MASSIMO DI PIERRO

GIOVANNI CANNATA

WEB2PY MANUAL

3RD EDITION

LULU.COM

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:

ISBN-0000

Printed in the United States of America.

To my family

Contents

Preface

1

Introduzione

web2py (1) è un framework open source per lo sviluppo agile di applicazioni web sicure ed incentrate sui dati; è scritto ed è programmabile in Python (2). web2py è un *full-stack* framework, contiene cioè tutti i componenti necessari per costruire applicazioni web pienamente funzionanti. web2py è progettato per aiutare lo sviluppatore a seguire le corrette pratiche di ingegnerizzazione del software, come, ad esempio, l'utilizzo dello schema di sviluppo MVC (Model, View, Controller). web2py separa la fase di rappresentazione dei dati (il modello, *model*) dalla fase di presentazione dei dati stessi (la vista, *view*) a loro volta separate dalla logica e dal flusso dell'applicazione (il controllo, *controller*). web2py fornisce le librerie necessarie ad aiutare lo sviluppatore a progettare, implementare e testare separatamente ognuna di queste tre componenti e ad unirle infine in un'unica applicazione. web2py è progettato per essere sicuro. Questo significa che tiene automaticamente conto di molti dei problemi che possono creare falle nella sicurezza dell'applicazione seguendo pratiche di sviluppo sicuro ben collaudate. Per esempio valida tutti gli input (per prevenire vulnerabilità di tipo *injection*), controlla (*escaping*) tutto l'output (per prevenire vulnerabilità di tipo *cross-site scripting*), rinomina i file caricati dagli utenti (per prevenire attacchi di tipo *directory traversal*). web2py lascia agli sviluppatori ben poche scelte in materia di sicurezza. web2py include uno strato di astrazione del database (DAL, *Directory Ab-*

straction Layer) che genera codice SQL (3) in modo dinamico. Lo sviluppatore non ha necessità di scrivere codice SQL, il DAL è in grado di generare SQL trasparentemente per i seguenti database: SQLite (4), MySQL (6), PostgreSQL (5), MSSQL (7), FireBird (8), Oracle (9), IBM DB2 (10), Informix (11), ed Ingres (12). Il DAL inoltre è in grado di generare codice per utilizzare *Google BigTable* quando il framework viene eseguito nel *Google App Engine* (GAE) (13). Quando vengono definiti una o più tabelle nel database web2py genera anche una completa interfaccia web di amministrazione per accedere al database e alle tabelle. web2py si differenzia dagli altri framework di sviluppo web in quanto è l'unico framework a sposare completamente il paradigma Web 2.0, dove il web è il computer. Infatti web2py non richiede installazione o configurazione. può essere eseguito su ogni piattaforma dove Python esiste (Windows, Windows CE, Mac OS X, iPhone e Unix/Linux) e le diverse fasi di sviluppo, installazione e manutenzione delle applicazioni possono essere eseguite localmente o remotamente tramite una interfaccia web. web2py utilizza CPython (per l'implementazione in C) o Jython (per l'implementazione in Java) nelle versioni 2.4, 2.5 e 2.6, sebbene sia 'ufficialmente' supportata solo la versione 2.5 di Python, altrimenti non potrebbe essere garantita la compatibilità con le applicazioni pre-esistenti. web2py fornisce un sistema di gestione degli errori (*ticketing*). In caso avvenga un errore nell'applicazione viene emesso un *ticket* per l'utente e l'errore è registrato per l'amministratore. web2py è un prodotto open source ed è rilasciato sotto la licenza GPL2.0. Le applicazioni sviluppate con web2py non sono soggette a nessun limite di licenza perchè, se non contengono codice di web2py, non sono considerate *lavori derivati*. web2py consente inoltre allo sviluppatore di creare versioni compilate in *byte-code* e di distribuirle come *closed source*, sebbene richiedano comunque web2py per essere eseguite. La licenza di web2py include una eccezione che permette agli sviluppatore di distribuire i loro prodotti con i binari precompilati di web2py senza il relativo codice sorgente.

Un'altra caratteristica di web2py è che noi, gli sviluppatori del framework, ci siamo impegnati a mantenere la retro-compatibilità nelle future versioni. Questo è stato fatto fin dal primo rilascio di web2py nell'ottobre del 2007. Nuove caratteristiche sono state aggiunte e i bug sono stati corretti, ma se

un programma ha funzionato con la versione 1.0 di web2py quel programma funziona ancora oggi con la versione attuale.

Ecco alcuni esempi del codice utilizzato in web2py che illustrano la sua semplicità e la sua potenza. Il seguente codice:

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

crea una tabella di database chiamata "person" con due campi: "name" (di tipo stringa), ed "image" (di tipo *upload*, un meccanismo che consente di caricare le reali immagini). Se la tabella già esiste ma non corrisponde a questa definizione viene appropriatamente modificata.

Con la tabella appena definita il seguente codice:

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

crea un modulo (*form*) di inserimento per questa tabella che consente all'utente di caricare le immagini. Inoltre valida il form, rinomina in modo sicuro l'immagine caricata, memorizza l'immagine in un file, inserisce il relativo record nel database, previene i doppi invii, e, se la validazione dei dati inseriti dall'utente non è positiva, modifica il form aggiungendo i relativi messaggi d'errore.

Il seguente codice:

```
@auth.requires_permission('read','person')
def f(): ....
```

impedisce agli utenti di accedere alla funzione `f` a meno che siano membri di un gruppo abilitato a "leggere" i record della tabella "person". Se l'utente non è autenticato viene reindirizzato alla pagina di login dell'applicazione (generata automaticamente da web2py).

Il seguente codice inserisce un componente di pagina in una vista:

```
{{=LOAD('other_controller','function.load',ajax=True, ajax_trap=True)}}
```

Questo consente a web2py di caricare in una vista il contenuto generato dalla funzione *other_controller* (può essere usata qualsiasi funzione). Il contenuto viene caricato tramite Ajax, viene inserito nella pagina corrente (utilizzando il layout della pagina corrente e non quello della funzione *other_controller*), vengono intercettate tutte le form nel contenuto caricato in modo che siano inviate tramite Ajax senza ricaricare l'intera pagina. Con questo meccanismo può essere anche caricato altro contenuto da applicazioni non web2py.

La funzione LOAD consente una programmazione estremamente modulare delle applicazioni ed è discussa in maggior dettaglio nell'ultimo capitolo di questo libro.

1.1 Principi

La programmazione in Python segue tipicamente questi principi di base:

- Non ripeterti (DRY, "Don't repeat yourself").
- Deve esserci solo un modo di fare le cose.
- Esplicito è meglio che implicito.

web2py sposa completamente i primi due principi obbligando lo sviluppatore ad utilizzare chiare pratiche di ingegnerizzazione software che scoraggiano la ripetizione del codice. web2py guida lo sviluppatore nella maggior parte dei comuni compiti della fase di sviluppo di una applicazione (creazione e gestione dei form, gestione delle sessioni, dei cookie, degli errori, etc.). Per ciò che riguarda il terzo principio web2py differisce dagli altri framework di sviluppo in quanto questo principio a volte è in conflitto con gli altri due. In particolare web2py non importa l'applicazione ma la esegue in un contesto predefinito. Il contesto espone le parole chiave (*keyword*) di Python insieme alle keyword di web2py stesso.

Ad alcuni questo potrebbe sembrare magico, ma in realtà non dovrebbe. In pratica alcuni moduli sono già importati senza che lo sviluppatore debba esplicitamente farlo. web2py tenta di evitare la fastidiosa caratteristica di altri framework che obbligano lo sviluppatore ad importare sempre gli stessi moduli all'inizio di ogni modello e di ogni controller. Importando automaticamene questi moduli web2py fa risparmiare tempo e previene gli errori, seguendo perciò lo spirito di "Non ripeterti" e di "Deve esserci sol un modo di fare le cose".

Se lo sviluppatore desidera utilizzare altri moduli di Python o di terze parti, quei moduli devono essere importati esplicitamente, come in ogni altro programma scritto in Python.

1.2 *Framework di sviluppo web*

Al suo livello più basso un'applicazione web consiste di un insieme di programmi (o funzioni) che sono eseguite quando la relativa URL è visitata. L'output del programma è ciò che viene restituito al browser che a sua volta lo visualizza all'utente.

L'obiettivo di tutti i framework web è di permettere allo sviluppatore di costruire nuove applicazioni velocemente, con facilità e senza errori. Questo viene fatto tramite *API* e strumenti (*tools*) che riducono e semplificano la quantità di codice necessario.

Le due tipiche modalità di sviluppo delle applicazioni web sono:

- Generazione del codice HTML (14; 15) da programma.
- Inclusione del codice all'interno delle pagine HTML.

Il primo modello è quello seguito, per esempio, dai primi script *CGI* (*Common Gateway Interface*). Il secondo modello è seguito, per esempio, da PHP (16)

(dove il codice è scritto in PHP, un linguaggio simile al C), ASP (dove il codice è in Visual Basic), e JSP (dove il codice è in Java).

Ecco un esempio di un programma PHP che, quando eseguito, recupera dati da un database e ritorna una pagina HTML che mostra i relativi record:

```
<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>
```

Il problema con questo approccio è che il codice è inserito direttamente nel codice HTML, ma lo stesso codice è utilizzato anche per generare codice HTML aggiuntivo e generare i comandi SQL di interrogazione del database. In questo modo i diversi livelli dell'applicazione si "aggrovigliano" e rendono difficile la manutenzione dell'applicazione stessa. La complessità cresce con il numero di pagine (files) che compongono l'applicazione stessa e la situazione peggiora ulteriormente per le applicazioni Ajax.

La stessa funzionalità può essere espressa in web2py con due linee di codice Python:

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

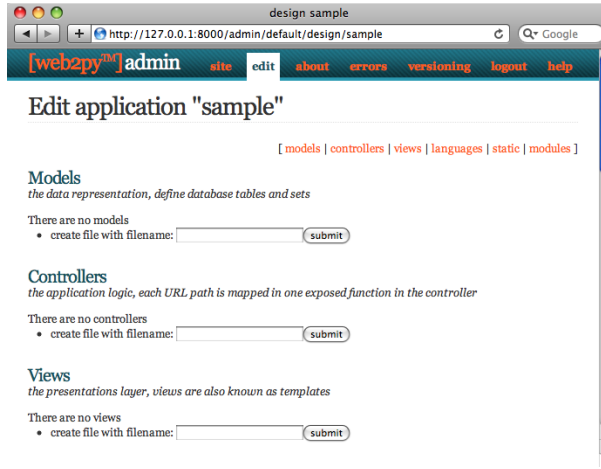
In questo semplice esempio, la struttura della pagina HTML è rappresentata direttamente nel codice Python dagli oggetti `HTML`, `BODY`, and `H1`; il database `db` è interrogato dal comando `select` ed infine il risultato è trasformato in HTML. `db` non è una keyword, ma una variabile definita dall'utente. Per evitare confusione questo nome sarà usato in questo libro per riferirsi ad una

connessione al database.

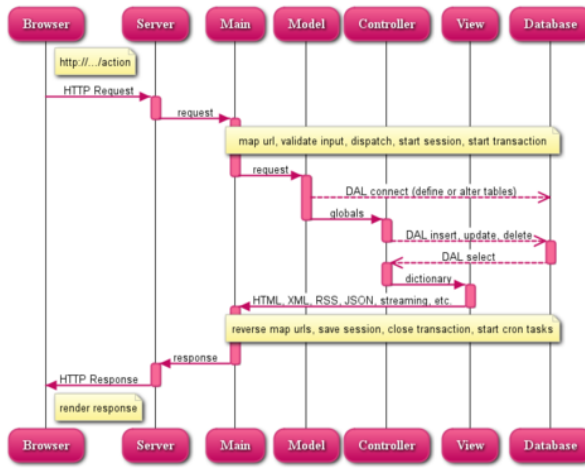
I framework web sono tipicamente divisi in due categorie: framework *glued* che sono costruiti assemblando (incollando insieme) diverse componenti di terze parti e framework *full-stack* che sono costruiti creando componenti specificatamente progettati per lavorare insieme e che sono quindi strettamente integrati. web2py è un framework *full-stack*. Quasi tutti i suoi componenti sono costruiti da zero e progettati per lavorare insieme, ma possono funzionare anche separamente dal framework stesso. Per esempio il DAL o il linguaggio di template possono essere usati indipendentemente dal resto del framework importando i moduli `gluon.sql` o `gluon.template` nelle applicazioni Python. `gluon` è il nome della cartella di web2py che contiene le librerie di sistema. Alcune librerie di web2py (come quella che genera e processa i form partendo dalle tabelle del database) dipendono da altre porzioni del framework. web2py può anche utilizzare librerie Python di terze parti (per esempio altri linguaggi di template o altri DAL), ma queste non saranno strettamente integrate come i componenti originali.

1.3 Model-View-Controller

web2py utilizza il paradigma di sviluppo MVC (Model-View-Controller) che obbliga lo sviluppatore a separare la definizione dei dati (il modello, *model*), la presentazione dei dati (la vista, *view*) e il flusso delle operazioni (il controllo, *controller*). Considerando ancora l'esempio precedente è possibile costruire una applicazione web2py intorno ad esso.



Il tipico flusso di una richiesta in web2py è descritta nel seguente diagramma:



Nel diagramma:

- Il Server può essere il server web interno di web2py oppure un server web esterno come Apache. Il Server gestisce il multi-threading.
- "main" è la applicazione WSGI principale. Esegue tutti i compiti comuni

e racchiude l'applicazione. Si occupa dei cookie, delle sessioni, delle transazioni dell'instradamento (*routing*) delle URL, dell'instradamento inverso (*reverse routing*) delle URL e del passaggio dalle URL al controller (*dispatching*). Può distribuire all'utente i file statici nel caso che il server web non sia configurato per farlo direttamente.

- I componenti Model, View e Controller formano l'applicazione.
- Possono esistere applicazione multiple ospitate dalla stessa istanza di web2py.
- Le linee tratteggiate rappresentano la comunicazione con il motore (o i motori) di database. Le query al database possono essere scritte direttamente in SQL (non consigliato) o utilizzando il DAL, lo strato di astrazione del database (consigliato), in modo che l'applicazione web2py non sia dipendente da uno specifico motore di database.
- Il *dispatcher* collega le URL richieste alle relative funzioni del controller. L'output della funzione può essere una stringa o un dizionario di simboli (una *hash table*). I dati nel dizionario sono visualizzati dalla vista. Se l'utente richiede una pagina HTML (questo è il comportamento di default) il dizionario è visualizzato in una pagina HTML. Se l'utente richiede la stessa pagina in XML web2py tenta di trovare una vista che possa visualizzare il dizionario in XML. Lo sviluppatore può creare delle viste per visualizzare le pagine in uno dei protocolli già supportati (HTML, XML, JSON, RSS, CSV, RTF) oppure aggiungerne altri.
- Tutte le chiamate sono racchiuse in una transazione ed ogni eccezione non gestita fa si che la transazione venga annullata. Se la richiesta è conclusa con successo la transazione viene completata.
- web2py gestisce automaticamente le sessioni e i cookie di sessione. Quando una transazione è completata la sessione è memorizzata a meno che non sia specificato diversamente.
- È possibile registrare un'operazione ricorrente in modo che sia eseguita ad orari prefissati (*cron*) e/o dopo il completamento di una specifica azione. In questo modo è possibile eseguire in background attività lunghe o pesanti senza rallentare la navigazione degli utenti.

Ecco una completa applicazione MVC minimale formata da soli tre file:

"db.py" è il modello:

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

Questo codice connette il database (in questo esempio è un database SQLite memorizzato nel file `storage.sqlite`) e definisce una tabella chiamata `contacts`. Se la tabella non esiste web2py la crea e trasparentemente genera il codice SQL nel dialetto appropriato per lo specifico motore di database utilizzato. Lo sviluppatore può vedere il codice SQL generato ma non ha la necessità di cambiare il codice se il database (di default è impostato SQLite) è sostituito con MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2, Informix, Interbase, Ingres o BigTables nel Google App Engine.

Una volta che la tabella è stata definita e creata web2py genera inoltre una interfaccia web di amministrazione per accedere al database e alle sue tabelle chiamata **appadmin**.

"default.py" è il controller:

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

In web2py le URL sono collegate ai moduli e alle chiamate alle funzioni di Python. In questo caso il controller contiene una singola funzione (o "azione") chiamata `contacts`. Una azione può ritornare una stringa o un dizionario (un insieme di coppie `key:value`). Se la funzione ritorna un dizionario questo è passato ad una vista (con il nome combinato del controller/azione) che lo visualizza. In questo esempio la funzione `contacts` esegue una `select` nel database e ritorna i record risultanti come un valore associato alla chiave `records` nel dizionario.

"default/contacts.html" è la vista:

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

Questa vista è chiamata automaticamente da web2py dopo che la funzione associata nel controller (azione) è eseguita. Lo scopo di questa vista è visualizzare le variabili del dizionario (passato dall'azione) `records=...` in HTML. La vista è scritta in HTML, ma incorpora del codice Python delimitato dai caratteri speciali `{{` e `}}`. Questo è diverso dall'esempio precedente in PHP perchè l'unico codice incorporato in HTML è quello dello strato di presentazione. Il file "layout.html" referenziato all'inizio della vista è incluso in web2py e costituisce il layout di base per tutte le applicazioni web2py. Il layout può essere facilmente modificato o sostituito.

1.4 Perchè web2py

web2py è uno dei molti framework di sviluppo web, ma ha caratteristiche uniche ed interessanti. web2py è stato inizialmente sviluppato come uno strumento di insegnamento, con queste fondamentali motivazioni:

- Rendere facile per gli utenti l'apprendimento dello sviluppo di applicazioni web lato server senza limitarne le funzionalità. Per questa ragione web2py non richiede nessuna installazione nè configurazione, non ha dipendenze (eccetto per la distribuzione del codice sorgente che richiede Python 2.5 e la sua libreria di moduli standard) ed espone la maggior parte delle sue funzionalità tramite un'interfaccia web.
- web2py è stabile fin dal suo primo giorno di sviluppo perchè segue una modalità di progettazione top-down: per esempio le sue API sono state progettate prima di essere state sviluppate. Ogni volta che una nuova funzionalità è stata aggiunta web2py non ha mai perso la retro-compatibilità e non perderà la retro-compatibilità quando verranno aggiunte ulteriori

funzionalità in futuro.

- web2py risolve in anticipo i problemi di sicurezza più importanti (determinati da OWASP (19)) che affliggono molte applicazioni web moderne e che sono elencati nel successivo paragrafo.
- web2py è leggero. Le sue librerie interne, che includono il DAL, il linguaggio di template e le funzioni accessorie, occupano circa 300 KB. Tutto il codice sorgente (incluse le applicazioni di esempio e le immagini) è di circa 2 MB.
- web2py ha una impronta in memoria limitata ed è molto veloce. Utilizza il server web WSGI Rocket (22) progettato da Timothy Farrell. È più veloce del 30% di Apache con mod_proxy. I test indicano che, su un PC di media potenza, restituisce una pagina creata dinamicamente, senza accesso al database, in circa 10ms. L'utilizzo del DAL rallenta l'applicazione per meno del 3%.

(17; 18) WSGI (*Web Server Gateway Interface*) è uno standard emergente per la comunicazione tra i server web e le applicazioni Python.

1.5 Sicurezza

Open Web Application Security Project (19) (OWASP) è una comunità libera ed aperta a livello mondiale che si occupa di migliorare la sicurezza del software.

OWASP ha elencato i dieci principali problemi di sicurezza che mettono a rischio le applicazioni web. Questo elenco è qui riprodotto con una descrizione del problema e di come è affrontato in web2py:

- "Cross Site Scripting (XSS): una falla di tipo XSS avviene quando una applicazione prende i dati forniti dall'utente e li inoltra ad un browser senza prima validarli o codificarli. XSS consente ad un attaccante di eseguire script nel browser della vittima che possono catturare le sessioni utente,

modificare siti web, introdurre virus, ed altro." web2py di default codifica (*escape*) tutte le variabili presentate nella vista evitando così attacchi di questo tipo.

- "Injection Flaws: nelle applicazioni web sono comuni falle di iniezione di codice, in modo particolare di tipo SQL. L'iniezione di codice avviene quando i dati immessi dall'utente sono inviati ad un interprete come parte di un comando o di una query. I dati ostili inviati dall'attaccante fanno si che l'interprete esegua comandi non voluti o modifichi i dati." web2py include uno strato di astrazione del database (DAL) che rende impossibili le iniezioni di codice SQL. Solitamente i comandi SQL non sono scritti dallo sviluppatore ma vengono generati dinamicamente dal DAL. Questo assicura che tutti i dati siano adeguatamente codificati.
- "Malicious File Execution: L'esecuzione malevola di codice nei file remoti consente all'attaccante di includere codice e dati ostili, creando attacchi devastanti, che possono portare alla compromissione totale del server." web2py permette l'esecuzione unicamente delle funzioni volutamente esposte, impedendo l'esecuzione di codice malevolo. Le funzioni importate non sono mai esposte, solo le azioni dei controller sono esposte. L'interfaccia di amministrazione di web2py rende molto facile tener traccia di cosa sia esposto e cosa no.
- "Insecure Direct Object Reference: un riferimento diretto ad un oggetto avviene quando uno sviluppatore espone un riferimento ad un oggetto interno dell'implementazione, come un file, una cartella, un record di database o una chiave, tramite una URL o un parametro del form. L'attaccante può manipolare quei riferimenti per accedere ad altri oggetti a cui non è autorizzato." web2py non espone alcun oggetto interno; inoltre web2py valida tutte le URL in modo da prevenire attacchi di attraversamento delle cartelle (*directory traversal*). web2py fornisce inoltre un semplice metodo che consente di creare form che validano automaticamente tutti i valori in input.
- "Cross Site Request Forgery (CSRF): Un attacco di tipo CSRF forza il browser di un utente già autenticato a mandare una richiesta pre-autenticata ad una applicazione web vulnerabile e successivamente obbliga il browser della vittima ed eseguire una azione ostile a vantaggio dell'attaccante.

CSRF può essere tanto potente quanto lo è l'applicazione che sta attaccando." web2py previene le falle di tipo CSRF così come evita il doppio invio di form assegnando un "gettone" (*token*) identificativo ad ogni form. Tale token è valido una sola volta. Inoltre web2py utilizza gli UUID per generare i cookie di sessione.

- "Information Leakage and Improper Error Handling: Le applicazioni possono rilasciare non intenzionalmente informazioni sulla loro configurazione e sul lavoro interno dell'applicazione o possono violare la privacy attraverso una varietà di problemi applicativi. Un attaccante può utilizzare queste debolezze per rubare dati sensibili o condurre attacchi di maggior gravità." web2py include un sistema di gestione delle segnalazioni d'errore (*ticketing*). Quando avviene un errore non viene mai visualizzato all'utente il codice sorgente che lo ha causato. L'errore, con le informazioni necessarie ad identificarlo, è registrato ed un ticket è inviato all'utente per consentire il tracciamento dell'errore. L'errore, le relative informazioni ed il codice sorgente dell'errore sono accessibili solo all'amministratore.
- "Broken Authentication and Session Management: le credenziali dell'acconto utente e i token di sessione spesso non sono protetti adeguatamente. Un attaccante può compromettere le password, le chiavi o i token di autenticazione per assumere l'identità di un altro utente." web2py fornisce un meccanismo predefinito per l'accesso dell'amministratore e gestisce le sessioni in modo indipendente per ogni applicazione. L'interfaccia amministrativa inoltre impone l'uso di cookie di sessione sicuri quando il browser non è collegato da *localhost*. Per le applicazioni web2py include una potente API di RBAC (*Role-Based Access Control*, Controllo d'accesso basato sui ruoli).
- "Insecure Cryptographic Storage: le applicazioni web raramente utilizzano in modo adeguato le funzioni di crittografia per proteggere i dati e le credenziali. Un attaccante può usare dati protetti debolmente per condurre un furto di identità ed altri crimini, come le frodi con la carta di credito." web2py utilizza algoritmi di cifratura di tipo MD5 o HMAC+SHA-512 per proteggere le password memorizzate. Altri algoritmi sono disponibili.
- "Insecure Communications: le applicazioni spesso non cifrano il traffico di rete quando sarebbe necessario proteggere comunicazioni sensibili."

web2py include il server web ssl (21) Rocket WSGI, ma può anche utilizzare Apache o Lighttpd e mod_ssl per fornire una cifratura SSL delle comunicazioni.

- "Failure to Restrict URL Access: Spesso una applicazione protegge le funzioni e i dati sensibili semplicemente prevenendo la visualizzazione di link o di URL agli utenti non autorizzati. Un attaccante può usare questa debolezza pe accedere ed eseguire operazioni non autorizzate accedendo a direttamente a tali URL." web2py collega le URL di richiesta a moduli e a funzioni Python. web2py fornisce un meccanismo per indicare quali funzioni sono pubbliche e quali richiedono autenticazione ed autorizzazione. Le API di RBAC incluse consentono agli sviluppatori di restringere l'accesso al livello di ogni funzione basandosi sul login, sull'appartenenza ad un certo gruppo o sui permessi assegnati ad un gruppo. I permessi sono estremamente granulari e possono essere combinati con le funzionalitò di creazione, lettura, aggiornamento, cancellazione (CRUD, *Create, read, update, delete*) per consentire, per esempio, l'accesso solo ad alcune tabelle e/o record.

web2py è stato valutato dal punto di vista della sicurezza e i risultati possono essere consultati in ref (20).

1.6 Cosa c'è "nella scatola"

Puoi scaricare web2py direttamente dal sito ufficiale

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

web2py è composto dai seguenti componenti:

- **librerie**: forniscono le funzionalità di base di web2py e sono accessibili da codice.
- **server web**: il server web Rocket WSGI.

- l'applicazione **admin**: utilizzata per creare, progettare e gestire altre applicazioni web2py. **admin** fornisce un ambiente integrato di sviluppo (IDE, *Integrated Development Environment*) per costruire le applicazioni. Include inoltre altre funzionalità come i test e la riga di comando.

- l'applicazione **examples**: questa applicazione contiene documentazione ed esempi interattivi. **examples** è una copia del sito ufficiale di web2py ed include la documentazione in modo epydoc e Sphinx.

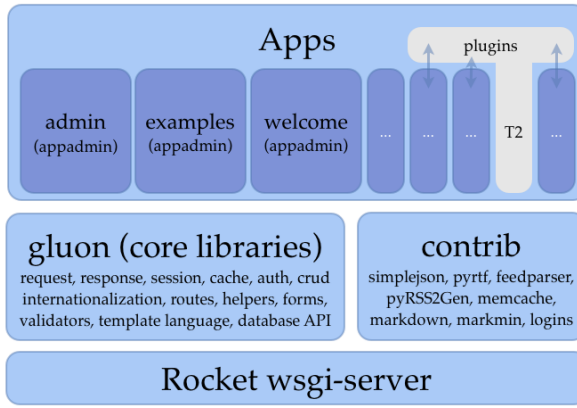
- l'applicazione **welcome**: questa applicazione è l'intelaiatura di base per ogni altra applicazione. Include di default un menu in CSS e l'autenticazione utente (discussa nel capitolo 8).

web2py è distribuito in codice sorgente e in forma binaria per Microsoft Windows e Mac OS X.

La distribuzione in codice sorgente può essere utilizzata in ogni piattaforma dove Python esiste e include i componenti sopra menzionati. Per eseguire il codice sorgente è necessario avere Python 2.5 installato sul sistema. È necessario inoltre aver installato uno dei motori di database supportati. Per il test e applicazioni leggere è possibile utilizzare il database SQLite, incluso con Python 2.5.

La versione binaria di web2py (per Microsoft Windows e Mac OS X) include al suo interno un interprete Python 2.5 ed il database SQLite. Tecnicamente questi non sono componenti di web2py ma l'inclusione nella distribuzione binaria consente di utilizzare web2py immediatamente.

La seguente immagine illustra la struttura complessiva di web2py:



1.7 *Licenza*

web2py è distribuito sotto la licenza GPL versione 2. Il testo completo della licenza è disponibile in (31).

La licenza include, ma non è limitata a, i seguenti articoli:

1. È lecito copiare e distribuire copie letterali del codice sorgente del Programma così come viene ricevuto, con qualsiasi mezzo, a condizione che venga riprodotta chiaramente su ogni copia una appropriata nota di copyright e di assenza di garanzia; che si mantengano intatti tutti i riferimenti a questa Licenza e all'assenza di ogni garanzia; che si dia a ogni altro destinatario del Programma una copia di questa Licenza insieme al Programma.

[...]

4. Non è lecito copiare, modificare, sublicenziare, o distribuire il Programma in modi diversi da quelli espressamente previsti da questa Licenza. Ogni tentativo di copiare, modificare, sublicenziare o distribuire altrimenti il Programma non è autorizzato, e farà terminare automaticamente i diritti garan-

titi da questa Licenza.

[...]

11. POICHÉ IL PROGRAMMA È CONCESSO IN USO GRATUITAMENTE, NON C'È GARANZIA PER IL PROGRAMMA, NEI LIMITI PERMESSI DALLE VIGENTI LEGGI. SE NON INDICATO DIVERSAMENTE PER ISCRITTO, IL DETENTORE DEL COPYRIGHT E LE ALTRE PARTI FORNISCONO IL PROGRAMMA "COSÌ COM'È", SENZA ALCUN TIPO DI GARANZIA, NÉ ESPLICITA NÉ IMPLICITA; CIÒ COMPRENDE, SENZA LIMITARSI A QUESTO, LA GARANZIA IMPLICITA DI COMMERCIABILITÀ E UTILIZZABILITÀ PER UN PARTICOLARE SCOPO. L'INTERO RISCHIO CONCERNENTE LA QUALITÀ E LE PRESTAZIONI DEL PROGRAMMA È DELL'ACQUIRENTE. SE IL PROGRAMMA DOVESSE RIVELARSI DIFETTOSO, L'ACQUIRENTE SI ASSUME IL COSTO DI OGNI MANUTENZIONE, RIPARAZIONE O CORREZIONE NECESSARIA.

12. NÈ IL DETENTORE DEL COPYRIGHT NÈ ALTRE PARTI CHE POSSONO MODIFICARE O RIDISTRIBUIRE IL PROGRAMMA COME PERMESSO IN QUESTA LICENZA SONO RESPONSABILI PER DANNI NEI CONFRONTI DELL'ACQUIRENTE, A MENO CHE QUESTO NON SIA RICHIESTO DALLE LEGGI VIGENTI O APPAIA IN UN ACCORDO SCRITTO. SONO INCLUSI DANNI GENERICI, SPECIALI O INCIDENTALI, COME PURE I DANNI CHE CONSEGUONO DALL'USO O DALL'IMPOSSIBILITÀ DI USARE IL PROGRAMMA; CIÒ COMPRENDE, SENZA LIMITARSI A QUESTO, LA PERDITA DI DATI, LA CORRUZIONE DEI DATI, LE PERDITE SOSTENUTE DALL'ACQUIRENTE O DA TERZI E L'INCAPACITÀ DEL PROGRAMMA A INTERAGIRE CON ALTRI PROGRAMMI, ANCHE SE IL DETENTORE O ALTRE PARTI SONO STATE AVVISATE DELLA POSSIBILITÀ DI QUESTI DANNI.

- web2py include codice di terze parti (per esempio l'inteprete Python, il server web Rocket e alcune librerie JavaScript). I loro rispettivi autori e licenze sono riconosciuti nel sito ufficiale di web2py (1) e nel codice sorgente stesso.

- Le applicazioni sviluppate con web2py, se non includono codice sorgente di web2py, non sono considerate lavori derivati. Questo significa che non sono legate alla licenza GPL V2 e possono essere distribute con qualsiasi licenza si desideri, incluse licenze closed source e commerciali.

Eccezione di Licenza Commerciale La licenza web2py include anche una eccezione commerciale: è concessa la distribuzione di una applicazione sviluppata con web2py insieme ad una versione binaria ufficiale non modificata di web2py, scaricata dal sito ufficiale (1), purchè sia indicato chiaramente nella licenza della applicazione quali file appartengano all'applicazione e quali appartengano a web2py.

1.8 Riconoscimenti

web2py è stato originariamente sviluppato e registrato da Massimo di Pierro. La prima versione (1.0) è stata rilasciata nell'ottobre del 2007. Da quella data web2py è stato adottato da molti utenti, alcuni dei quali hanno contribuito con report di bug, con test, debug, patch e correzioni a questo libro.

Alcuni dei principali contributori, in ordine alfabetico per nome, sono:

Alexey Nezhdanov, Alvaro Justen, Attila Csipa, Bill Ferrett, Boris Manojlovic, Brian Meredyk, Carsten Haese, Chris Clark, Chris Steel, Christopher Smiga, CJ Lazell, Craig Younkins, Daniel Lin, Denes Lengyel, Douglas Soares de Andrade, Fran Boon, Francisco Gama, Fred Yanowski, Graham Dumpleton, Gyuris Szabolcs, Hamdy Abdel-Badeea, Hans Donner, Hans Murx, Hans C. v. Stockhausen, Ian Reinhart Geiser, Jonathan Benn, Josh Goldfoot, Jose Jachuf, Keith Yang, Kyle Smith, Limodou, Marcel Leuthi, Marcello Della Longa, Mariano Reingart, Mark Larsen, Mark Moore, Markus Gritsch, Martin Hufsky, Mateusz Banach, Michael Willis, Michele Comitini, Nathan Freeze, Niall Sweeny, Niccolo Polo, Nicolas Bruxer, Ondrej Such, Pai, Phyo Arkar Lwin, Robin Bhattacharyya, Ruijun Luo (Iceberg), Sergey Podlesnyi, Sharriff

Aina, Sterling Hankins, Stuart Rackham, Telman Yusupov, Thadeus Burgess, Tim Michelsen, Timothy Farrell, Yair Eshel, Yarko Tymciurak, Younghyun Jo, Zahariash.

Sono sicuro di dimenticare qualcuno, per questo chiedo scusa.

Desidero ringraziare particolarmente Jonathan, Iceberg, Nathan, and Thadeus per i loro notevoli contributi a web2py e Alvaro, Denes, Felipe, Graham, Jonathan, Hans, Kyle, Mark, Richard, Robin, Roman, Scott, Shane, Sharriff, Sterling, Stuart, Thadeus e Yarko per aver corretto vari capitoli di questo libro. Il loro contributo è stato inestimabile. Se vi sono errori in questo libro sono esclusivamente per mia colpa, introdotti probabilmente con una modifica dell'ultimo minuto. Desidero anche ringraziare Ryan Steffen della Wiley Custom Learning Solutions per l'aiuto nella pubblicazione della prima versione di questo libro. web2py contiene codice dai seguenti autori, che desidero ringraziare: Guido van Rossum per Python (2), Peter Hunt, Richard Gordon, Timothy Farrell per il server web Rocket (22), Christopher Dolivet per EditArea (23), Bob Ippolito per simplejson (25), Simon Cusack e Grant Edwards per pyRTF (26), Dalke Scientific Software per pyRSS2Gen (27), Mark Pilgrim per feedparser (28), Trent Mick per markdown2 (29), Allan Saddi per fcgi.py, Evan Martin per il modulo Python memcache (30), John Resig per jQuery (32).

Il logo utilizzato sulla copertina di questo libro è stato progettato da Peter Kirchner della Young Designers.

Ringrazio Helmut Epp (Rettore della DePaul University), David Miller (Preside del College of Computing and Digital Media della DePaul University), ed Estia Eichten (Membro di MetaCryption LLC), per la loro continua fiducia ed il loro supporto.

Infine desidero ringraziare mia moglie, Claudia, e mio figlio, Marco, per avermi sopportato nelle molte ore che ho impiegato nello sviluppo di web2py, scambiando email con utenti e collaboratori e scrivendo questo libro. Questo libro è dedicato a loro.

1.9 *Struttura del libro*

Questo libro include i seguenti capitoli, oltre a questa introduzione:

- Il capitolo 2 è una introduzione minimalista al linguaggio Python, Richiede la conoscenza di concetti della programmazione procedurale ed orientata agli oggetti come cicli, condizioni, chiamate a funzioni e classi e copre la sintassi di base di Python. Contiene anche esempi di moduli Python che sono usati nel resto del libro. Se già conoscete Python potete saltare questo capitolo.
- Il capitolo 3 mostra come avviare web2py, discute l'interfaccia amministrativa e guida il lettore in vari esempi di complessità crescente: una applicazione che presenta una stringa, una applicazione che presenta un contatore, un blog di immagini, una applicazione completa per un wiki che permette di caricare immagini e commenti, fornisce autenticazione, autorizzazione, servizi web e un feed RSS. Nella lettura di questo capitolo si può far riferimento al capitolo 2 per la sintassi generale di Python ed ai capitoli successivi per una descrizione più dettagliata delle funzioni utilizzate.
- Il capitolo 4 copre in modo più sistematico la struttura centrale e le librerie di web2py: mapping delle URL, oggetto request, oggetto response, sessioni, cacheint, CRON, internazionalizzazione e flusso generale dell'applicazione.
- Il capitolo 5 è il riferimento per il linguaggio di template utilizzato per costruire le view. Mostra come incorporare codice Python in codice HTML e dimostra l'utilizzo delle funzioni ausiliarie (*helper*, che possono generare codice HTML).
- Il capitolo 6 illustra il DAL, lo strato di astrazione del database. La sintassi del DAL è presentata attraverso una serie di esempi.
- Il capitolo 7 spiega le form, la validazione delle form e la loro elaborazione. FORM è la funzione ausiliaria di basso livello per costruire le form. SQLFORM è il costruttore di form di alto livello. Nel capitolo 7 è illustrata anche la nuova API per la creazione, la lettura, l'aggiornamento e la cancellazione dei record (CRUD, Create/Read/Update/Delete).

- Il capitolo 8 descrive l'autenticazione, l'autorizzazione ed il meccanismo estendibile di controllo dell'accesso basato sui ruoli (RBAC, Role-Based Access Control) disponibile in web2py. È illustrata anche la configurazione del sistema di posta e del sistema di implementazione dei CAPTCHA poichè sono utilizzati nella fase di autenticazione. Nella terza edizione di questo libro sono stati aggiunti numerosi esempi di integrazione con meccanismi di autenticazione di terze parti, come OpenID, OAuth, Google, Facebook, Linkedin ed altri.

- Il capitolo 9 riguarda la creazione di servizi web (*web services*) in web2py. Sono forniti esempi di integrazione con il Web Toolkit di Google tramite Pyjamas e con Adobe Flash tramite PyAMF.

- Il capitolo 10 riguarda web2py e jQuery. web2py è progettato principalmente per la programmazione lato server, ma include jQuery, che riteniamo essere la migliore libreria open source in JavaScript disponibile per creare effetti e per Ajax. In questo capitolo è descritto come utilizzare jQuery con web2py.

- Il capitolo 11 illustra l'ambiente di produzione delle applicazioni web2py. Sono illustrati tre possibili scenari di produzione: messa in produzione su uno o più server Linux (questo è il principale ambiente di produzione), l'utilizzo come servizio in un ambiente Microsoft Windows e infine la messa in produzione sul Google Application Engine (GAE). In questo capitolo sono evidenziati anche i problemi di sicurezza e di scalabilità.

- Il capitolo 12 contiene una varietà di altre 'ricette' per risolvere specifici problemi, come gli aggiornamenti, la geolocalizzazione, la paginazione, le API per Twitter ed altro ancora.

- Il capitolo 13 è stato aggiunto nella terza edizione di questo libro e riguarda i componenti ed i plugin per web2py utilizzati per costruire applicazioni modulari. È fornito un esempio di un plugin che implementa una serie di funzionalità comuni come i grafici, il tagging, e i wiki.

Questo libro copre esclusivamente le funzionalità di base di web2py e l'API presente nel framework.

Questo libro non illustra le *appliance* di web2py (applicazioni già pronte). Le *appliance* di web2py possono essere scaricate dal relativo sito web (34).

Altri argomenti riguardanti web2py possono essere trovati su AlterEgo (35) la FAQ interattiva di web2py.

1.10 Elementi di stile

Ref. (36) contiene le buone pratiche di programmazione di Python. web2py non segue sempre queste regole non per mancanza o negligenza, siamo infatti convinti che lo sviluppatore di applicazioni web2py debba seguire queste regole (e lo incoraggiamo a farlo). Abbiamo deciso di non seguire alcune di queste regole nella definizione degli oggetti ausiliari (*helper*) per minimizzare la probabilità che i nomi di questi oggetti confliggano con quelli definiti dallo sviluppatore.

Per esempio la classe che rappresenta un `<div>` è chiamata `DIV`, mentre, secondo le regola di styte di Python dovrebbe chiamarsi `Div`. Noi riteniamo che, per questo specifico caso, l'utilizzo di un nome composto di tutte lettere maiuscole sia una scelta naturale. Inoltre questo approccio lascia libertà allo sviluppatore di creare una classe chiamata "Div" se ne ha necessità. Questa sintassi inoltre si sovrappone con naturalezza alla notazione del DOM (*Document Object Model*, la struttura del documento HTML) utilizzata da molti browser (come ad esempio FireFox).

Secondo la guida di stile di Python i nomi composti di tutte lettere maiuscole dovrebbero essere utilizzati per le costanti e non per le variabili. Continuando con l'esempio è possibile considerare `DIV` come una classe speciale che non deve mai essere modificata dallo sviluppatore perchè questo causerebbe danni alle applicazioni installate in web2py. Crediamo qundi che questo qualifichi la classe `DIV` come qualcosa che dovrebbe essere trattato come una costante, giustificando maggiormente la scelta di questa notazione.

Sono utilizzate le seguenti convenzioni:

- Le funzioni ausliarie per HTML sono tutte in caratteri maiuscoli per il motivo sopra descritto (per esempio `DIV`, `A`, `FORM`, `URL`).
- L'oggetto traduttore `T` è in maiuscolo nonostante sia l'istanza di un classe e non la classe stessa. Da un punto di vista logico l'oggetto traduttore esegue un'azione simile a quelli delle funzioni ausiliarie per l'HTML: modifica la parte di visualizzazione di un oggetto. Inoltre `T` è facile da trovare nel codice e deve avere un nome breve.
- Le classi del DAL seguono le regole di stile di Python (prima lettera maiuscola), come, per esempio, `Table`, `Field`, `Query`, `Row`, `Rows`, etc.

In tutti gli altri casi crediamo di aver seguito, per quanto sia stato possibile, la guida di stile di Python (PEP8). Per esempio tutte le istanze di oggetti sono in lettere minuscole (equest, response, session, cache) e tutte le classi interne hanno l'iniziale maiuscola.

In tutti gli esempi di questo libro le parole chiave di web2pyu sono mostrate in grassetto, mentre le stringhe e i commenti sono mostrati in corsivo.

2

Il linguaggio Python

2.1 Python

Python è un linguaggio di programmazione *general purpose* (utilizzabile quindi per gli scopi più diversi) estremamente di alto livello. La sua filosofia di progettazione pone l'accento sulla produttività del programmatore e sulla leggibilità del codice. Il suo nucleo ha una sintassi minimale con pochissimi comandi e con una semantica semplice, ma contiene un'ampia libreria standard con API per molte delle funzioni del sistema operativo su cui viene eseguito l'interprete. Il codice Python, seppur minimale, definisce oggetti di tipo liste (`list`), tuple (`tuple`), dizionari (`dict`) ed interi a lunghezza arbitraria (`long`).

Python supporta diversi paradigmi di programmazione: programmazione orientata agli oggetti (`class`), programmazione imperativa (`def`), e programmazione funzionale (`lambda`). Python ha un sistema di tipizzazione dinamica e un sistema automatico di gestione della memoria che utilizza il conteggio dei riferimenti (come Perl, Ruby e Scheme).

La prima versione di Python è stata rilasciata da Guido van Rossum nel 1991.

Il linguaggio ha un modello di sviluppo aperto, basato su una comunità di sviluppatori gestita dalla Python Software Foundation, una associazione non-profit. Ci sono molti interpreti e compilatori che implementano il linguaggio Python, tra cui uno in Java (Jython). In questo breve approfondimento si farà riferimento all'implementazione C creata da Guido.

Sul sito ufficiale di Python (2) si possono trovare molti tutorial, la documentazione ufficiale e il manuale di riferimento della libreria standard del linguaggio.

Per ulteriori approfondimenti si possono vedere i libri indicati nelle note. (37) e (38).

Questo capitolo può essere saltato se si ha già familiarità con il linguaggio Python.

2.2 *Avvio*

Le distribuzioni binarie di web2py per Microsoft Windows e Apple Mac OS X già comprendono l'interprete Python.

Su Windows web2py può essere avviato con il seguente comando (digitato al prompt del DOS):

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

Su Apple Mac OS X digitare il seguente comando nella finestra del Terminale (dalla stessa cartella contenente web2py.app):

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

Su una macchina Linux o su macchine Unix è molto probabile che Python sia già installato. In questo caso digitare al prompt dello shell il seguente comando:

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

Nel caso che Python 2.5 non sia presente dovrà essere scaricato e installato prima di eseguire web2py.

L'opzione della linea di comando `-S welcome` indica a web2py di eseguire la shell interattiva come se i comandi fossero inseriti in un controller dell'applicazione **welcome**, l'applicazione di base di web2py. Questo rende disponibili quasi tutte le classi, le funzioni e gli oggetti di web2py. Questa è l'unica differenza tra la shell interattiva di web2py e la normale linea di comando dell'interprete Python.

L'interfaccia amministrativa fornisce anche una shell utilizzabile via web per ogni applicazione. Per accedere a quella dell'applicazione "welcome" utilizzare l'indirizzo:

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

Tutti gli esempi di questo capitolo possono essere provati sia nella shell standard di Python che nella shell via web di web2py.

2.3 *help, dir*

Nel linguaggio Python sono presenti due comandi utili per ottenere informazioni sugli oggetti, sia interni che definiti dall'utente, istanziati nello *scope* corrente.

Utilizzare il comando `help` per richiedere la documentazione di un oggetto (per esempio "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)
...
```

e, poichè "1" è un intero, si ottiene come risposta la descrizione della classe `int` e di tutti i suoi metodi. Qui l'output del comando è stato troncato perchè molto lungo e dettagliato.

Allo stesso modo si può ottenere una lista di metodi dell'oggetto "1" con il comando `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 *Tipi*

Python è un linguaggio dinamicamente tipizzato. Questo significa che le variabili non hanno un tipo e pertanto non devono essere dichiarate. I valori, d'altra parte, hanno un tipo. Si può interrogare una variabile per sapere il tipo del valore che contiene:

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

Python include nativamente anche strutture dati come le liste e i dizionari.

2.4.1 *str*

Python supporta l'utilizzo di due differenti tipi di stringhe: stringhe ASCII e stringhe Unicode. Le stringhe ASCII sono delimitate dagli apici ('...'), dai doppi apici("...") o da tre doppi apici ("""..."""). I tre doppi apici delimitano le stringhe multilinea. Le stringhe Unicoode iniziano con il carattere `u` seguito dalla stringa che contiene i caratteri Unicode. Una stringa Unicode può essere convertita in una stringa ASCII scegliendo una codifica (*encoding*). Per esempio:

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

Dopo aver eseguito questi tre comandi la variabile `a` è una stringa ASCII che memorizza caratteri codificati con UTF8. Internamente web2py utlizza sempre la codifica UTF8 nelle stringhe.

È possibile scrivere il valore delle variabili nelle stringhe in diversi modi:

```
>>> 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
```

L'ultima notazione è la più esplicita e la meno soggetta ad errori. Ed è pertanto da preferire.

Molti oggetti in Python, per esempio i numeri, possono essere convertiti in stringhe utlizzando `str` o `repr`. Questi due comandi sono molto simili e producono risultati leggermente diversi. Per esempio:

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

Per le classi definite dall'utente `str` e `repr` possono essere definiti utilizzando gli speciali operatori `__str__` e `__repr__`. Questi metodi sono brevemente descritti più avanti. Per ulteriori informazioni si può fare riferimento alla documentazione ufficiale di Python (39). `repr` ha sempre un valore di default.

Un'altra caratteristica importante delle stringhe in Python è che sono oggetti *iterabili* (sui quali si può eseguire un ciclo che ritorna sequenzialmente ogni elemento dell'oggetto):

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

2.4.2 *liste*

I metodi principali delle liste in Python sono *append* (aggiungi), *insert* (inserisci) e *delete* (cancella):

```
>>> 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
```

Le liste possono essere *affettate* (*sliced*) per ottenerne solo alcuni elementi:

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

e concatenate:

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

Una lista è un oggetto iterabile quindi si può ciclare su di essa:

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

Gli elementi di una lista non devono obbligatoriamente essere dello stesso tipo, ma possono essere qualsiasi tipo di oggetto Python.

2.4.3 tuple

Una tupla è come una lista, ma la sua dimensione e i suoi elementi sono immutabili, mentre in una lista sono mutabili. Se un elemento di una tupla è un oggetto gli attributi dell'oggetto sono mutabili. Una tupla è definita dalle parentesi tonde:

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

Perciò, mentre questa assegnazione è valida per una lista:

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

l'assegnazione di un elemento all'interno di una tupla non è un'operazione valida:

```
>>> 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
```

La tupla, così come la lista, è un oggetto iterabile. È da notare che una tupla formata da un solo elemento deve includere una virgola alla fine:

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

Le tuple, a causa della loro immutabilità, sono molto efficienti per l'impacchettamento (*packing*) degli oggetti. Le parentesi spesso sono opzionali:

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

2.4.4 *dict*

Un dizionario (`dict`) in Python è una *hash table* che collega una chiave ad un valore. Per esempio:

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

Le chiavi possono essere di un qualsiasi tipo che implementa il metodo `__hash__` (int, stringa, o classe). I valori possono essere di qualsiasi tipo. Chiavi e valori diversi nello stesso dizionario non devono obbligatoriamente essere dello stesso tipo. Se le chiavi sono caratteri alfanumerici un dizionario può anche essere definito con una sintassi alternativa:

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

Metodi utili di un dizionario sono `has_key`, `keys`, `values` e `items`:

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

Il metodo `items` produce una lista di tuple ognuna contenente una chiave ed il suo valore associato.

Gli elementi di un dizionario o di una lista possono essere cancellati con il comando `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'}
```

Internamente Python utilizza l'operatore `hash` per convertire gli oggetti in interi ed utilizza quell'intero per determinare dove memorizzare l'oggetto.

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

2.5 *Indentazione del codice*

Python utilizza l'indentazione per delimitare blocchi di codice. Un blocco inizia con una linea che termina con i due punti (":") e continua per tutte le linee che hanno lo stesso livello di indentazione o un livello maggiore. Per esempio:

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

Normalmente si utilizzano 4 spazi per ogni livello di indentazione È una buona regola non mischiare i caratteri di tabulazione (*tab*) con gli spazi, perchè questo potrebbe causare problemi.

2.6 *for... in*

In Python è possibile ciclare (*loop*) sugli oggetti iterabili:

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

Una scorciatoia molto utilizzata è il comando `xrange`, che genera un iterabile senza memorizzare la lista degli elementi:

```
>>> for i in xrange(0, 4):
        print i
0
1
2
3
```

Questo è equivalente alla sintassi in C/C++/C#/Java:

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

Un altro utile comando è `enumerate`, che esegue il conteggio all'interno di un ciclo:

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

Esiste inoltre il comando `range(a, b, c)` che ritorna una lista di interi che ad iniziare dal valore `a`, incrementato del valore `c` e che si conclude con l'ultimo valore più piccolo di `b`. `a` ha come valore di default 0 e `c` ha default `1`. `xrange`

è simile ma non genera nessuna lista, solo un iteratore sulla lista, per questo è maggiormente utilizzato nei cicli.

Per uscire prematuramente da un ciclo si può utilizzare il comando `break`:

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

Per avviare la successiva iterazione del ciclo, senza eseguire tutto il blocco di codice che lo compone si può utilizzare il comando `continue`:

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

2.7 *while*

Il ciclo `while` in Python è simile a quello di molti altri linguaggi di programmazione: ripete l'iterazione per un numero infinito di volte controllando la condizione all'inizio ciclo. Se la condizione è `False` il ciclo termina.

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

Non esiste un costrutto di tipo `loop ... until` in Python.

2.8 def... return

Questa è una tipica funzione in Python:

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

Non c'è la necessità (e neanche la possibilità) di specificare i tipi degli argomenti o il tipo del valore che viene ritornato dalla funzione.

Gli argomenti delle funzioni possono avere valori di default e possono ritornare oggetti multipli:

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

Gli argomenti delle funzioni possono essere passati esplicitamente per nome:

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

Le funzioni possono avere un numero variabile di argomenti:

```
>>> 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'}
```

In questo esempio gli argomenti non passati per nome (`3`, `'hello'`) sono memorizzati nella lista `a` e gli argomenti passati per nome (`c`, `test`) sono memorizzati nel dizionario `b`.

Nel caso opposto una lista o una tupla può essere passata ad una funzione che richiede argomenti posizionali individuali tramite un'operazione di *spacchettamento* (*unpacking*):

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

e un dizionario può essere spacchettato per ottenere argomenti con nome:

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

2.9 *if... elif... else*

L'utilizzo dei comandi condizionali in Python è intuitivo:

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

"elif" significa "else if". Sia `elif` che `else` sono clausole opzionali. Può esistere più di un `elif` ma può essere presente un unico `else`. Condizioni di maggior complessità possono essere create utilizzando gli operatori `not`, `and` and `or`.

```
>>> 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 può generare un eccezione:

```
>>> try:
>>>     a = 1 / 0
>>> except Exception, e
>>>     print 'oops: %s' % e
>>> else:
>>>     print 'no problem here'
>>> finally:
>>>     print 'done'
oops: integer division or modulo by zero
done
```

Quando l'eccezione è generata viene intercettata dalla clausola `except`, che viene eseguita, mentre non viene eseguita la clausola `else`. Se non viene generata nessuna eccezione la clausola `except` non viene eseguita ma viene eseguita la clausola `else`. La clausola `finally` è sempre eseguita.

Possono esistere diverse clausole di tipo `except` per gestire differenti tipi di eccezione.

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

Le clausole `else` e `finally` sono opzionali:

Quella che segue è una lista di eccezioni predefinite in Python con l'aggiunta dell'unica eccezione che viene generata da web2py:

```
BaseException
 +-- HTTP (definita da 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
```

```
50      +-- UnicodeWarning
```

Per una descrizione dettagliata di ogni eccezione fare riferimento alla documentazione ufficiale di Python. web2py espone solamente una nuova eccezione, chiamata `HTTP`. Quando questa viene generata ritorna una pagina di errore HTTP (per maggiori informazioni fare riferimento al Capitolo 4). Qualsiasi oggetto può essere generato come un'eccezione, ma è buona norma generare solamente oggetti che estendono le eccezioni predefinite.

2.11 classi

Poichè Python è dinamicamente tipizzato a prima vista le classi e gli oggetti possono sembrare strani. In effetti non è necessario definire le variabili membro (attributi) quando si dichiara una classe e differenti istanze di una stessa classe possono avere differenti attributi. Gli attributi sono generalmente associati con l'istanza e non con la classe (tranne nel caso in cui sono dichiarati come *attributi di classe*, che è l'equivalente delle *variabili membro statiche* in C++/Java.

Ecco un esempio:

```
1 >>> class MyClass(object): pass
2 >>> myinstance = MyClass()
3 >>> myinstance.myvariable = 3
4 >>> print myinstance.myvariable
5 3
```

`pass` è un comando che non fa nulla. In questo caso è utilizzato per definire una classe `MyClass` che non contiene nulla. `MyClass()` richiama il costruttore della classe (in questo caso il costruttore di default) e ritorna on oggetto che è un'istanza della classe. `(object)` nella definizione della classe indica che la nuova classe estende la classe predefinita `object`. Questo non è obbligatorio ma è una buona regola da seguire.

Ecco qui una classe più complessa:

```
>>> 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
```

Le funzioni dichiarate all'interno di una classe sono chiamate *metodi*. Alcuni metodi speciali hanno nomi riservati. Per esempio `__init__` è il costruttore della classe. Tutte le variabili sono locali al metodo, tranne le variabili dichiarate all'esterno del metodo. Per esempio `z` è una *variabile di classe*, equivalente alle *variabili membro statiche* del C++, che contiene lo stesso valore per ogni istanza della classe.

È da notare che `__init__` ha tre argomenti e il metodo `add` ne ha uno ma vengono chiamati rispettivamente con due e con zero argomenti. Il primo argomento rappresenta, per convenzione, il nome locale usato all'interno dei metodi per riferirsi all'istanza corrente della classe. Qui viene utilizzato `self` per riferirsi all'oggetto corrente, ma potrebbe essere utilizzato un qualsiasi nome. `self` ha lo stesso ruolo di `*this` in C++ o `this` in Java, ma `self` non è una parola chiave riservata. In questo modo si evitano ambiguità nella dichiarazione di classi nidificate, come nel caso di una classe definita in un metodo interno ad un altra classe.

2.12 *Attributi, metodi ed operatori speciali*

Gli attributi, i metodi e gli operatori di classe che iniziano con un doppio underscore (__) sono solitamente considerati privati, anche se questa è solamente una convenzione è non è una regola forzata dall'interprete.

Alcuni di questi sono parole chiave riservate ed hanno un significato speciale.

Per esempio:

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

possono essere utilizzati per creare un oggetto contenitore che agisce come se fosse una lista:

```
>>> 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]
```

Altri operatori speciali sono `__getattr__` e `__setattr__`, che definiscono i modi di impostare e recuperare gli attributi della classe e `__sum__` e `__sub__`, che possono eseguire l'*overload* degli operatori aritmetici. Sono già stati menzionati gli operatori speciali `__str__` e `__repr__`. È consigliabile riferirsi a letture più approfondite per l'utilizzo di questi operatori.

2.13 *File Input/Output*

In Python è possibile aprire e scrivere in un file con:

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

Allo stesso modo si può rileggere il contenuto di un file con:

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

In alternativa si possono leggere i file in modalità binaria con "rb", scrivere in modalità binaria con "wb", e aprire i file in modalità append con "a", secondo la notazione standard del C.

Il comando `read` ha un argomento opzionale che è il numero di byte da leggere. Si può saltare ad una qualsiasi posizione all'interno del file con il comando `seek`.

Si può leggere il contenuto del file con `read`:

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

il file viene chiuso con:

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

sebbene spesso questo non sia necessario perchè il file è chiuso automaticamente quando la variable che lo rappresenta non è più nello scope.

> *Quando si usa web2py non si è a conoscenza di quale sia la directory corrente, perchè questo dipende da come web2py è configurato. Per questo motivo la variabile* `request.folder` *contiene il path dell'applicazione corrente. I path possono essere concatenati con il comando* `os.path.join` *discusso in seguito.*

2.14 *lambda*

Ci sono casi in cui è necessario generare dinamicamente una funzione anonima. Questo può essere fatto utilizzando la parola chiave `lambda`:

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

L'espressione "`lambda` [a]:[b]" si può leggere come "una funzione con argomento [a] che ritorna [b]". Anche se la funzione è anonima può essere memorizzata in una variabile e in questo modo acquisisce un nome. Questo tecnicamente è diverso dall'utilizzo di una `def`, perchè è la variabile che fa riferimento alla funzione anonima che ha un nome, non la funzione stessa.

A cosa servono le lamba? In effetti sono molto utili perchè permettono di riscrivere una funzione in un altra impostando dei parametri di default senza dover creare una nuova funzione. Per esempio:

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

Ecco un esempio più completo e più interessante: una funzione che controlla se un numero è primo:

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

Questa funzione, ovviamente, impiega molto tempo ad essere eseguita. Si supponga però di avere una funzione di cache `cache.ram` con tre argomenti: una chiave, una funzione e un numero di secondi:

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

La prima volta che viene chiamata la funzione `cache.ram` richiama la funzione `f()`, memorizza il risultato in un dizionario e ritorna il suo valore:

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

La seconda volta che la funzione `cache.ram` viene chiamata se la chiave è già nel dizionario e non è più vecchia del numero di secondi specificati (60), ritorna il valore corrispondente senza eseguire la chiamata alla funzione `f()`.

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

Come fare per memorizzare nella cache l'output della funzione **isprime** per ogni valore in input?

Ecco come:

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

L'output è sempre lo stesso, ma la prima volta che viene chiamata la funzione `cache.ram` viene eseguita anche `isprime`, la seconda volta no.

> *L'esistenza di* `lambda` *permette di riscrivere una funzione esistente in termini di un differente set di argomenti.* `cache.ram` *e* `cache.disk` *sono due funzioni di cache di web2py.*

2.15 *exec, eval*

A differenza di Java Python è un linguaggio completamente interpretato. Questo significa che si ha la possibilità di eseguire dei comandi Python inseriti in una stringa. Per esempio:

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

Cosa succede in questo caso? La funzione `exec` comanda all'interprete di richiamare se stesso ed eseguire il contenuto della stringa passata come ar-

gomento. È anche possibile eseguire il contenuto di una stringa con uno specifico contesto definito con i simboli inseriti in un dizionario:

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

In questo caso l'interprete, quando esegue il contenuto della stringa `a`, vede i simboli definiti in `c` (`b` nell'esempio), ma non vede `c` o `a`. Questo è differente dall'esecuzione in un ambiente ristretto poichè `exec` non limita cosa può essere fatto dal codice interno; semplicemente definisce le variabili visibili al codice da eseguire.

Una funzione molto simile ad `exec` è `eval`, l'unica differenza è che si aspetta che l'argomento passato ritorni un valore che viene a sua volta ritornato:

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

2.16 *import*

Il grande potere di Python risiede nella sua libreria di moduli. Questi infatti forniscono un ampio e consistente set di API (*Application Programming Interface*) per molte delle librerie di sistema (spesso in un modo indipendente dal sistema operativo).

Per esempio, per utilizzare il generatore di numeri casuali:

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

Questo codice stampa un intero tra 0 e 9 (incluso) generato casualmente, in questo caso il numero 5. La funzione `randint` è definita nel modulo `random`. È

anche possibile importare un oggetto da un modulo nel *namespace* corrente:

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

o importare tutti gli oggetti da un modulo nel *namespace* corrente:

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

o importare tutto il modulo in un nuovo *namespace*:

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

Nel resto del libro saranno utilizzati principalmente oggetti definiti nei moduli `os`, `sys`, `datetime`, `time` e `cPickle`.

> *Tutti gli oggetti di web2py sono accessibili tramite un modulo chiamato* `gluon`*, e questo sarà il soggetto dei capitoli successivi. Internamente web2py utilizza diversi moduli di Python (per esempio* `thread`*), ma sarà raramente necessario accedere ad essi direttamente.*

Nelle seguenti sezioni sono elencati i moduli di maggiore utilità.

2.16.1 os

Questo modulo fornisce una interfaccia alle API del sistema operativo. Per esempio:

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

> *Alcune delle funzioni del modulo* `os`*, come per esempio* `chdir`*, NON DEVONO essere utilizzate in web2py perchè non sono thread-safe (cioè sicure per essere utilizzate in applicazioni con più thread eseguiti contemporaneamente).*

`os.path.join` è estremamente utile; consente di concatenare i path in un modo indipendente dal sistema operativo.

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

Le variabili d'ambiente possono essere accedute con:

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

che è un dizionario in sola lettura.

2.16.2 sys

Il modulo `sys` contiene molte variabili e funzioni, ma quella che sarà maggiormente usata è `sys.path`. Contiene infatti una lista di path nei quali l'interprete Python ricerca i moduli. Quando si tenta di importare un modulo Python lo ricerca in tutte le cartelle elencate in `sys.path`. Se vengono installati moduli aggiuntivi in altre cartelle e si vuol far sì che Python li possa importare è necessario aggiungere il percorso al modulo in `sys.path`.

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

Quando si esegue web2py Python rimane residente in memoria ed esiste perciò un unico `sys.path` mentre ci sono molti thread che rispondono alle richieste HTTP. Per evitare problemi di memoria è bene controllare se il percorso è già presente prima di aggiungerlo a `sys.path`:

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

2.16.3 *datetime*

L'uso del modulo datetime è illustrato nei seguenti esempi:

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

A volte potrebbe essere necessario marcare un dato con un orario UTC (invece che locale). In questo caso è possibile utilizzare la seguente funzione:

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

Il modulo datetime contiene diverse classi: `date`, `datetime`, `time` e `timedelta`. La differenza tra due `date`, due `datetime` o due `time` è un `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` e `datetime` sono utilizzati per memorizzare i corrispondenti tipi SQL quando sono passati o ritornati dal database.

2.16.4 *time*

Il modulo `time` differisce da `date` e `datetime` perchè rappresenta il tempo come numero di secondi a partire dall'inizio del 1970 (*epoch*).

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

Si può far riferimento alla documentazione ufficiale di Python per le funzioni di conversione tra il tempo in secondi e il tempo come `datetime`.

2.16.5 *cPickle*

Questo è un modulo estremamente potente. Fornisce una serie di funzioni che possono serializzare quasi tutti gli oggeti Python, inclusi gli oggetti auto-referenziali. Ecco per esempio, un oggetto piuttosto strano:

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

ed ora:

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

In questo esempio `b` è la rappresentazione (serializzazione) di `a` in una stringa, e `c` è una copia di `a` generata dalla deserializzazione di `b`.

CPickle può anche serializzare e deserializzare da e su un file:

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

3

Panormaica di web2py

3.1 Avvio

web2py è distribuito in pacchetti binari per Windows e Mac OS X. E' scaricabile anche una versione in codice sorgente che può essere eseguita su Windows, Mac, Linux ed altri sistemi Unix. Le versioni binarie per Windows e per Mac OS X già includono il necessario interprete Python. La versione con codice sorgente presuppone che Python sia già installato nel computer. web2py non richiede installazione. Per iniziare è sufficiente scompattare il file zip per il proprio sistema operativo ed eseguire il corrispondente file `web2py`.

Su Windows:

```
web2py.exe
```

Su Mac OS X:

```
open web2py.app
```

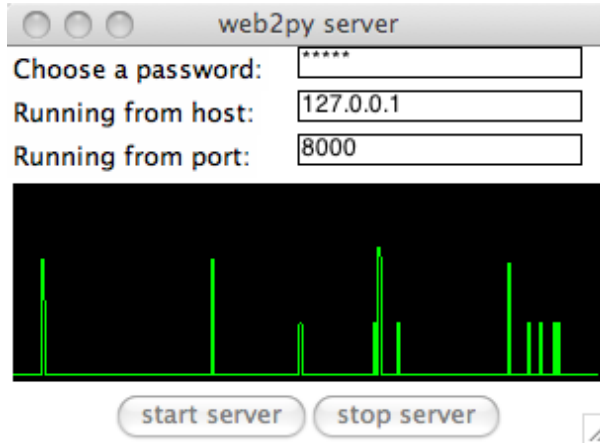
Su Unix e Linux, dal pacchetto del codice sorgente:

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

Il programma web2py accetta svariate opzioni sulla linea di comando che sono discusse in seguito.

Di default web2py visualizza una finestra di avvio:

e successivamente visualizza una finestra che chiede di inserire una password di amministrazione (valida solamente per la sessione corrente), l'indirizzo IP dell'interfaccia di rete utilizzata dal server web e un numero di porta sulla quale servire le richieste. Di default web2py è configurato per utilizzare l'indirizzo 127.0.0.1:8000 (la porta 8000 su *localhost*), ma può utilizzare un indirizzo IP e una porta qualsiasi tra quelli disponibili sul computer. Per conoscere l'indirizzo IP del proprio computer aprire una finestra della linea di comando e digitare il comando `ipconfig` su Windows o `ifconfig` su OS X and Linux. In questo libro si presuppone che web2py sia in esecuzione sulla porta 8000 di localhost (127.0.0.1:8000). Si può utlizzare 0.0.0.0:80 per rendere disponibile web2py su tutte le interfacce di rete del computer.



Se non si inserisce la password di amministrazione l'interfaccia amministrativa rimane disabilitata. Questa è una misura di sicurezza per evitare di esporre l'interfaccia di rete su una connessione pubblica non sicura.

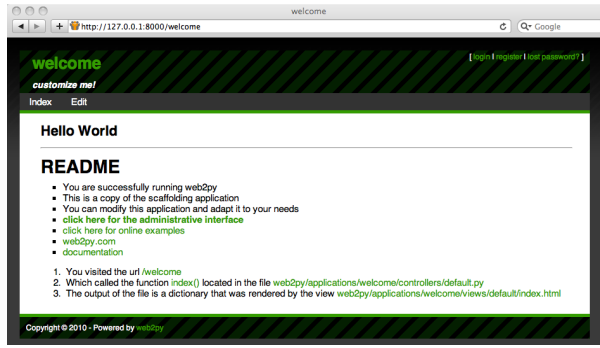
L'interfaccia amministrativa **admin** è accessibile esclusivamente da localhost a meno di non eseguire web2py tramite il modulo mod_proxy di Apache. Se **admin** riconosce di essere dietro un proxy il cookie di sessione è impostato in modalità sicura e il login ad **admin** non è attivato a meno che la comunicazione tra client e proxy sia su una connessione sicura (HTTPS). Questa è un'altra misura di sicurezza. Tutte le comunicazione tra il client e l'applicazione **admin** devono essere locali o cifrate altrimenti un attaccante potrebbe tentare un attacco di tipo *man-in-the-middle* (dove l'attaccante si interpone tra il client ed il server ed ascolta le comunicazioni) o un attacco di tipo *replay* (dove l'attaccante si impossessa delle credenziali di autenticazione e simula l'identità del client) per eseguire codice malevolo sul server.

Dopo che la password di amministrazione è stata impostata web2py avvia il browser web alla pagina:

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

Nel caso che il computer non abbia impostato il browser di default questa

operazione deve essere eseguita manualmente inserendo l'URL nel browser.



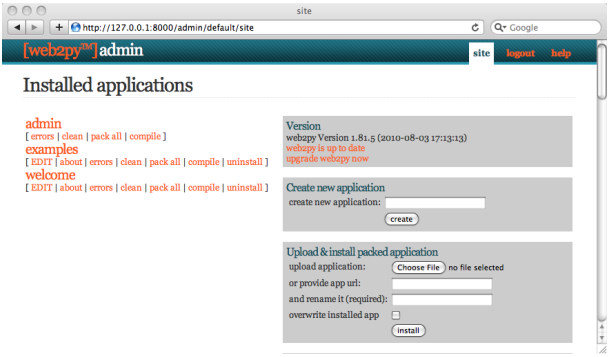
Cliccando su "administrative interface" si raggiunge la pagina di login per l'applicazione di amministrazione.

La password di amministratore è quella inserita all'avvio di web2py.

Esiste un solo amministratore, e pertanto una sola password di amministrazione. Per motivi di sicurezza lo sviluppatore deve utilizzare una nuova password ogni volta che web2py viene avviato a meno che non sia impostata l'opzione <recycle>. Questo comportamento non è previsto per le applicazioni sviluppate in web2py.

Dopo che l'amministratore si collega a web2py il suo browser è rediretto alla

pagina *site*.



Questa pagina elenca tutte le applicazioni web2py installate e consente all'amministrator
di gestirle. web2py include tre applicazioni:

- Un applicazione chiamata **admin** che è quella che si sta usando ora.
- Una applicazione **examples** con la documentazione interattiva online ed una replica del sito di web2py.
- Una applicazione **welcome**. Questa è il template di base per ogni altra nuova applicazione creata in web2py ed è definita come l'applicazione per "l'intelaiatura di base" (*scaffolding*). Questa è anche l'applicazione proposta di default all'utente.

Le applicazioni di web2py già pronte per l'utilizzo sono definite *appliance*. Molte appliance gratuite possono essere scaricate da (34). Gli utenti di web2py sono incoraggiati a proporre nuove appliance sia in codice sorgente (open source) che già compilate (closed source).

Dalla pagina *site* dell'applicazione **admin** possono essere eseguite queste operazioni:

- **installazione** di una applicazione compilando il modulo sulla parte destra della pagina definendo un nome per l'applicazione e selezionando il file

contenente l'applicazione compressa (o indicando la URL dove l'applicazione è disponibile).

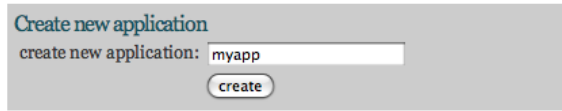
- **disinstallazione** di un'applicazione premendo l'apposito pulsante. Verrà richiesta una conferma prima dell'operazione.
- **creazione** di un'applicazione scegliendo un nome e premendo il pulsante *create*.
- **creare il pacchetto** di distribuzione di un'applicazione premendo il relativo pulsante. Una applicazione scaricata è un file di tipo *tar* che contiene tutti i componenti dell'applicazione, database incluso. Non è mai necessario scompattare questo file perchè web2py lo scompatta automaticamente quando viene installato tramite **admin**.
- **ripulitura** di un'applicazione dai file temporanei (sessioni, errori e file di cache).
- **modifica** di un'applicazione.

Quando si crea una nuova applicazione utilizzando **admin** *viene generato un clone dell'applicazione welcome con all'interno il modello "models/db.py" che definisce un database SQLite, si collega ad esso, instanzia Auth, Crud e Service e li configura. E' anche presente un "controller/default.py" che espone le azioni "index", "download", "user" (per la gestione dell'utente) e "call" (per i servizi). Negli esempi seguenti si presuppone che questi file siano stati cancellati in quanto verranno create da zero delle nuove applicazioni.*

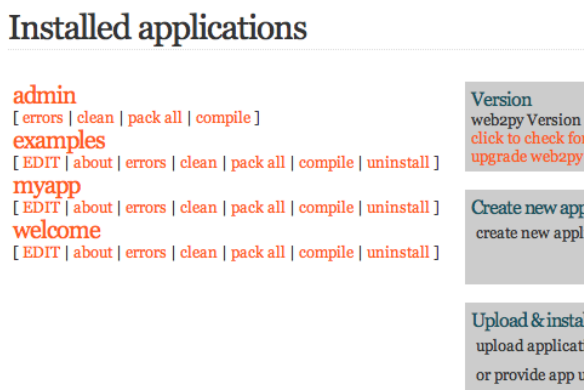
3.2 *Hello!*

Come esempio ecco una semplice applicazione che visualizza all'utente il messaggio "Hello from MyApp". L'applicazione sarà chiamata "myapp", Verrà aggiunto anche un contatore che indica quante volte lo stesso utente visita questa pagina.

Per creare una nuova applicazione è sufficiente inserire il suo nome nel campo a destra nella pagina **site** dell'applicazione **admin**:



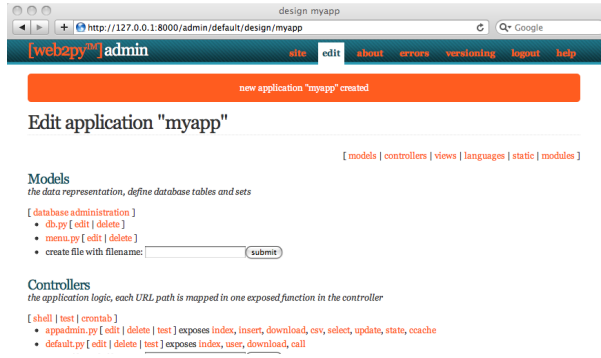
Dopo aver premuto il pulsante *submit* l'applicazione è creata come una copia dell'applicazione *welcome*.



Per eseguire la nuova applicazione utilizzare la seguente URL:

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

Questa, per ora, è una copia dell'applicazione *welcome*. Per modificare l'applicazione premere il pulsante *edit* nell'applicazione appena creata.



La pagina **edit** mostra il contenuto dell'applicazione.

Ogni applicazione creata in web2py consiste di un certo numero di file, la maggior parte dei quali cade in una di sei categorie:

- **modelli**: descrivono la rappresentazione dei dati.
- **controller**: descrivono la rappresentazione logica e il flusso dell'applicazione.
- **viste**: descrivono la presentazione dei dati.
- **linguaggi**: descrivono come tradurre le stringhe dell'applicazione in altre lingue.
- **moduli**: moduli Python che fanno parte dell'applicazione.
- **file statici**: immagini statiche, file CSS (40; 41; 42), file JavaScript (43; 44), ecc.

Tutti questi file sono organizzati seguendo il modello di progettazione MVC (*Model-View-Controller*). Ciascuna sezione nella pagina di *edit* corrisponde ad una sotto-cartella nella cartella dell'applicazione. Cliccando sull'intestazione di ogni sezione si visualizza o si nasconde il suo contenuto. I nomi delle cartelle all'interno della sezione dei file statici sono cliccabili allo stesso modo.

> *Ogni file elencato nelle sezioni corrisponde ad un file fisicamente memorizzato nella sotto-cartella. Ogni operazione eseguita su un file tramite l'interfaccia*

dell'applicazione **admin** *(creazione, cancellazione, modifica) può essere eseguita anche direttamente dalla shell utilizzando il proprio editor favorito.*

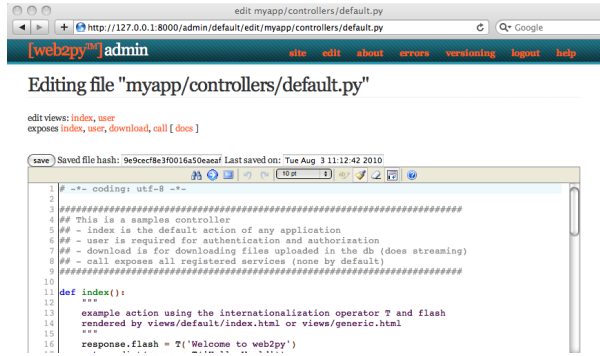
L'applicazione contiene al suo interno altri tipi di file (database, file di sessione, file di errori, ecc.) che non sono elencati nella pagina di *edit* perchè non sono creati o modificati dall'amministratore, ma sono utilizzati direttamente dall'applicazione stessa.

I controller contengono la logica ed il flusso dell'applicazione. Ogni URL è collegata ad una delle funzioni del controller (**azioni**). Ci sono due controller creati di default: "appadmin.py" e "default.py". **appadmin** contiene l'interfaccia amministrativa per il database (e non sarà utilizzato in questo esempio). "default.py" è il controller che dovrà essere modificato e che è chiamato quando nella URL non è indicato uno specifico controller.

Modificare la funzione "index", nel controller "default.py" nel seguente modo:

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

Ecco come apparirà il contenuto del controller nell'editor online:

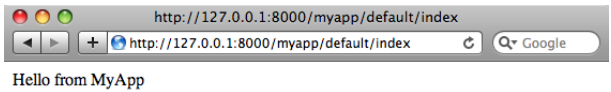


Memorizzare le modifiche e tornare indietro alla pagina *edit*. Cliccare sul link *index* per visitare la pagina appena creata.

Ora, quando si visita la URL

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

verrà eseguita l'azine *index* del controller *default* dell'applicazione "myapp'. Questa azione ritorna una stringa che viene visualizzata dal browser in questo modo:



Modificare ora la funzione *index* in questo modo:

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

Modificare inoltre nella pagina di *edit* la vista *default/index.html* (il nuovo file associato con l'azione creata) nel seguente modo:

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

Ora l'azione ritorna un dizionario contenente un elemento `message`. Quando una azione ritorna un dizionario web2py cerca una vista con il nome `[controller]/[funct` e la esegue. [extension] è l'estensione (che indica il formato del tipo di dati restituiti) richiesta e se non specificata è automaticamente impostata a "html". In questo caso la vista è un file HTML che incorpora codice Python all'interno dei tag {{ e }}. In questo esempio `{{=message}}` indica a web2py di sostistuire

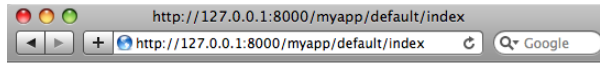
questo codice con il valore contenuto nell'elemento `message` del dizionario restituito dall'azione *index*. E' da notare che `message` non è una parola chiave di web2py ma è definito dall'azione. Per ora non è stata utilizzata nessuna parova chiave di web2py.

Se web2py non trova la vista richeista utilizza la vista "generic.html" che è presente in ogni applicazione.

> *Se viene utilizzata un'altra estensione al posto di "html" (per esempio "json") e la relativa vista "[controller]/[function].json" non esiste, web2py cerca una vista dal nome "generic.json". web2py contiene le viste "generic.html", "generic.json", "generic.xml" e "generic.rss". Queste viste generiche possono essere modificate singolarmente in ciascuna applicazione e viste aggiuntive possono essere facilmente aggiunte.*

Per altre informazioni su questo argomento consultare il capitolo 9.

Tornando alla pagina di *edit* e cliccando sull'azione *index* verrà visualizzata la seguente pagina HTML:



Hello form MyApp

3.3 *Contiamo!*

Verrà ora aggiunto un contatore alla pagina creata nell'esempio precedente che conterà quante volte lo stesso visitatore visualizza la pagina. web2py traccia le visite in modo automatico e trasparente tramite le sessioni e i cookie. Per ciascun nuovo visitatore viene creata una sessione a cui è assegnato un "*session_id*" univoco. La sessione è un contenitore per le variabili memorizzato sul server. Il *session_id* univoco è inviato al browser tramite un cookie. Quando il visitatore richieste un'altra pagina dalla stessa applicazione il browser rimanda indietro il cookie che è recuperato da web2py che ripristina la sessione corrispondente.

Modificare il controller *default.py* nel seguente modo per utilizzare le sessioni:

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

E' da notare che `counter` non è una parola chiave di web2py ma `session` lo è. In questo caso web2py controlla se la variabile `counter` è presente in `session` e nel caso che non sia presente la crea ed imposta il suo valore ad 1. Se `counter` già esiste web2py incrementa il suo valore di 1. Infine il valore del contatore viene passato alla vista. Un modo più compatto di scrivere la stessa funzione è:

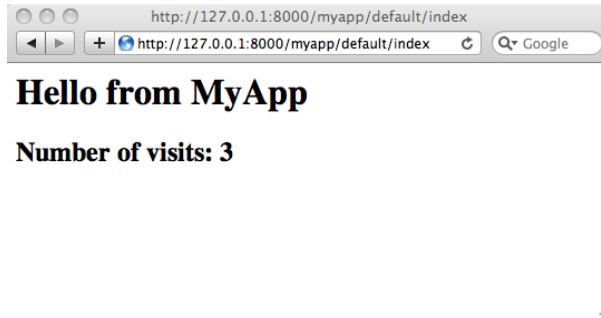
```
def index():
    session.counter = (session.counter or 0) + 1
    return dict(message="Hello from MyApp", counter=session.counter)
```

Modificare la vista per aggiungere una linea che visualizza il valore del contatore:

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

