

著者 : MASSIMO DI PIERRO

翻訳監修 : OMI CHIBA

翻訳: FUMITO MIZUNO, HITOSHI KATO, KENJI HOSODA,  
KENJI NAKAGAKI, MITSUHIRO TSUDA, YOTA ICHINO

# WEB2PY

FULL-STACK WEB FRAMEWORK, 4TH EDITION

EXPERTS4SOLUTIONS

Copyright 2008-2012 by Massimo Di Pierro2. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the web at [www.copyright.com](http://www.copyright.com). Requests to the Copyright owner for permission should be addressed to:

Massimo Di Pierro  
School of Computing  
DePaul University  
243 S Wabash Ave  
Chicago, IL 60604 (USA)  
Email: massimo.dipierro@gmail.com

**Limit of Liability/Disclaimer of Warranty:** While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

Library of Congress Cataloging-in-Publication Data:

ISBN: 978-0-578-09793-0

Build Date: July 17, 2012

*to my family*



# Contents

<b>1</b>	<b>はじめに</b>	<b>23</b>
1.1	原則	25
1.2	Web フレームワーク	26
1.3	モデル、ビュー、コントローラ	28
1.4	なぜ web2py か	32
1.5	セキュリティ	34
1.6	内部構造	37
1.7	ライセンス	38
1.8	謝辞	40
1.9	本書について	41
1.10	About this book	41
1.11	要素のスタイル	43
<b>2</b>	<b>Python 言語</b>	<b>47</b>
2.1	Python について	47
2.2	Starting up	48
2.3	help, dir	49
2.4	型	50
2.4.1	str	50
2.4.2	list	52
2.4.3	tuple	53
2.4.4	dict	54
2.5	インデントについて	55
2.6	for...in	56
2.7	while	57

2.8 if...elif...else . . . . .	57
2.9 try...except...else...finally . . . . .	58
2.10 def...return . . . . .	60
2.10.1 lambda . . . . .	63
2.11 class . . . . .	64
2.12 特殊属性、メソッド、演算子 . . . . .	66
2.13 ファイル入力/出力 . . . . .	67
2.14 exec, eval . . . . .	68
2.15 import . . . . .	69
2.15.1 os . . . . .	70
2.15.2 sys . . . . .	70
2.15.3 datetime . . . . .	71
2.15.4 time . . . . .	72
2.15.5 cPickle . . . . .	72
<b>3 概要</b>	<b>73</b>
3.1 はじめよう . . . . .	73
3.2 挨拶しよう . . . . .	78
3.3 数えよう . . . . .	82
3.4 名前を名乗ろう . . . . .	84
3.5 ポストバック (Postbacks) . . . . .	85
3.6 画像ブログ . . . . .	88
3.7 CRUD を追加しよう . . . . .	102
3.8 認証を追加しよう . . . . .	103
3.8.1 グリッドの追加 . . . . .	106
3.9 レイアウトの設定 . . . . .	108
3.10 Wiki . . . . .	108
3.10.1 date, datetime そして time の書式 . . . . .	120
3.11 admin の追加情報 . . . . .	121
3.11.1 サイト . . . . .	121
3.11.2 about . . . . .	124
3.11.3 edit . . . . .	124
3.11.4 errors . . . . .	127
3.11.5 Mercurial . . . . .	131
3.11.6 管理ウィザード (実験的) . . . . .	132

3.11.7 admin の設定 . . . . .	134
3.12 appadmin の追加情報 . . . . .	135
<b>4 コア . . . . .</b>	<b>139</b>
4.1 コマンドライン オプション . . . . .	139
4.2 ワークフロー . . . . .	143
4.3 ディスパッチ . . . . .	146
4.4 ライブリ . . . . .	150
4.5 アプリケーション . . . . .	157
4.6 API . . . . .	158
4.6.1 Python モジュールからの API アクセス . . . . .	160
4.7 request . . . . .	162
4.8 response . . . . .	167
4.9 session . . . . .	171
4.9.1 セッションの分割 . . . . .	173
4.10 cache . . . . .	173
4.11 URL . . . . .	176
4.11.1 絶対 URL . . . . .	179
4.11.2 デジタル署名つき URL . . . . .	179
4.12 HTTP と redirect . . . . .	180
4.13 T と国際化 . . . . .	182
4.14 クッキー . . . . .	185
4.15 init アプリケーション . . . . .	186
4.16 URL リライト . . . . .	187
4.16.1 パラメタベースのシステム . . . . .	187
4.16.2 パターンベースのシステム . . . . .	190
4.17 エラーのルーティング . . . . .	194
4.18 バックグランドでのタスク実行 . . . . .	195
4.18.1 クーロン . . . . .	196
4.18.2 ホームメード・タスクキュー . . . . .	199
4.18.3 スケジューラー (実験的) . . . . .	200
4.19 サードパーティのモジュール . . . . .	204
4.20 実行環境 . . . . .	205
4.21 協調 . . . . .	207
4.22 ロギング . . . . .	208

4.23 WSGI . . . . .	209
4.23.1 外部ミドルウェア . . . . .	210
4.23.2 内部ミドルウェア . . . . .	210
4.23.3 WSGI アプリケーションの呼び出し . . . . .	211
<b>5 ビュー . . . . .</b>	<b>213</b>
5.1 基本構文 . . . . .	215
5.1.1 for...in . . . . .	216
5.1.2 while . . . . .	216
5.1.3 if...elif...else . . . . .	216
5.1.4 try...except...else...finally . . . . .	217
5.1.5 def...return . . . . .	218
5.2 HTML ヘルパー . . . . .	219
5.2.1 XML . . . . .	221
5.2.2 組み込みヘルパー . . . . .	222
5.2.3 カスタム・ヘルパー . . . . .	238
5.3 BEAUTIFY . . . . .	239
5.4 サーバーサイドの DOM と構文解析 . . . . .	240
5.4.1 elements . . . . .	240
5.4.2 components . . . . .	242
5.4.3 parent . . . . .	242
5.4.4 flatten . . . . .	242
5.4.5 構文解析 . . . . .	243
5.5 ページレイアウト . . . . .	243
5.5.1 デフォルトのページのレイアウト . . . . .	246
5.5.2 デフォルトレイアウトのカスタマイズ . . . . .	250
5.5.3 モバイル開発 . . . . .	251
5.6 ビュー内の関数 . . . . .	252
5.7 ビュー内のブロック . . . . .	253
<b>6 データベース抽象化レイヤ . . . . .</b>	<b>255</b>
6.1 依存関係 . . . . .	255
6.2 接続文字列 . . . . .	257
6.2.1 接続プール . . . . .	258
6.2.2 接続の失敗 . . . . .	258

