ready(function(){
   jQuery('#taxpayer_spouse_name__row').hide();
   jQuery('#taxpayer_married').change(function(){
        if(jQuery('#taxpayer_married').attr('checked'))
            jQuery('#taxpayer_spouse_name__row').show();
        else jQuery('#taxpayer_spouse_name__row').hide();});
});
</script>
```

The script in the view has the effect of hiding the row containing the spouse's name:

When the taxpayer checks the "married" checkbox, the spouse's name field reappears:

Here "taxpayer_married" is the checkbox associated to the "boolean" field "married" of table "taxpayer". "taxpayer_spouse_name__row" it the row containing the input field for "spouse_name" of table "taxpayer".

Confirmation on Delete

Another useful application is requiring confirmation when checking a "delete" checkbox such as the delete checkbox that appears in edit forms.

Consider the above example and add the following controller action:

```
def edit():
    row = db(db.taxpayer.id==request.args[0]).select()[0]
    form = SQLFORM(db.taxpayer, row, deletable=True)
```

```
    if form.accepts(request.vars, session):
        response.flash = 'record updated'
    return dict(form=form)
```

and the corresponding view "default/edit.html"

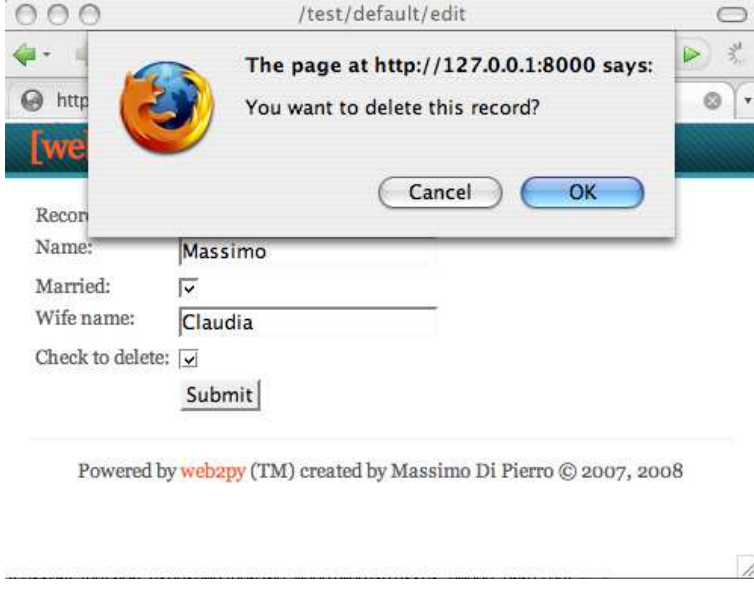
```
{{extend 'layout.html'}}
{{=form}}
```

The `deletable=True` argument in the SQLFORM constructor instructs WEB2PY to display a "delete" checkbox in the edit form.

WEB2PY's "web2py_ajax.html" includes the following code:

```
jQuery(document).ready(function(){
   jQuery('input.delete').attr('onclick',
     'if(this.checked) if(!confirm(
        "{{=T('Sure you want to delete this object?')}}"))
      this.checked=false;');
});
```

By convention this checkbox has a class equal to "delete". The jQuery code above connects the onclick event of this checkbox with a confirmation dialog (standard in JavaScript) and unchecks the checkbox if the taxpayer does not confirm:



10.3 The ajax Function

In web2py_ajax.html, WEB2PY defines a function called `ajax` which is based on, but should not be confused with, the jQuery function `$.ajax`. The latter is much more powerful than the former, and for its usage, we refer you to ref. [31] and ref. [68]. However, the former function is sufficient for many complex tasks, and is easier to use.

The `ajax` function is a JavaScript function that has the following syntax:

```
ajax(url, [id1, id2, ...], target)
```

It asynchronously calls the url (first argument), passes the values of the fields with the id equal to one of the ids in the list (second argument), then stores the response in the innerHTML of the tag with the id equal to target (the third argument).

Here is an example of a `default` controller:

```
def one():
    return dict()

def echo():
    return request.vars.name
```

and the associated "default/one.html" view:

```
{{extend 'layout.html'}}
<form>
   <input id="name" onkeyup="ajax('echo', ['name'], 'target')" />
</form>
<div id="target"></div>
```

When you type something in the INPUT field, as soon as you release a key (onkeyup), the `ajax` function is called, and the value of the `id="name"` field is passed to the action "echo", which sends the text back to the view. The `ajax` function receives the response and displays the echo response in the "target" DIV.

Eval target

The thrid argument of the ajax function can be the string ":eval". This means that the string returned by server will not be embedded in the document but it will be evaluated instead.

Here is an example of a `default` controller:

```
def one():
    return dict()

def echo():
    return "jQuery('#target').html(%s);" % repr(request.vars.name)
```

and the associated "default/one.html" view:

```
{{extend 'layout.html'}}
<form>
   <input id="name" onkeyup="ajax('echo', ['name'], ':eval')" />
</form>
<div id="target"></div>
```

This allows for more articulated responses than simple strings.

Auto-completion

Another application of the above `ajax` function is auto-completion. Here we wish to create an input field that expects a month name and, when the visitor types an incomplete name, performs auto-completion via an Ajax request. In response, an auto-completion drop-box appears below the input field.

This can be achieved via the following `default` controller:

```
def month_input():
    return dict()

def month_selector():
    if not request.vars.month:
        return ''
    months = ['January', 'February', 'March', 'April', 'May',
            'June', 'July', 'August', 'September' ,'October',
            'November', 'December']
    selected = [m for m in months \
                if m.startswith(request.vars.month.capitalize())]
    return ''.join([DIV(k,
                  _onclick="jQuery('#month').val('%s')" % k,
                  _onmouseover="this.style.backgroundColor='yellow'",
                  _onmouseout="this.style.backgroundColor='white'"
                  ).xml() for k in selected])
```

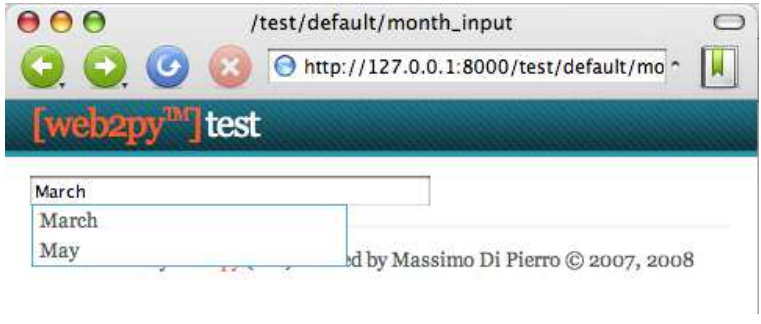
and the corresponding "default/month_input.html" view:

```
{{extend 'layout.html'}}
<style>
#suggestions { position: relative; }
.suggestions { background: white; border: solid 1px #55A6C8; }
.suggestions DIV { padding: 2px 4px 2px 4px; }
</style>

<form>
 <input type="text" id="month" style="width: 250px" /><br />
 <div style="position: absolute;" id="suggestions"
      class="suggestions"></div>
</form>
<script>
jQuery("#month").keyup(function(){
      ajax('complete', ['month'], 'suggestions')});
</script>
```

The jQuery script in the view triggers the Ajax request each time the visitor types something in the "month" input field. The value of the input field is submitted with the Ajax request to the "month_selector" action. This action finds a list of month names that start with the submitted text (selected), builds a list of DIVs (each one containing a suggested month name), and returns a string with the serialized DIVs. The view displays the response HTML in the "suggestions" DIV. The "month_selector" action generates both the suggestions and the JavaScript code embedded in the DIVs that must be executed when the visitor clicks on each suggestion. For example when the visitor types "Ma" the callback action returns:

```
<div onclick="jQuery('#month').val('February')"
     onmouseout="this.style.backgroundColor='white'"
     onmouseover="this.style.backgroundColor='yellow'">February</div>
```

Here is the final effect:



If the months are stored in a database table such as:

```
db.define_table('month', Field('name'))
```

then simply replace the `month_selector` action with:

```
def month_input():
    return dict()

def month_selector():
    it not request.vars.month:
        return ''
    pattern = request.vars.month.capitalize() + '%'
    selected = [row.name for row in db(db.month.name.like(pattern)).
        select()]
    return ''.join([DIV(k,
                    _onclick="jQuery('#month').val('%s')" % k,
                    _onmouseover="this.style.backgroundColor='yellow'",
                    _onmouseout="this.style.backgroundColor='white'"
                    ).xml() for k in selected])
```

jQuery provides an optional Auto-complete Plugin with additional functionalities, but that is not discussed here.

Form Submission

Here we consider a page that allows the visitor to submit messages using Ajax without reloading the entire page. It contains a form "myform" and a "target" DIV. When the form is submitted, the server may accept it (and perform a database insert) or reject it (because it did not pass validation). The corresponding notification is returned with the Ajax response and displayed in the "target" DIV.

Build a `test` application with the following model:

```
db = DAL('sqlite://db.db')
db.define_table('post', Field('your_message', 'text'))
db.post.your_message.requires = IS_NOT_EMPTY()
```

Notice that each post has a single field "your_message" that is required to be not-empty.

Edit the `default.py` controller and write two actions:

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

def new_post():
    form = SQLFORM(db.post)
    if form.accepts(request.vars, formname=None):
        return DIV("Message posted")
    elif form.errors:
        return TABLE(*[TR(k, v) for k, v in form.errors.items()])
```

The first action does nothing other than return a view.