```
5       <h2>Number of visits: {{=counter}}</h2>
6     </body>
7   </html>
```

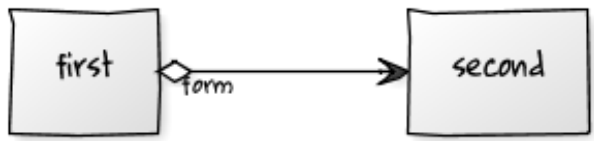
Quando si visualizza nuovamente la pagina *index* si ottiene la seguente pagina HTML:



Il contatore è associato ad ogni visitatore ed è incrementato ogni volta che il visitatore ricarica la pagina. Ogni visitatore vede il proprio contatore di pagina.

3.4 *Come ti chiami?*

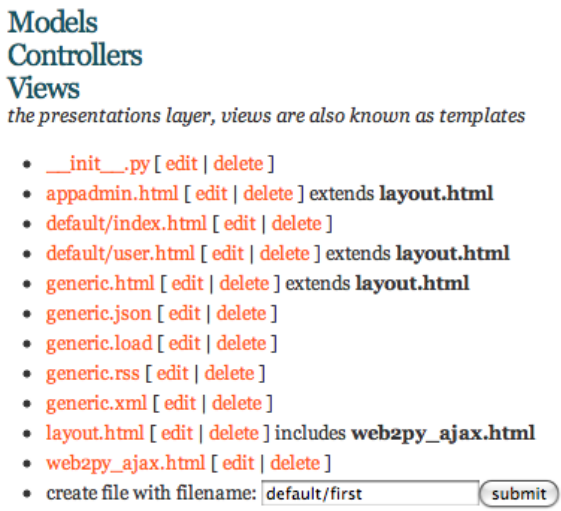
Saranno create ora due pagine (*first* e *second*): la prima chiederà il nome del visitatore tramite una form ed eseguirà una redirezione alla seconda che saluterà il visitatore chiamandolo per nome.



Scrivere le relative azioni nel controller di default:

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

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

Creare poi la vista "default/first.html" per la prima azione:



e scrivere:

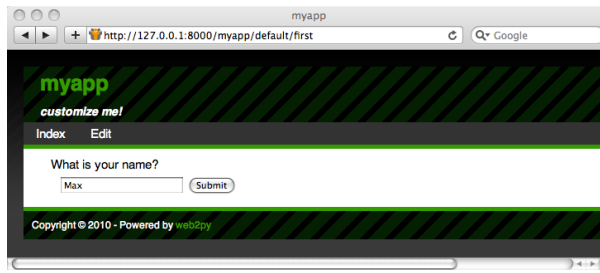
```
{{extend 'layout.html'}}
What is your name?
<form action="second">
  <input name="visitor_name" />
  <input type="submit" />
</form>
```

Crere infine la vista "default/second.html" per la seconda azione:

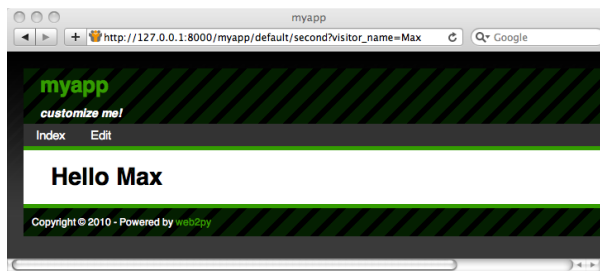
```
{{extend 'layout.html'}}
<h1>Hello {{=request.vars.visitor_name}}</h1>
```

Nelle due viste è stato esteso lo schema di base "layout.html" che è fornito con web2py. Il layout mantiene coerente il *look & feel* delle due pagine. Il file di layout può essere editato e sostituito facilmente poichè contiene principalmente codice HTML.

Se ora si visita la priam pagina, inserendo il proprio nome:



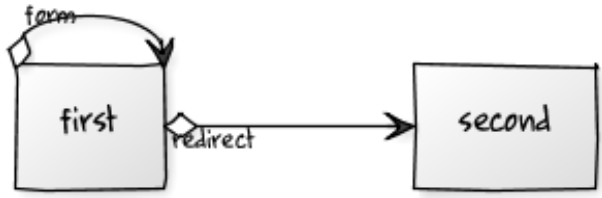
ed inviando la form, si riceverà un saluto:



3.5 *Auto-invio delle form*

Il meccanismo di invio delle form appena visto è molto utilizzato, ma non è una buona pratica di programmazione. Ogni inserimento in una form dovrebbe sempre essere validato e, nell'esempio precedente, il controllo dell'input ricadrebbe sulla seconda azione. In questo modo l'azione che esegue il controllo sarebbe diversa dall'azione che ha generato la form. Questo può

causare ridondanza nel codice. Uno schema migliore per l'invio delle form è di rimandarle alla stessa azione che le ha generate, in questo caso all'azione *first*. L'azione dovrebbe ricevere le variabili, validarle, memorizzarle sul server e ridirigere il visitatore alla pagina *second* che recupera le variabili. Questo meccanismo è chiamato *postback*.



Il controller di default deve essere modificato nel seguente modo per implementare l'auto invio del form:

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

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

Di conseguenza si deve modificare la vista "default/first.html":

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

e la vista "default/second.html" deve recuperare il nome dalla sessione (`session`) invece che dalle variabili della richiesta (`request.vars`:

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

Dal punto di vista del visitatore il comportamento dell'auto-invio del form è esattamente lo stesso dell'implementazione precedente. Per ora non è stata aggiunta nessuna validazione ma è chiaro che in questo esempio la validazione debba essere eseguita dalla prima azione.

Questo approccio è migliore perchè il nome del visitatore è memorizzato nella sessione e può essere acceduto da tutte le azioni e le viste dell'applicazione senza doverlo passare esplicitamente.

Inoltre se l'azione 'second" è chiamata prima che il nome del visitatore è impostato nella sessione sarà visualizzato "Hello anonymous" perchè `session.visitor_name` ha il valore `None`. In alternativa potremmo aver aggiunto il seguente codice nel controller (all'interno o all'esterno della funzione *second*):

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

Questo è un meccanismo generico per obbligare l'autorizzazione nei controller, sebbene nel capitolo 8 sarà descritto un metodo più completo per gestire l'autenticazione e l'autorizzazione.

Con web2py si può fare un passo in più e chiedere a web2py di generare il form con la validazione. web2py fornisce delle funzioni ausiliarie (*helper* per FORM, INPUT, TEXTAREA e SELECT/OPTION) con lo stesso nome dei corrispondenti tag HTML. Gli *helper* possono essere utilizzati per generare i form sia nei controller che nelle viste.

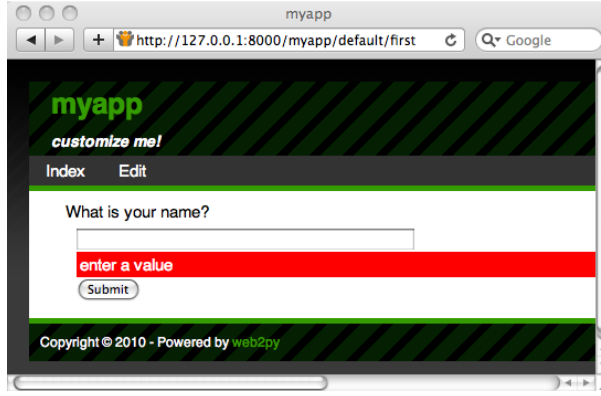
Per esempio questo è uno dei modi possibili di riscrivere l'azione *first*:

```
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('second'))
    return dict(form=form)
```

dove si definisce un form che contiene due tag di tipo INPUT. Gli attributi dei tag di input sono specificati dagli argomenti con nome che iniziano con un underscore. L'argomento con nome `requires` non è un tag HTML (perchè non inizia con un underscore) ma imposta un validatore per il valore del campo visitor_name. L'oggetto `form` può essere facilmente serializzato in HTML incorporandolo nella vista "default/first.html":

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

Il metodo `form.accepts` esegue i controlli indicati nei validatori. Se il form auto-inviato passa la validazione le variabili vengono memorizzate nella sessione e l'utente viene reindirizzato come nell'esempio precedente. Se il form non passa la validazione i messaggi d'errore sono inseriti nel form e mostrati all'utente:

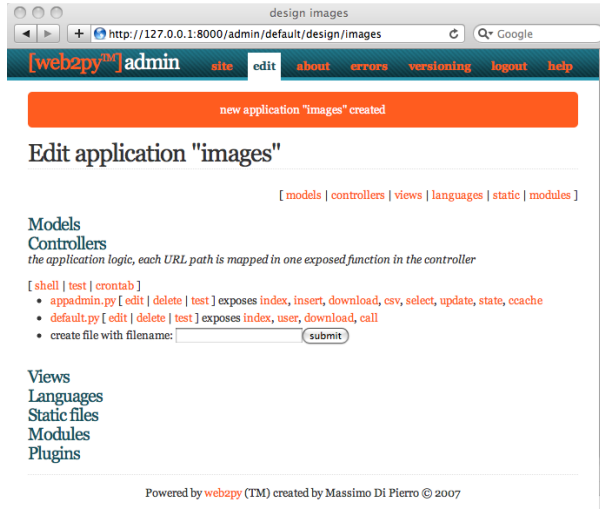


Nella prossima sezine si vedrà come i form possono essere generati automaticamente da un modello di dati.

3.6 *Un Blog con immagini*

Ecco un esempio più complesso. Si vuole creare un'applicazione web che consenta all'amministratore di inviare immagini e di dar loro un nome e agli utenti di vedere le immagini ed inviare commenti su di esse.

Come nell'esempio precedente creare la nuova applicazione dalla pagina **site** nell'applicazione **admin** ed entrare nella pagina **edit** dell'applicazione appena creata:



Procedere quindi alla creazione del modello, la rappresentazione dei dati persistenti dell'applicazione (le immagini da caricare, i loro nomi ed i commenti ad esse relativi). Andrà prima creato e modificato il file di modello "db.py". I modelli ed i controlli devono avere l'estensione `.py` perchè sono codice Python. Se l'estensione non è indicata questa è aggiunta automaticamente da web2py. Le viste invece sono create di default con l'estensione `.html` poichè contengono principalmente codice HTML.

Modificare il file "db.py" cliccando sul relativo pulsante *edit*:

ed inserire il seguente codice:

```
1  db = DAL("sqlite://storage.sqlite")
2
3  db.define_table('image',
4     Field('title'),
5     Field('file', 'upload'))
6
7  db.define_table('comment',
8     Field('image_id', db.image),
9     Field('author'),
10    Field('email'),
11    Field('body', 'text'))
12
13 db.image.title.requires = IS_NOT_IN_DB(db, db.image.title)
14 db.comment.image_id.requires = IS_IN_DB(db, db.image.id, '%(title)s')
15 db.comment.author.requires = IS_NOT_EMPTY()
16 db.comment.email.requires = IS_EMAIL()
17 db.comment.body.requires = IS_NOT_EMPTY()
18
19 db.comment.image_id.writable = db.comment.image_id.readable = False
```

Ecco l'analisi del codice linea per linea:

- Linea 1: definisce la variabile globale `db` che rappresenta la connessione al database. In questo esempio è una connessione ad un database SQLite memorizzato nel file "applications/images/databases/storage.sqlite". Nel caso di database di tipo SQLite se questi non esistono vengono creati automaticamente. Si può cambiare il nome del file ed anche la variable globale `db`, ma quest'ultima conviene mantenerla con questo nome, perchè è semplice da ricordare.

- Linee 3-5: definiscono la tabella "image". `define_table` è un metodo dell'oggetto `db`. Il primo argomento, "image" è il nome della tabella che si sta definendo. Gli altri argomenti sono i campi della tabella. Questa tabella ha un campo chiamato "title", un campo chiamato "file" ed un campo chiamato "id" per memorizzare la chiave primaria della tabella. ("id" non è dichiarato esplicitamente perchè tutte le tabelle in web2py hanno di default un campo "id"). Il campo "title" è una stringa ed il campo "file" è di tipo "upload". "upload" è un tipo speciale di campo utilizzato dal DAL (*Data Abstraction Layer*) di web2py per memorizzare i nomi dei file caricati dall'utente. web2py è in grado di caricare i file (tramite streaming se sono di grande dimensione), rinominarli in un modo sicuro e memorizzarli. Dopo la definizione di una tabella web2py intraprende una delle seguenti azioni: a) se la tabella non esiste, questa viene creata; b) se la tabella esiste e non corrisponde alla definizione, la tabella esistente è modificata di conseguenza e se un campo è di un tipo differente, web2py tenta di convertirne i contenuti; c) se la tabella esiste e corrisponde alla definizione, in questo caso web2py non fa nulla. Questo comportamento è chiamato "migrazione". In web2py le migrazioni sono automatiche, ma possono essere disattivate per ogni tabella passando l'argomento "migrate=False" come ultimo argomento in `define_table`.

- Linee 7-11: definiscono un altra tabella chiamata "comment". Un commento ha un autore (*author*), un indirizzo email (*email*) per memorizzare l'indirizzo email dell'autore del commento, un corpo (*body*) di tipo testo (*text*) per memorizzare il commento inviato dall'autore ed un id dell'immagine (*image_id*) di tipo riferimento (*reference*) che punta a `db.image` tramite il campo "id".

- Linea 13: `db.image.title` rappresenta il campo "title" della tabella "image". L'attributo `requires` permette di impostare dei vincoli che saranno applicati dai form di web2py. In questo caso si richiede che il "title" sia univoco: `IS_NOT_IN_DB(db, db.image.title)`. Gli oggetti che rappresentano questi vincoli sono chiamati "validatori". Validatori multipli possono essere raggruppati in una lista. I validatori sono eseguiti nell'ordine in cui appaiono. `IS_NOT_IN_DB(a, b)` è un validatore speciale che controlla che il valore del campo *b* per un nuovo record non sia già in *a*.

- Linea 14: definisce il vincolo per cui il campo "image_id" della tabella "comment" deve essere in `db.image.id`. Per come è definito il database questo è già stato dichiarato nella definizione della tabella "comment". In questo modo l'indicazione del vincolo è esplicita nel modello e viene controllata da web2py durante la validazione dei form, quando è inserito un nuovo commento. In questo modo i valori non validi non vengono propagati dal form al database. E' anche richiesto che il campo "image_id" sia rappresentato dal "title" (`'%(title)s'`) del record corrispondente.
- Linea 18: indica che il campo "image_id" della tabella "comment" non deve essere mostrato nei form (`writable=False`) nemmeno in quelli a sola lettura (`readable=False`).

Il significato dei validatori nelle linee 15-17 dovrebbe essere evidente.

E' da notare che il validatore

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

potrebbe essere omesso (e sarebbe implicito) se si specificasse un formato per la rappresentazione dell'immagine

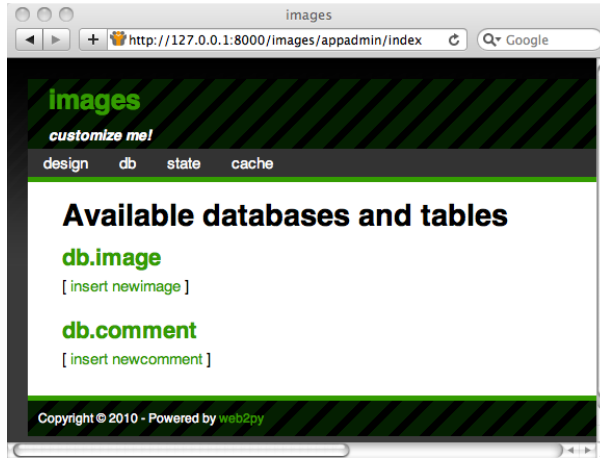
```
1 db.define_table('image',....,format='%(title)s')
```

dove il formato può essere una stringa o una funzione che richiede un record e ritorna una stringa.

Quando il modello è definito e se non ci sono errori web2py crea un'interfaccia amministrativa dell'applicazione per gestire il database alla quale accedere tramite il pulsante "database administration" nella pagina *edit* oppure direttamente con la URL:

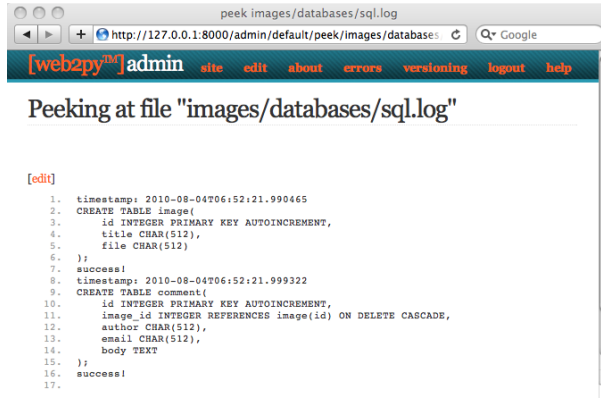
```
1 http://127.0.0.1:8000/images/appadmin
```

Ecco come si presenta l'interfaccia amministrativa **appadmin**:



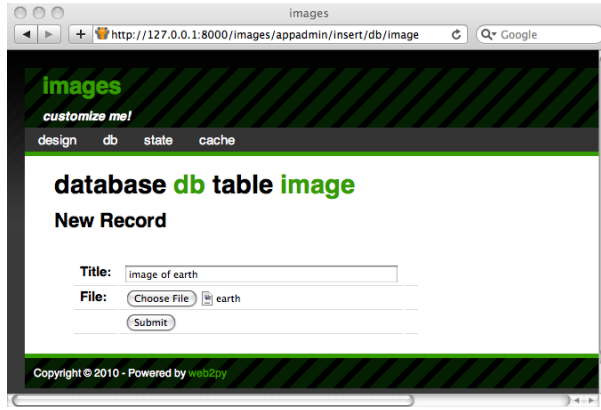
Questa interfaccia è codificata nel controller chiamato "appadmin.py" e nella vista corrispondente "appadmin.html". Da qui in poi questa interfaccia sarà identificata semplicemente come **appadmin**. **appadmin** consente all'amministratore di inserire nuovi record nel database, modificare e cancellare i record esistenti, scorrere le tabelle ed eseguire delle unioni di database (*join*).

La prima volta che **appadmin** viene acceduta il modello è eseguito e le tabelle vengono create. Il DAL di web2py traduce il codice Python in comandi SQL specifici per il database selezionanto (SQLite in questo esempio). Il codice SQL generato può essere ispezionato dalla pagina *edit* cliccando sul pulsante "sql.log" nella sezione "models". Questo link non è presente finchè le tabelle non vengono create.



Se si modifica il modello e si accede nuovamente ad **appadmin** web2py genererebbe codice SQL per alterare le tabelle esistenti. Il codice cgenerato è memorizzato in "sql.log".

Ora in **appadmin** si provi ad inserire un nuovo record immagine:



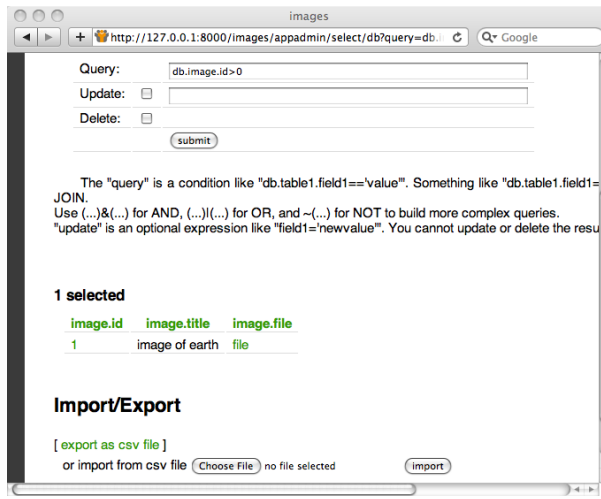
web2py ha trasformato il campo di "upload" di `db.image.file` in un form di caricamento per i file. Quando il form è inviato dall'utente ed un'immagine è caricata il file è rinominato in un modo sicuro (che preserva l'estensione), è memorizzato con il nuovo nome nella cartella "uploads" dell'applicazione ed il nuovo nome è memorizzato nel campo `db.image.file`. Questo processo

previene gli attacchi di tipo "directory traversal".

Ogni campo è visualizzato tramite un *widget*. I widget di default possono essere sostituiti. Quando si clicca sul nome di una tabella in **appadmin** web2py seleziona tutti i record della tabella corrente identificata dalla seguente query del DAL:

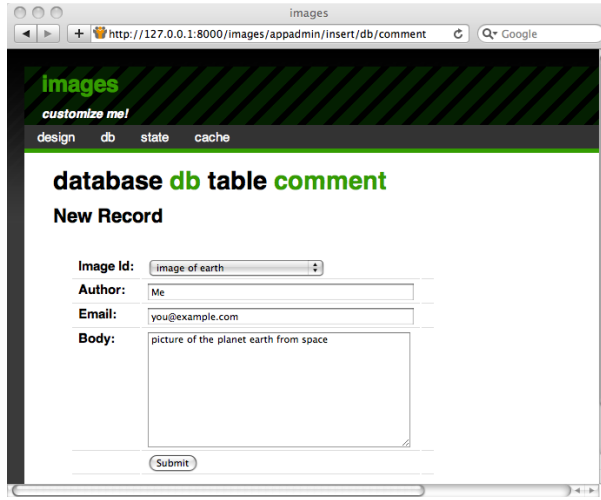
```
db.image.id > 0
```

e visualizza il risultato:



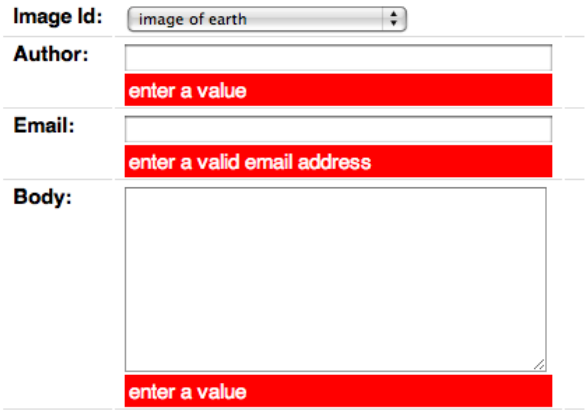
Si può selezionare un gruppo differente di record modicando la query SQL e premendo "apply".

Per modificare o cancellare un singolo record cliccare sul numero id del record.



A causa del validatore `IS_IN_DB` il campo di riferimento "image_id" è visualizzato con un menu a tendina. Gli elementi nel menu a tendina sono memorizzati come chiavi (`db.image.id`), ma sono rappresentati dal `db.image.title`, come specificato nel validatore.

I validatori sono oggetti potenti che sono in grado di rappresentare i campi, i valori dei campi nei filtri, generare gli errori e formattare i valori estratti dal campo. Il seguente schema mostra cosa succede quando si invia un form che non supera la validazione:



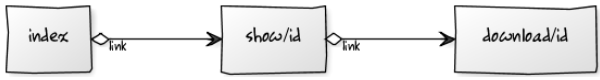
Gli stessi form che sono generati automaticamente da **appadmin** possono anche essere generati da codice con l'helper `SQLFORM` ed inseriti nelle applicazioni. Questi moduli sono configurabili e gestibili tramite CSS.

Ciascuna applicazione ha la sua **appadmin** che può pertanto essere modificata senza generare problemi per le altre applicazioni presenti.

Finora l'applicazione è in grado di memorizzare i dati, ai quali si può accedere tramite **appadmin**. L'accesso ad **appadmin** è riservato al'amministratore e **appmin** non deve essere considerata come un'interfaccia web da usare in produzione. Per questo ora verranno creati:

- Una pagina "index" che visualizza tutte le immagini disponibili ordinate per titolo con il link alle pagina di dettaglio per ogni immagine.
- Una pagina "show/[id]" che visualizza l'immagine richiesta e permette all'utente di visualizzare e inviare commenti.
- Un'azione "download/[name]" che consente di scaricare le immagini caricate.

Ecco la schematizzazione di queste azioni:



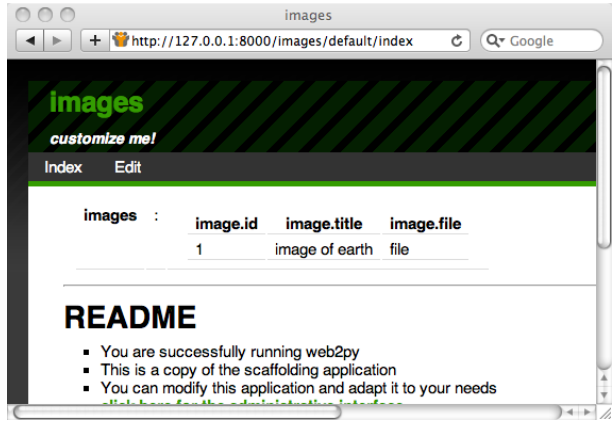
Tornando alla pagina *edit* modificare il controller "default.py" sostituendo il suo contenuto con:

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

Questa azione ritorna un dizionario. Le chiavi degli elementi del dizionario sono interpretate come variabili passate alla vista associata con l'azione. Se la vista non è presente l'azione è visualizzata dalla vista "generic.html" che è presente in ogni applicazione web2py.

L'azione *index* seleziona tutti i campi (`db.image.ALL`) dalla tabella *image* e li ordina per `db.image.title`. Il risultato della selezione è un oggetto `Rows` che contiene i record e che viene assegnato alla variabile locale `images` che verrà poi restituita alla vista tramite il dizionario. `images` è un oggetto iterabile ed i suoi elementi sono le righe selezionate nella tabella. Per ciascuna riga le colonne sono accedute come dizionari: `images[0]['title']` o `images[0].title`.

Nel caso non sia presente una vista il dizionario è visualizzto dalla vista "views/generic.html" è l'utilizzo dell'azione *index* avrebbe il seguente risultato:



Poichè non è stata creata nessuna vista per questa azione web2py visualizza i record in una semplice tabella.

Per creare una vista per l'azione *index* ritornare alla pagina di *edit* e modificare la vista "default/index.html" sostituendo il suo contenuto con:

```
1 {{extend 'layout.html'}}
2 <h1>Current Images</h1>
3 <ul>
4 {{for image in images:}}
5 {{=LI(A(image.title, _href=URL("show", args=image.id)))}}
6 {{pass}}
7 </ul>
```

La prima cosa da notare è che una vista è del semplice codice HTML con l'aggiunta degli speciali tag "{{" e "}}". Il codice incluso nei tag "{{" e "}}" è codice Python con una particolarità: l'indentazione è irrilevante. I blocchi di codice iniziano con linee che terminano con i due punti (:) e terminano con una linea che inizia con la parola chiave *pass*. In alcuni casi la fine del blocco è evidente dal contesto e l'utilizzo di *pass* non è necessario.

Le linee 5-7 ciclano su tutte le immagini e per ogni immagine eseguono il codice:

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

che genera codice HTML con i tag `<li>...</li>` che contiengono i tag `<a href="...">...</a>` che contiene `image.title`. Il valore del link (attributo *href*) è:

```
URL('show', args=image.id)
```

cioè la URL dell'applicazione e del controller che ha generato la richiesta corrente che richiede la funzione chiamata "show" passandole un singolo argomento `args=image.id`. `LI`, `A`, e le altre funzioni sono helper di web2py che creano i corrispondenti tag HTML. Gli argomenti di queste funzioni sono interpretati come oggetti serializzati ed inseriti all'interno del tag. Gli argomenti con nome che iniziano con underscore (per esempio `_href`) sono interpretati come attributi del tag ma senza l'underscore. Per esempio `_href` è l'attributo `href`, `_class` è l'attributo `class`, ecc.

Per esempio il seguente codice:

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

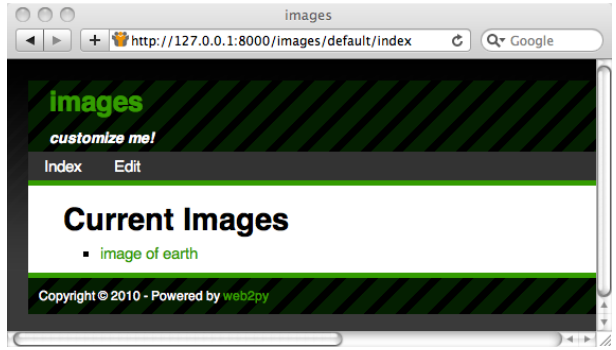
è trasformato in:

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

Alcuni helper (`INPUT`, `TEXTAREA`, `OPTION` e `SELECT`) dispongono anche di attributi con nome speciali che non iniziano con underscore (`value` e `requires`). Questi attributi sono importanti per costruire form personalizzati e saranno discussi in seguito. Tornado indietro alla pagina di *edit* ora indica che *default.py* espone *index*. Cliccando su *index* si esegue l'azione e si raggiunge la pagina appena creata:

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

che visualizza:



Se si clicca sul link di un'immagine si viene reindirizzati a:

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

e questo genera un errore poichè non esiste ancora un'azione chiamata "show" nel controller "default.py".

Modicare il controller "default.py" e sostituire il suo contenuto con:

```
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().first()
    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)
```

Il controller contiene due azioni: "show" e "download". L'azione "show" seleziona l'immagine con l'`id` indicato negli argomenti della richiesta con tutti i relativi commenti e passa tutti i dati recuperati alla vista "default-/show.html".

L'id dell'immagine identificato da:

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

in "default/index.html", può essere acceduto con: `request.args(0)` dall'azione "show".

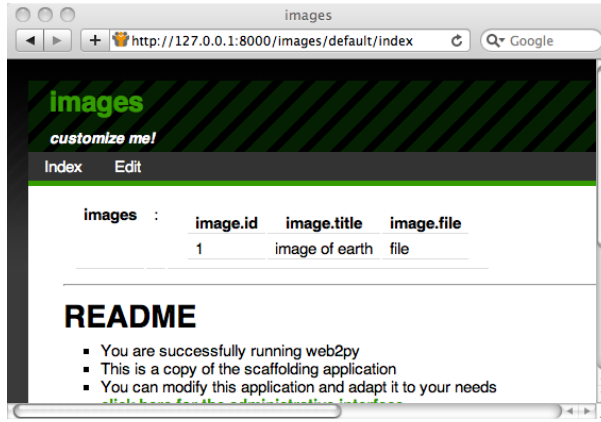
L'azione "download" si aspetta un nome di file in `request.args(0)`, costruisce un path alla cartella dove il file dovrebbe essere e lo manda al cliente. Se il file è di grandi dimensioni viene inviato in streaming all'utente per evitare problemi di sovraccarico della memoria.

Notare nelle seguenti righe:

- Linea 7: crea un SQLFORM di inserimento per la tabella `db.comment` utilizzando solamente i campi specificati.
- Linea 8: imposta il valore per il campo di riferimento che non fa parte del form in quanto non è elencato tra i campi specificati prima.
- Linea 9: elabora il form inviato (le variabili del form sono in `request.vars`) all'interno della sessione corrente (la sessione è utilizzata per evitare un doppio invio del form e per tener traccia della navigazione). Se le variabili sono validate il nuovo commento è inserito nella tabella `db.comment`; in caso contrario il form è modificato per includere i messaggi d'errore (per esempio, nel caso che l'indirizzo di email dell'autore non sia valido). Tutto questo è fatto nella sola linea 9!
- Linea 10: è eseguita solamente se il form è accettato, dopo che il record è stato inserito nella tabella del database. `response.flash` è una variabile di web2py che viene visualizzata nelle viste ed è usata per notificare agli utenti che una certa operazione è avvenuta.
- Linea 11: seleziona tutti i commenti dell'immagine corrente.

L'azione "download" è già definita nel controller "default.py" dell'applicazione "welcome" che viene utilizzata come schema di base delle nuove applicazioni.

L'azione "download" non restituisce un dizionario e così non necessita di una vista. L'azione "show" dovrebbe invece avere una vista. Tornare alla pagina di *edit* e creare una nuova vista chiamata "default/show.html" digitando "default/show" nel campo di creazione di una nuova vista:

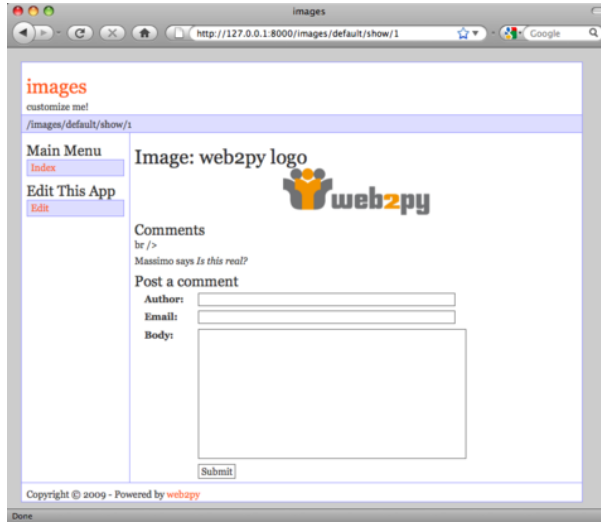


Modificare questo nuovo file e sostituire il suo contenuto con:

```
{{extend 'layout.html'}}
<h1>Image: {{=image.title}}</h1>
<center>
<img width="200px"
     src="{{=URL('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}}
```

Questa vista visualizza **image.file** richiamando l'azione "download" all'interno dei tag `<img ... />`. Se sono presenti commenti vengono tutti visualizzati tramite un ciclo.

Ecco come la pagina appare all'utente:



Quando un utente invia un commento tramite questa pagina il commento è memorizzato nel database ed aggiunto alla fine della pagina stessa.

3.7 *Aggiungere le funzionalità CRUD (Create, Read, Update, Delete)*

seb2py include anche un'API CRUD (*Create, Read, Update, Delete*) che semplifica ancor di più la gestione delle form. Per utilizzare CRUD è necessiario definirlo da qualche parte come per esempio nel modulo "db.py":

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

> *Queste due linee sono già presenti nel modello creato di default con la nuova applicazione*

L'oggetto `crud` dispone di metodi di alto livello come, per esempio:

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

che può essere utilizzato per sostituire il seguente schema di programmazione:

```
form = SQLFORM(table)
if form.accepts(request.post_vars,session):
    session.flash = '...'
    redirect('...')
```

Ecco come riscrivere l'azione "show" utilizzando CRUD ottenendo alcuni miglioramenti:

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

Prima di tutto notare che è stata usata la seguente sintassi:

```
db.image(request.args(0)) or redirect(...)
```

per recuperare il record richiesto. Poichè `table(id)` restituisce None se il record non è presente è possibile utilizzare `or redirect(...)` in una sola linea.

L'argomento `next` di `crud.create` è la URL a cui reindirizzare il form se viene accettato. L'argomento `message` è il messaggio che verrà visualizzato in caso di accettazione del form. Ulteriori informazioni sulla API CRUD sono disponibili nel capitolo 7.

3.8 Aggiungere l'autenticazione

L'API di web2py per il controllo d'accesso basato sui ruoli (RBAC, *Role-based Access Control*) è molto sofisticata, ma per ora si vedrà come restringere l'accesso alle azioni agli utenti autenticati, posticipando al capitolo 8 una discussione più dettagliata.

Per limitare l'accesso agli utenti autenticati, è necessario completare tre passaggi. In un modello, per esempio "db.py" è necessario aggiungere:

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

Nel controller è necessario aggiungere un'azione:

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

Infine le funzioni che devono essere riservate devono essere decorate:

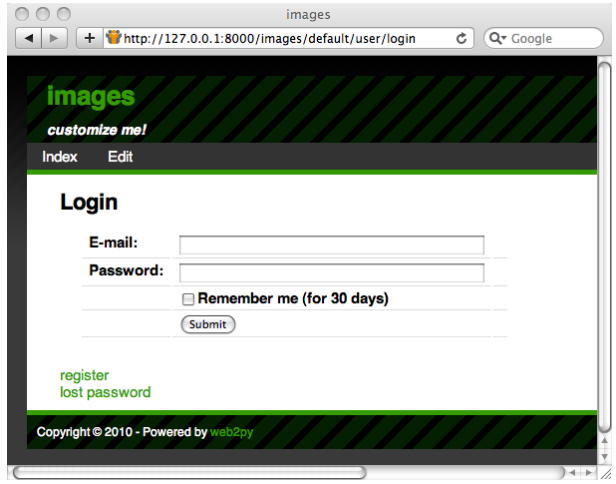
```
@auth.requires_login()
def show():
    image = db.image(request.args(0)) or redirect(URL('index'))
    db.comment.image_id.default = image.id
    form = crud.create(db.comment, next=URL(args=image.id),
                       message='your comment is posted')
    comments = db(db.comment.image_id==image.id).select()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

Ora, qualsiasi accesso a

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

richiederà l'accesso. Se l'utente non è collegato sarà rendirizzato a :

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



La funzione `user` espone, tra le altre, le seguenti azioni:

```
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
http://127.0.0.1:8000/images/default/user/request_reset_password
http://127.0.0.1:8000/images/default/user/retrieve_username
http://127.0.0.1:8000/images/default/user/retrieve_password
http://127.0.0.1:8000/images/default/user/verify_email
http://127.0.0.1:8000/images/default/user/impersonate
http://127.0.0.1:8000/images/default/user/not_authorized
```

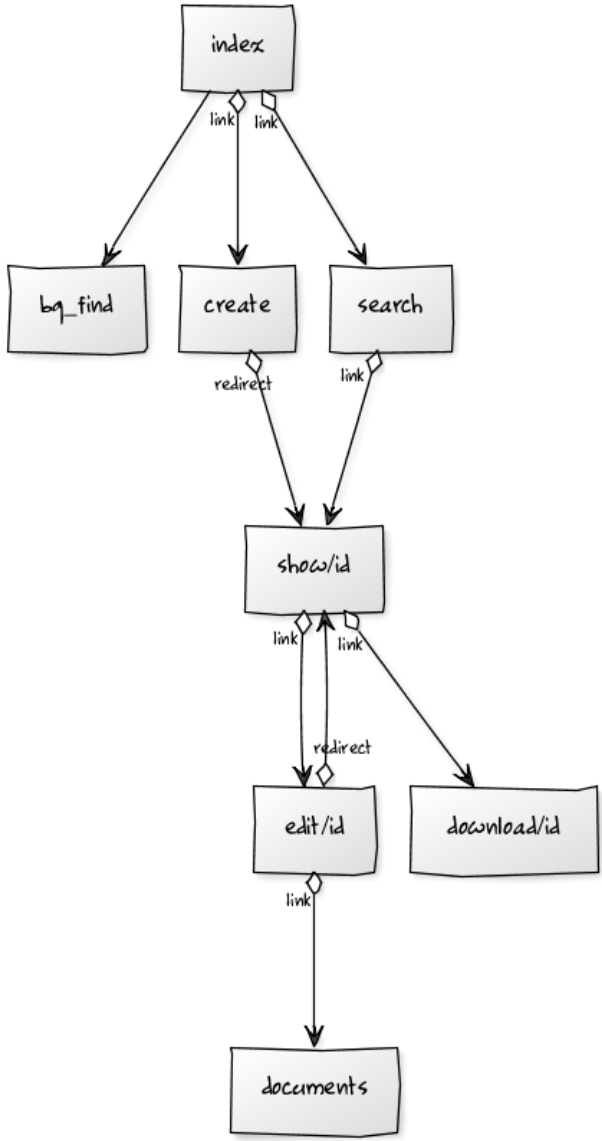
Un utente che si collega per la prima volta dovrà registrarsi per poter leggere ed inviare commenti.

Sia l'oggetto `auth` *che la funzione* `user` *sono già definite nell'applicazione di default. L'oggetto* `auth` *è ampiamente personalizzabile e può gestire la verifica dell'email, i CAPTCHA, l'approvazione della registrazione e metodi alternativi d'accesso tramite plug-in aggiuntivi.*

3.9 *Wiki*

In questa sezione si creerà un wiki. Il visitatore sarà in grado di creare le pagine, ricercarle per titolo, ed editarle. Il visitatore potrà anche inviare commenti (come nell'applicazione precedente) e potrà anche inviare documenti (come allegati alle pagine) e linkarli dalle pagine stesse. Sarà utilizzata la sintassi Markmin per le pagine del wiki. Sarà anche implementata una pagina di ricerca con Ajax, un feed RSS per le pagine e un handler per ricercare le pagine tramite XML-RPC (46).

Il seguente diagramma illustra le azioni da implementare e i link che le connettono tra di loro:



Per iniziare, creare una nuova applicazione chiamata "mywiki".

Il modello deve contenere tre tabelle: *page*, *comment* e *document*. Sia i com-

menti che i documenti fanno riferimento alle pagine perchè appartengono ad una pagina. Un documento contiene un campo *file* di tipo *upload* come nella precedente applicazione.

Ecco il modello completo:

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

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

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

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=auth.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=auth.user_id),
    format='%(name)s')

db.page.title.requires = IS_NOT_IN_DB(db, 'page.title')
db.page.body.requires = IS_NOT_EMPTY()
db.page.created_by.readable = db.page.created_by.writable = False
db.page.created_on.readable = db.page.created_on.writable = False

db.comment.body.requires = IS_NOT_EMPTY()
db.comment.page_id.readable = db.comment.page_id.writable = False
db.comment.created_by.readable = db.comment.created_by.writable = False
db.comment.created_on.readable = db.comment.created_on.writable = False

db.document.name.requires = IS_NOT_IN_DB(db, 'document.name')
db.document.page_id.readable = db.document.page_id.writable = False
db.document.created_by.readable = db.document.created_by.writable = False
db.document.created_on.readable = db.document.created_on.writable = False
```

Modificare il controller "default.py" e creare le seguenti azioni:

- index: elenca tutte le pagine del wiki
- create: crea una nuova pagina con i dati inseriti dall'utente
- show: mostra una pagina del wiki con i commenti e consente di aggiungerne altri
- edit: modifica una pagina esistente
- documents: gestisce i documenti allegati ad una pagina
- download: scarica un documento (come nell'esempio precedente)
- search: visualizza una finestra di recerca e, tramite una chiamata Ajax, restituisce tutti i titoli trovati mentre il visitatore sta ancora scrivendo
- bg_find: la funzione di ricerca Ajax. Restituisce il codice HTML da includere nella pagina di ricerca

Ecco il controller "default.py":

```
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)

@auth.requires_login()
def create():
    "creates a new empty wiki page"
    form = crud.create(db.page, next = URL('index'))
    return dict(form=form)

def show():
    "shows a wiki page"
    this_page = db.page(request.args(0)) or redirect(URL('index'))
    db.comment.page_id.default = this_page.id
    form = crud.create(db.comment) if auth.user else None
    pagecomments = db(db.comment.page_id==this_page.id).select()
    return dict(page=this_page, comments=pagecomments, form=form)

@auth.requires_login()
```

```
25 def edit():
26     "edit an existing wiki page"
27     this_page = db.page(request.args(0)) or redirect(URL('index'))
28     form = crud.update(db.page, this_page,
29          next = URL('show', args=request.args))
30     return dict(form=form)
31
32 @auth.requires_login()
33 def documents():
34     "lists all documents attached to a certain page"
35     this_page = db.page(request.args(0)) or redirect(URL('index'))
36     db.document.page_id.default = this_page.id
37     form = crud.create(db.document)
38     pagedocuments = db(db.document.page_id==this_page.id).select()
39     return dict(page=this_page, documents=pagedocuments, form=form)
40
41 def user():
42     return dict(form=auth())
43
44 def download():
45     "allows downloading of documents"
46     return response.download(request, db)
47
48 def search():
49     "an ajax wiki search page"
50     return dict(form=FORM(INPUT(_id='keyword',
51              _onkeyup="ajax('bg_find', ['keyword'], 'target');")),
52              target_div=DIV(_id='target'))
53
54 def bg_find():
55     "an ajax callback that returns a <ul> of links to wiki pages"
56     pattern = '%' + request.vars.keyword.lower() + '%'
57     pages = db(db.page.title.lower().like(pattern))\
58       .select(orderby=db.page.title)
59     items = [A(row.title, _href=URL('show', args=row.id)) \
60            for row in pages]
61     return UL(*items).xml()
```

Le linee 2-6 forniscono un commento per l'azione *index*. Le linee 4-5 all'interno del commento sono interpretate da Python come codice di test (doctest). I test possono essere eseguiti dall'interfaccia di amministrazione. In questo caso il test verifica che l'azione *index* sia eseguita senza errori.

Le linee 18, 27 e 35 tentano di recuperare un record `page` con l'id `request.args(0)`.

Le linee 13, 20 e 37 definiscono e processano i form di creazione rispettivamente per una nuova pagina, un nuovo commento ed un nuovo documento.

La linea 28 definisce e processa un form di aggiornamento per una pagina wiki.

Alla linea 51 c'è un po' di *magia*. Viene impostato l'attributo `onkeyup` del tag INPUT "keyword". Ogni volta che il visitatore preme o rilascia un tasto il codice Javascript all'interno dell'attributo `onkeyup` viene eseguito nel browser dell'utente. Ecco il codice JavaScript:

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

`ajax` è una funzione definita nel file "web2py_ajax.html" che è incluso nel layout di default "layout.html". Questa funzione richiede tre parametri: la URL dell'azione che esegue la chiamata sincrona ("bg_find"), una lista degli ID delle variabili che devono essere inviate alla funzione (["keyword"]) e l'ID dove deve essere inserita la risposta ("target").

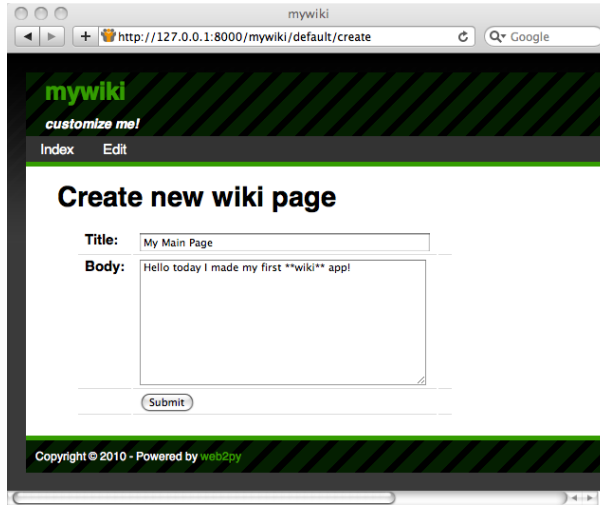
Non appena si digita qualcosa nel campo di ricerca e si rilascia il tasto, il client contatta il server e invia il contenuto del campo 'keyword' e, quando il server risponde, la risposta è inserita nella pagina stessa come codice HTML interno del tag 'target'.

Il tag 'target' è un DIV definito nella linea 75. Potrebbe essere stato definito anche nella vista.

Ecco il codice per la vista "default/create.html":

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

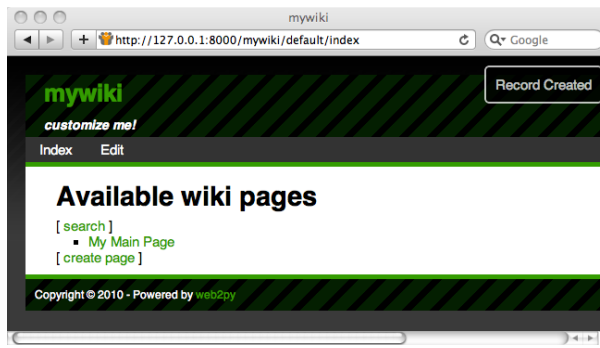
VIsitando la pagina **create** si ottiene:



Ecco il codice per la vista "default/index.html":

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

che genera la seguente pagina:



Ecco il codice per la vista "default/show.html":

```
{{extend 'layout.html'}}
<h1>{{=page.title}}</h1>
[ {{=A('edit', _href=URL('edit', args=request.args))}}
| {{=A('documents', _href=URL('documents', args=request.args))}} ]<br />
{{=MARKMIN(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}}
```

Se si desidera utilizzare la sintatti Markdown invece di quella Markmin, basta importare `WIKI` da `gluon.contrib.markdown` ed utilizzarlo invece dell'helper `MARKMIN`. Alternativamente si può decidere di accettare semplice HTML invece della sintassi Markmin, in questo caso si deve sostituire:

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

con:

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

(in modo da evitare che l'XML non venga controllato, che è il comportamento di default di web2py). Un modo migliore per fare questo è:

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

Impostando `sanitize=True` web2py controllerà i tag non sicuri di XML (come, per esempio, "<script>") evitando le vulnerabilità di tipo XSS.

Se ora si seleziona il titolo di una pagina dalla pagina *index*, sarà visualizzata la pagina che è stata creata:

Ecco il codice della vista "default/edit.html":

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

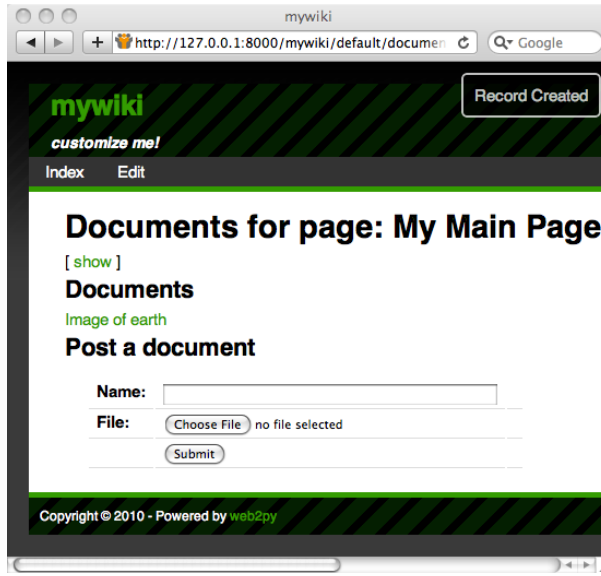
che genera una pagina che sembra quasi identica alla pagina di creazione:

Ecco il codice per la vista "default/documents.html":

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

```
{{=form}}
```

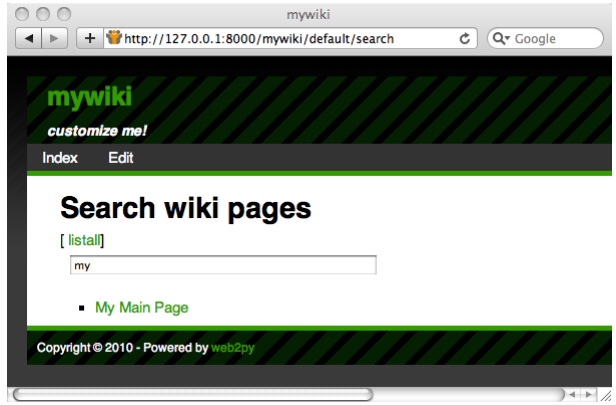
Se si clicca su un documento nella pagina "show", si può gestire il documento allegato alla pagina.



Ecco infine il codice per la vista "default/search.html":

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

che genera il seguente form di ricerca Ajax:



E' possibile anche tentare di richiamare l'azione direttamente visitando, per esempio, la seguente URL:

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

Nell'HTML della pagina di risposta si potrà vedere il codice restituito dalla funzione:

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

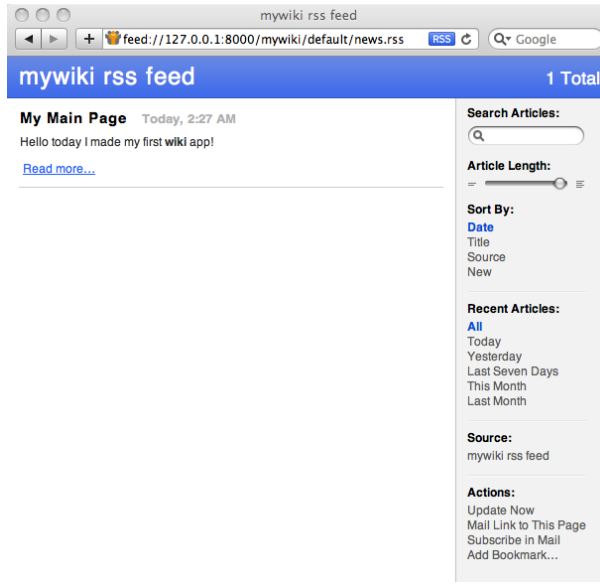
La generazione del feed RSS delle pagine memorizzate è semplice in web2py perchè web2py include `gluon.contrib.rss2`. E' sufficiente aggiungere lea seguente azione al controller di default:

```
def news():
    "generates rss feed form the wiki pages"
    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('show', args=row.id),
               description = MARKMIN(row.body).xml(),
               created_on = row.created_on
               ) for row in pages])
```

e quando si visita la pagina

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

sarà possibile visualizzare il feed (l'output esatto dipende dal visualizzatore di RSS). Il *dict* è automaticamente convertito in RSS, grazie all'estensione.rss nella URL.



web2py include anche *feedparser* per leggere feed esterni.

Ecco infine l'interfaccia XML-RPC per accedere alla ricerca nel wiki da un altro programma:

```
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()
```

Qui l'azione semplicemente pubblica (via XML-RPC) le funzioni specificate nella lista. In questo caso `find_by`. `find_by` non è un'azione (perchè richiede un argomento). Interroga il database con `.select()` ed estrae i record con `.response` come una lista e la restituisce.

Ecco un esempio di come accede all'handler XML-RPC da un programma esterno in Python:

```
>>> 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
```

L'handler può essere acceduto dai linguaggi di programmazione in grado di utilizzare XML-RPC (per esempio C, C++, C# e Java).

3.10 *Altre funzionalità di* **admin**

L'interfaccia amministrativa mette a disposizione le seguenti altre funzionalità, qui illustrate brevemente.

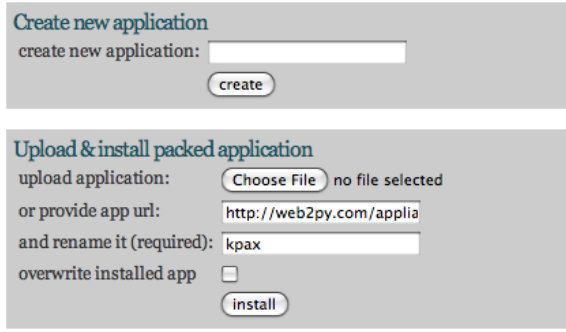
3.10.1 *site*

Questa pagina elenca tutte le applicazioni installate. Ci sono due form in basso, il primo consente di creare una nuova applicazione specificandone il nome il secondo consente di caricare un'applicazione esistente sia da un file locale che da una URL remota. Quando si carica un'applicazione deve essere specificato il nome con il quale la si vuole creare, che può anche non essere il suo nome originale. In questo modo è possibile installare più copie

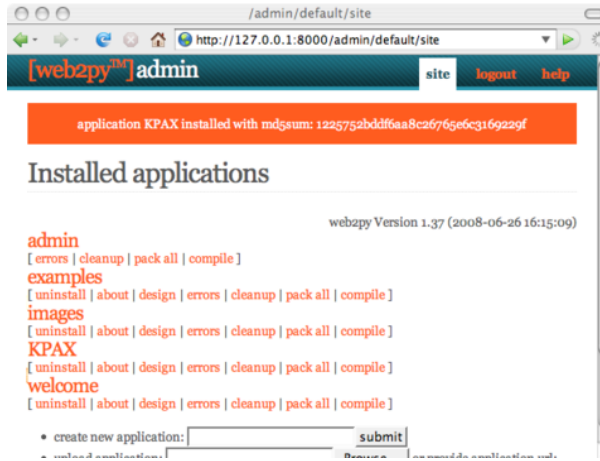
della stessa applicazione. Si può, per esempio, caricare il *Content Management System* KPAX da:

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

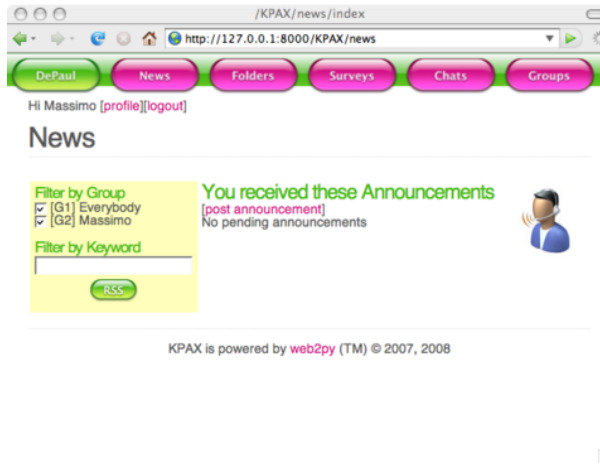
Le applicazioni caricate possono essere file di tipo `.tar` (utilizzati nelle precedenti versioni) e file di tipo `.w2p`. I file di tipo `.w2p` sono file tar compressi con gzip. Possono essere decompressi manualmente con il comando `tar zxvf [nomefile]` sebbene questo non sia necessario.



Dopo aver caricato un'applicazione web2py visualizza il checksum MD5 del file caricato che può essere utilizzato per verificare che il file non si sia corrotto nell'upload.



In **admin** cliccare su KPAX per esegure l'applicazione.



I file dell'applicazione sono memorizzati come file w2p (tar gzip), ma non è necessario comprimerli o decomprimerli manualmente perchè questo è fatto direttamente da web2py.

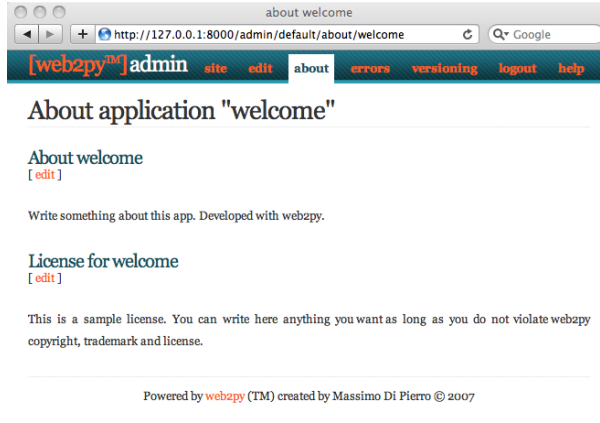
Per ogni applicazione la pagina *site* consente di:

- Disinstallare l'applicazione.
- Andare alla pagina *about* (illustrata più sotto)
- Andare alla pagina *edit* (illustrata più sotto)
- Andare alla pagina *errors* (illustrata più sotto)
- Azzerare i file temporanei (sessioni, errori e file di cache.disk)
- Comprimere l'applicazione. Questo restituisce un file tar contenente una copia completa dell'applicazione. E' bene cancellare i file temporanei prima di eseguire tale operazione.
- Compilare l'applicazione. Se non ci sono errori questa operazione compilerà in bytecode tutti i modelli, i controller e le viste. Poichè le viste possono includere altre viste, tutte le viste di un controller sono incluse in un unico file. Il risultato di questa operazione è che le applicazioni compilate in bytecode sono più veloci perchè non viene eseguita a runtime l'analisi dei template o la sostituzione delle stringhe.
- Comprimere l'applicazione compilata. Questa opzione è presente solamente nelle applicazioni compilate in bytecode. Consente di comprimere una applicazione senza il codice sorgente per distribuirla come *closed source*. Python (come qualsiasi altro linguaggio di programmazione) può tecnicamente essere decompilato; perciò la compilazione non fornisce una protezione completa del codice sorgente. La decompilazione comunque può essere complessa da eseguire ed essere illegale.
- Rimuovere un'applicazione compilata. Rimuove solamente i modelli, le viste ed i controller compilati in bytecode. Se l'applicazione è stata compressa con il codice sorgente, oppure è stata progettata internamente non c'è pericolo nel rimuovere i file compilati, l'applicazione continuerà a funzionare correttamente. Se l'applicazione è stata recuperata da un file compresso contenente un'applicazione compilata, allora non è buona regola rimuoverla, perchè non essendo disponibile il codice sorgente l'applicazione non funzionerà più.

Tutte le funzionalità disponibili dall'applicazione **admin** *di web2py sono anche accessibili da programma tramite le API definite nel modulo* `gluon/admin.py`. *Basta aprire una shell Python ed importare questo modulo.*

3.10.2 *about*

about consente di modificare la descrizione dell'applicazione e la sua licenza. Queste informazioni sono scritte nei file ABOUT e LICENSE nella cartella dell'applicazione.



E' possibile usare la sintassi `MARKMIN` o quella `gluon.contrib.markdown.WIKI` per questi file così come descritto in (29).

3.10.3 *edit*

La pagina *edit* è stata ampiamente usata in questo capitolo. Queste sono le altre funzialità accessibili da questa pagina:

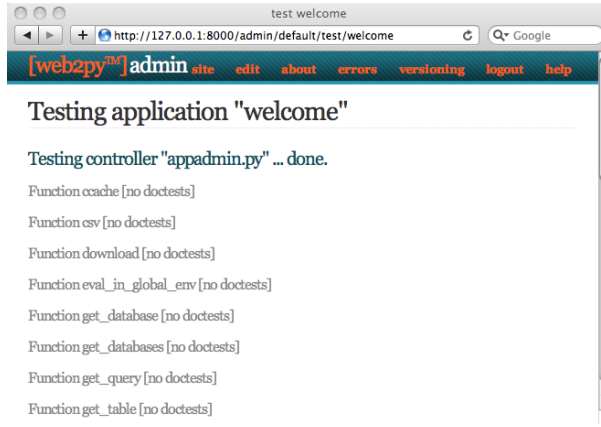
- Cliccando sul nome di un file se ne visualizza il contenuto con la sintassi Python evidenziata.
- Cliccando su *edit* si può modificare il file tramite un'interfaccia web.
- Cliccando su *delete* si può cancellare il file in modo permanente.

- Cliccando su *test* web2py eseguirà i test definiti dallo sviluppatore utilizzando il modulo *doctest* di Python. Ogni funzione dovrebbe avere i propri test.

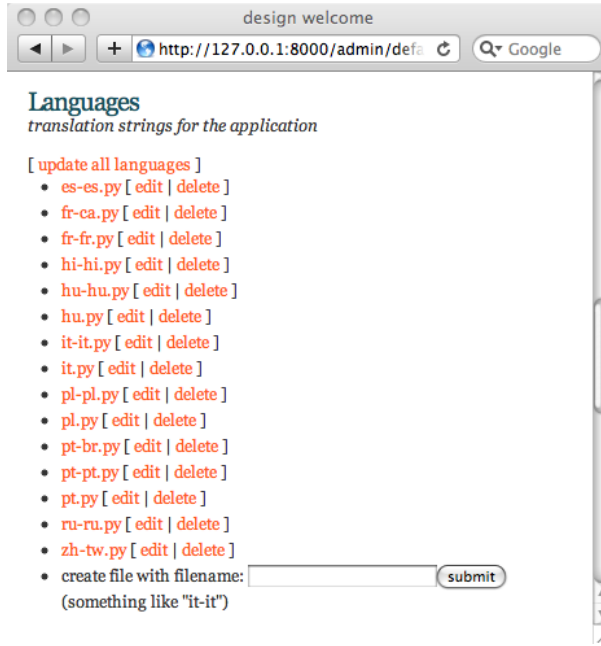
- E' possibile aggiungere nuovi file di linguaggio, scansionare l'applicazione per trovare nuove stringhe ed editare le traduzioni delle stringhe tramite l'interfaccia web.

- Se i file statici sono organizzati in una gerarchia di cartelle, queste possono essere aperte e chiuse cliccando sul loro nome.

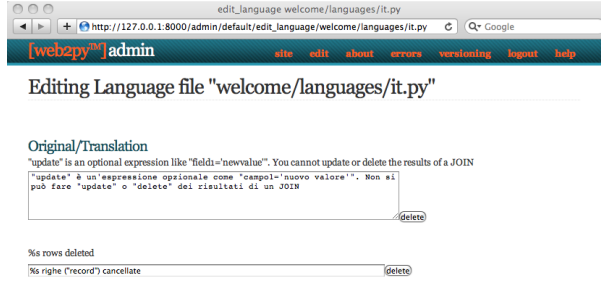
L'immagine seguente mostra l'output della pagina di test per l'applicazione *welcome*.



L'immagine segunte mostra la sezione dei linguaggi per l'applicazione *welcome*.



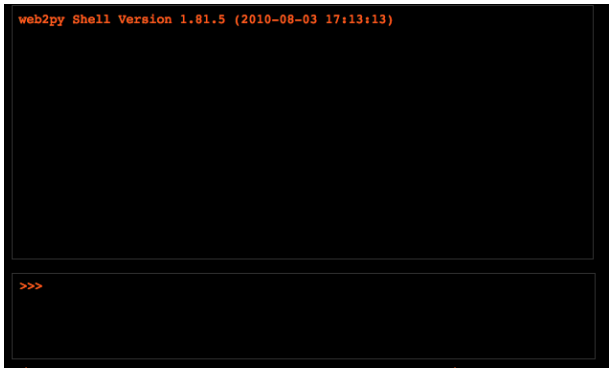
L'immagine seguente mostra come modificare un file di linguaggio, in questo caso quello relativo alla lingua "it" (Italiano) per l'applicazione *welcome*.



shell

Cliccando su "shell" nella sezione dei controller web2py aprira una pagina web con una shell Python dove eseguirà i modelli dell'applicazione. Questo

consente di interagire con l'applicazione.



crontab

Sempre sotto la sezione dei controller è presente il pulsante "crontab". Cliccando su di esso è possibile editare il file di crontab di web2py. Questo file segue la stessa sintassi del file di crontab di Unix, sebbene non utilizzi il *cron* di Unix. In effetti richiede solamente web2py e funziona anche in ambiente Windows. Questo consente di registrare delle azioni che saranno eseguite in background in momenti prestabiliti.

3.10.4 errors

Durante la programmazione di applicazioni in web2py è inevitabile fare errori ed introdurre bug. web2py aiuta lo sviluppatore in due modi: 1) consente di creare dei test per ogni funzione che può essere eseguita nel browser dalla pagina di *edit* e 2) quando si presenta un errore viene emesso un 'biglietto' (*ticket*) al visitatore e le informazioni relative all'errore sono memorizzate.

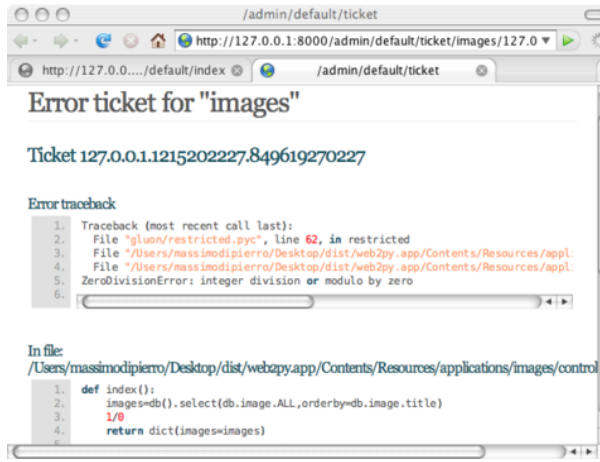
Ecco un errore introdotto di proposito nell'applicazione del blog di immagini:

```
1 def index():
```

```
    images = db().select(db.image.ALL,orderby=db.image.title)
    1/0
    return dict(images=images)
```

Quando si accede all'azione *index* si otterrà il seguente ticket:

Solo l'amministratore di web2py può accedere al ticket:



Il ticket mostra il *traceback* ed il contenuto del file che ha causato il problema. Se l'errore è all'interno di una vista web2py mostra la vista convertita dall'HTML in codice Python. Questo consente di identificare facilmente la struttura logica del file.

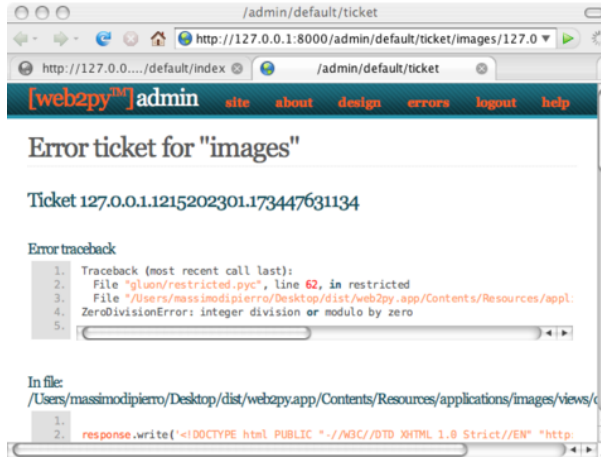
In **admin** il codice Python è sempre mostrato con la sintassi evidenziata dai colori (per esempio nei report d'errore le keyword di web2py sono mostrate

in arancione). Cliccando su una keyword di web2py si viene reindirizzati alla relativa pagina di documentazione.

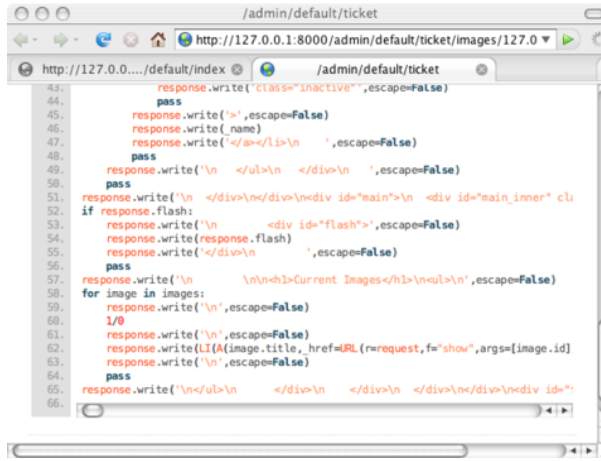
Ze si corregge il bug dell'azione *index* e si introduce un nuovo bug nella vista *index*:

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

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

si otterrà il seguente ticket:

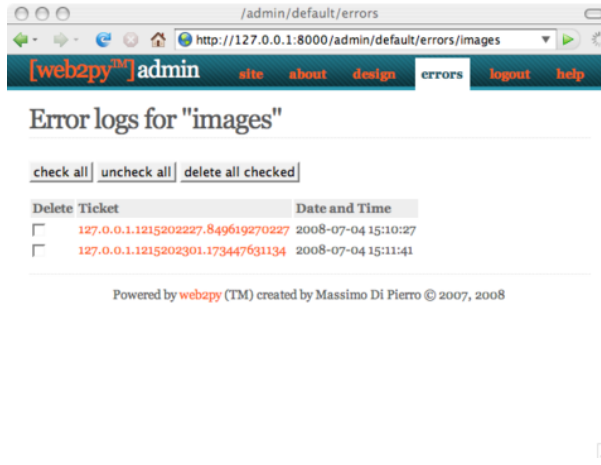


E' da notare che web2py ha convertito la vista dall'HTML in un file Python e per questo l'errore descritto nel ticket si riferisce al codice generato da Python e non alla vista originale.



Le prime volte questo può disorientare, ma in pratica rende il debug più facile perche l'indentazione di Python evidenzia la struttura logica del codice che è incorporato nelle viste.

Il codice è mostrato alla fine della stessa pagina. Tutti i ticket sono elencati in **admin** nella pagina *errors* di ogni applicazione:



3.10.5 *Mercurial*

Con le applicazioni in codice sorgente ed avendo installato le librerie di controllo delle versioni Mercurial:

```
easy_install mercurial
```

l'interfaccia amministrativa mostrerà un pulsante in più chiamato "mercurial". Questa opzione crea automaticamente un repository locale per l'applicazione utilizzando Mercurial. Premendo il pulsante "commit" l'applicazione attuale verrà salvata nel repository.

Questa funzione è sperimentale e sarà migliorata in futuro.

3.11 *Altre funzionalità di* **appadmin**

appadmin non è pensata per essere resa accessibile al pubblico. E' progettata per aiutare lo sviluppatore consentendo un facile accesso al database. Consiste di due soli file: un controller "appadmin.py" e una vista "appadmin.html" utilizzato da tutte le azioni del controller. Il controller *appadmin* è relativamente piccolo e leggibile e fornisce un esempio della progettazione di un'interfaccia per il database.

appadmin mostra quali database sono disponibili e quali tabelle esistono in ogni database. E' possibile aggiungere record ed elencare tutti i record per ogni tabella. **appadmin** presenta 100 record per pagina.

Quando un set di record è selezionato, l'intestazione delle pagine cambia, consentendo l'aggiornamento o la cancellazione dei record selezionati.

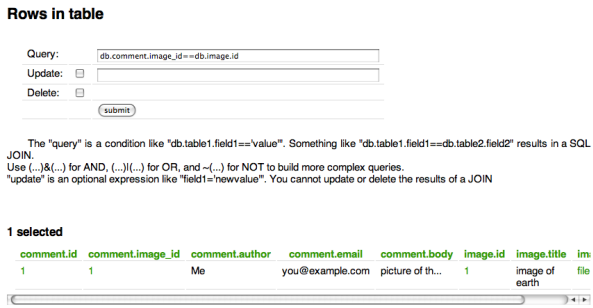
Per aggiornare i record inserire un comando SQL nel campo della stringa di Query:

```
title = 'test'
```

i valori delle stringhe devono essere racchiusi con apici singole (’). I campi possono essere separati dalla virgola. Per cancellare un record cliccare sulla sua casella di spunta e confermare l’operazione. Se il filtro SQL contiene una condizione che interessa più tabelle **appadmin** può anche eseguire delle unioni (*join*). Per esempio:

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

web2py passa questa condizione al DAL il quale comprende che la query collega due tabelle, e di conseguenza ambedue le tabelle sono selezionate con una INNER JOIN:



Cliccando il numero del campo ID si ottiene una pagina di modifica per il record con l’ID corrispondente. Se si clicca sul numero di un campo di riferimento si ottiente una pagina di modifica per il record referenziato. Non è possibile aggiornare o cancellare le righe selezionate in una JOIN perchè queste rappresentano record provenienti da tabelle multiple e l’operazione sarebbe ambigua.

4

Il nucleo di web2py

4.1 Opzioni della linea di comando

Per non visualizzare la finestra grafica d'avvio di web2py è possibile digitare dalla linea di comando:

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

Quando web2py si avvia crea un file chiamato "parameters_8000.py" dove memorizza la password codificata. Se si utilizza "<ask>" come password web2py la richiederà all'avvio.

Per maggior sicurezza è possibile avviare web2py con:

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

In questo caso web2py riutilizza la password precedentemente memorizzata. Se non è stata fornita nessuna password o se il file "parameters_8000.py" è stato cancellato l'interfaccia amministrativa sarà disabilitata.

SU alcuni sistemi Unix/Linux se la password è:

```
<pam_user:some_user>
```

web2py utilizzerà la password PAM dell'account `some_user` del sistema operativo come password dell'amministratore, a meno che questo sia bloccato dalla configurazione di PAM.

web2py utilizza normalmente CPython (l'implementazione in C dell'inteprete Python creato da Guido van Rossum) ma può essere eseguito anche con Jython (l'implementazione Java del medesimo interprete). Questa seconda opzione consente l'utilizzo di web2py nel contesto di un'infrastruttura J2EE. Per utilizzare Jython sostituire "python web2py.py..." con "jython web2py.py...". Ulteriori dettagli sull'installazione di Jython e sui moduli zxJDBC necessari per accedere ai database saranno forniti nel capitolo 12.

Lo script "web2py.py" può avere molte opzioni sulla linea di comando per specificare, per esempio, il numero massimo di thread, l'utilizzo di SSL, ecc. Per una lista completa digitare:

```
>>> 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)
--nogui               text-only, no GUI
-A ARGS, --args=ARGS  should be followed by a list of arguments to be passed
                      to script, to be used with -S, -A must be the last
                      option
--interfaces=INTERFACES
```

```
                        allows multiple interfaces to be served
```

Le opzioni in caratteri minuscoli sono utilizzate per configurare il server web. L'opzione `-L` indica a web2py di leggere le opzioni di configurazione da un file, `-W` installa web2py come un servizio Windows, mentre `-S`, `-P` e `-M` avviano una shell Python interattiva. L'opzione `-T` esegue i doctest dei controller nell'ambiente di esecuzione di web2py. Per esempio, il seguente comando esegue i doctest di tutti i controller nell'applicazione "welcome":

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

Se si esegue web2py come un servizio Windows (`-W`), non è conveniente passare gli argomenti di configurazione sulla linea di comando. Per questo motivo nella cartella di web2py c'è un file di configurazione "option_std.py" per il server web interno:

```
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()
```

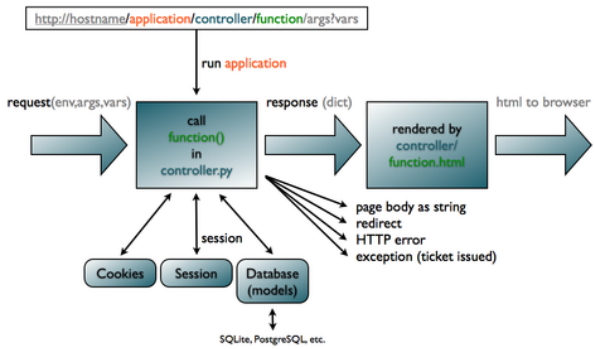
Questo file contiene le opzioni di default di web2py. Se si modifica questo file è necessario importarlo esplicitamente con l'opzione `-L` e funziona solamente se si esegue web2py come un servizio Windows.

4.2 Indirizzamento delle URL (dispatching)

web2py collega una URL nella seguente forma:

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

alla funzione `f()` nel controller "c.py" dell'applicazione "a". Se `f` non è presente, web2py utilizza come default la funzione `index` del controller. Se `c` non è presente, web2py utilizza come default il controller "default.py" e se `a` non è presente web2py utilizza come default l'applicazione `init`. Se questa non esiste web2py tenta di eseguire l'applicazione `welcome`. Questo è schematicamente indicato nella seguente immagine.



Di default ogni nuova richiesta crea una nuova sessione. In aggiunta un cookie di sessione è inviato al browser dell'utente per mantenere traccia della sessione.

L'estensione `.html` è opzionale; se non è indicata nessuna estensione `.html` è assunto come default. L'estensione utilizzata determina l'estensione della vista che produce l'output della funzione `f()` del controller. Questo rende possibile restituire lo stesso contenuto in diversi formati (html, xml, json, rss, etc.).

> *Le funzioni con argomenti o che iniziano con un doppio underscore (__) non sono esposte pubblicamente e possono essere chiamate solamente da altre fun-*

zioni.

Un'eccezione a questa regola di definizione delle URL è la forma:

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

Sebbene non ci sia nessun controller chiamato "static" web2py interpreta questa URL come la richiesta di un file chiamato "filename" nella sottocartella "static" dell'applicazione "a".

Quando i file statici sono scaricati web2py non crea una sessione nè invia un cookie o esegue il modello. web2py invia i file statici in blocchi di 1MB ed invia "PARTIAL CONTENT" quando il browser dell'utente manda una richiesta "RANGE" per una parte del file. web2py supporta inoltre il protocollo "IF_MODIFIED_SINCE" e non invia il file se questo è già memorizzato nella cache del browser ed il file non è cambiato rispetto a quella versione.

web2py collega le richieste GET/POST nella forma:

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

alla funzione `f` nel controller "c.py" dell'applicazione `a` e memorizza i parametri della URL nella variabile `request`:

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

e:

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

inoltre:

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

Nell'esempio precedente sia `request.args[i]` che `request.args(i)` possono essere utilizzati per recuperare l'elemento `i` di `request.args` ma mentre la prima

forma genera un'eccezione se la lista non contiene l'elemento la seconda ritorna `None`.