6.2.3 複製されたデータベース . . . . .	259
6.3 予約キーワード . . . . .	259
6.4 DAL, Table, Field . . . . .	260
6.5 レコードの表現 . . . . .	261
6.6 マイグレーション . . . . .	267
6.7 壊れたマイグレーションの修復 . . . . .	269
6.8 挿入 . . . . .	270
6.9 コミットとロールバック . . . . .	271
6.10 生の SQL . . . . .	272
6.10.1 クエリ実行時間の計測 . . . . .	272
6.10.2 executesql . . . . .	272
6.10.3 _lastsql . . . . .	272
6.11 drop . . . . .	273
6.12 インデックス . . . . .	273
6.13 レガシー・データベースとキー付きテーブル . . . . .	273
6.14 分散トランザクション . . . . .	275
6.15 手動アップロード . . . . .	275
6.16 Query, Set, Rows . . . . .	276
6.17 select . . . . .	277
6.17.1 ショートカット . . . . .	279
6.17.2 Fetching a Row . . . . .	280
6.17.3 再帰的な selects . . . . .	280
6.17.4 ビューにおける Rows のシリアル化 . . . . .	281
6.17.5 orderby, groupby, limitby, distinct . . . . .	283
6.17.6 論理演算子 . . . . .	285
6.17.7 count, isempty, delete, update . . . . .	286
6.17.8 式 . . . . .	286
6.17.9 update_record . . . . .	287
6.17.10 first と last . . . . .	287
6.17.11 as_dict と as_list . . . . .	288
6.17.12 find, exclude, sort . . . . .	288
6.18 その他のメソッド . . . . .	289
6.18.1 update_or_insert . . . . .	289
6.18.2 validate_and_insert, validate_and_update . .	290
6.18.3 smart_query (実験的) . . . . .	290

6.19 計算されたフィールド . . . . .	291
6.20 仮想フィールド . . . . .	291
6.20.1 古い形式の仮想フィールド . . . . .	292
6.20.2 新しい形式の仮想フィールド (実験的) . . . . .	294
6.21 1対多のリレーション . . . . .	295
6.21.1 内部結合 (Inner Joins) . . . . .	296
6.21.2 左外部結合 (Left Outer Join) . . . . .	298
6.21.3 グループ化とカウント . . . . .	298
6.22 Many to many . . . . .	299
6.23 多対多、list:<type>、contains . . . . .	300
6.24 その他の演算子 . . . . .	302
6.24.1 like, startswith, contains, upper, lower . . . . .	303
6.24.2 year, month, day, hour, minutes, seconds . . . . .	304
6.24.3 belongs . . . . .	304
6.24.4 sum, min, max and len . . . . .	304
6.24.5 サブストリング . . . . .	305
6.24.6 coalesce と coalesce_zero によるデフォルト値 . . . . .	305
6.25 生 SQL の生成 . . . . .	306
6.26 データのエクスポートとインポート . . . . .	307
6.26.1 CSV(一度に1つのテーブル) . . . . .	307
6.26.2 CSV(全てのテーブルを一度に) . . . . .	307
6.26.3 CSV とリモート・データベースの同期 . . . . .	308
6.26.4 HTML/XML の (一度に一つのテーブル) . . . . .	311
6.26.5 データ表現 . . . . .	312
6.27 選択のキャッシュ . . . . .	313
6.28 自己参照と別名 . . . . .	313
6.29 高度な機能 . . . . .	315
6.29.1 テーブル継承 . . . . .	315
6.29.2 コモンフィールドとマルチテナント . . . . .	316
6.29.3 コモンフィルタ . . . . .	317
6.29.4 カスタム Field 型 (実験的) . . . . .	317
6.29.5 テーブル定義なしで DAL を使用 . . . . .	318
6.29.6 異なる db からデータをコピー . . . . .	319
6.29.7 新しい DAL とアダプタの注意点 . . . . .	319
6.30 翻訳 . . . . .	324

<b>7 フォームとバリデータ</b>	<b>325</b>
7.1 FORM . . . . .	326
7.1.1 process と validate メソッド . . . . .	330
7.1.2 隠しフィールド . . . . .	331
7.1.3 keepvalues . . . . .	332
7.1.4 onvalidation . . . . .	333
7.1.5 レコード変更の検知 . . . . .	334
7.1.6 フォームとリダイレクト . . . . .	334
7.1.7 ページ毎に複数のフォーム . . . . .	335
7.1.8 フォームの共有 . . . . .	336
7.2 SQLFORM . . . . .	337
7.2.1 SQLFORM と insert/update/delete . . . . .	342
7.2.2 HTML における SQLFORM . . . . .	343
7.2.3 SQLFORM とアップロード . . . . .	345
7.2.4 元のファイル名の保存 . . . . .	348
7.2.5 autodelete . . . . .	349
7.2.6 参照レコードへのリンク . . . . .	349
7.2.7 フォームの事前入力 . . . . .	351
7.2.8 SQLFORM に要素の追加 . . . . .	352
7.2.9 データベース IO なしの SQLFORM . . . . .	352
7.3 SQLFORM.factory . . . . .	353
7.3.1 複数テーブルでひとつのフォーム . . . . .	354
7.4 CRUD . . . . .	355
7.4.1 設定 . . . . .	356
7.4.2 メッセージ . . . . .	359
7.4.3 メソッド . . . . .	360
7.4.4 レコードのバージョニング . . . . .	362
7.5 カスタムフォーム . . . . .	363
7.5.1 CSS の慣例 . . . . .	365
7.5.2 エラーの非表示 . . . . .	365
7.6 バリデータ . . . . .	366
7.6.1 バリデータ . . . . .	368
7.6.2 データベースのバリデータ . . . . .	379
7.6.3 カスタムバリデータ . . . . .	382
7.6.4 依存関係のバリデータ . . . . .	383