The second action is the Ajax callback. It expects the form variables in `request.vars`, processes them and returns `DIV("Message posted")` upon success or a `TABLE` of error messages upon failure.

Now edit the "default/index.html" view:

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

<div id="target"></div>

<form id="myform">
  <input name="your_message" id="your_message" />
  <input type="submit" />
</form>

<script>
jQuery('#myform').submit(function() {
  ajax('{{=URL(r=request, f='new_post')}}',
       ['your_message'], 'target');
  return false;
});
</script>
```

Notice how in this example the form is created manually using HTML, but it is processed by the SQLFORM in a different action than the one that

displays the form. The SQLFORM object is never serialized in HTML. `SQLFORM.accepts` in this case does not take a session and sets `formname=None`, because we chose not to set the form name and a form key in the manual HTML form.

The script at the bottom of the view connects the "myform" submit button to an inline function which submits the INPUT with `id="your_message"` using the WEB2PY `ajax` function, and displays the answer inside the DIV with `id="target"`.

Voting and Rating

Another Ajax application is voting or rating items in a page. Here we consider an application that allows visitors to vote on posted images. The application consists of a single page that displays the images sorted according to their vote. We will allow visitors to vote multiple times, although it is easy to change this behavior if visitors are authenticated, by keeping track of the individual votes in the database and associating them with the request.env.remote_addr of the voter.

Here is a sample model:

```
db = DAL('sqlite://images.db')
db.define_table('item',
    Field('image', 'upload'),
    Field('votes', 'integer', default=0))
```

Here is the `default` controller:

```
def list_items():
    items = db().select(db.item.ALL, orderby=~db.item.votes)
    return dict(items=items)

def download():
    return response.download(request, db)

def vote():
    item = db(db.item.id==request.vars.id).select()[0]
    new_votes = item.votes + 1
    item.update_record(votes=new_votes)
    return str(new_votes)
```

The download action is necessary to allow the list_items view to download images stored in the "uploads" folder. The votes action is used for the Ajax callback.

Here is the "default/list_items.html" view:

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

<form><input type="hidden" id="id" value="" /></form>
{{for item in items:}}
```

```
<p>
<img src="{{=URL(r=request, f='download', args=item.image)}}"
     width="200px" />
<br />
Votes=<span id="item{{=item.id}}">{{=item.votes}}</span>
[<span onclick="jQuery('#id').val('{{=item.id}}');
       ajax('vote', ['id'], 'item{{=item.id}}');">vote up</span>]
</p>
{{pass}}
```

When the visitor clicks on "[vote up]" the JavaScript code stores the item.id in the hidden "id" INPUT field and submits this value to the server via an Ajax request. The server increases the votes counter for the corresponding record and returns the new vote count as a string. This value is then inserted in the target `"item{{=item.id}}"` SPAN.

> *Ajax callbacks can be used to perform computations in the background, but we reccomment using CRON instead (discussed in chapter 4), since the web server enforces a timeout on threads. If the computation takes too long, the web server kills it. Refer to your web server parameters to set the timeout value.*

CHAPTER 11

DEPLOYMENT RECIPES

There are multiple ways to deploy WEB2PY in a production environment; the details depend on the configuration and the services provided by the host.

In this chapter we consider the following issues:

- Configuration of production-quality web servers (Apache, Lighttpd, Cherokee)
- Security Issues
- Scalability issues
- Deployment on the Google App Engine (GAE [12])

WEB2PY comes with an SSL [20] enabled web server, the CherryPy wsgiserver [21]. While this is a fast web server, it has limited configuration capabilities. For this reason it is best to deploy WEB2PY behind Apache [71], Lighttpd [75] or Cherokee [76]. These are free and open-source web servers that are customizable and have been proven to be reliable in high traffic production environments. They can be configured to serve static files directly, deal with HTTPS, and pass control to WEB2PY for dynamic content.

Until a few years ago, the standard interface for communication between web servers and web applications was the Common Gateway Interface (CGI) [70]. The main problem with CGI is that it creates a new process for each HTTP request. If the web application is written in an interpreted language, each HTTP request served by the CGI scripts starts a new instance of the interpreter. This is slow, and it should be avoided in a production environment. Moreover, CGI can only handle simple responses. It cannot handle, for example, file streaming.

WEB2PY provides a file `modpythonhandler.py` to interface to CGI.

One solution to this problem is to use the mod_python module for Apache. mod_python starts one instance of the Python interpreter when Apache starts, and serves each HTTP request in its own thread without having to restart Python each time. This is a better solution than CGI, but it is not an optimal solution, since mod_python uses its own interface for communication between the web server and the web application. In mod_python, all hosted applications run under the same user-id/group-id, which presents security issues.

WEB2PY provides a file `cgihandler.py` to interface to mod_python.

In the last few years, the Python community has come together behind a new standard interface for communication between web servers and web applications written in Python. It is called Web Server Gateway Interface (WSGI) [17, 18]. WEB2PY was built on WSGI, and it provides handlers for using other interfaces when WSGI is not available.

Apache supports WSGI via the module mod_wsgi [74] developed by Graham Dumpleton.

WEB2PY provides a file `wsgihandler.py` to interface to WSGI.

Some web hosting services do not support mod_wsgi. In this case, we must use Apache as a proxy and forward all incoming requests to the WEB2PY built-in web server (running for example on localhost:8000).

In both cases, with mod_wsgi and/or mod_proxy, Apache can be configured to serve static files and deal with SSL encryption directly, taking the burden off WEB2PY.

The Lighttpd web server does not currently support the WSGI interface, but it does support the FastCGI [77] interface, which is an improvement over CGI. FastCGI's main aim is to reduce the overhead associated with interfacing the web server and CGI programs, allowing a server to handle more HTTP requests at once.

According to the Lighttpd web site, "Lighttpd powers several popular Web 2.0 sites such as YouTube and Wikipedia. Its high speed IO-infrastructure allows them to scale several times better with the same hardware than with

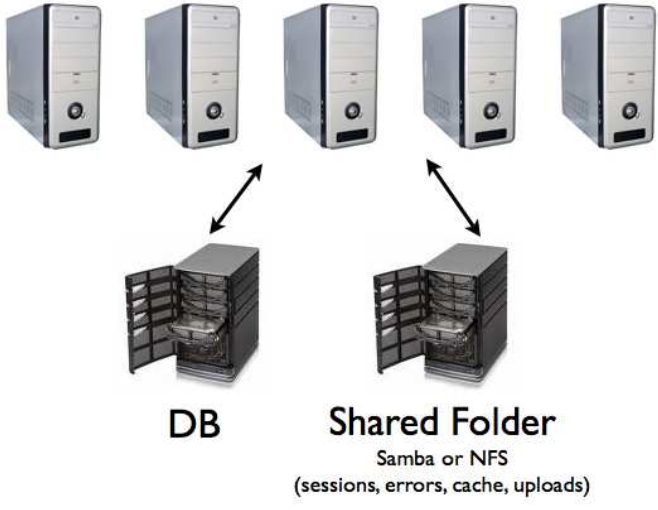
alternative web-servers". Lighttpd with FastCGI is, in fact, faster than Apache with mod_wsgi.

WEB2PY provides a file `fcgihandler.py` to interface to FastCGI.

WEB2PY also includes a `gaehandler.py` to interface with the Google App Engine (GAE). On GAE, web applications run "in the cloud". This means that the framework completely abstracts any hardware details. The web application is automatically replicated as many times as necessary to serve all concurrent requests. Replication in this case means more than multiple threads on a single server; it also means multiple processes on different servers. GAE achieves this level of scalability by blocking write access to the file system and all persistent information must be stored in the Google BigTable datastore or in memcache.