```
request.url
```

memorizza l'intera URL della richiesta corrente (escluse le variabili GET).

```
request.ajax
```

di default ritorna `False` ma se web2py determina che l'azione è stata eseguita da una richiesta Ajax ritorna `True`.

Se la richiesta HTTP è di tipo GET `request.env.request_method` ritorna "GET"; mentre se la richiesta è di tipo POST, `request.env.request_method` ritorna "POST". Le variabili della query dell'URL sono memorizzate nel dizionario (di tipo `Storage`) `request.vars`. Sono memorizzate anche in `request.get_vars` (per la richiesta di tipo GET) o in `request.post_vars` (per la richiesta di tipo POST). web2py memorizza le variabili d'ambiente WSGI e di web2py stesso in `request.env`, come per esempio:

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

come anche gli header HTTP:

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

web2py valida tutte le URL per prevenire attacchi di tipo directory traversal.

Le URL possono contenere esclusivamente caratteri alfanumerici, underscore (_) e slash (/); gli argomenti `args` possono conteneri più punti (.) non consecutivi. Gli spazi sono sostituiti dall'underscore prima della validazione. Se la sintassi della URL non è valida web2py ritorna un messaggio d'errore HTTP 400 (47; 48).

Se la URL corrisponde ad una richiesta per un file statico web2py lo legge e lo invia in streaming al browser dell'utente.

Se la URL non richiede un file statico web2py processa la richiesta nel seguente ordine:

- Legge i cookie.
- Crea un ambiente nel quale eseguire la funzione.
- Inizializza `request`, `response` e `cache`.
- Apre la sessione esistente (`session`) o ne crea una nuova.
- Esegue i modelli dell'applicazione richiesta.
- Esegue la funzione relativa all'azione richiesta del controller.
- Se la funzione ritorna un dizionario esegue la vista associata.
- Se tutto è stato eseguito con successo completa (*commit*) le transazioni aperte.
- Salva la sessione.
- Ritorna la risposta HTTP.

Il controller e la vista sono eseguiti in copie diverse dello stesso ambiente; pertanto la vista non vede il controller ma vede il modello e le variabili ritornate dalla funzione del controller relativa all'azione richiesta.

Se è generata una eccezione (diversa da `HTTP`) web2py si comporta nel seguente modo:

- Memorizza il *traceback* in un file d'errore e gli assegna un numero di ticket.
- Annulla (*rollback*) tutte le transazioni aperte.
- Ritorna una pagina d'errore che riporta il numero del ticket.

Se l'eccezione è di tipo `HTTP` questa è considerata come un comportamento voluto (per esempio una redirect HTTP) e tutte le transazioni aperte sono completate. Il comportamento successivo è specificato dalla stessa eccezione. La classe dell'eccezione `HTTP` non è un'eccezione standard di Python ma è definita da web2py.

4.3 *Librerie*

Le librerie di web2py sono esposte alle applicazioni come oggetti globali. Per esempio alcune variabili (`request`, `response`, `session` e `cache`), alcune classi (*helpers*, validatori, DAL) e alcune funzioni (`T` and `redirect`).

Questi oggetti sono definiti nei seguenti file di web2py:

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

L'applicazione compressa che è contenuta in web2py e che è utilizzata come base per le nuove applicazioni è:

```
welcome.w2p
```

E' creata durante l'installazione ed è sovrascritta durante gli aggiornamenti.

> *La prima volta che si avvia web2py vengono create due nuove cartelle:* `deposit` *e* `applications`*. L'applicazione "welcome" è compressa nel file "welcome.w2p". La cartella* `deposit` *è utilizzata come una cartella temporanea per installare e disinstallare le applicazioni.*

le *unit-test* di web2py sono in:

```
gluon/tests/
```

Ci sono moduli ausiliari per connettere diversi server web:

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

(fcgi.py è stato sviluppato da Allan Saddi)

Ci sono due file d'esempio:

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

Il primo è un file opzionale di configurazione che può essere passato a web2py con l'opzione `-L`. Il secondo è un file di esempio di *mapping* delle URL. Viene automaticamente caricato quando è rinominato in "routes.py".

I file:

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

sono file di configurazione necessari per il *deployment* sul *Google Application Engine*. Solitamente non è necessario modificarli; ulteriori informazioni sono disponibili nelle pagine di documentazione di Google sul GAE.

Sono presenti anche librerie aggiuntive, la maggior parte sviluppate da terze parti:

feedparser (28) di Mark Pilgrim per leggere i feed RSS ed Atom:

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

markdown2 (29) di Trent Mick per il markup nei Wiki:

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

markmin markup:

```
gluon/contrib/markmin.py
```

pysimplesoap è un'implementazione leggera di un server SOAP creata da Mariano Reingart:

```
gluon/contrib/pysimplesoap/
```

memcache (30) di Evan Martin è una API di Python per la gestione delle cache:

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

gql, è un *porting* del DAL per il *Google App Engine*:

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

memdb, è un *porting* del DAL per memcache:

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

gae_memcache è un'API per utilizzare memcache con il *Google App Engine*:

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

pyrtf (26) per generare documenti in formato RTF (*Rich Text Format*) sviluppato da Simon Cusack e revisionato da 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 (27) sviluppato da Dalke Scientific Software, per generare feed RSS:

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

simplejson (25) di Bob Ippolito, la libreria standard per la gestione degli oggetti JSON (Javascript Object Notation):

```
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
```

AuthorizeNet (93) fornisce un'API per accettare pagamenti con carta di credito tramite il network Authorize.net:

```
gluon/contrib/AuthorizeNet.py
```

PAM (72) è un'API di autenticazione creata da Chris AtLee:

```
gluon/contrib/pam.py
```

Un classificatore Bayesiano per popolare il database con dati fittizi per il test:

```
gluon/contrib/populate.py
```

Un file che consente l'interazione con la taskbar di Windows quando web2py è utilizzato come servizio:

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

Diversi **metodi di login** opzionali e form di login da utilizzare per l'autenticazione:

```
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/extended_login_form.py
gluon/contrib/login_methods/gae_google_account.py
gluon/contrib/login_methods/ldap_auth.py
gluon/contrib/login_methods/linkedin_account.py
gluon/contrib/login_methods/oauth20_account.py
gluon/contrib/login_methods/openid_auth.py
gluon/contrib/login_methods/pam_auth.py
gluon/contrib/login_methods/rpx_account.py
```

web2py contiene inoltre una cartella con script di utilità:

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

I primi due file sono particolarmente utili perchè tentano una installazione ed un setup completo di un ambiente di produzione di web2py. Sono discussi nel capitolo 12 ma sono abbastanza auto-documentati.

Infine web2py include i seguenti file necessari per costruire le distribuzioni binarie:

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

Questi sono script di setup rispettivamente per **py2exe** e **py2app** e sono richiesti solamente per costruire le distribuzioni binarie di web2py. NON DEVONO MAI ESSERE ESEGUITI!

Riepilogando, le librerie di web2py forniscono le seguenti funzionalità:

- Collegano le URL alle chiamate alle funzioni.
- Gestiscono il passaggio dei parametri via HTTP.
- Eseguono la validazione di questi parametri.
- Proteggono le applicazioni da molti problemi di sicurezza.
- Gestiscono la persistenza dei dati (database, sessioni, cache, cookie).
- Eseguono la traduzione delle stringhe nei diversi linguaggi supportati.
- Generano codice HTML (per esempio dalle tabelle di un database).

- Generano codice SQL tramite il DAL (Database Abstraction Layer).
- Generano l'output in formato RTF (Rich Text Format).
- Generano output in formato CSV (Comma Separated Value) dalle tabelle di un database.
- Generano feed RSS (Really Simple Syndication).
- Generano la serializzazione delle stringhe in JSON (Javascript Object Notation) per Ajax.
- Traducono il markup dei Wiki (Markdown) in HTML.
- Espongono servizi web XML-RPC.
- Caricano e scaricano file di grande dimensione tramite streaming.

Le applicazioni web2py contengono ulteriori file, in particolare librerie Javascript di terze parti, come jQuery, calendar, EditArea e nicEdit. Gli autori di queste librerie sono indicati all'interno delle librerie stesse.

4.4 Applicazioni

Le applicazioni sviluppate in web2py sono composte delle seguenti parti:

- I **modelli** descrivono una rappresentazione dei dati (come le tabelle dei database e le relazioni tra di esse).
- I **controller** descrivono la logica dell'applicazione ed il suo flusso di lavoro.
- Le **viste** descrivono come i dati dovrebbero essere presentati all'utente utilizzando HTML e Javascript.
- I **linguaggi** descrivono come tradurre le stringhe dell'applicazione in diverse lingue.
- I **file statici** non richiedeono nessuna elaborazione (per esempio immagini, CSS, ecc.).
- **ABOUT** e **README** contengono informazioni sull'applicazione.

- Gli **errori** memorizzano informazioni sugli errori generati dall'applicazione.
- Le **sessioni** memorizzano informazioni legate ad ogni specifico utente.
- I **database** memorizzano i database di SQLite ed informazioni aggiuntive sulle tabelle
- la **cache** memorizza oggetti utilizzati dall'applicazione
- I **moduli** sono altri moduli di Python aggiuntivi.
- I file **privati** sono acceduti dai controller ma non direttamente dallo sviluppatore.
- I file di **upload** sono acceduti dai modelli ma non direttamente dallo sviluppatore (per esempio i file caricati dagli utenti di un'applicazione).
- I file di **test** sono memorizzati in una cartella per memorizzare gli script di test e le *fixture*.

I modelli, le viste, i controller, i linguaggi e i file statici sono accessibili dall'interfaccia web di amministrazione. I file di ABOUT, README e gli errori sono anch'essi accessibili dall'interfaccia web di amministrazione tramite la corrispondente sezione. Le sessioni, la cache, i moduli ed i file privati sono accessibili all'applicazione ma non all'iterfaccia web di amministrazione.

Tutti questi file sono organizzati in una precisa struttura di cartelle, replicata per ogni applicazione web2py, anche se l'utente non ha mai bisogno di accedere direttamente al filesystem:

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

"__init__.py" è un file vuoto che è necessario per consentire a Python (e a web2py) di importare i moduli presenti nella cartella `modules`.

L'applicazione **admin** fornisce solamente una interfaccia web alle applicazioni web2py presenti sul filesystem del server. Le applicazioni web2py possono anche essere create e sviluppate dalla linea di comando, non è indispensabile utilizzare l'interfaccia web dell'applicazione **admin**. Una nuova applicazione

può essere creata manualmente replicando la struttura di cartelle in una nuova cartella all'interno della cartella `applications` (o più semplicemente scompattando il file `welcome.w2p` nella nuova cartella). I file che compongono l'applicazione possono essere creati e modificati dalla linea di comando senza dover usare l'interfaccia web dell'applicazione **admin**.

4.5 API

I modelli, i controller e le viste sono eseguiti in un ambiente dove i seguenti oggetti sono automaticamente importati:

Oggetti globali:

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

Navigazione:

```
redirect, HTTP
```

Internationalizzazione:

```
T
```

Helpers:

```
XML, URL, BEAUTIFY

A, B, BEAUTIFY, BODY, BR, CENTER, CODE, DIV, EM, EMBED,
FIELDSET, FORM, H1, H2, H3, H4, H5, H6, HEAD, HR, HTML,
I, IFRAME, IMG, INPUT, LABEL, LEGEND, LI, LINK, OL, UL,
MARKMIN, MENU, META, OBJECT, ON, OPTION, P, PRE, SCRIPT,
OPTGROUP, SELECT, SPAN, STYLE, TABLE, TAG, TD, TEXTAREA,
TH, THEAD, TBODY, TFOOT, TITLE, TR, TT, URL, XHTML,
xmlescape, embed64
```

Validatori

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

Database:

```
DAL, Field
```

Per compatibilità con le versioni precedenti di web2py sono presenti anche `SQLDB=DAL` e `SQLField=Field`. E' bene però utilizzare la nuova sintassi `DAL` e `Field` invece della vecchia.

Altri moduli ed oggetti sono definiti nelle librerie ma non sono automaticamente importati perchè non usati molto spesso.

Le entità fondamentali nell'ambiente d'esecuzione di web2py sono `request`, `response`, `session`, `cache`, `URL`, `HTTP`, `redirect` e `T` e sono illustrate di seguito.

Alcuni oggetti e funzioni (come **Auth**, **Crud** e **Service**) sono definiti in "gluon/tools.py" e devono essere importati se richiesti:

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

4.6 request

L'oggetto `request` è un'istanza della onnipresente classe di web2py chiamata `gluon.storage.Storage`, che estende la classe `dict` standard di Python. E' fondamentalmente un dizionario ma i suoi valori possono essere acceduti anche come attributi:

```
request.vars
```

è lo stesso di:

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

A differenza di un dizionario se un attributo (o una chiave) non esiste non viene generata un'eccezione ma viene ritornato `None`.

`request` ha le seguenti chiavi (attributi), alcune delle quali sono esse stesse un istanza della classe `Storage`:

- **request.cookies**: un oggetto di tipo `Cookie.SimpleCookie()` che contiene i cookie passati nella richiesta HTTP. Agisce come un dizionario di cookie dove ogni cookie è un oggetto Morsel di Python.
- **request.env**: un oggetto di tipo `Storage` che contiene le variabli d'ambiente passate al controller cioè le variabili presenti nell'header HTTP dalla richiesta HTTP ed i parametri standard WSGI. Le variabili d'ambiente sono tutte convertite in minuscolo e i punti sono convertiti in underscore per una memorizzazione più facile.
- **request.application**: il nome dell'applicazione richiesta (estratta da `request.env.path_`
- **request.controller**: il nome del controller richiesto (estratto da `request.env.path_info`)
- **request.function**: il nome della funzione richiesta (estratto da `request.env.path_info`).
- **request.extension**: l'estensione dell'azione richiesta. Ha come default "html". Se la funzione del controller ritorna un dizionario e non specifica una vista l'estensione è usata per determinare quale file di vista deve essere usato per visualizzare il dizionario (estratto da `request.env.path_info`).
- **request.folder**: la cartella dell'applicazione. Per esempio per l'applicazione "welcome" `request.folder` è impostato al percorso assoluto "/path/to/welcome". Nei programmi deve sempre essere usata questa variabile e la funzione `os.path.join` per costruire il path ai file. Sebbene web2py utilizzi sempre path assoluti è buona regola non cambiare mai la cartella di lavoro corrente poichè questa non è una pratica sicura per i thread.
- **request.now**: un oggetto di tipo `datetime.datetime` che contiene l'orario della richiesta corrente.

- **request.args**: una lista delle componenti delle URL che seguono il nome della funzione; equivalente a `request.env.path_info.split('/')[3:]`
- **request.vars**: un oggetto di tipo `gluon.storage.Storage` che contiene le variabili della query HTTP GET o HTTP POST.
- **request.get_vars**: un oggetto di tipo `gluon.storage.Storage` contenente solo le variabili della query HTTP GET.
- **request.post_vars**: un oggetto di tipo `gluon.storage.Storage` contenente solo le variabili della query HTTP POST.
- **request.client**: L'indirizzo IP del client come determinato da `request.env.remote_addr` o da `request.env.http_x_forwarded_for` se presente. Sebbene questo può essere utile non dovrebbe essere considerato come un dato affidabile perchè `http_x_forwarded_for` può essere falsificato.
- **request.body**: uno stream file in sola lettura che contiene il corpo della richiesta HTTP. Questo stream è analizzato per recuperare `request.post_vars` ed è poi reinizializzato. Può essere letto con `request.body.read()`.
- **request.wsgi** un *hook* che consente di chiamare applicazioni WSGI di terze parti dall'interno delle azioni.

Per esempio, la seguenta chiamata su un tipico sistema web2py:

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

risulta nel seguente dizionario:

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.wsgi	hook
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 1.81.5
request.env.web2py_runtime_gae	(opzionale, definita solo se è utilizzato Google App Engine)
request.env.wsgi_errors	<open file, 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

Quali variabili d'ambiente siano effettivamente definite dipende dal server web. Qui è utilizzato il server WSGI Rocket, presente all'interno di web2py.

Il set di variabili non è molto diverso quando si usa il server web Apache.

Le variabili `request.env.http_*` sono estratte dall'header della richiesta HTTP.

Le variabili `request.env.web2py_*` non sono estratte dall'ambiente del server web ma sono create da web2py. Sono utili se l'applicazione dovesse aver bisogno di informazioni relative alla posizione o alla versione di web2py o conoscere se l'applicazione sta girando su Google App Engine (perchè potrebbe essere necessaria una specifica ottimizzazione). Anche le variabili `request.env.wsgi_*` sono specifiche dell'adattatore WSGI.

4.7 response

`response` è un'altra istanza della classe `Storage`. Contiene le seguenti variabili:

- **response.body**: un oggetto di tipo `StringIO` nel quale web2py scrive l'output del corpo della pagina. NON CAMBIARE MAI QUESTA VARIABILE.
- **response.cookies**: simile a **request.cookies** ma mentre quest'ultima contiene i cookie inviati dal client al server **response.cookies** contiene i cookie inviati dal server al client. Il cookie di sessione è gestito automaticamente.
- **response.download(request, db)**: un metodo utilizzato per implementare la funzione del controller che consente lo scarico dei file caricati dagli utenti.
- **response.files**: una lista dei file.css e.js necessari per la pagina di risposta. I loro link saranno automaticamente inclusi nell'header di "layout.html" standard. Per includere un nuovo file.css o.js è sufficiente aggiungerlo a questa lista. I duplicati sono gestiti dalla lista stessa e l'ordine d'inserimento è significativo.
- **response.flash**: un parametro opzionale che può essere incluso nelle viste. Normalmente utilizzato per notificare informazioni all'utente.
- **response.headers**: un `dict` per gli header HTTP della risposta.

- **response.menu**: un parametro opzionale che può essere incluso nelle viste, solitamente usato per passare un menu di navigazione alla vista. Può essere visualizzato tramite l'helper MENU.
- **response.meta**: un oggetto di tipo `Storage` che contiene informazioni *meta* opzionali come `response.meta.author`, `response.meta.description` e `response.meta.keyw`
  Il contenuto della variabile `response.meta` è automaticamente inserito nel tag `META` dal codice in "web2py_ajax.html" che è incluso nella vista di default "views/layout.html".
- **response.postprocessing**: è una lista di funzioni, vuota di default. Queste funzioni sono utilizzate per filtrare l'oggetto response dopo l'output dell'azione e prima che venga visualizzato dalla vista. Può essere usato per implementare il supporto per altri linguaggi di template.
- **response.render(view, vars)**: un metodo utilizzato per chiamare la vista esplicitamente dall'interno del controller. `view` è un parametro opzionale con il nome del file della vista e `vars` è un dizionario di chiavi/valori passato alla vista.
- **response.session_file**: è uno stream del file che contiene la sessione.
- **response.session_file_name**: è il nome del file dove sarà memorizzata la ssssione.
- **response.session_id**: l'id della sessione corrente, generato automaticamente. NON CAMBIARE MAI QUESTA VARIABILE.
- **response.session_id_name**: il nome del cookie di sessione per l'applicazione. NON CAMBIARE MAI QUESTA VARIABILE.
- **response.status**: Il valore del codice di stato HTTP che deve essere passato alla response. Il default è 200 (OK).
- **response.stream(file, chunk_size)**: quando un controller ritorna questo valore web2py invia il contenuto al client in blocchi delle dimensioni di `chunk_size`.
- **response.subtitle**: parametro opzionale che può essere aggiunto alla vista. Dovrebbe contenere il sottotitolo della pagina.
- **response.title**: parametro opzionale che può essere aggiunto alla vista. Dovrebbe contenere il titolo della pagina e dovrebbe essere visualizzato dal tag TITLE nell'header della pagina.

- **response._vars**: questa variabile è accessibile solo in una vista, non in un'azione. Contiene i valori ritornati dall'azione alla vista.
- **response.view**: il nome della vista che deve visualizzare la pagina. E' impostato per default a:

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

o, se il file precedente non esiste, è impostato a:

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

Cambiare il valore di questa variabile per modificare il file della vista associato con una azione particolare.

- **response.xmlrpc(request, methods)**: quando un controller ritorna questo valore, questa funzione espone i metodi via XML-RPC (46). Questa funzione è deprecata in quanto esiste un meccanismo migliore illustrato nel capitolo 9.
- **response.write(text)**: un metodo per scrivere all'interno del corpo della pagina.

Poichè *response** è un oggetto di tipo `gluon.storage.Storage` può essere utilizzato per memorizzare altri attributi che si vuol passare alla vista. Sebbene non ci siano limitazioni tecniche è bene memorizzare solamente variabili che devono essere visualizzate in tutte le pagine del layout globale "layout.html".

In ogni modo le seguenti variabili dovrebbero essere utilizzate:

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

perchè questo rende più facile sostituire il template standard "layout.html" incluso in web2py con un altro file di layout che utilizzi le stesse variabili.

Le vecchie versioni di web2py usano `response.author` invece di `response.meta.author`, lo stesso vale per gli altri attributi *meta*.

4.8 session

`session` è un'altra istanza della classe `Storage`. Tutto quello che è memorizzato in `session` per esempio:

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

può essere successivamente recuperato:

```
a = session.myvariable
```

purchè il codice sia eseguito nella stessa sessione dello stesso utente (e che l'utente non abbia cancellato i cookie di sessione e che la sessione stessa non sia scaduta). Poichè `session` è un oggetto `Storage` se si tenta di accedere ad un attributo/chiave che non esiste non si genera nessuna eccezione ma si ottiene `None`.

L'oggetto `session` ha due metodi importanti, il primo è **forget**:

```
session.forget()
```

ed indica a web2py di non memorizzare la sessione. Questo dovrebbe essere utilizzato in quei controller le cui azioni sono chiamate spesso e non necessitano di tracciare l'attività dell'utente.

L'altro metodo è **connect**:

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

dove `db` è il nome di una connessione di database aperta (come ritornata dal DAL). Indica a web2py che si vuole memorizzare le sessioni nel database e non nel filesystem. web2py crea una tabella:

```
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'))
```

è memorizza con *cPickle* le sessioni nel campo `session_data`.

L'opzione `masterapp=None` indica a web2py di recuperare una sessione esistente per l'applicazione con il nome indicato (come memorizzato in `request.applicatio` all'interno della applicazione corrente.

Se si vuole che due o più applicazioni condividano le sessioni basta impostare `masterapp` al nome dell'applicazione principale.

E' possibile controllare lo stato dell'applicazione in ogni momento visualizzando le variabili di sistema `request`, `session` e `response`. Un modo di fare questo è definire un'azione dedicata:

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

4.9 *cache*

`cache` è un oggetto globale definito nell'ambiente d'esecuzione di web2py. Ha due attributi:

- **cache.ram**: la cache dell'applicazione è memorizzata nella memoria principale.
- **cache.disk**: la cache dell'applicazione è memorizzata nel filesystem.

`cache` è un oggetto richiamabile (*callable*), questo consente di utilizzarlo come decoratore per le azioni e le viste.

Il seguente esempio utilizza la ram per la cache della funzione `time.ctime()`:

```
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))
```

L'output della funzione `lambda: time.ctime()` è memorizzato nella ram per 5 secondi. La stringa `'time'` è utilizzata come chiave:

Il seguente esempio utilizza il filesystem per la stessa operazione:

```
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))
```

L'output di `lambda: time.ctime()` è memorizzato sul disco (utilizzando il modulo *shelve*) per 5 secondi.

Il seguente esempio utilizza sia la ram che il filesystem:

```
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))
```

L'output di `lambda: time.ctime()` è memorizzato su disco (utilizzando il modulo shelve) e poi in ram per 5 secondi. web2py cerca prima nella ram e se non trova nulla cerca su disco. Se non trova nulla neanche sul disco web2py esegue la funzione `lambda: time.ctime()` e la cache viene aggiornata. Questa tecnica è utile in un ambiente multi-processo. I due orari non devono essere per forza gli stessi.

Il seguente esempio memorizza in ram l'output della funzione del controller (ma non della vista):

```
1
2 @cache(request.env.path_info, time_expire=5, cache_model=cache.ram)
3 def cache_controller_in_ram():
4     import time
5     t = time.ctime()
6     return dict(time=t, link=A('click me', _href=request.url))
```

Il dizionario restituito da `cache_controller_in_ram` è memorizzato in ram per 5 secondi. Il risultato di una SELECT su un database non può essere memorizzato nella cache se non viene prima serializzato. Un modo migliore di effettuare la cache di un database è di eseguire il metodo `select` con l'attributo `cache`

Il seguente esempio memorizza l'output di una funzione di un controller (ma non della vista) su disco:

```
1 @cache(request.env.path_info, time_expire=5, cache_model=cache.disk)
2 def cache_controller_on_disk():
3     import time
4     t = time.ctime()
5     return dict(time=t, link=A('click to reload',
6                                _href=request.url))
```

Il dizionario restituito da `cache_controller_on_disk` è memorizzato su disco per 5 secondi. web2py non può memorizzare nella cache oggetti che non sono serializzabili tramite *cPickle*.

E' anche possibile memorizzare una vista nella cache. Il trucco è nel preparare l'output della vista in una funzione del controller in modo che sia ritornata come una stringa. Per fare questo è necessario ritornare la funzione `response.render(d)` dove `d` è il dizionario che si vuole passare alla vista:

Il seguente esempio memorizza nella ram l'output della funzione (incluso l'output della vista):

```
1 @cache(request.env.path_info, time_expire=5, cache_model=cache.ram)
2 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)` restituisce l'output della vista come una stringa che è memorizzata in ram per 5 secondi. Questo è il modo migliore e più veloce di utilizzare la cache.

E' anche possibile definire altri meccanismi di cache come, per esempio, `memcache`. `memcache` è disponibile nel modulo `gluon.contrib.memcache` ed è discusso con maggior dettaglio nel capitolo 11.

4.10 URL

La funzione `URL` è una delle più importanti in web2py. Genera i percorsi delle URL interni all'applicazione per le azioni e i file statici.

Ecco un esempio:

```
URL('F')
```

è trasformato in:

```
/[application]/[controller]/F
```

L'output della funzione `URL` dipende dall'applicazione corrente, dal controller in uso e da altri parametri. web2py supporta sia il collegamento diretto delle URL (*mapping*) che il collegamento inverso (*reverse mapping*). Il mapping delle URL consente di ridefinire il formato delle URL esterne. Se si utilizza la funzione `URL` per generare tutte le URL dell'applicazione l'aggiunta o la modifica delle URL eviterà la presenza di link errati all'interno dell'applicazione.

E' possibile passare parametri aggiuntivi alla funzione URL, per esempio parti aggiuntive nel path (argomenti, `args`) e variabili di query (`vars`):

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

è trasformata in:

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

Gli attributi presenti in `args` sono automaticamente analizzati, decodificati e memorizzati in `request.args` da web2py. Allo stesso modo le variabili nel dizionario `vars` sono analizzate, decodificate e memorizzate in `request.vars`. `args` e `vars` sono il meccanismo di base con il quale web2py scambia informazioni con il browser dell'utente. Se `args` contiene un solo elemento non è necessario passarlo all'interno di una lista.

E' anche possibile utilizzare la funzione `URL` per generare URL verso azioni presenti in altri controller ed in altre applicazioni:

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

è trasformato in:

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

E' anche possibile specificare l'applicazione il controller e la funzione utilizzanto argomenti con nome:

```
URL(a='a', c='c', f='f')
```

Se l'applicazione non è indicata è utilizzato quella corrente.

```
URL('c', 'f')
```

Se il controller non è indicato è utilizzato quello corrente.

```
URL('f')
```

Invece di passare il nome di una funzione di un controller è possibile passare direttamente la funzione:

```
URL(f)
```

Per le ragioni sopra indicate si dovrebbe sempre utilizzare la funzione `URL` per generare le URL dei file statici dell'applicazione. I file statici sono memorizzati nella cartella `static` dell'applicazione (ed è la cartella dove vengono caricati dall'interfaccia amministrativa). web2py mette a disposizione un controller virtuale chiamato "static" il cui compito è recuperare i file dalla cartella `static`, determinare tipo del loro contenuto (*content-type*) ed inviare i file all'utente. Il seguente esempio genera l'URL per il file statico "image.png":

```
URL('static', 'image.png')
```

che viene trasformato in:

```
/[application]/static/image.png
```

Non è necessario codificare o validare (*escape*) gli argomenti di `args` e `vars` in quanto questa operazione è eseguita direttamente ed automaticamente da web2py.

4.11 HTTP e la redirezione

web2py definisce solo una nuova eccezione, chiamata `HTTP`. Questa eccezione può essere generata in un qualsiasi punto di un modello, un controller o una vista con il comando:

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

e fa si che il flusso dell'elaborazione si interrompa e torni immediatamente a web2py che ritorna una risposta HTTP del tipo:

```
HTTP/1.1 400 BAD REQUEST
Date: Sat, 05 Jul 2008 19:36:22 GMT
Server: Rocket WSGI Server
Content-Type: text/html
Via: 1.1 127.0.0.1:8000
```

```
Connection: close
Transfer-Encoding: chunked

my message
```

Il primo argomento di `HTTP` è il codice di stato HTTP. Il secondo argomento è la stringa che verrà ritornata come corpo della risposta. Argomenti con nome (aggiuntivi ed opzionali) sono utilizzati per costruire l'header della risposta HTTP. Per esempio:

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

genera:

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

my message
```

Se non si vuole che le transazioni del database siano finalizzate (*commit*) è necessario annullare le transazioni (*rollback*) prima di generare l'eccezione.

Qualsiasi altra eccezione diversa da `HTTP` fa sì che web2py annulli tutte le transazioni di database aperte, registri il traceback dell'errore, emetta un ticket per l'utente e ritorni una pagina standard d'errore. Questo significa che solamente `HTTP` può essere utilizzata per cambiare il flusso di controllo tra le diverse pagine. Altre eccezioni devono essere gestite dall'applicazione, altrimenti genereranno un ticket di web2py.

Il comando:

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

E' semplicemente una scorciatoia per:

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

Gli argomenti con nome del metodo `HTTP` sono tradotti in direttive dell'header HTTP, in questo caso la destinazione della redirezione. `redirect` ha un secondo parametro opzionale che è il codice di stato HTTP della redirezione (303 di default). Si può cambiare questo codice a 307 per una redirezione temporanea o a 301 per una redirezione permanente.

Il modo più comune di usare le redirezioni è quello di reindirezzare ad altre pagine della stessa applicazione e (opzionalmente) passare dei parametri:

```
redirect(URL('index',args=(1,2,3),vars=dict(a='b')))
```

4.12 *T e l'internazionalizzazione*

L'oggetto `T` è il traduttore di linguaggio. Costituisce una singola istanza globale della classe `gluon.language.translator` di web2py. Tutte le stringe costanti (e solamente quelle costanti) dovrebbero essere utilizzate con `T`, per esempio:

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

web2py identifica tutte le stringhe utilizzate con `T` come stringhe che necessitano della traduzione in un altro linguaggio e quindi verranno tradotte quando il codice (nel modello, nel controller o nella vista) sarà eseguito. Se la stringa da tradurre non è una costante ma è una variabile sarà aggiunta al file di traduzione durante l'esecuzione (tranne che su GAE) per essere tradotta successivamente.

L'oggetto `T` può anche contenere variabili interpolate, per esempio:

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

La prima parte della stringa è tradotta secondo il file di linguaggio richiesto mentre il valore della variabile `name` è utilizzato indipendentemente dal linguaggio.

Il concatenamento di più stringhe da tradurre con `T` non è una buona idea, per questo è impedito da web2py:

```
T("blah ") + name + T(" blah")   # non valido!
```

mentre è consentito:

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

o anche la sintassi alternativa:

```
T("blah %(name)s blah") % dict(name='Tim')
```

In ambedue i casi la traduzione viene effettuata prima che la variabile `name` sia sostituita nella posizione "%(name)s" La sintassi seguente invece NON DEVE ESSERE USATA:

```
T("blah %(name)s blah" % dict(name='Tim'))
```

perchè la traduzione avverrebbe dopo la sostituzione.

Il linguaggio richiesto è determinato dal campo "Accept-Language" dell'header HTTP ma può essere forzato da programma richiedendo uno specifico file di traduzione, per esempio:

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

che obbliga web2py a leggere il file di linguaggio "languages/it-it.py". I file di linguaggio possono essere creati e modificati dall'interfaccia amministrativa.

Normalmente la traduzione delle stringhe è eseguita alla fine, quando l'output della vista viene generato; per questo il metodo `force` del traduttore non dovrebbe essere chiamato all'interno di una vista.

E' possibile disabilitare questo comportamento "pigro" (*lazy*) della valutazione delle stringhe da tradurre con:

```
T.lazy = False
```

In questo modo le stringhe sono tradotte immediatamente dall'operatore `T` in base al linguaggio attualmente accettato o forzato.

Un problema tipico è il seguente: l'applicazione originale contiene le stringhe in Inglese, esiste un file di linguaggio in italiano (languages/it-it.py) e il client HTTP dichiara di accettare Inglese (en) ed Italiano (it-it) in quest'ordine. In questo caso si ha il seguente comportamento (non voluto): web2py non sa che l'applicazione contiene le stringhe in Inglese, perciò preferisce le stringhe in Italiano (it-it) perchè il file di traduzione in Inglese (en) non esiste. Se non esistesse nemmeno il file languages/it-it.py web2py avrebbe utilizzato le stringhe in Inglese presenti nell'applicazione.

Ci sono due soluzioni per questo problema: creare un file di traduzione in Inglese, che sarebbe ridondante e non necessario, oppure (ed è la soluzione consigliata) indicare a web2py in quale linguaggio sono le stringhe contenute nel codice dell'applicazione. Questo può essere fatto con:

```
T.set_current_language('en', 'en-en')
```

Questo comando memorizza in `T.current_languages` una lista dei linguaggi che non necessitano di traduzione e forza una rilettura dei file di linguaggio.

E' da notare che "it" e "it-it" sono due linguaggi diversi dal punto di vista di web2py. Per supportare tutti e due è necessario avere due file di traduzione (uno per "it" e uno per "it-it"), sempre in minuscolo. Lo stesso è valido per tutti gli altri linguaggi.

Il linguaggio attualmente accettato è memorizzato in:

```
T.accepted_language
```

E' da ricordare che T(...) non serve a tradurre solamente le stringhe ma anche le variabili:

```
>>> a="test"
>>> print T(a)
```

In questo caso la parola "test" è tradotta ma, se non viene trovata nel file di linguaggio e il filesystem è scrivibile verrà aggiunta alla lista delle parole da tradurre nel file di linguaggio.

4.13 Cookie

web2py utilizza i moduli standard di Python per la gestione dei cookie.

I cookie ricevuti dal browser sono presenti in `request.cookies` e i cookie inviati dal server sono in `response.cookies`.

Un cookie può essere impostato nel seguente modo:

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

La seconda linea indica al browser di mantenere il cookie per 24 ore. La terza linea indica al browser di reinviare il cookie ad ogni applicazione del dominio corrente.

I cookie possono essere resi sicuri con:

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

Un cookie sicuro è rinviato solamente su una connessione HTTPS e non su una connessione HTTP.

Il cookie può essere recuperato con:

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

A meno che le sessioni siano disabilitate web2py, automaticamente, imposta il seguente cookie e lo utilizza per gestire le sessioni:

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

4.14 *L'applicazione* **init**

Quando si installa web2py, si potrebbe volere un'applicazione di default, cioè l'applicazione che viene eseguita quando il path della URL è vuoto, come in:

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

Per default, quando riceve un path vuoto web2py cerca un'applicazione chiamata **init**. Se l'applicazione init non è presente cerca un applicazione chiamata **welcome**.

Ecco tre modi per impostare l'applicazione di default:

- Chiamare l'applicazione che si vuole avere come default "init".
- Creare un link simbolico da "applications/init" alla cartella dell'applicazione richiesta.
- Usare la riscrittura delle URL come indicato nella prossima sezione.

4.15 *Riscrittura delle URL*

web2py ha la capacità di riscrivere il path delle URL delle richieste entranti prima di chiamare l'azione di un controller (*mapping*) e allo stesso modo

web2py può riscrivere il path delle URL generate dalla funzione `URL` (*reverse mapping*). Un motivo per voler fare questo è per la gestione di URL pre-esistenti o per semplificare i path e renderli più brevi.

Per usare questa funzionalità si deve creare un nuovo file nella cartella "web2py" chiamato "routes.py" e definire due liste (o tuple) `routes_in` e `routes_out` contenenti delle tuple. Ogni tupla contiene due elementi, il pattern che deve essere sostituito e la stringa che lo sostituisce. Per esempio:

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

Con questi instradamenti la URL:

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

è trasformata nella URL:

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

Per l'utente tutti i link alla pagina index sono `/testme`.

I pattern possono avere la stessa sintassi delle espressioni regolari di Python. Per esempio:

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

indirizza tutte le URL che finiscono con ".php" alla pagina `/init/default/index`.

Nel caso che esista solo una applicazione potrebbe essere utile eliminare il nome dell'applicazione dalla URL. Questo può essere fatto con:

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

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

Per gli instradamenti esiste una sintassi alternativa che può essere utilizzata insieme alle espressioni regolari: consiste nell'utilizzare `name` invece di `(?P<name>[\w_]+)` o `\g<name>`. Per esempio:

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

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

elimina il nome dell'applicazione in tutte le URL.

Usando la notazione con il carattere `$` si può eseguire la mappatura automatica da `routes_in` a `routes_out` purchè non si utilizzino espressioni regolari. Per esempio:

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

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

In caso di instradamenti multipli viene eseguito il primo per il quale la URL soddisfa l'espressione regolare. Se nessun instradamento viene trovato il path rimane inalterato.

E' possibile utilizzare `$anything` per indicare qualsiasi carattere fino alla fine della linea.

Ecco un file "routes.py" minimale per gestire correttamente le richieste per `favicon.ico` e per `robot.txt`:

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

```
)
routes_out = ()
```

Questo è un esempio più complesso che espone una singola applicazione ("myapp") senza prefisso ma rende disponibili anche **admin**, **appadmin** e i file statici:

```
routes_in = (
  ('/admin/$anything', '/admin/$anything'),
  ('/static/$anything', '/myapp/static/$anything'),
  ('/appadmin/$anything', '/myapp/appadmin/$anything'),
  ('/favicon.ico', '/myapp/static/favicon.ico'),
  ('/robots.txt', '/myapp/static/robots.txt'),
)
routes_out = [(x, y) for (y, x) in routes_in[:-2]]
```

La sintassi per gli instradamenti è più complessa di quella indicata nei semplici esempi visti finora. Ecco una esempio più generale e complesso:

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

Questo esempio instrada le richieste `https` `POST` all'host `www.web2py.com` da un IP remoto che corrisponde alla seguente espressione regolare

```
140\.191\.\d+\.\d+
```

e che richiede una pagina che corrisponde alla seguente espressione

```
/(?P<any>.*)\.php!
```

in:

```
/test/default/index?vars=\g<any>
```

dove `\g<any>` è sostituito dal valore trovato dall'espressione regolare corrispondente.

La sintassi generale è:

```
[remote address]:[protocol]://[host]:[method] [path]
```

Tutta la stringa è considerata un'espressione regolare, perciò il "." deve sempre essere codificato ed ogni sub-espressione trovata può essere catturata con "(?P<...>...)", secondo la sintassi di Python per le espressioni regolari.

In questo modo è possibile reinstradare le richieste basandosi sull'indirizzo IP del client, sul dominio, sul tipo della richiesta, sul metodo e sul path. Inoltre è possibile mappare virtual host differenti su applicazioni differenti. Ogni sub-espressione trovata può essere usata per costruire la URL ed, eventualmente, passata come una variabile di tipo GET.

Tutti i principali server web, come per esempio Apache e lighttpd, hanno la capacità di riscrivere le URL. In un ambiente di produzione è consigliato utilizzare direttamente tali funzionalità.

4.16 Instradamenti degli errori

E' possibile utilizzare "routes.py" per reinstradare gli utenti verso azioni speciali in caso di errore sul server. Questa regola può essere indicata globalmente, per ciascuna applicazione, per ogni codice di errore e per la combinazione di applicazione e codice d'errore. Ecco un esempio:

```
routes_onerror = [
  ('init/400', '/init/default/login'),
  ('init/*', '/init/static/fail.html'),
  ('*/404', '/init/static/cantfind.html'),
  ('*/*', '/init/error/index')
]
```

Per ogni tupla la prima stringa è ricercata in "[appname]/[error code]". Se la stringa viene trovata l'utente è reindirizzato alla URL nella seconda stringa della tupla. Se web2py ha emesso un ticket per l'errore questo viene passato alla nuova URL come una variabile di tipo GET chiamata "ticket".

Gli errori che non corrispondono a nessun instradamento vengono visualizzati in una pagina d'errore di default. Questa pagina può essere personalizzata:

```
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>'''
```

La prima variabile contiene il messaggio d'errore relativo alla richiesta di una applicazione non valida. La seconda variabile contiene il messaggio d'errore relativo all'emissione di un ticket.

4.17 Cron

La funzionalità *cron* di web2py fornisce alle applicazioni la capacità di eseguire delle operazioni ad orari prefissati, in modo indipendente dalla piattaforma (Linux, Unix, Mac OS X, Windows) su cui web2py è in esecuzione. Per ogni applicazione questa funzionalità è definita dal file di crontab "app/cron/crontab" che segue la sintassi definita in (45) (con alcune estensioni specifiche per web2py).

Questo significa che ciascuna applicazione può avere una configurazione separata e che questa può essere cambiata dall'interno di web2py senza conseguenze sul Sistema Operativo.

Ecco un esempio:

```
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
```

Le ultime due linee dell'esempio precedente utilizzano delle estensioni alla sintassi standard di cron per fornire funzionalità aggiuntive di web2py. Il

cron di web2py ha una sintsassi extra per supportare eventi specifici di web2py.

Se il task (o lo script) è preceduto da un asterisco (*) e termina con ".py" sarà eseguito nell'ambiente runtime di web2py. Questo significa che saranno disponibili tutti i controller e i modelli. Se si utilizzano due asterischi (**) i modelli non saranno eseguiti. E' preferibile utilizzare questa seconda modalità in quanto ha meno sovraccarico nell'esecuzione e consente di evitare eventuali problemi di deadlock. Le funzioni eseguite all'interno dell'ambiente di runtime di web2py richiedono una esplicita chiamata alla funzione `db.commit()` per completare la transazione che altrimenti sarà annullata. web2py non genera ticket o traceback utili quando è usato in modalità *shell* (modalità nella quale sono eseguiti i task di cron). E' bene assicurarsi che il codice sia eseguito senza errori prima di utilizzarlo come un task automatizzato con cron, perchè è improbabile che possa essere controllato mentre è in esecuzione. Inoltre è bene fare attenzione nell'utilizzo dei modelli: poichè l'esecuzione avviene in un processo separato, devono essere considerati eventuali lock al database per evitare che le pagine web dell'applicazione attendano che il task sia completato. Per questo, se non è necessario utilizzare il modello nel task di cron è bene utilizzare la sintassi ** per eseguire il task.

E' anche possibile eseguire una funzione di un controller. Non è necessario specificarne il path. Il controller e la funzione saranno quelli dell'applicazione in cui il cron è definito. Anche in questo caso è bene fare attenzione ai problemi sopra indicati. Per esempio:

```
1 */30  *  *  *  *  root *mycontroller/myfunction
```

Se nel primo campo di una linea del file di crontab si specifica @reboot il task relativo sarà eseguito solo una volta all'avvio di web2py. Questa caratteristica può essere utilizzata per precaricare, controllare od inizializzare dati all'avvio dell'applicazione. Poichè i task di cron sono eseguiti contemporaneamente all'applicazione in caso che l'applicazione debba aspettare il completamento del task per funzionare correttamente dovranno essere implementati gli opportuni controlli. Per esempio:

```
@reboot  *  *  *  *  root *mycontroller/myfunction
```

A seconda di come viene eseguito web2py sono disponibili quattro modalità di operazione per il cron:

- *Soft cron*: disponibile in tutte le modalità d'esecuzione di web2py
- *Hard cron*: disponibile se si usa il server web interno di web2py (sia direttamente che tramite il modulo mod_proxy di Apache)
- *External cron*: disponibile se si ha accesso al servizio cron del Sistema Operativo
- Nessun cron

Il default è *hard cron* se si sta utilizzando il server web interno, in tutti gli alri casi il default è *soft cron*.

soft cron è il default anche se si sta usando CGI, FASTCGI o WSGI. I task indicati nel file di cron saranno eseguiti alla prima chiamata (caricamento di una pagina) a web2py dopo che sarà scaduto il tempo specificato nel crontab (ma successivamente al caricamento della pagina stessa, in modo che non vi siano ritardi per l'utente). Ovviamente in questo modo l'effettiva esecuzione del task dipende dal traffico che il sito riceve. Inoltre il task può essere interrotto se il server web ha impostato un timeout per il caricamento delle pagine. Se queste limitazioni non sono accettabili si può utilizzare *external cron*. *Soft cron* è un compromesso ragionevole ma se il server web mette a disposizioni altri metodi di cron questi dovrebbero essere preferiti.

Hard cron è il default se si utilizza il server web interno (anche tramite mod_proxy). *Hard cron* è eseguito in un thread parallelo e pertanto non ha limitazioni riguardo la precisione dei tempi di esecuzione.

External cron non è automaticamente usato in nessun caso perchè richiede l'accesso al cron di sistema. Viene eseguito in un processo parallelo quindi le limitazioni del *soft cron* non sono presenti. Questo è il modo raccomandato

per utilizzare il cron con WSGI o FASTCGI. Ecco un esempio della linea da aggiungere alla crontab di sistema (solitamente /etc/crontab):

```
0-59/1 * * * * web2py cd /var/www/web2py/ && python web2py.py -C -D 1 >> /tmp/cron.
    output 2>&1
```

Se si utilizza *external cron* assicurarsi che web2py sia avviato con il parametro -N in modo da non avere collisioni con altri tipi di cron.

Se non è necessaria nessuna funzionalità legata a cron questo può essere disabilitato con il parametro -N durante l'avvio di web2py. La disabilitazione di cron potrebbe disattivare anche alcuni task di manutenzione (come la pulizia automatica delle cartelle delle sessioni). L'utilizzo più comune di questo parametro si ha nel caso che si utilizzi *external cron* oppure si voglia eseguire il debug dell'applicazione senza nessuna interferenza da parte di cron.

4.18 Processi in backgroup e code dei task

Sebbene cron sia utile per eseguire task ad intervalli regolari non è sempre la soluzione migliore per eseguire un processo in backgroud. Per questo specifico compito web2py mette a disposizione la possibilità di eseguire qualsiasi script di Python come se fosse eseguito dall'interno di un controller:

```
python web2py.py -S app -M -N -R applications/app/private/myscript.py -A a b c
```

dove `-S app` indica a web2py di eseguire "myscript.py nell'applicazion "app", `-M` indica a web2py di eseguire i modelli, `-N` indica a web2py di nono eseguire cron e `-A a b c` passa per parametri opzionali della linea di comando a "myscript.py" (come `sys.args=['a','b','c']`).

Un tipico caso d'utilizzo consiste nel processare una coda: con il modello

```
db.define_table('queue',
    Field('status'),
    Field('email'),
    Field('subject'),
```

```
    Field('message'))
```

ed una applicazione che accoda i messaggi da inviare con

```
db.queue.insert(status='pending',
                email='you@example.com',
                subject='test',
                message='test')
```

Il processo in background che invia le email potrebbe essere il seguente script:

```
# in file myscript.py
import time
while True:
    rows = db(db.queue.status=='pending').select()
    for row in rows:
        if mail.send(to=row.email,
            subject=row.subject,
            message=row.message):
            row.update_record(status='sent')
        else:
            row.update_record(status='failed')
        db.commit()
    time.sleep(60) # check every minute
```

L'oggetto `mail` è definito nel file `db.py` dell'applicazione di base ed è accessibile perchè web2py è stato avviato con l'opzione `-M`. Il file `db.py` dovrebbe essere configurato per funzionare correttamente. Inoltre è importane completare le transazioni prima possibile per non bloccare il database che potrebbe essere acceduto da altri processi concorrenti.

Un processo di questo tipo, funzionante in background, non deve essere eseguito via cron (tranne nel caso di @reboot) perchè non dovrebbe esserci più di una sua istanza in esecuzione. Se si esegue con cron è possibile che il processo sia ancora attivo quando cron, alla successiva iterazione, lo rilancia causando così problemi nella gestione della coda.

4.19 Moduli di terze parti

web2py è scritto in Python e quindi può importare ed utlizzare qualsiasi modulo Python, inclusi moduli di terze parti. web2py deve solamente essere in grado di importarli.

I moduli possono essere installati nella cartella ufficiale di Python "site-packages" o in un qualsiasi altra posizione in cui l'applicazione possa trovarli.

I moduli nella cartella "site-packages" sono disponibili globalmente. Le applicazioni che richiedono moduli installati in site-packages non sono portabili a meno di installare i tali moduli separatamente. Il vantaggio di installare i moduli in "site-packages" è che più applicazioni possono utilizzarli. Per esempio il package di disegno chiamato "matplotlib" può essere installato dalla shell utilizzando il comand PEAK `easy_install`:

```
easy_install py-matplotlib
```

e può essere importato in ogni modello, vista o controller con:

```
import matplotlib
```

la distribuzione binaria di web2py per Windows ha la cartella "site-packages" al primo livello. La distribuzione binaria per Mac ha la cartella "site-packages" in:

```
web2py.app/Contents/Resources/site-packages
```

I moduli possono essere installati localmente utilizzando la cartella "modules" di ogni applicazione. Il vantaggio di tale modalità è che i moduli saranno automaticamente copiati e distribuiti con l'applicazione sebbene non disponibili globalmente a tutte le applicazioni Python. web2py dispone anche di una funzione "local_import". Ecco come si deve usare:

```
mymodule = local_import(mymodule)
```

Questa funzione ricerca "mymodule" nella cartella "modules" dell'applicazione e importa il modulo con il nome indicato a sinistra del carattere di assegnazione.

Questa funzione richiede tre argomenti: `name`, `reload` ed `app`. Quando si specifica `reload=True` il modulo sarà reimportato ad ogni richiesta, altrimenti il processo Python lo importerà una volta sola. Il default è `reload=False`. `app` è il nome dell'applicazione da cui importare il modulo e ha come default l'applicazione corrente (indicata in `request.application`).

Il motivo dell'esistenza di questa funzione è che poichè un server potrebbe eseguire diverse istanze di web2py non è agevole aggiungere a `sys.path` i diversi path dei moduli la cui ricerca diventerebbe dipendente dal loro ordine in `sys.path`.

4.20 L'ambiente d'esecuzione

> *Sebbene tutto quello che è stato appena discusso funzioni correttamente è bene costruire le applicazioni utilizzando i componenti, come descritto nel capitolo 13.*

I file di modello e i controller di web2py non sono moduli Python standard poichè non possono essere importati con il comando `import` di Python. Questo perchè i modelli e i controller sono progettati per essere eseguiti in un ambiente appositamente preparato da web2py che è stato pre-popolato con alcuni oggetti globali (request, response, session, cache e T) ed alcune funzioni ausiliarie. Questo è necessario perchè Python è un linguaggio con uno *scope* (ambito di validità) statico, mentre l'ambiente di web2py è creato dinamicamente. web2py mette a disposizione la funzione `exec_environment` per consentire l'accesso diretto ai modelli e ai controller. `exec_environment` crea un ambiente d'esecuzione di web2py, carica il file all'interno di quell'ambiente e ritorna un oggetto di tipo `Storage` che contiene l'ambiente appena creato e

che serve anche come *namespace*. Qualsiasi file Python progettato per essere eseguito nell'ambiente di esecuzione di web2py può essere caricato usando `exec_environment`. I possibili utilizzi di `exec_environment` includono:

- Accedere ai dati (modelli) da altre applicazioni.
- Accedere ad oggetti globali da altri modelli o controller.
- Eseguire le funzioni di un controller dall'interno di un altro controller.
- Caricare librerie ausiliarie globali.

Questo esempio legge le righe dalla tabella `user` nell'applicazione `cas`:

```
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)
```

Un altro esempio: se il controller "other.py" contiene la funzione:

```
1 def some_action():
2     return dict(remote_addr=request.env.remote_addr)
```

Questa azione può essere chiamata da un altro controller (o dalla shell di web2py) con:

```
1 from gluon.shell import exec_environment
2 other = exec_environment('applications/app/controllers/other.py', request=request)
3 result = other.some_action()
```

Nella linea 2 `request=request` è opzionale. Ha l'effetto di passare la richiesta corrente all'ambiente di esecuzione di "other". Senza questo argomento l'ambiente conterrebbe un nuovo oggetto request vuoto (tranne che per `request.folder`). E' anche possibile passare un oggetto `response` e un oggetto `session` ad `exec_environment`. Fare attenzione quando si passano gli oggetti `request`, `response` e `session` perchè eventuali modifiche eseguite dall'azione chiamata o da altre dipendenze nell'azione chiamata potrebbero avere effetti collaterali inaspettati.

La chiamata alla funzione nella linea 3 non esegue la vista ma ritorna solamente il dizionario a meno che `response.render` è chiamato esplicitamente in "some_action".

Un'avvertenza finale: non usare `exec_environment` in modo non appropriato. Se si vuole utilizzare il risultato di un'azione in un'altra applicazione probabilmente si dovrebbe implementare una API XML-RPC (implementare una API XML-RPC in Python è quasi banale). Non si deve usare `exec_environment` come meccanismo di redirezione, per questo è disponibile la funzione ausiliaria `redirect`.

4.21 Cooperazione

Ci sono diversi modi in cui le applicazioni possono cooperare:

- Le applicazioni possono connettersi allo stesso database e condividerne le tabelle. Non è necessario che tutte le tabelle del database siano definite in tutte le applicazioni, è sufficiente che siano definite dalle applicazioni che le utilizzano. Tutte le applicazioni, tranne una, che usano la stessa tabella devono definire le tabelle con `migrate=False`.
- Le applicazioni possono includere componenti da altre azioni utilizzando la funzione ausiliaria `LOAD` (descritta nel capitolo 13).
- Le applicazioni possono condividere le sessioni.
- Un'applicazione può chiamare l'azione di un altra applicazione remotamente con XML-RPC.
- Un'applicazione può accedere ai file di un'altra applicazione tramite il filesystem (se risiedono sullo stesso filesystem).
- Un'applicazione può chiamare le azioni di un'altra applicazione localmente utilizzando `exec_environment` come discusso più sopra.
- Un'applicazione può importare i moduli di un'altra applicazione utilizzando la sintassi:
- Le applicazioni possono importare qualsiasi modulo raggiungibile dal path di ricerca in `PYTHONPATH`, `sys.path`.
- Un'applicazione può caricare la sessione di un'altra applicazione con il comando:

```
session.connect(request, response, masterapp='appname', db=db)
```

Qui "appname" è il nome dell'applicazione *master*, quella che imposta il valore iniziale di `session_id` nel cookie. `db` è la connessione al database che contiene la tabella delle sessioni (`web2py_session`). Tutte le applicazioni che condividono le sessioni devono utilizzare lo stesso database per memorizzarle.

Un'applicazione può caricare un modulo da un'altra applicazione con:

```
othermodule = local_import('othermodule',app='otherapp')
```

Se la funzione di un modulo ha bisogno di accedere ad uno degli oggetti globali (`request`, `response`, `session`, `cache` e `T`), l'oggetto deve essere esplicitamente passato alla funzione. Non deve essere consentito al modulo di creare un'altra istanza dell'oggetto globale altrimenti la funzione si comporterà in modo inaspettato.

4.22 WSGI

web2py e WSGI hanno una relazione di amore-odio. La prospettiva degli sviluppatori di web2py nell'utilizzo di WSGI è che questo sia stato sviluppato per essere un protocollo portabile di connessione tra il server web e le applicazioni Python e quindi è utilizzato con questo obiettivo. La parte più interna di web2py è un'applicazione WSGI: `gluon.main.wsgibase`. Alcuni sviluppatori hanno spinto questa idea al limite e utlizzano WSGI come un protocollo per comunicazioni *middleware*, sviluppando le applicazioni web come una "cipolla" con molti strati (in cui ogni strato è un middleware WSGI sviluppato indipendentemente dall'intero framework). web2py non adotta questa struttura internamente. Questo perchè le funzionalità centrali di un framework (gestione dei cookie, delle sessioni, degli errori, delle transazioni, dispatching) possono essere meglio ottimizzate per la velocità e la sicurezza

se sono gestite da un unico strato che le raccoglie tutte. web2py consente comunque di utilizzare applicazioni WSGI di terze parti in tre modi diversi (incluse le loro combinazioni):

- E' possibile modificare il file "wsgihandler.py" ed includere qualsiasi middleware WSGI di terze parti.
- E' possibile connettere middleware WSGI di terze parti a qualsiasi azione di un'applicazione.
- E' possibile chiamare un'applicazione WSGI da qualsiasi azione di un'applicazione.

L'unica limitazione è che non è possibile utilizzare middleware di terze parti per sostituire le funzioni centrali di web2py.

4.22.1 Middleware esterno

Con il seguente file "wsgibase.py":

```
#...
LOGGING = False
#...
if LOGGING:
    application = gluon.main.appfactory(wsgiapp=gluon.main.wsgibase,
                                        logfilename='httpserver.log',
                                        profilerfilename=None)
else:
    application = gluon.main.wsgibase
```

quando LOGGING è impostato a `True`, `gluon.main.wsgibase` è incluso nella funzione middleware e fornisce il logging nel file "httpserver.log". Allo stesso modo si può aggiungere qualsiasi middleware di terze parti. Riferirsi alla documentazione ufficiale di WSGI per ulteriori dettagli.

4.22.2 *Middleware interno*

Per qualsiasi azione (per esempio `index`) in un controller e con un'applicazione middleware di terze parti (per esempio `MyMiddelware` che converte l'output in maiuscolo) si può usare un decoratore di web2py per applicare il middleware all'azione. Ecco l'esempio:

```
class MyMiddleware:
    """converts output to upper case"""
    def __init__(self,app):
        self.app = app
    def __call__(self,environ, start_response):
        items = self.app(environ, start_response)
        return [item.upper() for item in items]

@request.wsgi.middleware(MyMiddleware)
def index():
    return 'hello world'
```

Non è possibile garantire che tutte le applicazioni middleware di terze parti funzionino con questo meccanismo.

4.22.3 *Chiamare le applicazioni WSGI*

E' facile chiamare un'applicazione WSGI da un'azione di un controller di web2py. Ecco un esempio:

```
def test_wsgi_app(environ, start_response):
    """this is a test WSGI app"""
    status = '200 OK'
    response_headers = [('Content-type','text/plain'),
                        ('Content-Length','13')]
    start_response(status, response_headers)
    return ['hello world!\n']

def index():
    """a test action that call the previous app and escapes output"""
    items = test_wsgi_app(request.wsgi.environ,
                          request.wsgi.start_response)
    for item in items:
        response.write(item,escape=False)
```

```
15     return response.body.getvalue()
```

In questo caso l'azione `index` richiama `test_wsgi_app` e codifica i valori ritornati prima di ritornarli a sua volta. Poichè `index` non è un'applicazione WSGI deve utilizzare le normali API di web2py (come per esempio `response.write` per scrivere nel socket).

Le viste web2py utilizza Python per i modelli, i controller e le viste, sebbene per quest'ultime utilizzi una sintassi leggermente modificata per consentire di creare codice più leggibile senza imporre alcuna restrizione sul corretto utilizzo di Python.

Lo scopo di una vista è di includere codice Python in un documento HTML. Questo pone due problemi di fondo:

- Come dev'essere identificato il codice?
- L'indentazione dovrebbe essere basata sulle regole di Python o su quelle dell'HTML?

web2py utilizza `{{ ... }}` per isolare il codice inserito nell'HTML. Il vantaggio dell'utilizzo di parentesi graffe invece che di parentesi angolari è che è sono trasparenti per i comuni editor HTML. Questo consente allo sviluppatore di utilizzare qualsiasi editor per creare le viste di web2py.

Poichè lo sviluppatore sta inserendo codice Python nell'HTML il documento dovrebbe essere indentato secondo le regole dell'HTML e non secondo quelle di Python. Per questo è consentito l'utilizzo di codice Python non indentato all'interno dei tag `{{ ... }}`. Poichè Python normalmente utilizza l'indentazione per delimitare i blocchi di codice è necessario un metodo diverso per delimitarli, questo è il motivo per cui il linguaggio di template di web2py fa uso della parola chiave Python `pass`.

Un blocco di codice inizia con una linea terminante con il carattere due punti

(:) e termina con una linea che inizia con il comando `pass`*. La parola chiave* `pass` *non è necessaria quando la fine del blocco è evidente dal contesto.*

Ecco un esempio:

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

E' da notare che `pass` è una parola chiave di Python, non di web2py. Alcuni editor (come per esempio Emacs) utilizzano `pass` per dividere i blocchi e re-indentare il codice automaticamente.

Il linguaggio di template di web2py funziona in questo modo. Quando trova qualcosa come:

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

lo traduce in un programma:

```
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` scrive in `response.body`.

In caso di errori in una vista di web2py il report dell'errore mostra il codice generato dalla vista, non il codice della vista scritto dallo sviluppatore. Lo sviluppatore è aiutato nel debug del codice dall'evidenziazione della parte di codice che ha causato l'errore (e che può essere controllato con un editor HTML o un analizzatore di DOM nel browser).

Notare anche che:

```
{{=x}}
```

genera

```
response.write(x)
```

Le variabili iniettate nel codice HTML in questo modo sono già correttamente codificate. La codifica è ignorata nel caso in cui `x` sia un oggetto `XML` anche se `escape` è impostato a `True`.

Ecco un esempio che introduce la funzione helper `H1`:

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

che è tradotto in:

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

durante la valutazione dell'espressione l'oggetto `H1` ed i suoi componenti sono serializzati ricorsivamente, codificati e scritti nel corpo della risposta. I tag generati da `H1` ed il codice HTML interno non sono codificati. Questo meccanismo garantisce che tutto il testo (e solamente il testo) visualizzato nella pagina web sia sempre correttamente codificato, prevenendo così vulnerabilità di tipo XSS. Allo stesso tempo il codice è semplice e facile da correggere.

Il metodo `response.write(obj, escape=True)` richiede due argomenti: l'oggetto da scrivere e un argomento `escape` che indica se questo deve essere codificato (impostato a `True` di default). Se `obj` ha un metodo `.xml()` questo viene chiamato e il risultato è scritto nel corpo della risposta (e l'argomento `escape` viene ignorato). Altrimenti viene utilizzato il metodo `__str__` dell'oggetto per serializzarlo e se l'argomento `escape` è impostato a `True` viene codificato. Tutte le funzioni ausiliarie (come `H1` nell'esempio) sono oggetti che sono in grado di serializzarsi tramite il metodo `.xml()`.

Tutto questo è eseguito automaticamente. Non c'è mai necessità (e non dovrebbere essere fatto) di chiamare esplicitamente il metodo `response.write`.

4.23 Sintassi di base

Il linguaggio di template di web2py supporta tutte le strutture di controllo di Python. Di seguito sono illustrati alcuni esempi di tali strutture che possono essere nidificate secondo le normali tecniche di programmazione.

4.23.1 for... in

Nei template è possibile ciclare su ogni oggetto iterabile:

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

che genera:

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

In questo esempio `item` è un oggetto iterabile (come una lista, una tupla, un oggetto `Rows` o qualsiasi oggetto implementato come un iteratore). Gli elementi visualizzati sono serializzati e codificati.

4.23.2 while

Si può creare un ciclo utilizzando la keyword `while`:

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

che genera:

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

4.23.3 if... elif... else

E' possibile usare le clausole condizionali:

```
{{
import random
k = random.randint(0, 100)
}}
<h2>
{{=k}}
{{if k % 2:}}is odd{{else:}}is even{{pass}}
</h2>
```

che genera:

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

Poichè è evidente che `else` chiude il primo blocco `if` non è necessario il comando `pass` ed il suo utilizzo sarebbe scorretto. E' tuttavia necessario chiudere esplicitamente il blocco `else` con il comando `pass`.

In Python "else if" è scritto `elif` come nel seguente esempio:

```
{{
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>
```

che genera:

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

4.23.4 *try... except... else... finally*

E' anche possibile utilizzare i comandi `try ... except` come nel seguente esempio:

```
{{try:}}
Hello {{= 1 / 0}}
{{except:}}
division by zero
{{else:}}
no division by zero
{{finally}}
<br />
{{pass}}
```

che produrrà il seguente output:

```
Hello
division by zero
<br />
```

Questo esempio evidenzia come tutto l'ouput generato prima di un'eccezione è visualizzato anche se è inserito all'interno del blocco `try` (incluso l'output che precede l'eccezione). "Hello" viene scritto perchè precede l'eccezione.

4.23.5 def... return

Il linguaggio di template di web2py consente allo sviluppatore di definire e di implementare funzioni che possono ritornare qualsiasi oggetto Python o una stringa. Ecco due esempi:

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

produrrà il seguente output:

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

La funzione `itemize1` ritorna un oggetto che è inserito nel punto in cui è chiamata la funzione.

Con il seguente codice:

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

viene prodotto esattamente lo stesso ouptut di prima. In questo caso la funzione `itemize2` rappresenta un pezzo di codice HTML che andrà a sostituire il tag in cui la funzione viene chiamata. E' da notare che nella chiamata della funzione `itemize2` non è presente il carattere di uguale (=) in quanto la funzione non ritorna del testo, ma lo scrive direttamente nel corpo della risposta. C'è un accorgimento da seguire: le funzioni definite all'interno di una vista devono terminare con il comando `return` altrimenti non funzionerà l'indentazione automatica.

4.24 Helper HTML

In una vista il seguente codice:

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

genera il seguente HTML:

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

`DIV` è una classe helper, nel senso che è una classe che può essere utilizzata per costruire codice HTML da programma. Corrisponde al tag HTML `<div>`.

Gli argomenti posizionali sono interpretati come oggetti contenuti all'interno dei tag di apertura e chiusura. Gli argomenti con nome che iniziano con un underscore (_) sono interpretati come attributi del tag HTML (senza il carattere di underscore). Alcuni helper hanno argomenti con nome che non iniziano con un underscore e che sono specifici del tag.

Gli helper `A`, `B`, `BEAUTIFY`, `BODY`, `BR`, `CENTER`, `CODE`, `DIV`, `EM`, `EMBED`, `FIELDSET`, `FORM`, `H1`, `H2`, `H3`, `H4`, `H5`, `H6`, `HEAD`, `HR`, `HTML`, `I`, `IFRAME`, `IMG`, `INPUT`, `LABEL`, `LEGEND`, `LI`, `LINK`, `OL`, `UL`, `MARKMIN`, `MENU`, `META`, `OBJECT`, `ON`, `OPTION`, `P`, `PRE`, `SCRIPT`, `OPTGROUP`, `SELECT`, `SPAN`, `STYLE`, `TABLE`, `TAG`, `TD`, `TEXTAREA`, `TH`, `THEAD`, `TBODY`, `TFOOT`, `TITLE`, `TR`, `TT`, `URL`, `XHTML`, `XML`, `xmlescape`, `embed64` possono essere usati per costruire espressioni complesse che possono essere serializate in XML (49) (50).

Per esempio:

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

genera il seguente codice HTML:

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

Gli helper in web2py sono più che un sistema per generare codice HTML senza concatenare le stringhe, sono una rappresentazione lato server del

Modello ad oggetti del documento (DOM, *Document Object Model*). Gli oggetti che compongono il DOM sono referenziati con la loro posizione, e gli helper si comportano come liste rispetto ai loro componenti:

```
>>> 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>
```

mentre gli attributi degli helper sono referenziati con il loro nome e gli helper si comportano come dizionari rispetto ai loro attributi:

```
>>> 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>
```

4.24.1 XML

`XML` è un oggetto usato per incapsulare il testo che non deve essere codificato. Il testo può o non può contenere XML valido, potrebbe, per esempio contenere Javascript.

Il testo in questo esempio è codificato:

```
>>> print DIV("<b>hello</b>")
&lt;b&gt;hello&lt;/b&gt;
```

utilizzando `XML` si può impedire la codifica:

```
>>> print DIV(XML("<b>hello</b>"))
<b>hello</b>
```

A volte potrebbe essere necessario utilizzare il codice HTML memorizzato in

una variabile, ma l'HTML può contenere tag non sicuri, come per esempio uno script:

```
>>> print XML('<script>alert("unsafe!")</script>')
<script>alert("unsafe!")</script>
```

Un input eseguibile di questo tipo (come per esempio un commento inserito in un blog) non è sicuro perchè potrebbe essere utilizzato per generare attacchi di tipo XSS (*Cross Site Scripting*) contro gli altri visitatori della pagina.

L'helper `XML` di web2py può verificare il testo per evitere l'inserimento di codice pericolo codificando tutti i tag tranne quelli esplicitamente consentiti. Per esempio:

```
>>> print XML('<script>alert("unsafe!")</script>', sanitize=True)
&lt;script&gt;alert(&quot;unsafe!&quot;)&lt;/script&gt;
```

Il costruttore di `XML` di default considera alcuni tag e alcuni attributi sicuri. Questo comportamento può essere modificato utilizzando gli argomenti opzionali `permitted_tags` e `allowed_attributes`. Questi sono i valori di default dei due argomenti:

```
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']})
```

4.24.2 Gli helper di web2py

A

Questo helper è utilizzato per costruire i link.

```
>>> 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

Trasforma il testo in grassetto.

```
>>> print B('<hello>', XML('<i>world</i>'), _class='test', _id=0)
<b id="0" class="test">&lt;hello&gt;<i>world</i></b>
```

BODY

Genera il corpo della pagina HTML.

```
>>> print BODY('<hello>', XML('<b>world</b>'), _bgcolor='red')
<body bgcolor="red">&lt;hello&gt;<b>world</b></body>
```

CENTER

Centra il testo nella pagina.