7.7 Widgets . . . . .	384
7.7.1 Autocomplete widget . . . . .	386
7.8 SQLFORM.grid と SQLFORM.smartgrid (実験的) . . . . .	387
<b>8 Email と SMS</b>	<b>395</b>
8.1 メールの設定 . . . . .	395
8.1.1 Google App Engine でのメール設定 . . . . .	396
8.1.2 x509 と PGP 暗号化 . . . . .	396
8.2 メール送信 . . . . .	396
8.2.1 テキストメール . . . . .	397
8.2.2 HTML メール . . . . .	397
8.2.3 テキストと HTML の混在メール . . . . .	397
8.2.4 cc と bcc を使用したメール . . . . .	398
8.2.5 添付ファイル . . . . .	398
8.2.6 複数の添付ファイル . . . . .	398
8.3 SMS メッセージの送信 . . . . .	398
8.4 メッセージ作成で使用するテンプレートシステム . . . . .	399
8.5 バックグラウンドタスクを使用したメッセージ送信 . . . . .	400
<b>9 アクセス制御</b>	<b>403</b>
9.1 認証 . . . . .	405
9.1.1 登録の制限 . . . . .	409
9.1.2 OpenID, Facebook などとの統合 . . . . .	410
9.1.3 CAPTCHA と reCAPTCHA . . . . .	412
9.1.4 Auth のカスタマイズ . . . . .	413
9.1.5 Auth テーブルの名前変更 . . . . .	415
9.1.6 その他のログイン方式とログインフォーム . . . . .	415
9.2 Mail と Auth . . . . .	422
9.3 認可 . . . . .	423
9.3.1 デコレータ . . . . .	426
9.3.2 権限要求の組み合わせ . . . . .	427
9.3.3 権限と CRUD . . . . .	427
9.3.4 認可とダウンロード . . . . .	429
9.3.5 アクセス制御とベーシック認証 . . . . .	429
9.3.6 手動認証 . . . . .	430

9.3.7 設定とメッセージ . . . . .	430
9.4 Central Authentication Service . . . . .	436
9.4.1 web2py を使用した、非 web2py アプリケーションの認可	438
<b>10 サービス</b>	<b>441</b>
10.1 辞書のレンダリング . . . . .	441
10.1.1 HTML, XML, そして JSON . . . . .	442
10.1.2 汎用 (generic) ビュー . . . . .	443
10.1.3 Rows のレンダリング . . . . .	445
10.1.4 カスタムフォーマット . . . . .	445
10.1.5 RSS . . . . .	446
10.1.6 CSV . . . . .	448
10.2 リモートプロシージャコール . . . . .	449
10.2.1 XMLRPC . . . . .	451
10.2.2 JSONRPC . . . . .	452
10.2.3 JSONRPC と Pyjamas . . . . .	453
10.2.4 Amfrpc . . . . .	457
10.2.5 SOAP . . . . .	459
10.3 低レベル API とその他のレシピ . . . . .	460
10.3.1 simplejson . . . . .	460
10.3.2 PyRTF . . . . .	461
10.3.3 ReportLab と PDF . . . . .	462
10.4 Restful Web サービス . . . . .	463
10.4.1 parse_as_rest (実験的試み) . . . . .	466
10.4.2 smart_query (実験的試み) . . . . .	471
10.4.3 アクセス制御 . . . . .	472
10.5 サービスとアクセス制御 . . . . .	472
<b>11 jQuery と Ajax</b>	<b>475</b>
11.1 web2py_ajax.html . . . . .	475
11.2 jQuery エフェクト . . . . .	479
11.2.1 フォームの条件付フィールド . . . . .	483
11.2.2 削除の確認 . . . . .	485
11.3 ajax 関数 . . . . .	486
11.3.1 Eval ターゲット . . . . .	487

11.3.2 オートコンプリーション . . . . .	488
11.3.3 Ajax フォーム送信 . . . . .	490
11.3.4 投票と評価 . . . . .	492
<b>12 コンポーネントとプラグイン . . . . .</b>	<b>495</b>
12.1 コンポーネント . . . . .	495
12.1.1 クライアント・サーバー コンポーネント通信 . . . . .	501
12.1.2 Ajax トランクル . . . . .	503
12.2 プラグイン . . . . .	503
12.2.1 コンポーネントプラグイン . . . . .	507
12.2.2 プラグインマネージャー . . . . .	509
12.2.3 レイアウトプラグイン . . . . .	510
12.3 plugin_wiki . . . . .	511
12.3.1 MARKMIN 構文 . . . . .	514
12.3.2 ページの権限 . . . . .	516
12.3.3 スペシャルページ . . . . .	517
12.3.4 plugin_wiki の設定 . . . . .	519
12.3.5 現在の widgets . . . . .	520
12.3.6 ウィジットの拡張 . . . . .	526
12.3.7 Extending widgets . . . . .	526
<b>13 デプロイレシピ . . . . .</b>	<b>529</b>
13.0.8 anyserver.py . . . . .	532
13.1 Linux と Unix . . . . .	533
13.1.1 本番デプロイへの第一歩 . . . . .	533
13.1.2 Apache セットアップ . . . . .	533
13.1.3 mod_wsgi . . . . .	534
13.1.4 mod_wsgi と SSL . . . . .	538
13.1.5 mod_proxy . . . . .	539
13.1.6 Linux デーモンとして起動 . . . . .	541
13.1.7 Lighttpd . . . . .	542
13.1.8 mod_python を使った共有ホスティング . . . . .	543
13.1.9 Cherokee と FastCGI . . . . .	544
13.1.10 Postgresql . . . . .	546
13.2 Windows . . . . .	547