On non-GAE platforms, scalability is an issue that needs to be addressed, and it may require some tweaks in the WEB2PY applications. The most common way to achieve scalability is by using multiple web servers behind a load-balancer (a simple round robin, or something more sophisticated, receiving heartbeat feedback from the servers).

Even if there are multiple web servers, there must be one, and only one, database server. By default, WEB2PY uses the file system for storing sessions, error tickets, uploaded files, and the cache. This means that in the default configuration, the corresponding folders have to be shared folders:



In the rest of the chapter, we consider various recipes that may provide an improvement over this naive approach, including:

- Store sessions in the database, in cache or do not store sessions at all.

- Store tickets on local filesystems and move them into the database in batches.
- Use memcache instead of cache.ram and cache.disk.
- Store uploaded files in the database instead of the shared filesystem.

While we recommend following the first three recipes, the fourth recipe may provide an advantage mainly in the case of small files, but may be counterproductive for large files.

11.1 Setup Apache on Linux

In this section, we use Ubuntu 8.04 Server Edition as the reference platform. The configuration commands are very similar on other Debian-based Linux distribution, but they may differ for Red Hat-based systems.

First, make sure all the necessary Python and Apache packages are installed by typing the following shell commands:

```
sudo apt-get update
sudo apt-get -y upgrade
sudo apt-get -y install openssh-server
sudo apt-get -y install python
sudo apt-get -y install python-dev
sudo apt-get -y install apache2
sudo apt-get -y install libapache2-mod-wsgi
```

Then, enable the SSL module, the proxy module, and the WSGI module in Apache:

```
sudo a2enmod ssl
sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod wsgi
```

Create the SSL folder, and put the SSL certificates inside it:

```
sudo mkdir /etc/apache2/ssl
```

You should obtain your SSL certificates from a trusted Certificate Authority such as verisign.com, but, for testing purposes, you can generate your own self-signed certificates following the instructions in ref. [73]

Then restart the web server:

```
sudo /etc/init.d/apache2 restart
```

The Apache configuration file is:

```
/etc/apache2/sites-available/default
```

The Apache logs are in:

```
/var/log/apache2/
```

11.2 Setup mod_wsgi on Linux

Download and unzip WEB2PY source on the machine where you installed the web server above.

Install WEB2PY under `/users/www-data/`, for example, and give ownership to user www-data and group www-data. These steps can be performed with the following shell commands:

```
cd /users/www-data/
sudo wget http://web2py.com/examples/static/web2py_src.zip
sudo unzip web2py_src.zip
sudo chown -R www-data:www-data /user/www-data/web2py
```

To set up WEB2PY with mod_wsgi, create a new Apache configuration file:

```
/etc/apache2/sites-available/web2py
```

and include the following code:

```
<VirtualHost *:80>
  ServerName web2py.example.com
  WSGIDaemonProcess web2py user=www-data group=www-data
                           display-name=%{GROUP}
  WSGIProcessGroup web2py
  WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py

  <Directory /users/www-data/web2py>
    AllowOverride None
    Order Allow,Deny
    Deny from all
    <Files wsgihandler.py>
      Allow from all
    </Files>
  </Directory>

  AliasMatch ^/([^/]+)/static/(.*)
           /users/www-data/web2py/applications/$1/static/$2
  <Directory /users/www-data/web2py/applications/*/static/>
    Order Allow,Deny
    Allow from all
  </Directory>

  <Location /admin>
  Deny from all
  </Location>

  <LocationMatch ^/([^/]+)/appadmin>
  Deny from all
```

```
  </LocationMatch>

  CustomLog /private/var/log/apache2/access.log common
  ErrorLog /private/var/log/apache2/error.log
</VirtualHost>
```

When you restart Apache, it should pass all the requests to web2y without going through the CherryPy wsgiserver.

Here are some explanations:

```
WSGIDaemonProcess web2py user=www-data group=www-data
                        display-name=%{GROUP}
```

defines a daemon process group in context of "web2py.example.com". By defining this inside of the virtual host, only this virtual host, including any virtual host for same server name but on a different port, can access this using WSGIProcessGroup. The "user" and "group" options should be set to the user who has write access to the directory where WEB2PY was setup. You do not need to set "user" and "group" if you made the WEB2PY installation directory writable to the user that Apache runs as by default. The "display-name" option is so that process name appears in "ps" output as "(wsgi:web2py)" instead of as name of Apache web server executable. As no "processes" or "threads" options specified, the daemon process group will have a single process with 15 threads running within that process. This is usually more than adequate for most sites and should be left as is. If overriding it, do not use "processes=1" as doing so will disable any in browser WSGI debugging tools that check the "wsgi.multiprocess" flag. This is because any use of the "processes" option will cause that flag to be set to true, even if a single process and such tools expect that it be set to false. Note that if your own application code or some third party extension module you are using with Python is not thread safe, instead use options "processes=5 threads=1". This will create five processes in the daemon process group where each process is single threaded. You might consider using "maximum-requests=1000" if your application leaks Python objects through inability for them to be garbage collected properly.

```
WSGIProcessGroup web2py
```

delegates running of all WSGI applications to the daemon process group that was configured using the WSGIDaemonProcess directive.

```
WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
```

mounts the WEB2PY application. In this case it is mounted at the root of the web site. Not known how to get WEB2PY to mount at a sub URL as doesn''t appear to be a good WSGI citizen and work out where it is mounted from value of SCRIPT_NAME and then automatically adjust everything appropriately without further manual user configuration.

```
<Directory /users/www-data/web2py>
  ...
</Directory>
```

gives Apache permission to access the WSGI script file.

```
<Directory /users/www-data/web2py/applications/*/static/>
  Order Allow,Deny
  Allow from all
</Directory>
```

Instructs Apache to bypass web2py when sering static files.

```
<Location /admin>
  Deny from all
</Location>
```

and

```
<LocationMatch ^/([^/]+)/appadmin>
  Deny from all
</LocationMatch>
```

block public access to **admin** and **appadmin**

Normally would just allow permission to the whole directory the WSGI script file is located in, but cant do that with WEB2PY, as it places the WSGI script file in a directory which contains other source code, including the file containing the admin interface password. Opening up the whole directory would cause security issues, because technically Apache would be given permission to serve all the files up to a user if there was any way of traversing to that directory via a mapped URL. To avoid security problems, explicitly deny access to the contents of the directory, except for the WSGI script file and prohibit a user from doing any overrides from a .htaccess file to be extra safe.

You can find a completed, commented, Apache wsgi configuration file in:

```
scripts/web2py-wsgi.conf
```

This section was created with help from Graham Dumpleton, developer of mod_wsgi.

mod_wsgi and SSL

To force some applications (for example **admin** and **appadmin**) to go over HTTPS, store the SSL certificate and key files:

```
/etc/apache2/ssl/server.crt
/etc/apache2/ssl/server.key
```

and edit the Apache configuration file `web2py.conf` and append:

```
<VirtualHost *:443>
  ServerName web2py.example.com
  SSLEngine on
  SSLCertificateFile /etc/apache2/ssl/server.crt
  SSLCertificateKeyFile /etc/apache2/ssl/server.key

  WSGIProcessGroup web2py

  WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py

  <Directory /users/www-data/web2py>
    AllowOverride None
    Order Allow,Deny
    Deny from all
    <Files wsgihandler.py>
      Allow from all
    </Files>
  </Directory>

  AliasMatch ^/([^/]+)/static/(.*) /users/www-data/web2py/
      applications/$1/static/$2

  <Directory /users/www-data/web2py/applications/*/static/>
    Order Allow,Deny
    Allow from all
  </Directory>

  CustomLog /private/var/log/apache2/access.log common
  ErrorLog /private/var/log/apache2/error.log

</VirtualHost>
```

Restart Apache and you should be able to access:

```
https://www.example.com/admin
https://www.example.com/examples/appadmin
http://www.example.com/examples
```

but not:

```
http://www.example.com/admin
http://www.example.com/examples/appadmin
```

11.3 Setup mod_proxy on Linux

Some Unix/Linux distributions can run Apache, but do not support mod_wsgi. In this case, the simplest solution is to run Apache as a proxy and have Apache deal with static files only.