```
>>> print CENTER('<hello>', XML('<b>world</b>'),
>>>               _class='test', _id=0)
<center id="0" class="test">&lt;hello&gt;<b>world</b></center>
```

CODE

Questo helper esegue l'evidenziazione della sintassi per il codice Python, C, C++, HTML e web2py. E' preferibile utilizzare `CODE` piuttosto che `PRE` per la visualizzazione del codice. `CODE` è anche in grado di creare i corretti link per la documentazione delle API di web2py.

Ecco un esempio con del codice Python:

```
>>> 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>
```

Questo è un esempio per il codice HTML:

```
>>> print CODE(
>>>   '<html><body>{{=request.env.remote_add}}</body></html>',
>>>   language='html')
```

```

<table> ... <code> ...
<html><body>{{=request.env.remote_add}}</body></html>
... </code> ... </table>
```

Questi sono gli argomenti di default per l'helper `CODE`:

```
CODE("print 'hello world'", language='python', link=None, counter=1, styles={})
```

I valori supportati per l'argomento `language` sono "python", "html_plain", "c", "cpp", "web2py" ed "html". Con "html" `{{` e `}}` sono interpretati come tag contenenti codice "web2py" mentre con "html_plain" non lo sono.

Se l'argomento `link` specifica un valore, per esempio "/examples/global/vars/", le API di web2py referenziate nel codice sono collegate alla documentazione alla URL del link. Per esempio "request" sarebbe collegata a "/examples/global/vars/request". Nel precedente esempio la URL del link è gestita dall'azione "var" del controller "global.py" che è distribuito come parte dell'applicazione "examples".

L'argomento `counter` è utilizzato per la numerazione delle linee. Può assumere tre differenti valori: `None` per disattivare la numerazione delle linee;

un valore numerico da cui iniziare la numerazione delle linee; una stringa, intepretata come un prompt, senza numerazione.

DIV

Tutti gli helper (tranne `XML`) sono derivati da `DIV` ed ereditano i suoi metodi di base.

```
>>> print DIV('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<div id="0" class="test">&lt;hello&gt;<b>world</b></div>
```

EM

Enfatizza il suo contenuto.

```
>>> print EM('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<em id="0" class="test">&lt;hello&gt;<b>world</b></em>
```

FIELDSET

Usato per creare un campo di imput con la sua etichetta di testo.

```
>>> print FIELDSET('Height:', INPUT(_name='height'), _class='test')
<fieldset class="test">Height:<input name="height" /></fieldset>
```

FORM

Questo è uno degli helper più importanti di web2py. Nella sua forma più semplice crea un tag `<form> ... </form>` ma, poichè gli helper sono oggetti in grado di analizzare il loro contenuto, `FORM` è in grado di analizzare i dati inseriti (per esempio per eseguirne la validazione). Questo sarà spiegato in dettaglio nel Capitolo 7.

```
>>> print FORM(INPUT(_type='submit'), _action='', _method='post')
<form enctype="multipart/form-data" action="" method="post">
<input type="submit" /></form>
```

L'attributo "enctype" è "multipart/form-data" per default.

Il costruttore di un FORM e di un SQLFORM può anche avere un argoment speciale chiamato . Quando un dizionario è passato come i suoi oggetti sono trasformati in campi di INPUT nascosti. Per esempio:

```
>>> 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

Questi helper servono per le intestazioni dei paragrafi e dei sotto-paragrafi:

```
>>> print H1('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<h1 id="0" class="test">&lt;hello&gt;<b>world</b></h1>
```

HEAD

Serve ad indicare l'HEAD della pagina HTML.

```
>>> print HEAD(TITLE('<hello>', XML('<b>world</b>')))
<head><title>&lt;hello&gt;<b>world</b></title></head>
```

HTML

Questo helper si comporta in modo leggermente differente. Oltre a creare i tag <html>... </html> lo fa precedere dalle stringhe *doctype* (52; 53; 54).

```
>>> 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>
```

L'helper HTML può avere anche alcuni argomenti opzionali che hanno i seguenti default:

```
HTML(..., lang='en', doctype='transitional')
```

dove doctype può assumere i valori 'strict', 'transitional', 'frameset', 'html5' o una stringa.

XHTML

XHTML è simile ad HTML ma crea un doctype di tipo XHTML.

```
XHTML(..., lang='en', doctype='transitional', xmlns='http://www.w3.org/1999/xhtml')
```

dove doctype può essere 'strict', 'transitional', 'frameset' o una stringa:

INPUT

Crea un tag di tipo `<input ... />`. Questo tag può contenere altri tag ed è chiuso con `/>` invece di `>`. Ha un attributo opzionale `_type` che può essere impostato a "text" (il default), "submit", "checkbox" o "radio".

```
>>> print INPUT(_name='test', _value='a')
<input value="a" name="test" />
```

Ha anche un altro argomento opzionale chiamato "value", distinto da "_value". Quest'ultimo imposta il valore di default per il campo di input; il primo imposta il suo valore corrente. Per un input di tipo "text" il primo ha la precedenza sul secondo:

```
>>> print INPUT(_name='test', _value='a', value='b')
<input value="b" name="test" />
```

Per i pulsanti di tipo "radio" `INPUT` imposta l'attributo "checked" per il valore specificato in "value":

```
>>> 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
```

e allo stesso modo per le checkbox:

```
>>> 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

Questo helper include una pagina web nella pagina corrente. La URL dell'altra pagina è indicata nell'attributo "_src".

```
>>> print IFRAME(_src='http://www.web2py.com')
<iframe src="http://www.web2py.com"></iframe>
```

LABEL

Utilizzato per creare una etichetta per un campo di input.

```
>>> print LABEL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<label id="0" class="test">&lt;hello&gt;<b>world</b></label>
```

LI

Crea un oggetto di una lista e dovrebbe essere contenuto in un tag UL o OL.

```
>>> print LI('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<li id="0" class="test">&lt;hello&gt;<b>world</b></li>
```

LEGEND

Usato per creare un tag di legenda per un campo in un form.

```
>>> print LEGEND('Name', _for='somefield')
<legend for="somefield">Name</legend>
```

META

Può essere usato per costruire dei tag di tipo META nell'HEAD dell'HTML. Per esempio:

```
>>> print META(_name='security', _content='high')
<meta name="security" content="high" />
```

MARKMIN

Implementa la sintassi *markmin* per i wiki. Converte il testo in input in codice html seconde le regole della sintassi *markmin*:

```
>>> print MARKMIN("''this is **bold** or ''italic'' and this [[a link http://web2py.
    com]]"')
<p>this is <b>bold</b> or <i>italic</i> and
this <a href="http://web2py.com">a link</a></p>
```

La sintassi *markmin* è descritta nel seguente file incluso in web2py:

```
http://127.0.0.1:8000/examples/static/markmin.html
```

OBJECT

E' usato per incorporare oggetti (per esempio, un file di Flash) nell'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

Una lista ordinata. La lista dovrebbe contenere tag `LI`. Gli argomenti di `OL` che non sono di tipo `LI` sono automaticamente racchiusi nei tag `<li> ... </li>`.

```
>>> 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

Questo helper esiste solo per mantenere la retro-compatibilità con le versioni precedenti di web2py ed è semplicemente un alias per `True`. E' usato esclusivamente per le checkbox ed è deprecato in quanto l'utilizzo di `True` è più *pythonico*.

```
>>> print INPUT(_type='checkbox', _name='test', _checked=ON)
<input checked="checked" type="checkbox" name="test" />
```

OPTGROUP

Consente di raggruppare le opzioni di una SELECT ed è utile per personalizzare i campi utilizzando i CSS.

```
>>> print SELECT('a', OPTGROUP('b', 'c'))
<select>
  <option value="a">a</option>
  <optgroup>
    <option value="b">b</option>
    <option value="c">c</option>
  </optgroup>
</select>
```

OPTION

Questo helper va usato in combinazione con SELECT/OPTION.

```
>>> print OPTION('<hello>', XML('<b>world</b>'), _value='a')
<option value="a">&lt;hello&gt;<b>world</b></option>
```

Come nel caso di INPUT, web2py fa distinzione tra "_value" (il valore di OPTION), e "value" (il valore corrente della SELECT). L'opzione è selezionata quando i due valori sono uguali.

```
>>> print SELECT('a', 'b', value='b'):
<select>
<option value="a">a</option>
<option value="b" selected="selected">b</option>
</select>
```

P

Questo è il tag per definire un paragrafo.

```
>>> print P('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<p id="0" class="test">&lt;hello&gt;<b>world</b></p>
```

PRE

Genera un tag <pre> ... </pre> per visualizzare testo già formattato. Per la visualizzazione di codice è preferibile usare l'helper CODE.

```
>>> print PRE('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<pre id="0" class="test">&lt;hello&gt;<b>world</b></pre>
```

SCRIPT

Questo tag serve per includere uno script (come Javascript) o un link ad uno script. Il contenuto tra i tag è visualizzato come un commento per non creare problemi ai browser molto vecchi.

```
>>> print SCRIPT('alert("hello world");', _language='javascript')
<script language="javascript"><!--
alert("hello world");
//--></script>
```

SELECT

Crea un tag `<select> ... </select>`. Questo è usato con l'helper `OPTION`. Gli argomenti di `SELECT` che non sono oggetti `OPTION` sono automaticamente convertiti.

```
>>> 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

`SPAN` è simile a `DIV` ma è utilizzato per il contenuto inline piuttosto che in un blocco.

```
>>> print SPAN('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<span id="0" class="test">&lt;hello&gt;<b>world</b></span>
```

STYLE

E' simile a `SCRIPT` ma è usato per includere codice CSS o un link ad esso. Qui, per esempio, è incluso del codice CSS:

```
>>> print STYLE(XML('body {color: white}'))
<style><!--
body { color: white }
//--></style>
```

e qui è definito un collegamento ad un foglio di stile CSS:

```
>>> print STYLE(_src='style.css')
<style src="style.css"><!--
//--></style>
```

TABLE, TR, TD

Questi tag (insieme ai tag opzionali creati con gli helper `THEAD`, `TBODY` e `TFOOTER`) sono utilizzati per costruire tabelle HTML.

```
>>> 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` si aspetta come contenuto degli oggetti di tipo `TD`; gli argomenti che non sono oggetti `TD` sono convertiti automaticamente:

```
>>> 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>
```

E' facile convertire un'array in Python in una tabella HTML usando la notazione * di Python per gli argomenti delle funzioni, che trasforma gli elementi di una lista in una serie di argomenti posizionali per la funzione. Ecco come fare:

```
>>> 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>
```

E' anche possibile utilizzare una sintassi più compatta:

```
>>> 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

Questo helper opzionale è utilizzato per le righe che contengono il corpo della tabella, per differenziarle dalle righe che compongono l'intestazione e la chiusura della tabella:

```
>>> print TBODY(TR('<hello>'), _class='test', _id=0)
<tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>
```

TEXTAREA

Con questo helper è possibile creare il tag `<textarea> ... </textarea>`.

```
>>> print TEXTAREA('<hello>', XML('<b>world</b>'), _class='test')
<textarea class="test" cols="40" rows="10">&lt;hello&gt;<b>world</b></textarea>
```

L'unico accorgimento di cui tener conto è che utilizzando l'argomento "value" (che è opzionale) si sovrascrive il suo contenuto (l'HTML interno).

```
>>> print TEXTAREA(value="<hello world>", _class="test")
<textarea class="test" cols="40" rows="10">&lt;hello world&gt;</textarea>
```

TFOOT

E' utilizzato per definire le righe di chiusura delle tabelle ed è opzionale:

```
>>> print TFOOT(TR(TD('<hello>')), _class='test', _id=0)
<tfoot id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tfoot>
```

TH

`TH` è utilizzato al posto di `TD` nelle intestazioni delle tabelle.

```
>>> print TH('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<th id="0" class="test">&lt;hello&gt;<b>world</b></th>
```

THEAD

Utilizzato per definire le righe di intestazione di una tabella.

```
>>> print THEAD(TR(TD('<hello>')), _class='test', _id=0)
<thead id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></thead>
```

TITLE

Utilizzato per impostare il titolo di una pagina nell'header HTML.

```
>>> print TITLE('<hello>', XML('<b>world</b>'))
<title>&lt;hello&gt;<b>world</b></title>
```

TR

Questo tag identifica una riga di una tabella. Dovrebbe essere utilizzato all'interno di una tabella e dovrebbe contenere i tag `<td> ... </td>`. Gli argomenti di `TR` che non sono oggetti di tipo `TD` sono automaticamente convertiti.

```
>>> 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

Con questo tag è possibile utilizzare caratteri a spaziatura fissa:

```
>>> print TT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
<tt id="0" class="test">&lt;hello&gt;<b>world</b></tt>
```

UL

Identifica una lista non ordinata e dovrebbe contenere oggetti di tipo `LI`. Se il suo contenuto non è di tipo `LI` gli oggetti vengono convertiti automaticamente.

```
>>> 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>
```

4.24.3 Helper personalizzati

Potrebbe essere necessario generare dei tag XML personalizzati. web2py dispone dell'helper `TAG`, un generatore di tag universale.

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

genera il seguente codice XML:

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

Gli argomenti "a", "b" e "d" sono automaticamente codificati; per evitare questa codifica utilizzare l'helper `XML`. Utilizzando `TAG` è possibile generare tag HTML/XML non previsti nell'API di web2py. I tag possono essere nidificati e sono serializzati con `str()`.

Una sintassi equivalente è;

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

E' da notare che `TAG` è un oggetto e che `TAG.name` o `TAG['name']` sono funzioni che ritornano una classe helper temporanea.

MENU

L'helper `MENU` prende una lista di liste nella forma di `response.menu` (come descritto nel Capitolo 4) e genera una struttura ad albero utilizzando liste non ordinate che rappresenta il menu. Per esempio:

```
>>> 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>
```

Ogni oggetto menu può avere un quarto elemento che rappresenta un sottomenu nidificato (in modo ricorsivo):

```
>>> 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>
```

L'helper `MENU` può avere i seguenti argomenti opzionali:

- `_class`: ha come default "web2py-menu web2py-menu-vertical" ed imposta la classe degli elementi `UL` esterni
- `ul_class`: ha come default "web2py-menu-vertical" ed imposta la classe degli elementi `UL` interni.
- `li_class`: ha come default "web2py-menu-expand" ed imposta la classe degli elementi `LI` interni.

Il file "base.css" dell'applicazione di default comprende i seguenti tipi base di menu: "web2py-menu web2py-menu-vertical" e "web2py-menu web2py-menu-horizontal".

4.25 *BEAUTIFY*

`BEAUTIFY` è utilizzato per costruire rappresentazioni HTML di oggetti composti come le liste, le tuple ed i dizionari:

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

`BEAUTIFY` ritorna un oggetto di tipo XML, serializzabile in XML, con una chiara rappresentazione dei suoi elementi. In questo caso a rappresentazione XML di:

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

verrà visualizzata come:

```
<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>
```

4.26 DOM server side e Parsing

4.26.1 elements

L'helper `DIV` è tutti gli helper derivati da esso dispongono di due metodi di ricerca: `element` and `elements`. `element` ritorna il primo elemento figlio che corrisponde alla condizione di ricerca (oppure `None` se non c'e' nessuna corrisponenza trovata). `elements` ritorna una lista di tutti gli elementi figli trovati.

element and **elements** utilizzano la stessa sintassi per definire le condizioni di ricerca. Questa sintassi consente tre possibilità diverse che possono essere mischiate insieme: espressioni di tipo jQuery, ricerca per un valore esatto di un attributo e ricerca tramite espressioni regolari.

Ecco un semplice esempio:

```
>>> a = DIV(DIV(DIV('a', _id='target',_class='abc')))
>>> d = a.elements('div#target')
>>> d[0] = 'changed'
>>> print a
<div><div><div id="target" class="abc">changed</div></div></div>
```

L'argomento senza nome del metodo `elements` è una stringa che può contenere: il nome di un tag, l'id di un tag preceduto dal simbolo cancelletto (#), la classe dell'elemento preceduta da un punto (.) o il valore esplicito di un attributo racchiuso in parentesi quadre.

Ecco quattro modi equivalenti di cercare il tag precedente tramite `id`:

```
>>> d = a.elements('#target')
>>> d = a.elements('div#target')
>>> d = a.elements('div[id=target]')
>>> d = a.elements('div',_id='target')
```

Ecco quattro modi equivalenti di cercare il tag precedente tramite `class`:

```
>>> d = a.elements('.abc')
>>> d = a.elements('div.abc')
>>> d = a.elements('div[class=abc]')
>>> d = a.elements('div',_class='abc')
```

Qualsiasi attributo può essere usato per localizzare un elemento (non solamente `id` e `class`), inclusi attributi multipli (la funzione `element` può avere più argomenti con nome) ma solo il primo elemento trovato verrà ritornato.

Usando la sintassi di jQuery "div#target" è possibile specificare criteri di ricerca multipli separati da uno spazio:

```
>>> a = DIV(SPAN('a', _id='t1'),div('b',_class='c2'))
>>> d = a.elements('span#t1, div#c2')
```

o, in modo equivalente:

```
>>> a = DIV(SPAN('a', _id='t1'),div('b',_class='c2'))
>>> d = a.elements('span#t1','div#c2')
```

Se il valore di un attributo di ricerca è specificato utilizzando un argomento con nome, questo può essere una stringa o un'espressione regolare:

```
>>> a = DIV(SPAN('a', _id='test123'),div('b',_class='c2'))
>>> d = a.elements('span',_id=re.compile('test\d{3}')
```

Uno speciale argomento con nome di `DIV` (e degli helper derivati) è `find`. Questo argomento può essere usato per specificare un valore (o un'espressione regolare) da cercare nel testo contenuto nel tag. Per esempio:

```
>>> a = DIV(SPAN('abcde'),div('fghij'))
>>> d = a.elements(find='bcd')
>>> print d[0]
<span>abcde</span>
```

oppure:

```
>>> a = DIV(SPAN('abcde'),div('fghij'))
>>> d = a.elements(find=re.compile('fg\w{3}'))
>>> print d[0]
<div>fghij</div>
```

4.26.2 *parent*

`parent` ritorna il genitore dell'elemento corrente.

```
>>> a = DIV(SPAN('a'),DIV('b'))
>>> d = a.element('a').parent()
>>> d['_class']='abc'
>>> print a
<div class="abc"><span>a</span><div>b</div></div>
```

4.26.3 *flatten*

Il metodo `flatten` serializza ricorsivamente in testo regolare (senza tag) il contenuto dei figli di un dato elemento:

```
>>> a = DIV(SPAN('this',DIV('is',B('a'))),SPAN('test'))
>>> print a.flatten()
thisisatest
```

Al metodo `flatten` può essere passato un argomento opzionale `render` che è una funzione che "appiattisce" il contenuto utilizzando un protocollo differente, Ecco un esempio di come serializzare alcuni tag nella sintassi dei wiki Markmin:

```
>>> a = DIV(H1('title'),P('example of a ',A('link',_href='#test')))
>>> from gluon.html import markmin_serializer
>>> print a.flatten(render=markmin_serializer)
# title

example of [[a link #test]]
```

Al momento della scrittura di questo manuale sono disponibili solo due serializzatori: `markmin_serializer` e `markdown_serializer`.

4.26.4 *Parsing*

L'oggetto TAG è anche un analizzatore XML/HTML. Può leggere del testo e convertirlo in una struttura ad albero composta di helper. Questo ne consente la manipolazione utilizzando le API sopra descritte:

```
>>> html = '<h1>Title</h1><p>this is a <span>test</span></p>'
>>> parsed_html = TAG(html)
>>> parsed_html.element('span')[0]='TEST'
>>> print parsed_html
<h1>Title</h1><p>this is a <span>TEST</span></p>
```

4.27 *Page Layout*

Le viste possono essere estese ed includere altre viste in una struttura ad albero.

Per esempio una vista chiamata "index.html" potrebbe estendere "layout.html" ed includere "body.html". Allo stesso tempo "layout.html" può includere un "header.html" ed un "footer.html".

La radice dell'albero viene chiamata vista di layout. Proprio come ogni altro file di template HTML può essere modificata utilizzanto l'interfaccia amministrativa di web2py. Il nome "layout.html" è solamente una convenzione.

Ecco una pagina minimale che estende "layout.html" ed include la vista "page.html":

```
{{extend 'layout.html'}}
<h1>Hello World</h1>
{{include 'page.html'}}
```

Il file di layout esteso deve contenere una direttiva `{{include}}` del tipo:

```
<html><head><title>Page Title</title></head>
  <body>
    {{include}}
  </body>
```

```
</head>
```

Quando la vista viene chiamata il layout esteso è caricato e la vista chiamante sostituisce la direttiva `{{include}}` nel layout. Il processo continua ricorsivamente fino a che tutte le direttive `extend` ed `include` sono state elaborate. Il template risultante è poi trasformato in codice Python.

> `extend` *ed* `include` *non sono comandi Python ma direttive speciali dei template di web2py.*

I layout sono utilizzati per incapsulare funzionalità comuni delle pagine (intestazioni, piè di pagina, menu) e, sebbene non siano obbligatori, rendono le applicazioni più facili da scrivere e da mantenere. In particolare è bene scrivere layout che tengano conto delle seguenti variabili che possono essere impostate dal controller. Usando queste variabili note i layout potranno essere interscambiabili:

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

Eccetto `menu` e `files` queste variabili contengono tutte stringhe ed il loro significato dovrebbe essere evidente.

`response.menu` è una lista di tuple di 3 o di 4 elementi. I tre elemeni sono: il nome del link, un valore booleano che indica se il link è attivo (è, cioè, la pagina corrente) e la URL della pagina. Per esempio:

```
response.menu = [('Google', False, 'http://www.google.com',[]),
                 ('Index',  True,  URL('index'), [])]
```

il quarto elemento della tupla è un sotto-menu ed è opzionale.

`response.files` è una lista di file CSS e JS che sono richiesti dalla pagina.

E' bene inoltre di usare:

```
{{include 'web2py_ajax.html'}}
```

nell'intestazione dell'HTML, in quando questo file include le librerie jQuery e definisce alcune funzioni Javascript necessarie per la retro-compatibilità e per effetti speciali in Ajax. "web2py_ajax.html" include i tag `response.meta` nella vista, la base jQuery, il calendario jQuery e tutti i file CSS e JS richiesti da `response.files`.

4.27.1 Layout di default delle pagine

Ecco un esempio minimale di "views/layout.html" che è distribuito con l'applicazione **welcome** di web2py. Ogni nuova applicazione creata con web2py avrà un layout di default simile:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="{{=T.accepted_language or 'en
     '}}">
  <head>
    <title>{{=response.title or request.application}}</title>
    <link rel="shortcut icon"
      href="{{=URL(request.application,'static','favicon.ico')}}"
      type="image/vnd.microsoft.icon">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />

    {{######  require CSS and JS files for this page (read info in base.css) }}
    {{response.files.append(URL(request.application,'static','base.css'))}}
    {{response.files.append(URL(request.application,'static','superfish.js'))}}

    {{###### include web2py specific js code (jquery, calendar, form stuff) }}
    {{include 'web2py_ajax.html'}}
  </head>
  <body>
    <div class="flash">{{=response.flash or ''}}</div>
    <div class="ez-mr wrapper" id="layout">

      {{###### Layout 3 from http://www.ez-css.org/layouts }}
```

```
      <div class="ez-wr">
    <div class="ez-box" id="header">

      {{try:}}{{=auth.navbar(action=URL(request.application,'default','user'))}}{{
           except:pass}}
          <h1>
            <a href="">{{=response.title or 'response.title'}}</a>
          </h1>
          <h2>
        {{=response.subtitle or 'response.subtitle'}}
      </h2>
    </div>
    <div class="ez-box" id="statusbar">

      {{###### superfish menu }}
      {{=MENU(response.menu,_class='sf-menu')}}

          <script>
        jQuery(document).ready(function(){
              jQuery('ul.sf-menu').superfish({delay:400});});
      </script>
    </div>
    <div class="ez-wr">
      <div class="ez-fl ez-negmx">
            <div class="ez-box" id="left_sidebar">{{###### unused space}}</div>
      </div>
      <div class="ez-fl ez-negmr">
            <div class="ez-box" id="content">{{include}}</div>
      </div>
      <div class="ez-last ez-oh">
            <div class="ez-box" id="right_sidebar">{{###### unused space}}</div>
      </div>
    </div>
    <div class="ez-box" id="footer">
          {{=T('Copyright')}} &#169; 2010 -
          {{=T('Powered by')}} <a href="http://www.web2py.com">web2py</a>
    </div>
      </div>
    </div>
  </body>
</html>
```

Alcune caratteristiche di questo layout di default lo rendono molto semplice da usare e da personalizzare:

- `{{# ... }}` sono commenti speciali che non appariranno nel corpo della pagina HTML.
- Sono visualizzati sia `response.title` che `response.subtitle` che possono essere definiti nel modello. Se non sono stati impostati il titolo è il nome dell'applicazione.
- Il file `web2py_ajax.html` è incluso nell'header.
- Due file sono esplicitamente richiesti: "base.css" and "superfish.js". Il primo contiene il CSS completo per la pagina, è personalizzabile e molto ben documentato. Il secondo contiene il codice Javascript per il menu di default.
- `{{=auth.navbar( ... )}}` visualizza il messaggio di saluto per l'utente corrente insieme con i link alle funzioni di login, logout, registrazione, cambio della password, ecc. a seconda del contesto. E' posizionato all'interno di una `{{try:}} ... {{except:pass}}` per il caso in cui `auth` non sia definito.
- `{{=MENU(response.menu)` visualizza la struttura di menu come una lista di tipo `<ul> ... </ul>`.
- E' presente uno script per attivare il menu a cascata *Superfish* che può essere rimosso se non necessario.
- `{{include}}` è sostituito con il contenuto della vista indicata quando la pagina è visualizzata.
- Di default è usato un layout di pagina a tre colonne, con i seguenti identificativi di `DIV`: "header", "left_sidebar", "content", "right_sidebar" e "footer" sebbene il CSS "base.css" imposti l'altezza delle sidebars a zero.
- E' utilizzata la convenzione "ez.css" per i nomi del layout CSS definita in ref (51). In particolare è utilizzato il Layout numero 3. Un "ez.css" minimale è incluso in "base.css".

4.27.2 Personalizzare il layout di default

La personalizzazione del layout di default è molto semplice e ben documentata nel file "static/base.css" che è organizzato nelle seguenti sezioni:

- ez.css
- reimposta i flag comuni
- sceglie i font di default
- sceglie lo stile dei link
- aggiunge la linea dei pulsanti alle righe delle tabelle
- imposta alcune etichette in grassetto e alcune centrate nella pagina
- imposta tutti i campi di input alla stessa dimensione
- aggiunge la corretta separazione tra i tag h1-h6 e il testo
- indenta sempre la prima riga e aggiunge spazio sotto i paragrafi
- indenta le liste numerate e non numerate
- allinea i form e le tabelle
- blocchi di codice
- aggiunge spazio a sinistra e a destra del testo quotato
- allineamento della pagina, larghezza e padding (da cambiare per gli spazi)
- larghezza delle colonne (da cambiare per usare `left_sidebar` e `right_sidebar`)
- immagini e colori di sfondo (da cambiare per i colori)
- stile dei menu (per superfish.js)
- attività specifiche di web2py (.flash,.error)

Per cambiare la larghezza di left_sidebar, content, right_sidebar basta modificare la relativa sezione in "base.css":

```
/********** column widths **********/
#left_sidebar { width: 0px; }
#content { width: 840px; }
#right_sidebar { width: 0px; }
```

Per cambiare i colori e le immagini di background basta modificare la seguente sezione:

```
/********** backrgound images and colors **********/
body { background: url('images/background.png') repeat-x #3A3A3A; }

a { color: #349C01; }
.auth_navbar {
   top: 0px;
   float: right;
   padding: 3px 10px 3px 10px;
   font-size: 0.9em;
}
code { color: green; background: black; }
input:focus, textarea:focus { background: #ccffcc; }
#layout { background: white; }
#header, #footer { color: white; background: url('images/header.png') repeat
     #111111;}
#header h1 { color: #349C01; }
#header h2 { color: white; font-style: italic; font-size: 14px;}
#statusbar { background: #333333; border-bottom: 5px #349C01 solid; }
#statusbar a { color: white; }
#footer { border-top: 5px #349C01 solid; }
```

Il menu è costruito senza tener conto dei colori, ma può essere cambiato.

Ovviamente i file "layout.html" e "base.css" possono essere completamente sostituiti.

4.28 Funzioni nelle viste

Con questo "layout.html":

```
<html>
  <body>
    {{include}} <!-- must come before the two blocks below -->
    <div class="sidebar">
      {{if 'mysidebar' in globals():}}{{mysidebar()}}{{else:}}
        my default sidebar
      {{pass}}
  </body>
</html>
```

e con questa vista che lo estende:

```
{{def mysidebar():}}
my new sidebar!!!
{{return}}
{{extend 'layout.html'}}
Hello World!!!
```

la funzione mysidebar() è definita prima di `{{extend ... }}` ed è inclusa nella vista estesa senza il prefisso =.

Il codice generato è il seguente:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      {{block mysidebar}}
        my new sidebar!!!
      {{end}}
  </body>
</html>
```

Notare che le funzioni sono definite in HTML (sebbene possano contenere anche codice Python) e che `response.write` è usato per scrivere il loro contenuto (le funzioni non ritornano nessun contenuto). Questo è il motivo per cui la funzione deve essere chiamata utilizzando `{{mysidebar()}}` piuttosto che `{{=mysidebar()}}`. Le funzioni definite in questo modo possono anche avere argomenti.

4.29 Blocchi nelle viste

Un altro modo di rendere una vista più modulare è quello di utilizzare `{{block ... }}`. Questo meccanismo è un'alternativa a quello descritto nella sezione precedente.

Con il file "layout.html":

```
<html>
  <body>
```

```
    {{include}} <!-- must come before the two blocks below -->
    <div class="sidebar">
      {{block mysidebar}}
        my default sidebar
      {{end}}
  </body>
</html>
```

e questa vista che lo estende:

```
{{extend 'layout.html'}}
Hello World!!!
{{block mysidebar}}
my new sidebar!!!
{{end}}
```

il codice generato è il seguente:

```
<html>
  <body>
    Hello World!!!
    <div class="sidebar">
      {{block mysidebar}}
        my new sidebar!!!
      {{end}}
  </body>
</html>
```

Si possono definire molti blocchi e se un blocco è presente nella vista estesa ma non in quella che lo estende è utilizzato il contenuto della vista estesa.

4.30 Utilizzare i template per inviare email

E' possibile utilizzare il sistema dei template pe generare delle email. Per esempio, con la seguente tabella:

```
db.define_table('person', Field('name'))
```

se si vuole inviare ad ogni persona del database il seguente messaggio, memorizzato nella vista "message.html":

```
Dear {{=person.name}},
You have won the second prize, a set of steak knives.
```

basta eseguire il seguente codice:

```
>>> 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)
```

dove la maggior parte del lavoro è fatto nella linea:

```
response.render('message.html', context)
```

che visualizza la vista "file.html" con le variabili definite nel dizionario "context", e ritorna una stringa con il testo della email. "context" è un dizionario che contiene le variabili che saranno visibili al file di template.

Se il messaggio inizia con `<html>` e termina con `</html>` l'email sarà inviata in formato HTML.

Lo stesso meccanismo usato per generare il testo dell'email può essere anche utilizzato per generare SMS o altri tipi di messaggi basati su template.

5

Lo strato di astrazione del database

5.1 Dipendenze

web2py dispone di uno strato di astrazione del database (Database Abstraction Layer, DAL), una API che trasforma oggetti Python in oggetti del database come query, tabelle e record. IL DAL genera dinamicamente il codice SQL in tempo reale utilizzando il dialetto SQL specifico del database indicato dall'adattatore utilizzato. In questo modo non è necessario scrivere codice SQL o imparare dialetti SQL differenti (il termine SQL è qui utilizzato in modo generico) e l'applicazione sarà facilmente portabile tra differenti tipi di database. Attualmente i database supportati sono SQLite (che è incluso in Python e quindi in web2py), PostgreSQL, MySQL, Oracle, MSSQL, FireBird, DB2, Informi, Ingres e, solo parzialmente, Google App Engine (GAE). GAE è trattato più approfonditamente nel capitolo 11.

La distribuzione binaria per Windows è già configurata per utilizzare immediatamente SQLite e MySQL. La distribuzione binaria per Mac OS X è configurata per SQLite. Per utilizzare un altro database, è necessario utilizzare la distribuzione sorgente ed installare i driver appropriati per il database richiesto.

Una volta che il driver corretto è installato basta lanciare web2py dal sorgente ed il driver sarà disponibile. Ecco una lista dei driver supportati:

database	driver (source)
SQLite	sqlite3 o pysqlite2 o zxJDBC (56) (in Jython)
PostgreSQL	psycopg2 (57) o zxJDBC (56) (in Jython)
MySQL	MySQLdb (58)
Oracle	cx_Oracle (59)
MSSQL	pyodbc (60)
FireBird	kinterbasdb (61)
DB2	pyodbc (60)
Informix	informixdb (62)
Ingres	ingresdbi (12)

Il DAL in web2py è composto dalle seguenti classi:

DAL rappresenta una connessione al database. Per esempio:

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

Table rappresenta una tabella del database. Non è necessario istanziare Table direttamente, ma le tabelle vengono instanziate con la funzione `define_table` del DAL:

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

I metodi più importanti di Table sono:

`.insert`, `.truncate`, `.drop` e `.import_from_csv_file`.

Field rappresenta un campo di un database. Può essere istanziato e passato come argomento a `define_table`.

Rows è l'oggetto ritornato da una selezione in un database. Può essere visto come una lista di `Row`:

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

Row rappresenta una riga contenente i valori dei campi:

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

Query è un oggetto che rappresenta la clausola "where" di SQL:

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

Set è un oggetto che rappresenta un insieme di record. I suoi metodi principali sono: `count`, `select`, `update` e `delete`. Per esempio:

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

Expression è un oggetto che rappresenta una espressione utilizzata in `orderby` o `groupby`. La classe Field è derivata da Expression. Ecco un esempio:

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

5.2 *Stringhe di connessione*

Una connessione con il database è definita creando un'istanza dell'oggetto DAL:

```
>>> db = DAL('sqlite://storage.db', pool_size=0)
```

`db` non è una parola chiave; è una variabile locale che memorizza l'oggetto per la connessione `DAL` e può avere un nome qualsiasi. Il costruttore di `DAL` richiede un argomento obbligatorio, la stringa di connessione. La stringa di connessione è l'unica parte di codice di web2py che è specifica per il database utilizzato. Ecco degli esempi di stringhe di connessione per i database supportati (si presuppone che il database "test" sia installato localmente sulla sua porta di default):

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`
Ingres	`ingres://username:password@localhost/test`
DB2	`db2://username:password@test`
Informix	`informix://username:password@test`
Google App Engine	`gae`

In SQLite il database consiste di un unico file (che verrà creato se non esistente). Il file è bloccato ogni volta che è acceduto. Nel caso di MySQL, PostgreSQL, MSSQL, FireBird, Oracle, DB2 ed Informix il database "test" deve essere creato esternamente a web2py. Una volta che la connessione è stabilita web2py creerà, modificherà e cancellerà le tabelle nel modo appropriato.

E' anche possibile impostare la stringa di connessione a `None`. In questo caso il DAL non si connetterà a nessun database, ma le API saranno disponibili per il test. Esempi di questa tecnica saranno discussi nel capitolo 7.

5.2.1 Raggruppamento delle connessioni (Pooling)

Il secondo argomento del costruttore DAL è `pool_size` e ha come default 0.

Poichè stabilire una nuova connessione per ogni richiesto richiede del tempo web2py implementa un meccanismo di pooling delle connessioni. Quando una connessione è stabilita dopo che la pagina è stata servita e la transazione è stata completata la connessione non viene chiusa, ma rimane in un gruppo di connessioni disponibili (*pool*). Quando arriva la successiva richiesta HTTP web2py tenta di recuperare una connessione dal gruppo di quelle disponibili e di utilizzarla per la nuova transazione. Solo se non ci sono connessioni disponibili nel pool ne viene creata una nuova.

Il parametro `pool_size` è ignorato per SQLite (dove non avrebbe nessun ben-

eficio) e per GAE.

Le connessioni del pool sono condivise sequenzialmente tra i thread, nel senso he possono essere utilizzate da due differenti thread ma non simultaneamente. Esiste un unico pool per ogni processo di web2py.

Quando web2py si avvia il pool è sempre vuoto. Il pool cresce al valore minimo tra `pool_size` ed il numero massimo di richieste concorrenti. Questo significa che se `pool_size=10` ma il server non ha ricevuto più di 5 richieste concorrenti allora l'effettiva dimensione del pool sarà di 5. Con `pool_size=0` il pooling delle connessioni è disattivato.

5.2.2 *Errori di connessione*

Se web2py non riesce a connettersi al database aspetta un secondo e tenta di nuovo fino a 5 volte prima di segnalare un problema. Nel caso del pooling delle connessioni è possibile che una connessione rimanga aperta ed inutilizzata in web2py e che il database la chiuda. Grazie ai tentativi ripetuti da parte di web2py queste connessioni sono riaperte senza segnalare nessun errore.

5.2.3 *Database replicati*

Il primo argomanto di `DAL( ... )` può essere una lista di URI. In questo caso web2py tenta di connettersi ad una di loro. Lo scopo prncipale di questo meccanismo è di gestire server di database multipli e distribuire il carico di lavoro. Ecco un tipico caso d'uso:

```
db=DAL(['mysql://...1','mysql://...2','mysql://...3'])
```

In questo caso il DAL tenta di connettersi al primo database e, se non vi riesce, tenterà il secondo e poi il terzo. In questo modo è anche possibile

distribuire il carico in una configurazione di database master-slave. Maggiori dettagli saranno forniti nel capitolo 11.

5.3 *Parole chiave riservate*

C'è un ulteriore argomento che può essere passato al costruttore del DAL per controllare i nomi delle tabelle e delle colonne per evitare che vengano utilizzate parole chiave riservate del database. Questo argomento è `check_reserved` e per default è `None`. Questo argomento contiene una lista di stringhe che corrispondono ai nomi degli adattatori di database che si intende utilizzare.

Il nome dell'adattatore è lo stesso nome utilizzato nella stringa di connessione. Così se, per esempio, si vogliono controllare le parole chiave riservate in PostgreSQL e MSSQL la stringa di connessione potrebbe essere:

```
db = DAL('sqlite://storage.db',
         check_reserved=['postgres', 'mssql'])
```

Il DAL controllerà le parole chiave nello stesso ordine della lista.

Ci sono due opzioni extra: "all" e "common". Se si specifica "all" verranno controllate tutte le parole chiave SQL conosciute da web2py. Se si specifica "common" verranno controllate solo le parole chiave più comuni di SQL come `SELECT`, `INSERT`, `UPDATE`, ecc.

Per i database supportati si può specificare di voler controllare anche parole chiave SQL non riservate, in questo caso si deve aggiungere `_nonreserved` al nome dell'adattatore come, per esempio:

```
check_reserved=['postgres', 'postgres_nonreserved']
```

I seguenti adattattori di database supportano il controllo delle parole chiave riservate:

PostgreSQL	`postgres(_nonreserved)`
MySQL	`mysql`
FireBird	`firebird(_nonreserved)`
MSSQL	`mssql`
Oracle	`oracle`

5.4 DAL, tabelle e campi

Il modo migliore di comprendere le API del DAL è di provarne le diverse funzioni. Questo può essere fatto interattivamente grazie allo shell di web2py anche se il codice del DAL è utilizzato dai modelli e dai controller.

La prima operazione da fare è creare la connessione. Per semplicità si può usare SQLite, ma questi esempi sono validi con qualsiasi adattatore di database.

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

Il database è ora collegato e la connessione è memorizzata nella variabile globale `db`.

La stringa di connessione può essere recuperata con:

```
>>> print db._uri
sqlite://storage.db
```

ed il nome del database con:

```
>>> print db._dbname
sqlite
```

La stringa di connessione è chiamata `_uri` perchè è un'istanza di un identificatore uniforme di risorsa (*Uniform Resource Identifier*).

Il DAL consente connessioni multiple con lo stesso database o con database differenti, anche di diverso tipo. Negli esempi che seguono si assumerà la presenza di un singolo database, visto che questa è la condizione più comune.

Il metodo più importante di un DAL è `define_table`:

```
>>> db.define_table('person', Field('name'))
```

che definisce, memorizza e ritorna un oggetto `Table` chiamato "person" che contiene un campo (colonna) chiamato "name". Questo oggetto è accessibile con `db.person`, quindi non è necessario memorizzare il valore di ritorno della funzione.

5.5 *Rappresentazione dei record*

E' raccomandato, anche se non è obbligatorio, specificare una rappresentazione del formato dei campi del record:

```
>>> db.define_table('person', Field('name'), format='%(name)s')
```

oppure:

```
>>> db.define_table('person', Field('name'), format='%(name)s %(id)s')
```

o, in modo più complesso, utilizzando una funzione:

```
>>> db.define_table('person', Field('name'),
        format=lambda r: r.name or 'anonymous')
```

L'attributo `format` è utilizzato per due scopi distinti:

- Per rappresentare i record referenziati nei menu a discesa (select/option).
- Per impostare l'attributo `db.othertable.person.represent` per tutti i campi campi che referenziano questa tabella. Questo significa che SQLTABLE non mostrerà riferimenti per id ma utilizzerà invece il fomato preferito per la rappresentazione.

Questi sono i valori di default per il costruttore di `Field`:

```
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, compute=None, uploadseparate=None)
```

Non tutti gli argomenti sono pertinenti per tutti i tipi di campo. "length" è utilizzato solo per i campi di tipo "string", "password", "upload". "uploadfield" ed "authorize" sono utilizzati solo per i campi di tipo "upload". "ondelete" è rilevante solo per i campi di tipo "reference" ed "upload".

- `length` imposta la lunghezza massima di un campo di tipo "string", "password" o "upload". Se `length` non è specificato verrà utilizzato un valore di default che però non è garantito essere retro-compatibile. **Per evitare migrazioni (vedere la relativa sezione in questo capitolo) non volute durante gli upgrade è importante specificare un valore la la lunghezza dei campi di tipo "string", "password" e "upload".**

- `default` imposta il valore di default del campo. Questo valore è utilizzato quando si esegue una INSERT se un valore non è indicato esplicitamente. E' anche utilizzato per pre-popolare i form costruiti dalla tabella utilizzando SQLFORM.

- `required` indica al DAL che nessun inserimento deve essere consentito nella tabella se il valore per questo campo non è esplicitamente indicato.

- `requires` è un validatore o una lista di validatori. Non è utilizzato direttamente dal DAL, ma è utilizzato da SQLFORM. I validatori di default per i diversi tipi di campo sono indicati nella seguente tabella:

field type	default field validators
string	IS_LENGTH(length)
blob	None
boolean	None
integer	IS_INT_IN_RANGE(-1e100, 1e100)
double	IS_FLOAT_IN_RANGE(-1e100, 1e100)
decimal(n,m)	IS_DECIMAL_IN_RANGE(-1e100, 1e100)
date	IS_DATE()
time	IS_TIME()
datetime	IS_DATETIME()
password	None
upload	None
reference	IS_IN_DB(db,referenced_id_field,referenced_table_format)

"Decimal" richiede e ritorna valori come oggetti `Decimal` definiti nel modulo `decimal` di Python. Poichè SQLite non gestisce oggetti di tipo `decimal` questo tipo di campo viene internamente convertito in un `double`. Gli argomenti (n, m) indicano il numero di cifre prima e dopo il punto decimale.

> *E' importante notare che* `requires = ...` *è controllato al livello del form,* `required=True` *è controllato al livello del DAL (nelle INSERT), mentre* `notnull`, `unique` *e* `ondelete` *sono oontrollati al livello del database. Sebbene possa sembrare ridondante è importante mantenere questa distinzione quando si programma con il DAL.*

- `ondelete` si traduce nella clausola SQL "ON DELETE". Di default "CASCADE" indica al database che quando un record è cancellato devono essere cancellati anche tutti i record che si riferiscono ad esso.
- `notnull=True` si traduce nella clausola SQL "NOT NULL" e richiede al database di prevenire l'inserimento di valori nulli nel campo.
- `unique=True` si traduce nella clausola SQL "UNIQUE" e richiede al database di garantire l'univocità del valore del campo nella tabella.
- `uploadfield` si applica solo a campi di tipo "upload". Un campo di tipo "upload" memorizza il nome di un file registrato da qualche altra parte (per default nella cartella "uploads/" dell'applicazione). Se `uploadfield` è impostato il file è memorizzato in un campo *blob* all'interno della stessa

tabella ed il valore di `uploadfield` è il nome del campo *blob*. Questo sarà discusso in maggior dettaglio nella sezione riguardante SQLFORM.

- `uploadseparate` se impostato a `True` indica a web2py di memorizzare i file in sottocartelle della cartella "upload". Questo server per ottimizzare l'accesso ed evitare che un grande numero di file sia memorizzato in un'unica cartella. ATTENZIONE: il valore di `uploadseparate` non può essere modificato da `True` a `False` senza corrompere il sistema di upload. web2py utilizza o non utilizza le sottocartelle separate. Se si cambia questo comportamento dopo che alcuni file sono stati caricati web2py non è più in grado di recuperarli. Se questo succede è possibile spostare i file e correggere il problema, ma questo non è descritto in questo manuale.
- `widget` deve essere uno degli oggetti *widget* disponibili, inclusi i widget personalizzati, per esempio `SQLFORM.widgets.string.widget`. Una lista dei widget disponibili sarà presentata in seguito. Ogni tipo di campo ha un widget di default.
- `label` è una stringa (o qualsiasi oggetto che può essere serializzata in una stringa) che contiene l'etichetta da usare per il campo nei form auto-generati.
- `comment` è Fededdduna stringa (o qualsiasi oggetto che può essere serializzata in una stringa) che contiene un commento associato al campo e che sarà visualizzato alla destra del campo di input nei form auto-generati.
- `writable` indica che, se impostato a `True` il campo può essere modificato nei form auto-generati di creazione e di aggiornamento.
- `readable` indica che, se impostato a `True` il campo sarà leggibile e quindi visibile nei form a sola lettura. Se un campo non è nè `writable` nè `readable` non sarà visualizzato nei form di creazione e di aggiornamento.
- `update` contiene il valore di default per il campo quando il record viene aggiornato.
- `compute` è uan funzione opzionale. Se un record è inserito e non è presente nessun valore per un campo che ha questo attributo impostato il valore viene calcolato passando il record (come un dizionario) alla funzione indicata.

- `authorize` può essere utilizzato per richiedere il controllo d'accesso sul corrispondente campo. E' valido solo per i campi di tipo "upload". Sarà discusso im maggior dettaglio nel capitolo riguardante l'autorizzazione e l'autenticazione.
- `autodelete` determina se il corrispondente file di upload debba essere cancellato quando il record che lo referenzia viene cancellato. E' valido solo per i campi di tipo "upload".
- `represent` può essere `None` o può puntare ad una funzione che prende il valore di un campo e ne ritorna una rappresentazione alternativa, come nei seguenti esempi:

```
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('download', args=value))
```

i campi *blob* sono speciali. Per default i dati binari sono codificati in *base64* prima di essere memorizzati nel campo del database e sono decodificati quando sono estratti. Questo ha l'effetto negativo di utilizzare circa il 25% in più di spazio ma ha due vantaggi: riduce la quantità di dati nella comunicazione tra web2py e il database e rende la comunicazione indipendente dalle particolari convenzioni di codifica del database.

Per conoscere le tabelle esistenti si può interrogare il database:

```
>>> print db.tables
['person']
```

Per conoscere i campi esistenti si può interrogare una tabella:

```
>>> print db.person.fields
['id', 'name']
```

Non si deve mai dichiarare un campo chiamato "id" perchè questo viene creato automaticamente da web2py. Infatti ogni tabella ha di default un campo chiamato "id", un campo intero auto-incrementato (che parte da 1) utilizzato come valore di riferimento del record e che rende ogni record unico. "id" è la

chiave primaria della tabella (il fatto che l'auto-incremento inizi da 1 è legato al database e, per esempio, non si applica a GAE).

Opzionalmente è possibile definire un campo di tipo `type='id'` che verrà utilizzato da web2py come campo "id" auto-incrementato. Questo non è raccomandato a meno di non accedere a tabelle di database pre-esistenti. E' anche possibile, con alcune limitazioni, utilizzare chiavi primarie differenti come discusso nella sezione che si occupa dei database pre-esistenti.

Si può interrogare il tipo di una tabella:

```
>>> print type(db.person)
<class 'gluon.sql.Table'>
```

oppure con:

```
>>> print type(db['person'])
<class 'gluon.sql.Table'>
```

Allo stesso modo di spuo accedere al tipo di un campo in diversi modi equivalenti:

```
>>> 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'>
```

Con un campo si può accedere agli attributi impostati nella sua definizione:

```
>>> print db.person.name.type
string
>>> print db.person.name.unique
False
>>> print db.person.name.notnull
False
>>> print db.person.name.length
32
```

inclusa la tabella a cui appartiene, il nome della tabella e la relativa connessione:

```
>>> db.person.name._table == db.person
True
>>> db.person.name._tablename == 'person'
True
>>> db.person.name._db == db
True
```

5.6 Migrazioni

La funzione `define_table` controlla se la tabella esiste. In caso negativo genera il codice SQL necessario per la sua creazione e lo esegue. Se la tabella esiste ma non corrisponde a quella definita viene generato ed eseguito il codice SQL necessario alla sua modifica. Se un campo ha cambiato tipo ma non nome `define_table` tenterà di convertire il dato (se questo non è il comportamento voluto sarà necessario ridefinire la tabella due volte: la prima volta facendo rimuovere il campo a web2py; la seconda volta aggiungendo il nuovo campo alla definizione della tabella in modo che web2py possa crearlo). Se la tabella esiste e corrisponde alla definizione corrente, non subirà variazioni. In ogni caso viene istanziato l'oggetto `db.person` che rappresenta la tabella.

Questo comportamento è definito "migrazione". web2py registra tutte i tentativi di migrazione e le migrazioni completate nel file "databases/sql.log".

Il primo argomento di `define_table` è sempre il nome della tabella. L'altro argomento senza nome sono i campi. La funzione ha anche un argomento opzionale con nome chiamato "migrate":

```
>>> db.define_table('person', Field('name'), migrate='person.table')
```

Il valore di "migrate" è il nome del file (presente nella cartella "databases" dell'applicazione) dove web2py memorizza le informazioni interne della migrazione per questa tabella. Questi file sono molto importanti e non devono mai essere rimossi tranne quando l'intero database viene eliminato. In questo caso i file ".table" devono essere rimossi manualmente. Per default "migrate" è impostato a `True`. Questo fa sì che web2py generi il nome del file da un hash

della stringa di connessione. Se migrate è impostato a `False` la migrazione non viene eseguita e web2py assume che la tabella esista nel database e che contenga (almeno) i campi definiti in `define_table`. La pratica migliore è dare un nome esplicito per il file di migrazione.

Non possono esistere due tabelle nella stessa applicazione con lo stesso nome di file di migrazione.

5.7 Correggere le migrazioni errate

CI sono due tipici problemi con le migrazioni e ci sono diversi modi di risolverli.

Un primo problema è specifico a SQLite. SQLite non verifica i tipi delle colonne e non può eliminare le singole colonne. Questo significa che se si rimuove una colonna di un certo tipo (per esempio stringa) questa non è realmente rimossa. Se si aggiunge la stessa colonna con un tipo differente (per esempio datetime) si ottiene una colonna (di tipo datetime) che contiene valori errati (in questo caso delle stringhe ed è quindi inutilizzabile). web2py non segnala nessun errore in questo caso perchè non è in grado di leggere cosa contiene il database fino a che non tenta di recuperare i dati e genera un errore.

Se web2py ritorna un errore nella funzione `gluon.sql.parse` questo è il problema: dati corrotti in una tabella per il motivo appena indicato.

La soluzione consiste nell'aggiornare tutti i record della tabella sostituendo con `None` i valori nella colonna.

Il secondo problema è più generico ma è tipico di MySQL. MySQL non consente più di un comando ALTER TABLE in una transazione. Questo significa che web2py deve suddividere transazioni complesse in transazioni più piccole (un ALTER TABLE per volta) ed eseguirle sequenzialmente. E' perciò

possibile che alcune parti di una transazione complessa vengano eseguite e una parte fallisca lasciando web2py in uno stato corrotto. Perchè dovrebbe fallire una parte di una transazione? Perchè per esempio web2py potrebbe tentare di trasformare una colonna di tipo stringa in una di tipo datetime e non riuscire nella conversione dei dati. In questo caso web2py non è in grado di sapere con certezza qual è la struttura della tabella effettivamente memorizzata nel database.

La soluzione consiste nel disabilitare le migrazioni per tutte le tabelle ed abilitare le migrazioni fittizie:

```
db.define_table(....,migrate=False,fake_migrate=True)
```

Questo comando ricostruisce i metadata di web2py per la tabella secondo la definizione corrente. Tentare le diverse definizioni delle tabelle per verificare quale funziona (quella prima della migrazione fallita e quella dopo la migrazione fallita). Quando l'operazione va a buon fine si può rimuovere l'attributo `fake_migrate=True`.

Prima di tentare di correggere questi problemi di migrazione è opportuno eseguire una copia di sicurezza dei file "applications/yourapp/databases/*.table".

5.8 insert

E' possibile inserire un record in una tabella con:

```
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
```

la funzione insert ritorna un "id" univoco per ogni record inserito.

E' possibile troncare una tabella, cioè rimuovere tutti i record e reimpostare il contatore dell'id.

```
>>> db.person.truncate()
```

Ora, se si inserisce un nuovo record il contatore degli id ricomicia da 1 (anche se questo comportamento è specifico del database utilizzato e non si applica per GAE):

```
>>> db.person.insert(name="Alex")
1
```

web2py dispone anche di un metodo `bulk_insert` per inserimento massivo:

```
>>> db.person.bulk_insert([{'name':'Alex'}, {'name':'John'}, {'name':'Tim'}])
[3,4,5]
```

che richiede una lista di dizionari di campi da inserire ed esegue inserimenti multipli tutti insieme e ritorna gli ID dei record inseriti. Sui database relazionali supportati da web2py non c'è nessun vantaggio nell'utilizzare questa funzione rispetto a ciclare sugli inserimenti individuali, ma su GAE si ottiene un notevole incremento della velocità.

5.9 *Completamento (commit) e annullamento (rollback) delle transazioni*

Nessuna operazione di creazione, rimozione, inserimento, troncamento, cancellazione o aggiornamento è effettivamente eseguita finchè non viene dato il comando `commit`:

```
>>> db.commit()
```

Per verificare, dopo l'inserimento di un nuovo record:

```
>>> db.person.insert(name="Bob")
2
```

ed eseguendo un annullamento (rollback) della transazione, per ignorare tutte le operazioni eseguite dall'ultimo commit:

```
>>> db.rollback()
```

se ora si inserisce nuovamente un record il contatore sarà ancora impostato a 2 poichè il precedente inserimento è stato annullato.

```
>>> db.person.insert(name="Bob")
2
```

Il codice dei modelli, delle viste e dei controller è racchiuso da codice web2py del seguente tipo:

```
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
```

Non c'è necessità di eseguire esplicitamente le operazioni di `commit` o `rollback` in web2py a meno che non si voglia un controllo maggiore delle transazioni.

5.10 Codice SQL esplicito

5.10.1 executesql

Il DAL permette di eseguire esplicitamente del codice SQL:

```
>>> print db.executesql('SELECT * FROM person;')
[(1, u'Massimo'), (2, u'Massimo')]
```

In questo caso i valori di ritorno non sono analizzati o trasformati dal DAL e il formato dipende dallo specifico database. Questo utilizzo non è normalmente necessario per le SELECT, ma è comune con gli indici.

`executesql` ha due argomenti opzionali: `placeholders` e `as_dict`. `placeholders` è una sequenza opzionale di valori che devono essere sostituiti o, se supportato dall'adattatore del database, un dizionario con chiavi corrispondenti ai segnaposto presenti nel codice SQL. Se `as_dict` è impostato a `True` il risultato ritornato dall'adattatore sarà convertito in una sequenza di dizionari con le chiavi composte dai nomi dei campi della tabella. I risultati ritornati con `as_dict = True` sono gli stessi di quelli ritornati applicando la funzione **.to_list()** ad una normale select.

```
[{field1: value1, field2: value2}, {field1: value1b, field2: value2b}]
```

5.10.2 _lastsql

Sia nel caso di SQL generato manualmente (con `executesql`) o di codice SQL generato automaticamente dal DAL è sempre possibile ottenere il codice SQL utilizzato in `db._lastsql`. Questo può essere utile per il debug.

```
>>> rows = db().select(db.person.ALL)
>>> print db._lastsql
SELECT person.id, person.name FROM person;
```

> *web2py non genera mai query che utilizzano l'operatore "*", ma è sempre esplicito indicando nella i nomi dei campi nella SELECT.*

5.11 drop

Le tabelle possono essere eliminate (e tutti i dati andranno persi) con:

```
>>> db.person.drop()
```

5.12 Indici

Attualmente le API del DAL non forniscono comandi per creare gli indici delle tabelle ma queste devono essere create utilizzando un comando SQL con `executesql`. Questo perchè l'esistenza degli indici rende complesse le migrazioni ed è quindi meglio gestirli esplicitamente. Gli indici possono essere necessari per i campi che sono utilizzati spesso nelle query.

Ecco un esempio di come creare un indice in SQLite:

```
>>> db = DAL('sqlite://storage.db')
>>> db.define_table('person', Field('name'))
>>> db.executesql('CREATE INDEX IF NOT EXISTS myidx ON person name;)
```

Altri dialetti SQL possono avere sintassi simili ma potrebbero non supportare la direttiva opzionale `IF NOT EXISTS`.

5.13 Database pre-esistenti e tabelle con chiave

web2py può collegarsi a database pre-esistenti ad alcune condizioni. Il modo più semplice è quando le seguenti condizioni sono rispettate:

- Ogni tabella deve avere un campo intero auto-incrementato chiamato "id".
- I record devono essere referenziati esclusivamente utilizzanto il campo "id".

Quando si accede ad una tabella pre-esistente, non creata da web2py nell'applicazione corrente, va sempre impostato `migrate=False`.

Se la tabella pre-esistente ha un campo intero di auto-incremento che non è chiamato "id" web2py è in grado di accedervi ma la definizione della tabella

deve contenere esplicitamente un campo di tipo "id" (`Field(' ... ', 'id')` dove... è il nome del campo intero di auto-incremento.

Inoltre se la tabella pre-esistente utilizza una chiave primaria che non è un campo di auto-incremento è ancora possibile utilizzare una "tabella con chiave". Per esempio:

```
db.define_table('account',
    Field('accnum','integer'),
    Field('acctype'),
    Field('accdesc'),
    primarykey=['accnum','acctype'],
    migrate=False)
```

In questo esempio l'attributo `primarikey` è una lista di campi che compongono la chiave primaria della tabella. Al momento della scrittura di questo manuale non è garantito che l'attributo `primarykey` funzioni con ogni tabelle pre-esistente e con ogni adattatore di database supportato. Per semplicità è bene, se possibile, creare una vista del database che contiene un campo "id" intero di auto-incremento.

5.14 Transazioni distribuite

Questa caratteristica è supportata solo da PostgreSQL perchè fornisce una API per un commit in due fasi.

Assumento di avere due (o più) connessioni a database PostgreSQL distinti, per esempio:

```
db_a = DAL('postgres://...')
db_b = DAL('postgres://...')
```

nel modello o nel controller, è possibile eseguire la commit contemporanea con:

```
DAL.distributed_transaction_commit(db_a, db_b)
```

Se non eseguita con successo questa funzione esegue un rollback e genera una `Exception`.

Nei controller, quando un'azione finisce, se si hanno due connessioni distinte e non viene chiamata `distributed_transaction_commit` web2py esegue il loro commit separatamente. Questo significa che c'è una possibilità che uno dei commit vada a buon fine e l'altro no. Le transazioni distribuite evitano che questo possa accadere.

5.15 Upload manuali

Nel seguente modello:

```
db.define_table('myfile',Field('image','upload'))
```

Normalmente un INSERT è gestito automaticamente con un SQLFORM o un form di tipo Crud (che è sempre un SQLFORM) ma a volta il file è già presente nel filesystem e si vuole caricarlo da programma. Questo può essere fatto con:

```
stream = open(filename,'rb')
db.myfile.insert(image=db.myfile.image.store(stream,filename))
```

Il metodo `store` di un oggetto di un campo di tipo "upload" ha come argomenti uno stream di un file ed un nome di file ed esegue le seguenti operazioni: utilizza il nome del file per determinare il tipo (dall'estensione) del file; crea un nuovo nome temporaneo per il file (secondo il meccanismo di upload di web2py); carica il contenuto del file nel nuovo file temporaneo (nella cartella "upload" dell'applicazione, a meno che non sia specificato diversamente); ritorna il nuovo nome temporaneo del file che sarà memorizzato nel campo `image` della tabella `db.myfile`.

5.16 *Query, Set, Rows*

Nella tabella definita (e cancellata) precedentemente si inseriscano tre record:

```
>>> db.define_table('person', Field('name'))
>>> db.person.insert(name="Alex")
1
>>> db.person.insert(name="Bob")
2
>>> db.person.insert(name="Carl")
3
```

La tabella può essere memorizzata in una variabile, per esempio la variabile `person`:

```
>>> person = db.person
```

Si può anche memorizzare un campo in una variabile come, per esempio, `name`:

```
>>> name = person.name
```

Si può anche costruire una query (utilizzando operatori come ==, !=, <, >, <=, >=, like, belongs) e memorizzare una query in una variabile `q`:

```
>>> q = name=='Alex'
```

Quando si richiama `db` con una query si definisce un *set* di record che possono essere memorizzati in una variabile, per esempio `s`:

```
>>> s = db(q)
```

E' da notare che nessuna query al database è stata ancora eseguita. DAL e query semplicemente definiscono un set di record in questo db che soddisfano la query. web2py determina dalla query quale tabella (o tabelle) è coinvolta. In effetti non è necessario specificarle.

5.17 *select*

Con un set, per esempio `s`, è possibile recuperare i record con il comando `select`:

```
>>> rows = s.select()
```

che ritorna un oggetto iterabile di classe `gluon.sql.Rows` i cui elementi sono di classe `gluon.sql.Row`. Gli oggetti row si comportano come dizionari, ma i loro elementi possono anche essere acceduti come attributi, come `gluon.storage.Storage`. Il primo differisce dal secondo perchè i suoi valori sono in sola lettura.

L'oggetto *rows* consente di eseguire un ciclo sul risultato della SELECT e di stampare i valori dei campi selezionati per ogni riga

```
>>> for row in rows:
        print row.id, row.name
1 Alex
```

Tutti i passi possono essere eseguiti in un unico comando:

```
>>> for row in db(db.person.name=='Alex').select():
        print row.name
Alex
```

Il comando `select` può avere degli argomenti. Tutti gli argomenti senza nome sono interpretati come i nomi dei campi che si vogliono recuperare. Per esempio si può esplicitamente richiedere i campi "id" e "name":

```
>>> for row in db().select(db.person.id, db.person.name):
        print row.name
Alex
Bob
Carl
```

L'attributo di tabella `ALL` consente di specificare tutti i campi:

```
>>> for row in db().select(db.person.ALL):
        print row.name
Alex
```

```
Bob
Carl
```

Notare che non è indicata nessuna stringa di query. web2py interpreta che se si richiedono tutti i campi di una tabella senza nessuna informazione aggiuntiva allora deve restituire tutti i record della tabella.

Una sintassi alternativa equivalente è:

```
>>> for row in db(db.person.id > 0).select():
        print row.name
Alex
Bob
Carl
```

e web2py intepreta questo come una richiesta per tutti i campi di tutti i record (id > 0) della tabella.

Un oggetto `Rows` è un contenitore per:

```
rows.colnames
rows.response
```

`colnames` è una lista dei nomi delle colonne ritornate dalla select. `response` è una lista di tuple che contiene la risposta non elaborata della select, prima che venga analizzata e convertita nel formato web2py corretto.

Mentre un oggetto rows non può essere serializzato nè con Pickle nè con XML-RPC `colnames` and `response` possono esserlo.

Ancora, molte di queste opzioni sono specifiche per l'adattatore di database utilizzato. In questo caso la selezione dei campi avviene in modo diverso su GAE.

5.17.1 Abbreviazioni

Il DAL dispone di diverse abbreviazioni per semplificare il codice. In particolare:

```
myrecord = db.mytable[id]
```

ritorna il record con l'`id` specificato, se esiste. Se non esiste ritorna `None`. L'esempio precedente è equivalente a:

```
myrecord = db(db.mytable.id==id).select().first()
```

```
del db.mytable[id]
```

è equivalente a:

```
db(db.mytable.id==id).delete()
```

e cancella il record con l'`id` specificato, se esiste.

```
db.mytable[0] = dict(myfield='somevalue')
```

è equivalente a:

```
db.mytable.insert(myfield='somevalue')
```

e crea un nuovo record con i valori dei campi specificati dal dizionario.nd side.

```
db.mytable[id] = dict(myfield='somevalue')
```

è equivalente a:

```
db(db.mytable.id==id).update(myfield='somevalue')
```

ed aggiorna un record esistente con i valori dei campi specificati nel dizionario.

5.17.2 *Recuperare un record*

Ancora un'altra utile sintassi è:

```
record = db.mytable(id)
record = db.mytable(db.mytable.id==id)
record = db.mytable(id,myfield='somevalue')
```

Apparentemente simile a `db.mytable[id]` questa sintassi è più flessibile e più sicura. Prima di tutto controlla che `id` sia un intero (o che `str(id)` sia un intero) e ritorna `None` se non lo è (ma non genera mai un'eccezione). Inoltre consente di specificare ulteriori condizioni che il record deve soddisfare. Se non sono soddisfatte ritorna `None`.

5.17.3 *Select ricorsive*

Con la tabella "person" precedentemente definita e con una nuova tabella "dog" che referenzia una persona:

```
>>> db.define_table('dog', Field('name'), Field('owner',db.owner))
```

con una semplice select si ottiene una lista di row:

```
>>> dogs = db(db.dog.id>0).select()
```

Per ogni row ottenuta da dog è possibile recuperare non solo i campi dal record dalla tabella seleazionata ma anche dalla tabella collegata (ricorsivamente):

```
>>> for dog in dogs: print dog.info, dog.owner.name
```

In questo caso `dog.owner.name` richiede una select nel database per ogni dog in dogs ed è per questo motivo inefficiente. Una soluzione migliore è eseguire una JOIN ogni volta che questo sia possibile invece delle select ricorsive. Questo metodo è invece conveniente e pratico quando si accedono singoli record.

E' anche possibile effettuare la ricerca nel verso opposto, selezionando i dog referenziati da una person:

```
person =  db.person(id)
for dog in person.dog.select(orderby=db.dog.name):
    print person.name, 'owns', dog.name
```

In quest'ultimo esempio `person.dog` è una scorciatoia per:

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

cioè l'insieme degli oggetti `dog` è referenziato dalla `person` corrente. Questa sintassi non funziona più se la tabella che referenzia ha riferimenti multipli alla tabella referenziata. In questo caso è necessario essere più espliciti ed utilizzare una query completa.

5.17.4 Serializzare gli oggetti row nelle viste

Il risultato di una select può essere visualizzato in una vista con la seguente sintassi:

```
{{extend 'layout.html'}}
<h1>Records</h2>
{{=db().select(db.person.ALL)}}
```

e viene automaticamente convertito in una tabella HTML con un header contenente i nomi della tabella ed una riga per ogni record. Le righe sono alternativamente marcate con una classe "even" (pari) e "odd" (dispari). Internamente la riga è prima convertita in un oggetto SQLTABLE (da non confondere con Table) e poi serializzata. I valori estratti dal database sono anche formattati dai validatori associati ai diversi campi e poi codificati (nota: utilizzare questo metodo per visualizzare i dati non è una pratica corretta nel paradigma MVC).

Inoltre è possibile, e a volte è conveniente, chiamare SQLTABLE esplicitamente. Il costruttore SQLTABLE ha i seguenti argomenti opzionali:

- `linkto`: la URL o l'azione da usare per i campi referenziati (il default è None)
- `upload`: la URL dell'azione di download per consentire lo scaricamento dei file caricati (il default è None)
- `headers`: un dizionario per collegare i nomi dei campi alle etichette da usare come intestazioni: (il default è `{}`). Può anche essere un'istruzione, attualmente è supportata `headers='fieldname:capitalize'`.
- `truncate`: il numero di caratteri per troncare i valori lunghi della tabella (il default è 16)
- `columns`: la lista dei nomi dei campi da mostrare come colonne. I campi non elencati non sono visualizzati (il default è tutti i campi)
- `attributes`: attributi ausiliari generici da passare all'oggetto TABLE più esterno.

Ecco un esempio:

```
{{extend 'layout.html'}}
<h1>Records</h2>
{{=SQLTABLE(db().select(db.person.ALL),
   headers='fieldname:capitalize',
   truncate=100,
   upload=URL('download'))
}}
```

5.17.5 orderby, groupby, limitby, distinct

Il comando `select` ha cinque argomenti opzionali: orderby, groupby, limitby, left e cache. Ora verranno descritti i primi tre:

I record possono essere recuperati ordinati per nome:

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name):
        print row.name
Alex
Bob
Carl
```

Per recuperare i record in ordine inverso (notare la tilde):

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name):
        print row.name
Carl
Bob
Alex
```

E' possibile ordinare i record per campi multipli concatenandoli con "|":

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name|db.person.id):
        print row.name
Carl
Bob
Alex
```

Utilizzando groupby insieme con orderby si possono raggruppare i record con lo stesso valore per il campo specificato (questo dipende dall'adattatore di database utilizzato e non è possibile su GAE):

```
>>> for row in db().select(db.person.ALL, orderby=db.person.name,
                           groupby=db.person.name):
        print row.name
Alex
Bob
Carl
```

Con l'argomento `distinct=True` si può specificare di voler solo record differenti. Questo ha lo stesso effetto di raggruppare i record utilizzando tutti i campi specificati tranne quelli che non richiedono l'ordinamento. Quando si usa DISTINCT è importante non selezionare tutti i campi e in particolar modo non selezionare il campo "id", altrimenti tutti i record saranno sempre differenti.

Ecco un esempio:

```
>>> for row in db().select(db.person.name, distinct=True):
        print row.name
Alex
Bob
Carl
```

Con limitby è possibile selezionare un sotto-insieme dei record (nell'esempio seguente solo i primi due, partendo da zero):

```
>>> for row in db().select(db.person.ALL, limitby=(0, 2)):
        print row.name
Alex
Bob
```

Attualmente "limitby" è supportato solo parzialmente da MSSQL in quanto il database Microsoft non dispone di un meccanismo per recuperare record che non partono da zero.

5.17.6 Operatori logici

Le query possono essere combinate utilizzando l'operatore binario logico AND ("&"):

```
>>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
4 Alex
```

o l'operatore binario logico OR ("|"):

```
>>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
1 Alex
```

Una query (o una sotto-query) può essere negata con l'operatore binario di negazione ("!="):

```
>>> rows = db((db.person.name!='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
2 Bob
3 Carl
```

o con l'operatore unario di negazione ("|") per una negazione esplicita:

```
>>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
>>> for row in rows: print row.id, row.name
```

```
2 Bob
3 Carl
```

A causa delle retrizioni di Python per l'overloading degli operatori AND e OR questi non possono essere direttamente utilizzati nelle query.

5.17.7 count, delete, update

I record in un set possono essere contati con:

```
>>> print db(db.person.id > 0).count()
3
```

I record in un set possono essere cancellati con:

```
>>> db(db.person.id > 3).delete()
```

Tutti i record in un set possono essere aggiornati passando argomenti con nome corrispondente al campo che deve essere aggiornato:

```
>>> db(db.person.id > 3).update(name='Ken')
```

5.17.8 Espressioni

Il valore assegnato ad un comando di aggiornamento può essere un'espressione. Per esempio, considerando questo modello:

```
>>> db.define_table('person',
        Field('name'),
        Field('visits', 'integer', default=0))
>>> db(db.person.name == 'Massimo').update(
        visits = db.person.visits + 1)
```

I valori utilizzati nella query possono essere anche espressioni:

```
>>> 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()
```

5.17.9 *update_record*

web2py inoltre consente di aggiornare un singolo record che è già in memoria utilizzando `update_record`:

```
>>> rows = db(db.person.id > 2).select()
>>> row = rows[0]
>>> row.update_record(name='Curt')
```

Questo non deve essere confuso con:

```
>>> row.update(name='Curt')
```

perchè per una singola riga il metodo `update` aggiorna l'oggetto della row ma nono il record nel database, come nel caso di `update_record`.

5.17.10 *first e last*

Con un oggetto Rows che contiene dei record

```
>>> rows = db(query).select()
```

i comandi

```
>>> first_row = rows.first()
>>> last_row = rows.last()
```

sono equivalenti a:

```
>>> first_row = rows[0] if len(rows)>0 else None
>>> last_row = rows[-1] if len(rows)>0 else None
```

5.17.11 *as_dict e as_list*

Un oggetto Row può essere serializzato in un normale dizionario usando il metodo `as_dict()` e un oggetto Rows può essere serializzato in una lista di dizionari usano metodo `as_list()`. Ecco alcuni esempi:

```
>>> rows = db(query).select()
>>> rows_list = rows.as_list()
>>> first_row_dict = rows.first().as_dict()
```

Questi metodi sono utili per passare oggetti Rows a delle viste generiche o per memorizzare gli oggetti Rows nelle sessioni (l'oggetto Rows non può essere memorizzato nella sessione perchè contiene un riferimento ad una connessione di database aperta):

```
>>> rows = db(query).select()
>>> session.rows = rows # not allowed!
>>> session.rows = rows.as_list() # allowed!
```

5.17.12 *find, exclude, sort*

A volte è necessario eseguire due SELECT con una di queste che contiene un sotto-insieme dell'altra. In questi casi non c'è nessun vantaggio nell'accedere nuovamente al database. I metodi `find`, `exclude` e `sort` consentono di manipolare un oggetto Rows e di generarne un altro senza accedere al database:

- `find` ritorna un set di oggetti Row filtrati da una condizione e lascia inalterato l'oggetto Rows originale.
- `exclude` ritorna un set di oggetti Row filtrati da una condizione e li rimuove dall'oggetto Rows originale.
- `sort` ritorna un set di oggetti Row ordinati in base ad una condizione e lascia l'oggetto Rows originale inalterato.

Tutti questi metodi hanno un singolo argomento, una funzione che agisce su ogni Row.

Ecco un esempio d'utilizzo:

```
>>> db.define_table('person',Field('name'))
>>> db.insert(name='John')
>>> db.insert(name='Max')
>>> db.insert(name='Alex')
>>> rows = db(db.pearon.id>0).select()
>>> for row in rows.find(lambda row: row.name[0]=='M'): print row.name
Max
>>> print len(rows)
3
>>> for row in rows.extract(lambda row: row.name[0]=='M'): print row.name
Max
>>> print len(rows)
2
>>> for row in rows.sort(lambda row: row.name): print row.name
Alex
John
```

Possono anche essere combinati insieme:

```
>>> rows = db(db.pearon.id>0).select()
>>> rows = rows.find(lambda row: 'x' in row.name).sort(lambda row: row.name)
>>> for row in rows: print row.name
Alex
Max
```

5.18 Campi calcolati

I campi del DAL possono avere un attributo `compute` che deve essere una funzione (o una lambda) che ha come argomento un oggetto Row e ritorna un valore per il campo. Quando un nuovo record è inserito (o aggiornato) se non è fornito un valore per il campo web2py tenta di calcolarlo dai valori degli altri campi utilizzando la funzione specificata in `compute`. Ecco un esempio:

```
>>> db.define_table('item',
        Field('unit_price','double'),
        Field('quantity','integer'),
        Field('total_price',compute=lambda r: r['unit_price']*r['quantity']))
>>> r = db.item.insert(unit_price=1.99, quantity=5)
>>> print r.total_price
9.95
```

E' da notare che il valore calcolato è memorizzato nel database e non è ricalcolato quando si recupera il record, a differenza di quello che avviene per i campi virtuali, descritti più sotto. Tipici utilizzi di un campo calcolato possono essere:

- nelle applicazioni wiki per memorizzare il testo inserito in una pagina wiki come html, per evitare di riprocessare ogni richiesta.
- nelle ricerche, per calcolare un valore normalizzato per un campo da utilizzare per la ricerca.

5.19 Campi virtuali

I campi virtuali sono anch'essi campi calcolati (come nella sezione precedente) ma differiscono da quelli perchè non sono memorizzati nel database e perciò sono ricalcolati ogni volta che un record è estratto dal database. Possono essere usati per semplificare il codice senza occupare ulteriore spazio nel database, ma non possono essere usati nelle ricerche.

Per definire uno o più campi virtuali deve essere definita una classe contenitore che deve essere istanziata e collegata ad una tabella o ad una select. Per esempio, considerando la seguente tabella:

```
>>> db.define_table('item',
        Field('unit_price','double'),
        Field('quantity','integer'),
```

E' possibile definire un campo virtuale "total_price" con:

```
>>> class MyVirtualFields:
        def total_price(self):
            return self.item.unit_price*self.item.quantity
>>> db.item.virtualfields.append(MyVirtualFields())
```

Notare che ogni metodo della classe che prende un singolo argomento (`self`) è un nuovo campo virtuale. `self` si riferisce a ciascuna riga della select. I

campi virtuali sono referenziati dal percorso completo come in `self.item.unit_price`. La tabella è collegata ai campi virtuali aggiungendo un'istanza della classe all'attributo `virtualfields` della tabella.

I campi virtuali possono anche accedere ai campi in modo ricorsivo come in:

```
>>> db.define_table('item',
        Field('unit_price','double'))
>>> db.define_table('order_item',
        Field('item',db.item),
        Field('quantity','integer'))
>>> class MyVirtualFields:
        def total_price(self):
            return self.order_item.item.unit_price*self.order_item.quantity
>>> db.order_item.virtualfields.append(MyVirtualFields())
```

dove `self` in nel campo ricorsivo `self.order_item.item.unit_price` è il record di loop.

I campi virtuali possono anche agire come il risultato di una JOIN:

```
>>> db.define_table('item',
        Field('unit_price','double'))
>>> db.define_table('order_item',
        Field('item',db.item),
        Field('quantity','integer'))
>>> rows =  db(db.order_item.item==db.item.id).select()
>>> class MyVirtualFields:
        def total_price(self):
            return self.item.unit_price*self.order_item.quantity
>>> rows.setvirtualfields(order_item=MyVirtualFields())
>>> for row in rows: print row.order_item.total_price
```

In questo caso la sintassi è differente. Il campo virtuale accede sia a `self.item.unit_price` che a `self.order_item.quantity` che appartiene alla JOIN. Il campo virtuale è collegato alle riga della tabella utilizzando il metodo `setvirtualfields` dell'oggetto Rows. Questo metodo richiede un numero arbitrario di argomenti con nome e può essere usato per impostare campi virtuali multipli, definite in classi diverse, e li collega alle tabelle:

```
>>> class MyVirtualFields1:
        def discounted_unit_price(self):
            return self.item.unit_price*0.90
```

```
>>> class MyVirtualFields2:
        def total_price(self):
            return self.item.unit_price*self.order_item.quantity
        def discounted_total_price(self):
            return self.item.discounted_unit_price*self.order_item.quantity
>>> rows.setvirtualfields(item=MyVirtualFields1(),order_item=MyVirtualFields2())
>>> for row in rows: print row.order_item.discounted_total_price
```

I campi virtuali possono essere "pigri" (lazy). Tutto quello che devono fare è ritornare una funzione e sono acceduti chiamando la funzione:

```
>>> db.define_table('item',
        Field('unit_price','double'),
        Field('quantity','integer'),
>>> class MyVirtualFields:
        def lazy_total_price(self):
            def lazy(self=self):
                return self.item.unit_price*self.item.quantity
            return lazy
>>> db.item.virtualfields.append(MyVirtualFields())
>>> for item in db(db.item.id>0).select(): print item.lazy_total_price()
```

o, in modo più compatto, utilizzando una funzione lambda:

```
>>> class MyVirtualFields:
        def lazy_total_price(self):
            return lambda self=self: self.item.unit_price*self.item.quantity
```

5.20 Relazione uno a molti

Per illustrare come implementare una relazione uno a molti con il DAL di web2py, definire un'altra tabella "dog" (cane) che referenzia la tabella "person":

```
>>> db.define_table('person',
                    Field('name'),
                    format='%(name)s')
>>> db.define_table('dog',
                    Field('name'),
                    Field('owner', db.person),
                    format='%(name)s')
```

La tabella "dog" ha due campi, il nome del cane ed il possessore del cane. Quando il tipo di un campo è un'altra tabella è sottinteso che il campo referenzia l'altra tabella tramite il suo campo "id". In effetti, stampando il tipo del campo si ottiene:

```
>>> print db.dog.owner.type
reference person
```

Inserendo tre cani, due posseduti da Alex e uno da 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
```

si può richiedere una selezione come si farebbe con qualsiasi altra tabella:

```
>>> for row in db(db.dog.owner==1).select():
        print row.name
Skipper
Snoopy
```

Poichè un cane ha un riferimento ad una persona, una persona può avere più cani così ogni record della tabella "person" acquisisce un nuovo attributo "dog" che è un Set che definisce i cani posseduti da quella persona. Questo consente di ciclare su tutte le persone ed ottenere facilmente i cani posseduti:

```
>>> 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
```

5.20.1 *Join interne*

Un altro modo per ottenere un risultato simile è utilizzare una INNER JOIN. web2py esegue le join automaticamente e trasparentemente quando la query collega due o più tabelle come nell'esempio successivo:

```
>>> 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
```

E' da notare che web2py ha unito le due tabelle e così ora ogni riga contiene due record, uno da ogni tabella, tra di loro collegati. Poichè i due record potrebbero avere campi con nomi uguali si deve specificare la tabella quando si vuole utilizzare un campo della riga. Questo significa che mentre prima era possibile utilizzare:

```
row.name
```

ed era evidente dal contesto se questo era il nome di una persona o il nome di un cane, nel risultato di una join è necessario essere più espliciti indicando il nome della tabella:

```
row.person.name
```

oppure:

```
row.dog.name
```

5.20.2 *Join esterne sinistre*

Nel risultato della join precedente Carl non appare perchè non ha nessun cane. Se si vuole selezionare le persone (indipendentemente dal fatto che abbiano associato un cane oppure no) e i loro cani (nel caso che ne abbiano)

allora deve essere eseguita una LEFT OUTER JOIN. Per fare questo è necessario utilizzare l'argomento "left" nel comando della select:

```
>>> 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
```

dove:

```
left = db.dog.on(...)
```

esegue la query della join a sinistra. In questo caso l'argomento di `db.dog.on` è la condizione necessaria per la join (la stessa usata precedentemente per la join interna). Nel caso di una join a sinistra è necessario indicare esplicitamente quale campo deve eseguire la select.