13.2.1 Apache と mod_wsgi . . . . .	547
13.2.2 Windows サービスとして起動 . . . . .	550
13.3 セッション保護と admin . . . . .	551
13.4 効率とスケーラビリティ . . . . .	552
13.4.1 効率のトリック . . . . .	553
13.4.2 データベースでのセッション . . . . .	554
13.4.3 HAProxy 高可用性ロードバランサ . . . . .	555
13.4.4 セッションのクリーンアップ . . . . .	556
13.4.5 データベース上でのファイルアップロード . . . . .	557
13.4.6 チケットの収集 . . . . .	558
13.4.7 Memcache . . . . .	559
13.4.8 Memcache でのセッション . . . . .	559
13.4.9 Redis によるキャッシング . . . . .	560
13.4.10 アプリケーションの削除 . . . . .	560
13.4.11 レプリカデータベースの使用 . . . . .	560
13.5 Google App Engine でのデプロイ . . . . .	561
13.5.1 設定 . . . . .	563
13.5.2 実行とデプロイメント . . . . .	565
13.5.3 ハンドラの設定 . . . . .	566
13.5.4 ファイルシステムの無効 . . . . .	567
13.5.5 Memcache . . . . .	567
13.5.6 Datastore の問題 . . . . .	568
13.5.7 GAE と HTTPS . . . . .	569
13.6 Jython . . . . .	569
<b>14 その他のレシピ</b> . . . . .	<b>571</b>
14.1 アップグレード . . . . .	571
14.2 バイナリでアプリケーションを配布する方法 . . . . .	571
14.3 web2py の最小構成 . . . . .	572
14.4 外部 URL をフェッチ . . . . .	573
14.5 便利な日時表現 (Pretty dates) . . . . .	573
14.6 ジオコーディング . . . . .	574
14.7 ページネーション . . . . .	574
14.8 httpserver.log とログファイルフォーマット . . . . .	575
14.9 ダミーデータをデータベースに登録 . . . . .	577

14.10 クレジットカード払いの承認 . . . . .	577
14.10.1 Google Wallet . . . . .	578
14.10.2 Paypal . . . . .	580
14.10.3 Stripe.com . . . . .	580
14.10.4 Authorize.Net . . . . .	581
14.11 Dropbox API . . . . .	583
14.12 Twitter API . . . . .	584
14.13 仮想ファイルのストリーミング . . . . .	584
<b>Bibliography</b>	<b>587</b>





# *Preface*

`web2py` は 2007 年に立ち上げられ、4 年間の継続的な開発を経た現在、私たちは待望の第 4 版を完成しました。この間に `web2py` は、数千人の知識豊富なユーザと百人以上の開発者の愛情を得てきました。私たちの努力の結果、実存する中で一番多機能なオープンソースの web フレームワークの一つが完成しました。

当初、私は教材として `web2py` を作成しました。なぜなら、良質な web アプリケーションを構築することは、自由でオープンな社会の成長に非常に重要だと信じているからです。そして大手企業や団体による情報の独占を防止します。今もその思いに変わりはなく、むしろさらに重要性を増しています。

一般的にどの web フレームワークの目的も、簡単に、早く、特にセキュリティに関する開発者の間違いを防いで、web 開発を行うことです。`web2py` では、私たちは三つの主要な目標で、これらの問題に取り組んでいます。

使いやすさは、`web2py` の一番の目標です。私たちにとって、学習と開発に関わる時間を減らすという意味があります。またこれは `web2py` が、他に依存する物を持たないフルスタックフレームワークである理由です。インストールも設定ファイルも必要ありません。`web` サーバー、データベース、全ての機能にアクセスできる web ベースの IDE を含む全てが標準で動作します。API は 12 のコアオブジェクトに単純化されており、簡単に覚えて使うことができます。大部分の `web` サーバーやデータベース、そして全ての Python ライブラリと互換性を持ちます。

開発の高速化は二つ目の目標です。`web2py` のどの関数も(上書き可能な)デフォルトの振る舞いを持っています。例えば、データモデルを指定するとすぐに、web ベースのデータベース管理パネルにアクセスできます。また、`web2py` はデータ

のフォームを自動で作成し、HTML、XML、JSON、RSSなどのデータを簡単に公開できます。

セキュリティは web2py の心臓部で、システムとデータを安全に保つために全てを保護することが目標です。このためデータベースレイヤは、SQL インジェクションを取り除きます。テンプレート言語は、クロスサイトスクリプティングの脆弱性を妨ぎます。web2py によって生成されるフォームは、フィールドのバリデーションを提供し、クロスサイトリクエストフォージェリをブロックします。パスワードは常にハッシュ化して保存されます。セッションはクッキーの改ざんを防ぐために、デフォルトでサーバーサイドに保存され、セッションクッキーはクッキーの盗難を防ぐために uuid を使用します。

web2py は常にユーザ視点で開発され、何時でも後方互換を保ちながら、より速くよりスリムになるように常に内部最適化が続けられます。

web2py は無償で使用できます。もしあなたが web2py で利益を得ることができた場合には、どのような形でもいいので社会に恩返しをする心を持っていただけることを希望します。

2011 年の InforWorld magazine で、最も人気のあるフルスタックで Python ベースの web フレームワークの 6 つに選ばれ、その中でも最高ランクでした。また同じく 2011 年に、Bossie Award の最も優れたオープンソース開発ソフトを受賞しました。

第 3 版 - 翻訳: Omi Chiba レビュー: 中垣健志

第 4 版 - 翻訳: Omi Chiba レビュー: Hitoshi Kato

## 翻訳

この本は web2py Japan のメンバーによって翻訳されました。

名前のアルファベット順:

- Fumito MIZUNO (GitHub <https://github.com/ounziw> )
- Hitoshi Kato (blog : <http://todayspython.blogspot.com> )
- Kenji Hosoda (hosoda@s-cubism.jp)
- Kenji Nakagaki (blog 「IT Virtuoso」 )
- Mitsuhiro Tsuda (mtsuda@ipallet.org)
- Omi Chiba (blog 「Python Roll」 )
- Yota Ichino (GitHub <https://github.com/nus> )



# 1

## はじめに

web2py [1] は、セキュアなデータベース駆動型の Web アプリケーションをアジャイルで開発するための、フリーでオープンな Web フレームワークです。フレームワーク自身が Python [2] で書かれていて、かつ Python でプログラムすることができます。web2py はフルスタックフレームワークです、つまり完全な Web アプリケーションの開発に必要なすべてのコンポーネントを含んでいます。web2py は、Web 開発者をソフトウェア開発の良いプラクティス、いわゆるモデル / ビュー / コントローラ (MVC) パターンに導くためのガイドとなるよう、デザインされています。web2py は、データの構造の表現 (モデル) を、データの見せ方の表現 (ビュー) やアプリケーションのロジックやワークフロー (コントローラ) から分離します。web2py は開発者を助けるために、それら 3 つの構成に対して、それぞれ、設計、実装、テストできるようなライブラリを提供し、協調して動作させることができます。web2py は、セキュリティを考慮して構築されています。つまり、確立された手法に沿った形で、セキュリティの脆弱性につながるような問題を解決しています。たとえばこのフレームワークでは、すべての入力を検証し (インジェクションを防ぐため) すべての出力をエスケープし (クロスサイトスクリプティングを防ぐため) アップロードしたファイル名を変更します (ディレクトリトラバーサルを防ぐため)。web2py はアプリケーション開発者に対して、セキュリティを考慮した実装をせざるをえないようにします。web2py には、開発者が SQL を記述しなくてもよいようにするために、SQL [3] を動的に作成するデータベース抽象化レイヤ (DAL) が含まれています。DAL は、SQLite [4]、MySQL [6]、PostgreSQL [5]、MSSQL [7]、Firebird [8]、Oracle [9]、IBM DB2 [10]、Informix [11]、そして Ingres [12] について、それぞ

れのデータベースに依存しない透過的な SQL を生成することができます。さらに、Google App Engine(GAE) [13] 上で動いている時は、Google データストアに対する関数呼び出しも生成することができます。実験的にはさらに多くのデータベースに対応しています。最近のアダプタの情報を知るために、web2py のサイトやメーリングリストをチェックしてください。一旦、データベースのテーブルを定義すると、データベースとテーブルにアクセスできる十分な機能を持った Web ベースのデータベース管理インターフェースが生成されます。web2py は、Web はコンピュータである、という Web2.0 のパラダイムを完全に取り入れるための最適なフレームワークという点で、他の Web フレームワークとは異なります。実際、web2py にはインストールや設定は要求されません。Python が実行できる任意のアーキテクチャ (Windows、Windows CE、Mac OS X、iOS、Unix/Linux) で動作します。そして開発、デプロイ、メンテナンスといったアプリケーション開発の各フェーズに対して、ローカルやリモートから操作のできる Web のインターフェイスを提供しています。web2py は CPython (C による実装) および Jython (Java 実装) で動きります。Python のバージョン 2.4、2.5、2.6 および 2.7 に対応しますが、アプリケーションの後方互換性を保証できるように、”正式” には 2.5 をサポートしています。web2py は、チケットシステムを提供しています。エラーが発生した場合、チケットが発行され、エラーが管理者用に記録されます。web2py はオープンソースであり、LGPL 3 ライセンスの下に公開されています。web2py のもう 1 つの特徴は、将来のバージョンにおいて後方互換性を維持することを誓っていることです。web2py は 2007 年 10 月に初めて公開されました。新機能が追加されてバグも修正されていますが、しかし web2py 1.0 で動いているプログラムは、最新版でも動きます。

さてここで、web2py の記述の強力さとシンプルさを示すいくつかの例を示します。次のコードをみてください：

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

このコードは、”name” という文字列、”image” というアップロードが必要なもの（実際の画像）という二つのフィールドを持つ”person” というテーブルを、データベースに作成します。テーブルがすでに存在しているが定義と一致しない場合には、適切に変更されます。

上記のようにテーブルが定義されている場合、次のコードを見てください：

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

このコードは、画像もアップロードすることのできる person テーブル用の登録フォームを作成します。このコードはまた、送信されたフォームを検証し、アップロードされた画像を安全な方法でリネームし、画像をファイルに保存し、対応するレコードをデータベースに挿入し、二重投稿を防ぎ、最終的には、もし送信されたデータが検証が通らない場合にはエラーメッセージをフォームに加えます。

次のコードを見てください。

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

このコードは、訪問者が”person” テーブルのレコードを”read” できる権限を持つグループのメンバーでない限り、関数 `f` へアクセスを拒否します。もし訪問者がログインをしていない場合、ログインページ ( web2py のデフォルトとして提供されている ) にリダイレクトされます。

次のコードは、ページのコンポーネントをコードに埋め込んでいます。

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

このコードは web2py に対して、ビューの中に他のコントローラー関数 ( これは任意の関数で動作 ) によって生成されたコンテンツをロードするように指示します。コンテンツは Ajax を介して読み込まれ、現在のページ ( 現在のレイアウトが使用されます。 `other_controller` 関数のレイアウトは使用されません ) に埋め込まれます。また 読み込まれたコンテンツに含まれるすべてのフォームの送信処理をトラップするので、ページをリロードすることなく Ajax 経由で送信できます。非 web2py アプリケーションからコンテンツを読み込むことができます。