Here is a minimalist Apache configuration:

```
NameVirtualHost *:80
### deal with requests on port 80
```

```
<VirtualHost *:80>
   Alias / /users/www-data/web2py/applications
   ### serve static files directly
   <LocationMatch "^/welcome/static/.*">
    Order Allow, Deny
    Allow from all
   </LocationMatch>
   ### proxy all the other requests
   <Location "/welcome">
     Order deny,allow
     Allow from all
     ProxyPass http://localhost:8000/welcome
     ProxyPassReverse http://localhost:8000/
   </Location>
   LogFormat "%h %l %u %t "%r" %>s %b" common
   CustomLog /var/log/apache2/access.log common
</VirtualHost>
```

The above script exposes only the "welcome" application. To expose other applications, you need to add the corresponding <Location>...</Location> with the same syntax as done for the "welcome" app.

The script assumes there is a WEB2PY server running on port 8000. Before restarting Apache, make sure this is the case:

```
nohup python web2py.py -a '<recycle>' -i 127.0.0.1 -p 8000 &
```

You can specify a password with the `-a` option or use the "<recycle>" parameter instead of a password. In the latter case, the previously stored password is reused and the password is not stored in the shell history.

You can also use the parameter "<ask>", to be prompted for a password.

The `nohup` commands makes sure the server does not die when you close the shell. `nohup` logs all output into `nohup.out`.

To force admin and appadmin over HTTPS use the following Apache configuration file instead:

```
NameVirtualHost *:80
NameVirtualHost *:443
### deal with requests on port 80
<VirtualHost *:80>
   Alias / /usres/www-data/web2py/applications
   ### admin requires SSL
   <LocationMatch "^/admin">
     SSLRequireSSL
   </LocationMatch>
   ### appadmin requires SSL
   <LocationMatch "^/welcome/appadmin/.*">
     SSLRequireSSL
   </LocationMatch>
   ### serve static files directly
   <LocationMatch "^/welcome/static/.*">
     Order Allow,Deny
     Allow from all
   </LocationMatch>
```

```
   ### proxy all the other requests
   <Location "/welcome">
     Order deny,allow
     Allow from all
     ProxyPass http://localhost:8000/welcome
     ProxyPassReverse http://localhost: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://localhost:8000/
     ProxyPassReverse http://localhost:8000/
   </Location>
   LogFormat "%h %l %u %t \"%r\" %>s %b" common
   CustomLog /var/log/apache2/access.log common
</VirtualHost>
```

The administrative interface must be disabled when WEB2PY *runs on a shared host with mod_proxy, or it will be exposed to other users.*

11.4 Start as Linux Daemon

Unless you are using mod_wsgi, you should setup the WEB2PY server so that it can be started/stopped/restarted as any other Linux daemon, and so it can start automatically at the computer boot stage.

The process to set this up is specific to various Linux/Unix distributions.

In the WEB2PY folder, there are two scripts which can be used for this purpose:

```
scripts/web2py.ubuntu.sh
scripts/web2py.fedora.sh
```

On Ubuntu and other Debian-based Linux distributions, edit the script "web2py.ubuntu.sh" and replace the "/usr/lib/web2py" path with the path of your WEB2PY installation, then type the following shell commands to move the file into the proper folder, register it as a startup service, and start it:

```
sudo cp scripts/web2py.ubuntu.sh /etc/init.d/web2py
sudo update-rc.d web2py defaults
sudo /etc/init.d/web2py start
```

On Fedora and other distributions based on Red Hat, edit the script "web2py.fedora.sh" and replace the "/usr/lib/web2py" path with the path of your WEB2PY installation, then type the following shell commands to move the file into the proper folder, register it as a startup service and start it:

```
sudo cp scripts/web2py.fedora.sh /etc/rc.d/init.d/web2pyd
sudo chkconfig --add web2pyd
sudo service web2py start
```

11.5 Setup Apache and mod_wsgi on Windows

Installing Apache, and mod_wsgi under Windows requires a different procedure. Here are assuming Python 2.5 is installed, you are running from source and WEB2PY is located at `c:/web2py`.

First download the requires packages:

- Apache `apache_2.2.11-win32-x86-openssl-0.9.8i.msi` from

```
http://httpd.apache.org/download.cgi
```

- mod_wsgi from

```
http://adal.chiriliuc.com/mod_wsgi/revision_1018_2.3/
    mod_wsgi_py25_apache22/mod_wsgi.so
```

Second, run `apache...msi` and follow the wizard screens. On the server information screen

enter all requested values:

- **Network Domain**: enter the DNS domain in which your server is or will be registered in. For example, if your server's full DNS name is server.mydomain.net, you would type mydomain.net here
- **ServerName**: Your server's full DNS name. From the example above, you would type server.mydomain.net here. Enter a fully qualified domain name or IP address from the WEB2PY install, not a shortcut, for more information see `http://httpd.apache.org/docs/2.2/mod/core.html`.
- **Administrator's Email Address**. Enter the server administrator's or webmaster's email address here. This address will be displayed along with error messages to the client by default.

Continue with a typical install to the end unless otherwise required

The wizard, by default, installed Apache in the folder:

```
C:/Program Files/Apache Software Foundation/Apache2.2/
```

From now on we refer to this folder simply as `Apache2.2`.

Third, copy the downloaded mod_wsgi.so to `Apache2.2/modules`

The following information about SSL certificates was found in

```
http://port25.technet.com/videos/images/
    TechnicalAnalysisInstallingApacheonWindo_C21A/
    InstallingApacheonWindows.pdf
```

written by Chris Travers, published by the Open Source Software Lab at Microsoft, December 2007.

Fourth, create and place the `server.crt` and `server.key` certificates (as created in the previous section) into `Apache2.2/conf`. Notice the cnf file is in `Apache2.2/conf/openssl.cnf`.

Fifth, edit `Apache2.2/conf/httpd.conf`, remove the comment mark (the # character) from the line

```
LoadModule ssl_module modules/mod_ssl.so
```

add the following line after all the other LoadModule lines

```
LoadModule wsgi_module modules/mod_wsgi.so
```

look for "Listen 80" and add this line after it

```
Listen 443
```

append the following lines at the end changing drive letter, port number, ServerName according to your values

```
NameVirtualHost *:443
<VirtualHost *:443>
  DocumentRoot "C:/web2py/applications"
```

```
  ServerName server1

  <Directory "C:/web2py">
    Order allow,deny
    Deny from all
  </Directory>

  <Location "/">
    Order deny,allow
    Allow from all
  </Location>

  <LocationMatch "^(/[\w_]*/static/.*)">
    Order Allow,Deny
    Allow from all
  </LocationMatch>

  WSGIScriptAlias / "C:/web2py/wsgihandler.py"

  SSLEngine On
  SSLCertificateFile conf/server.crt
  SSLCertificateKeyFile conf/server.key

  LogFormat "%h %l %u %t \"%r\" %>s %b" common
  CustomLog logs/access.log common
</VirtualHost>
```

Save and check the config using: [Start > Program > Apache HTTP Server 2.2 > Configure Apache Server > Test Configuration]

If there are no problems you will see a command screen open and close. Now you can start Apache:

[Start > Program > Apache HTTP Server 2.2 > Control Apache Server > Start]

or better yet start the taskbar monitor

[Start > Program > Apache HTTP Server 2.2 > Control Apache Server]

Now you can right click on the red feather like taskbar icon to Open Apache Monitor and from it start, stop and restart Apache as required.

This section was created by Jonathan Lundell.

11.6 Start as Windows Service

What Linux calls a daemon, Windows calls a service. The WEB2PY server can easily be installed/started/stopped as a Windows service.

In order to use WEB2PY as a Windows service, you must create a file "options.py" with startup parameters:

```
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()
```

You don't need to create "options.py" from scratch since there is already an "options_std.py" in the WEB2PY folder that you can use as a model.