5.20.3 Raggruppamento e conteggio

Quando si eseguono le join si potrebbe voler raggruppare le righe secondo un criterio e contarle. Per esempio contare il numero dei cani posseduti da ogni persona. web2py consente di fare questo in tre passi: primo, è necessario avere un operatore di conteggio; secondo, si deve eseguire una join tra la tabella person e la tabella dog secondo il proprietario; terzo, si devono selezionare tutte le righe (person + dog), raggrupparle per person e contarle nel raggruppamento:

```
>>> 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[count]
Alex 2
Bob 1
```

Notare che l'operatore di conteggio (che è interno a web2py) è utilizzato come campo. Il problema è come recuperare l'informazione. Ogni riga chiaramente contiene una persona ed il conteggio, che non è però un campo nè di person, nè di table. Quindi da dove viene? In realtà si trova all'interno dell'oggetto Storage che rappresenta il record con una chiave uguale alla espressione della query.

5.21 Relazione molti a molti

Negli esempi precedenti un cane poteva avere un solo padrone ma una persona poteva possedere molti cani. Ma se Skipper fosse stato il cane di Alex e di Carl? Questo avrebbe richiesto una relazione molti a molti che viene realizzata tramite una tabella intermedia che collega una persona ad un cane con una relazione.

Ecco come fare:

```
>>> db.define_table('person',
                    Field('name'))
>>> db.define_table('dog',
                    Field('name'))
>>> db.define_table('ownership',
                    Field('person', db.person),
                    Field('dog', db.dog))
```

la relazione esistente di possesso del cane da parte di un proprietario può essere riscritta come:

```
>>> 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
```

Ora si può aggiungere la nuova relazione, Carl è il co-proprietario di Skipper:

```
>>> db.ownership.insert(person=3, dog=1) # Curt owns Skipper too
```

Poichè ora esiste una relazione a tre vie tra le tabelle può essere conveniente definire un nuovo set di righe su cui eseguire le operazioni:

```
>>> persons_and_dogs = db((db.person.id==db.ownership.person) | (db.dog.id==db.
    ownership.dog))
```

Ora è facile selezionare tutte le persone e i loro cani dal nuovo set:

```
>>> for row in persons_and_dogs.select():
        print row.person.name, row.dog.name
Alex Skipper
Alex Snoopy
Bob Puppy
Carl Skipper
```

Allo stesso modo è possibile trovare tutti i cani di cui Alex è il proprietario:

```
>>> for row in persons_and_dogs(db.person.name=='Alex').select():
        print row.dog.name
Skipper
Snoopy
```

e tutti i proprietari di Skipper:

```
>>> for row in persons_and_dogs(db.dog.name=='Skipper').select():
        print row.owner.name
Alex
Carl
```

Un'alternativa meno pesante rispetto alle relazioni molti a molti è il *tagging*. Il tagging è descritto nel contesto del validatore `IS_IN_DB`. Il tagging funziona anche con i adattatori di database che non supportano le join, come GAE.

6

Altri operatori

web2py ha altri operatori che rendono disponibile una API equivalente a quella degli operatori SQL. Ecco una nuova tabella chiamata "log" dove memorizzare gli eventi di sicurezza, l'orario (timestamp) dell'evento e il livello di severità (come numero intero):

```
>>> db.define_table('log', Field('event'),
                           Field('timestamp', 'datetime'),
                           Field('severity', 'integer'))
```

Come negli esempi precedenti si aggiungano un paio di eventi, un "port scan", un "xss injection" e un "unauthorized login". Per semplicità, gli eventi possono avere lo stesso timestamp ma differenti livelli di severità (1, 2 e 3).

```
>>> 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
```

6.0.1 *like, upper, lower*

I campi hanno un operatore `like` che può essere usato per ricercare le stringhe:

```
>>> for row in db(db.log.event.like('port%')).select():
        print row.event
port scan
```

In questo esempio "port%" indica una stringa che inizia con i caratteri "port". Il carattere del percento ('%') significa "qualsiasi sequenza di caratteri".

Allo stesso modo si possono usare i metodi `upper` e `lower` per convertire il valore di un campo in maiuscolo o in minuscolo. Questi metodi possono essere utilizzati insieme all'operatore `like`.

```
>>> for row in db(db.log.event.upper().like('PORT%')).select():
        print row.event
port scan
```

6.0.2 *year, month, day, hour, minutes, seconds*

I campi di tipo `date` e `datetime` possiedono i metodi `day`, `month` e `year` rispettivamente usati per ottenere il valore del giorno, del mese e dell'anno. I campi di tipo `time` e `datetime` possiedono i metodi `hour`, `minutes` e `seconds` per ottenere il valora dell'ora, dei minuti e dei secondi. Ecco un esempio:

```
>>> for row in db(db.log.timestamp.year()==2009).select():
        print row.event
port scan
xss injection
unauthorized login
```

6.0.3 *belongs*

L'opeartore SQL `IN` è realizzato con il metodo `belongs` che ritorna `True` quando il valore del campo appartiene ad uno specifico insieme di valori (liste o tuple):

```
>>> for row in db(db.log.severity.belongs((1, 2))).select():
        print row.event
port scan
xss injection
```

Il DAL consente anche di utilizzare una select nidificata come argomento dell'operatore `belongs` . L'unica accortezza da avere è quella di usare `_select` (e non `select`) e di selezionare esplicitamente un solo campo, che è quello che definisce l'insieme di valori.

```
>>> 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
```

In precedenza è stato usato l'operatore `count` per contare i record. Allo stesso modo si può usare l'operatore `sum` per sommare i valori di uno specifico campo in un gruppo di record. Come nel caso del conteggio il risultato della somma deve essere recuperato tramite l'oggetto Storage.

```
>>> sum = db.log.severity.sum()
>>> print db().select(sum).first()[sum]
6
```

6.1 *Denormalizzazione*

{DA COMPLETARE}

6.2 *Generazione del codice SQL*

A volte si può voler generare il codice SQL senza eseguirlo. In web2py questo è facile poichè ogni comando che esegue un'operazione di I/O nel database ha un comando equivalente che non esegue nessuna operazione e semplicemente ritorna il codice SQL che sarebbe stato eseguito. Questi comandi hanno lo stesso nome e la stessa sintassi dei comandi effettivi ma sono preceduti da un carattere di *underscore* ('_', sottolineatura):

Ecco l'esempio per `_insert`:

```
>>> print db.person._insert(name='Alex')
INSERT INTO person(name) VALUES ('Alex');
```

Ecco l'esempio per `_count`:

```
>>> print db(db.person.name=='Alex')._count()
SELECT count(*) FROM person WHERE person.name='Alex';
```

Ecco l'esempio per `_select`:

```
>>> print db(db.person.name=='Alex')._select()
SELECT person.id, person.name FROM person WHERE person.name='Alex';
```

Ecco l'esempio per `_delete`:

```
>>> print db(db.person.name=='Alex')._delete()
DELETE FROM person WHERE person.name='Alex';
```

Ed ecco l'esempio per `_update`:

```
>>> print db(db.person.name=='Alex')._update()
UPDATE person SET  WHERE person.name='Alex';
```

Inoltre è sempre possibile utilizzare `db._lastsql` *per ottenere il codice dell'ultimo codice SQL generato automaticamente dal DAL o manualmente con* `executesql`.

6.3 Esportare ed importare i dati

6.3.1 CSV (una tabella alla volta)

Quando un oggetto Rows è convertito in stringa è automaticamente serializzato in CSV (*Comma separated value*, valori separati da virgola):

```
>>> 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
```

Ecco come serializzare un'intera tabella in CSV e memorizzarla nel file "test.csv":

```
>>> open('test.csv', 'w').write(str(db(db.person.id).select()))
```

ed ecco come importare il contenuto del file CSV in una tabella:

```
>>> db.person.import_from_csv_file(open('test.csv', 'r'))
```

Durante l'importazione web2py cerca i nomi dei campi nell'header del file CSV. In questo esempio trova due colonne: "person.id" and "person.name", ignora il prefisso "person." ed ignora il campo "id". Tutti i record sono aggiunti con un nuovo valore "id". Le operazioni di importazione ed esportazione possono essere eseguite anche dall'interfaccia applicativa web **appadmin**.

6.3.2 CSV (tutte le tabelle contemporanemente)

In web2py è possibile eseguire il backup ed il restore di un intero database con due comandi:

Per esportare:

```
>>> db.export_to_csv_file(open('somefile.csv', 'wb'))
```

Per importare:

```
>>> db.import_from_csv_file(open('somefile.csv', 'rb'))
```

Questi metodi possono essere usati anche se il database da importare è di tipo differente rispetto a quello esportato. I dati sono memorizzati in un singolo file CSV ("somefile.csv") dove ogni tabella inizia con una linea che indica il nome della tabella ed un'altra linea con i nomi dei campi:

```
TABLE tablename
field1, field2, field3, ...
```

Due tabelle sono separate con i caratteri `\r\n\r\n`. Il file termina con la riga `END`.

Il file CSV non include i file caricati dagli utenti a meno che questi file non siano memorizzati in un campo del database. In ogni caso è abbastanza facile comprimere la cartella "uploads" separatamente.

Quando si esegue l'importazione i nuovi record saranno aggiunti al database. Generalmente i nuovi record non avranno lo stesso "id" del database originale ma web2py farà si che i riferimenti tra gli "id" dei record restino corretti anche se i valori sono cambiati.

Se una tabella contiene un file chiamato "uuid" questo campo è usato per identificare i duplicati: se un record importato ha lo stesso "uuid" di un record esistente, quest'ultimo sarà aggiornato con i valori del record importato.

6.3.3 CSV e la sincronizzazione remota del database

Con il seguente modello:

```
db = DAL('sqlite:memory:')
db.define_table('person',
    Field('name'),
    format='%(name)s')
db.define_table('dog',
    Field('owner', db.person),
    Field('name'),
    format='%(name)s')

if not db(db.person.id>0).count():
    id = db.person.insert(name="Massimo")
    db.dog.insert(owner=id, name="Snoopy")
```

Ogni record è identificato da un id e referenziato tramite quello stesso id. Se si hanno due copie del database usate da due distinte installazioni di web2py l'id è unico solo all'intero di ciascun database (e non tra di loro). Questo è un problema tipico che si ha quando si uniscono i record da due database differenti.

Per far si che i record siano univocamente identificabili devono:

- avere un id univoco (uuid);
- avere un timestamp (per riconoscere quale è il più recente nelle differenti copie);
- referenziare l'uuid invece dell'id.

Questo può essere ottenuto senza modifiche a web2py. Ecco cosa fare:

1. Cambiare il precedente modello in:

```
db.define_table('person',
    Field('uuid', length=64, default=uuid.uuid4()),
    Field('modified_on', 'datetime', default=now),
    Field('name'),
    format='%(name)s')

db.define_table('dog',
    Field('uuid', length=64, default=uuid.uuid4()),
    Field('modified_on', 'datetime', default=now),
    Field('owner', length=64),
```

```
    Field('name'),
    format='%(name)s')

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")
```

2. Creare un'azione di un controller per esportare il database:

```
def export():
    s = StringIO.StringIO()
    db.export_to_csv_file(s)
    response.headers['Content-Type'] = 'text/csv'
    return s.getvalue()
```

3. Creare un'azione di un controller per importare una copia salvata dell'altro database e sincronizzarne i record:

```
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)
```

4. Creare manualmente un indice per eseguire più velocemente la ricerca per uuid.

I passi 2 e 3 sono validi per qualsiasi modello di database, non sono specifici per questo esempio.

In alternativa è possibile utilizzare XML-RPC per esportare od importare il file.

Se i record referenziano dei file caricati dagli utenti è anche necessario esportare od importare il contenuto della cartella "uploads". I file all'interno della cartella sono già nominati tramite uuid così che non è necessario preoccuparsi nè dei conflitti di nome nè delle referenze.

6.3.4 HTML/XML (una tabella alla volta)

Gli oggetti Rows hanno anche un metodo `xml` (come le funzioni ausiliarie) per serializzarsi in 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>
    ...
  </tbody>
</table>
```

Se è necessario serializzare l'oggetto Rows in un altro formato XML con tag personalizzati questo si può fare facilmente utilizzando l'helper TAG e la notazione con *:

```
>>> 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>
  ...
</result>
```

6.3.5 *Rappresentazione dei dati*

La funzione `export_to_csv_file` accetta un argomento chiamato `represent`. Se impostato a `True` web2py utilizzerà l'output della funzione `represent` dei campi per l'esportazione dei dati.

La funzione `export_to_csv_file` accetta anche un argomento chiamato `colnames` che può contenere la lista dei nomi delle colonne che si desidera esportare. Il default è tutte le colonne.

Sia `export_to_csv_file` che `import_from_csv_file` accettano degli argomenti con nome che indicano a web2py come memorizzare o caricare il file:

- `delimiter`: indica il separatore dei valori dei campi (default ',')
- `quotechar`: indica il carattere per identificare le stringe (default '"', carattere di doppi apici)
- `quoting`: sistema di definizione delle stringhe (default `csv.QUOTE_MINIMAL`)

Ecco un esempio d'utilizzo:

```
>>> import csv
>>> db.export_to_csv_file(open('/tmp/test.txt', 'w'),
        delimiter='|',
        quotechar='"',
        quoting=csv.QUOTE_NONNUMERIC)
```

Che produce un output simile a:

```
"hello"|35|"this is the text description"|"2009-03-03"
```

Per altre informazioni consultare la documentazione ufficiale di Python.`quoteall`::cite

6.4 Caching delle SELECT

Il metodo `select` ha anche un argomento "cache" che per default è impostato a `None`. Per eseguire il caching deve essere impostato ad una tupla in cui il primo elemento è il modello di cache (cache.ram, cache.disk) da usare ed il secondo elemento è il tempo di validità della cache espresso in secondi.

Nel seguente esempio si vede un controller che esegue una select con cache sulla tabella "db.log" precedentemente definita. La select effettiva recupera i dati dal database non più di una volta ogni 60 secondi e memorizza il risultato in memoria. Se la prossima chiamata a questo controller avviene in meno di 60 secondi dall'ultima operazione, la select non viene eseguita ma i dati sono semplicemente recuperati dalla memoria.

```
def cache_db_select():
    logs = db().select(db.log.ALL, cache=(cache.ram, 60))
    return dict(logs=logs)
```

I risultati di una select sono oggetti complessi e perciò non possono essere memorizzati con Pickle nè essere memorizzati in una sessione o utilizzati con un meccanismo di cache diverso da quello illustrato in questa sezione.

6.5 Auto-riferimenti ed alias

E' possibile definire delle tabelle con campi che si riferiscono a se stessi, sebbene la normale notazione non funzioni correttamente.

Il seguente esempio non è corretto in quanto utilizza la variabile `db.person` prima che questa sia definita:

```
db.define_table('person',
    Field('name'),
    Field('father_id', db.person),
    Field('mother_id', db.person))
```

La soluzione consiste nell'utilizzare una notazione alternativa:

```
db.define_table('person',
    Field('name'),
    Field('father_id', 'reference person'),
    Field('mother_id', 'reference person'))
```

In effetti `db.tablename` e `"reference tablename"` sono tipi di campo equivalenti.

Se la tabella si riferisce a se stessa allora non è possibile eseguire una JOIN per selezionare una persona e i suoi genitori senza utilizzare la parola chiave AS di SQL. Per fare questo in web2py si utilizza la funzione `with_alias`. Ecco un esempio:

```
>>> 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
```

In questo esempio è stata fatta una chiara distinzione tra:

- "father_id": il nome del campo usato nella tabella "person";

- "father": l'alias usato per la tabella referenziata nel campo precedente; questo è poi comunicato al database;
- "Father": la variabile utilizzata da web2py per riferirsi all'alias.

La differenza è sottile e non ci sarebbe nulla di male nell'utilizzare lo stesso nome al posto dei tre diversi:

```
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
```

ma è necessario avere chiara la distinzione per costruire query corrette.

6.6 *Ereditarietà delle tabelle*

E' possibile creare una tabella che contiene tutti i campi da un'altra tabella. E' sufficiente passare a `define_table` la tabella pre-esistente al posto di un campo. Per esempio:

```
db.define_table('person', Field('name'))

db.define_table('doctor', db.person, Field('specialization'))
```

E' anche possibile definire una tabella fittizia che non è memorizzata in un database e riutilizzarla in altre tabelle. Per esempio:

```
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'))
```

Questo esempio assume che sia utilizzato il sistema standard di autenticazione di web2py (descritto nel capitolo 8).

7

Form e validatori

Ci sono quattro diverse modalità di creazione dei form in web2py:

- `FORM` rende disponibile una implementazione a basso livello basata sugli helper. Un oggetto `FORM` può essere serializzato in HTML ed è consapevole dei campi che contiene. Un oggetto `FORM` è in grado di validare i valori immessi nei campi.
- `SQLFORM` rende disponibile un API di alto livello per costruire form di creazione, modifica e cancellazione partendo da una tabella di un database.
- `SQLFORM.factory` è un livello di astrazione basato su `SQLFORM` per ottenere gli stessi vantaggi della creazione dei form anche se non è presente un database. Genera un form simile a `SQLFORM` partendo dalla descrizione di una tabella ma senza che sia necessario crearla effettivamente in un database
- `CRUD` (Create, Read, Update, Delete). QUesta API fornisce funzionalità equivalenti quelle di `SQLFORM` (ed è in effetti basata su `SQLFORM` ma con una notazione più compatta).

Tutti questi moduli sono auto-coscienti e, se i dati in input non superano la validazione possono modificarsi ed emettere uno o più messaggi d'errore.

I form possono essere interrogati per ottenere le variabili validate e per i messaggi d'errore generati nella validazione dell'input. Del codice HTML arbitrario può essere inserito o estratto dai form utilizzando gli helper.

7.1 FORM

In una applicazione **test** con il seguente controller "default.py":

```
def display_form():
    return dict()
```

e con la seguente vista associata "default/display_form.html":

```
{{extend 'layout.html'}}
<h2>Input form</h2>
<form enctype="multipart/form-data"
      action="{{=URL()}}" method="post">
Your name:
<input name="name" />
<input type="submit" />
</form>
<h2>Submitted variables</h2>
{{=BEAUTIFY(request.vars)}}
```

che è un normale form HTML che richiede il nome dell'utente. Quando questa form viene compilata e viene premuto il pulsante "submit" il form si auto-invia e la variabile `request.vars.name` è visualizzata, insieme al suo valore, nella pagina.

E' possibile generare lo stesso form utilizzando gli helper. Questo può essere fatto nella vista o nell'azione. Poichè web2py processa il form nell'azione è possibile definire il form stesso nell'azione.

Ecco il nuovo controller:

```
def display_form():
   form=FORM('Your name:', INPUT(_name='name'), INPUT(_type='submit'))
   return dict(form=form)
```

e la vista associata "default/display_form.html":

```
{{extend 'layout.html'}}
<h2>Input form</h2>
{{=form}}
<h2>Submitted variables</h2>
{{=BEAUTIFY(request.vars)}}
```

Il codice è equivalente a quello dell'esempio precedente, ma ora il form è generato in risposta al comando `{{=form}}` che serializza l'oggetto `FORM`.

E' possibile aggiungere un ulteriore livello di complessita aggiungendo la validazione e la gestione del form modificando il controller nel seguente modo:

```
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)
```

e la relativa vista "default/display_form.html":

```
{{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)}}
```

Notare che:

- Nell'azione è stato aggiunto il validatore `requires=IS_NOT_EMPTY()` per il campo "name".

- Nell'azione è stata aggiunta una chiamata alla funzione `form.accepts( ... )`
- Nella vista ora sono visualizzate le variabili `form.vars`, `form.errors` e `request.vars` oltre al form stesso.

Tutto il lavoro è eseguite dal metodo `accepts` dell'oggetto `form`. Questo metodo infatti filtra le variabili in `request.vars` secondo le clausole dei validatori dei campi (presenti nella definizione del form) e memorizza in `form.vars` le variabili che superano la validazione. Se il valore di un campo non supera la validazione, il validatore ritorna un errore che viene memorizzato in `form.errors`. SIa `form.vars` che `form.errors` sono oggetti di tipo `gluon.storage.Storage` simili a `request.vars`. Il primo contiene i valori che superano la validazione, per esempio:

```
form.vars.name = "Max"
```

Il secondo contiene gli errori, per esempio:

```
form.errors.name = "Cannot be empty!"
```

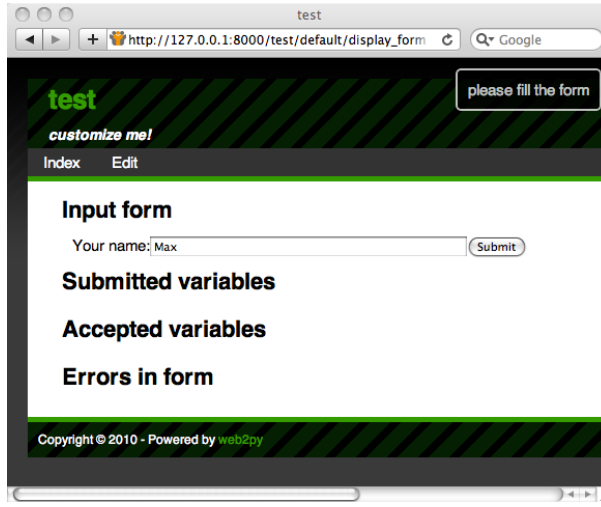
La sintassi completa del metodo `accepts` è la seguente:

```
form.accepts(vars, session=None, formname='default',
             keepvalues=False, onvalidation=None,
             dbio=True, hideerror=False):
```

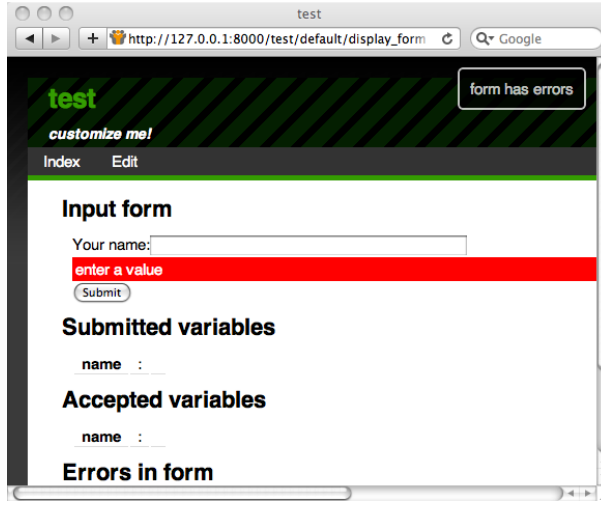
Il significato dei parametri opzionali è spiegato nelle prossime sezioni:

La funzione `accepts` ritorna `True` se il form è valido, altrimenti ritorna `False`. Un form non è accettato se ha errori o quando non è stato ancora inviato (per esempio, la prima volta che viene visualizzato).

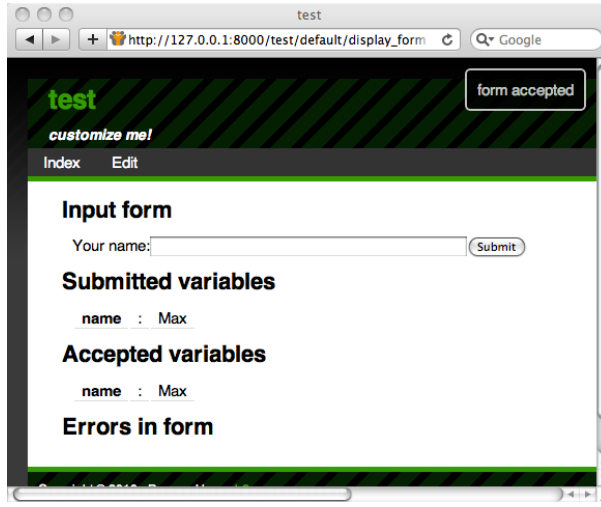
Ecco come appare questa pagina la prima volta che viene visualizzata:



Ecco come appare dopo un invio di dati non validi:



Ecco come appare dopo che i dati inviati sono stati tutti validati:



7.1.1 *Campi nascosti*

Quando il precedente oggetto form è serializzato con `{{=form}}` e dopo la chiamata al metodo `accepts` ha il seguente aspetto:

```
<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>
```

Notare la presenza di due campi nascosti: "_formkey" e "_formname". La loro presenza è intercettata dalla chiamata al metodo `accepts` è hanno due ruoli fondamentali:

- Il campo chiamato "_formkey" è un codice univoco che web2py utilizzare per evitare l'invio multiplo dello stesso form. Il valore di questo campo è generato quando il form è serializzato e memorizzato nell'oggetto `session`. Quando il form è inviato il valore nel form deve corrispondere a quello

memorizzato nella sessione, altrimenti `accepts` ritorna `False` senza nessun errore, come se il form non fosse stato inviato perchè web2py non è in grado di capire se il form è stato inviato correttamente.

- Il campo nascosto chiamato "_formname" è generato da web2py come nome per il modulo (ma può essere sovrascritto). Questo campo è necessario per il corretto funzionamento delle pagine che contengono form multipli perchè web2py li distingue l'uno dall'altro utilizzando il loro nome.
- Campi nascosti opzionali specificati in `FORM( ... , hidden=dict( ... ))`.

Il ruolo di questi campi nascosti e il loro utilizzo nelle form personalizzate e nelle pagine con form multiple è discusso in maggior dettaglio successivamente.

Se il form precedente è inviato con un campo "name" vuoto, il form non supera la validazione. Il form viene quindi nuovamente serializzato nel seguente codice HTML:

```
<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>
```

Notare la presenza di un DIV di classe "error" nell'HTML del form serializzato. web2py inserisce questo messaggio d'errore nel form per notificare all'utente che il campo non ha passato la validazione. Il metodo `accepts`, dopo l'invio, determina che il form è stato inviato, controlla se il campo "name" è vuote e se è obbligatorio e alla fine inserice il messaggio d'errore, generato dal validatore, nel form.

La vista di base "layout.html" ha anche il compito di gestire i DIV di classe "error" view is expected to handle DIVs of class "error". Questo layout utilizza gli effetti di jQuery per far apparire l'errore con un effetto di scorrimento in basso e un colore di sfondo rosso. Vedere il capitolo 10 per maggiori dettagli.

7.1.2 *keepvalues*

Il parametro opzionale `keepvalues` indica a web2py cosa fare quando il form è accettato e non c'è redirezione, così che il form è presentato di nuovo. Il default è che il form sia svuotato e ripresentato come nuovo. Se `keepvalues` è impostato a `True`, il form è pre-caricato con i valori precedentemente inseriti. Questo è utile quando un form deve essere usato ripetutamente per inserire record multipli simili. Se l'argomento `dbio` è impostato a `False` web2py non esegue nessuna operazione di inserimento/aggiornamento dopo aver accettato il form. Se `hideerror` è impostato a `True` ed il form contiene degli errori questi non saranno visualizzati (è compito del programmatore visualizzarli da `form.errors` in qualche modod). L'argomento `onvalidation` è spiegato nel prossimo paragrafo.

7.1.3 *onvalidation*

L'argomento `onvalidation` può essere `None` o una funzione che ha come argomento il form e non ritorna nulla. Tale funzione viene chiamata (con il form come argomento) subito dopo la validazione (se questa è superata) e prima di ogni anltra operazione. Questa funzione può avere diversi utilizzi: può essere usata, per esempio, per eseguire controlli aggiuntivi sul form ed eventualmente aggiungere errori oppure per caloclare il valore di alcuni campi basandosi sul contenuto dei campi inseriti oppure può essere usata per intercettare delle azione (come inviare una email) prima che il record sia creato o aggiornato. Ecco un esempio:

```
db.define_table('numbers',
    Field('a', 'integer'),
    Field('b', 'integer'),
    Field('c', '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(URL())
    return dict(form=form)
```

7.1.4 *Form e redirezione*

Il modo più comune di usare i form è tramte l'auto-invio, in modo che le variabili del form inviato siano processate dalla stessa azione che ha generato il form. Una volta che il form è accettato non dovrebbe essere ripresentata la stessa pagina (anche se, per mantenere semplici questi esempi, qui viene fatto così). E' più comune reindirizzare l'utente ad un'altra pagina "next". Ecco l'esempio del nuovo 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('next'))
    elif form.errors:
        response.flash = 'form has errors'
    else:
        response.flash = 'please fill the form'
    return dict(form=form)

def next():
    return dict()
```

Per impostare un messaggio sulla pagina successiva invece che sulla pagina corrente deve essere utilizzato `session.flash` invece di `response.flash`. web2py sposta il primo nel secondo dopo la redirezione. L'utilizzo di `session.flash` richiede che non venga utilizzato `session.forget()`.

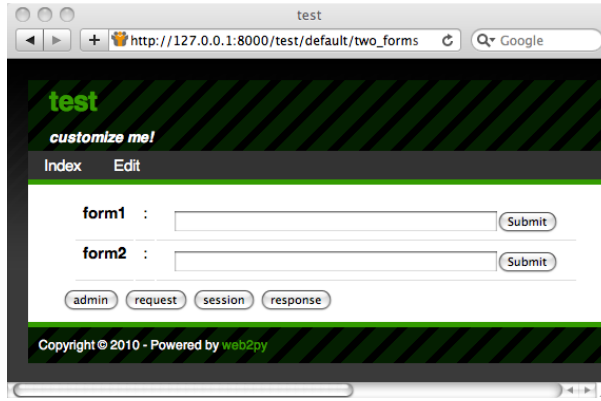
7.1.5 *Form multipli per pagina*

Il contenuto di questa sezione è valido sia per gli oggetti `FORM` che `SQLFORM`.

E' possibile avere più form sulla stessa pagina ma web2py deve essere in grado di distinguerli. Se i form sono derivati da tabelle diverse con `SQLFORM` web2py è in grado di dare ad ogni form un nome diverso, altrimenti il nome di ogni form deve essere univoco ed esplicitamente indicato. Inoltre quando più form sono presenti nella stessa pagina i meccanismo per evitare i doppi invii non funziona più in modo adeguato e quindi non deve essere utilizzato l'argomento`session` nella chiata al metodo `accepts`. Ecco un esempio:

```
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)
```

e questo è l'output che produce:



Quando l'utente invia un form1 vuoto solo form1 visualizza un messaggio

d'errore; se l'utente invia un form2 vuoto solo form2 visualizza un messaggio d'errore.

7.1.6 Postback o no?

Il contenuto di questa sezioen si applica sia agli oggetti `FORM` che `SQLFORM`. Il meccanismo descritto in questo capitolo è possibile ma non è raccomandato in quanto è sempre meglio avere dei form che si auto-inviano. A volte però non si ha questa possibilità perchè l'azione che invia il modulo e quella che lo riceve appartengono ad applicazioni diverse.

E' possibile generare un form che si invia ad un'azione differente. Questo è fatto specificano la URL di destinazione dell'azione negli attributi dell'oggetto `FORM` o `SQLFORM`. Per esempio:

```
form = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
        INPUT(_type='submit'), _action=URL('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()
```

Notare che, poichè sia "page_one" che "page_two" utilizzano lo stesso oggetto `form` questo è definito solo una volta al di fuori delle due azioni, in modo che non sia necessario ripeterne la definizione. La parte di codice comune all'inizio di un controller viene eseguita ogni volta prima di dare il controllo all'azione chiamata.

Poichè "page_one" non chiama il metodo `accepts` il form non ha ne nome ne chiave quindi non deve essere passatl 'oggetto `session` e deve essere impostato `formname=None` in `accepts`, altrimenti il form non sarà validato quando

"page_two" lo riceve.

7.2 *SQLFORM*

Per illustrare il livello successivo è necessario fornire un file di modello applicazione:

```
db = DAL('sqlite://storage.sqlite')
db.define_table('person', Field('name', requires=IS_NOT_EMPTY()))
```

e modificare il controller nel modo seguente:

```
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)
```

Non è necessario modificare la vista.

Nel nuovo controller non è necessario costruire un oggetto `FORM` poichè il costruttore di `SQLFORM` ne definisce uno partendo dalla tabella `db.person` definita nel modello. Questo nuovo form, quando viene serializzato, appare come:

```
<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>
```

Il form generato automaticamente è più complesso di quello creato con l'oggetto `FORM` di basso livello. Prima di tutto contiene una tabella di rige ed ogni riga ha tre colonne. La prima colonna contiene le etichette dei campi (come indicato in `db.person`), la seconda colonna contiene i campi di input (ed evantuali messaggi d'errore) e la terza colonna è opzionale ed inizialmente vuota (può essere popolata con le descrizioni dei campi nel costruttore di `SQLFORM`).

Tutti i tag del form hanno nomi derivati dal nome della tabella stessa e dai nomi dei campi. Questo permette una più facile personalizzazione tramite CSS e Javascript. Questa funzionalità è descritta in maggior dettaglio nel capitolo 10. Cosa più importante è che il metodo `accepts` esegue più lavoro di prima. Rispetto al precedente esempio oltre alla validazione dell'input, se questo ha esito positivo, esegue anche l'inserimento del nuovo record nel database e memorizza in `form.vars.id` l'id univoco del record appena creato.

Un oggetto `SQLFORM` gestisce automaticamente anche i campi di "upload" salvando i file caricati dagli utenti nella cartella "upload" dell'applicazione (dopo aver rinominato il file in modo sicuro per evitare conflitti e attacchi di tipo *Directory Traversal*) e memorizza il (nuovo) nome del file nel campo appropriato del database.

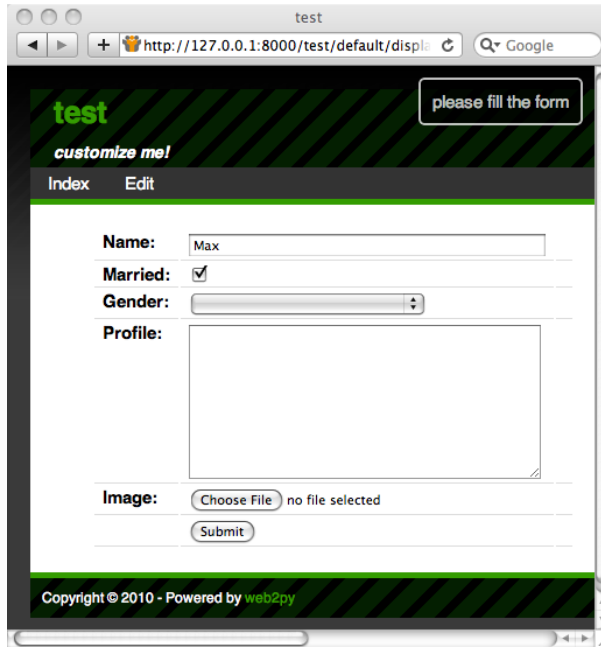
Un form derivato da `SQLFORM` visualizza i campi booleani con delle caselle di spunta, i campi di testo con aree di testo, campi contenenti valori definiti in un gruppo o in un databaase con menu a discesa e campi di "upload" dei fle con collegamenti che consentono agli utenti di scaricare il file caricato. I campi di tipo "blob" sono nascosti, perchè trattati in modo diverso, come descritto più avanti.

Per esmpio, con il seguente modello:

```
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'))
```

`SQLFORM(db.person)` genera questo form:



Il costruttore di `SQLFORM` consente diverse personalizzazioni come, per esempio, mostrare solo un sotto-insieme dei campi, cambiare le etichette, aggiungere valori alla terza colonna opzionale e creare form di aggiornamento (UPDATE) o di cancellazione (DELETE) in alternativa al form di inserimento (INSERT). `SQLFORM` è il singolo componente di web2py che fa risparmiare più tempo.

La classe `SQLFORM` è definita in "gluon/sqlhtml.py". Può essere facilmente estesa sostituendo il suo metodo `xml` che serializza gli oggetti per cambiarne

l'output.

La sintassi del costruttore di `SQLFORM` constructor è la seguente:

```
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, formstyle='table3cols',**attributes)
```

- Il secondo argomento opzionale trasforma il form di inserimento in un form di aggiornamento (UPDATE) per il record indicato (per maggiori dettagli vedere la prossima sezione).
-

Se `deletable` è impostato a `True`, il form di UPDATE visualizza la casella di spunta "Check to delete". Il testo dell'etichetta di questo campo è impostato con l'argomento `delete_label`.

- `submit_button` imposta il testo del pulsante di invio.
- `id_label` imposta il testo dell'etichetta dell'id del record.
- L'ìde del record non è mostrato se `showid` è impostato a `False`.
- `fields` è una lista (opzionale) dei nomi dei campi che si vogliono visualizzare nel form. Se è presente solo i campi nella lista saranno visualizzati. Per esempio:

```
fields = ['name']
```

- `labels` è un dizionario di etichette dei campi. La chiave del dizionario è il nome del campo ed il valore è ciò che viene visualizzato come etichetta. Se un'etichetta non è presente web2py la genera automaticamente partendo dal nome del campo (con l'iniziale maiuscola e con spazi al posto del carattere di sottolineatura). Per esempio:

```
labels = {'name':'Your Full Name:'}
```

- `col3` è un dizionario di valori per la terza colonna del form. Per esempio:

```
col3 = {'name':A('what is this?',
      _href='http://www.google.com/search?q=define:name')}
```

- `linkto` ed `upload` sono URL opzionali a controller definiti dall'utente che consentono al form di gestire i campi di riferimento. Sono discussi con maggior dettaglio in seguito.
- `readonly`. Se impostato a `True` visualizza il form in sola lettura.
- `comments`. Se impostato a `False` non visualizza la colonna dei commenti col3.
- `ignore_rw`. Normalmente per un modulo di creazione o aggiornamento sono visualizzati solo i campi indicati con `writable=True` e per i form in sola lettura sono visualizzati solo i campi indicati con `readable=True`. Impostando `ignore_rw=True` fa sì che questi vincoli siano ignorati e tutti i campi sono visualizzati. Questo è utilizzato principalmente nell'interfaccia dell'applicazione **appadmin** per visualizzare tutti i campi di una tabella.
- `formstyle` determina lo stile che web2py deve utilizzare per serializzare il form in HTML. Può assumere i seguenti valori: "table3cols" (tre colonne, il valore di default); "table2cols" (2 righe, una per etichetta e commento e l'altra per l'output); "ul" (per generare una lista non ordinata di campi di input); "divs" (rappresenta il form utilizzando i DIV per personalizzare il form tramite CSS). `formystyle` può anche essere una funzione (con gli argomenti record_id, field_label, field_widget e field_comment) che ritorna un oggetto di tipo TR().
- `attributes` include argomenti opzionali che si vuole far passara al tag `FORM`. Per esempio:

```
_action = '.'
_method = 'POST'
```

C'è anche uno speciale attribbuto . Quando un dizionario è passato come i suoi elementi sono trasformati in campi nascosti di input (vedere l'esempio per l'helper `FORM`nel capitolo 5).

7.2.1 SQLFORM e gli inserimenti, gli aggiornamenti e le cancellazioni

Se si passa un record come secondo argomento opzionale al costruttore di `SQLFORM` il form diventa un modulo d'aggiornamento per quel record. Questo significa che quando il form è inviato il record esistente viene aggiornato e nessun nuovo record è inserito. Se si imposta l'argomento `deletable=True` il form di aggiornamento visualizza anche una casella di spunta "Check to delete". Se viene selezionata il record è cancellato.

E' possibile modificare il controller dell'esempio precedente per far passare un argomento addizionale nella URL, come in:

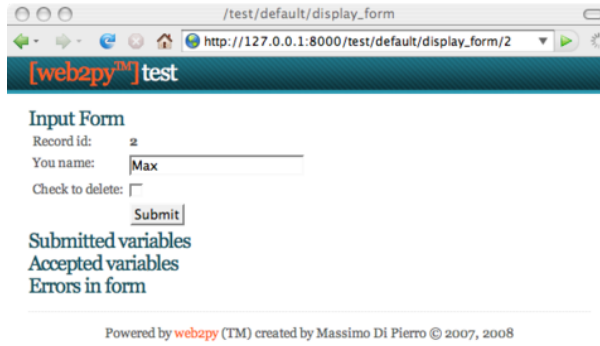
```
/test/default/display_form/2
```

in modo che se ci fosse un record con il corrisponendente id l'oggetto `SQLFORM` genera un form di aggiornamento/cancellazione per il record:

```
def display_form():
   record = db.person[request.args(0)]
   form = SQLFORM(db.person, record)
   if form.accepts(request.vars, session):
       response.flash = 'form accepted'
   elif form.errors:
       response.flash = 'form has errors'
   return dict(form=form)
```

In questo esempio la linea 3 recupera il record, la linea 5 genera un form di aggiornamento/cancellazione e la linea 7 crea un form di inserimento. La linea 8 fa tutto il resto per la gestione del form.

Ecco la pagina risultante:



`deletable=False` per default.

I form di aggiornamento cotengono anche un campo nascosto di input con `name="id"` che è usato per indentificare il record. Questo id è anche memorizzato sul server per maggio sicurezza e, se l'utente prova a modificarlo, l'aggiornamento nel database non viene eseguito. In questo caso web2py ritorna un SyntaxError con descrizione "user is tampering with form".

Quando un campo è indicato con `writable=False`, il campo non è mostrato nell'interfaccia di creazione del from ma è mostrato (in sola lettura) solamente nelle form di aggiornamento. Se un camo è indicato con `writable=False` e `readable=False` allora non è mostrato per nulla neanche nei form di aggiornamento.

I form creati con:

```
form = SQLFORM(...,ignore_rw=True)
```

ignorano gli attributi `readable` e `writable` e mostrano sempre tutti i campi. I form in **appadmin** ignorano per default questi attributi.

I form creati con:

```
form = SQLFORM(table,record_id,readonly=True)
```

mostrano sempre tutti i campi in modalità in sola lettura e non sono accettati per la modifica.

7.2.2 *SQLFORM in HTML*

Ci sono volte in cui si vogliono avere i benefici derivanti dall'uso di `SQLFORM` come la generazione del form e il processo di validazione ma si vuole avere un livello di personalizzazione dell'HTML del form che non può essere raggiunto con i soli parametri dell'oggetto `SQLFORM`. In questo caso è necessario progettare il form utilizzanod direttamente HTML.

Con una nuova azione nel precedente controller:

```
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()
```

il form può essere inserito manualmente nella relativa vista "default/display_manual_form.html":

```
{{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>
```

Notare che l'azione non ritorna il form perchè non ha bisogno di passarlo

alla vista che contiene un form creato direttamente in HTML. Il form contiene un campo nascosto "_formname" che deve essere lo stesso "formname" specificato come argomento del metodo `accepts` dell'azione. web2py utilizza il nome del form in caso di form multipli sulla stessa pagina per determinare quale form è stato inviato. Se la pagina contiene un solo form si può impostare `formname=None` ed omettere il campo nascosto nella vista.

7.2.3 SQLFORM e Upload

I campi di tipo "upload" sono speciali. Sono visualizzati come campi di INPUT di tipo `type="file"`. A meno che non sia specificato diversamente il file è caricato utilizzando un buffer ed è memorizzato nella cartella "uploads" dell'applicazione utilizzando un nome sicuro, assegnato automaticamente. Il nome del file è memorizzato nel campo di tipo "uploads".

Come esempio, considerando il seguete modello:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('image', 'upload'))
```

si può usare la stessa azione "display_form" mostrata precedentemente.

Quando si inserisce un nuovo record il form consente di selezionare un file. Se per esempio si sceglie un'immagine jpg il file è caricato e memorizzato come:

```
applications/test/uploads/person.image.XXXXX.jpg
```

"XXXXXX" è un identificatore casuale per il file assegnato da web2py.

Di default il nome originale di un file caricato è trasformato con la codifica "b16encode" ed utilizzato per costruire un nuovo nome per il file. Il nome è recuperato dall'azione di default "download" ed usato per impostare il contenuto dell'header del file originale.

Per motivi di sicurezza solo l'estensione del file è mantenuta, poichè il nome del file potrebbe contenere caratteri speciali che potrebbero consentire attacchi di tipo "directory traversal" o altre operazioni pericolose.

Il nuovo nome del file è memorizzato anche in `form.vars.image_newfilename`.

Quando si modifica un record utilizzando un form di aggiornamento sarebbe utile visualizzare un collegamento al file precedentemente caricato. web2py è in grado di fare questo.

Se si passa una URL al costruttore di `SQLFORM` tramite l'argomento "upload" web2py lo utilizza per scaricare il file. Nella seguente azione:

```
def display_form():
   record = db.person[request.args(0)]
   form = SQLFORM(db.person, record, deletable=True, upload=url)
   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)
```

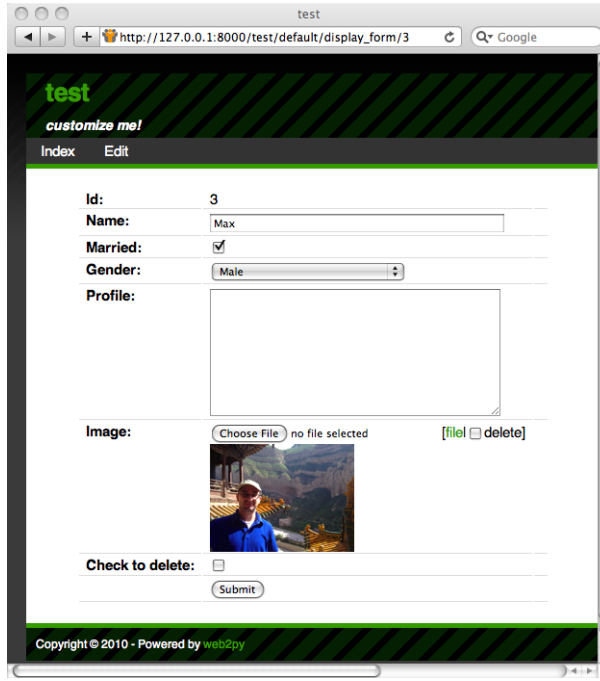
si inserisca un nuovo recordo alla URL:

```
http://127.0.0.1:8000/test/default/display_form
```

caricando un immagine, inviando il form e poi modificando il nuovo record appena creato visitando la URL:

```
http://127.0.0.1:8000/test/default/display_form/3
```

(nell'ipotesi che l'ultimo record inserito abbia id=3). Il form avrà il seguente aspetto:



Questo form, quando serializzato, genera il seguente 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>
```

che contiene un link per consentire lo scarico del file precedentemente caricato. Contiene inolter una casella di spunta per rimuovere il file dal record del database, memorizzanto NULL nel campo "image".

Perchè è disponibile questo meccanismo? Perchè è necessario scrivere la funzione di download? Il motivo è che si potrebbe voler utilizzare qualche meccanismo di autorizzazione nella funzione di download. Vedere il capitolo

8 per un esempio.

7.2.4 Memorizzare il nome originale del file

web2py memorizza automaticamente il nome del file originale all'interno del nuovo file codificato e lo recupera quando il file è scaricato. Al momento del download il nome originale del file è memorizzato nell'header che identifica il contenuto della risposta HTTP. Questo è eseguito automaticamente da web2py e non necessita alcun intervento da parte del programmatore.

A volte si potrebbe voler memorizzare il nome originale del file in un campo del database. In questo caso è necessario modificare il modello ed aggiungere un campo in cui memorizzarlo:

```
db.define_table('person',
    Field('name', requires=IS_NOT_EMPTY()),
    Field('image_filename'),
    Field('image', 'upload'))
```

va poi modificato il controller per gestire il nuovo campo:

```
def display_form():
    record = db.person[request.args(0)]
    url = URL('download')
    form = SQLFORM(db.person, record, deletable=True,
                   upload=url, 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)
```

Notare che `SQLFORM` non visualizza il campo "image_filename". L'azione "display_form" sposta il nome del file da `request.vars.image` a `form.vars.image_filename` in modo che possa essere processato da `accepts` e memorizzato nel database. La funzione di download, prima di restituire il file, controlla nel data il nome originale e lo imposta nell'header della risposta HTTP.

7.2.5 *autodelete*

Quando si esegue la cancellazione di un record `SQLFORM` non cancella i file caricati dall'utente e referenziati nel record. La ragione è che web2py non è in grado di determinare se lo stesso file è collegato ad altre tabelle o è utilizzato per altre operazioni. Se la rimozione del file è sicura quando il corrispondente record del database è cancellato si può impostare l'attributo `autodelete` a `True`:

```
db.define_table('image',
    Field('name'),
    Field('file','upload',autodelete=True))
```

L'attributo `autodelete` è impostato a `False` di default. Se viene impostato a `True` fa sì che web2py rimuova il file quando il relativo record è cancellato.

7.2.6 *Collegamenti per referenziare i record*

Si consideri il caso di due tabelle collegate da un campo di riferimento. Per esempio:

```
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')
```

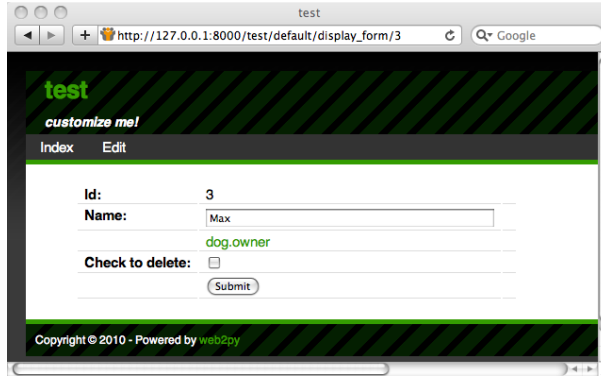
Una persona ha dei cani ed ogni cane appartiene ad un proprietario, che è una persona. A person has dogs, and each dog belongs to an owner, which is a person. Il proprietario del cane deve referenziare un `db.person.id` valido tramite `'%(name)s'`.

Con l'interfaccia **appadmin** di questa applicazione si aggiungano alcune persone e i loro cani.

Quando si modifica un record di persona esistente il form di aggiornamento di **appadmin** mostra un link ad una pagina che elenca i cani che appartengono a quella persona. Questo comportamento può essere replicato utilizzando l'argomento `linkto` dell'oggetto `SQLFORM`. `linkto` deve puntare alla URL di una nuova azione che riceve una stringa di ricerca dall'oggetto `SQLFORM` ed elenca i record corrispondenti. Ecco un esempio:

```
def display_form():
    record = db.person[request.args(0)]
    url = URL('download')
    link = URL('list_records')
    form = SQLFORM(db.person, records, deletable=True,
                   upload=url, linkto=link)
    if form.accepts(request.vars, session):
        response.flash = 'form accepted'
    elif form.errors:
        response.flash = 'form has errors'
    return dict(form=form)
```

Ecco la pagina:



C'è un collegamento chiamato "dog.owner". Il nome del collegamento può essere cambiato con l'argomento `labels` di `SQLFORM`, per esempio:

```
labels = {'dog.owner':"This person's dogs"}
```

Se si seleziona il link si è rediretti a:

```
/test/default/list_records/dog?query=dog.owner%3D5
```

"list_records" è l'azione specificata, con `request.args(0)` impostato al nome della tabella referenziata e con `request.vars.query` impostato alla stringa di ricerca SQL. La URL risultante in questo caso contiene il valore "dog.owner=5" correttamente codificato. web2py autoamticamente decodifica questa URL quando è ricevuta.

E' facile implementare una generica azione "list_record" nel seguente modo:

```
def list_records():
    table = request.args(0)
    query = request.vars.query
    records = db(query).select(db[table].ALL)
    return dict(records=records)
```

con la relativa vista "default/list_records.html":

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

Quando un set di record è ritornato da una select e serializzato in una vista è prima convertito in un oggetto SQLTABLE (da non confondere con Table) e poi serializzato in una tabella HTML dove ogni campo corrisponde ad una colonna.

7.2.7 *Pre-caricamento del form*

E' sempre possibile pre-caricare un form utilizzando la sintassi:

```
form.vars.name = 'fieldvalue'
```

Comandi di questo tipo devono essere inseriti dopo la dichiarazione del form e prima che il form sia accettato indipendentemente dal fatto che il campo ("name" in questo esempio) sia esplicitamente visualizzato nel form.

7.2.8 SQLFORM senza attività di I/O nel database

In alcuni casi si potrebbe voler generare un form da una tabella di database utilizzando `SQLFORM` per validare i dati inseriti ma senza generare nessuna operazione di inserimento, aggiornamento o cancellazione nel database. Questo è il caso, per esempio, di quando è necessario calcolare i campi da inserire dal valore di altri campi inseriti nel form, oppuer di quando è necessario eseguire operazioni aggiuntive di validazione sui dati inseriti che non possono essere fatti dai validatori standard.

Per fare questo è sufficiente suddividere il codice dell'azione:

```
form = SQLFORM(db.person)
if form.accepts(request.vars, session):
    response.flash = 'record inserted'
```

in:

```
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'
```

Lo stesso può essere fatto per i form di aggiornamento e cancellazione suddividendo:

```
form = SQLFORM(db.person,record)
if form.accepts(request.vars, session):
    response.flash = 'record updated'
```

in:

```
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 tutti e due i casi per quello che riguarda la gestione dei file caricati web2py si comporta come se `dbio=True`. Il nome del file caricato è in:

```
form.vars['%s_newfilename' % fieldname]
```

Per maggiori dettagli riferirsi al codice sorgente in "gluon/sqlhtml.py".

7.3 *SQLFORM.factory*

Ci sono casi in cui si vuole generare un form *come se* si avesse una tabella di database. In realtà si vuole solamente ottenere i vantaggi delle funzionalità di `SQLFORM` per generare un form utilizzabile tramite CSS e forse per eseguire dei caricamenti di file.

Per fare questo è disponibile l'API `form_factory`. Ecco un esempio dove viene generato un form che esegue la validazione dei dati inseriti, carica un file e memorizza il tutto nell'oggetto `session`:

```
def form_from_factory()
    form = SQLFORM.factory(
        Field('your_name', requires=IS_NOT_EMPTY()),
        Field('your_image', 'upload'))
    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)
```

Ecco la vista associata "default/form_from_factory.html":

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

E' necessario utilizzare un underscore invece degli spazi per le etichette dei campi o si deve esplicitamente passare a `form_factory` un dizionario di `labels`, come si farebbe con `SQLFORM`. Per default `SQLFORM.factory` genera il form con gli

attributi HTML "id" definiti come se il form fosse generato partendo da una tabella chiamata "no_table". Per cambiare questo nome di defaut utilizzare l'attributo `table_name`:

```
form = SQLFORM.factory(...,table_name='other_dummy_name')
```

E' necessario utilizzare `table_name` se si vogliono posizionare nella stessa pagina due form generati con il metodo factory per evitare conflitti CSS.

7.4 CRUD

Una delle aggiunte più recenti a web2py è la API CRUD (*Create/Read/Update/Delete*, Crea/Leggi/Aggiorna/Cancella) che si basa sulla API SQLFORM. CRUD crea un SQLFORM ma semplifica la codifica perchè incorpora la creazione del form, la sua gestione, la notifica e la redirezione tutto in una singola funzione.

La prima cosa da notare è che CRUD differisce dalle altre API di web2py utilizzate finora perchè non è automaticamente disponibile. Infatti deve essere prima importata e collegata al database su cui deve operare. Per esempio:

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

Il primo argomento del costruttore è il contesto corrente `globals()` così che CRUD possa accedere agli oggetti `request`, `response` e `session`. Il secondo argomento è un oggetto di connessione al database (`db` in questo caso).

L'oggetto `crud` appena definito fornisce la seguente API:

.

- `crud.tables()` ritorna una lista delle tabelle definite nel database.
- `crud.create(db.tablename)` ritorna un form di creazione per la tabella `tablename`.

- `crud.read(db.tablename, id)` ritorna un form a sola lettura per il record `id` della tabella `tablename`.
- `crud.update(db.tablename, id)` ritorna un form di aggiornamento per il record `id` della tabella `tablename`.
- `crud.delete(db.tablename, id)` cancella il record `id` della tabella `tablename`.
- `crud.select(db.tablename, query)` ritorna una lista dei record selezionati dalla tabella.
- `crud.search(db.tablename)` returns una tupla `(form, records)` dove `form` è un form di ricerca e `records` è una lista di record basati sul modulo di ricerca inviato
- `crud()` ritorna uno dei risultati di questo elenco basandosi sul contenuto di `request.args()`.

Per esempio, l'azione:

```
def data: return dict(form=crud())
```

espone le seguenti URL:

```
http://.../[app]/[controller]/data/tables
http://.../[app]/[controller]/data/create/[tablename]
http://.../[app]/[controller]/data/read/[tablename]/[id]
http://.../[app]/[controller]/data/update/[tablename]/[id]
http://.../[app]/[controller]/data/delete/[tablename]/[id]
http://.../[app]/[controller]/data/select/[tablename]
```

Mentre la seguente azione:

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

espone solo il metodo di creazione:

```
http://.../[app]/[controller]/create_tablename
```

E la seguente azione:

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

espone solamente il metodo di aggiornamento:

```
http://.../[app]/[controller]/update_tablename/[id]
```

e così via.

Il comportamento dell'oggetto CRUD può essere personalizzato in due modi: impostando alcuni suoi attributi o passando parametri opzionali ad ognuno dei suoi metodi.

7.4.1 Impostazioni

Ecco la lista completa degli attributi dell'oggetto CRUD, con il valore di default e la descrizione:

Per specificare la URL a cui rendirizzare dopo l'avvenuta creazione di un record:

```
crud.settings.create_next = URL('index')
```

Per specificare la URL a cui rendirizzare dopo l'avvenuto aggiornamento di un record:

```
crud.settings.update_next = URL('index')
```

Per specificare la URL a cui rendirizzare dopo l'avvenuta cancellazione di un record:

```
crud.settings.delete_next = URL('index')
```

Per specificare la URL da usare per collegare i file caricati:

```
crud.settings.download_url = URL('download')
```

Per specificare ulteriori funzioni da eseguire dopo le procedure standard di validazione nei form definiti con `crud.create`:

```
crud.settings.create_onvalidation = StorageList()
```

`StorageList` è lo stesso oggetto `Storage`, sono ambedue definiti in "gluon.storage", ma ha come default `[]` invece di `None`. `StorageList` permette la seguente sintassi:

```
crud.settings.create_onvalidation.mytablename.append(lambda form:....)
```

Per specificare ulteriori funzioni da eseguire dopo le procedure standard di validazione nei form definiti con `crud.update`:

```
crud.settings.update_onvalidation = StorageList()
```

Per specificare ulteriori funzioni da eseguire dopo il completamento dei form definiti con `crud.create`:

```
crud.settings.create_onaccept = StorageList()
```

Per specificare ulteriori funzioni da eseguire dopo il completamento dei form definiti con `crud.update`:

```
crud.settings.update_onaccept = StorageList()
```

Per specificare ulteriori funzioni da eseguire dopo il completamento dei form definiti con `crud.update` nel caso di cancellazione del record:

```
crud.settings.update_ondelete = StorageList()
```

Per specificare ulteriori funzioni da eseguire dopo il completamento dei form definiti con `crud.delete`:

```
crud.settings.delete_onaccept = StorageList()
```

Per determinare se il form generato con `crud.update` debba avere un pulsante di per la cancellazione del recorod:

```
crud.settings.update_deletable = True
```

Per determinare se il form generato con `crud.update` debba mostrare l'id del record corrente:

```
crud.settings.showid = False
```

Per determinare se i form devono mantenere i valori precedentemente inseriti devono essere reimpostati al loro default dopo un inserimento completato con successo:

```
crud.settings.keepvalues = False
```

Lo stile del form può essere cambiato con:

```
crud.settings.formstyle = 'table3cols' o 'table2cols' o 'divs' o 'ul'
```

7.4.2 *Messaggi*

Questa è la lista dei messaggi personalizzabili:

```
crud.messages.submit_button = 'Submit'
```

imposta il testo del pulsante di invio per i form di creazione e di aggiornamento.

```
crud.messages.delete_label = 'Check to delete:'
```

imposta il testo dell'etichetta del pulsante di cancellazione per i form di aggiornamento.

```
crud.messages.record_created = 'Record Created'
```

imposta il messaggio flash in caso di avvenuta creazione di un record.

```
crud.messages.record_updated = 'Record Updated'
```

imposta il messaggio flash in caso di avvenuto aggiornamento di un record.

```
crud.messages.record_deleted = 'Record Deleted'
```

imposta il messaggio flash in caso di avvenuta cancellazione di un record.

```
crud.messages.update_log = 'Record %(id)s updated'
```

imposta il messaggio di log per l'avvenuto aggiornamento di un record

```
crud.messages.create_log = 'Record %(id)s created'
```

imposta il messaggio di log per l'avvenuta creazione di un record

```
crud.messages.read_log = 'Record %(id)s read'
```

imposta il messaggio di log per l'avvenuta accesso in lettura ad un record.

```
crud.messages.delete_log = 'Record %(id)s deleted'
```

imposta il messaggio di log per l'avvenuta cancellazione di un record.

Notare che `crud.messages` appartiene alla classe `gluon.storage.Message` che è simile a `gluon.storage.Storage` ma i suoi valori sono automaticamente tradotti, senze la necessita di chiamare l'helper `T`.

I messaggi di log sono usati solamente se CRUD è connesso ad Auth, come discusso nel capitolo 8. Gli eventi sono registrati nella tabella Auth "auth_events".

7.4.3 Metodi

Il comportamento dei metodi dell'oggetto CRUD può essere personalizzato anche durante la chiamata al metodo stesso. Ecco la lista dei possibili argomenti:

```
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)
crud.search(table, query, queries, query_labels, fields, field_labels, zero)
```

- `table` è una tabella del DAL o il nome di una tabella su cui il metodo deve agire.
- `record` e `record_id` sono gli id del record su cui il metodo deve agire.
- `next` è la URL a cui il metodo deve reindirizzare in caso di completamento positivo dell'operazione. Se la URL contiene la stringa "[id]" questa sarà sostituita dall'id del record attualmente creato o modificato.
- `onvalidation` ha la stessa funzione di SQLFORM(..., onvalidation)
- `onaccept` è una funzione da chiamare dopo che l'invio del form è stato validato ma prima della redirezione.
- `log` è un messaggio di log. I messaggi di log possono accedere alle variabili nel dizionario `form.vars` come per esempio "%(id)s".
- `message` è il messaggio flash di conferma dell'accettazione del form.
- `ondelete` è chiamato al posto di `onaccept` quando il record viene cancellato tramite un form di aggiornamento.
- `deletable` determina se il form di aggiornamento deve avere una opzione per la cancellazione del record.
- `query` è la query da utilizzare per la selezione dei record.
- `fields` è la lista dei campi da selezionare.
- `orderby` determina l'ordine in cui i campi devono essere selezionati (vedere il capitolo 6).
- `limitby` determina il range dei record selezionati che devono essere visualizzati (vedere il capitolo 6).
- `headers` è un dizionario con i nomi delle intestazioni della tabella.
- `queries` è una lista come `['equals', 'not equal', 'contains']` che contiene i metodi consentiti nel form di ricerca.
- `query_labels` è un dizionario come `query_labels=dict(equals='Equals')` che assegna i nomi ai metodi di ricerca.
- `fields` è una lista di campi da visualizzare nel widget di ricerca.
- `field_labels` è un dizionario che collega i nomi dei campi alle etichette.
- `zero` ha come default "choose one" se è usato come un opzione di default per i menu a discesa nel widget di ricerca.

Ecco un esempio di utilizzo in una azione di un controller:

```
# assuming db.define_table('person', Field('name'))
def people():
    form = crud.create(db.person, next=URL('index'),
           message=T("record created"))
    persons = crud.select(db.person, fields=['name'],
           headers={'person.name', 'Name'})
    return dict(form=form, persons=persons)
```

Ecco un'altra funzione molto generica che consente di ricercare, creare e modificare qualsiasi record da qualsiasi tabella. Il nome della tabella è passato in `request.args(0)`:

```
def manage():
    table=db[request.args(0)]
    form = crud.update(table,request.args(1))
    table.id.represent = lambda id: \
       A('edit:',id,_href=URL(args=(request.args(0),id)))
    search, rows = crud.select(table)
    return dict(form=form,search=search,rows=rows)
```

Notare la linea `table.id.represent=...` che indica a web2py di cambiare la rappresentazione del campo id e visualizza un collegamento alla pagina stessa e passa l'id come `request.args(1)` che trasforma la pagina di creazione in una pagina di modifica.

7.4.4 Versioni del record

CRUD dispone di una modalità per gestire le versioni dei record utile per mantenere la revisione completa (*history*) di tutte le modifiche ai record.

Per attivare l'history su una tabella è sufficiente:

```
  form=crud.update(db.mytable,myrecord,onaccept=crud.archive)
```

`crud.archive` definisce una nuova tabella chiamata "db.mytable_history" (il nome è derivato dala tabella dell'esempio) è dopo ogni modifica ai record

memorizza nella nuova tabella una copia del record prima della modifica, includendo un riferimento al record corrente. Poichè il record è solamente aggiornato (e solo il suo stato precedente è archiviato) i riferimenti restano sempre validi. Questo è eseguito in automatico ma è comunque possibile accedere alla tabella di history. Per fare questo è necessario definire la nuova tabella nel modello:

```
db.define_table('mytable_history',
   Field('current_record',db.mytable),
   db.mytable)
```

La nuova tabella estende `db.mytable`, include cioè tutti i suoi campi e ha un riferimento a `current_record`.

`crud.archive` non memorizza informazioni sull'ora della modifica a meno che la tabella originale non abbia campi di tipo `timestamp`, per esempio:

```
db.define_table('mytable',
    Field('saved_on','datetime',
          default=request.now,update=request.now,writable=False),
    Field('saved_by',auth.user,
          default=auth.user_id,update=auth.user_id,writable=False),
```

Questi campi non hanno nulal di speciale e possono avere un nome qualsiasi. Sono riempiti prima che il record venga archiviato e sono archiviati con ogni copia del record.

Per cambiare il nome della tabella di history o il nome del campo di riferimento utilizzare gli argomenti `archive_table` e `current_record`:

```
db.define_table('myhistory',
    Field('parent_record',db.mytable),
    db.mytable)
# ...
form = crud.update(db.mytable,myrecord,
                   onaccept=lambda form:crud.archive(form,
                   archive_table=db.myhistory,
                   current_record='parent_record'))
```

7.5 Form personalizzati

Se un form è creato con SQLFORM, SQLFORM.factory o CRUD ci sono diversi modi, con diverse possibilità di personalizzazione, di inserirlo in una vista. Per esempio, con il seguente modello:

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

ed un'azione di upload:

```
def upload_image():
    return dict(form=crud.create(db.image))
```

Il modo più semplice per inserire il form nella relativa vista di `upload_image` è:

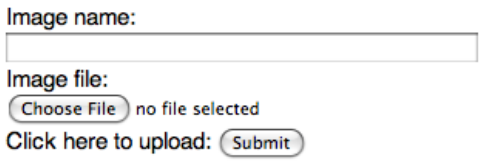
```
{{=form}}
```

che rappresenta il form con un layout di tabella standard. Se si vuole utilizzare un layout differente si può suddividere il form nei suoi componenti:

```
{{=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}}
```

Dove `form.custom.widget[fieldname]` viene serializzato nel corretto widget per il campo. Se il form è inviato e contiene degli errori questi sono visualizzati sotto il widget, come al solito.

Il risultato del precedente esempio è mostrato nella seguente immagine:



Se non si vogliono utilizzare i widget serializzati da web2py questi possono essere sostituiti con del codice HTML. Ci sono alcune variabili che risultano utili per questo scopo:

- `form.custom.label[fieldname]` contiene l'etichetta per il campo.
- `form.custom.dspval[fieldname]` rappresentazione del campo dipendente dal tipo di form e dal tipo di campo.
- `form.custom.inpval[fieldname]` valori da utilizzare nel codice del campo, dipendenti dal tipo del form e dal tipo del campo.

E' importante rispettare le convenzioni indicate più sotto.

7.5.1 CSS Conventions

I tag dei form generati da SQLFORM, SQLFORM.factory e CRUD seguono una rigida convenzione per i nomi CSS che può essere utilizzata per personalizzare ulteriormente i form.

Per una tabella "mytable", un campo "myfield" di tipo "string" è visualizzato per default da:

```
SQLFORM.widgets.string.widget
```

che corrisponde a:

```
<input type="text" name="myfield" id="mytable_myfield"
       class="string" />
```

Notare che:

- la classe del tag INPUT è la stessa del tipo del campo. Questo è fondamentale per il corretto funzionamento del codice jQuery in "web2py_ajax.html" e fa sì che si possano avere solamente numeri in campi "integer" e "double" e che "time", "date" e "datetime" visualizzino il pop-up per il calendario.
- l'îd è il nome della classe più il nome del campo, uniti da un carattere di sottolineatura. Questo permette di riferirsi in modo univoco ad un campo con, per esempio,`jQuery('#mytable_myfield')` è modificare il foglio di stile del campo o gli eventi associati al campo ((focus, blur, keyup, ecc.).
- il nome è, come è logico aspettarsi, il nome del campo.

7.5.2 Disattivare gli errori

A volte può essere necessario disabilitare il posizionamento automatico dei messaggi d'errore per visualizzarli in altre posizioni sulla pagina. Questo può essere fatto facilmente.

- Nel caso di FORM o SQLFORM va impostato `hiderror=True` quando si richiama il metodo `accepts`.
- Nel caso di CRUD va impostato `crud.settings.hiderror=True`

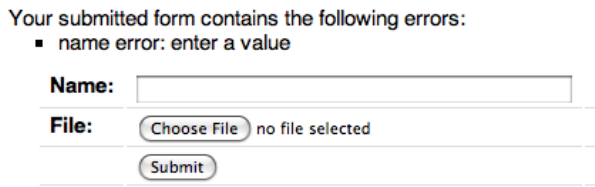
E' anche necessario modificare la vista per visualizzare l'errore (poichè non è più posizionato automaticamente).

Ecco un esempoio dove gli errori sono visualizzati sopra il form e non al suo interno.

```
{{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}}
```

Gli errori saranno visualizzati come nell'immagine seguente.



Questo meccanismo funziona anche per i form personalizzati.

7.6 *Validatori*

I validatori sono delle classi utilizzate per verificare i valori dei campi inseriti nei form (inclusi i form generati dalle tabelle di un database).

Ecco un esempio dell'utilizzo di un validatore in un `FORM`:

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

Ecco un esempio dell'utilizzo di un validatore in un campo di una tabella di database:

```
db.define_table('person', Field('name'))
db.person.name.requires = IS_NOT_EMPTY()
```

I validatori sono sempre assegnati utilizzando l'attributo `requires` di un campo. Un campo può avere uno o più validatori. In caso di più validatori questi devono essere inseriti in una lista:

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

I validatori sono chiamati dalla funzione `accepts` di un `FORM` o da altri helper HTML che contengono un oggetto form e sono eseguiti nell'ordine in cui sono elencati.

Il costruttore dei validatori standard di web2py ha un argomento opzionale `error_message` che consente di sovrascrivere il messaggio d'errore di default.

Ecco un esempio di validatore su una tabella di database:

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

dove è stato usato l'operatore `T` di traduzione per l'internazionalizzazione. I messaggi d'errore di default non sono tradotti.

7.6.1 Validatori di base

IS_ALPHANUMERIC

Questo validatore controlla che un campo contenga esclusivamente caratteri nel range "a-z", "A-Z" o "0-9".

```
requires = IS_ALPHANUMERIC(error_message=T('must be alphanumeric!'))
```

IS_DATE

Questo validatore controlla che il valore del campo contenga una data valida nel formato specificato. E' bene specificare il formato utilizzando l'operatore `T` di traduzione per supportare differenti formati in *locale* diversi.

```
requires = IS_DATE(format=T('%Y-%m-%d'),
                   error_message=T('must be YYYY-MM-DD!'))
```

Per la descrizione completa della direttiva "%" vedere la descrizione del validatore IS_DATETIME.

IS_DATE_IN_RANGE

Simile al validatore precedente ma consente di specificare un range di date:

```
requires = IS_DATE(format=T('%Y-%m-%d'),
                   minimum=datetime.date(2008,1,1),
                   maximum=datetime.date(2009,12,31),
                   error_message=T('must be YYYY-MM-DD!'))
```

Per la descrizione completa della direttiva "%" vedere la descrizione del validatore IS_DATETIME.

IS_DATETIME

Questo validatore controlla che un campo contenga una data e un orario validi nel formato specificato. E' bene specificare il formato utilizzando l'operatore `T` di traduzione per supportare differenti formati in *locale* diversi.