`LOAD` ヘルパーは、アプリケーションをモジュール化されたデザインにすることを助けます。本書の最後の章でくわしく解説します。

## 1.1 原則

Python のプログラミングは、主として、以下の基本原則に従います：

- 同じことを繰り返さない (DRY)
- あることを実現する方法は一通りだけである
- 暗黙的よりも明示的を良しとする

web2py は、最初の 2 つの原則は十分に取り入れています。コードの重複を抑制する健全なソフトウェアエンジニアリングのプラクティスの使用を促しています。また、Web アプリケーションの開発に共通するほぼすべてのタスク（フォームの作成や操作、セッション管理、クッキー、エラーなど）について指針を示します。

web2py が他のフレームワークと異なるのは、3 つ目の原則に関してです。これは最初の 2 つの原則と時々相反する事があります。特に、web2py はユーザーアプリケーションをインポートするのではなく、事前定義されたコンテキストの上でそれらを実行します。このコンテキストは、web2py のキーワードだけでなく、Python のキーワードも公開しています。

これらの機能は魔法のように見えるかもしれません、そうではありません。簡単に言うと、実際には、いくつかのモジュールはすでにインポートされ、あなたはそれを必要ありません。他のフレームワークの面倒な特性である、すべてのモデルやコントローラの先頭に同じモジュールをインポートすることを強いることを避けるようにしています。web2py は、独自のモジュールをインポートすることにより、時間を節約し、ミスを防ぎます。このことは、「おなじ事を繰り返さない」「あることを実現する方法は一通りだけである」という精神に基づいています。

他の Python モジュールやサードパーティ製のモジュールを使用したい場合は、他の Python プログラムと同様に、それらのモジュールを明示的にインポートしなければなりません。

## 1.2 Web フレームワーク

最も基本的なレベルでは、Web アプリケーションは、対応する URL が訪問されたときに実行されるプログラム（または関数）の集合からなります。プログラムの出力は訪問者に返され、ブラウザでレンダリングされます。

Web フレームワークの目的は、新しいアプリケーションを、迅速かつ、容易に、ミスなく構築できるようにすることです。これは、API やツールを提供することによって行われます。それにより、必要とされるコードの量が減り、単純化されます。

Web アプリケーションの開発には 2 つの古典的なアプローチがあります：

- HTML [14, 15] をプログラムで生成する
- HTML ページにコードを埋め込む

第 1 のモデルは、例えば初期の CGI スクリプトが従ったモデルです。第 2 のモデルは、PHP [16](コードは C 言語に似た PHP)、ASP( コードは Visual Basic ) JSP ( コードは Java ) などが従っているものです。

これは、実行時にデータベースからデータを取得し、選択したレコードを表示する HTML ページを返す PHP プログラムの例です：

```

1 <html><body><h1>Records</h1><?
2   mysql_connect (localhost,username,password);
3   @mysql_select_db(database) or die( "Unable to select database");
4   $query="SELECT * FROM contacts";
5   $result=mysql_query($query);
6   mysql_close();
7   $i=0;
8   while ($i < mysql_numrows($result)) {
9     $name=mysql_result($result,$i,"name");
10    $phone=mysql_result($result,$i,"phone");
11    echo "<b>$name</b><br>Phone:$phone<br /><br /><hr /><br />";
12    $i++;
13  }
14 ?></body></html>
```

このアプローチの問題は、HTML にコードが埋め込まれる際、非常に同じようなコードが、HTML を生成するために、また、データベースに問い合わせる SQL 文を生成するために必要となることです。それにより、アプリケーションの複数のレイヤーが絡み合って可読性や保守性が困難になります。この状況は Ajax アプリケーションにおいてより悪化します。さらに、アプリケーションを構成するページ(ファイル)の数とともに複雑さが増加します。

上記の例の機能は、web2py を使うと 2 行の Python コードで表現できます：

```

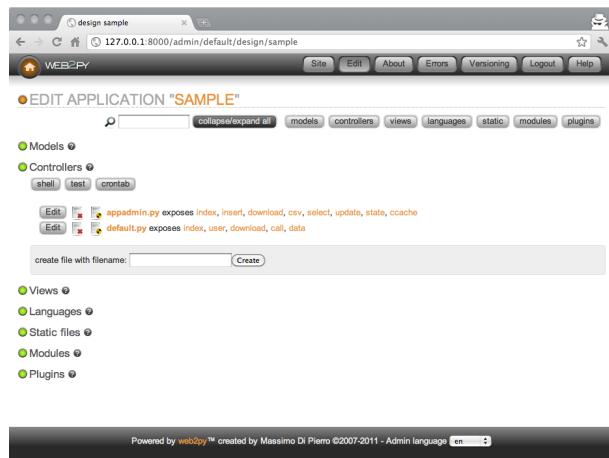
1 def index():
2   return HTML(BODY(H1('Records'), db().select(db.contacts.ALL)))
```

この単純な例では、HTML ページの構造が、HTML、BODY、H1 オブジェクトを用いてプログラム的に表現されています。データベース db は select コマンドによって問い合わせられ、最終的にすべて HTML へと加工されます。なお、db はキーワードではなく、ユーザー定義変数です。本書では、曖昧を避けるため、この名前をデータベースコネクションとして一貫して参照します。