After creating "options.py" in the WEB2PY installation folder, you can install WEB2PY as a service with:

```
python web2py.py -W install
```

and start/stop the service with:

```
python web2py.py -W start
python web2py.py -W stop
```

11.7 Setup Lighttpd

You can install Lighttpd on a Ubuntu or other Debian-based Linux distribution with the following shell command:

```
apt-get -y install lighttpd
```

Once installed, you need to edit the Lighttpd configuration file:

```
/etc/lighttpd/lighttpd.conf
```

and, in it, write something like:

```
  server.port = 80
  server.bind = "0.0.0.0"
  server.event-handler = "freebsd-kqueue"
  server.modules = ( "mod_rewrite", "mod_fastcgi" )
  server.error-handler-404 = "/test.fcgi"
  server.document-root = "/users/www-data/web2py/"
  server.errorlog      = "/tmp/error.log"
  fastcgi.server =    ( ".fcgi" =>
                        ( "localhost" =>
                            ( "min-procs" => 1,
                              "socket"    => "/tmp/fcgi.sock"
                            )
                        )
                    )
```

Start the WEB2PY fcgihandler before the web-server is started, with:

```
nohup python fcgihandler.py &
```

Then, (re)start the web server with:

```
/etc/init.d/lighttpd restart
```

Notice that FastCGI binds the WEB2PY server to a Unix socket, not to an IP socket:

```
/tmp/fcgi.sock
```

This is where Lighttpd forwards the HTTP requests to and receives responses from. Unix sockets are lighter than Internet sockets, and this is one of the reasons Lighttpd+FastCGI+web2py is fast. As in the case of Apache, it is possible to setup Lighttpd to deal with static files directly, and to force some applications over HTTPS. Refer to the Lighttpd documentation for details.

> *The administrative interface must be disabled when* WEB2PY *runs on a shared host with FastCGI, or it will be exposed to the other users.*

11.8 Apache2 and mod_python in a shared hosting environment

There are times, specifically on shared hosts, when one does not have the permission to configure the Apache config files directly. You can still run WEB2PY. Here we show an example of how to set it up using mod_python[6]

- Place contents of WEB2PY into the "htdocs" folder.
- In the WEB2PY folder, create a file "web2py_modpython.py" file with the following contents:

```
    from mod_python import apache
    import modpythonhandler

    def handler(req):
        req.subprocess_env['PATH_INFO'] = \
            req.subprocess_env['SCRIPT_URL']
        return modpythonhandler.handler(req)
```

- Create/update the file ".htaccess" with the following contents:

```
    SetHandler python-program
    PythonHandler web2py_modpython
    ##PythonDebug On
```

[6]Examples provided by Niktar

11.9 Setup Cherokee with FastGGI

Cherokee is a very fast web server and, like WEB2PY, it provides an AJAX-enabled web-based interface for its configuration. Its web interface is written in Python. In addition, there is no restart required for most of the changes.

Here are the steps required to setup WEB2PY with Cherokee:

- Download Cherokee [76]
- Untar, build, and install:

```
tar -xzf cherokee-0.9.4.tar.gz
cd cherokee-0.9.4
./configure --enable-fcgi && make
make install
```

- Start WEB2PY normally at least once to make sure it creates the "applications" folder.
- Write a shell script named "startweb2py.sh" with the following code:

```
#!/bin/bash
cd /var/web2py
python /var/web2py/fcgihandler.py &
```

  and give the script execute privileges and run it. This will start WEB2PY under FastCGI handler.

- Start Cherokee and cherokee-admin:

```
sudo nohup cherokee &
sudo nohup cherokee-admin &
```

  By default, cherokee-admin only listens at local interface on port 9090. This is not a problem if you have full, physical access on that machine. If this is not the case, you can force it to bind to an IP address and port by using the following options:

```
-b,  --bind[=IP]
-p,  --port=NUM
```

  or do an SSH port-forward (more secure, recommended):

```
ssh -L 9090:localhost:9090 remotehost
```

- Open "http://localhost:9090" in your browser. If everything is ok, you will get cherokee-admin.
- In cherokee-admin web interface, click "info sources". Choose "Local Interpreter". Write in the following code, then click "Add New".

```
Nick: web2py
Connection: /tmp/fcgi.sock
Interpreter: /var/web2py/startweb2py.sh
```

- Click "Virtual Servers", then click "Default".
- Click "Behavior", then, under that, click "default".
- Choose "FastCGI" instead of "List and Send" from the list box.
- At the bottom, select "web2py" as "Application Server"
- Put a check in all the checkboxes (you can leave Allow-x-sendfile). If there is a warning displayed, disable and enable one of the checkboxes. (It will automatically re-submit the application server parameter. Sometimes it doesn't, which is a bug).
- Point your browser to "http://ipaddressofyoursite", and "Welcome to web2py" will appear.

11.10 Setup PostgreSQL

PostgreSQL is a free and open source database which is used in demanding production environments, for example, to store the .org domain name database, and has been proven to scale well into hundreds of terabytes of data. It has very fast and solid transaction support, and provides an autovacuum feature that frees the administrator from most database maintenance tasks.

On an Ubuntu or other Debian-based Linux distribution, it is easy to install PostgreSQL and its Python API with:

```
sudo apt-get -y install postgresql
sudo apt-get -y install python-psycopg2
```

It is wise to run the web server(s) and the database server on different machines. In this case, the machines running the web servers should be connected with a secure internal (physical) network, or should establish SSL tunnels to securely connect with the database server.

Start the database server with:

```
sudo /etc/init.d/postgresql restart
```

When restarting the PostgreSQL server, it should notify which port it is running on. Unless you have multiple database servers, it should be 5432.

The PostgreSQL configuration file is:

```
/etc/postgresql/x.x/main/postgresql.conf
```

(where `x.x` is the version number).

The PostgreSQL logs are in:

```
/var/log/postgresql/
```

Once the database server is up and running, create a user and a database so that WEB2PY applications can use it:

```
sudo -u postgres createuser -P -s myuser
createdb mydb
echo 'The following databases have been created:'
psql -l
psql mydb
```

The first of the commands will grant superuser-access to the new user, called `myuser`. It will prompt you for a password.

Any WEB2PY application can connect to this database with the command:

```
db = DAL("postgres://myuser:mypassword@localhost:5432/mydb")
```

where `mypassword` is the password you entered when prompted, and 5432 is the port where the database server is running.

Normally you use one database for each application, and multiple instances of the same application connect to the same database. It is also possible for different applications to share the same database.

For database backup details, read the PostgreSQL documentation; specifically the commands `pg_dump` and `pg_restore`.

11.11 Security Issues

It is very dangerous to publicly expose the **admin** application and the **appadmin** controllers unless they run over HTTPS. Moreover, your password and credentials should never be transmitted unencrypted. This is true for WEB2PY and any other web application.

In your applications, if they require authentication, you should make the session cookies secure with:

```
session.secure()
```

An easy way to setup a secure production environment on a server is to first stop WEB2PY and then remove all the `parameters_*.py` files from the WEB2PY installation folder. Then start WEB2PY without a password. This will completely disable admin and appadmin.

Next, start a second Python instance accessible only from localhost:

```
nohup python web2py -p 8001 -i 127.0.0.1 -a '<ask>' &
```

and create an SSH tunnel from the local machine (the one from which you wish to access the administrative interface) to the server (the one where WEB2PY is running, example.com), using:

```
ssh -L 8001:127.0.0.1:8001 username@example.com
```

Now you can access the administrative interface locally via the web browser at `localhost:8001`.

This configuration is secure because **admin** is not reachable when the tunnel is closed (the user is logged out).

> *This solution is secure on shared hosts if and only if other users do not have read access to the folder that contains* WEB2PY*; otherwise users may be able to steal session cookies directly from the server.*

11.12 Scalability Issues

WEB2PY is designed to be easy to deploy and to setup. This does not mean that it compromises on efficiency or scalability, but it means you may need to tweak it to make it scalable.

In this section we assume multiple WEB2PY installations behind a NAT server that provides local load-balancing.

In this case, WEB2PY works out-of-the-box if some conditions are met. In particular, all instances of each WEB2PY application must access the same database server and must see the same files. This latter condition can be implemented by making the following folders shared:

```
applications/myapp/sessions
applications/myapp/errors
applications/myapp/uploads
applications/myapp/cache
```

The shared folders must support file locking. Possible solutions are ZFS[7], NFS[8], or Samba (SMB).

It is possible, but not a good idea, to share the entire WEB2PY folder or the entire applications folder, because this would cause a needless increase of network bandwidth usage.

We believe the configuration discussed above to be very scalable because it reduces the database load by moving to the shared filesystems those resources

[7]ZFS was developed by Sun Microsystems and is the preferred choice.
[8]With NFS you may need to run the nlockmgr daemon to allow file locking.

that need to be shared but do not need transactional safety (only one client at a time is supposed to access a session file, cache always needs a global lock, uploads and errors are write once/read many files).

Ideally, both the database and the shared storage should have RAID capability. Do not make the mistake of storing the database on the same storage as the shared folders, or you will create a new bottle neck there.

On a case-by-case basis, you may need to perform additional optimizations and we will discuss them below. In particular, we will discuss how to get rid of these shared folders one-by-one, and how to store the associated data in the database instead. While this is possible, it is not necessarily a good solution. Nevertheless, there may be reasons to do so. One such reason is that sometimes we do not have the freedom to set up shared folders.