```
requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),
                       error_message=T('must be YYYY-MM-DD HH:MM:SS!'))
```

I seguenti simboli possono essere usati per la stringa di formato:

```
        as a decimal number [00,53]. All days in a new year preceding
        the first Sunday are considered to be in week 0.
        as a decimal number [00,53]. All days in a new year preceding
        the first Monday are considered to be in week 0.
```

IS_DATETIME_IN_RANGE

Simile al validatore precedente ma permette di specificare un range di data e orario:

```
requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),
                       minimum=datetime.datetime(2008,1,1,10,30),
                       maximum=datetime.datetime(2009,12,31,11,45),
                       error_message=T('must be YYYY-MM-DD HH:MM::SS!'))
```

Per la descrizione completa della direttiva "%" vedere la descrizione del validatore IS_DATETIME.

IS_DECIMAL

```
INPUT(_type='text', _name='name', requires=IS_DECIMAL_IN_RANGE(0, 10))
```

Determina se l'argomento è (o può essere rappresentato da) un oggetto *Decimal* di Python e garantisce che sia all'interno del range specificato. La comparazione è eseguita con l'aritmetica *Decimal* di Python. I limiti minimo e massimo possono essere impostati a `None` (nessun limite).

IS_EMAIL

Controlla che il valore del campo abbia un formato valido per un indirizzo di email. Non tenta di inviare una mail per conferma.

```
requires = IS_EMAIL(error_message=T('invalid email!'))
```

IS_EQUAL_TO

Controlla se il valore è uguale ad un altro valore (che può essere una variabile):

```
requires = IS_EQUAL_TO(request.vars.password,
                       error_message=T('passwords do not match'))
```

IS_EXPR

Il suo primo argomento è una stringa che contiene un'espressione logica con la variabile `value`. Valida il valore del campo se l'espressione è valutata `True`. Per esempio:

```
requires = IS_EXPR('int(value)%3==0',
                   error_message=T('not divisible by 3'))
```

E' necessario controllare prima che il valore sia un intero in modo da non generare un'eccezione.

```
requires = [IS_INT_IN_RANGE(0, 100), IS_EXPR('value%3==0')]
```

IS_FLOAT_IN_RANGE

Controlla che il valore del campo è un numero a virgola mobile con un range definito, `0 < value < 100` nel seguente esempio:

```
requires = IS_FLOAT_IN_RANGE(0, 100,
         error_message=T('too small or too large!'))
```

IS_INT_IN_RANGE

Controlla che il valore del campo sia un numero intero con un range definito, `0 < value < 100` nel seguente esempio:

```
requires = IS_INT_IN_RANGE(0, 100,
         error_message=T('too small or too large!'))
```

IS_IN_SET

Controlla che i valori del campo siano in un insieme predefinito:

```
requires = IS_IN_SET(['a', 'b', 'c'],zero=T('choose one'),
         error_message=T('must be a or b or c'))
```

L'argomento `zero` è opzionale e determina il testo dell'opzione selezionata di default, un'opzione che può non essere accettata dal validatore `IS_IN_SET`. Se non si vuole questa opzione impostare `zero=False`.

L'opzione `zero` è stata introdotto nella revisione 1.67.1 di web2py. Non impatta sulla retro-compatibilità nel senso che non modifica le applicazioni ma cambia il loro comportamento.

Gli elementi del set devono essere sempre di tipo stringa a meno che il validatore è preceduto da `IS_INT_IN_RANGE` (che converte il valore in un intero) o `IS_FLOAT_IN_RANGE` (che converte il valore in virgola mobile). Per esempio:

```
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 e i tag

Il validatore `IS_IN_SET` ha un attributo opzionale `multiple=False`. Se impostato a `True` più valori possono essere memorizzati nel campo. Il campo in questo caso deve essere di tipo stringa. I valori multipli sono memorizzati separati da "|". Referenze multiple sono gestite automaticmante nei form di creazione e aggiornamento ma sono trasparenti per il DAL. Per visualizzare i campi multipli è fortemente consigliato l'utilizzo del plugin *multiselect* di jQuery.

IS_LENGTH

Controlla che la lunghezza del valore del campo sia tra i limiti indicati, si può utilizzare sia per il testo che per l'upload dei file.

I suoi argomenti sono:

- maxsize: la lunghezza massima consentita
- minsize: la lunghezza minima consentita

Esempi:

Per controllare che la lunghezza del testo inserito sia minore di 33 caratteri:

```
INPUT(_type='text', _name='name', requires=IS_LENGTH(32))
```

Per controllare che il campo password sia più lungo di 5 caratteri:

```
INPUT(_type='password', _name='name', requires=IS_LENGTH(minsize=6))
```

Per controllare che la dimensione del file caricato sia tra 1 KB e 1 MB:

```
INPUT(_type='file', _name='name', requires=IS_LENGTH(1048576, 1024))
```

Per tutti i campi tranne quelli di tipo file questo validatore controlla la lunghezza del valore inserito. Nel caso di un file il valore è un `cookie.FieldStorage` che valida la lunghezza dei dati del file.

IS_LIST_OF

Questo non è esattamente un validatore. Il suo uso è quello di consentire la validazione di campi che ritornano valori multipli. E' usato nei rari casi in cui un form contiene campi multipli con lo stesso nome o un selettore multiplo. Il suo unico argomento è un altro validatore e quello che `IS_LIST_OF` fa è applicare l'altro validatore ad ogni elemento della lista. Per esempio la seguente espressione controlla che ogni oggetto nella lista sia un intero nel range 0-10:

```
requires = IS_LIST_OF(IS_INT_IN_RANGE(0, 10))
```

`IS_LIST_OF` non ritorna mai un errore nè contiene un messaggio d'errore. Il validatore specificato come argomento controlla la generazione degli errori.

IS_LOWER

Questo validatore non ritorna mai un errore ma converte il valore in minuscolo.

```
requires = IS_LOWER()
```

IS_MATCH

Questo validatore confronta il valore con un'espressione regolare e ritorna un errore se non corrisponde.

Ecco un esempio di utilizzo per verificare un codice postale americano:

```
requires = IS_MATCH('^\d{5}(-\d{4})?$',
         error_message='not a zip code')
```

Ecco un esempio per verificare un indirizzo IPv4:

```
requires = IS_MATCH('^\d{1,3}(\.\d{1,3}){3}$',
         error_message='not an IP address')
```

Ecco un esempio per validare un numero di telefono americano:

```
requires = IS_MATCH('^1?((-)\d{3}-?|\(\d{3}\))\d{3}-?\d{4}$',
         error_message='not a phone number')
```

Fare riferimento alla documentazione ufficiale di Python per maggiori informazioni sulle espressioni regolari.

IS_NOT_EMPTY

Questo validatore controlla che il valore contenuto nel campo non sia una stringa vuota.

```
requires = IS_NOT_EMPTY(error_message='cannot be empty!')
```

IS_TIME

Questo validatore controlla che il valore del campo contenga un orario valido nel formato specificato.

```
requires = IS_TIME(error_message=T('must be HH:MM:SS!'))
```

IS_URL

Questo validatore rifiuta una stringa contenente una URL nel caso in cui almeno una di queste condizioni sia vera:

- La stringa è vuota o è `None`.
- La stringa utilizza caratteri che non sono permessi in una URL.
- La stringa non rispetta le regole sintattiche dell'HTTP.
- Lo *schema* della URL (se è specificato) non è 'http' o 'https'.
- Il *top-level domain* (se l'host è stato specificato) non esiste.

(Queste regole sono basate sulla RFC 2616 (65))

Questa funzione controlla solamente la sintassi dell'URL, non controlla che la URL punti ad un documento reale o che abbia senso semanticamente.

Questa funzione aggiunge "http://" davanti alla URL nel caso che lo *schema* non sia specificato (per esempio "google.ca"). Se il parametro `mode='generic'` è utilizzato allora il comportamento della funzione cambia: rifiuta una stringa contenente una URL nel caso in cui almeno una di queste condizioni sia vera:

- La stringa è vuota o è `None`.
- La stringa utilizza caratteri che non sono permessi in una URL.
- Lo *schema* della URL (se è specificato) non è valido.

(Queste regole sono basate sulla RFC 2396 (66))

La lista degli *schema* consentiti è personalizzabile con il parametro `allowed_schemes` Se si esclude `None` dalla lista allora le URL abbreviate (senza *schema*) saranno rifiutate.

Lo schema aggiunto alla URL abbreviata è personalizzabile con il parametro `prepend_scheme`. Se si imposta `prepend_scheme` a `None` allora l'aggiunta sarà disabilitata. Le URL che richiedono l'aggiunta saranno ancora accettate ma non valore di ritorno non sarà modificato.

IS_URL è compatibile con lo standard IDN (*Internationalized Domain Name*) specificato nella RFC 3490 (67). Come risultato le URL possono essere stringhe regolari o di tipo unicode. Se il componente del dominio della URL (per esempio "google.ca") contiene caratteri non US-ASCII allora il dominio sarà convertito in *Punycode* (definito nella RFC 3492 (68)). IS_URL supera leggermente lo standard è consente ai caratti non US-ASCII di essere presenti nel path e nelle componenti della query string dell'URL. Questi caratteri non US-ASCII saranno codificati. Per esempio lo spazio sara codificato come '%20', il carattere unicode con codice hex 0x4e86 diventerà '%4e%86'.

Esempi:

```
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_SLUG

```
requires = IS_SLUG(maxlen=80, check=False, error_message='must be slug')
```

Se `check` è impostato a `True` controlla che il valore sia uno *slug* (solo caratteri alfanumerici e trattini non ripetuti).

Se `check` è impostato a `False` (il default) converte il valore in input in uno *slug*.

IS_STRONG

Esegue i controlli di complessità su un campo (solitamente un campo password).

Esempio:

```
requires = IS_STRONG(min=10, special=2, upper=2)
```

dove:

- `min` è la lunghezza minima del campo.
- `special` è il numero minimo di caratteri speciali (`!@#$%&*(){}[]-+`) richiesti.
- `upper` è il numero minimo di caratteri maiuscoli.

IS_IMAGE

Questo validatore controlla se il file caricato dall'utente utilizza uno dei seguenti formati d'immagine con le dimensioni (larghezza e altezza) entro i limiti prestabiliti. Non controlla la lunghezza massima del file (utilizzare il validatore `IS_LENGTH` per questo controllo). Ritorna un errore di validazione se non è stato caricato nessun dato. Supporta i formati BMP, GIF, JPEG e PNG e non richiede la *Python Imaging Library*.

Alcune parti di codice sono prese da ref. (69)

Ha i seguenti argomenti:

- `extensions`: un iterabile contenente le estensioni di file valide (in minuscolo).
- `maxsize`: un interabile contenente la larghezza e l'altezza massima dell'immagine.
- `minsize`: un interabile contenente la larghezza e l'altezza minima dell'immagine.

Usare `(-1, -1)` in `minsize` per ignorare il controllo sulle dimensioni dell'immagine.

Ecco alcuni esempi:

- per controllare che il file caricato sia in un formato supportato:

```
requires = IS_IMAGE()
```

- per controllare se il file caricato è in formato JPEG oppure PNG:

```
requires = IS_IMAGE(extensions=('jpeg', 'png'))
```

- per controllare se il file caricato è in formato PNG con dimensione massima di 200x200 pixel:

```
requires = IS_IMAGE(extensions=('png'), maxsize=(200, 200))
```

IS_UPLOAD_FILENAME

Questo validatore controlla se il nome e l'estensione del file caricaco dall'utente corrisponde ad un dato criterio. Non controlla in nessun caso il tipo del file e restituisce un errore di validazione in caso di nessun dato caricato.

I suoi argomenti sono:

- `filename`: espressione regolare per il nome del file (prima del punto).
- `extension`: espressione regolare per l'estensione del file (dopo il punto).
- `lastdot`: quale punto (*dot*) deve essere considerato come separatore tra nome del file ed estensione: `True` indica l'ultimo punto (per esempio "file.tar.gz" sarà suddiviso in "file.tar" + "gz") mentre `False` indica il primo punto (per esempio "file.tar.gz" sarà suddiviso in "file" + "tar.gz").
- `case`: 0 - lascia inalterati i caratteri maiuscoli/minuscoli, 1 - trasforma la stringa in minuscolo (il default), 2 - trasforma la stringa in maiuscolo.

Se nel campo non ci sono punti il controllo dell'estensione verrà eseguito su una stringa vuota e il nome del file sarà l'intero valore.

Esempi:

- Per controlare se un file ha un'estensione "pdf" (senza controllo delle maiuscole/minuscole):

```
requires = IS_UPLOAD_FILENAME(extension='pdf')
```

- per controllare se un file ha una estensione "tar.gz" e il nome che inizia con "backup":

```
requires = IS_UPLOAD_FILENAME(filename='backup.*', extension='tar.gz', lastdot=
    False)
```

- per controllare che il file non abbia estensione e il nome sia "README" (tutto maiuscolo):

```
requires = IS_UPLOAD_FILENAME(filename='^README$', extension='^$', case=0)
```

IS_IPV4

Questo validatore controlla se il valore di un campo è un indirizzo IP (versione 4) in forma decimale. Può essere impostato per controllare che il valore

sia in un range di indirizzi specifico. L'espressione regolare per IPv4 è stata presa da ref. (70)

I suoi argomenti sono:

- `minip` una stringa contenente l'indirizzo IP più basso consentito, per esempio 192.168.0.1 oppure un iterabile di numeri ([192, 168, 0, 1]) oppure un intero (3232235521).
- `maxip` una stringa contenente l'indirizzo IP più alto consentito con la stessa sintassi di `minip`).

Tutti e tre i valori d'esempio sono uguali poichè gli indirizzi sono convertiti in interi per il controllo d'inclusione con la seguente funzione:

```
number = 16777216 * IP[0] + 65536 * IP[1] + 256 * IP[2] + IP[3]
```

Exampi:

- per controllare un indirizzo IP versione 4 valido:

```
requires = IS_IPV4()
```

- per controllare un indirizzo IP versione 4 privato:

```
requires = IS_IPV4(minip='192.168.0.1', maxip='192.168.255.255')
```

IS_UPPER

Questo validatore non ritorna mai un errore ma converte il valore in maiuscolo.

```
requires = IS_UPPER()
```

IS_NULL_OR

Deprecato, è un alias per `IS_EMPTY_OR` descritto più sotto.

IS_EMPTY_OR

A volte è necessario che un cambio abbia un certo formato oppure sia vuoto, per esempio un campo può contenere una data ma potrebbe anche essere vuoto. Il validatore `IS_EMPTY_OR` serve a questo:

```
requires = IS_NULL_OR(IS_DATE())
```

CLEANUP

Questo è un filtro e non ritorna mai un errore ma semplicemente rimuove tutti i caratteri il cui valore ASCII non sia nella lista [10, 13, 32-127].

```
requires = CLEANUP()
```

CRYPT

Anche questo è un filtro ed esegue un *hash* sicuro sull'input ed è usato per evitare che le password siano memorizzate in chiaro nel database.

```
requires = CRYPT()
```

Se una chiave non è specificata utilizza l'algoritmo "MD5". Se invece è specificata una chiave `CRYPT` utilizza l'algoritmo "HMAC". La chiave può contenere un prefisso che determina l'algoritmo da utilizzare con "HMAC", per esempio "SHA512":

```
requires = CRYPT(key='sha512:thisisthekey')
```

Questa è la sintassi raccomandata. La chiave deve essere una stringa univoca associata al database utilizzato e non deve mai essere cambiata. Se la chiave viene persa il valore criptato diventa inutilizzabile.

7.6.2 *Validatori di database*

IS_NOT_IN_DB

Considerare il seguente esempio:

```
db.define_table('person', Field('name'))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

Questo modello controlla che quando si inserisce una nuova persona il suo nome non sia già nel campo `person.name` del database `db`. Come con tutti gli altri validatori questo controllo è eseguito a livello di gestione del form e non del database. Questo significa che c'è una seppur minima possibilità che, se due utenti tentano di inserire contemporaneamente due record con lo stesso `person.name` tutti e due i record saranno accettati. E' quindi più sicuro informare anche il database dell'univocità dei valori nel campo `person.name`:

```
db.define_table('person', Field('name', unique=True))
db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

Ora, se due utenti tentano di inserire lo stesso `person.name` nello stesso istante il database genera un `OperationalError` e solo uno dei due record è inserito.

Il primo argomento di `IS_NOT_IN_DB` può essere una oggetto di connessione ad un database oppure un oggetto Set. Nell'ultimo caso si controllano solo i valori presenti nel Set.

Il codice seguente, per esempio, non consente di ripetere la registrazione di due persone con lo stesso nome prima di 10 giorni dal primo inserimento.

```
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

La seguente tabella:

```
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',
                                 zero=T('choose one'))
```

è controllata al livello del form di inserimento/aggiornamento/cancellazione di un record `dog`. Richiede che un `dog.owner` sia un id valido del campo `person.id` nel database `db`. Grazie a questo validatore il campo `dog.owner` è rappresentato con un menu a tendina. Il terzo argomento del validatore è una stringa che descrive gli elementi nel menu a tendina. Nell'esempio si vuole che sia visualizzato il nome (`%(name)s`) della persona invece che il suo id (`%(id)s`). `%( ... )s` è sostituito dal valore del campo nelle parentesi per ogni record.

L'opzione `zero` è simile a quella per il il validatore `IS_IN_SET`.

Se si vuol far eseguire la validazione, ma senza il menu a tendina, si deve porre il validatore in una lista.

```
db.dog.owner.requires = [IS_IN_DB(db, 'person.id', '%(name)s')]
```

Il primo argomento del validatore può essere una connessione ad un database oppure un oggetto Set del DAL, come in `IS_NOT_IN_DB`.

A volte si può volere il menu a tendina (e quindi non si può porre il validatore in una lista) e altri validatori aggiuntivi. Per questo motivo il validatore `IS_IN_DB` ha l'argomento opzionale `_and` che può puntare ad una lista di altri validatori applicati in caso che il valore superi il controllo di `IS_IN_DB`. Per esempio per validare tutti i `dog.owner` nel db che non sono in un sotto-insieme:

```
subset=db(db.person.id>100)
db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
                                 _and=IS_NOT_IN_DB(subset,'person.id'))
```

IS_IN_DB and Tagging

Il validatore `IS_IN_DB` ha un attributo opzionale `multiple=False`. Se questo attributo viene impostato a `True` nel campo può essere memorizzato più di un valore. Il campo in questo caso non può essere una referenza ma deve essere una stringa. I valori multipli sono memorizzati separati dal carattere "`|`". Le referenze multiple sono gestite automaticamente nei form di creazione e di aggiornamento ma sono trasparenti per il DAL. Si suggerisce l'utilizzo del plugin *multiselect* di jQuery per visualizzare campi multipli.

7.6.3 Validatori personalizzati

Tutti i validatori seguono questo prototipo:

```
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)
```

Quando è chiamato per validare un valore un validatore ritorna una tupla `(x, y)`. Se `y` è `None` allora il valore ha passato la validazione e `x` contiene un valore elaborato. Per esempio, se il validatore richiede che il valore sia un intero `x` è convertito in `int(value)`. Se il valore non passa la validazione `x` contiene il valore di input e `y` contiene un messaggio d'errore che indica perchè la validazione è fallita. Il messaggio d'errore è usato per riportare l'errore nel form per il campo che non è stato validato.

Il validatore può anche contenere un metodo `formatter` che deve eseguire la conversione opposta del metodo `__call__`. Per esempio, considerato il codice sorgente di `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))
```

se la validazione ha successo il metodo `__call__` legge una stringa dal form e la converte in un oggetto di tipo "datetime.date" utlizzando il formato della stringa specificato nel costruttore. Il metodo `formatter` invece prende un oggetto di tipo "datetime.date" e lo converte in una stringa utilizzando lo stesso formato. Il metodo `formatter` viene chiamato automaticamente nei form ma può anche essere chiamato esplicitamente per convertire gli oggetti nella loro corretta rappresentazione. Per esempio:

```
>>> 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))
>>> row = db.atable[id]
>>> print db.atable.formatter(row.birth)
01/01/2008
```

Quando è necessario più di un validatore, questi sono memorizzati in una lista e vengono eseguiti nell'ordine di inserimento. L'output di ognuno è passato come input al validatore successivo. La catena si interrompe non appena uno dei validatori fallisce.

Allo stesso modo, quando viene chiamato il metodo `formatter` di un campo i metodi `formatter` dei validatori associati sono richiamati in ordine inverso.

7.6.4 *Validatori con dipendenze*

Potrebbe essere necessario validare un campo il cui validatore dipende dal valore di un altro campo. Per fare questo è necessario impostare il validatore nel controller, quando il valore dell'altro campo è noto. Per esempio ecco una pagina che genera un form di registrazione che richiede un utente e una password per due volte. Nessuno dei campi può essere vuoto e le due password inserite devono corrispondere:

```
def index():
    form = SQLFORM.factory(
        Field('username', requires=IS_NOT_EMPTY()),
        Field('password', requires=IS_NOT_EMPTY()),
        Field('password_again',
              requires=IS_SAME_AS(request.vars.password)))
    if form.accepts(request.vars, session):
        pass # or take some action
    return dict(form=form)
```

Lo stesso meccanismo può essere applicato agli oggetti FORM e SQLFORM.

7.7 *Widgets*

Questi sono i widget disponibili in web2py:

```
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
SQLFORM.widgets.autocomplete
```

I primi dieci sono i widget di default per i corrispondenti tipi di campo. Il widget "options" è usato quando un campo richiede il validatore `IS_IN_SET` o `IS_IN_DB` con `multiple=False` (il default). Il widget "multiple" è utilizzato quando un campo richiede il validatore `IS_IN_SET` o `IS_IN_DB` con `multiple=True`. I widget "radio" e "checkboxes" non sono mai usati di default ma possono essere impostati manualmente. Il widget "autocomplete" è speciale ed è discusso in una sezione dedicata più avanti in questo capitolo.

Per esempio, per rappresentare un campo "string" con una "textarea":

```
Field('comment', 'string', widget=SQLFORM.widgets.text.widget)
```

E' anche possibile creare nuovi widget o estendere quelli esistenti.

`SQLFORM.widgets[type]` è una classe e `SQLFORM.widgets[type].widget` è una funzione membro statica della classe corrispondente. Ciascuna funzione di un widget ha due argomenti: l'oggetto `Field` e il valore corrente di quel campo. La funzione ritorna la rappresentazione del widget. Per esempio il widget per l'oggetto `string` potrebbe essere riscritto come segue:

```
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)
```

Un widget può contenere i suoi validatori ma è buona regola associare il validatore all'attributo "requires" del campo e far sì che il widget lo legga da lì.

7.7.1 Widget di Autocomplete

Ci sono due possibili utilizzi per il widget "autocomplete": per completare automaticamente un campo che prende il valore da una lista o per completare

automaticamente un campo di riferimento (dove la stringa che deve essere completata automaticamente è una rappresentazione del riferimento che è implementato come un id).

Il primo caso è semplice:

```
db.define_table('category',Field('name'))
db.define_table('product',Field('name'),Field('category')
db.product.category.widget = SQLHTML.widgets.autocomplete(
     request, db.category.name, limitby=(0,10), min_length=2)
```

Dove `limitby` indica al widget di non visualizzare più di 10 suggerimenti per volta e `min_length` indica al widget di eseguire una chiamata Ajax per recuperare i suggerimenti solo dopo che l'utente ha digitato almeno 2 caratteri nel campo di ricerca.

Il secondo caso è più complesso:

```
db.define_table('category',Field('name'))
db.define_table('product',Field('name'),Field('category')
db.product.category.widget = SQLHTML.widgets.autocomplete(
     request, db.category.name, id_field=db.category.id)
```

In questo caso il valore di `id_field` indica al widget che anche se il valore da completare è un `db.category.name` quello che deve essere memorizzato è il corrispondente `db.category.id`. Un parametro opzionale è `orderby` che istruisce il widget su come ordinare i suggerimenti (in ordine alfabetico per default).

Questo widget utilizza Ajax ma dov'è il *callback* della funzione Ajax? Effettivamente in questo widget vi è una certa dose di "magia". Il metodo *callback* è l'oggetto widget stesso. Com'è esposto? In web2py qualsiasi parte di codice può generare una risposta con una eccezione HTTP. Questo widget sfrutta questa possibilità nel seguente modo: il widget invia una chiamata Ajax alla stessa URL che ha generato il widget è mette un token speciale in `request.vars`. Il widget viene nuovamente istanziato, trova il token e genera l'eccezione HTTP che risponde alla richiesta. Tutto questo è fatto in automatico ed è nascosto allo sviluppatore.

8

Controllo d'accesso

web2py include un meccanismo di Controllo d'accesso basato sui ruoli (*Role Based Access Control*) potente e personalizzabile.

Questa è la definizione di RBAC da Wikipedia:

"Nella sicurezza informatica, il Role-based access control (in italiano: Controllo degli accessi basato sui ruoli) in sigla RBAC è un approccio a sistemi ad accesso ristretto per utenti autorizzati. É una più recente alternativa al Mandatory Access Control (Controllo degli accessi vincolato) e al Discretionary Access Control (Controllo degli accessi discrezionale).

RBAC è una tecnologia per il controllo d'accesso flessibile e neutrale sufficientemente potente per simulare DAC e MAC. D'altra parte MAC può simulare RBAC se il grafo dei ruoli è ristretto ad un albero piuttosto che ad un set parzialmente ordinato.

Prima dello sviluppo di RBAC, MAC e DAC erano considerati gli unici modelli conosciuti di controllo d'accesso: se un modello non era di tipo MAC era considerato essere di tipo DAC e vice versa. Ricerche eseguite negli anni '90 hanno dimostrato che RBAC non ricade in nessuna dei queste due categorie.

All'interno di una organizzazione i ruoli sono creati per diverse funzioni di lavoro. I permessi per eseguire specifiche operazioni sono assegnate a specifici ruoli. Ai membri di un gruppo sono assegnati particolari ruoli e, attraverso queste assegnazioni, questi acquisiscono il permesso di eseguire specifiche funzioni. A differenza dei sistemi d'accesso basati sul contesto (*Context based access control*, CBAC) RBAC non tiene in considerazione il contesto del messaggio (come per esempio l'origine della connessione). Poichè i permessi non sono assegnati direttamente agli utenti ma vengono acquisiti solo tramite il ruolo (o i ruoli) ad essi assegnati, la gestione dei diritti individuali per un utente diventa una semplice assegnazione dei ruoli appropriati per l'utente stesso. Questo semplifica le operazioni comuni, come l'aggiunta di un utente o il cambio di dipartimento.

RBAC si differenzia dalle Liste di controllo d'accesso (*Access Control List*, ACL) usate nei sistemi di controllo discrezionali per il fatto che assegna i permessi alle operazioni specifiche che hanno un significato all'interno dell'organizzazione piuttosto che agli oggetti e ai dati di livello più basso. Per esempio una ACL può essere utilizzata per assegnare o rimuovere il diritto di scrittura ad un particolare file ma non indica con quali procedure il file può essere modificato".

La classe di web2py che implementa RBAC è chiamata **Auth**.

Per funzionare **Auth** necessita delle seguenti tabelle (che vengono definite in automatico):

- `auth_user` memorizza il nome dell'utente, l'indirizzo email, la password e lo status (registrazione in attesa, registrazione accettata, utente bloccato).
- `auth_group` memorizza i gruppi o i ruoli per gli utenti in una struttura molti a molti. Per default ogni utente ha un suo proprio gruppo ma può anche essere assegnato a più gruppi e, di conseguenza, un gruppo può contenere più utenti. Ogni gruppo è identificato da un ruolo e da una descrizione.
- `auth_membership` memorizza la relazione tra utenti e gruppi in una struttura molti a molti.

- `auth_permission` memorizza la relazione tra gruppi e permessi. Un permesso è identificato da un nome e, opzionalmente, dal riferimento ad una tabella e ad un record. Per esempio i membri di uno specifico gruppo possono avere il permesso di aggiornamento su uno specifico record di una specifica tabella.
- `auth_event` registra le modifiche nelle altre tabelle e l'accesso tramite CRUD agli oggetti controllati da RBAC.

Non ci sono restrizioni sui nomi dei ruoli e i nomi dei permessi. Lo sviluppatore può crearli scegliendo nomi che rispecchiano i ruoli e i permessi dell'organizzazione. Una volta che questi sono stati creati web2py mette a disposizione una API per controllare se un utente è collegato, se è membro di uno specifico gruppo o è membro di un qualsiasi gruppo che ha uno specifico permesso. web2py mette a disposizione dei decoratori (basati sul login, sui ruoli e sui permessi) per limitare l'accesso a qualsiasi funzione. web2py include già alcuni permessi (per esempio quelli che hanno un nome che corrisponde ai metodi CRUD, create, read, update e delete) è può automaticamente applicarli senza l'utilizzo di alcun decoratore.

In questo capitolo saranno discusse le diverse parti dell'implementazione di RBAC in web2py.

8.1 Autenticazione

Per utilizzare RBAC l'utente deve essere identificato. Questo significa che deve essere registrato e collegato.

Auth mette a disposizione diversi metodi di accesso (*login*). Il metodo di default consiste nell'identificare l'utente in base alla tabella locale `auth_user`. In alternativa si può eseguire il login verso sistemi di autenticazione di terze parti come Google, PAM, LDAP, Facebook, Linkedin, OpenID, OAuth ed altri.

Per iniziare ad utilizzare `Auth` è necessario che in un modello sia presente il seguente codice (che è anche presente nell'applicazione *welcome*). In questo caso si ipotizza che l'oggetto di connessione al database sia chiamato `db`:

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

Impostare `username=True` se per il login si vuole utilizzare il nome utente invece che l'indirizzo email.

Per esporre `Auth` è necessario anche includere la seguente funzione in un controller (per esempio in "default.py"):

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

L'oggetto `auth` *e l'azione* `user` *sono già presenti nell'applicazione "welcome" utilizzata come base per la creazione delle nuove applicazioni.*

web2py include anche una vista d'esempio "default/user.html" per visualizzare correttamente questa funzione:

```
{{extend 'layout.html'}}
<h2>{{=request.args(0)}}</h2>
{{=form}}
{{if request.args(0)=='login':}}
<a href="{{=URL(args='register')}}" >register</a><br />
<a href="{{=URL(args='retrieve_password')}}" >lost password</a><br />
{{pass}}
```

Notare che questa funzione visualizza semplicemente un oggetto `form` e quindi può essere personalizzata utilizzando la normale sintassi dei form. L'unica particolarità è che il form visualizzato dipende dal valore di `request.args(0)` quindi potrebbe essere necessario del codice simile al seguente:

```
{{if request.args(0)=='login':}}...custom login form...{{pass}}
```

Questo controller espone diverse azioni:

```
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** consente agli utenti di registrarsi. E' integrato con *CAPTCHA*, sebbene quest'ultimo sia disabilitato per default.
- **login** consente agli utenti che sono registrati di accedere nei seguenti casi: registrazione verificata; verifica della registrazione non necessaria; registrazione approvata; registrazione che non necessita di approvazione; utente non bloccato.
- **logout** esegue lo scollegamento dell'utente dall'applicazione ed inoltre, come per gli altri metodi, può essere utilizzato per intercettare alcuni eventi.
- **profile** consente agli utenti di modificare il proprio profilo, memorizzato nella tabella `auth_user`. Questa tabella non ha una struttura fissa ma può essere personalizzata.
- **change_password** consente agli utenti di modificare la propria password in modo sicuro.
- **verify_email**. In caso di verifica della registrazione l'utente riceve una email con un link per confermare le informazioni di registrazione. Il link punta a questa azione.
- **retrieve_username**. Per default **Auth** utilizza l'indirizzo email e la password per far accedere l'utente ma può anche, opzionalmente, utilizzare il nome utente invece dell'indirizzo email. In questo caso se l'utente dimentica il suo nome utente l'azione `retrieve_username` consente all'utente di inserire il suo indirizzo email e di recuperare il nome utente con un messaggio di posta elettronica.

- **retrieve_password** consente all'utente che ha dimenticato la propria password di riceverne una nuova per email. Il nome di questa azione potrebbe generare confusione perchè questa funzione non recupera la password attuale dell'utente (che sarebbe impossibile perchè la password è memorizzata cifrata), ma ne genera una nuova.
- **impersonate** consente ad un utente di impersonarne un altro. Questo è utile per il debug e per le opzioni di supporto. `request.args[0]` è l'id dell'utente che deve essere impersonato. Questa operazione è consentita solamente se l'utente collegato ha il permesso *impersonate* (`has_permission('impersonat db.auth_user, user_id)`).
- **groups** elenca i gruppi di cui l'utente collegato è membro.
- **not_authorized** visualizza un messaggio d'errore quando l'utente tenta di eseguire un'operazione alla quale non è autorizzato.
- **navbar** è una funzione ausiliaria che genera un menu con i diversi link per **Auth** (come, per esempio, login, register, ecc.).

Le azioni *logout*, *profile*, *change_password*, *impersonate* e *groups* richiedono che l'utente sia già autenticato.

Per default tutte le azioni sono esposte ma è possibile restringere l'accesso solo ad alcune di esse. Tutti i metodi possono essere estesi o sostituiti derivando la classe **Auth**.

Per restringere l'accesso ad una funzione solo agli utenti autenticati si deve decorare la funzione come nel seguente esempio:

```
@auth.requires_login()
def hello():
    return dict(message='hello %(first_name)' % auth.user)
```

Qualsiasi funzione può essere decorata, non solo quelle esposte. Ovviamente questo è solo un esempio molto semplice di controllo d'accesso. Esempi più complessi saranno descritti successivamente in questo capitolo.

`auth.user` *contiene una copia del record di* `db.auth_user` *per l'utente colle-*

gato oppure contiene `None`. *E' anche presente* `auth.user_id` *equivalente a* `auth.user.id` *che contiene l'id dell'utente collegato oppure* `None`.

8.1.1 Limitazioni sulla registrazione

Se si vuole consentire agli utenti di registrarsi senza poter accedere fino a che la registrazione non sia stata approvata da un amministratore impostare:

```
auth.settings.registration_requires_approval = True
```

Una registrazione può essere approvata tramite l'interfaccia **appadmin**. Nella tabella `auth_user` le registrazioni in attesa hanno il campo `registration_key` impostato a "pending". Una registrazione è approvata quando questa campo è vuoto. Dall'interfaccia **appadmin** è anche possibile impedire il login di un utente impostando il relativo campo `registration_key` della tabella `auth_user` a "blocked". Gli utenti bloccati non possono più eseguire il login ma se sono già connessi quando vengono bloccati non sono forzati ad eseguire il logout.

E' possibile bloccare completamente l'accesso alla pagina di registrazione con il seguende comando:

```
auth.settings.actions_disabled.append('register')
```

Le altre azioni di **Auth** possono essere bloccate allo stesso modo.

8.1.2 Integrazione con OpenID, Facebook ed altri servizi di autenticazione

E' possibile utilizzare il sistema RBAC di web2py con un servizio esterno di autenticazione, come per esempio OpenID, Facebook, Linkedin, Google, MySpace, Fliker ed altri. Il modo più semplice di fare questo è con il servizio RPX (Janrain.com).

Janrain.com è un servizio che mette a disposizione un middleware per l'autenticazione. Dopo essersi registrati su Janrain.com ed aver registrato un dominio (il nome dell'applicazione) con le URL che si intende utilizzare Janrain.com rilascierà una chiave per le loro API.

Per utilizzare RPX il seguente codice deve essere posizionato dopo la definizione dell'oggetto `auth` nel modello dell'applicazione web2py:

```
from gluon.contrib.login_methods.rpx_account import RPXAccount
auth.settings.actions_disabled=['register','change_password','
    request_reset_password']
auth.settings.login_form = RPXAccount(request,
    api_key='...',
    domain='...',
    url = "http://localhost:8000/%s/default/user/login" % request.application)
```

La prima linea importa il nuovo metodo di login, la seconda linea disabilita la registrazione locale, la terza linea richiede a web2py di utilizzare il metodo di login di RPX. Nelle linee successive devono essere specificate la `api_key` ricevuta da Janrain.com, il nome del dominio scelto durante la registrazione e la `url` esterna della pagina di login.

[INSERT IMAGE HERE]

Quando un nuovo utente si registra per la prima volta web2py crea un nuovo record associato all'utente nella tabella `db.auth_user` ed utilizza il campo `registration_id` per memorizzare un id univoco dell'utente. La maggior parte dei metodi d'autenticazione forniscono anche un nome utente, una email, un nome ed un cognome (ma questo non è garantito). Quali campi sono forniti dipende dal metodo di login selezionato dall'utente. Se lo stesso utente si collega con due meccanismi d'autenticazione differenti (per esempio OpenID e Facebook) Janrain potrebbe non riconoscerli come il medesimo utente e potrebbe emettere due `registration_id` differenti.

Si può personalizzare il collegamento tra i dati forniti da Janrain e i dati memorizzati in `db.auth_user`. Ecco un esempio per Facebook:

```
auth.settings.login_form.mappings.Facebook = lambda profile:\
```

```
            dict(registration_id = profile["identifier"],
                 username = profile["preferredUsername"],
                 email = profile["email"],
                 first_name = profile["name"]["givenName"],
                 last_name = profile["name"]["familyName"])
```

Le chiavi nei campi del dizionario sono i campi in `db.auth_user` ed i valori sono i dati inseriti nel profilo fornito da Janrain. Vedere la documentazione online di Janrain per maggiori dettagli. Janrain mantiene anche le statistiche d'accesso per i propri utenti.

Questo form di login è completamente integrato con RBAC di web2py: si possono creare gruppi, assegnare membri ai gruppi, assegnare i permessi, bloccare gli utenti, ecc.

Se si preferisce non utilizzare Janrain ma si vuole usare un metodo differente (LDAP, PAM, Google, OpenID, OAuth/Facebook, Linkedin, ecc.) è possibile farlo. Le API per questi metoodi di login sono descritte più avanti in questo capitolo.

8.1.3 CAPTCHA e reCAPTCHA

Per evitare che gli spammer e i 'bot si registrino al sito è possibile richiedere una registrazione con CAPTCHA. web2py supporta **reCAPTCHA** (71) senza necessità di modifiche. E' stato scelto **reCAPTCHA** perchè è molto ben progettato, gratuito, accessibile (può leggere le parole agli utenti), facile da configurare e non richiede installazione di librerie di terze parti.

Per utilizzare **reCAPTCHA** è necessario eseguire le seguenti operazioni:

- Registrarsi sul sito di reCAPTCHA (71) ed ottenere una coppia di stringhe per la chiave pubblica e la chiave privata (PUBLIC_KEY, PRIVATE_KEY) per l'account.

- Aggiungere il seguente codice al modello dopo che l'oggetto `auth` è definito:

```
from gluon.tools import Recaptcha
auth.settings.captcha = Recaptcha(request,
    'PUBLIC_KEY', 'PRIVATE_KEY')
```

reCAPTCHA potrebbe non funzionare se si accede all'applicazione tramite 'localhost' o '127.0.0.1' perchè ha bisogno di siti visibili pubblicamente.

Il costruttore `Recaptcha` ha alcuni argomenti opzionali:

```
Recaptcha(..., use_ssl=True, error_message='invalid')
```

Notare che per default `use_ssl=False`

Se non si vuol utilizzare reCAPTCHA si può ispezionare il codice per la classe `Recaptcha` in "gluon/tools.py" che può essere facilmente adattato ad altri sistemi di CAPTCHA.

8.1.4 Personalizzare l'autenticazione

La chiamata a:

```
auth.define_tables()
```

definisce tutte le tabelle per **Auth** che non sono state già definite. Questo significa che è possibile definire la propria tabella `auth_user`. Utilizzando una sintassi simile a quella mostrata più sotto si può personalizzare qualsiasi altra tabella di **Auth**.

Questo è il modo corretto di definire la tabella utenti:

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

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=512,
          readable=False, label='Password'),
    Field('registration_key', length=512,
          writable=False, readable=False, default=''),
    Field('reset_password_key', length=512,
          writable=False, readable=False, default=''),
    Field('registration_id', length=512,
          writable=False, readable=False, default=''))

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()
```

Si può aggiungere qualsiasi altro campo si desideri, ma non si possono rimuovere quelli indicati in questo esempio. E' importante che i campi "password", "registration_key", "reset_password_key" e "registration_id" siano definiti con `readable=False` e `writable=False` perchè non devono essere modificati dagli utenti. Se si aggiunge un campo chiamato "username" questo sarà utilizzato al posto di "email" per il login. Se questo viene fatto è necessario aggiungere anche un validatore:

```
auth_table.username.requires = IS_NOT_IN_DB(db, auth_table.username)
```

8.1.5 Rinominare le tabelle di **Auth**

I nomi effettivi delle tabelle di **Auth** sono memorizzate 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'
```

I nomi delle tabelle possono essere cambiati riassegnando le variabili dopo che l'oggetto `auth` è stato definito e prima che le tabelle sono definite. Per esempio:

```
auth = Auth(globals(),db)
auth.settings.table_user_name = 'person'
#...
auth.define_tables()
```

Le tabelle effettive possono essere referenziate indipendentemento dal loro nome con:

```
auth.settings.table_user
auth.settings.table_group
auth.settings.table_membership
auth.settings.table_permission
auth.settings.table_event
```

8.1.6 Altri metodi di login e form di login

Auth rende disponibili diversi metodi di login (ed è possibile crearne di nuovi). Ogni metodo supportato corrisponde ad un file nella cartella:

```
gluon/contrib/login_methods/
```

Per approfondire l'utilizzo di un metodo di login fare riferimento alla documentazione nel relativo file. Qui verranno forniti solo alcuni esempi.

Prima di tutto è necessario fare una distinzione tra due famiglie di metodi di login:

- metodi di login che utilizzano il form di login di web2py (con le credenziali verificate in un servizio esterno a web2py come, per esempio, LDAP).

- metodi di login che richiedono un form esterno di single sign-on (come per esempio Google o Facebook).

Nel secondo caso le credenziali di login non sono ricevute da web2py, viene invece ricevuto un token identificativo emesso dal service provider, memorizzato in `db.auth_user.registration_id`.

Ecco alcuni esempi del primo caso:

Basic Authentication

Con un servizio d'autenticazione di tipo "basic authentication", accedendo alla URL:

```
https://basic.example.com
```

il server accetterà richieste HTTPS con un header nel formato:

```
GET /index.html HTTP/1.0
Host: basic.example.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

dove l'ultima stringa è la codifica in base64 della stringa "username:password". Il servizio risponde con 200 OK se l'utente è autorizzato e con 400, 401, 402, 403 o 404 se non lo è.

L'utente e la password possono essere inseriti utilizzando il form di login standard di `Auth` e verificati con questa URL. Quello che è necessario fare è aggiungere il seguente codice all'applicazione:

```
from gluon.contrib.login_methods.basic_auth import basic_auth
auth.settings.login_methods.append(
    basic_auth('https://basic.example.com'))
```

Notare che `auth.settings.login_methods` è una lista di metodi d'autenticazione che sono eseguiti in sequenza. Di default questa lista è impostata a:

```
auth.settings.login_methods = [auth]
```

Quando un metodo alternativo è aggiunto, per esempio `basic_auth`, **Auth** tenta prima di identificare l'utente in base al primo elemento contenuto in `auth_user` e se questo non riesce tenta il metodo successivo nella lista. Se un metodo d'autenticazione ha successo e se `auth.settings.login_methods[0]==auth`, `Auth` intraprende le seguenti azioni:

- se l'utente non esiste in `auth_user` viene creato un nuovo record per l'utente e vengono memorizzati il nome utente (o l'email) e la password.
- se l'utente esiste in `auth_user` ma la nuova password non corrisponde a quella memorizzata nel record la vecchia password è sostituita con la nuova (le password sono sempre memorizzate codificate, a meno che non sia indicato esplicitamente di non farlo). Se non si desidera memorizzare la nuova password in `auth_user` è sufficiente cambiare l'ordine dei metodi di login nella lista o rimuovere `auth` dalla lista. Per esempio:

```
from gluon.contrib.login_methods.basic_auth import basic_auth
auth.settings.login_methods = \
    [basic_auth('https://basic.example.com')]
```

Lo stesso vale per ogni altro metodo di login descritto in questo capitolo.

SMTP e Gmail

E' possibile verificare le credenziali di login utilizzando un server SMTP remoto, per esempio quello di Gmail. In questo caso l'utente è considerato autenticato se l'email e la password che inserisce nel form sono credenziali valide per accedere al server SMTP di GMail (`smtp.gmail.com:587`). Tutto quello che è necessario fare è inserire il seguente codice:

```
from gluon.contrib.login_methods.email_auth import email_auth
auth.settings.login_methods.append(
    email_auth("smtp.gmail.com:587", "@gmail.com"))
```

Il primo argomento di `email_auth` è l'indirizzo e la porta del server SMTP. Il secondo argomento è il dominio dell'email.

Questo metodo funziona con ogni server SMTP che richiede l'autenticazione TSL.

PAM

L'autenticazione di tipo PAM (*Pluggable Authentication Module*) tipica dei sistemi Linux funziona allo stesso modo e consente a web2py di autenticare gli utenti utilizzando gli account presenti sul sistema operativo:

```
from gluon.contrib.login_methods.pam_auth import pam_auth
auth.settings.login_methods.append(pam_auth())
```

LDAP

L'autenticazione tramite LDAP (*Lightweigth Directory Access Protocol*) funziona in modo simile ai casi precedenti:

Per utilizzare l'autenticazione LDAP con Active Directory di Microsoft:

```
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'))
```

Per utilizzare l'autenticazione LDAP con Lotus Notes e Domino di IBM:

```
auth.settings.login_methods.append(ldap_auth(mode='domino',
   server='my.domino.server'))
```

Per utilizzare l'autenticazione LDAP con OpenLDAP (tramite l'UID):

```
auth.settings.login_methods.append(ldap_auth(server='my.ldap.server',
   base_dn='ou=Users,dc=domain,dc=com'))
```

Per utilizzare l'autenticazione LDAP con OpenLDAP (tramite CN):

```
auth.settings.login_methods.append(ldap_auth(mode='cn',
   server='my.ldap.server', base_dn='ou=Users,dc=domain,dc=com'))
```

Google App Engine

Quando l'applicazione è eseguita su Google App Engine l'autenticazione tramite Google si comporta in modo differente: la pagina di login di web2py viene saltata e l'utente è reindirizzato alla pagina di login di Google che lo rimanda all'applicazione solo dopo che l'autenticazione è avvenuta con successo. Poichè questo comportamento è differente dai casi precedenti la API da utilizzare è leggeremente diversa:

```
from gluon.contrib.login_methods.gae_google_login import GaeGoogleAccount
auth.settings.login_form = GaeGoogleAccount()
```

OpenID

In una sezione precedente è stata discussa l'integrazione con Janrain.com (che fornisce l'accesso anche tramite OpenID) e quello è il modo più semplice di autenticarsi con OpenID in web2py. Tuttavia potrebbe non essere possibile utilizzare un servizio di terze parti per l'autenticazione, in questo caso si può accedere ad un provider di autenticazione OpenID direttamente dall'applicazione (che in questo caso si comporta da service consumer). Ecco l'esempio:

```
from gluon.contrib.login_methods.openid_auth import OpenIDAuth
auth.settings.login_form = OpenIDAuth(auth)
```

`OpenIDAUth` richiede il modulo "python-open" (installato separatamente).

Questo metodo di login definisce automaticamente la seguente tabella:

```
db.define_table('alt_logins',
    Field('username', length=512, default=''),
    Field('type', length =128, default='openid', readable=False),
    Field('user', self.table_user, readable=False))
```

che memorizza i nomi OpenID per ogni utente. Se si vuole visualizzare gli openid per l'utente collegato:

```
{{=auth.settings.login_form.list_user_openids()}}
```

OAuth2.0 e Facebook

Come nella sezione precedente Janrain.com (che fornisce l'accesso anche tramite Facebook) è il modo più semplice di autenticarsi con OAuth2.0 a Facebook in web2py. Tuttavia potrebbe non essere possibile utilizzare un servizio di terze parti per l'autenticazione, in questo caso si può accedere ad un provider di autenticazione OAuth2.0 (come Facebook) direttamente dall'applicazione. Ecco come fare:

```
from gluon.contrib.login_methods.oauth20_account import OAuthAccount
auth.settings.login_form=OAuthAccount(globals(),YOUR_CLIENT_ID,YOUR_CLIENT_SECRET)
```

Le cose diventano un po' più complesse se si vuole utilizzare l'autenticazione OAuth2.0 di Facebook per accedere dall'applicazione web2py direttamente a Facebook per utilizzare le sue API. Ecco un esempio di come accedere all'API *Graph* di Facebook:

- Prima di tutto deve essere installato il modulo "pyfacebook".
- Poi deve essere aggiunto il seguente codice nel modello:

```
# import required modules
from facebook import GraphAPI
from gluon.contrib.login_methods.oauth20_account import OAuthAccount
# extend the OAUthAccount class
class FaceBookAccount(OAuthAccount):
    """OAuth impl for FaceBook"""
    AUTH_URL="https://graph.facebook.com/oauth/authorize"
    TOKEN_URL="https://graph.facebook.com/oauth/access_token"
    def __init__(self, g):
        OAuthAccount.__init__(self, g,
                              YOUR_CLIENT_ID,
                              YOUR_CLIENT_SECRET,
                              self.AUTH_URL,
                              self.TOKEN_URL)
        self.graph = None
    # override function that fecthes user info
    def get_user(self):
        "Returns the user using the Graph API"
        if not self.accessToken():
            return None
        if not self.graph:
            self.graph = GraphAPI((self.accessToken()))
        try:
            user = self.graph.get_object("me")
```

```
            return dict(first_name = user['first_name'],
                        last_name = user['last_name'],
                        username = user['id'])
        except GraphAPIError:
            self.session.token = None
            self.graph = None
            return None
# use the above class to build a new login form
auth.settings.login_form=FaceBookAccount(globals())
```

Linkedin

Come nella sezione precedente Janrain.com (che fornisce l'accesso anche tramite Linkedin) è il modo più semplice di autenticarsi a Linkedin in web2py. Tuttavia potrebbe non essere possibile utilizzare un servizio di terze parti per l'autenticazione, in questo caso si può accedere al provider di autenticazione Linkedin direttamente dall'applicazione (in questo modo è anche possibile ottenere più informazioni rispetto a quelle recuperate tramite Janrain.com). Ecco come fare:

```
from gluon.contrib.login_methods.linkedin_account import LinkedInAccount
auth.settings.login_form=LinkedInAccount(request,KEY,SECRET,RETURN_URL)
```

`LinkedInAccount` richiede il modulo "python-linkedin" (installato separatamente).

Form multipli di login

Mentre alcuni metodi di login lasciano invariato il form di login altri lo modificano. Questi metodi non possono coesistere con altri se non fornendo form di login multipli nella stessa pagina. In web2py è possibile farlo, come è indicato nell'esempio seguente che combina il metodo standard di login (`auth`) con il metodo di login `RPX` (per Janrain.com):

```
from gluon.contrib.login_methds.extended_login_form import ExtendedLoginForm
other_form = RPXAccount(request, api_key="...", domain="...", url="...") auth.settin
= ExtendedLoginForm(request, auth, other_form, signals=['token'])
```

Se `signals` è impostato e un parametro nella richiesta corrisponde a un elemento di `signals` verrà ritornata la chiamata a `other_form.login_form`. `other_form` può gestire situazioni particolari, per esempio passaggi multipli nel login OpenID all'interno di `other_form.login_form`. Altrimenti visualizzerà il form standard di login combinato con `other_form.login_form`.

8.2 Auth e la configurazione dell'email

Per default la verifica dell'email è disabilitata. Per abilitare l'email aggiungere le seguenti linee nel modello dove è definito `auth`:

```
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'
```

E' necessario sostituire i diversi mail.settings... con i parametri corretti del server SMTP da utilizzare. Impostare `mail.settings.login=False` se il server SMTP non richiede l'autenticazione. E' anche possibile sostituire la stringa

```
'Click on the link ...'
```

in `auth.messages.verify_email` con la corretta URL dell'azione `verify_email`. Questo potrebbe essere necessario quando web2py è installato dietro un proxy e non può quindi determinare con certezza la sua URL pubblica.

Una volta che `mail` è definito può anche essere usato per inviare esplicitamente delle email con:

```
mail.send(to=['somebody@example.com'],
          subject='hello',
          message='hi there')
```

`mail` restituisce `True` se riesce a mandare l'email, altrimenti ritorna `False`.

8.2.1 Debug dell'invio delle email

Per eseguire il debug dell'invio delle email è possibile impostare:

```
mail.settings.server = 'logging'
```

in questo modo le email non saranno realmente inviate ma verranno registrate sulla console.

8.2.2 Email da Google App Engine

Per inviare email da Google App Engine:

```
mail.settings.server = 'gae'
```

Al momento della scrittura di questo manuale web2py non supporta gli allegati e le email cifrate su Google App Engine.

8.2.3 Altri esempi di email

Email con testo semplice

```
mail.send('you@example.com',
  'Message subject',
  'Plain text body of the message')
```

Email con testo in HTML

```
mail.send('you@example.com',
  'Message subject',
  '<html>html body</html>')
```

Se il corpo dell'email inizia con `<html>` e termina con `</html>` verrà inviato come una email HTML.

Combinare email con testo semplice ed HTML

Il testo del messaggio può essere una tupla (text, html):

```
mail.send('you@example.com',
  'Message subject',
  ('Plain text body', '<html>html body</html>'))
```

email con copia (CC) e copia nascosta (BCC)

```
mail.send('you@example.com',
  'Message subject',
  'Plain text body',
  cc=['other1@example.com', 'other2@example.com'],
  bcc=['other3@example.com', 'other4@example.com'])
```

Allegati

```
mail.send('you@example.com',
  'Message subject',
  '<html><img src="cid:photo" /></html>',
  attachments = Mail.Attachment('/path/to/photo.jpg' content_id='photo'))
```

Allegati multipli

```
mail.send('you@example.com,
  'Message subject',
  'Message body',
  attachments = [Mail.Attachment('/path/to/fist.file'),
                 Mail.Attachment('/path/to/second.file')])
```

8.2.4 Email cifrate con PGP e X509

E' possibile inviare email cifrate con X509 (SMIME) utilizzando le seguenti impostazioni:

```
mail.settings.cipher_type = 'x509'
mail.settings.sign = True
mail.settings.sign_passphrase = 'your passphrase'
mail.settings.encrypt = True
mail.settings.x509_sign_keyfile = 'filename.key'
mail.settings.x509_sign_certfile = 'filename.cert'
mail.settings.x509_crypt_certfiles = 'filename.cert'
```

E' possibile inviare email cifrate con PGP utilizzando le seguenti impostazioni:

```
from gpgme import pgp
mail.settings.cipher_type = 'gpg'
mail.settings.sign = True
mail.settings.sign_passphrase = 'your passphrase'
mail.settings.encrypt = True
```

L'invio di email con PGP richiede la presenza del modulo python-pyme.

8.3 Autorizzazione

Quando un nuovo utente si registra viene creato un nuovo gruppo contenente l'utente. Il ruolo del nuovo utente è convenzionalmente "user_[id]" dove [id] è l'id del nuovo utente. La creazione del gruppo può essere disabilitata con:

```
auth.settings.create_user_groups = False
```

sebbene non sia consigliato farlo.

Gli utenti possono essere membri dei gruppi. Ogni gruppo è identificato da un nome/ruolo. I gruppi hanno dei permessi che vengono ereditati dagli utenti a seconda dei gruppi a cui appartengono. E' possibile creare gruppi, indicare l'appartenenza di un utente ad un gruppo e assegnare permessi con l'applicazione **appadmin** o da codice utilizzando i seguenti metodi:

```
auth.add_group('role', 'description')
```

che ritorna l'id del nuovo gruppo appena creato.

```
auth.del_group(group_id)
```

cancella il gruppo con id `group_id`.

```

auth.del_group(auth.id_group('user_7'))
```

cancella il gruppo con ruolo "user_7", cioè il gruppo univoco associato all'utente con id 7.

```
auth.user_group(user_id)
```

ritorna l'id del gruppo univocamente associato all'utente identificato da `user_id`.

```
auth.add_membership(group_id, user_id)
```

assegna l'utente con id `user_id` al gruppo con id `group_id`. Se `user_id` non è specificato web2py presuppone che si tratti dell'utente collegato.

```
auth.del_membership(group_id, user_id)
```

rimuove l'utente con id `user_id` dai membri del gruppo con id `group_id`. Se `user_id` non è specificato web2py presuppone che si tratti dell'utente collegato.

```
auth.has_membership(group_id, user_id, role)
```

controlla se l'utente con id `user_id` appartiene al gruppo con id `group_id` o al gruppo con il ruolo specificato. Solamente uno tra `group_id` e `role` deve essere passato alla funzione. Se `user_id` non è specificato web2py presuppone che si tratti dell'utente collegato.

```
auth.add_permission(group_id, 'name', 'object', record_id)
```

assegna il permesso "name" (definito dall'utente) sull'oggetto "object" (anch'esso definito dall'utente) ai membri del gruppo con id `group_id`. Se "object" è un nome di tabella allora il permesso si riferisce all'intera tabella (`record_id==0`) o ad uno specifico record (`record_id>0`). Quando si danno i permessi sulle

tabelle è normale utilizzare i seguenti nomi dei permessi: 'create', 'read', 'update', 'delete', 'select' poichè questi permessi sono automaticamente gestiti da CRUD.

```
auth.del_permission(group_id, 'name', 'object', record_id)
```

revoca il permesso.

```
auth.has_permission('name', 'object', record_id, user_id)
```

controlla se l'utente identificato da `user_id` appartiene ad un gruppo con il permesso richiesto.

```
rows = db(accessible_query('read', db.sometable, user_id))\
    .select(db.mytable.ALL)
```

restituisce tutte le righe della tabella "sometable" su cui l'utente con id `user_id` ha il permesso "read". Se `user_id` non è specificato web2py presuppone che si tratti dell'utente collegato. La query `accessible_query( ... )` può essere combinata con altre query per crearne di più complesse. `accessible_query( ... )` è l'unico metodo di **Auth** che richiede una JOIN e quindi non funziona su Google App Engine.

Con la seguente definizione:

```
>>> 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')
```

Questo è un esempio di utilizzo della funzione `auth.has_permission`:

```
>>> 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
```

8.3.1 *Decoratori*

Il modo più semplice per controllare i permessi non è tramite la chiamata esplicita dei metodi elencati nella sezione precedente ma è l'utilizzo dei decoratori delle funzioni. In questo modo i permessi sono controllati sempre per l'utente collegato. Ecco alcuni esempi:

```
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)
```

L'accesso a tutte le funzioni oltre la prima è ristretto in base ai permessi che ha l'utente collegato. Se l'utente collegato non ha eseguito il login allora i permessi non possono essere controllati. In questo caso l'utente è reindirizzato verso la pagina di login e poi riportato alla pagina che richiede i permessi. Se l'utente collegato non dispone dei permessi per accedere ad una specifica funzione questo è reindirizzato alla URL definita da:

```
auth.settings.on_failed_authorization = \
    URL('user',args='on_failed_authorization')
```

Questa variabile può essere modificata per reindirizzare l'utente su un'altra pagina.

8.3.2 *Combinare i vincoli*

A volte potrebbe essere necessario combinare i vincoli richiesti. Questo può essere fatto con il generico decoratore `requires` che richiede un singolo argomento, una condizione True o False. Per esempio, per dare accesso agli utenti del gruppo "agents" ma solo di lunedì:

```
@auth.requires(auth.has_membership(group_id=agents) \
               and request.now.weekday()==1)
def function_seven():
    return 'Hello agent, it must be Tuesday!'
```

oppure:

```
@auth.requires(auth.has_membership(role='Secret Agent') \
               and request.now.weekday()==1)
def function_seven():
    return 'Hello agent, it must be Tuesday!'
```

8.3.3 *Autorizzazione e CRUD*

L'utilizzo dei decoratori e/o dei controlli espliciti con le funzioni sopra elencate è un modo per implementare un sistema di controllo d'accesso. Un altro modo per implementarlo è quello di utilizzare sempre CRUD (invece che SQLFORM) per accedere al database e richiedere che CRUD verifichi il controllo d'accesso sulle tabelle e sui record del database. Questo può essere fatto collegando `Auth` e CRUD con il seguente comando:

```
crud.settings.auth = auth
```

In questo modo l'utente non potrà accedere a nessuna funzione CRUD a meno che non sia identificato ed abbia un accesso esplicito. Per esempio, per consentire ad un utente di inserire nuovi commenti e di aggiornare solo quelli da lui inseriti (presupponendo che `crud`, `auth` e `db.comment` siano definiti):

```
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)
```

E' possibile anche selezionare specifici record (quelli che hanno il permesso 'read'):

```
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)
```

8.3.4 Autorizzazione e scarico dei file

L'utilizzo dei decoratori o l'uso di `crud.settings.auth` non controlla l'autorizzazione sui file scaricati con la normale funzione `download`:

```
def download(): return response.download(request, db)
```

Per effettuare i controlli sui download si deve dichiarare esplicitamente quali campi di "upload" contengono i file che devono essere sottoposti al controllo durante il download. Per esempio:

```
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)
```

L'attributo `authorization` dei campi di upload può essere `None` (il default) oppure una funzione che decide se l'utente collegato ha il permesso di leggere ('read') il record corrente. In questo esempio non c'è nessuna restrizione per il download delle immagini collegate dal campo "small_image" ma è richiesto il controllo d'accesso per le immagini collegate dal campo "large_image".

8.3.5 Controllo d'accesso e Basic Authentication

Potrebbe essere necessario esporre come servizi alcune azioni che richiedono il controllo d'accesso tramite dei decoratori. Per esempio per chiamare tali azioni da un programma o da uno script ed essere ancora in grado di utilizzare l'autenticazione per il controllo dell'autorizzazione. Poichè **Auth** consente il login tramite la Basic Authentication:

```
auth.settings.allow_basic_authentication = True
```

un'azione di questo tipo:

```
@auth.requires_login()
def give_me_time():
    import time
    return time.ctime()
```

può essere chiamata, per esempio, dalla linea di comando:

```
wget --user=[username] --password=[password]
    http://.../[app]/[controller]/give_me_time
```

La Basic Authentication è spesso l'unica opzione per i servizi (descritti nel prossimo capitolo) ma è disabilitata per default.

8.3.6 Impostazioni e messaggi

Ecco una lista di tutti i parametri che possono essere personalizzati per **Auth**:

```
auth.settings.actions_disabled = []
```

Indica quali azioni devono essere disabilitate, per esempio ['register'].

```
auth.settings.registration_requires_verification = False
```

Impostare a `True` per far sì che durante la registrazione dei nuovi utenti questi ricevano una email di verifica per completare la registrazione tramite un link di risposta.

```
auth.settings.registration_requires_approval = False
```

Impostare a `True` per far sì che i nuovi utenti non possano collegarsi fino a quando non siano stati approvati. L'approvazione è eseguita impostando a '' (campo vuoto) il campo `registration_key` nel record relativo al nuovo utente, tramite appadmin o da programma.

```
auth.settings.create_user_groups = True
```

Impostare a `False` se non si vuole che venga creato automaticamente un gruppo per ogni utente registrato.

```
auth.settings.login_url = URL('user', args='login')
```

Indica a web2py la URL della pagina di login.

```
auth.settings.logged_url = URL('user', args='profile')
```

Se l'utente tenta di accedere alla pagina di registrazione quando è già autenticato viene reindirizzato a questa URL.

```
auth.settings.download_url = URL('download')
```

Indica a web2py la URL per il download dei documenti caricati. E' necessaria per creare la pagina di profilo dell'utente in caso che contenga dei campi di upload, come, per esempio, la foto dell'utente.

```
auth.settings.mailer = None
```

Deve puntare ad un oggetto con un metodo di invio con la stessa sintassi di `gluon.tools.Mail.send`.

```
auth.settings.captcha = None
```

Deve puntare ad un oggetto con un metodo di invio con la stessa sintassi di `gluon.tools.Recaptcha`.

```
auth.settings.expiration = 3600  # seconds
```

La scadenza di una sessione di login in secondi.

```
auth.settings.on_failed_authorization = \
    URL('user',args='on_failed_authorization')
```

La URL a cui si è reindirizzati dopo una autorizzazione fallita.

```
auth.settings.password_field = 'password'
```

Il nome del campo password memorizzato nel database. L'unico motivo per cui si potrebbe voler cambiare questo campo è quando "password" è una parola chiave riservata per il database e non può quindi essere usata come nome di un campo (come nel caso del database Firebird).

```
auth.settings.showid = False
```

Indica se la pagina di profilo deve mostrare l'id dell'utente.

```
auth.settings.login_next = URL('index')
```

Per default dopo un login avvenuto con successo la pagina di login reindirizza l'utente alla pagina chiamante (solo se quest'ultima richiedeva il login). Se non vi è una pagina chiamante l'utente è reindirizzato alla URL indicata in questa variabile.

```
auth.settings.login_onvalidation = None
```

Funzione da chiamare dopo la validazione del login ma prima del login effettivo. La funzione deve avere un solo argomento che corrisponde all'oggetto form.

```
auth.settings.login_onaccept = None
```

Funzione da chiamare dopo il login ma prima del reindirizzamento dell'utente. La funzione deve avere un solo argomento che corrisponde all'oggetto form.

```
auth.settings.login_methods = [auth]
```

Determina i metodi alternativi di login, come discusso precedentemente in questo stesso capitolo.

```
auth.settings.login_form = auth
```

Imposta un form di login alternativo per il single sign-on, come discusso precedentemente in questo stesso capitolo.

```
auth.settings.allows_basic_auth = False
```

Se impostato a `True` permette di richiamare le azioni che hanno un decoratore per il controllo d'accesso previa verifica dell'utente tramite la Basic Authentication.

```
auth.settings.logout_next = URL('index')
```

La URL a cui si è reindirizzati dopo il logout.

```
auth.settings.register_next = URL('user', args='login')
```

La URL a cui si è reindirizzati dopo la registrazione.

```
auth.settings.register_onvalidation = None
```

Funzione da richiamare dopo la validazione del form di registrazione ma prima che venga effettuata l'effettiva registrazione e prima che venga inviata una email di verifica. La funzione deve avere un singolo argomento che corrisponde all'oggetto form.

```
auth.settings.register_onaccept = None
```

Funzione da richiamare dopo la registrazione ma prima del reindirizzamento. La funzione deve avere un singolo argomento che corrisponde all'oggetto form.

```
auth.settings.verify_email_next = \
    URL('user', args='login')
```

La URL a cui reindirizzare un utente dopo la verifica dell'indirizzo email.

```
auth.settings.verify_email_onaccept = None
```

Funzione da richiamare dopo aver completato la verifica della email ma prima del reindirizzamento. La funzione deve avere un singolo argomento che corrisponde all'oggetto form.

```
auth.settings.profile_next = URL('index')
```

La URL a cui reindirizzare gli utenti dopo che hanno modificato il loro profilo.

```
auth.settings.retrieve_username_next = URL('index')
```

La URL a cui reindirizzare gli utenti dopo che hanno richiesto il recupero del loro nome utente.

```
auth.settings.retrieve_password_next = URL('index')
```

La URL a cui reindirizzare gli utenti dopo che hanno richiesto il recupero della loro password.

```
auth.settings.change_password_next = URL('index')
```

La URL a cui reindirizzare gli utenti dopo che hanno richiesto una nuova password per email.

E' anche possibile personalizzare i seguenti messaggi:

```
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".

8.4 CAS, Central Authentication Service

web2py fornisce supporto per l'autenticazione e l'autorizzazione tramite *appliance* (applicazioni già pronte). In questa sezione è illustrata l'appliance **cas** (Central Authentication Service). Notare che al momento della scrittura di questo manuale **cas** non funziona con **Auth**. Questo cambierà in futuro.

CAS è un protocollo aperto per l'autenticazione distribuita e funziona nel seguente modo: quando un utente raggiunge il sito web l'applicazione controlla nella sessione se l'utente è già autenticato (per esempio verifica la presenza di un oggetto `session.token` valido). Se l'utente non è autenticato il controller reindirizza l'utente all'appliance **cas** dove l'utente può registrarsi e gestire le sue credenziali (nome, email e password). Se l'utente si registra riceve una email di verifica e la registrazione non è completa fino a quando risponde alla email. Una volta che l'utente si è registrato con successo ed è acceduto l'appliance reindirizza l'utente all'applicazione originaria insieme con una chiave. L'applicazione utilizza la chiave per recuperare le credenziali dell'utente con una richiesta HTTP in background all'appliance **cas**.

Con questo meccanismo più applicazioni possono usare un servizio di single sign-on tramite un unica appliance **cas**. Il server che fornisce l'autenticazione è chiamato *service provider*. Le applicazioni che cercano di autenticare gli

utenti sono chiamate *service consumer*.

CAS è simile ad OpenID con una grande differenza. Nel caso di OpenID l'utente può scegliere il service provider, nel caso di CAS l'applicazione fa questa scelta, rendendo CAS un meccanismo più sicuro.

In web2py è possibile utilizzare CAS come service provider, service consumer o ambedue (in una o più applicazioni). Per utilizzare CAS come un service consumer è necessario scaricare il file:

```
https://www.web2py.com/cas/static/cas.py
```

e memorizzarlo come un modello chiamato "cas.py". Si deve poi modificare il controller che ha bisogno dell'autenticazione (per esempio "default.py") ed aggiungere all'inizio il seguente codice:

```
CAS.login_url='https://www.web2py.com/cas/cas/login'
CAS.check_url='https://www.web2py.com/cas/cas/check'
CAS.logout_url='https://www.web2py.com/cas/cas/logout'
CAS.my_url='http://127.0.0.1:8000/myapp/default/login'

if not session.token and not request.function=='login':
    redirect(URL('login'))
def login():
    session.token=CAS.login(request)
    id,email,name=session.token
    return dict()
def logout():
    session.token=None
    CAS.logout()
```

E' necesasrio modificare gli attributi dell'oggetto CAS che per default puntano al service provider CAS che è in funzione su "https://mdp.cti.depaul.edu". Questo servizio è fornito solamente per eseguire il test. L'attributo `CAS.my_url` deve essere la URL completa dell'azione di login definita nell'applicazione. Il service provider CAS ha bisogno di reindirizzare il browser del'utente a questa azione.

Questo service provider CAS restituisce un token che contiene una tupla (id, email, nome) dove id è il record univoco dell'utente (come assegnato dal

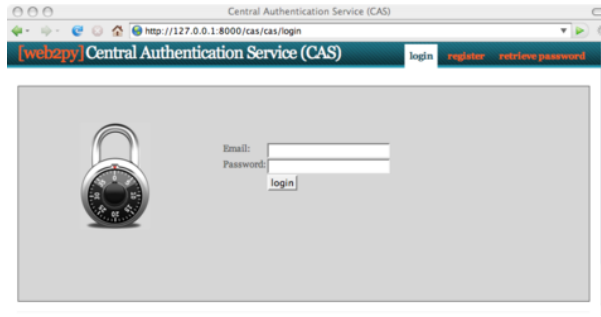
database del service provider), email è l'indirizzo email dell'utente (come dichiarato dall'utente durante la registrazione al service provider e da questo verificato) e nome è il nome dell'utente (scelto dall'utente stesso, senza garanzie che sia un nome reale). Se si visita l'URL di login dell'applicazione:

```
/myapp/default/login
```

si viene reindirizzati alla pagina di login del service provider CAS:

```
https://mdp.cti.depaul.edu/cas/cas/login
```

che avrà un aspetto simile a questo:



E' anche possibile utilizzare service provider CAS di terze parti ma in questo caso è necessario modificare le dieci linee di codice indicate precedentemente poichè service provider differenti possono tornare token contenenti valori differenti. Controllare la documentazione del service provider CAS a cui si vuol accedere per i dettagli. La maggior parte dei servizi ritorna solamente (id, nome).

Dopo aver effettuato con successo il login si viene reindirizzati all'azione di login dell'aplicazione. La vista dell'azione di login è eseguita solo dopo che il login CAS ha avuto esito positivo.

Si può scaricare l'appliance **cas** per il service provider CAS da ref. (33) ed eseguirlo direttamente. In questo caso si devono modificare anche le prime

linee del modello "email.py" dell'appliance in modo che punti al server SMTP corretto.

E' anche possibile unire i file dell'appliance **cas** con quelli della propria applicazione (models con models, ecc.) purchè non vi siano conflitti di nome di file.

9

Servizi

Il W3C (*World Wide Web Consortium*) definisce un web service come "un sistema software progettato per supportare l'interazione interoperabile macchina-macchina su una rete". Questa è una definizione molto ampia e racchiude un gran numero di protocolli progettati non per la comunicazione tra uomo e macchina ma tra macchina e macchina (come XML, JSON, RSS, ecc). web2py rende immediatamente disponibile il supporto per molti protocolli: XML, JSON, RSS, CSV, XMLRPC, JSONRPC, AMFRPC e SOAP. Inoltre può essere esteso per supportare ulteriori protocolli.

Ciascuno di questi protocolli è supportato in più modi che si possono dividere in due tipologie:

- Produrre l'output di una funzione in un dato formato (per esempio XML, JSON, RSS e CSV)

- Chiamate RPC (*Remote Procedure Calls*) (per esempio XMLRPC, JSONRPC e AMFRPC)

9.1 Riprodurre l'output di un dizionario

9.1.1 HTML, XML e JSON

La seguente azione:

```
def count():
    session.counter = (session.counter or 0) + 1
    return dict(counter=session.counter, now=request.now)
```

ritorna un contatore che è incrementato di uno quando un utente ricarica la pagina. Ritorna anche un timestamp del momento della richiesta della pagina corrente.

Normalmente questa pagina è acceduta con:

```
http://127.0.0.1:8000/app/default/count
```

e riprodotta come HTML. Senza dover scrivere ulteriore codice si può richiedere a web2py di riproduttre la medesima pagina utilizzando un protocollo differente semplicemente aggiungendo la corretta estensione alla 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
```

Il dizionario ritornato dall'azione sarà riprodotto in HTML, XML e JSON.

Ecco l'output in XML:

```
<document>
   <counter>3</counter>
   <now>2009-08-01 13:00:00</now>
</document>
```

Ecco l'output in JSON:

```
{ 'counter':3, 'now':'2009-08-01 13:00:00' }
```

Notare che gli oggetti di tipo date, time e datetime sono riprodotti come stringhe in formato ISO. Questo non è parte dello standard JSON ma è una convenzione di web2py.

9.1.2 *Viste generiche*

Quando, per esempio, è chiamata un'azione del controller "default" con l'estensione ".xml" web2py cerca un file di template chiamato "default/count.xml" e se non la trova cerca un template dal nome "generic.xml". I file "generic.html, "generic.xml", "generic.json" sono forniti in ogni nuova applicazione generata dall'applicazione base "welcome".

Altre estensioni possono essere facilmente definite dall'utente.

Non è necessaria alcuna operazione per abilitare questo comportamento in web2py. Per utilizzarlo in una vecchia installazione di web2py è sufficiente copiare i file "generic.*" dalla applicazione "welcome" (dalla versione 1.60 in poi).

Ecco il codice di "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>
```

Questo è il codice per "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')
}}
```

E questo è il codice per "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')
}}
```

Ogni dizionario può essere riprodotto in HTML, XML e JSON purchè contenga solo tipi primitivi di Python (interi, virgola mobile, stringhe, liste e tuple). `response._vars` contiene il dizionario ritornato dall'azione.

Se il dizionario contiene altri oggetti definiti dall'utente o specifici di web2py questi devono essere riprodotti utilizzando una vista personalizzata.

9.1.3 Riprodurre le righe di una SELECT

Se è necessario riprodurre in XML, JSON o in un altro formato un gruppo di righe restituite da una SELECT è necessario trasformare l'oggetto Rows in una lista di dizionari utilizzando il metodo `as_list()`.

Considerando, per esempio, il modello:

```
db.define_table('person', Field('name'))
```

La seguente azione può essere riprodotta in HTML ma non in XML o JSON:

```
def everybody():
    people = db().select(db.person.ALL)
    return dict(people=people)
```

Mentre la seguente azione può essere riprodotta in XML e JSON:

```
def everybody():
    people = db().select(db.person.ALL).as_list()
    return dict(people=people)
```

9.1.4 Formati personalizzati

Se, per esempio, si volesse riprodurre un'azione come un oggetto Pickle di Python:

```
http://127.0.0.1:8000/app/default/count.pickle
```

è sufficiente creare una nuova vista con nome "default/count.pickle" che contiene:

```
{{
import cPickle
response.headers['Content-Type'] = 'application/python.pickle'
response.write(cPickle.dumps(response._vars),escape=False)
}}
```

Se si vuole avere la possibilità di riprodurre un oggetto pickle in qualsiasi azione è sufficiente salvare il precedente file con il nome "generic.pickle".

Non tutti gli oggetti possono essere trasformati in oggetti Pickle di Python e non tutti gli oggetti trasformati con Pickle possono essere riconvertiti al loro tipo originale, è quindi buona regola limitarsi agli oggetti primitivi di Python e alle loro combinazioni. Solitamente gli oggetti che non contengono riferimenti agli stream delle connessioni di database sono trasformabili in oggetti Pickle, ma possono essere ritrasformati nel loro tipo originario solamente in un ambiente in cui le classi degli oggetti in essi contenuti sono già definite.