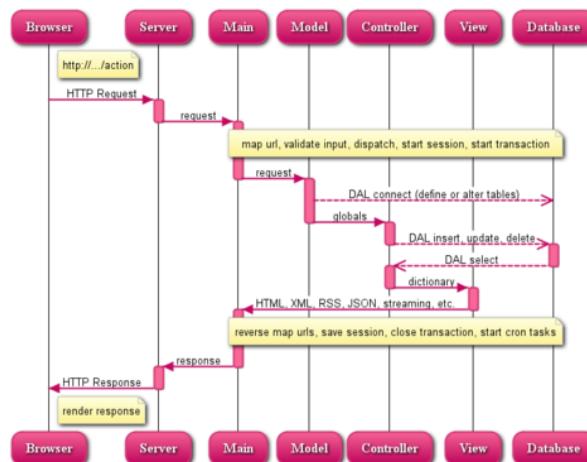
Web フレームワークは典型的に 2 つのタイプに分類されます：一つは、いくつかのサードパーティのコンポーネントを組み立てて（接合して）構築された「グルー（接着）」フレームワークです。もう一つは、緊密に組み合わせされて協調して動作するように特別に設計されたコンポーネントを組み合わせて構築された「フルスタック」フレームワークです。web2py はフルスタックフレームワークです。ほぼすべてのコンポーネントは、スクラッチで構築され一体となって動作するように設計されています。ただし、それらは web2py フレームワーク全体の外でも同様に機能します。たとえば、データベース抽象化レイヤ（DAL）や言語テンプレートは、web2py フレームワークと独立に使用することができます。これは、独自の Python アプリケーションに `gluon.dal` や `gluon.template` をインポートする事によって可能です。`gluon` は web2py のモジュールの名前で、システムライブラリを含んでいます。いくつかの web2py のライブラリは、たとえばデータベースのテーブルからフォームを構築し処理する機能などは、別の web2py の一部に依存しています。web2py はまた、他のテンプレート言語や DAL などのサードパーティー製の Python のライブラリとともに動作させることも可能です。しかし、それらはオリジナルのコンポーネントほどには緊密に連携することはないでしょう。

### 1.3 モデル、ビュー、コントローラ

web2py では、データ表現（モデル）、データの表示（ビュー）、アプリケーションの作業手順（コントローラ）を分離することを奨励しています。前述の例をもう一度考えて、web2py のアプリケーションが、この点でどのように構築されるのかを見てみましょう。ここで示すのは、web2py の MVC を編集するインターフェイスの例です：



次図は、web2py のリクエストに対する典型的な実行順序を示したものです：



この図において：

- Server は、web2py 内蔵の Web サーバー、もしくは、Apache などのサードパーティ製のサーバーにすることができます。Server はマルチスレッドで処理します。
- "main" は、メインの WSGI アプリケーションです。これは、すべての共通タスクを実行し、ユーザーアプリケーションを制御します。クッキー、セッ

ション、トランザクション、URL のルーティングと逆ルーティング、ディスパッチ処理を扱います。

Web サーバー側ですでに行われていなくても、静的ファイルを送信することができます。

- モデル、ビュー、コントローラのコンポーネントは、ユーザーアプリケーションを構成します。
- 複数のアプリケーションは、同じ web2py のインスタンスでホストすることができます。
- 破線の矢印は、( 単一/複数の ) データベースエンジンとの通信を表しています。データベースへの問い合わせは、SQL で直接( 非推奨 ) もしくは、web2py のデータベース抽象化レイヤーを利用( 推奨 ) して記述することができます。後者の場合、web2py のアプリケーションコードは特定のデータベースエンジンに依存しないものとなります。
- ディスパッチャーは、リクエストされた URL をコントローラの関数呼び出しにマッピングします。関数の実行結果は、文字列かまたは複数のシンボルからなる辞書( ハッシュテーブル )として返すことができます。辞書内のデータはビューによって表示されます。HTML ページをリクエストした場合( デフォルトの挙動 )、辞書は HTML ページの中でレンダリングされます。同じページを XML としてリクエストした場合、web2py は辞書を XML として表示することができるビューを探します。開発者はすでにサポートされるプロトコル( HTML、XML、JSON、RSS、CSV、RTF )や、追加のカスタムプロトコルで、ページを表示するビューを作成することができます。
- すべての呼び出しがトランザクション内で操作され、キャッチされない例外が発生した場合はどれも、トランザクションがロールバックされます。リクエストが成功した場合は、トランザクションがコミットされます。
- web2py はまた、セッションとセッションのクッキーを自動的に処理し、トランザクションがコミットされるときに、特に指定がない場合、セッションも同時に保存されます。
- (cron による) 定期タスクを登録し、予定された時刻に、または/かつ、特定のアクションが完了した後に実行することができます。この方法により、時間のかかるタスクやコンピュータ負荷の高いタスクを、操作性を落とすことなくバックグラウンドで実行することが可能になります。

次に示すのは、最小限かつ完結した MVC アプリケーションです。3 つのファイルから構成されています：

*"db.py"* はモデルです：

```
1 db = DAL('sqlite://storage.sqlite')
2 db.define_table('contact',
3     Field('name'),
4     Field('phone'))
```

ここでは、データベース（この例では `storage.sqlite` ファイルに保存される SQLite）に接続し、`contact` というテーブルを定義しています。テーブルが存在しない場合は、web2py がそれを作成します。そして、透過的に、バックグラウンドで、利用するデータベースエンジン固有の適切な文法において SQL コードを生成します。生成された SQL を見ることができます、データベースを、デフォルトの SQLite から、MySQL や、PostgreSQL、MSSQL、Firebird、Oracle、DB2、Informix、Interbase、Ingress、さらに、Google App Engine(SQL と NoSQL のいずれも) に置き換えるてもコードを変更する必要はありません。

一旦、テーブルが定義され作成されると、`appadmin` と呼ばれるデータベースやテーブルにアクセスするために十分な機能を持つ Web ベースのデータベース管理インターフェイスが利用できます。

*"default.py"* はコントローラです：

```
1 def contacts():
2     grid=SQLFORM.grid(db.contact, user_signature=False)
3     return locals()
```

web2pyにおいて、URL は Python モジュールと関数呼び出しにマッピングされます。この例では、コントローラーが `contacts` という単一の関数（または”アクション”）を持っています。アクションは文字列（返される Web ページ）か Python の辞書（key:value ペアの集合）、あるいはローカル変数（このサンプルを参考）を返すことになります。関数が辞書を返す場合は、コントローラ/関数と同じ名前のビューに渡され、結果的にページがレンダリングされます。この例では、`contacts` 関数は、`db.contact` テーブルのための select/search/create/update/delete のグリッドを作成します。そしてそのグリッドをビューに返します。