Sessions in Database

It is possible to instruct WEB2PY to store sessions in a database instead of in the sessions folder. This has to be done for each individual WEB2PY application although they may all use the same database to store sessions.

Given a database connection

```
db = DAL(...)
```

you can store the sessions in this database (db) by simply stating the following, in the same model file that establishes the connection:

```
session.connect(request, response, db)
```

If it does not exist already, WEB2PY creates a table in the database called `web2py_session_`*appname* containing the following fields:

```
Field('locked', 'boolean', default=False),
Field('client_ip'),
Field('created_datetime', 'datetime', default=now),
Field('modified_datetime', 'datetime'),
Field('unique_key'),
Field('session_data', 'text')
```

"unique_key" is a uuid key used to identify the session in the cookie. "session_data" is the cPickled session data.

To minimize database access, you should avoid storing sessions when they are not needed with:

```
session.forget()
```

With this tweak the "sessions" folder does not need to be a shared folder because it will no longer be accessed.

> *Notice that, if sessions are disabled, you must not pass the* `session` *to* `form.accepts` *and you cannot use* `session.flash` *nor CRUD.*

Pound, a High Availability Load Balancer

If you need multiple WEB2PY processes running on multiple machines, instead of storing sessions in the database or in cache, you have the option to use a load balancer with sticky sessions.

Pound [78] is an HTTP load balancer and Reverse proxy that provides sticky sessions.

By sticky sessions, we mean that once a session cookie has been issued, the load balancer will always route requests from the client associated to the session, to the same server. This allows you to store the session in the local filesystem.

To use Pound:

First, install Pound, on out Ubuntu test machine:

```
sudo apt-get -y install pound
```

Second edit the configuration file "/etc/pound/pound.cfg" and enable Pound at startup:

```
startup=1
```

Bind it to a socket (IP, Port):

```
ListenHTTP 123.123.123.123,80
```

Specify the IP addresses and ports of the machines in the farm running WEB2PY:

```
UrlGroup ".*"
BackEnd 192.168.1.1,80,1
BackEnd 192.168.1.2,80,1
BackEnd 192.168.1.3,80,1
Session IP 3600
EndGroup
```

The ",1" indicates the relative strength of the machines. The last line will maintain sessions by client IP for 3600 seconds.

Third, enable this config file and start Pound:

```
/etc/default/pound
```

Cleanup Sessions

If you choose to keep your sessions in the filesystem, you should be aware that on a production environment they pile up fast. WEB2PY provides a script called:

```
scripts/sessions2trash.py
```

that when run in the background, periodically deletes all sessions that have not been accessed for a certain amount of time. This is the content of the script:

```
SLEEP_MINUTES = 5
EXPIRATION_MINUTES = 60
import os, time, stat
path = os.path.join(request.folder, 'sessions')
while 1:
   now = time.time()
   for file in os.listdir(path):
      filename = os.path.join(path, file)
      t = os.stat(filename)[stat.ST_MTIME]
      if now - t > EXPIRATION_MINUTES * 60:
         unlink(filename)
   time.sleep(SLEEP_MINUTES * 60)
```

You can run the script with the following command:

```
nohup python web2py.py -S yourapp -R scripts/sessions2trash.py &
```

where yourapp is the name of your application.

Upload Files in Database

By default, all uploaded files handled by SQLFORMs are safely renamed and stored in the filesystem under the "uploads" folder. It is possible to instruct WEB2PY to store uploaded files in the database instead.

Consider the following table:

```
db.define_table('dog',
    Field('name')
    Field('image', 'upload'))
```

where `dog.image` is of type upload. To make the uploaded image go in the same record as the name of the dog, you must modify the table definition by adding a blob field and link it to the upload field:

```
db.define_table('dog',
    Field('name')
    Field('image', 'upload', uploadfield='image_data'),
    Field('image_data', 'blob'))
```

Here "image_data" is just an arbitrary name for the new blob field.

Line 3 instructs WEB2PY to safely rename uploaded images as usual, store the new name in the image field, and store the data in the uploadfield called "image_data" instead of storing the data on the filesystem. All of this is be done automatically by SQLFORMs and no other code needs to be changed.

With this tweak, the "uploads" folder is no longer needed.

No Google App Engine files are stored by default in the database without need to define an uploadfield, one is created by default.

Collecting Tickets

By default, WEB2PY stores tickets (errors) on the local file system. It would not make sense to store tickets directly in the database, because the most common origin of error in a production environment is database failure.

Storing tickets is never a bottleneck, because this is ordinarily a rare event, hence, in a production environment with multiple concurrent servers, it is more than adequate to store them in a shared folder. Nevertheless, since only the administrator needs to retrieve tickets, it is also OK to store tickets in a non-shared local "errors" folder and periodically collect them and/or clear them.

One possibility is to periodically move all local tickets to a database.

For this purpose, WEB2PY provides the following script:

```
scripts/tickets2db.py
```

which contains:

```
import sys
import os
import time
import stat
import datetime

from gluon.utils import md5_hash
from gluon.restricted import RestrictedError

SLEEP_MINUTES = 5
DB_URI = 'sqlite://tickets.db'
ALLOW_DUPLICATES = True

path = os.path.join(request.folder, 'errors')

db = SQLDB(DB_URI)
db.define_table('ticket', SQLField('app'), SQLField('name'),
                SQLField('date_saved', 'datetime'), SQLField('layer')
                    ,
                SQLField('traceback', 'text'), SQLField('code', 'text
                    '))

hashes = {}

while 1:
    for file in os.listdir(path):
        filename = os.path.join(path, file)

        if not ALLOW_DUPLICATES:
            file_data = open(filename, 'r').read()
            key = md5_hash(file_data)

            if key in hashes:
                continue

            hashes[key] = 1
```

```
        error = RestrictedError()
        error.load(request, request.application, filename)

        modified_time = os.stat(filename)[stat.ST_MTIME]
        modified_time = datetime.datetime.fromtimestamp(modified_time
            )

        db.ticket.insert(app=request.application,
                         date_saved=modified_time,
                         name=file,
                         layer=error.layer,
                         traceback=error.traceback,
                         code=error.code)

        os.unlink(filename)

    db.commit()
    time.sleep(SLEEP_MINUTES * 60)
```

This script should be edited. Change the DB_URI string so that it connects to your database server and run it with the command:

```
nohup python web2py.py -S yourapp -M -R scripts/tickets2db.py &
```

where yourapp is the name of your application.

This script runs in the background and every 5 minutes moves all tickets to the database server in a table called "ticket" and removes the local tickets. If ALLOW_DUPLICATES is set to False, it will only store tickets that correspond to different types of errors. With this tweak, the "errors" folder does not need to be a shared folder any more, since it will only be accessed locally.

Memcache

We have shown that WEB2PY provides two types of cache: `cache.ram` and `cache.disk`. They both work on a distributed environment with multiple concurrent servers, but they do not work as expected. In particular, `cache.ram` will only cache at the server level; thus it becomes useless. `cache.disk` will also cache at the server level unless the "cache" folder is a shared folder that supports locking; thus, instead of speeding things up, it becomes a major bottleneck.

The solution is not to use them, but to use memcache instead. WEB2PY comes with a memcache API.

To use memcache, create a new model file, for example `0_memcache.py`, and in this file write (or append) the following code:

```
from gluon.contrib.memcache import MemcacheClient
memcache_servers = ['127.0.0.1:11211']
cache.memcache = MemcacheClient(request, memcache_servers)
```

```
4 cache.ram = cache.disk = cache.memcache
```

The first line imports memcache. The second line has to be a list of memcache sockets (server:port). The third line redefines `cache.ram` and `cache.disk` in terms of memcache.

You could choose to redefine only one of them to define a totally new cache object pointing to the Memcache object.

With this tweak the "cache" folder does not need to be a shared folder any more, since it will no longer be accessed.

This code requires having memcache servers running on the local network. You should consult the memcache documentation for information on how to setup those servers.

Sessions in Memcache

If you do need sessions and you do not want to use a load balancer with sticky sessions, you have the option to store sessions in memcache:

```
1 from gluon.contrib.memdb import MEMDB
2 session.connect(request,response,db=MEMDB(cache.memcache))
```

Removing Applications

In a production setting, it may be better not to install the default applications: **admin**, **examples** and **welcome**. Although these applications are quite small, they are not necessary.

Removing these applications is as easy as deleting the corresponding folders under the applications folder.