9.1.5 RSS

web2py include una vista "generic.rss" che può riprodurre come feed RSS il dizionario ritornato dall'azione. Poichè i feed RSS hanno una struttura fissa (titolo, link, descrizione, oggetti, ecc.) il dizionario ritornato dall'azione deve avere la seguente struttura:

```
{'title'      : '',
 'link'       : '',
 'description': '',
 'created_on' : '',
 'entries'    : []}
```

e ciascun oggetto di `entries` deve avere una struttura simile:

```
{'title'      : '',
 'link'       : '',
 'description': '',
 'created_on' : ''}
```

Per esempio la seguente azione può essere riprodotta come un feed RSS:

```
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")
                ])
```

accedendo alla URL:

```
http://127.0.0.1:8000/app/default/feed.rss
```

In alternativa, con il seguente modello:

```
db.define_table('rss_entry',
    Field('title'),
    Field('link'),
    Field('created_on','datetime'),
    Field('description'))
```

la seguente azione può essere riprodotta come feed RSS:

```
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())
```

Il metodo `as_list()` di un oggetto Rows converte le righe della tabella in una lista di dizionari. In aggiunta gli oggetti del dizionario che hanno nomi di chiavi non esplicitamente indicate nella precedente lista sono ignorati.

Questa è la vista generica "generic.rss" fornita con 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')
}}
```

Ecco ancora un esempio di un'applicazione RSS: un aggregatore che raccoglie dati dal feed di "slashdot" e lo restituisce come un feed di web2py.

```
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])
```

Può essere acceduto con:

```
http://127.0.0.1:8000/app/default/aggregator.rss
```

9.1.6 CSV

Il formato CSV (*Comma Separated Values*, Valori separati da virgola) è un protocollo usato per rappresentare dati tabellari.

Dato il seguente modello:

```
db.define_model('animal',
    Field('species'),
    Field('genus'),
    Field('family'))
```

e la seguente azione:

```
def animals():
    animals = db().select(db.animal.ALL)
    return dict(animals=animals)
```

per ottenere un output in CSV si deve definire una vista personalizzata "default/animals.csv" che serializza la tabella animal in un CSV (web2py non fornisce una vista "generic.csv"). Ecco una possibile implementazione della vista:

```
{{
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)
}}
```

Notare che sarebbe possibile definire una vista generica "generic.csv" ma dovrebbe essere specificato il nome dell'oggetto da serializzare ("animals" nel caso dell'esempio precedente). Questo è il motivo per cui non è presente una vista generica "generic.csv".

9.2 *Remote Procedure Calls (RPC)*

web2py dispone di un meccanismo per trasformare qualsiasi funzione in un *web service*. Questo meccanismo differisce da quelli descritti precedentemente in quanto:

- La funzione può avere argomenti.
- La funzione può essere definita in un modello o in un modulo invece che in un controller.
- Si può specificare in dettaglio quali siano i metodi che devono essere supportati.
- Si può obbligare una convenzione di naming dell URL più stringente.
- E' più versatile del precedente metodo perchè funziona con un gruppo prefissato di protocolli. Per lo stesso motivo non è facilmente estendibile.

Per usare questo meccanismo:

Per prima cosa si deve importare ed istanziare un oggetto "Service":

```
from gluon.tools import Service
service = Service(globals())
```

Questo codice è già presente nel modello "db.py" dell'applicazione "welcome" usata come base per le nuove applicazioni.

Come seconda cosa si deve esporre il gestiore del servizio in un controller:

```
def call():
    session.forget()
    return service()
```

Anche questo codice è già presente nel controller "default.py" delle nuove applicazioni generate da "welcome". Rimuovere `session.forget()` *se si intende utilizzare i cookie di sessione con i servizi.*

Per ultima cosa, le funzioni che devono essere esposte come servizi devono essere decorate. Ecco una lista dei decoratori supportati:

```
@service.run
@service.xml
@service.json
@service.rss
@service.csv
@service.xmlrpc
@service.jsonrpc
@service.amfrpc3('domain')
```

Come esempio considerare la seguente funzione:

```
@service.run
def concat(a,b):
    return a+b
```

Questa funzione può essere definita in un modello o in un controller e può essere chiamata remotamente in due modi differenti:

```
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 ambedue i casi la richiesta HTTP ritornerà:

```
helloworld
```

Se fosse stato utilizzato il decoratore `@service.xml` la funzione sarebbe stata chiamata con:

```
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
```

e l'output ritornato in XML sarebbe stato:

```
<document>
   <result>helloworld</result>
</document>
```

L'output della funzione può essere serializzato anche se è un oggetto Rows del DAL. In questo caso la chiamata al metodo `as_list()` è automatica.

Se fosse stato usato il decoratore `@service.json` la funzione sarebbe stata chiamata tramite:

```
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
```

e l'output sarebbe stato restituito come JSON.

Se fosse stato utilizzato il decoratore `@service.csv` il gestore del servizio avrebbe richiesto come valore di ritorno un oggetto iterabile contenente oggetti iterabili (come una lista di liste). Ecco un esempio:

```
@service.csv
def table1(a,b):
    return [[a,b],[1,2]]
```

Questo servizio può essere chiamato accedendo ad una delle seguenti URL:

```
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
```

e restituisce:

```
hello,world
1,2
```

Il decoratore `@service.rss` si aspetta un valore di ritorno dello stesso formato di quello richiesto dalla vista "generic.rss" discussa nella precedente sezione.

Per ogni funzione è possibile definire decoratori multipli.

Tutto quello discusso finora in questa sezione è semplicemente un'alternativa al metodo descritto nella precedente sezione. Le vere potenzialità dell'oggetto Service vengono sfruttate con XMLRPC, JSONRPC e AMFRPC.

9.2.1 XMLRPC

Con il seguente codice nel controller "default.py":

```
@service.xmlrpc
def add(a,b):
    return a+b

@service.xmlrpc
def div(a,b):
    return a+b
```

E' possibile eseguire in uno shell Python:

```
>>> 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
```

Il modulo di Python `xmlrpclib` fornisce un client per il protocollo XMLRPC. web2py si comporta da server.

Il client si collega al server tramite `ServerProxy` e può chiamare remotamente le funzioni decorate nel server. I dati (a, b) sono passati alla funzione non tramite variabili GET o POST, ma correttamente codificati nel corpo della richiesta utilizzando il protocollo XMLRPC e perciò sono in grado di mantenere le informazioni sul tipo (intero, stringhe, ecc.). Lo stesso è valido per il valore o i valori di ritorno. Inoltre qualsiasi eccezione che avviene sul server si propaga al client.

Esistono librerie XMLRPC per molti linguaggi di programmazione (incluso C, C++, Java, C#, Ruby e Perl) e possono interoperare tra di loro. Questo è uno dei migliori metodi per creare applicazioni che parlano l'una con l'altra,

indipendentemente dal linguaggio di programmazione.

Il client XMLRPC può anche essere implementato all'interno di un'azione di web2py. In questo modo un'azione può scambiare informazioni con un'altra applicazione di web2py (anche nella stessa installazione) utilizzando XML-RPC. Si deve però fare attenzione al rischio di *deadlock*. Se un azione chiama con XMLRPC una funzione nella stessa applicazione il chiamante deve rilasciare il lock di sessione prima della chiamata:

```
session.forget()
session._unlock(response)
```

9.2.2 *JSONRPC e Pyjamas*

JSONRPC è molto simile a XMLRPC ma per codificare i dati utilizza un protocollo basato sulla sintassi JSON invece di XML. Per descrivere la sua applicazione sarà discusso l'utilizzo di Pyjamas con web2py. Pyjamas è una trasposizione in Python del *Google Web Toolkit* originariamente scritto in Java. Pyjamas consente di scrivere un'applicazione client in Python che verrà poi tradotta in Javascript. web2py distribuisce il codice Javascript e comunica con esso tramite richieste Ajax originate dal client e intercettate dalle azioni dell'utente.

Qui è descritto come far funzionare Pyjamas con web2py. Non è richiesta nessuna libreria aggiuntiva oltre a web2py e Pyjamas.

Verrà costruita una semplice applicazione "To Do" con un client in Pyjamas (tutto in Javascript) che parla al server esclusivamente tramite JSONRPC.

Ecco come fare:

Per prima cosa, creare una nuova applicazione chiamata "todo". Come seconda cosa aggiungere il seguente codice in "models/db.py":

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

```
    db.define_table('todo', Field('task'))

    from gluon.tools import Service     # import rpc services
    service = Service(globals())
```

Come terza operazione inserire il seguente codice in "controllers/default.py":

```
    def index():
    redirect(URL('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()
```

Il significato do ogni funzione dovrebbe essere evidente.

Come quarta operazione aggiungere in "views/default/todoApp.html" il seguente codice:

```
<html>
  <head>
    <meta name="pygwt:module"
     content="{{=URL('static','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('static','output/pygwt.js')}}">
    </script>
  </body>
</html>
```

Questa viste esegue semplicemente il codice Pyjamas in "static/output/todoapp" (questo codice ancora non è stato scritto).

Come quinta operazione aggiungere in "static/TodoApp.py" (attenzione, TodoApp, non todoApp), il seguente codice:

```
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)
```

```
28          panel.add(self.Status)
29          self.remote.getTasks(self)
30
31          RootPanel().add(panel)
32
33      def onKeyUp(self, sender, keyCode, modifiers):
34          pass
35
36      def onKeyDown(self, sender, keyCode, modifiers):
37          pass
38
39      def onKeyPress(self, sender, keyCode, modifiers):
40          """
41          This function handles the onKeyPress event, and will add the
42          item in the text box to the list when the user presses the
43          enter key. In the future, this method will also handle the
44          auto complete feature.
45          """
46          if keyCode == KeyboardListener.KEY_ENTER and \
47             sender == self.todoTextBox:
48              id = self.remote.addTask(sender.getText(),self)
49              sender.setText("")
50              if id<0:
51                  RootPanel().add(HTML("Server Error or Invalid Response"))
52
53      def onClick(self, sender):
54          id = self.remote.deleteTask(
55                  sender.getValue(sender.getSelectedIndex()),self)
56          if id<0:
57              RootPanel().add(
58                  HTML("Server Error or Invalid Response"))
59
60      def onRemoteResponse(self, response, request_info):
61          self.todoList.clear()
62          for task in response:
63              self.todoList.addItem(task[0])
64              self.todoList.setValue(self.todoList.getItemCount()-1,
65                                     task[1])
66
67      def onRemoteError(self, code, message, request_info):
68          self.Status.setText("Server Error or Invalid Response: " \
69                              + "ERROR " + code + " - " + message)
70
71  class DataService(JSONProxy):
72      def __init__(self):
73          JSONProxy.__init__(self, "../../default/call/jsonrpc",
74                             ["getTasks", "addTask","deleteTask"])
75
76  if __name__ == '__main__':
```

```
77     app = TodoApp()
78     app.onModuleLoad()
```

Come sesta opearzione eseguire Pyjamas prima di avviare l'applicazione:

```
1 cd /path/to/todo/static/
2 python /python/pyjamas-0.5p1/bin/pyjsbuild TodoApp.py
```

Questo tradurrà il codice Python in Javascript in modo che possa essere eseguito dal browser. per accedere all'applicazione, visitare l'URL:

```
1 http://127.0.0.1:8000/todo/default/todoApp
```

Questa sotto-sezione è stata creata da Chris Prinos con l'aiuto di Luke Kenneth Casson Leighton (creatori di Pyjamas) ed aggiornata da Alexei Vinidiktov. E' stato testato con Pyjamas 0.5p1. L'esempio è stato ispirato da questa pagina di Django (74).

9.2.3 AMFRPC

AMFRPC è il protocollo RPC utilizzato dai client Flash per comunicare con un server. web2py supporta AMFRPC ma richiede che web2py sia eseguito dal sorgente e che sia pre-installata la libreria PyAMF. Questa libreria può essere installata dallo shell Linux o Windows con:

```
1 easy_install pyamf
```

(consultare la pagina della documentazione di PyAMF per maggiori dettagli).

In questa sotto-sezione si suppone di aver conoscenza della programmazione ActionScript.

Verrà creato un semplice servizio che richiede due valori numerici, li somma e ritorna il valore della somma. L'applicazione web2py sarà chiamata "pyamf_test" e il servizio sarà chiamato `addNumbers`.

Per prima cosa usando Adobe Flash (qualsiasi versione a partire da MX 2004) creare l'applicazione Flash client creando un nuovo file Flash FLA. Nel primo frame del file aggiungere le seguenti linee:

```
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();
```

Questo codice conesente al client Flash di connettersi ad un servizio che corrisponde ad una funzione chiamata "addNumbers" nel file "/pyamf_test/default/gatew
Si deve anche importare le classi di *remoting* di ActionScript versione 2 MX per abilitare il Remoting in Flash. Aggiungere il path a queste classi nelle impostazioni del *classpath* nell'IDE di Adobe Flash oppure posizionare la cartella "mx" a fianco del nuovo file appena creato.

Notare gli argomenti del costruttore Service. Il primo argomento è la URL corrispondente al servizio che si vuole creare. Il terzo argomento è il dominio del servizio. In questo caso è stato scelto il dominio "mydomain".

Per seconda cosa, creare un campo dinamico di testo chiamato "txt_result" e posizionarlo sullo stage.

Per terza cosa è necessario impostare un gateway di web2py che può comunicare con il client Flash definito sopra creando una nuova applicazione in web2py chiamata `pyamf_test` che ospiterà il nuovo servizio e il gateway AMF per il client Flash.

Modificare il controller "default.py" in modo che includa:

```
@service.amfrpc3('mydomain')
def addNumbers(val1, val2):
    return val1 + val2

def call(): return service()
```

Per quarta cosa compilare ed esportare/pubblicare il client Flash SWF con il nome `pyamf_test.swf`, posizionare i file "pyamf_test.amf", "pyamf_test.html", "AC_RunActiveContent.js" e "crossdomain.xml" nella cartella "static" della nuova applicazione che ospita il gateway "pyamf_test".

Si può testare il client accedendo a:

```
http://127.0.0.1:8000/pyamf_test/static/pyamf_test.html
```

Il gateway è chiamato in background quando il client si connette ad `addNumbers`.

Se si sta utilizzando AMF0 invece di AMF3 si può anche utilizzare il decoratore:

```
@service.amfrpc
```

invece di:

```
@service.amfrpc3('mydomain')
```

In questo caso la URL del servizio sarà:

```
http://127.0.0.1:8000/pyamf_test/default/call/amfrpc
```

9.2.4 SOAP

web2py include un client e un server SOAP creati da Mariano Reingart che può essere utilizzato in modo molto simile a XMLRPC.

Si consideri il seguente codice in un controller "default.py":

```
@service.soap('MyAdd',returns={'result':int,args={'a':int,'b':int,})
def add(a,b):
    return a+b
```

Ora in uno shell di Python si possono eseguire i seguenti comandi:

```
>>> from gluon.contrib.pysimplesoap.client import SoapClient
>>> client = SoapClient(
        location = "http://localhost:8000/app/default/call/soap",
        action = 'http://example.com/', # SOAPAction
        namespace = "http://example.com/sample.wsdl",
        soap_ns='soap', # classic soap 1.1 dialect
        trace = True, # print http/xml request and response
        ns = False) # do not add target namespace prefix

>>> print client.MyAdd(a=1,b=2)
3
```

Il WSDL del servizio è raggiungibile all'indirizzo:

```
http://127.0.0.1:8000/app/default/call?WSDL
```

E la documentazione può essere ottenuta, per ognuno dei metodi esposti, all'indirizzo:

```
http://127.0.0.1:8000/app/default/call?op=MyAdd
```

9.3 API di basso livello ed altre ricette

9.3.1 *simplejson*

web2py include `gluon.contrib.simplejson`, sviluppato da Bob Ippolito. Questo modulo fornisce l'implementazione standard di Python per la codifica e la decodifica di JSON.

SimpleJSON consiste di due funzioni:

- `gluon.contrib.simplesjson.dumps(a)` codifica l'oggetto Python `a` in JSON.
- `gluon.contrib.simplejson.loads(b)` decodifica l'oggetto Javascript `b` in un oggetto Python.

I tipi di oggetti che possono essere serializzati includono i tipi primitivi, le liste e i dizionari. Gli oggetti composti possono essere serializzati ad esclusione delle classi definite dall'utente.

Ecco un azione (che può essere definita nel controller "default.py") che serializza una lista che contiene i giorni della settimana utilizzando le API di basso livello:

```
def weekdays():
    names=['Sunday','Monday','Tuesday','Wednesday',
           'Thursday','Friday','Saturday']
    import gluon.contrib.simplejson
    return gluon.contrib.simplejson.dumps(names)
```

Questa è la pagina HTML che invia la richiesta Ajax all'azione precedentemente definita, riceve il messaggio di risposta JSON e memorizza la lista in una variabile Javascript:

```
{{extend 'layout.html'}}
<script>
$.getJSON('/application/default/weekdays',
          function(data){ alert(data); });
</script>
```

Il codice utilizza la funzione jQuery `$.getJSON` che esegue la chiamata Ajax,

memorizza la risposta (i giorni della settimana) in una variabile locale Javascript `data` e passa la variabile alla funzione di callback. Nell'esempio la funzione di callback avvisa l'utente che il dato è stato ricevuto.

9.3.2 PyRTF

Un'altra esigenza comune ai siti web è quella di generare documenti leggibili da Microsoft Word. Il modo più semplice di fare questo è utilizzare il formato RTF (*Rich Text Format*). Questo formato è stato inventato da Microsoft ed è diventato uno standard. web2py include `gluon.contrib.pyrtf`, sviluppato da Simon Cusack e revisionato da Grant Edwards. Questo modulo permette di generare documenti RTF con immagini e testo formattato e colorato.

Nel seguente esempio sono istanziate due classi base di RTF, `Document` e `Section`, la seconda è aggiunta alla prima e del testo è inserito in essa:

```
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)
```

Alla fine della funzione `Document` è serializzato da `q.dumps(doc)`. Notare che prima di restituire un documento RTF è necessario specificare il `Content-Type` della risposta nell'header, altrimenti il browser non sarà in grado di gestire il file. A seconda della configurazione del client il browser potrà chiedere se salvare questo file oppure lo aprirà con un text editor.

9.3.3 *ReportLab e PDF*

web2py può anche generare documenti di tipo PDF con una libreria aggiuntiva chiamata "ReportLab" (73).

Se web2py è stato eseguito dal sorgente è sufficiente avere installato ReportLab. Se si sta eseguendo la distribuzione binaria per Windows sarà necessario decomprimere ReportLab nella cartella web2py. Se si sta eseguendo la distribuzione binaria per Mac OS X sarà necessario decomprimere ReportLab nella cartella "web2py.app/Contents/Resources/".

In questo esempio si assume che ReportLab sia installato e che web2py possa trovarlo correttamente. Sarà creata una semplice azione, chiamata "get_me_a_pdf", che genera un documento PDF:

```
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
```

Notare che il PDF è generato in un file temporaneo, `tmpfilename`, che viene successivamente letto e cancellato.

Pe altre informazioni sulle API di ReportLab riferirsi alla documenti di ReportLab. E' raccomandato l'utilizzo delle API *Platypus* di ReportLab come `Paragraph`, `Spacer`, ecc.

9.4 Servizi ed autenticazione

Nel capitolo precedente è stato discusso l'uso dei seguenti decoratori:

```
@auth.requires_login()
@auth.requires_memebership(...)
@auth.requires_permission(...)
```

Per le azioni normali (non decorate come `@service`) questi decoratori possono essere usati anche se l'output è riprodotto in un formato diverso da HTML.

Per le funzioni definite come servizi e decorate utilizzando il decoratore `@service` il decoratore `@auth` non dovrebbe essere mai utilizzato. I due tipi di decoratore non devono infatti essere mischiati. Se è necessaria l'autenticazione è l'azione `call` che dovrà essere decorata:

```
@auth.requires_login()
def call(): return service()
```

Notare che è anche possibile istanziare più oggetti Service, registrare la differenti funzioni ed esporre alcune di loro con autenticazione ed altre senza:

```
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'
```

```
10
11 def public_call(): return public_service()
12
13 @auth.requires_login()
14 def private_call(): return private_service()
```

Questo presuppone che il chiamante stia passando le credenziali nell'header HTTP (con un cookie di sessione valida oppure utilizzando la *basic authentication*, come discusso nella sezione precedente). Il client deve però supportare questa funzionalità.

10

Ricette Ajax

Sebbene web2py sia pensato principalmente per lo sviluppo lato server, l'applicazione **welcome** (utilizzata come base per tutte le nuove applicazioni di web2py) include la libreria base del framework jQuery (32), i calendari jQuery (per selezionare una data, per selezionare una data ed un orario o per selezionare solo un orario), il menu "superfish.js" ed alcune altre funzioni aggiuntive Javascript basate su jQuery.

Non c'è nessuna restrizione in web2py riguardo l'utilizzo di altre librerie Ajax (75) come, per esempio Prototype, ExtJS, or YUI, ma è stato deciso di includere jQuery perchè è considerata più facile da usare e più potente rispetto ad altre librerie equivalenti. jQuery rispecchia inoltre lo spirito di web2py nell'essere funzionale e concisa.

10.1 web2py_ajax.html

Nelle applicazioni create in web2py è incluso un file chiamato

```
views/web2py_ajax.html
```

Questo file è incluso nella sezione HEAD del template di default "layout.html" e rende disponibili i seguenti servizi:

- Include `static/jquery.js`.
- Include `static/calendar.js` e `static/calendar.css`, se esistono.
- Definisce una funzione `ajax` (basata sulla sintassi $.ajax di jQuery).
- Fa sì che ogni DIV di classe "error" e ogni oggetto con tag di classe "flash" abbia l'effetto di scivolamento (*slide down*).
- Impedisce l'inserimento di caratteri non validi nei campi di input di classe "integer".
- Impedisce l'inserimento di caratteri non validi nei campi di input di classe "double".
- Collega i campi di input di tipo "date" con un popup di selezione della data.
- Collega i campi di input di tipo "datetime" con un popup di selezione della data e dell'ora.
- Collega i campi di input di tipo "time" con un popup di selezione dell'ora.
- Definisce `web2py_ajax_component`, un tool molto importante che verrà descritto nel capitolo 13.

Include inoltre le funzioni `popup`, `collapse` e `fade` per compatibilità con le versioni precedenti di web2py.

Ecco un esempio do come questi effetti entrano in gioco:

Considerare una applicazione **test** con il seguente modello:

```
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'))
```

con il controller "default.py":

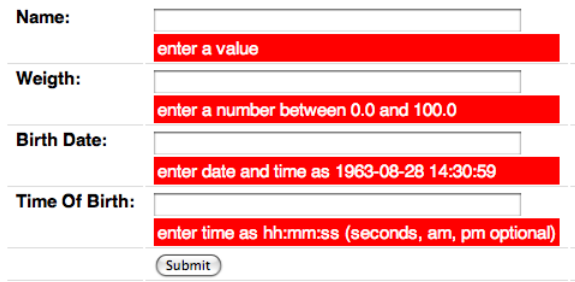
```
def index():
    form = SQLFORM(db.mytable)
    if form.accepts(request.vars, session):
        response.flash = 'record inserted'
    return dict(form=form)
```

e la vista "default/index.html":

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

L'azione "index" genera il seguente form:

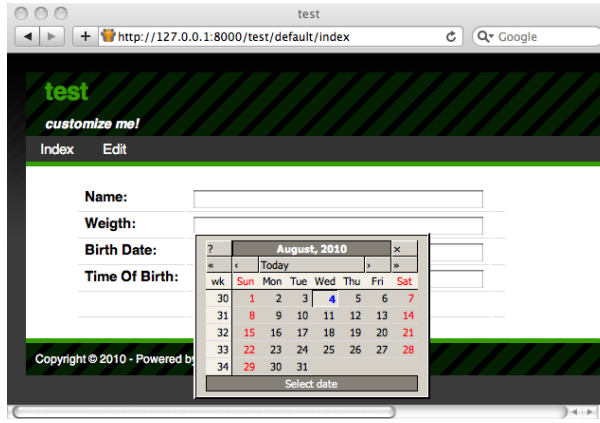
Se viene inviato un form non valido il server ritorna la pagina con il form modificato contenente i messaggi d'errore. I messaggi d'errore sono DIV di classe "error" e, grazie al codice presente in "web2py_ajax.html" gli errori appaiono con un effetto di scorrimento in basso:



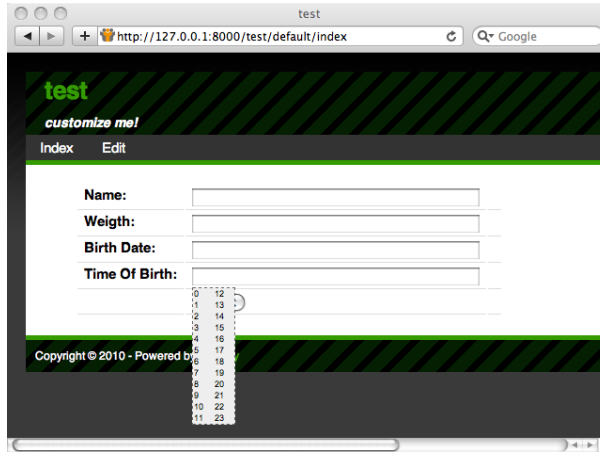
Il colore dei messaggi d'errore è definito nel codice CSS in "layout.html".

Il codice in "web2py_ajax.html" impedisce di inserire un valore non valido in un campo di input. Questo è fatto prima che il dato sia inviato al server e non è una sostituzione per la validazione lato server.

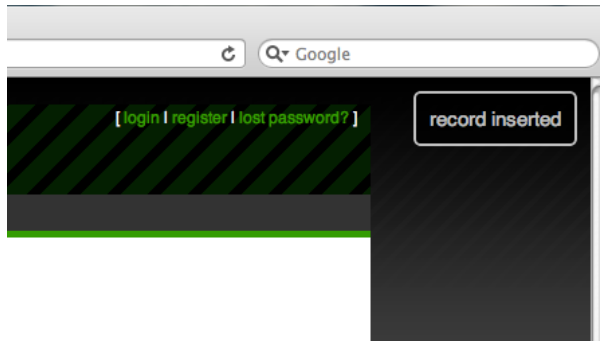
Il codice in "web2py_ajax.html" visualizza inoltre un selezionatore della data quando si inserisce un campo di input di classe "date" e visualizza un selettore di data e ora quando si inserisce un campo di input di classe "datetime". Ecco un esempio:



Infine il codice in "web2py_ajax.html" visualizza un selezionatore di orario quando si inserisce un campo di input di classe "time":



Dopo l'invio del form l'azione del controller imposta il messaggio flash di risposta a "record inserted". Il layout di default visualizza il messaggio in un DIV con id "flash". Il codice in "web2py_ajax.html" è responsabile per l'effetto di scorrimento in basso e per farlo scomparire quando questo viene cliccato:



Questi ed altri effetti sono accessibili da codice nelle viste e con gli helper nei controller.

10.2 Effetti con jQuery

Gli effetti di base descritti di seguito non richiedono nessun file aggiuntivo; tutto ciò che serve è già incluso in "web2py_ajax.html".

Gli oggetti HTML/XHTML possono essere identificati con il loro tipo (per esempio "DIV"), la loro classe o il loro id. Per esempio:

```
<div class="one" id="a">Hello</div>
<div class="two" id="b">World</div>
```

Questi due DIV appartengono rispettivamente alle classi "one" e "two" e hanno id uguali a "a" e "b".

In jQuery ci si può riferire al primo dei due oggetti con la seguente notazione (simile a quella CSS):

```
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"
```

ed al secondo con:

```
jQuery('.two')
jQuery('#b')
jQuery('DIV.two')
jQuery('DIV #b')
```

o ad ambedue con:

```
jQuery('DIV')
```

Grazie ai tag gli oggetti sono associati agli eventi, come "onclick". jQuery consente di collegare questi eventi agli effetti, per esempio a "slideToggle":

```
<div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
<div class="two" id="b">World</div>
```

Se ora si clicca su "Hello", la parola "World" scompare. Se si clicca di nuovo su "World" questa riappare. E' possibile rendere un oggetto nascosto di default assegnandogli una classe "hidden":

```
<div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
<div class="two hidden" id="b">World</div>
```

E' anche possibile collegare le azioni ad eventi esterni all'oggetto tag. Il codice precedente può anche essere riscritto come:

```
<div class="one" id="a">Hello</div>
<div class="two" id="b">World</div>
<script>
jQuery('.one').click(function(){jQuery('.two').slideToggle()});
</script>
```

Gli effetti ritornano l'oggetto chiamante, in modo da poter essere concatenati. `click` imposta la funzione da richiamare quanso si verifica l'evento click. Allo stesso modo funzionano `change`, `keyup`, `keydown`, `mouseover`, ecc.

Una situazione comune è la necessità di eseguire del codice Javascript solo dopo aver caricato l'intero documento. Questo è solitamente fatto dall'attributo `onload` del tag BODY ma jQuery mette a disposizione un modo alternativo che non richiede la modifica del 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>
```

Il codice della funzione anonima è eseguito solo quando il documento è pronto, dopo che è stato completamente caricato.

Questa è una lista di nomi di evento comuni:

Eventi del Form

- onchange: Script eseguito quando l'elemento è modificato.
- onsubmit: Script eseguito quando il form è inviato.
- onreset: Script eseguito quando il form è reimpostato.
- onselect: Script eseguito quando l'elemento viene selezionato.
- onblur: Script eseguito quando l'elemento perde il focus.
- onfocus: Script eseguito quando l'elemento acquisisce il focus.

Eventi di tastiera

- onkeydown: Script eseguito quando un tasto viene premuto.
- onkeypress: Script eseguito quando un tasto viene premuto e rilasciato.
- onkeyup: Script eseguito quando un tasto viene rilasciato.

Eventi del mouse

- onclick: Script eseguito dopo un click del mouse.
- ondblclick: Script eseguito dopo un doppio click del mouse.
- onmousedown: Script eseguito quando viene premuto il pulsante del mouse.
- onmousemove: Script eseguito quando il puntatore del mouse viene spostato.
- onmouseout: Script eseguito quando il puntatore del mouse viene spostato fuori da un elemento.
- onmouseover: Script eseguito quando il puntatore del mouse viene spostato sopra un elemento.
- onmouseup: Script eseguito quando viene rilasciato il pulsante del mouse.

Ecco una lista di effetti utili definiti in jQuery:

Effetti

- jQuery(...).attr(name): ritorna il nome dell'attributo.
- jQuery(...).attr(name, value): imposta l'attributo `name` a `value`.
- jQuery(...).show(): rende l'oggetto visibile.
- jQuery(...).hide(): rende l'oggetto invisibile.
- jQuery(...).slideToggle(speed, callback): fa scivolare l'oggetto in su o in giù.
- jQuery(...).slideUp(speed, callback): fa scivolare l'oggetto in su.
- jQuery(...).slideDown(speed, callback): fa scivolare l'oggetto in giù
- jQuery(...).fadeIn(speed, callback): fa apparire l'oggetto.
- jQuery(...).fadeOut(speed, callback): fa scomparire l'oggetto.

L'argomento `speed` è solitamente "slow", "fast" oppure può essere omesso (il default). `callback` è una funzione opzionale che è chiamata quando l'effetto è completo.

Gli effeti di jQuery possono anche essere facilmente inseriti in un helper, per esempio, in una vista:

```
{{=DIV('click me!', _onclick="jQuery(this).fadeOut()")}}
```

jQuery è una libreria Ajax compatta e concisa per questo web2py non ha bisogno di uno strato d'astrazione aggiuntivo per utilizzarla (tranne che per la funzione `ajax` discussa più avanti). Le API di jQuery sono accessibili ed immediatamente disponibili quando necessario. Consultare la documentazione delle API di jQuery per ulteriori informazioni sugli effetti disponibili. La libreria jQuery può anche essere estesa utilizzando plugin e Widget per l'interfaccia utente. Questi argomenti non sono discussi in questo manuale, si può fare riferimento a (77) per altre informazioni.

10.2.1 Campi condizionali nei form

Un'applicazione tipica degli effetti jQuery è un form che cambia in base ai valori dei suoi campi. Questo è facilmente realizzabile in web2py perchè l'helper SQLFORM genera dei form *CSS friendly* (cioè facilmente gestibili tramite CSS): il form contiene una tabella con delle righe; ogni riga contiene un'etichetta, un campo di input ed una terza colonna opzionale. Gli oggetti hanno id derivati dal nome della tabella e dal nome dei campi. La convenzione è che ogni campo di input ha un nome uguale a `tablename_fieldname` ed è contenuto in una riga chiamata `tablename_fieldname__row`.

Come esempio, verrà creato un form di input che chiede il nome di un utente è il nome del coniuge, ma solo se l'utente dichiara di essere sposato.

In una applicazione di test creare il seguente modello:

```
db = DAL('sqlite://db.db')
db.define_table('taxpayer',
    Field('name'),
    Field('married', 'boolean'),
    Field('spouse_name'))
```

Creare poi il seguente controller "default.py":

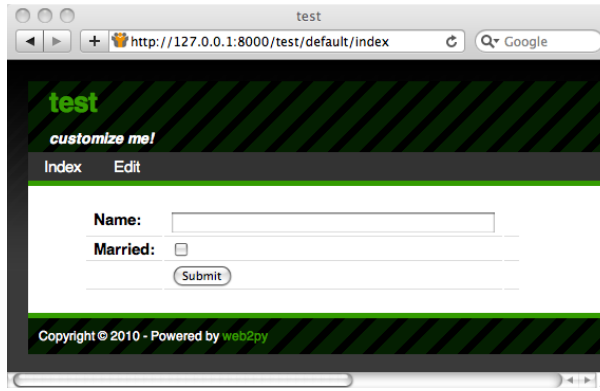
```
def index():
    form = SQLFORM(db.taxpayer)
    if form.accepts(request.vars, session):
        response.flash = 'record inserted'
    return dict(form=form)
```

e la seguente vista "default/index.html":

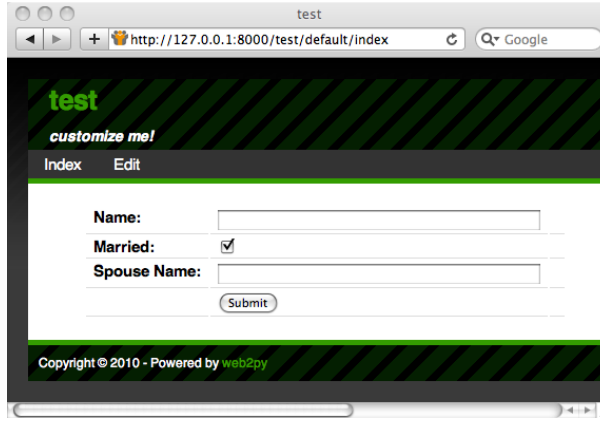
```
{{extend 'layout.html'}}
{{=form}}
<script>
jQuery(document).ready(function(){
   jQuery('#taxpayer_spouse_name__row').hide();
   jQuery('#taxpayer_married').change(function(){
        if(jQuery('#taxpayer_married').attr('checked'))
            jQuery('#taxpayer_spouse_name__row').show();
        else jQuery('#taxpayer_spouse_name__row').hide();});
```

```
10 });
11 </script>
```

Lo script nella vista serve per nascondere la riga contenente il nome del coniuge:



Quando viene selezionata la checkbox "married" il campo per il nome del coniuge riappare:



"taxpayer_married" è il checkbox associato al campo booleano "married" della

tabella "taxpayer". "taxpayer_spouse_name__row" è la riga contenente il campo di input per "spouse_name" della tabella "taxpayer".

10.2.2 Conferma della cancellazione

Un'altra utile applicazione è quella di richiedere la conferma quando si seleziona una checkbox "delete", come, per esempio, la checkbox di cancellazione che appare nei form di modifica.

Aggiungere al controller precedente la seguente azione:

```
def edit():
    row = db.taxpayer[request.args(0)]
    form = SQLFORM(db.taxpayer, row, deletable=True)
    if form.accepts(request.vars, session):
        response.flash = 'record updated'
    return dict(form=form)
```

e la corrispondente vista "default/edit.html"

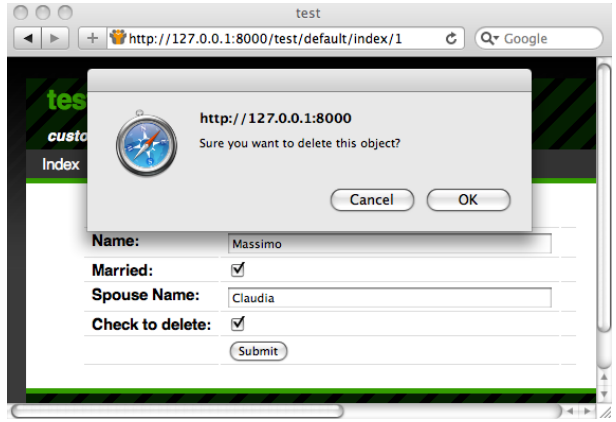
```
{{extend 'layout.html'}}
{{=form}}
```

L'argomento `deletable=True` nel costruttore di SQLFORM indica a web2py di visualizzare una checkbox "delete" nel form di modifica.

"web2py_ajax.html" include il seguente codice:

```
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;');
});
```

la checkbox ha una classe uguale a "delete". Questo codice jQuery collega l'evento "onclick" della checkbox con una finestra di conferma (standard Javascript) e deseleziona la checkbox se la conferma è negativa:



10.3 *La funzione* `ajax`

In "web2py_ajax.html" è definita una funzione chiamata `ajax` che è basata sulla funzione di jQuery `$.ajax` ma non dovrebbe essere confusa con essa. La funzione `$.ajax` di jQuery è molto più completa e per il suo utilizzo si rimanda alla documentazione in (32) e in (76). Tuttavia la funzione `ajax` di web2py è sufficiente per molti compiti, anche complessi, ed è più semplice da utilizzare.

`ajax` è una funzione Javascript con la seguente sintassi:

```
ajax(url, [id1, id2, ...], target)
```

Esegue una chiamata asincrona alla URL (il primo argomento), passa i valori dei campi con id uguali a quella della lista (secondo argomento) e memorizza la risposta nell'innerHTML del tag con id uguale al terzo argomento.

Ecco un esempio in un controller "default.py":

```
def one():
    return dict()

def echo():
```

```
    return request.vars.name
```

associato alla vista "default/one.html":

```
{{extend 'layout.html'}}
<form>
   <input id="name" onkeyup="ajax('echo', ['name'], 'target')" />
</form>
<div id="target"></div>
```

Quando si digita nel campo di input, non appena si rilascia un tasto (evento `onkeyup`) la funzione `ajax` viene eseguita e il valore del campo `id="name"` è passato all'azione "echo" che rimanda indietro il testo alla vista. La funzione `ajax` riceve la risposta e visualizza il testo ricevuto nel DIV "target".

10.3.1 Valutazione del target

Il terzo argomento della funzione `ajax` può essere la stringa ":eval". In questo caso la stringa ricevuta non sarà inserita in un tag del documento ma sarà valutata. Ecco un esempio, in un controller "default.py":

```
def one():
    return dict()

def echo():
    return "jQuery('#target').html(%s);" % repr(request.vars.name)
```

associato alla vista "default/one.html":

```
{{extend 'layout.html'}}
<form>
   <input id="name" onkeyup="ajax('echo', ['name'], ':eval')" />
</form>
<div id="target"></div>
```

Questo consente una risposta più complessa rispetto ad una semplice stringa.

10.3.2 Auto-completamento

web2py contiene un widget di auto-completamento, descritto nel capitolo relativo ai form. Qui è presentato un sistema di auto-completamento più semplice costruito da zero.

Un'altra applicazione della funzione `ajax` è l'auto-completamento. In questo esempio verrà creato un campo di input che si aspetta un nome di un mese e, quando l'utente inizia a digitare, esegue l'auto-completamento tramite una richiesta Ajax. In risposta una dropbox di auto-completamento apparirà sotto il campo di input.

Ecco il controller "default.py":

```
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])
```

associato alla vista "default/month_input.html":

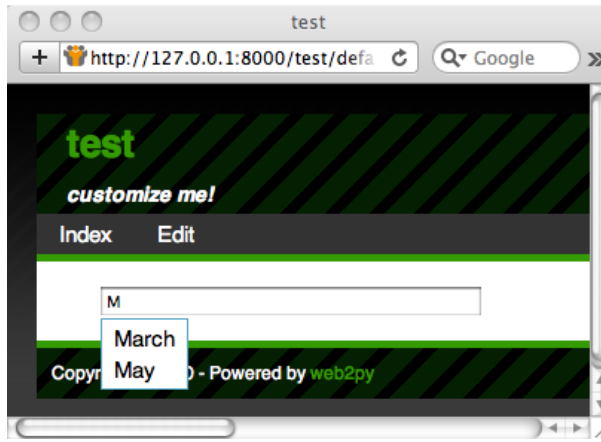
```
{{extend 'layout.html'}}
<style>
#suggestions { position: relative; }
.suggestions { background: white; border: solid 1px #55A6C8; }
.suggestions DIV { padding: 2px 4px 2px 4px; }
</style>

<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>
```

Lo script di jQuery nella vista intercetta la richiesta Ajax ogni volta che l'utente digita qualcosa nel campo "month". Il valore inserito nel campo è inviato tramite una richiesta Ajax all'azione "month_selector". Questa azione recupera una lista di nomi di mese che iniziano con il testo ricevuto (`selected`), costruisce una lista di DIV (ognuno contenente il nome di un mese) e la ritorna una stringa. La vista visualizza l'HTML di risposta nel DIV "suggestions". L'azione "month_selector" genera sia i suggerimenti che il codice inserito nei DIV che deve essere eseguito quando l'utente clicca su uno di loro. Per esempio, quando l'utente digita "Fe" l'azione ritorna:

```
<div onclick="jQuery('#month').val('February')"
     onmouseout="this.style.backgroundColor='white'"
     onmouseover="this.style.backgroundColor='yellow'">February</div>
```

Ecco il risultato finale:



Se i mesi sono memorizzati in una tabella di database come:

```
db.define_table('month', Field('name'))
```

allora si deve semplicemente sostituire l'azione `month_selector` con:

```
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 mette a disposizione un plugin "Auto-complete" con funzionalità aggiuntive che non è discusso in questo manuale.

10.3.3 Invio dei form con Ajax

Si consideri una pagina che consente ad un utente di inviare messaggi utilizzando Ajax senza dover ricaricare la pagina intera. web2py ha un meccnismo migliore per eseguire questo tipo di operazioni, descritto nel capitolo 13 e basato sull'utilizzo dell'helper "LOAD". In questo capitolo è indicato come eseguire l'operazione utilizzando jQuery.

La pagina contiene un form "myform" ed un DIV "target". Quando il form è inviato il server può accettarlo (ed eseguire un inserimento nel database) o rifiutarlo (perchè non passa i controlli di validazione). La notifica corrispondente è restituita nella risposta Ajax e visualizzata nel DIV "target".

Costruire una applicazione `test` con il seguente modello:

```
db = DAL('sqlite://db.db')
db.define_table('post', Field('your_message', 'text'))
db.post.your_message.requires = IS_NOT_EMPTY()
```

Ogni record "post" ha un solo campo "your_message" che non può essere vuoto.

Modificare il controller "default.py" nel seguente modo:

```
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()])
```

La prima azione non fa nulla se non ritornare la vista.

La seconda azione è una funzione di ritorno Ajax. Si aspetta le variabili del form in `request.vars`, le elabora e ritorna `DIV("Message posted")` se la validazione è stata positiva oppure una `TABLE` di messaggi d'errore se la validazione è stata negativa.

Modificare ora la vista "default/index.html":

```
{{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('new_post')}}',
       ['your_message'], 'target');
  return false;
});
</script>
```

Notare che in questo esempio il form è creato manualmente utilizzando HTML, ma è elaborato da SQLFORM in un'azione diversa da quella che

visualizza il form. L'oggetto SQLFORM non è mai serializzato in HTML. `SQLFORM.accepts` in questo caso non ha una sessione e `formname` è impostato a `None` perchè nel form creato manualmente non è presente nè un nome nè una chiave.

Lo script alla fine della vista collega il pulsante di invio di "myform" ad una funzione anonima che invia l'input con `id="your_message"` utilizzando la funzione `ajax` di web2py e visualizza la risposta all'interno del DIV con `id="target"`.

10.3.4 Votare e valutare

Votare o dare una valutazione in una pagina è un'altra tipica applicazione dove Ajax può essere utilizzato. Si consideri un'applicazione che consente agli utenti di votare delle immagini caricate. L'applicazione consiste di una sola pagina che visualizza le immagini ordinate secondo il numero di voti. Gli utenti possono votare più volte, sebbene sia facile modificare questo comportamento per far sì che gli utenti autenticati votino una sola volta tenendo traccia dei voti individuali associati con l'indirizzo IP dell'utente presente in `request.env.remote_addr`.

Ecco un modello d'esempio:

```
db = DAL('sqlite://images.db')
db.define_table('item',
    Field('image', 'upload'),
    Field('votes', 'integer', default=0))
```

Ecco il controller "default.py":

```
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.item[request.vars.id]
    new_votes = item.votes + 1
    item.update_record(votes=new_votes)
    return str(new_votes)
```

L'azione "download" è necessaria per consentire alla vista "list_items.html" di scaricare le immagini memorizzate nella cartella "uploads". L'azione "vote" è usata come funzione di risposta Ajax.

Ecco la vista "default/list_items.html":

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

<form><input type="hidden" id="id" value="" /></form>
{{for item in items:}}
<p>
<img src="{{=URL('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}}
```

Quando l'utente clicca su "[vote up]" il codice Javascript memorizza l'id dell'immagine nel campo nascosto di input "id" ed invia questo valore al server con una richiesta Ajax. Il server aumenta il contatore dei voti per il record corrispondente e ritorna il contatore aggiornato come stringa. Questo valore è poi inserito nel tag SPAN `item{{=item.id}}`.

> *Le funzioni di risposta Ajax possono essere utilizzate per effettuare elaborazioni in background ma per questo tipo di attività è meglio utilizzare* **cron** *o un processo di background (come discusso nel capitolo 4) poichè il server web imposta dei timeout sui thread. Se il tempo di elaborazione diventa troppo lungo il server web potrebbe bloccare il thread. Fare riferimento alla documentazione dei parametri del server web per impostare il valore di timeout.*

11

Ricette per l'installazione e la distribuzione

Ci sono diversi modi di installare e distribuire web2py in un ambiente di produzione; i dettagli dipendono dalla configurazione e dai servizi resi disponibili dall'host.

In questo capitolo saranno considerati i seguenti argomenti:

- Distribuzione in ambiente di produzione (Apache, Lighttpd e Cherokee)
- Sicurezza
- Scalabilità
- Distribuzione sulla piattaforma GAE (Google App Engine (13))

web2py include al suo interno un server web SSL (21), il server wsgi Rocket (22). Sebbene questo sia un server web veloce ha poche possibilità di configurazione. Per questo motivo è meglio installare web2py con i server web Apache (79), Lighttpd (86) o Cherokee (87). Sono tutti server web open source personalizzabili ed in grado di sopportare alti volumi di traffico in ambiente di produzione in modo affidabile. Possono essere configurati per distribuire autonomamente i file statici, per utilizzare HTTPS e passare il controllo a web2py per il contenuto dinamico.

Fino a pochi anni fa il meccanismo standard per la comunicazione tra i server web e le applicazioni era CGI (*Common Gateway Interface*) (78). Il problema principale con CGI è che crea un nuovo processo per ogni richiesta HTTP. Se l'applicazione web è scritta in un linguaggio interpretato ogni richiesta HTTP servita dagli script CGI avvia una nuova istanza dell'interprete. Questo genera lentezza e dovrebbe essere evitato in un ambiente di produzione. Inoltre CGI può gestire solamente risposte semplici, non può per esempio, gestire lo streaming di un file. In web2py è presente il file `cgihandler.py` per interfacciarsi con CGI.

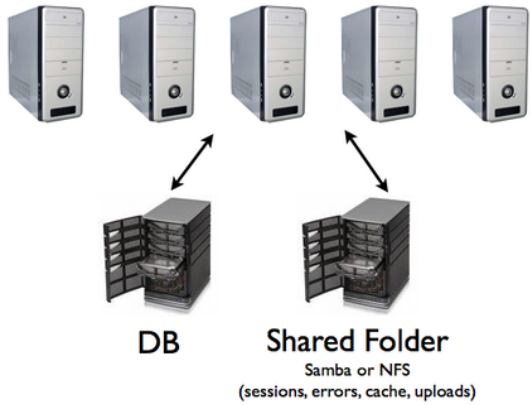
Una soluzione a questo problema è l'utilizzo del modulo `mod_python` per Apache. Sebbene questo modulo sia stato ufficialmente abbandonato della *Apache Software Foundation* il suo uso è ancora molto comune. `mod_python` genera un'istanza dell'interprete Python all'avvio di Apache e serve ogni richiesta HTTP nel suo thread senza dover riavviare Python ogni volta. Questa soluzione è migliore di CGI ma non è ottimale, poichè `mod_python` utilizza una sua particolare interfaccia tra l'applicazione e il server web. In `mod_python` tutte le applicazioni in esecuzione sul server web utilizzano lo stesso userid e groupid e questo può portare a rischi per la sicurezza. In web2py è presente il file `modpythonhandler.py` per interfacciarsi con `mod_python`.

Negli anni recenti la comunità Python si è accordata su un nuovo standard per la comunicazione tra i server web e le applicazioni web scritte in Python. Questo meccanismo è chiamato WSGI (*Web Server Gateway Interface*) (17; 18). web2py è stato costruito su WSGI e fornisce dei gestori ('handler") per utilizzare altre interfacce quando WSGI non è disponibile. Apache supporta WSGI con il modulo `mod_wsgi` (85) sviluppato da Graham Dumpleton. In web2py è presente il file `wsgihandler.py` per interfacciarsi con WSGI.

Alcuni servizi di hosting web non supportano WSGI, in questo caso è necessario utilizzare Apache come server proxy ed inoltrare tutte le richieste in entrata al server web presente in web2py (in esecuzione, per esempio, sulla porta 8000 di localhost). Sia utilizzando `mod_wsgi` che `mod_proxy` Apache può essere configurato per servire i file statici e gestire direttamente la cifratura SSL, alleggerendo il carico di lavoro di web2py.

Il server web Lighttpd attualmente non supporta l'interfaccia WSGI ma supporta l'interfaccia FastCGI (88), che è un miglioramento di CGI. L'obiettivo principale di FastCGI è quello di ridurre il sovraccarico associato all'interfacciamento del server web e dei programmi CGI consentendo al server di rispondere a più richieste HTTP contemporaneamente. Secondo il sito web di Lighttpd, "Lighttpd fa funzionare molti famosi siti del Web 2.0, come YouTube e Wikipedia. La sua infrastruttura di I/O ad alta velocità consente di scalare molte volte di più di altri server web sul medesimo hardware". In effetti Lighttpd con FastCGI è più veloce di Apache con `mod_wsgi`. In web2py è presente il file `fcgihandler.py` per interfacciarsi con FastCGi. web2py include anche il file `gaehandler.py` per interfacciarsi a *Google App Engine* (GAE). Con GAE le applicazioni web sono eseguite nel "cloud". Questo significa che il framework astrae completamente ogni dettaglio legato all'hardware. L'applicazione web è replicata automaticamente per servire tutte le richieste contemporanee. Per replicazione in questi caso si intende non solo thread multipli su un server singolo ma anche processi multipli su server differenti. GAE raggiunge questo dettaglio di scalablità impedendo l'accesso al file system. Tutte le informazioni devono essere memorizzate nel *Google Big Table Datastore* o in *memcache*.

Sulle altre piattaforme la scalabilità è un problema che deve essere gestito e che può richiedere alcune modifiche nelle applicazioni di web2py. Il modo più comune per ottenere la scalabilità è utilizzare più server web dietro un bilanciatore di carico (un semplice *round robin* o qualcosa di più sofisticato con il controllo dell'heartbeat sui server). Anche in caso di più server web deve esistere uno, e solamente uno, server di database. Per default web2py utilizza il file system per memorizzare le sessioni, i ticket degli errori, i file caricati e la cache. Questo significa che nella configurazione di default la corrispondenti cartelle dovranno essere condivise:



Nel resto del capitolo saranno illustrate varie *ricette* che possono essere utilizzate per migliorare la configurazione base. Sarà possibile, per esempio:

- Memorizzare le sessioni nel database, nella cache oppure non memorizzare alcuna sessione
- Memorizzare i ticket d'errore sul file system locale e spostarli nel database.
- Utilizzare `memcache` invece di `cache.ram` e `cache.disk`.
- Memorizzare i file caricati dagli utenti un un database inveche che nel file system.

Le prime tre ricette sono consigliate in tutti i casi, mentre la quarta offre vantaggi solo in caso di file di piccole dimensioni, ma potrebbe essere controproducente in caso di file di grandi dimensioni.

11.1 Linux/Unix

11.1.1 Deployment in un singolo passaggio

Ecco alcuni passi per installare Apache, Python, mod_wsgi, web2py e Postgresql da zero.

Su Ubuntu:

```
wget http://web2py.googlecode.com/svn/trunk/scripts/setup-web2py-ubuntu.sh
chmod +x setup-web2py-ubuntu.sh
sudo ./setup-web2py-ubuntu.sh
```

Su Fedora:

```
wget http://web2py.googlecode.com/svn/trunk/scripts/setup-web2py-fedora.sh
chmod +x setup-web2py-fedora.sh
sudo ./setup-web2py-fedora.sh
```

Questi due script dovrebbero funzionare senza problemi ma ogni installazione Linux è leggermente diversa dalle altre e quindi potrebbe essere necessario controllare il codice sorgente prima di eseguirli. La maggior parte delle operazioni eseguite dagli script sono spiegate in seguito seguendo il caso dell'installazione su Ubuntu. Questi script non implementano le ottimizzazioni di scalabilità discusse più avanti nel capitolo.

11.1.2 Configurazione di Apache

In questa sezione è utilizzato Ubuntu 8.04 Server Edition come piattaforma di riferimento. I comandi di configurazione sono molti simili sulle altre distribuzioni Linux basate su Debian, ma possono essere diversi sui sistem basati su Fedora (che usano `yast` invece di `apt-get`).

Prima di tutto assicurarsi che i package di Python e di Apache siano installati utilizzando i seguenti comandi:

```
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
sudo apt-egt -y install libapache2-mod-proxy-html
```

Abilitare quindi i moduli SSL, proxy e WSGI in Apache:

```
sudo ln -s /etc/apache2/mods-available/proxy_http.load \
           /etc/apache2/mods-enabled/proxy_http.load
sudo a2enmod ssl
sudo a2enmod proxy
sudo a2enmod proxy_http
sudo a2enmod wsgi
```

Creare la cartella SSL, dove andranno inseriti i certificati:

```
sudo mkdir /etc/apache2/ssl
```

I certificati SSL dovrebbero essere ottenuti da una Certificate Authority riconosciuta come, per esempio Verisign.com, ma per un ambiente di test si possono generare certificati auto-firmati seguendo le istruzioni in (84)

Riavviare il server web:

```
sudo /etc/init.d/apache2 restart
```

Il file di configurazione di Apache è:

```
/etc/apache2/sites-available/default
```

Mentre i file di log di Apache si trovano in:

```
/var/log/apache2/
```

11.1.3 mod_wsgi

Scaricare e decomprimere il pacchetto sorgente di web2py sulla macchina dove è stato precedentemente installato il server web.

Installare web2py in `/users/www-data/`, per esempio, ed assegnare il proprietario all'utente *www-data* e al gruppo *www-data*. Questo può essere fatto con i seguenti comandi:

```
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
```

Per integrare web2py con `mod_wsgi` creare un nuovo file di configurazione di Apache:

```
/etc/apache2/sites-available/web2py
```

ed includere il seguente codice:

```
<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>
```

```
23
24   <Location /admin>
25   Deny from all
26   </Location>
27
28   <LocationMatch ^/([^/]+)/appadmin>
29   Deny from all
30   </LocationMatch>
31
32   CustomLog /private/var/log/apache2/access.log common
33   ErrorLog /private/var/log/apache2/error.log
34 </VirtualHost>
```

Quando Apache viene riavviato tutte le richieste a web2py saranno eseguite direttamente senza passare dal server wsgi interno di web2py "Rocket".

Ecco una breve spiegazione:

```
1 WSGIDaemonProcess web2py user=www-data group=www-data
2                          display-name=%{GROUP}
```

definisce un gruppo di processi in background (demoni) nel contesto di "web2py.example.com". Poichè questo parametro è definito all'interno del virtual host, solamente questo virtual host, incluso ogni altro virtual host con lo stesso nome ma su una porta diversa, può utilizzare questo WSGIProcessGroup. Le opzioni "user" e "group" devono essere impostate all'utente che ha i diritti d'accesso alla directory dove web2py è stato installato. Non è necessario impostare "user" e "group" nel caso in cui la directory di installazione di web2py sia scrivibile dall'utente con cui è eseguito Apache. L'opzione "display-name" è impostata in modo che il nome del processo appaia come "(wsgi:web2py)" in `ps` invece che il nome dell'eseguibile del server web Apache. Poichè non sono specificate le opzioni "processes" o "threads" il demone avrà un singolo processo con 15 thread al suo interno. Questo è solitamente più che sufficiente per la maggior parte dei siti e dovrebbe essere lasciato così. Se questi parametri dovessero essere modificati non utilizzare "processes=1" perchè in questo modo si disattiverebbe qualsiasi tool di debug WSGI che utilizza il flag "wsgi.multiprocess". L'utilizzo dell'opzione "processes" infatti imposta questo flag a true anche nel caso di un processo singolo ma i tool di debug si aspettano che sia impostato a false. Se il codice

dell'applicazione o le librerie utilizzate non sono sicure in ambiente multi-thread (*thread-safe*) è bene impostare "processes=5 threads=1". In questo modo saranno creati cinque processi nel gruppo dove ogni processo avrà un solo thread. Si può anche prendere in considerazione l'opzione "maximum-requests=1000" nel caso che l'applicazione abbia problemi nel cancellare correttamente dalla memoria gli oggetti non più utilizzati.

```
WSGIProcessGroup web2py
```

delega l'esecuzione di tutte le applicazioni WSGI al gruppo di processi che è stato configurato con la direttiva WSGIDaemonProcess.

```
WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
```

attiva l'applicazione web2py. In questo caso è montata alla root del sito web.

```
<Directory /users/www-data/web2py>
  ...
</Directory>
```

da ad Apache il permesso di accedere allo script WSGI.

```
<Directory /users/www-data/web2py/applications/*/static/>
  Order Allow,Deny
  Allow from all
</Directory>
```

Indica ad Apache di non utilizzare web2py quando si ricercano i file statici.

```
<Location /admin>
  Deny from all
</Location>
```

e

```
<LocationMatch ^/([^/]+)/appadmin>
  Deny from all
</LocationMatch>
```

impediscono l'accesso pubblico ad **admin** e ad **appadmin**.

Normalmente si potrebbe consentire l'accesso a tutta la cartella che contiene lo script di WSGI ma in web2py non è possibile farlo perchè lo script si trova nella directory che contiene il codice sorgente ed il file che contiene la password dell'interfaccia amministrativa. L'accesso completo alla directory causerebbe dei problemi di sicurezza perchè sarebbe possibile leggere qualsiasi file al suo interno. Per evitare questi problemi deve essere esplicitamente consentito l'accesso al solo file di script WSGI mentre l'accesso agli altri file deve essere negato. Per maggior sicurezza questa configurazione può essere eseguita in un file.htaccess.

Un file di configurazione di Apache per WSGI completo e commentato è presente in:

```
scripts/web2py-wsgi.conf
```

Questa sezione è stata creata grazie all'aiuto di Graham Dumpleton, sviluppatore di `mod_wsgi`.

11.1.4 mod_wsgi e SSL

Per obbligare alcune applicazioni (per esempio **admin** e **appadmin**) ad essere eseguite in HTTPS memorizzare i file del certificato e della chiave privata in:

```
/etc/apache2/ssl/server.crt
/etc/apache2/ssl/server.key
```

e modificare il file di configurazione di Apache `web2py.conf` aggiungendo:

```
<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>
```

Riavviando Apache si dovrebbe essere in grado di accedere a:

```
https://www.example.com/admin
https://www.example.com/examples/appadmin
http://www.example.com/examples
```

mentre non sarà più possibile l'accesso a:

```
http://www.example.com/admin
http://www.example.com/examples/appadmin
```

11.1.5 mod_proxy

Alcune distribuzioni Unix/Linux possono utilizzare Apache ma non supportano `mod_wsgi`. In questo caso la soluzione più semplice è eseguire Apache come un proxy verso web2py e far gestire direttamente ad Apache solo i file statici.

Ecco una semplice configurazione di Apache come proxy:

```
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
     ProxyRequests off
     ProxyPass http://localhost:8000/welcome
     ProxyPassReverse http://localhost:8000/
     ProxyHTMLURLMap http://127.0.0.1:8000/welcome/ /welcome
   </Location>
   LogFormat "%h %l %u %t "%r" %>s %b" common
   CustomLog /var/log/apache2/access.log common
</VirtualHost>
```

Questa configurazione espone solamente l'applicazione "welcome". Per esporre altre applicazioni si devono aggiungere le corrispondenti direttive <Location>... </Location> con la stessa sintassi usata per l'applicazione "welcome".

Lo script assume che il server web2py stia in attesa sulla porta 8000. Prima di riavviare Apache assicurarsi che questo sia corretto:

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

Si può specificare la password con l'opzione `-a` oppure utilizzare il parametro "<recycle>" invece della password. In questo caso la password precedentemente memorizzata sarà riutilizzata e non verrà memorizzata nell'history della linea di comando.

Si può anche usare il parametro "<ask>" per richiedere una nuova password ad ogni riavvio.

Il comando `nohup` fa si che il server non venga terminato quando si chiude la

shell. `nohup` memorizza tutto il suo output in `nohup.out`.

Per obbligare l'uso di admin e di appadmin su HTTPS utilizzare il seguente file di configurazione di Apache:

```
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>
```

L'interfaccia amministrativa deve essere disabilitata quando web2py è in esecuzione su un host condiviso con `mod_proxy` *altrimenti altri utenti non autorizzati potrebbero accedervi.*

11.1.6 Avvio come demone Linux

Se non si usa `mod_wsgi` si dovrebbe impostare il server web2py in modo che possa essere avviato, fermato e riavviato come ogni altro demone Linux così da automatizzarne l'avvio durante il boot del computer. Questo si fa in modo diverso sulle diverse distribuzioni Linux/Unix.

Nella cartella web2py ci sono due script che possono essere utilizzati per questo scopo:

```
scripts/web2py.ubuntu.sh
scripts/web2py.fedora.sh
```

Su Ubuntu, come su ogni altra distribuzione basata su Debian, modificare il file "web2py.ubuntu.sh" e sostituire "/usr/lib/web2py" con il path dell'installazione di web2py. Digitare quindi i seguenti comandi per spostare il file nella cartella corretta, registrarlo come servizio d'avvio ed avviarlo:

```
sudo cp scripts/web2py.ubuntu.sh /etc/init.d/web2py
sudo update-rc.d web2py defaults
sudo /etc/init.d/web2py start
```

Su Fedora, o su ogni altra distribuzione basata su Fedora, modificare il file "web2py.fedora.sh" a sostituire "/usr/lib/web2py" con il path dell'installazione di web2py. Digitare quindi i seguenti comandi per spostare il file nella cartella corretta, registrarlo come servizio d'avvio ed avviarlo:

```
sudo cp scripts/web2py.fedora.sh /etc/rc.d/init.d/web2pyd
sudo chkconfig --add web2pyd
sudo service web2py start
```

11.1.7 Lighttpd

E' possibile installare Lighttpd su Ubuntu o su altre distribuzioni basate su Debian con il seguente comando:

```
apt-get -y install lighttpd
```

Una volta installato modificare `/etc/rc.local` e creare un processo fcgi in background per web2py:

```
cd /var/www/web2py && sudo -u www-data nohup python fcgihandler.py &
```

Modificare quindi il file di configurazione di Lighttpd:

```
/etc/lighttpd/lighttpd.conf
```

in modo che possa trovare il socket creato del processo precedente. Inserire nel file di configurazione:

```
server.modules = (
        "mod_access",
        "mod_alias",
        "mod_compress",
        "mod_rewrite",
        "mod_fastcgi",
        "mod_redirect",
        "mod_accesslog",
        "mod_status",
)

server.port = 80
server.bind = "0.0.0.0"
server.event-handler = "freebsd-kqueue"
server.error-handler-404 = "/test.fcgi"
server.document-root = "/users/www-data/web2py/"
server.errorlog      = "/tmp/error.log"

fastcgi.server = (
  "/handler_web2py.fcgi" => (
      "handler_web2py" => ( #name for logs
         "check-local" => "disable",
         "socket" => "/tmp/fcgi.sock"
      )
   ),
```

```
)

$HTTP["host"] = "(^|\.)example\.com$" {
 server.document-root="/var/www/web2py"
    url.rewrite-once = (
      "^(/.+?/static/.+)$" => "/applications$1",
      "(^|/.*)$" => "/handler_web2py.fcgi$1",
    )
}
```

Provare poi a controllare la presenza di eventuali errori di sintassi:

```
lighttpd -t -f /etc/lighttpd/lighttpd.conf
```

e riavviare il server web con:

```
/etc/init.d/lighttpd restart
```

Notare che FastCGI collega il server web2py ad un socket Unix, non ad un socket IP:

```
/tmp/fcgi.sock
```

Questo socket è il punto dove Lighhtpd inoltra le richieste HTTP e riceve le risposte da web2py. I socket Unix sono più leggeri dei socket IP e questo è uno dei motivi per cui l'utilizzo di web2py con Lighttpd e FastCGI è più veloce. Come nel caso di Apache è possibile configurare Lighttpd per gestire direttamente i file statici e obbligare l'utilizzo di alcune applicazioni in HTTPS. Fare riferimento alla documentazione di Lighttpd per ultariori dettagli. Gli esempi di questa sezione sono stati presi dai post di John Heenan in *web2pyslices*.

> *L'interfaccia amministrativa deve essere disabilitata quando web2py è eseguito in un host condiviso con FastCGI, altrimenti altri utenti non autorizzati potrebbero accedervi.*

11.1.8 Host condiviso con mod_python

Spesso sugli host condivisi non si ha la possibilità di modificare i file di configurazione di Apache. Solitamente questi host ancora utilizzano mod_python (anche se questo modulo non è più mantenuto) invece di mod_wsgi. Anche in questo caso è possibile eseguire web2py. Ecco una configurazione d'esempio:

Copiare i contenuti di web2py nella cartella "htdocs".

Nella cartella web2py creare il file "web2py_modpython.py" con il seguente contenuto:

```
from mod_python import apache
import modpythonhandler

def handler(req):
    req.subprocess_env['PATH_INFO'] = req.subprocess_env['SCRIPT_URL']
    return modpythonhandler.handler(req)
```

Creare (o aggiornare) il file ".htaccess" con il seguente contenuto:

```
SetHandler python-program
PythonHandler web2py_modpython
#PythonDebug On
```

Questo esempio è stato fornito da Niktar.

11.1.9 Cherokee con FastGGI

Cherokee è un server web molto veloce e, come web2py, fornisce una interfaccia web Ajax per la sua configurazione scritta in Python. Inoltre non necessita di riavvio per la maggior parte dei cambiamenti alla sua configurazione. Ecco i passi necessari per utilizzare web2py con Cherokee:

Scaricare Cherokee (87)

Decomprimere il file scaricato, configurare ed installare Cherokee:

```
tar -xzf cherokee-0.9.4.tar.gz
cd cherokee-0.9.4
./configure --enable-fcgi && make
make install
```

Avviare web2py normalmente almeno una volta per assicurarsi che la cartella "applications" venga creata.

Scrivere un file di script chiamato "startweb2py.sh" con il seguente contenuto:

```
#!/bin/bash
cd /var/web2py
python /var/web2py/fcgihandler.py &
```

Assegnare allo script i privilegi di esecuzione ed eseguirlo. Lo script avvierà web2py con il gestore FastCGI.

Avviare Cherokee e cherokee-admin:

```
sudo nohup cherokee &
sudo nohup cherokee-admin &
```

Per default cherokee-admin accetta solo le richiesta sulla porta 9090 dell'interfaccia locale. Questo non è un problema se si ha accesso direttamente alla macchina dove Cherokee è installato, altrimenti si deve configurare Cherokee per ascoltare su uno specifico IP e una specifica porta con le seguenti opzioni:

```
-b,  --bind[=IP]
-p,  --port=NUM
```

oppure, per maggior sicurezza, eseguire un forward di porta in SSH:

```
ssh -L 9090:localhost:9090 remotehost
```

Aprire la pagina "http://localhost:9090" sul proprio browser. Se la configurazione è corretta si potrà accedere all'interfaccia web di amministrazione di Cherokee.

Nell'interfaccia web di amministrazione selezionare "info sources". Scegliere "Local interpreter" ed aggiungere il seguente codice. Alla fine selezionare "Add New".

```
Nick: web2py
Connection: /tmp/fcgi.sock
Interpreter: /var/web2py/startweb2py.sh
```

I passi da seguire per completare la configurazione sono:

- Selezionare "Virtual Servers" e scegliere "Default".
- Selezionare "Behavior" e scegliere "default".
- Scegliere "FastCGI" al posto di "List and Send" dalla lista.
- Alla fine selezionare "web2py" come "Application Server".
- Selezionare tutte le caselle di spunta (si può lasciare deselezionata Allow-x-sendfile). Se appare un messaggio d'errore disabilitare e riabilitare una delle caselle di spunta (in questo modo i parametri dell'application server saranno aggiornati).
- Accedere con il browser a "http://yoursite" dove dovrebbe apparire la pagina "Welcome to web2py".

11.1.10 PostgreSQL

PostgreSQL è un database open source utilizzato anche in ambienti di produzione molto esigenti, per esempio per memorizzare i nomi di dominio.org, ed è in grado di gestire senza problemi centinaia di terabyte di dati. Ha un supporto alle transazioni solido e veloce con una caratteristica di "auto-vacuum" che libera l'amministratore dalla maggior parte dei compiti di manutenzione.

Su una distribuzione Ubuntu (o altre distribuzioni Linux basate su Debian) è facile installare PostgreSQL e le sue API Python con:

```
sudo apt-get -y install postgresql
sudo apt-get -y install python-psycopg2
```

E' bene eseguire il server web (o i web server) e il server di database su macchine diverse. In questo caso le macchine che eseguono il web server dovrebbero essere collegate con una rete fisica sicura o dovrebbero stabilire una connessione in un tunnel SSL con il server di database.

Modificare il file di configurazione di PostgreSQL:

```
sudo nano /etc/postgresql/8.4/main/postgresql.conf
```

ed assicurarsi che contenga le due linee:

```
...
track_counts = on
...
autovacuum = on   # Enable autovacuum subprocess?  'on'
...
```

Avviare il server di database con:

```
sudo /etc/init.d/postgresql restart
```

Quando il server PostgreSQL viene riavviato dovrebbe segnalare su quale porta sta ascoltando. A meno di non avere più server di database sulla stessa macchina, la porta dovrebbe essere la 5432.

I log di PostgreSQL sono in:

```
/var/log/postgresql/
```

Una volta che il server di database è funzionante deve essere creato un utente ed un database per le applicazioni web2py:

```
sudo -u postgres createuser -PE -s myuser
postgresql> createdb -O myself -E UTF8 mydb
postgresql> echo 'The following databases have been created:'
postgresql> psql -l
postgresql> psql mydb
```

Il primo di questi comandi garantisce l'accesso di *superuser* al nuovo utente `myuser`. Verrà richiesta una password.

Qualsiasi applicazione web2py può collegarsi al database con il comando:

```
db = DAL("postgres://myuser:mypassword@localhost:5432/mydb")
```

dove `mypassword` è la password precedentemente inserita e 5432 è la porta dove il database è in ascolto.

Normalmente si deve utilizzare un database per ogni applicazione con più istanze della stessa applicazione che accedono allo stesso database. E' comunque possibile far condividere lo stesso database ad applicazioni diverse.

Per eseguire il backup del database consultare la configurazione di PostgreSQL, in particolare la sezione riguardante i comandi `pg_dump` e `pg_restore`.

11.2 Windows

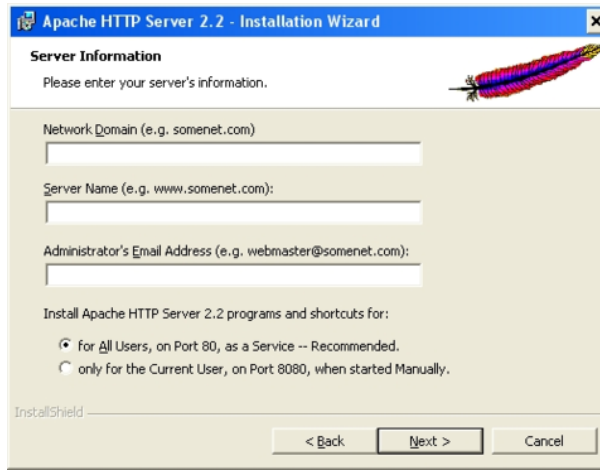
11.2.1 Apache e mod_wsgi

L'installazione di Apache e di mod_wsgi in Windows richiede una procedura differente. Con Python 2.5 e web2py installato dal sorgente in `c:/web2py` i passi da seguire sono i seguenti:

Scaricare i pacchetti necessari:

- Apache `apache_2.2.11-win32-x86-openssl-0.9.8i.msi` da (80)
- mod_wsgi da (81)

Eseguire il file di installazione `apache ... .msi` e seguire le schermate dell'installazione. Sulla schermata di informazioni del server:



inserire i seguenti valori:

- **Network Domain**: inserire il dominio DNS nel quale il server sarà registrato. Per esempio se il nome completo del server è server.mydomain.net deve essere inserito mydomain.net.
- **ServerName**: Il nome DNS completo del server. Continuando con l'esempio precedente andrebbe inserito server.mydomain.net. Inserire un nome completo DNS o un indirizzo IP dell'installazione di web2py, non una abbreviazione. Per maggiori informazioni vedere (83).
- **Administrator's Email Address**. Inserire l'indirizzo email dell'amministratore o del gestore del sito. L'indirizzo verrà visualizzato per default nelle pagine d'errore.

Continuare con l'installazione tipica fino alla fine a meno di dover aggiungere particolari configurazioni. La procedura d'installazione per default installa Apache nella cartella:

```
C:/Program Files/Apache Software Foundation/Apache2.2/
```

Nel resto di questa documentazione questa cartella sarà chiamata semplicemente `Apache2.2`.

Copiare il modulo mod_wsgi.so in `Apache2.2/modules`

(scritto da Chris Travers, pubblicato da Open Source Software Lab di Microsoft nel dicembre 2007)

Creare i file di certificato `server.crt` e `server.key` (come già discusso nella precedente sezione) e posizionarli nella cartella `Apache2.2/conf`. Il file di configurazione è in `Apache2.2/conf/openssl.cnf`.

Modificare `Apache2.2/conf/httpd.conf`, rimuovendo il commento (il carattere #) dalla linea:

```
LoadModule ssl_module modules/mod_ssl.so
```

ed aggiungere la seguente linea dopo tutte le altre linee *LoadModule*:

```
LoadModule wsgi_module modules/mod_wsgi.so
```

cercare la linea "Listen 80" ed aggiungere la seguente linea subito dopo:

```
Listen 443
```

aggiungere le seguenti linee alla fine cambiando la lettera del disco, il numero di porta e il nome del server con i valori corretti per la propria installazione:

```
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>
```

Salvare la configurazione e controllarla utilizzando [Start > Program > Apache HTTP Server 2.2 > Configure Apache Server > Test Configuration]

Se non ci sono problemi apparirà una finestra di linea di comando che si chiuderà immediatamente. Ora si può avviare Apache:

[Start > Program > Apache HTTP Server 2.2 > Control Apache Server > Start]
o, meglio ancora, è possibile avviare il monitor nella taskbar:

```
[Start > Program > Apache HTTP Server 2.2 > Control Apache Server]
```

Ora si può cliccare con il tasto destro sull'icona di Apache nella taskbar per aprire il monitor di Apache e da lì avviare, fermare e riavviare Apache.

Questa sezione è stata creata da Jonathan Lundell.

11.2.2 Avviare web2py come un servizio Windows

Quello che in Linux è chiamato "demone" in Windows è chiamato "servizio". Il server web2py può facilmente essere installato, avviato e fermato come un servizio Windows.

Per eseguire web2py come un servizio Windows è necessario creare il file

"options.py" con i parametri d'avvio:

```
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()
```

Non è necessario creare "options.py" da zero perchè nella cartella web2py è presente il file "options_std.py" che può essere utilizzato come modello.

Dopo aver creato il file "options.py" nella cartella d'installazione di web2py si può installare web2py come servizio con:

```
python web2py.py -W install
```

ed avviare o fermare il servizio con:

```
python web2py.py -W start
python web2py.py -W stop
```

11.3 Rendere sicure le sessioni e l'applicazione **admin**

E' pericoloso esporre pubblicamente le applicazioni **admin** e i controller **appadmin** a meno che non siano potretti tramite HTTPS. Inoltre la password e le credenziali d'accesso non dovrebbero mai essere trasmesse in chiaro. Questo vale per web2py come per qualsiasi altra applicazione web.

Nelle applicazioni che richiedono l'autenticazione si dovrebbero rendere sicuri i cookie di sessione con:

```
session.secure()
```

Un modo sicuro di proteggere un ambiente di produzione su un server è quello di fermare web2py e rimuovere tutti i file `parameters_*.py` dalla cartella di installazione di web2py e di riavviare web2py senza password. In questo modo l'applicazione admin e i controller appadmin saranno completamente disabilitati.

Avviare quindi una seconda instanza di web2py accessibile solo tramite localhost:

```
nohup python web2py -p 8001 -i 127.0.0.1 -a '<ask>' &
```

e creare un tunnel SSH dala macchina locale (quella da cui si desidera accedere all'interfaccia amministrativa) al server su cui web2py è in esecuzione (per esempio example.com) con:

```
ssh -L 8001:127.0.0.1:8001 username@example.com
```

Ora si può accedere all'interfaccia amministrativa sulla propria macchina tramite il browser web all'indirizzo `localhost:8001`.

Questa configurazione è sicura perchè **admin** non è raggiungibile quando il tunnel è chiuso (e l'utente e' scollegato).

> *Questa soluzione è sicura sugli host condivisi solamente se altri utenti non hanno accesso alla cartella che contiene web2py altrimenti gli utenti possono essere in grado di rubare i cookie di sessione direttamente dal server.*

11.4 Trucchi e consigli per la scalabilità

web2py è progettato per essere facile da installare e configurare. Questo però non significa che la sua efficienza e scalabilità siano penalizzate. Potrebbe

però essere necessario qualche aggiustamento per rendere l'installazione di web2py scalabile.

In questa sezione si ipotizza di avere installazioni multiple di web2py poste dietro un server NAT che fornisce un bilanciamento del carico locale. In questo caso web2py funziona immediatamente se vengono rispettate alcune condizioni. In particolare tutte le istanze di ogni applicazione di web2py devono accedere allo stesso database e devono vedere gli stessi file. Quest'ultima condizione può essere implementata condividendo le seguenti cartelle:

```
applications/myapp/sessions
applications/myapp/errors
applications/myapp/uploads
applications/myapp/cache
```

Queste cartelle devono essere condivise con un sistema che supporta il lock dei file. Possibili soluzioni sono il file system ZFS (sviluppato da Sun Microsystems) che è la scelta preferita, NFS (con NFS potrebbe essere necessario eseguire il demone `nlockmgr` per consentire il lock dei file) oppure Samba (SMB).

E' anche possibile condividere l'intera cartella di web2py o le intere cartelle delle applicazioni sebbene non sia una buona idea poichè questo creerebbe un inutile aumento dell'utilizzo della banda di rete.

La configurazione sopra discussa è molto scalabile perchè riduce il carico del database spostando si un disco condiviso le risorse che devono essere comuni tra i web server ma che non necessitano di transazioni (poichè si presuppone che solo un client alla volta acceda ai file di sessione, che la cache sia sempre acceduta con un lock globale e che i file caricati e gli errori siano scritti una sola volta e letti molte volte).

Idealmente sia il database che le cartelle condivise dovrebbero avere capacità RAID. Non fare l'errore di memorizzare il database e le cartelle condivise sullo stesso dispositivo perchè questo creerebbe un collo di bottiglia prestazionale.

In casi specifici potrebbe essere necessario eseguire ottimizzazioni aggiuntive che verranno discusse in seguito. In particolare sarà discusso come eliminare la necessità delle cartelle condivise e come memorizzare i relativi dati nel database. Sebbene questo sia possibile non è detto che sia una buona soluzione. Ci sono comunque valide ragioni per utilizzare questo approccio come nel caso in cui non sia possibile la condivisione delle cartelle tra i server web.

11.4.1 Sessioni nel database

E' possibile configurare web2py per memorizzare le sessioni in un database invece che nella cartella "sessions". Questo deve essere fatto per ogni applicazione di web2py sebbene il database utilizzato per memorizzare le sessioni possa essere lo stesso.

Con una connessione di database:

```
db = DAL(...)
```

è possibile memorizzare le sessioni in questo database (`db`) semplicemente indicando, nello stesso modello che stabilisce la connessione:

```
session.connect(request, response, db)
```

Se non è già esistente web2py crea in automatico nel database una tabella chiamata `web2py_session_`*appname* con i seguenti campi:

```
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" è una chiave univoca (UUID) utilizzata per identificare la sessione nel cookie. "session_data" contiene i campi di sessione memorizzati con cPickle.