11.13 Google App Engine

It is possible to run WEB2PY code on Google App Engine (GAE) [12], including DAL code, with some limitations. The GAE platform provides several advantages over normal hosting solutions:

- Ease of deployment. Google completely abstracts the underlying architecture.
- Scalability. Google will replicate your app as many times as it takes to serve all concurrent requests

- BigTable. On GAE, instead of a normal relational database, you store persistent information in BigTable, the datastore Google is famous for.

The limitations are:

- You have no read or write access to the file system.
- No transactions
- You cannot perform complex queries on the datastore, in particular there are no JOIN, OR, LIKE, IN, and DATE/DATETIME operators.

This means that WEB2PY cannot stores sessions, error tickets, cache files and uploaded files on disk; they must be stored somewhere else. Therefore, on GAE, WEB2PY automatically stores all uploaded files in the datastore, whether or not "upload" Field(s) have a `uploadfield` attribute. You have to be explicit about where to store sessions and tickets:

You can store them in the datastore too:

```
db = DAL('gae')
session.connect(request,response,db)
```

Or, you can store them in memcache:

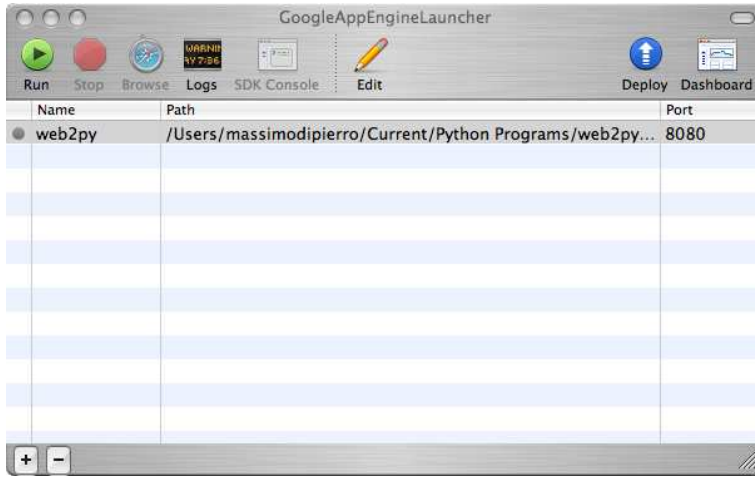
```
from gluon.contrib.gae_memcache import MemcacheClient
from gluon.contrib.memdb import MEMDB
cache.memcache = MemcacheClient(request)
cache.ram = cache.disk = cache.memcache

db = DAL('gae')
session.connect(request,response,MEMDB(cache.memcache))
```

The absence of transactions and typical functionalities of relational databases are what sets GAE apart from other hosting environment. This is the price to pay for high scalability. If you can leave with these limitations, then GAE is an excellent platform. If you cannot, then you should consider a regular hosting platform with a relational database.

If a WEB2PY application does not run on GAE, it is because of one of the limitations discussed above. Most issues can be resolved by removing JOINs from WEB2PY queries and denormalizing the database.

To upload your app in GA,E we recommend using the Google App Engine Launcher. You can download the software from ref. [12].

Choose [File][Add Existing Application], set the path to the path of the top-level WEB2PY folder, and press the [Run] button in the toolbar. After you have tested that it works locally, you can deploy it on GAE by simply clicking on the [Deploy] button on the toolbar (assuming you have an account).

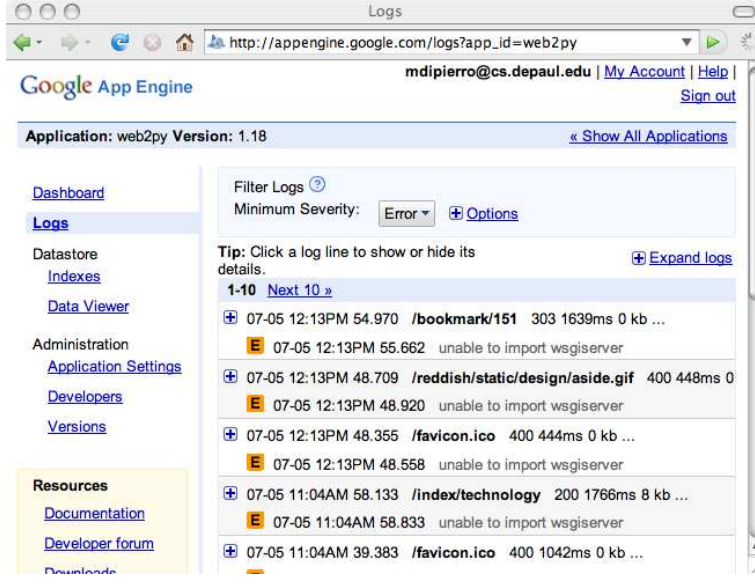


On Windows and Linux systems, you can also deploy using the shell:

```
cd ..
/usr/local/bin/dev_appserver.py web2py
```

When deploying, WEB2PY ignores the **admin**, **examples**, and **welcome** applications since they are not needed. You may want to edit the `app.yaml` file and ignore other applications as well.

On GAE, the WEB2PY tickets/errors are also logged into the GAE administration console where logs can be accessed and searched online.



You can detect whether WEB2PY is running on GAE using the variable

```
request.env.web2py_runtime_gae
```

CHAPTER 12

OTHER RECIPES

12.1 Upgrading WEB2PY

In the near future WEB2PY will be able to upgrade itself but this has not yet been implemented at the time of publishing.

Upgrading WEB2PY manually is very easy.

> *Simply unzip the latest version of* WEB2PY *over the old installation.*

This will upgrade all the libraries but none of the applications, not even the standard applications (**admin**, **examples**, **welcome**), because you may have changed them and WEB2PY does not want to mess with them. The new standard applications will be in the corresponding .w2p files in the WEB2PY root folder. After the upgrade, the new "welcome.w2p" will be used as a scaffolding application.

You can upgrade the existing standard applications with the shell command:

```
1 python web2py.py --upgrade yes
```

This will upgrade **admin**, **example**, and **welcome**.

12.2 Fetching a URL

Python includes the `urllib` library for fetching urls:

```
import urllib
page = urllib.urlopen('http://www.web2py.com').read()
```

This is often fine, but the `urllib` module does not work on the Google App Engine. Google provides a different API for downloading URL that works on GAE only. In order to make your code portable, WEB2PY includes a `fetch` function that works on GAE as well as other Python installations:

```
from google.tools import fetch
page = fetch('http://www.web2py.com')
```

12.3 Geocoding

If you need to convert an address (for example: "243 S Wabash Ave, Chicago, IL, USA") into geographical coordinates (latitude and longitude), WEB2PY provides a function to do so.

```
from gluon.tools import geocode
address = '243 S Wabash Ave, Chicago, IL, USA'
(latitude, longitude) = geocode(address)
```

The function `geocode` requires a network connection and it connect to the Google geocoding service for the geocoding. The function returns `(0,0)` in case of failure. Notice that the Google geocoding service caps the number of requests and you should check their service agreement. The `geocode` function is built on top of the `fetch` function and thus it works on GAE.

12.4 Pagination

This recipe is a useful trick to minimize database access in case of pagination, e.g., when you need to display a list of rows from a database but you want to distribute the rows over multiple pages.

Start by creating a **primes** application that stores the first 1000 prime numbers in a database.

Here is the model `db.py`:

```
db=DAL('sqlite://primes.db')
db.define_table('prime',Field('value','integer'))
```

```
def isprime(p):
    for i in range(2,p):
        if p%i==0: return False
    return True
if len(db().select(db.prime.id))==0:
   p=2
   for i in range(1000):
       while not isprime(p): p+=1
       db.prime.insert(value=p)
       p+=1
```

Now create an action `list_items` in the "default.py" controller that reads like this:

```
def list_items():
    if len(request.args): page=int(request.args[0])
    else: page=0
    items_per_page=20
    limitby=(page*items_per_page,(page+1)*items_per_page+1)
    rows=db().select(db.prime.ALL,limitby=limitby)
    return dict(rows=rows,page=page,items_per_page=items_per_page)
```

Notice that this code selects one more item than is needed, $20 + 1$. The reason is that the extra element tells the view whether there is a next page.