Per ridurre l'accesso al database si dovrebbe evitare di memorizzare le sessioni quando non sono necessarie utilizzando:

```
session.forget()
```

Con questo accorgimento la cartella "sessions" non deve più essere condivisa in quanto non necessaria.

Notare che, se le sessioni sono disabilitate non si deve passare l'oggetto `session` *a* `form.accepts` *e non è possibile utilizzare nè* `session.flash` *nè CRUD.*

11.4.2 HAProxy, un bilanciatore di carico in altà disponibilità

In caso sia necessario avere più processi web2py in esecuzione su più macchine invece di memorizzare le sessioni in un database o nella cache è possibile utilizzare un bilanciatore di carico che gestisce le sessioni di connessione.

Pound (89) e HAProxy (90) sono due bilanciatori di carico HTTP con funzioni di reverse proxy che mettono a disposizione la gestione delle sessioni "sticky". Nel prossimo paragrafo sarà illustrato HAProxy perchè sembra essere più utilizzato nei sistemi di hosting commerciali.

Con sessione "sticky" si intende che una volta che un cookie di sessione è stato emesso il bilanciatore di carico indirizzerà sempre allo stesso server le richieste dal client associato alla sessione. Questo consente di memorizzare le sessioni sul file system locale senza aver bisogno di cartelle condivise.

Per utilizzare HAProxy:

Prima di tutto installarlo su una macchina di test Ubuntu:

```
sudo apt-get -y install haproxy
```

Modificare poi il file di configurazione "/etc/haproxy.cfg" più o meno nel seguente modo:

```
# this config needs haproxy-1.1.28 or haproxy-1.2.1

global
      log 127.0.0.1   local0
      maxconn 1024
      daemon

defaults
      log     global
      mode    http
      option  httplog
      option  httpchk
      option  httpclose
      retries 3
      option redispatch
      contimeout      5000
      clitimeout      50000
      srvtimeout      50000

listen 0.0.0.0:80
      balance url_param WEB2PYSTICKY
      balance roundrobin
      server  L1_1 10.211.55.1:7003  check
      server  L1_2 10.211.55.2:7004  check
      server  L1_3 10.211.55.3:7004  check
      appsession WEB2PYSTICKY len 52 timeout 1h
```

La direttiva `listen` indica a HAProxy su quale porta attendere le connessioni. La direttiva `server` indica a HAProxy dove trovare i server da bilanciare. La direttiva `appsession` crea una sessione sticky ed utilizza un cookie chiamato `WEB2PYSTICKY` per questo scopo.

Abilitare infine il file di configurazione ed avviare HAProxy:

```
/etc/init.d/haproxy restart
```

E' possibile trovare istruzioni equivalenti per configurare Pound alla URL:

```
http://web2pyslices.com/main/slices/take_slice/33
```

11.4.3 Pulizia delle sessioni

Se si decide di tenere le sessioni nel file system locale si deve tener conto del fatto che in un ambiente di produzione le sessioni si accumulano velocemente. web2py mette a disposizione uno script chiamato

```
scripts/sessions2trash.py
```

che, quando eseguito in background periodicamente cancella tutte le sessioni che non sono state accedute per un certo periodo di tempo. Il contenuto dello script è il seguente:

```
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:
         os.unlink(filename)
   time.sleep(SLEEP_MINUTES * 60)
```

Lo script può essere eseguito con il seguente comando:

```
nohup python web2py.py -S yourapp -R scripts/sessions2trash.py &
```

dove yourapp è il nome dell'applicazione.

11.4.4 Caricare i file in un database

Per default tutti i file caricati gestiti da SQLFORM sono rinominati in modo sicuro e memorizzati nella cartella "uploads" del file system. E' possibile configurare web2py per memorizzare i file caricati in un database.

Considerando la seguente tabella:

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

dove `dog.image` è di tipo *upload*. Per far sì che l'immagine caricata vada nello stesso record è necessario modificare la definizione della tabella aggiungendo un campo di tipo *blob* collegato al campo di upload:

```
db.define_table('dog',
    Field('name')
    Field('image', 'upload', uploadfield='image_data'),
    Field('image_data', 'blob'))
```

"image_data" è un nome arbitrario per il nuovo campo di tipo blob.

La terza linea indica a web2py di rinominare, come prima, in modo sicuro l'immagine, di memorizzare il nuovo nome dell'immagine nel campo "image" e di memorizzare i dati del file caricato in "image_data" invece che nel file system. Tutto questo è fatto automaticamente da SQLFORM e non è necessario cambiare altro codice dell'applicazione.

Con questo accorgimento la cartella "uploads" non deve essere più condivisa in quanto non più necessaria.

Su Google App Engine i file caricati sono memorizzati di default nel database senza necessità di definire *uploadfield* in quanto questo viene creato di default.

11.4.5 Ticket d'errore

Per default web2py memorizza i ticket d'errore sul file system locale. Non avrebbe senso memorizzare i ticket direttamente nel database in quanto la più comune causa d'errore in un ambiente di produzione è un errore di database. La memorizzazione dei ticket non è mai un collo di bottiglia in quanto un errore dovrebbe essere un evento sporadico e quindi in un ambiente di produzione con più server è adeguato memorizzarli in una cartella

condivisa. Non di meno. poichè solamente l'amministratore ha bisogno di recuperare i ticket d'errore è anche possibile memorizzarli nelle cartelle "errors" locali dei server e raccoglierli periodicamente o cancellarli.

Una possibilità è quella di spostare periodicamente tutti gli errori in un database. Per questo scopo è disponibile in web2py il seguente script:

```
scripts/tickets2db.py
```

che contiene:

```
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)
```

Questo script deve essere modificato prima di essere utilizzato. Cambiare la stringa DB_URI in modo che si connetta al server di database ed eseguirlo con il comando:

```
nohup python web2py.py -S yourapp -M -R scripts/tickets2db.py &
```

dove yourapp è il nome dell'applicazione.

Questo script, eseguito in background, ogni cinque minuti trasferisce i ticket al server di database in una tabella chiamata "ticket" e cancella i ticket nel file system locale. Se ALLOW_DUPLICATES è impostato a False saranno memorizzati solo i ticket che corrispondono a tipi di errore differenti.

Con questo accorgimento la cartella "errors" non deve più essere condivisa in quanto sarà acceduta solo localmente.

11.4.6 Memcache

web2py mette a disposizione due differenti tipi di cache: `cache.ram` e `cache.disk` che sebbene funzionino in ambienti distribuiti con server multipli, non si

comportano come ci si aspetterebbe. In particolare `cache.ram` gestirà la cache solo al livello del server ed è quindi inutilizzabile in un ambiente distribuito. Anche `cache.disk` gestirà la cache solo al livello del server a meno che la cartella "cache" non sia una cartella condivisa che supporta il lock dei file. Per questo motivo, invece di aumentare la velocità dell'applicazione questa potrebbe diventare un pesante collo di bottiglia prestazionale.

La soluzione è di non usare questi meccanismi di cache ma di utilizzare "memcache". web2py ha al suo interno una API per memcache. Per utilizzare memcache creare un nuovo modello di file, per esempio `0_memcache.py`, e in questo file scrivere (o aggiungere) il seguente codice:

```
from gluon.contrib.memcache import MemcacheClient
memcache_servers = ['127.0.0.1:11211']
cache.memcache = MemcacheClient(request, memcache_servers)
cache.ram = cache.disk = cache.memcache
```

La prima linea importa memcache. La seconda linea è una lista dei socket di memcache (server:port). La terza linea sostituisce `cache.ram` e `cache.disk` con memcache.

Si potrebbe scegliere di ridefinire solo uno di loro per definire un nuovo oggetto di cache che punta all'oggetto Memcache.

Con questo accorgimento la cartella "cache" non deve più essere una cartella condivisa in quanto non più necessaria.

Questo codice richiede che memcache sia in esecuzione su uno o più server della rete locale. E' necessario consultare la documentazione di memcache per la configurazione di tali server.

11.4.7 Sessioni in memcache

Se si ha bisogno delle sessioni ma non si vuole usare un bilanciatore di carico con le sessioni sticky è possibile memorizzare le sessioni in memcache:

```
from gluon.contrib.memdb import MEMDB
session.connect(request,response,db=MEMDB(cache.memcache))
```

11.4.8 Rimuovere le applicazioni

In un ambiente di produzione è bene non installare le applicazioni di default **admin**, **examples** e **welcome** perchè, sebbene queste applicazioni non occupino troppo spazio non sono necessarie.

Per rimuovere queste applicazioni è sufficiente cancellare la corrispondente cartella nella cartella "applications".

11.4.9 Utilizzare server di database multipli

In un ambiente ad elevate prestazioni si può avere una architettura di database di tipo master-slave con molti slave replicati ed alcuni server master replicati. Il DAL può gestire questa situazione e connettersi a differenti server in base ai parametri della richiesta. L'API per fare questo è stata descritta nel Capitolo 6. Ecco un esempio:

```
from random import shuffle
db=DAL(shuffle(['mysql://...1','mysql://...2','mysql://...3']))
```

In questo caso differenti richieste HTTP saranno servite da database differenti scelti in modo casuale ed ogni database avrà la stessa possibilità di essere selezionato.

E' anche possibile implementare un semplice round-robin (l'utilizzo cioè di un server dopo l'altro):

```
def fail_safe_round_robin(*uris):
    i = cache.ram('round-robin', lambda: 0, None)
    uris = uris[i:]+uris[:i] # rotate the list of uris
    cache.ram('round-robin', lambda: (i+1)%len(uris), 0)
```

```
    return uris
db = DAL(fail_safe_round_robin('mysql://...1','mysql://...2','mysql://...3'))
```

Questo tipo di configurazione è sicura nel senso che se il server di database assegnato ad una richiesta non è in grado di connettersi il DAL selezionerà il server successivo.

E' anche possibile connettersi a differenti database a seconda dell'azione richiesta nel controller. In una architettura master-slave alcune azioni eseguono solo letture nel database ed altre eseguono sia letture che scritture. Nel primo caso ci si può collegare ad un server di database slave, mentre nel secondo caso ci si deve connettere ad un server di database master:

```
if request.action in read_only_actions:
   db=DAL(shuffle(['mysql://...1','mysql://...2','mysql://...3']))
else:
   db=DAL(shuffle(['mysql://...3','mysql://...4','mysql://...5']))
```

dove 1,2,3 sono i server replicati e 3,4,5 i server slave.

11.5 Google App Engine

E' possibile eseguire il codice di web2py su Google App Engine (GAE) (13), incluso il codice del DAL, con alcune limitazioni. La piattaforma GAE fornisce diversi vantaggi rispetto alle normali soluzioni di hosting:

- Facilità di distribuzione in quanto GAE astrae completamente l'architettura di sistema.
- Scalabilità. GAE replicherà l'applicazione tante volte quante sono necessarie per servire tutte le richieste concorrenti.
- BigTable. Su GAE invece di un normale database le informazioni persistenti sono memorizzate in BigTable, il datastore per cui Google è famoso.

Le limitazioni sono:

- Non è possibile avere accesso in lettura o in scrittura al file system.
- Nessuna transazione.
- Non è possibile eseguire query nel datastore. In particolare non ci sono operatori di JOIN, LIKE, IN e DATE/DATETIME.

Questo significa che web2py non può memorizzare le sessioni, i ticket d'errore, i file di cache e i file caricati nel filesystem; queste informazioni devono pertanto essere memorizzate da qualche altra parte.

Pertanto su GAE web2py memorizza automaticamente i file caricati nel datastore, anche se il campo di tipo "upload" non ha l'attributo `uploadfield`. Bisogna invece specificare dove memorizzare le sessioni e i ticket d'errore. Questi possono anche essere memorizzati nel datastore:

```
db = DAL('gae')
session.connect(request,response,db)
```

Oppure in memcache:

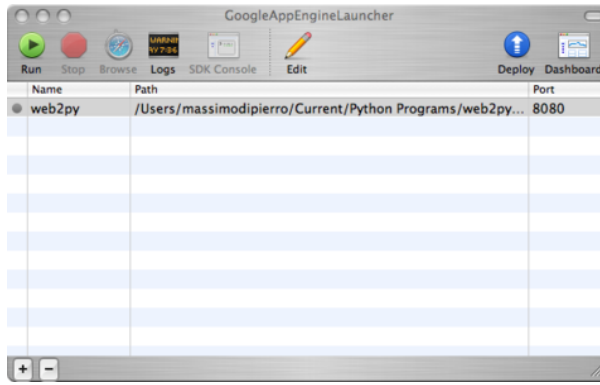
```
from gluon.contrib.gae_memcache import MemcacheClient
from gluon.contrib.memdb import MEMDB
cache.memcache = MemcacheClient(request)
cache.ram = cache.disk = cache.memcache

db = DAL('gae')
session.connect(request,response,MEMDB(cache.memcache))
```

L'assenza delle transazioni multi-entità e delle tipiche funzionalità dei database relazionali sono ciò che diversifica GAE rispetto agli altri ambienti di hosting. Questo è il prezzo da pagare per l'alta scalabilità. GAE è un eccellente piattaforma se queste limitazioni sono tollerabili, in caso contrario dovrebbe essere considerata una soluzione tradizionale di hosting con la possibilità di accedere ad un database relazionale.

Se un'applicazione web2py non può essere eseguita su GAE è a causa di una delle limitazioni sopra discusse. La maggior parte dei problemi possono

essere risolti rimuovendo le JOIN dalle query di web2py e denormalizzando il database.

Per caricare l'applicazione in GAE si può usare Google App Engine Launcher. Il software può essere scaricato da (13). Scegliere [File][Add Existing Application], impostare il percorso a quello della cartella web2py e premere il pulsante [Run] nela toolbar. Dopo aver testato il funzionamento localmente è possibile distribuire l'applicazione su GAE selezionando semplicemente il pulsante [Deploy] sulla toolbar, (sempre che si abbia un account GAE valido).

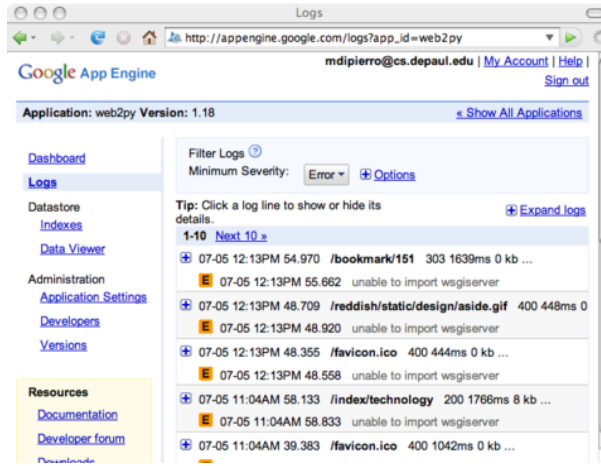


Sui sistemi Windows e Linux su può anche utilizzare la shell per distribuire l'applicazione:

```
cd ..
/usr/local/bin/dev_appserver.py web2py
```

Durante la distribuzione web2py ignora le applicazioni **admin**, **examples** e **welcome** poichè non sono necessarie. E' possibile modificare il file `app.yaml` per impedire la distribuzione di altre applicazioni.

Su GAE i ticket d'errore di web2py sono anche registrati nella console amministrativa di GAE dove i log possono essere acceduti e ricercati online.



E' possibile controllare se l'applicazione è in esecuzione su GAE valutando la variabile:

```
request.env.web2py_runtime_gae
```

Google App Engine supporta alcuni tipi di campo che non sono direttamente riconducibili ai tipi presente nel DAL, per esempio `StringListProperty`. E' comunque possibile utilizzare questi tipi in web2py con la seguente sintassi:

```
from gluon.contrib.gql import gae
db.define_table('myitem',
    Field('name'),
    Field('keywords', type=gae.StringListProperty())
```

In questo esempio il campo "keyword" è di tipo `StringListProperty` perciò il suo valore deve essere una lista di stringhe, come indicato dalla documentazione di GAE.

12

Altre ricette

12.1 Aggiornamenti

Nella pagina "site" dell'interfaccia amministrativa è presente un pulsante "upgrade now". In caso che questa opzione non funzioni (per esempio a causa di un lock su un file) l'aggiornamento manuale di web2py è comunque estremamente semplice:

Decomprimere l'ultima versione di web2py sull'installazione precedente.

In questo modo saranno aggiornate tutte le librerie e le applicazioni **admin**, **examples**, **welcome**. Verrà anche creato un nuovo file vuoto chiamato "NEWINSTALL". Al riavvio web2py cancellerà questo file e comprimerà l'applicazione **welcome** in "welcome.w2p". Questo file sarà utilizzato per creare le nuove applicazioni. web2py non aggiorna nessun file nelle altre applicazioni esistenti.

12.2 *Come distribuire le applicazioni in codice binario*

E' possibile allegare un'applicazione alla distribuzione binaria di web2py e distribuirle contemporaneamente. La licenza di web2py consente di fare questo purchè nella licenza dell'applicazione venga chiaramente indicato che questa è "in bundle" con web2py e venga fornito un link al sito `web2py.com`. Ecco come fare su Windows:

- Creare l'applicazione (nel solito modo)
- Tramite **admin**, compilare l'applicazione (un click)
- Tramite **admin** comprimere l'applicazione appena compilata (un altro click)
- Creare la cartella "myapp"
- Scaricare la distribuzione binaria di web2py per Windows
- Decomprimere la distribuzione binaria di web2py in "myapp" ed avviarla (due click)
- Tramite **admin** caricare l'applicazione precedentemente compilata e compressa con il nome "init" (un click)
- Creare il file "myapp/start.bat" contenente i comandi "cd web2py; web2py.exe"
- Creare il file "myapp/license" contentente la licenza dell'applicazione ed assicurarsi che contenga la frase "distribuita con una copia non modificata di web2py da web2py.com" ("distributed with an unmodified copy of web2py from web2py.com")
- Comprimere la cartella myapp nel file "myapp.zip"
- Distribuire e/o vendere "myapp.zip"

Quando gli utenti decomprimeranno "myapp.zip" e selezioneranno "run" vedranno l'applicazione "init" invece che l'applicazione "welcome". Non è necessario nessun prerequisito da parte dell'utente, non è neanche necessario che Python sia pre-installato.

Per Mac OS X il processo è simile ma non c'è necessità di creare il file ".bat".

12.3 Recuperare una URL esterna

Python include la libreria `urllib` per recuperare URL esterne:

```
import urllib
page = urllib.urlopen('http://www.web2py.com').read()
```

Questa libreria funziona bene, ma il modulo `urllib` non funziona su GAE. Google rende disponibile una differente API per il download delle URI. Questa API funziona solo su Google App Engine. Per rendere il codice portabile web2py include una funzione `fetch` che funziona sia su GAE che sulle altre installazioni di Python:

```
from google.tools import fetch
page = fetch('http://www.web2py.com')
```

12.4 Date descrittive

E' spesso utile rappresentare un *datetime* non nel formato "2009-07-25 14:34:56" ma come una stringa descrittiva del tipo "un anno fa". web2py mette a disposizione una funzione di utilità per fare questo:

```
d = datetime.datetime(2009,7,25,14,34,56)
from google.tools import prettydate
pretty_d = prettydate(d,T)
```

Il secondo argomento (T) deve essere passato per consentire l'internazionalizzazione dell'output.

12.5 Geocodifica

E' possibile convertire un indirizzo (per esempio "243 S Wabash Ave, Chicago, IL, USA") in coordinate geografiche (latitudine e longitudine):

```
from gluon.tools import geocode
address = '243 S Wabash Ave, Chicago, IL, USA'
(latitude, longitude) = geocode(address)
```

La funzione `geocode` richiede l'accesso ad Internet e si connette al servizio di geocodifica di Google. La funzione ritorna `(0,0)` in caso d'errore. Notare che il servizio di geocodifica di Google pone un limite al numero di richieste, come indicato nell'accordo di utilizzo del servizio. La funzione `geocode` è costruita utilizzando la funzione `fetch` e pertanto funziona con GAE.

12.6 Paginazione

Questa ricetta è un comodo trucco per minimizzare l'accesso al database in caso di paginazione, come nel caso in cui sia necessario visualizzare una lista di righe di un database e si vuole che queste siano distribuite su più pagine web.

Creare un'applicazione **primes** che memorizza i primi mille numeri primi in un database.

Ecco il modello `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
```

Creare ora un'azione `list_items` nel controller "default.py" simile alla seguente:

```
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)
```

Notare che questo codice recupera dal database un elemento in più del dovuto (20 + 1). L'elemento extra, se presente, indica alla vista che c'è almeno un altra pagina da visualizzare.

Ecco la vista "default/list_items.html":

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

{{for i,row in enumerate(rows):}}
{{if i==items_per_page: break}}
{{=row.value}}<br />
{{pass}}

{{if page:}}
<a href="{{=URL(args=[page-1])}}">previous</a>
{{pass}}

{{if len(rows)>items_per_page:}}
<a href="{{=URL(args=[page+1])}}">next</a>
{{pass}}
```

In questo modo è stata ottenuta la paginazione con una singola SELECT per ogni chiamata all'azione, e quella SELECT recupera solo una riga in più del dovuto.

12.7 httpserver.log e il formato di file del log

Il server web di web2py memorizza tutte le richieste in un file chiamato:

```
httpserver.log
```

nella cartella d'installazione di web2py. Un nome di file e una posizione alternativa possono essere specificati tramite opzioni della linea di comando.

Delle nuove linee sono aggiunte alla fine del file ogni volta che una richiesta è eseguita. Ogni linea è simile alla seguente:

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

Il formato è:

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

Dove:

- ip è l'indirizzo IP del client che ha inviato la richiesta
- timestamp è la data e l'ora della richiesta in formato ISO 8601 (YYYY-MM-DDT HH:MM:SS)
- method può essere GET oppure POST
- path è il percorso richiesto dal client
- protocol è il protocollo utilizzato per la risposta, solitamente HTTP/1.1
- status è uno dei codici di status HTTP (92)
- time_taken è il tempo impiegato dal server per processare la richiesta, in secondi, escluso il tempo di upload/download.

Nel sito delle applicazioni già pronte di web2py (34) è possibile trovare un'applicazione per l'analisi del log. Questo log è disabilitato per default quando si utilizza mod_wsgi in quanto sarebbe lo stesso di quello generato da Apache.

12.8 Popolare il database con dati fittizi

Per testare un'applicazione potrebbe essere conveniente popolare le tabelle di database con dati fittizi. web2py include un classificatore Bayesiano già

addestrato per generare dati fittizi ma leggibili per questo scopo.

Ecco il modo più semplice di utilizzarlo:

```
from gluon.contrib.populate import populate
populate(db.mytable,100)
```

Questo codice inserirà 100 record fittizi in db.mytable. Tenterà di farlo in modo intelligente generando un testo breve per i campi stringa, un testo più lungo per i campi testo, numeri interi, doppi, date, orari, booleani, ecc. per i corrispondenti campi. Tenterà inoltre di rispettare i vincoli imposti dai validatori. Per i campi contenenti la parola "name" tenterà di generare dei nomi fittizi. Per i campi di riferimento genererà delle referenze valide.

Se si hanno due tabelle (A e B) dove B fa riferimento ad A, assicurarsi di popolare prima A e poi B. Poichè il popolamento è eseguito in una transazione non tentare di popolare troppi record in una volta sola, in particolare se sono presenti referenze. Eseguire invece un ciclo di 100 record fittizi per volta.

```
for i in range(10):
    populate(db.mytable,100)
    db.commit()
```

E' possibile addestrare il classificatore Bayesiano per imparare da un testo e generare testo fittizio simile ma senza significato:

```
from gluon.contrib.populate import Learner, IUP
ell=Learner()
ell.learn('some very long input text ...')
print ell.generate(1000,prefix=None)
```

12.9 Invio degli SMS

L'invio di messaggi SMS da un'applicazione web2py richiede un servizio di terze parti in grado di consegnare il messaggio al destinatario. Questo tipo di servizi solitamente non è gratuito, ma la situazione cambia da paese a paese.

Non è stato possibile testare molti servizi di questo tipo poichè le compagnie telefoniche bloccano le email di richiesta di invio SMS perchè potrebbero essere fonte di spam.

Un modo migliore è quello di usare direttamente le compagnie telefoniche per inviare gli SMS. Ogni compagnia telefonica ha un indirizzo email associato con ogni numero di telefono e l'SMS può essere spedito come una email associata a quel numero. web2py disponde di un modulo per questo processo:

```
from gluon.contrib.sms_utils import SMSCODES, sms_email
email = sms_email('1 (111) 111-1111','T-Mobile USA (tmail)')
mail.sent(to=email,subject='test',message='test')
```

SMSCODES è un dizionario che mappa i nomi delle principali compagnie telefoniche al dominio dell'email del numero di telefono. La funzione `sms_email` richiede un numero di telefono (come una stringa) e il nome della compagnia telefonica e restituisce l'indirizzo email del telefono.

12.10 Accettare pagamenti con carta di credito

web2py mette a disposizione diversi modi per accettare i pagamenti con carta di credito nelle applicazioni come, per esempio

- **Google Checkout Plugin**

```
http://web2py.com/plugins/static/web2py.plugin.google_checkout.w2p
```

- **Paypal**

```
http://www.web2pyslices.com/main/slices/take_slice/9
```

- **Authorize.Net**

I primi due meccanismi delegano il processo di autenticazione dell'utente ad un servizio esterno. Sebbene questo sia il modo migliore per la sicurezza (l'applicazione non gestisce alcuna informazione sulle carte di credito) rende il processo di pagamento complicato (l'utente deve collegarsi due volte, una all'applicazione e l'altra, per esempio, a Google) e non consente all'applicazione di gestire i pagamenti ricorrenti in modo automatico.

A volte è necessario avere più controllo. Per questo motivo web2py fornisce l'integrazione con le API di Authorize.Net (il modulo è stato sviluppato da John Conde e leggermente modificato). Ecco un esempio di un workflow e di tutte le variabili che sono esposte:

```
from gluon.contrib.AuthorizeNet import AIM
payment = AIM(login='cnpdev4289',
              transkey='SR2P8g4jdEn7vFLQ',
              testmod=True)
payment.setTransaction(creditcard, expiration, total, cvv, tax, invoice)
payment.setParameter('x_duplicate_window', 180) # three minutes duplicate windows
payment.setParameter('x_cust_id', '1324')       # customer ID
payment.setParameter('x_first_name', 'Agent')
payment.setParameter('x_last_name', 'Smith')
payment.setParameter('x_company', 'Test Company')
payment.setParameter('x_address', '1234 Main Street')
payment.setParameter('x_city', 'Townsville')
payment.setParameter('x_state', 'NJ')
payment.setParameter('x_zip', '12345')
payment.setParameter('x_country', 'US')
payment.setParameter('x_phone', '800-555-1234')
payment.setParameter('x_description', 'Test Transaction')
payment.setParameter('x_customer_ip', socket.gethostbyname(socket.gethostname()))
payment.setParameter('x_email', 'you@example.com')
payment.setParameter('x_email_customer', False)

payment.process()
if payment.isApproved():
    print 'Response Code: ', payment.response.ResponseCode
    print 'Response Text: ', payment.response.ResponseText
    print 'Response: ', payment.getResultResponseFull()
    print 'Transaction ID: ', payment.response.TransactionID
    print 'CVV Result: ', payment.response.CVVResponse
    print 'Approval Code: ', payment.response.AuthCode
    print 'AVS Result: ', payment.response.AVSResponse
elif payment.isDeclined():
    print 'Your credit card was declined by your bank'
elif payment.isError():
```

```
    print 'It did not work'
print 'approved',payment.isApproved()
print 'declined',payment.isDeclined()
print 'error',payment.isError()
```

Questo codice utilizza un account di test fittizio. E' necessario registrarsi con Authorize.Net (non si tratta di un servizio gratuito). e fornire al costruttore AIM il proprio login, la propria transkey e impostare testmode a True o a False.

12.11 API per Twitter

Ecco degli esempi veloci su come inviare e ricevere dei tweet. Non è necessaria nessuna libreria di terze parti, poichè Twitter utilizza delle semplici API di tipo REST.

Ecco un esempio di come inviare un tweet:

```
def post_tweet(username,password,message):
    import urllib, urllib2, 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())
```

Ecco un esempio di come ricevere i tweet:

```
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)
```

Per operazioni più complesse fare riferimento alla documentazione delle API di Twitter.

12.12 Streaming di file virtuali

E' comune, in caso di attacco, subire una scansione del sito alla ricerca delle possibili vulnerabilità. Gli attaccanti utilizzano scanner di sicurezza (come Nessus) per esplorare i siti alla ricerca di script che contengono vulnerabilità note. Un'analisi dei log del server web attaccato o direttamente del database di Nessus rivela che la maggior parte delle vulnerabilità conosciute sono presenti negli script PHP e ASP. Sebbene web2py non abbia queste vulnerabilità potrebbe subire ugualmente una scansione di questo tipo. Per far capire agli attaccanti che stanno perdendo il loro tempo è possibile rispondere a questi attacchi.

Una possibilità è quella di redirigere tutte le richieste di tipo.php e.asp (e ogni altra richiesta sospetta) ad una azione fittizia che risponderà all'attaccante tenendolo occupato per un notevole lasso di tempo. A quel punto l'attaccante abbandonerà l'attacco e probabilmente non lo ripeterà.

Questa ricetta richiede due parti.

La prima è un'applicazione dedicata chiamata **jammer** con il seguente controller "default.py":

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

Quando questa azione viene chiamata risponde con uno stream di dati composto da caratteri "x", 40.000 caratteri alla volta.

La seconda parte è il file "route.py" che reindirizza qualsiasi richiesta che finisce in.php,.asp, ecc. (sia maiuscolo che minuscolo) al controller appena

definito.

```
route_in=(
 ('.*\.(php|PHP|asp|ASP|jsp|JSP)','jammer/default/jam'),
)
```

La prima volta che un attaccante prova ad attaccare il sito si potrebbe avere un leggero sovraccarico ma solitamente gli attacchi vengono interrotti e non si ripetono.

12.13 Jython

web2py normalmente è eseguito con CPython (l'interprete Python scritto in C) ma può anche essere eseguito con Jython (l'interprete Python scritto in Java). In questo modo web2py può essere eseguito all'interno di un'infrastruttura Java. web2py può essere eseguito in Jython direttamente, anche se ci sono alcuni passi da eseguire per impostare Jython e zxJDBC (l'adattatore di database di Jython):

- Scaricare il file "jython_installer-2.5.0.jar" (o 2.5.x) da `Jython.org`
- Installarlo con:

```
java -jar jython_installer-2.5.0.jar
```

- Scaricare ed installare "zxJDBC.jar" da (94)
- Scaricare ed installare "sqlitejdbc-v056.jar" da (95)
- Aggiungere zxJDBC e sqlitejdbc al CLASSPATH di Java
- Avviare web2py con Jython

```
/path/to/jython web2py.py
```

Al momento della scrittura di questo manuale sono supportati sy Jython solamente gli adattatori di database per `sqlite` e `postgres`.

13

Componenti e plugin

I componenti ed i plugin sono funzionalità relativamente nuove di web2py e c'è ancora disaccordo tra gli sviluppatori su cosa siano e cosa dovrebbero essere. La maggior parte della confusione è generata dall'uso che si fa di questi termini in altri progetti software e dal fatto che gli sviluppatori stiano ancora lavorando per definirne le specifiche.

Queste comunque sono funzionalità importanti di web2py ed è quindi necessario fornire una definizione che sebbene non definitiva sia consistente con gli schemi di programmazione discussi in questo capitolo e che affrontano le seguenti due problematiche:

- Come costruire applicazioni modulari che riducano il carico del server e massimizzino il riutilizzo del codice.
- Come distribuire parti di codice in un modo il più possibile immediato (*plugin*-and-play).

I *componenti* affrontano il primo problema; i *plugin* affrontano il secondo.

13.1 Componenti

Un **componente** *è una parte funzionalmente autonoma di una pagina web.*

Un componente può essere composto di moduli, controller e viste. Un componente non ha nessun prerequisito particolare, se non quello di poter essere inserito in un singolo tag di una pagina web (per esempio un DIV, uno SPAN o un IFRAME) dove esegue il suo compito indipendentemente dal resto della pagina. I componenti sono caricati nella pagina e comunicano con il controller tramite Ajax.

Un esempio di componente è un "componente per i commenti" che è contenuto in un DIV e mostra i commenti degli utenti insieme ad un form per l'invio di un nuovo commento. Quando il form è compilato viene inviato al server tramite una chiamata Ajax, la lista viene aggiornata e il commento è memorizzato dal server nel database. Alla fine il contenuto del DIV è aggiornato senza ricaricare il resto della pagina. La funzione LOAD di web2py rende questo compito estremamente semplice da eseguire senza nessuna conoscenza della programmazione Javascript o Ajax.

L'obiettivo è quello di riuscire a sviluppare un'applicazione web assemblando i componenti nelle pagine.

Con una semplice applicazione "test" che estende l'applicazione di default con il seguente modello "models/db_comments.py":

```
db.define_table('comment',
   Field('body','text',label='Your comment'),
   Field('posted_on','datetime',default=request.now),
   Field('posted_by',db.auth_user,default=auth.user_id))
db.comment.posted_on.writable=db.comment.posted_on.readable=False
db.comment.posted_by.writable=db.comment.posted_by.readable=False
```

e con un'azione nel "controllers/comments.py" definita come:

```
@auth.requires_login()
def post():
```

```
    return dict(form=crud.create(db.comment),
                comments=db(db.comment.id>0).select())
```

e con la corrispondente vista in "views/comments/post.html":

```
{{extend 'layout.html'}}
{{for comment in comments:}}
<div class="comment">
  on {{=comment.posted_on}} {{=comment.first_name}}
  says <span class="comment_body">{{=comment.body}}</span>
</div>
{{pass}}
{{=form}}
```

si può accedere all'azione all'indirizzo:

```
http://127.0.0.1:8000/test/comments/post
```

[image]

Per ora non c'è nulla di speciale in questa azione ma è possibile trasformarla in un componente semplicemente definendo una nuova vista con estensione ".load" che non estende il layout. Ecco il contenuto di "views/comments/post.load":

```
{{#extend 'layout.html' <- notice this is commented out!}}
{{for comment in comments:}}
<div class="comment">
  on {{=comment.posted_on}} {{=comment.first_name}}
  says <span class="comment_body">{{=comment.body}}</span>
</div>
{{pass}}
{{=form}}
```

E' possibile accedere a questa vista con l'URL:

```
http://127.0.0.1:8000/test/comments/post.load
```

e sarà visualizzata come:

[image]

Questo è un componente che può essere inserito in ogni altra pagina semplicemente con:

```
{{=LOAD('comments','post.load',ajax=True)}}
```

Per esempio si può aggiungere nel controller "controllers/default.py":

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

e nella vista corrispondente si può aggiungere il componente:

```
{{extend 'layout.html'}}
<h1>{{='bla '*100}}</h1>
{{=LOAD('comments','post.load',ajax=True)}}
```

Accedendo alla pagina

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

sarà mostrato il contenuto normale e il componente dei commenti:

[add image]

Il componente `{{=LOAD( ... )}}` è visualizzato tramite il seguente HTML:

```
<script type="text/javascript"><!--
web2py_component("/test/comment/post.load","c282718984176")
//--></script><div id="c282718984176">loading...</div>
```

(il codice effettivamente generato dipende dalle opzioni passate alla funzione LOAD). La funzione `web2py_component(url,id)` è definita in "views/web2py_ajax.html" ed esegue tutte le operazioni necessarie: chiama la `url` tramite Ajax e inserisce la risposta nel DIV con l'`id` corrispondente; intercetta ogni invio del form nel DIV e lo invia tramite Ajax. Il target Ajax è sempre lo stesso DIV.

La sintassi completa dell'helper LOAD è il seguente:

```
LOAD(c=None, f='index', args=[], vars={},
     extension=None, target=None,
```

```
    ajax=False, ajax_trap=False,
    url=None):
```

Dove:

- i primi due argomenti `c` e `f` sono il controller e la funzione che si vuole chiamare.
- `args` e `vars` sono gli argomenti che si vuol passare alla funzione. Il primo è una lista, il secondo è un dizionario.
- `extension` è una estensione opzionale. Notare che l'estensione può anche essere passata come parte della funzione (come, per esempio `f='index.load'`).
- `target` è l'`id` del DIV. Se non è specificato verrà generato un `id` casuale.
- `ajax` deve essere impostato a `True` se il DIV deve essere riempito tramite una chiamata Ajax e deve essere impostato a `False` se il DIV deve essere riempito prima che la pagina sia ritornata (evitando così la chiamata Ajax).
- `ajax_trap=True` significa che ogni invio del form nel DIV deve essere catturato ed inviato tramite Ajax e che la risposta deve essere inserita nel DIV. `ajax_trap=False` significa che il form deve essere inviato normalmente, ricaricando l'intera pagina. `ajax_trap` è ignorato e considerato `True` nel caso che `ajax=True`.
- `url`, se specificato, ignora i valori di `c`, `f`, `args`, `vars` ed `extension` e carica il componente alla `url` indicata. Questo parametro è utilizzato per caricare un componente da un'altra applicazione (non necessariamente creata con web2py).

Se non è specificata nessuna vista `.load` viene utilizzata la vista `generic.load` che visualizza senza layout il dizionario ritornato dall'azione. Questa vista generica funziona meglio se il dizionario contiene un singolo elemento.

Se si carica un componente con estensione `.load` e la relativa azione reindirizza ad un'altra azione (per esempio per il login), l'estensione `.load` viene propagata alla nuova URL (quella a cui si viene reindirizzati) che viene caricata con l'estensione `.load`.

Quando un'azione di un componente è chiamata tramite Ajax web2py passa due header HTTP aggiuntivi con la richiesta:

```
web2py-component-location
web2py-component-element
```

che possono essere acceduti nell'azione tramite le variabili:

```
request.env.http_web2py_component_location
request.env.http_web2py_component_element
```

Il primo contiene la URL della pagina che ha chiamato l'azione del componente. Il secondo contiene l'`id` del DIV che conterrà la risposta.

L'azione del componente può anche memorizzare informazioni in due header speciali che saranno interpretati dalla pagina completa durante la risposta:

```
web2py-component-flash
web2py-component-command
```

e possono essere acceduti tramite:

```
response.headers['web2py-component-flash']='....'
response.headers['web2py-component-command']='...'
```

Il primo contiene del test che deve apparire nel messaggio alla risposta. Il secondo contiene codice Javascript che si vuole far eseguire alla risposta. Non può contenere il carattere di ritorno a capo.

Come esempio ecco un componente per la richiesta di un form di contatti. L'azione in "controllers/contact/ask.py" consente all'utente di porre una domanda che il componente invierà all'amministratore tramite email. Infine visualizza il messaggio "thank you" e rimuove il componente dalla pagina:

```
def ask():
    form=SQLFORM.factory(
        Field('your_email',requires=IS_EMAIL()),
        Field('question',requires=IS_NOT_EMPTY()))
    if form.accepts(request.vars,session):
        if mail.send(to='admin@example.com',
                     subject='from %s' % form.vars.your_email,
```

```
                message = form.vars.question):
            div_id = request.env.http_web2py_component_element
            command="jQuery('#%s').hide()" % div_id
            response.headers['web2py-component-command']=command
            response.headers['web2py-component-flash']='thanks you'
        else:
            form.errors.your_email="Unable to send the email"
    return dict(form=form)
```

Le prime quattro linee definiscono il form e lo validano. L'oggetto `mail` utilizzato per inviare la domanda è definito nell'applicazione di default. Le ultime quattro linee implementano tutta la logica specifica del componente leggendo i dati dagli header della richiesta HTTP ed impostando gli header HTTP nella risposta.

Ora è possibile inserire questo form in qualsiasi pagina con:

```
{{=LOAD('contact','ask.load',ajax=True)}}
```

Notare che non è stata definita nessuna vista `.load` per il componente `ask`. Questo non è necessario perchè il componente ritorna un solo oggetto (form) e per questo è sufficiente la vista "generic.load".

Quando è usata la vista "generic.load" si può utilizzare la sintassi:

```
response.flash='...'
```

che è equivalente a:

```
response.headers['web2py-component-flash']='...'
```

13.2 *Plugin*

Un **plugin** *è un sotto-insieme di qualsiasi file di un'applicazione.*

e si intende realmente "qualsiasi":

- Un *plugin* non è un modulo, non è un modello, non è un controller e nemmeno una vista, ma può contenere moduli, modelli, controllers e viste.
- Un *plugin* non deve essere funzionalmente autonomo e può dipendere da altri plugin o codice specifico.
- Un *plugin* non è un *sistema di plugin* pertanto i plugin non sono nè "registrati" nè "isolati" sebbene esistano delle regole per mantenere un certo isolamento.
- In questo capitolo si parla di plugin per l'applicazione non plugin per web2py.

Perchè questa funzionalità è chiamata *plugin*? Perchè rende disponibile un meccanismo per includere un sotto-insieme di un'applicazione in un'altra applicazione. Da questa prospettiva qualsiasi file di un'applicazione può essere considerato un plugin.

Quando un'applicazione viene distribuita i suoi plugin sono "impacchettati" e distribuiti con essa.

In pratica l'applicazione **admin** fornisce un'interfaccia per impacchettare e spacchettare i plugin separatamente dall'applicazione. I file e le cartelle dell'applicazione che hanno nomi che iniziano con `plugin_`*name* possono essere impacchettati insieme in un file chiamato:

`web2py.plugin.`*name*`.w2p` e distribuiti insieme.

[ADD IMAGE]

I file che compongono un plugin non sono trattati da web2py in modo differente da ogni altro file tranne per il fatto che **admin** comprende (dai loro nomi) che devono essere distribuiti insieme e li visualizza in una pagina separata:

[ADD IMAGE]

Per la definizione di plugin data all'inizio di questo capitolo i plugin hanno un utilizzo più ampio oltre al fatto di essere riconosciuti dall'applicazione **admin**.

In pratica in web2py si possono utilizzare due tipi di plugin:

- *plugin di componenti*. Sono plugin che contengono componenti, come definiti nella sezione precedente. Un plugin di componenti può contenerne uno o più di uno. Per esempio potrebbe esistere un `plugin_comments` che contiene il componente *comments* della sezione precedente. Un altro esempio potrebbe essere `plugin_tagging` che contiene un componente *tagging* ed un componente *tag-cloud* che condividono alcune tabelle di database definite dal plugin stesso.
- *plugin di layout*. Sono plugin che contengono le viste per un layout e i file statici necessari al layout stesso. Quando il plugin è applicato all'applicazione le dà un look differente.

In base alle definizioni precedenti i componenti creati nella precedente sezione, come per esempio "controllers/contact.py", sono già considerati plugin. Possono essere spostati da un'applicazione ad un'altra per utilizzare i componenti che definiscono. Non sono però riconosciuti come plugin dall'applicazione **admin** perchè non c'è nulla che li etichetta come plugin. Ci sono quindi ancora due problemi da risolvere:

- Utilizzare una convenzione per i nomi dei file che compongono il plugin in modo che l'applicazione **admin** possa riconoscerli come appartenenti allo stesso plugin.
- Se il plugin ha file di modello stabilire una convenzione per impedire agli oggetti che sono definiti nel modello di mischiarsi nello stesso *namespace* dell'applicazione.

Queste sono le regole che dovrebbe seguire un plugin chiamato *name*:

Regola 1: I plugin di tipo modello e di tipo controller dovrebbero essere chiamati rispettivamente:

- `models/plugin_`*name*`.py`
- `controllers/plugin_`*name*`.py`

e quelli di tipo vista, modulo, statici e file privati dovrebbero essere chiamati:

- `views/plugin_`*name*`/`
- `modules/plugin_`*name*`/`
- `static/plugin_`*name*`/`
- `private/plugin_`*name*`/`

Regola 2: I plugin di tipo modello possono definire solamente oggetti i cui nomi iniziano con:

- `plugin_`*name*
- `Plugin`*Name*
- `_`

Regola 3: I plugin di tipo modello possono definire solamente variabili i cui nomi iniziano con:

- `session.plugin_`*name*
- `session.Plugin`*Name*

Regola 4: I plugin dovrebbero includere la licenza e la documentazione che dovrebbero essere presenti in:

- `static/plugin_`*name*`/license.html`

- `static/plugin_`*name*`/about.html`

Regola 5: Un plugin può fare affidamento solo sull'esistenza degli oggetti globali definiti nell'applicazione di base, come per esempio:

- una connessione al database chiamata `db`
- una istanza di `Auth` chiamata `auth`
- una istanza di `Crud` chiamata `crud`
- una istanza di `Service` chiamata `service`

Alcuni plugin possono essere più complessi ed avere parametri di configurazione per il caso in cui esista più di una istanza di `db`.

Regola 6: Se un plugin necessita di parametri di configurazione questo dovrebbero essere aggiunti tramite **PluginManager**, come descritto in seguito.

Seguendo le regole sopra indicate si può essere sicuri che:

- **admin** riconosce tutti i file e le cartelle `plugin_`*name* come facenti parte di una singola entità.
- I plugin non interferiscono l'uno con l'altro.

Queste regole non risolvono però il problema delle diverse versioni del plugin e delle sue dipendenze.

13.2.1 Plugin di componenti

I plugin di componenti sono plugin che definiscono dei componenti. I componenti solitamente accedono al database con i loro modelli.

Ecco come trasformare il precedente componente `comments` in un plugin `comments_plugin` mantenendo lo stesso identico codice scritto precedentemente ma seguento tutte le regole dei plugin illustrate in precedenza.

Creare, per prima cosa, un modello chiamato "models/plugin_comments.py":

```
db.define_table('plugin_comments_comment',
   Field('body','text',label='Your comment'),
   Field('posted_on','datetime',default=request.now),
   Field('posted_by',db.auth_user,default=auth.user_id))
db.plugin_comments_comment.posted_on.writable=False
db.plugin_comments_comment.posted_on.readable=False
db.plugin_comments_comment.posted_by.writable=False
db.plugin_comments_comment.posted_by.readable=False

def plugin_comments():
    return LOAD('plugin_comments','post',ajax=True)
```

(le ultime due linee di codice definiscono una funzione che semplifica l'inserimento del plugin in un'applicazione).

Definre quindi un controller "controllers/plugin_comments.py"

```
@auth.requires_login()
def post():
    comments = db.plugin_comments_comment
    return dict(form=crud.create(comment),
                comments=db(comment.id>0).select())
```

Creare, come terza cosa, una vista chiamata "views/plugin_comments/post.load":

```
{{for comment in comments:}}
<div class="comment">
  on {{=comment.posted_on}} {{=comment.first_name}}
  says <span class="comment_body">{{=comment.body}}</span>
</div>
{{pass}}
{{=form}}
```

E' ora possibile utilizzare **admin** per prepare il plugin per la distribuzione. Il plugin sarà salvato da admin come:

```
web2py.plugin.comments.w2p
```

Il plugin può essere ora usato in qualsiasi vista semplicmente installandolo in **admin** tramite la pagina **edit** ed aggiungendolo ad una vista con:

```
{{=plugin_comments()}}
```

Ovviamente è possibile rendere il plugin più complesso con componenti che richiedono parametri e opzioni di configurazione. Più i plugin sono complessi più si deve far attenzione ad evitare collisioni di nomi. **Plugin Manager** è progettato per evitare questi problemi.

13.2.2 Plugin Manager

La classe `PluginManager` è definita in `gluon.tools`. Prima di spiegare come funziona ecco come si utilizza.

E' possibile migliorare il precedente plugin `comments_plugin` con la possibilità di aggiungere la personalizzazione di:

```
db.plugin_comments_comment.body.label
```

senza doverne modificare il codice.

Prima di tutto, riscrivere il plugin "models/plugin_comments.py" nel seguente modo:

```
db.define_table('plugin_comments_comment',
   Field('body','text',label=plugin_comments.comments.body_label),
   Field('posted_on','datetime',default=request.now),
   Field('posted_by',db.auth_user,default=auth.user_id))

def plugin_comments()
    from gluon.tools import PluginManager
    plugins = PluginManager('comments',body_label='Your comment')

    comment = db.plugin_comments_comment
    comment.label=plugins.comments.body_label
    comment.posted_on.writable=False
    comment.posted_on.readable=False
    comment.posted_by.writable=False
```

```
    comment.posted_by.readable=False
    return LOAD('plugin_comments','post',ajax=True)
```

Tutto il codice (tranne la definizione della tabella) è incapsulato in una singola funzione che crea un'istanza di `PluginManager`.

E' ora possibile configurare in ogni modello dell'applicazione (per esempio in "models/db.py") il plugin nel seguente modo:

```
from gluon.tools import PluginManager
plugins = PluginManager()
plugins.comments.body_label=T('Post a comment')
```

L'oggetto `plugins` *è istanziato nell'applicazione di base in "models/db.py"*

L'oggetto PluginManager è un oggetto *singleton* di tipo Storage che contiene altri oggetti di tipo Storage con il blocco a livello del thread. Questo significa che si può istanziare tante volte quante è necessario all'interno di una applicazione, anche con nomi diversi, ma agisce come se fosse un'unica istanza di PluginManager.

In particolare ogni file di plugin può definire il suo oggetto PluginManager e registrarlo, insieme ai suoi parametri di default, con:

```
plugins = PluginManager('name', param1='value', param2='value')
```

E' possibile modificare i parametri in altre parti del codice (per esempio in "models/db.py") con:

```
plugins = PluginManager()
plugins.name.param1 = 'other value'
```

E' possibile inoltre configurare plugin diversi nello stesso codice:

```
plugins = PluginManager()
plugins.name.param1 = '...'
plugins.name.param2 = '...'
plugins.name1.param3 = '...'
```

```
plugins.name2.param4 = '...'
plugins.name3.param5 = '...'
```

Quando il plugin è definito l'oggetto PluginManager deve avere due argomenti: il nome del plugin e degli argomenti opzionali con nome che sono i suoi parametri di default. Invece quando i plugin sono configurati il costruttore di PluginManager non deve avere argomenti. La configurazione deve precedere la definizione del plugin (per esempio deve essere in un modello che abbia un nome che alfabeticamente sia prima degli altri modelli).

13.2.3 Plugin di layout

I plugin di layout sono più semplici dei plugin di componenti perchè solitamente non contengono codice ma solamente viste e file statici. Ci sono comunque delle regole da seguire: Primo, creare una cartella chiamata "static/plugin_layout_*name*/" (dove *name* è il nome del layout) e posizionare in essa tutti i file statici del plugin.

Secondo, creare un file di layout chiamato "views/plugin_layout_*name*/layout.html" che contiene il layout e i link alle immagini, ai file CSS e ai file JS in "static/-plugin_layout_*name*/"

Terzo, modificare la vista "views/layout.html" aggiungendo:

```
{{include 'plugin_layout_name/layout.html'}}
```

Il beneficio di questo design è che gli utenti di questo plugin possono installre più layout e scegliere quello da applicare all'applicazione semplicemente modificando il file "views/layout.html". Inoltre "views/layout.html" non verrà incluso da **admin** insieme al plugin in modo da non rischiare che un altro plugin sovrascriva il codice scritto dall'utente.

13.3 *plugin_wiki*

AVVISO: plugin_wiki è ancora in una fase iniziale di sviluppo e pertanto non ne è garantita la retro-compatibilità allo stesso livello delle funzioni centrali di web2py.

plugin_wiki è un plugin "con gli steroidi". Infatti definisce diversi componenti e può cambiare il modo di sviluppare le applicazioni:

E' possibile scaricarlo da:

```
http://web2py.com/examples/static/web2py.plugin.wiki.w2p
```

L'idea alla base di **plugin_wiki** è che la maggior parte delle applicazioni includono pagine semi-statiche. Si tratta di pagine che non includono una logica complessa e che contengono testo strutturato (come per esempio una pagina di help), immagini, audio, video, form CRUD o un insieme di componenti standard (commenti, tag, grafici, mappe, ecc.). Queste pagine possono essere pubbliche oppure richiedere un login o avere altre limitazione di autorizzazione. Possono essere raggiunte tramite un menu o da un modulo di autocomposizione. **plugin_wiki** mette a disposizione un modo semplice per aggiungere ad una applicazione web2py pagine che rientrano in questa categoria.

In particolare **plugin_wiki** mette a disposizione:

- Un'interfaccia di tipo wiki che consente di aggiungere pagine alla propria applicazione e referenziarle con uno *slug*. Queste pagine, chiamate pagine wiki, hanno una versione e sono memorizzate in un database.
- Pagine pubbliche e pagine private (che richiedono l'autenticazione). Se una pagina richiede l'autenticazione può anche richiedere una particolare autorizzazione per un utente.
- Tre livelli: 1, 2, 3. Al livello 1 le pagine possono includere solamente testo, immagini, audio e video. Al livello 2 le pagine possono anche includere

dei widget (questi sono componenti, definiti nella sezione precedente che possono essere inclusi nelle pagine wiki). Al livello 3 le pagine possono anche includere codice di template di web2py.

- La possibilità di modificare le pagine con la sintassi *markmin* o con tag HTML utilizzando un editor WYSIWYG (What you see is what you get).
- Una raccolta di widget: implementati come componenti sono auto-documentati e possono essere inseriti nelle normali viste di web2py come componenti, o possono esser inclusi in una pagina wiki utilizzando una sintassi semplificata.
- Un gruppo di pagine speciali (`meta-code`, `meta-menu`, ecc.) che possono essere utilizzate per personalizzare il plugin (per esempio per definire codice che il plugin dovrebbe eseguire, per personalizzare il menu, ecc.).

L'applicazione **welcome** *insieme al plugin* **plugin_wiki** *possono essere considerati come un ambiente di sviluppo utile per costruire semplici applicazioni web come, per esempio, un blog.*

Nel resto del capitolo si presuppone che il plugin **plugin_wiki** sia stato applicato ad una copia dell'applicazione di base **welcome**.

La prima cosa che si nota dopo aver installato il plugin è che è disponibile un nuovo menu chiamato *pages*.

[ADD IMAGE]

Cliccando sul menu *pages* si accede alle azioni del plugin:

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

La pagina *index* del plugin elenca le pagine create con il plugin stesso e consente di creare nuove pagine scegliento uno *slug*. Provare a creare una pagina `home`. Si verrà rediretti a:

```
http://127.0.0.1:8000/myapp/plugin_wiki/page/home
```

Cliccare su *create page* per modificare il contenuto della pagina.

[ADD IMAGE]

> *Per default il plugin è a livello 3, che significa che è possibile inserire widget e codice nelle pagine. Per default è utilizzata la sintassi markmin per descrivere il contenuto della pagina.*

13.3.1 Sintassi di Markmin

Ecco un tutorial per la sintassi markmin:

markmin	**html**
`# titolo`	`<h1>titolo</h1>`
`## sottotitolo`	`<h2>sottotitolo</h2>`
`### sotto-sottotitolo`	`<h3>sotto-sottotitolo</h3>`
`**grassetto**`	`<b>grassetto</b>`
`”corsivo”`	`<i>corsivo</i>`
`http://...com`	`<a href="http://...com">http:...com</a>`
`[[name http://example.com]]`	`<a href="http://example.com">name</a>`
`[[name http://...png left 200px]]`	`<img src="http://...png" alt="name" align="left" width="200px" />`

E’ possibile aggiungere collegamenti ad altre pagine

```
[[mylink name page:slug]]
```

Se la pagina non esiste verrà chiesto se la si vuole creare:

[ADD IMAGE]

La pagina di edit consente di aggiungere allegati alla pagina (per esempio file statici)

[ADD IMAGE] che possono essere collegati con:

```
[[mylink name attachment:3.png]]
```

o inseriti direttamente nella pagina con:

```
[[myimage attachment:3.png center 200px]]
```

La dimensione (`200px`) è opzionale. `center` non è opzionale ma può essere sostituito con `left` or `right`.

SI può inserire testo quotato con:

```
-----
this is blockquoted
-----
```

è anche possibile inserire tabelle con:

```
-----
0 | 0 | X
0 | X | 0
X | 0 | 0
-----
```

e testo letterale con:

```
``
verbatim text
``
```

E' anche possibile aggiungere `:class` all'ultima stringa `---` o `"`. Per il testo quotato e per le tabelle `:class` verrà trasformato nella classe del tag, per esempio:

```
-----
test
-----:abc
```

viene visualizzato come:

```
<blockquote class="abc">test</blockquote>
```

Per il testo letterale la classe può essere utilizzata per includere contenuto di tipo differente.

E' possibile, per esempio, includere codice con la sintassi evidenziata specificando il linguaggio con `:code_`*language*

```
``
def index(): return 'hello world'
``:code_python
```

E' anche possibile incorporare dei widget:

```
``
name: widget_name
attribute1: value1
attribute2: value2
``:widget
```

(vedere la prossima sezione per la lista dei widget)

E' anche possibile incorporare codice nel linguaggio di template di web2py:

```
``
{{for i in range(10):}}<h1>{{=i}}</h1>{{pass}}
``:template
```

13.3.2 Permessi di pagina

Quando si modifica una pagina sono presenti i seguenti campi:

- **active** (default a `True`). Se una pagina non è attiva non sarà accessibile agli utenti (anche se è una pagina pubblica).
- **public** (default a `True`). Se una pagina è definita come pubblica potrà essere acceduta dagli utenti senza la necessità di autenticarsi.
- **Role** (default a `None`). Se una pagina ha un ruolo potrà essere acceduta solamente dagli utenti che si sono autenticati e che sono membri del gruppo con il ruolo corrispondente.

13.3.3 Pagine speciali

meta-menu contiene il menu. Se questa pagina non esiste web2py utilizza l'oggetto `response.menu` definito in "models/menu.py". Il contenuto della pagina **meta-menu** sovrascrive l'oggetto menu. La sintassi è la seguente:

```
Item 1 Name http://link1.com
   Submenu Item 11 Name http://link11.com
   Submenu Item 12 Name http://link12.com
   Submenu Item 13 Name http://link13.com
Item 2 Name http://link1.com
   Submenu Item 21 Name http://link21.com
      Submenu Item 211 Name http://link211.com
      Submenu Item 212 Name http://link212.com
   Submenu Item 22 Name http://link22.com
   Submenu Item 23 Name http://link23.com
```

dove l'indentazione determina la struttura del sotto-menu. Ogni riga è composta del testo del menu seguito da un link. Un link può essere `page:`*slug*. Un link può essere `None` se quel menu non punta a nessuna pagina. Gli spazi in più sono ignorati.

Ecco un altro esempio:

```
Home             page:home
Search Engines   None
   Yahoo         http://yahoo.com
   Google        http://google.com
   Bing          http://bing.com
Help             page:help
```

Che viene visualizzato come:

[ADD IMAGE]

`meta-code`inxx **meta-code** è un'altra pagina speciale e deve contenere codice web2py. Questa pagina può essere considerata come un'estensione del modello in quanto in essa è possibile aggiungere codice di modello. E' eseguita insieme a "models/plugin_wiki.py". In **meta-code** è possibile definire delle

tabelle. Si può, per esempio, creare una semplice tabella `friends` inserendo il seguente codice in **meta-code**:

```
db.define_table('friend',Field('name',requires=IS_NOT_EMPTY()))
```

e si può creare una pagina di gestione della tabella *friend* inserendo in una qualsiasi pagina il seguente codice:

```
# List of friends
``
name: jqgrid
table: friend
``:widget

# New friend
``
name: create
table: friend
``:widget
```

Questa pagina ha due intestazioni (che iniziano con #): "List of friends" e "New friend". Contiene inoltre due widget (nelle rispettive intestazioni): un widget `jqgrid` che elenca tutti i record e un widget `CRUD` per aggiungere un nuovo record.

[ADD IMAGE]

`meta-header`, `meta-footer`, `meta-sidebar` non sono utilizzate nel layout di default "welcome/views/layout.html". Se si vogliono utilizzare si deve modificare la pagina "layout.html" tramite **admin** (o da linea di comando) ed inserire i seguenti tag nella posizione appropriata:

```
{{=plugin_wiki.embed_page('meta-header') or ''}}
{{=plugin_wiki.embed_page('meta-sidebar') or ''}}
{{=plugin_wiki.embed_page('meta-footer') or ''}}
```

In questo modo il contenuto di quelle pagine sarà mostrato nell'intestazione, nella barra laterale e nel piè di pagina del layout.

13.3.4 *Configurare plugin_wiki*

Come con qualsiasi altro plugin è possibile inserire il codice seguente in "models/db.py":

```
from gluon.tools import PluginManager
plugins = PluginManager
plugins.wiki.editor = auth.user.email==mail.settings.sender
plugins.wiki.level = 3
plugins.wiki.mode = 'markmin' or 'html'
plugins.wiki.theme = 'ui-darkness'
```

dove:

- **editor** è True se l'utente collegato è autorizzato a modificare le pagine del plugin
- **level** è il permesso: 1 - per modificare pagine standard, 2 - per includere widget, 3 - per inserire codice
- **mode** determina se utilizzare l'editor "markmin" oppure l'editory "html" WYSIWYG.
- **theme** è il nome del tema UI di jQuery richiesto. Per default è installato solo il tema senza colori "ui-darkness".

E' possibile aggiungere temi in:

```
static/plugin_wiki/ui/%(theme)s/jquery-ui-1.8.1.custom.css
```

13.3.5 *Widget*

Ciascun widget può essere incorporato sia nelle pagine di plugin che nei normali template di web2py.

Per esempio il seguente codice server ad incorporare un video di YouTube in una pagina di plugin:

```
``
name: youtube
code: l7AWnfFRc7g
``:widget
```

o per incorporare lo stesso widget in una vista web2py si può utilizzare il seguente codice:

```
{{=plugin_wiki.widget('youtube',code='l7AWnfFRc7g')}}
```

Gli argomenti del widget che non hanno un valore di default sono obbligatori.

Ecco la lista dei widget attualmente disponibili:

```
read(table,record_id=None)
```

Legge e visualizza un record

- `table` è il nome della tabella
- `record_id` è il numero del record

```
create(table,message='',next='',readonly_fields='',
       hidden_fields='',default_fields='')
```

Visualizza un form di creazione di un record

- `table` è il nome della tabella
- `message` è il messaggio da visualizzare dopo la creazione del record
- `next` è dove reindirizzare dopo la creazione, per esempio a "page/index/[id]"
- `readonly_fields` è una lista di campi in sola lettura, separati dalla virgola
- `hidden_fields` è una lista di campi nascosti, separati dalla virgola
- `default_fields` è una lista dei campi con i valori di default ("fieldname=value")

```
update(table,record_id='',message='',next='',
       readonly_fields='',hidden_fields='',default_fields='')
```

Visualizza un form di aggiornamento di un record

- `table` è il nome della tabella
- `record_id` è il record da aggiornare oppure {{=request.args(-1)}}
- `message` è il messaggio da visualizzare dopo l'aggiornamento del record
- `next` è dove reindirizzare dopo l'aggiornamento, per esempio a "page/index/[id]"
- `readonly_fields` è una lista di campi in sola lettura, separati dalla virgola
- `hidden_fields` è una lista di campi nascosti, separati dalla virgola
- `default_fields` è una lista dei campi con i valori di default ("fieldname=value")

```
select(table,query_field='',query_value='',fields='')
```

Elenca i record nella tabella

- `table` è il nome della tabella
- `query_field` e `query_value` se presenti filtreranno la tabella con query_field==query_va
- `fields` è una lista dei campi da visualizzare, separati dalla virgola

```
search(table,fields='')
```

E' un widget per selezionare i record

- `table` è il nome della tabella
- `fields` è una lista dei campi da visualizzare, separati dalla virgola

```
jqgrid(table,fieldname=None,fieldvalue=None,col_widths='',
       _id=None,fields='',col_width=80,width=700,height=300)
```

Incorpora un plugin di tipo jqGrid

- `table` è il nome della tabella

- `fieldname`, `fieldvalue` sono campi di filtro opzionali (dove fieldname==fieldvalue)
- `_id` è l'id del DIV che contiene l'oggetto jqGrid
- `columns` è la lista dei nomi delle colonne da visualizzare
- `col_width` è la larghezza di ciascuna colonna (default)
- `height` è l'altezza dell'oggetto jqGrid
- `width` è la larghezza dell'oggetto jqGrid

```
latex(expression)
```

Usa le API dei grafici di Google per incorporare un oggetto LaTeX

```
pie_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

Incorpora un grafico a torta

- `data` è una lista di valori, separati da virgola
- `names` è una lista di etichette, separate da virgola (una per ogni elemento)
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine
- `align` determina l'allineamento dell'immagine

```
bar_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

Usa le API dei grafici di Google per incorporare un grafico a barre

- `data` è una lista di valori, separati da virgola
- `names` è una lista di etichette, separate da virgola (una per ogni elemento)
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine
- `align` determina l'allineamento dell'immagine

```
slideshow(table,field='image',transition='fade',width=200,height=200)
```

Incorpora uno *slideshow*. Le immagini sono recuperate da una tabella Embeds a slideshow. It gets the images from a table.

- `table` è il nome della tabella
- `field` è il campo di upload della tabella che contiene le immagini
- `transition` determina il tipo di transizione
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine

```
youtube(code,width=400,height=250)
```

Incorpora un video di YouTube (tramite codice)

- `code` è il codice del video
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine

```
vimeo(code,width=400,height=250)
```

Incorpora un video di Vimeo (tramite codice)

- `code` è il codice del video
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine

```
mediaplayer(src,width=400,height=250)
```

Incorpora un file multimediale (come un video Flash o un file audio mp3)

- `src` è il collegamento al video
- `width` è la larghezza dell'immagine
- `height` è l'altezza dell'immagine

```
comments(table='None',record_id=None)
```

Incorpora i commenti in una pagina. I commenti possono essere collegati ad una tabella e/o ad un record

- `table` è il nome della tabella
- `record_id` è l'id del record

```
tags(table='None',record_id=None)
```

Incorpora dei tag in una pagina. I tag possono essere collegati ad una tabella e/o ad un record

- `table` è il nome della tabella
- `record_id` è l'id del record

```
tag_cloud()
```

Inserisce una nuvola di tag (*tag cloud*)

```
map(key='....', table='auth_user', width=400, height=200)
```

Incorpora una mappa di Google Prende i punti di una mappa da una tabella

- `key` è la chiave delle API di Google Map (default 127.0.0.1)
- `table` è il nome della tabella
- `width` è la larghezza della mappa
- `height` è l'altezza della mappa

```
La tabella deve avere le seguenti colonne: latidude, longitude e map_popup.
Quando si seleziona un punto appare il messaggio memorizzato in map_popup.
```

```
iframe(src, width=400, height=300)
```

Incorpora una pagina nei tag `<iframe> ... </iframe>`

```
load_url(src)
```

Carica il contenuto della URL utilizzando la funzione LOAD

```
load_action(action, controller='', ajax=True)
```

Carica il contenuto della URL(request.application, controller, action) utilizzando la funzione LOAD

13.3.6 Estendere i widget

Per aggiungere un widget a **plugin_wiki** si deve creare un nuovo modello chiamato "models/plugin_wiki_"*name* dove *name* è un nome arbitrario. Il file deve contenere:

```
class PluginWikiWidgets(PluginWikiWidgets):
    @staticmethod
    def my_new_widget(arg1, arg2='value', arg3='value'):
        """
        document the widget
    """
        return "body of the widget"
```

La prima linea dichiare che si sta estendendo la lista dei widget. ALl'interno della classe si possono definire quante funzioni si vuole. Ogni funzione statica che non inizia con il carattere di underscore (_) è un nuovo widget. La funzione può avere un numero qualsiasi di argomenti che possono avere un valore di default. La *docstring* della funzione deve documentare l'utilizzo della funzione con la sintassi markmin.

Quando i widget sono incorporati nelle pagine di plugin gli argomenti saranno passati al widget come stringhe. Per questo il widget deve essere in grado di ricevere stringhe per ogni argomento ed eventualmente convertirlo nella rappresentazione appropriata che deve essere adeguatamente documentata nella *docstring*.

Il widget può restituire una stringa o helper di web2py che verranno serializzati con `.xml()`.

Il nuovo widget può accedere a qualsiasi variabile dichiarata nel namespace globale.

Ecco, come esempio, un widget che visualizza il form "contact/ask" creato all'inizio di questo capitolo. Il widget è contenuto in "models/plugin_wiki_contact":

```
class PluginWikiWidgets(PluginWikiWidgets):
    @staticmethod
    def ask(email_label='Your email', question_label='question'):
        """
    This plugin will display a contact us form that allows
    the visitor to ask a question.
    The question will be emailed to you and the widget will
    disappear from the page.
    The arguments are

    - email_label: the label of the visitor email field
    - question_label: the label of the question field

    """
        form=SQLFORM.factory(
           Field('your_email',requires=IS_EMAIL(),label=email_label),
           Field('question',requires=IS_NOT_EMPTY()),label=question_label)
        if form.accepts(request.vars,session):
           if mail.send(to='admin@example.com',
                        subject='from %s' % form.vars.your_email,
                        message = form.vars.question):
               div_id = request.env.http_web2py_component_element
               command="jQuery('#%s').hide()" % div_id
           response.headers['web2py-component-command']=command
               response.headers['web2py-component-flash']='thanks you'
        else:
            form.errors.your_email="Unable to send the email"
        return form.xml()
```

I widget di plugin non sono visualizzati dalla vista a meno che la funzione `response.render(...)` *sia esplicitamente chiamata dal widget stesso.*

Index

Bibliography

[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://www.ingres.com/

[13] http://code.google.com/appengine/

[14] http://en.wikipedia.org/wiki/HTML

[15] http://www.w3.org/TR/REC-html40/

[16] http://www.php.net/

[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://www.pythonsecurity.org

[21] http://en.wikipedia.org/wiki/Secure_Sockets_Layer

[22] https://launchpad.net/rocket

[23] http://www.cdolivet.net/editarea/

[24] http://nicedit.com/

[25] http://pypi.python.org/pypi/simplejson

[26] http://pyrtf.sourceforge.net/

[27] http://www.dalkescientific.com/Python/PyRSS2Gen.html

[28] http://www.feedparser.org/

[29] http://code.google.com/p/python-markdown2/

[30] http://www.tummy.com/Community/software/python-memcached/

[31] http://www.fsf.org/licensing/licenses/info/GPLv2.html

[32] http://jquery.com/

[33] https://www.web2py.com/cas

[34] http://www.web2py.com/appliances

[35] http://www.web2py.com/AlterEgo

[36] http://www.python.org/dev/peps/pep-0008/

[37] http://www.network-theory.co.uk/docs/pytut/

[38] http://oreilly.com/catalog/9780596158071

[39] http://www.python.org/doc/

[40] http://en.wikipedia.org/wiki/Cascading_Style_Sheets

[41] http://www.w3.org/Style/CSS/

[42] http://www.w3schools.com/css/

[43] http://en.wikipedia.org/wiki/JavaScript

[44] http://www.amazon.com/dp/0596000480

[45] http://en.wikipedia.org/wiki/Cron#crontab_syntax

[46] http://www.xmlrpc.com/

[47] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[48] http://www.w3.org/Protocols/rfc2616/rfc2616.html

[49] http://en.wikipedia.org/wiki/XML

[50] http://www.w3.org/XML/

[51] http://www.ez-css.org

[52] http://en.wikipedia.org/wiki/XHTML

[53] http://www.w3.org/TR/xhtml1/

[54] http://www.w3schools.com/xhtml/

[55] http://www.web2py.com/layouts

[56] http://sourceforge.net/projects/zxjdbc/

[57] http://pypi.python.org/pypi/psycopg2

[58] http://sourceforge.net/projects/mysql-python

[59] http://python.net/crew/atuining/cx_Oracle/

[60] http://pyodbc.sourceforge.net/

[61] http://kinterbasdb.sourceforge.net/

[62] http://informixdb.sourceforge.net/

[63] http://pypi.python.org/simple/ingresdbi/

[64] http://docs.python.org/library/csv.html#csv.QUOTE_ALL

[65] http://www.faqs.org/rfcs/rfc2616.html

[66] http://www.faqs.org/rfcs/rfc2396.html

[67] http://tools.ietf.org/html/rfc3490

[68] http://tools.ietf.org/html/rfc3492

[69] ttp://mail.python.org/pipermail/python-list/2007-June/617126.html

[70] ttp://mail.python.org/pipermail/python-list/2007-June/617126.html

[71] http://www.recaptcha.net

[72] http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules

[73] http://www.reportlab.org

[74] http://gdwarner.blogspot.com/2008/10/brief-pyjamas-django-tutorial.html

[75] http://en.wikipedia.org/wiki/AJAX

[76] http://www.learningjquery.com/

[77] http://ui.jquery.com/

[78] http://en.wikipedia.org/wiki/Common_Gateway_Interface

[79] http://www.apache.org/

[80] http://httpd.apache.org/download.cgi

[81] http://adal.chiriliuc.com/mod_wsgi/revision_1018_2.3/mod_wsgi_py25_apache22/mod_wsgi.so

[82] http://httpd.apache.org/docs/2.0/mod/mod_proxy.html

[83] http://httpd.apache.org/docs/2.2/mod/core.html

[84] http://sial.org/howto/openssl/self-signed

[85] http://code.google.com/p/modwsgi/

[86] http://www.lighttpd.net/

[87] http://www.cherokee-project.com/download/

[88] http://www.fastcgi.com/

[89] http://www.apsis.ch/pound/

[90] http://haproxy.1wt.eu/

[91] http://pyamf.org/

[92] http://en.wikipedia.org/wiki/List_of_HTTP_status_codes

[93] http://www.authorize.net/

[94] http://sourceforge.net/projects/zxjdbc/

[95] http://www.zentus.com/sqlitejdbc/