Here is the "default/list_items.html" view:

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

{{for i,row in enumerate(rows):}}
{{if i==items_per_page: break}}
{{=row.value}}<br />
{{pass}}

{{if page:}}
<a href="{{=URL(r=request,args=[page-1])}}">previous</a>
{{pass}}

{{if len(rows)>items_per_page:}}
<a href="{{=URL(r=request,args=[page+1])}}">next</a>
{{pass}}
```

In this way we have obtained pagination with one single select per action, and that one select only selects one row more then we need.

12.5 Streaming Virtual Files

It is common for malicious attackers to scan web sites for vulnerabilities. They use security scanners like Nessus to explore the target web sites for scripts that are known to have vulnerabilities. An analysis of web server logs from a scanned machine or directly of the Nessus database reveals that most of the known vulnerabilities are in PHP scripts and ASP scripts. Since

we are running WEB2PY, we do not have those vulnerabilities, but we will still be scanned for them. This annoying, so we like to like to respond to those vulnerability scans and make the attacker understand their time is being wasted.

One possibility is to redirect all requests for .php, .asp, and anything suspicious to a dummy action that will respond to the attack by keeping the attacker busy for a large amount of time. Eventually the attacker will give up and will not scan us again.

This recipe requires two parts.

A dedicated application called **jammer** with a "default.py" controller as follows:

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

When this action is called, it responds with an infinite data stream full of "x"-es. 40000 characters at a time.

The second ingredient is a "route.py" file that redirects any request ending in .php, .asp, etc. (both upper case and lower case) to this controller.

```
route_in=(
 ('.*\.(php|PHP|asp|ASP|jsp|JSP)','jammer/default/jam'),
)
```

The first time you are attacked you may incur a small overhead, but our experience is that the same attacker will not try twice.

12.6 httpserver.log and the log file format

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 one of the HTTP status codes [80]
- **time_taken** is the amount of time the server took to process the request, in seconds, not including upload/download time.

In the appliances repository[33], you will find an appliance for log analysis.

This logging is disabled by default when using mod_wsgi since it would be the same as the Apache log.

12.7 Send an SMS

Sending SMS messages from a WEB2PY application requires a third party service that can relay the messages to the receiver. Usually this is not a free service, but it differs from country to country.

In the US, `aspsms.com` is one of these services. They require signing up for the service and the deposit of an amount of money to cover the cost of the SMS messages that will be sent. They will assign a userkey and a password.

Once you have these parameters you need to define a function that can send SMS messages through the service. For this purpose you can define a model file in the application called "0_sms.py" and in this file include the following code:

```
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
```

You can call the function from any controller in the application.

Notice that the service is ASP-based, but it communicates via XML, so you can call it from a Python/WEB2PY program.

12.8 Twitter API

Here are some quick examples on how to post/get tweets. No third-party libraries are required, since Twitter uses simple RESTful APIs.

Here is an example of how to post a tweet:

```
def post_tweet(username,password,message):
    import urllib, urlib2, base64
    import gluon.contrib.simplejson as sj
    args= urllib.urlencode([('status',message)])
    headers={}
    headers['Authorization'] = 'Basic '+base64.b64encode(username+':'
        +password)
    request = urllib2.Request('http://twitter.com/statuses/update.
        json', args, headers)
    return  sj.loads(urllib2.urlopen(req).read())
```

Here is an example of how to receive tweets:

```
def get_tweets():
    user='web2py'
    import urllib
    import gluon.contrib.simplejson as sj
    page = urllib.urlopen('http://twitter.com/%s?format=json' % user)
        .read()
    tweets=XML(sj.loads(page)['#timeline'])
    return dict(tweets=tweets)
```

For more complex operations, refer to the Twitter API documentation.

12.9 Jython

WEB2PY normally runs on CPython (the Python interpreter coded in C), but it can also run on Jython (the Python interpreter coded in Java). This allows WEB2PY to run in a Java infrastructure.

Even though WEB2PY runs with Jython out of the box, there is some trickery involved in setting up Jython and in setting up zxJDBC (the Jython database adaptor). Here are the instructions:

- Download the file "jython_installer-2.5.0.jar" (or 2.5.x) from `Jython.org`
- Install it:

```
java -jar jython_installer-2.5.0.jar
```

- Download and install "zxJDBC.jar" from
  `http://sourceforge.net/projects/zxjdbc/`
- Download and install the file "sqlitejdbc-v056.jar" from
  `http://www.zentus.com/sqlitejdbc/`
- Add zxJDBC and sqlitejdbc to the java CLASSPATH
- Start WEB2PY with Jython

```
/path/to/jython web2py.py
```

You will be able to use `DAL('sqlite://...')` and `DAL('postgres://...')` only.

References

1. http://www.web2py.com
2. http://www.python.org
3. http://en.wikipedia.org/wiki/SQL
4. http://www.sqlite.org/
5. http://www.postgresql.org/
6. http://www.mysql.com/
7. http://www.microsoft.com/sqlserver
8. http://www.firebirdsql.org/
9. http://www.oracle.com/database/index.html
10. http://www-01.ibm.com/software/data/db2/
11. http://www-01.ibm.com/software/data/informix/
12. http://code.google.com/appengine/
13. http://en.wikipedia.org/wiki/HTML
14. http://www.w3.org/TR/REC-html40/
15. http://www.php.net/
16. http://www.cherrypy.org/browser/trunk/cherrypy/wsgiserver/__init__.py
17. http://en.wikipedia.org/wiki/Web_Server_Gateway_Interface

18. http://www.python.org/dev/peps/pep-0333/
19. http://www.owasp.org
20. http://en.wikipedia.org/wiki/Secure_Sockets_Layer
21. http://www.cherrypy.org
22. http://www.cdolivet.net/editarea/
23. http://nicedit.com/
24. http://pypi.python.org/pypi/simplejson
25. http://pyrtf.sourceforge.net/
26. http://www.dalkescientific.com/Python/PyRSS2Gen.html
27. http://www.feedparser.org/
28. http://code.google.com/p/python-markdown2/
29. http://www.tummy.com/Community/software/python-memcached/
30. http://www.fsf.org/licensing/licenses/info/GPLv2.html
31. http://jquery.com/
32. https://www.web2py.com/cas
33. http://www.web2py.com/appliances
34. http://www.web2py.com/AlterEgo
35. http://www.python.org/dev/peps/pep-0008/
36. Guido van Rossum, and Fred L. Drake, *An Introduction to Python (version 2.5)*, Network Theory Ltd, 164 pages (November 2006)
37. Mark Lutz, *Learning Python*, O'Reilly & Associates, 701 pages (October 2007)
38. http://www.python.org/doc/
39. http://en.wikipedia.org/wiki/Cascading_Style_Sheets
40. http://www.w3.org/Style/CSS/
41. http://www.w3schools.com/css/
42. http://en.wikipedia.org/wiki/JavaScript
43. David Flanagan, *JavaScript: The Definitive Guide JavaScript: The Definitive Guide*, O'Reilly Media, Inc.; 5 edition (August 17, 2006)
44. http://www.xmlrpc.com/
45. http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
46. http://www.w3.org/Protocols/rfc2616/rfc2616.html
47. http://en.wikipedia.org/wiki/XML
48. http://www.w3.org/XML/
49. http://en.wikipedia.org/wiki/XHTML
50. http://www.w3.org/TR/xhtml1/
51. http://www.w3schools.com/xhtml/
52. http://www.web2py.com/layouts

53. http://sourceforge.net/projects/zxjdbc/
54. http://pypi.python.org/pypi/psycopg2
55. http://sourceforge.net/projects/mysql-python
56. http://python.net/crew/atuining/cx_Oracle/
57. http://pyodbc.sourceforge.net/
58. http://kinterbasdb.sourceforge.net/
59. http://informixdb.sourceforge.net/
60. http://www.web2py.com/sqldesigner
61. http://www.faqs.org/rfcs/rfc2616.html
62. http://www.faqs.org/rfcs/rfc2396.html
63. http://tools.ietf.org/html/rfc3490
64. http://tools.ietf.org/html/rfc3492
65. http://www.recaptcha.net
66. http://www.reportlab.org
67. http://en.wikipedia.org/wiki/AJAX
68. Karl Swedberg and Jonathan Chaffer, *Learning jQuery*, Packt Publishing
69. http://ui.jquery.com/
70. http://en.wikipedia.org/wiki/Common_Gateway_Interface
71. http://www.apache.org/
72. http://httpd.apache.org/docs/2.0/mod/mod_proxy.html
73. http://sial.org/howto/openssl/self-signed
74. http://code.google.com/p/modwsgi/
75. http://www.lighttpd.net/
76. http://www.cherokee-project.com/download/''
77. http://www.fastcgi.com/
78. http://www.apsis.ch/pound/
79. http://pyamf.org/
80. http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

Index

D

E

F

G

H