*"default/contacts.html"* はビューです：

```
1 {{extend 'layout.html'}}
2 <h1>Manage My Contacts</h1>
3 {{=grid}}
```

このビューは、関連するコントローラの関数（アクション）が実行された後に、web2py によって自動的に呼び出されます。このビューの目的は、返された辞書内の変数（今回は `grid`）を HTML にレンダリングすることです。ビューのファイルは HTML で書かれますが、`{} と {}` の特殊文字で区切られた Python コードを埋め込みます。これは、PHP コードのサンプルとは大きく異なります。なぜなら、HTML に埋め込まれているコードは”プレゼンテーション層”のコードだけだからです。ビューの先頭で参照されている”layout.html” ファイルは、web2py によって提供されたもので、すべての web2py アプリケーションのための基本的なレイアウトを定めます。レイアウトファイルは簡単に修正したり置き換えたりすることができます。

## 1.4 なぜ *web2py* か

web2py は数ある Web アプリケーションの中の 1 つですが、魅力的でユニークな特徴を持っています。web2py は、元々、次のような主要な動機に従いながら、教育用ツールとして開発されました。

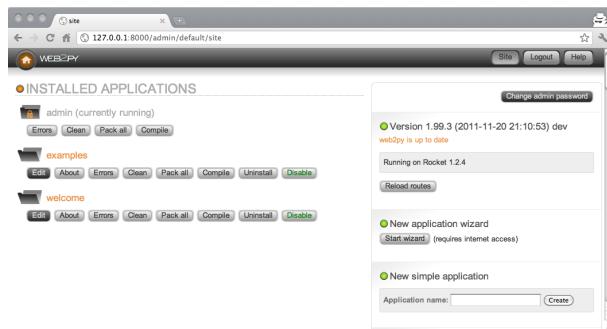
- サーバーサイドの Web 開発について、機能の妥協なしに、簡単に学ぶことができます。この理由から、web2py はインストールと設定を必要とせず、依存性がなく（ただし、配布したソースコードは Python2.5 とその標準ライブラリを必要とします）、大抵の機能は Web ブラウザのインターフェースを通して利用することができます。
- web2py は当初から安定しています。なぜなら、トップダウンの設計に従っているからです。つまり、API は実装される前に設計されました。新しい機能が追加されても、web2py は後方互換性を破りませんでしたし、これから機能追加しても破ることはないでしょう。
- web2py は最も重要なセキュリティの問題に積極的に取り組んでいます。それらは多くの現在の Web アプリケーションを悩ませるもので、後述する OWASP [19] によって定められています。
- web2py は軽量です。コアライブラリは、データベース抽象化レイヤ、テンプレート言語、すべてのヘルパーを含んで、合計 1.4MB です。すべてのソースコードは、サンプルアプリケーションや画像を含んで、合計 10.4MB です。
- web2py は必要なメモリ量が小さく、非常に高速に動作します。Timothy

Farrell によって開発された Rocket [22] WSGI Web サーバーを使用しています。これは、mod\_wsgi を利用した Apache とほぼ同等の速度です。我々のテストでは、平均的な PC で、データベースにアクセスしない平均的な動的ページにおいて約 10ms に抑えることが示されています。DAL は極めて低いオーバーヘッドしかなく、通常は 3 %未満です。

- web2py は、モデル、コントローラ、ビューを実装するために Python の構文を使用していますが、(他のすべての Python のフレームワークがしているように) モデルやコントローラをインポートはしません。その代わりにそれらを実行します。これはアプリケーションのインストール、アンインストール、そして修正を、Web サーバー（製品版だとしても）を再起動せず行なえることを意味します。また複数のアプリケーションのモジュールが互いに干渉することなく共存できることも意味します。
- web2py は、オブジェクトリレーションナルマッパー（ORM）の代わりに、データベース抽象化レイヤ（DAL）を使用します。概念的には、データベースのテーブルはそれぞれ `Table` クラスのインスタンスにマッピングされ、異なるクラスにはマッピングされません。そしてレコードは `Row` クラスのインスタンスにマッピングされ、対応するテーブルクラスのインスタンスにはマッピングされません。実用性からみると、これは SQL の構文が DAL の構文にはほぼ 1 対 1 に対応することを意味します。そして、仕組みを隠蔽化する一般的な ORM を使う時のような、メタクラスを使ったプログラミングの複雑さはありません。

WSGI [17, 18] (Web Server Gateway Interface) は、Web サーバーと Python アプリケーション間を通信するための Python 標準として取り上げられています。

これは、メインとなる web2py の admin インターフェースのスクリーンショットです：



## 1.5 セキュリティ

The Open Web Application Security Project [19](OWASP) は、アプリケーションソフトウェアのセキュリティを向上させることに焦点を置いた、自由で開かれた世界的なコミュニティです。

OWASP では、Web アプリケーションのリスクとなるセキュリティ問題のトップ 10 を挙げています。ここではこのリストを、web2py がどのようにこれらの問題を解決しているかという説明とともに、再掲します：

- 「クロスサイトスクリプティング (Cross Site Scripting, XSS): アプリケーションがユーザからデータを受信して、そのデータに対して検証やエンコードを行わずにブラウザに送信した時には、常に XSS の脆弱性が発生します。XSS は、セッションのハイジャック、Web サイトの改ざん、ワーム侵入の許可を可能にするような、犠牲となるブラウザ上で動くスクリプトを攻撃者が実行できるようにしてしまいます。」 web2py はデフォルトで、ビューに表示されるすべての変数をエスケープすることで XSS を防ぎます。
- 「インジェクションフロー (Injection Flaws): インジェクションフロー、特に SQL インジェクションは、Web アプリケーションで一般的なものです。インジェクションは、ユーザーが入力したデータがコマンドまたはクエリの一部としてインタプリタに送信されるときに発生します。攻撃者の不正なデータはインタプリタに対して意図しないコマンドを実行したり、データを変更します。」 web2py には、データベース抽象化レイヤが含まれてて、それが SQL インジェクションを防ぎます。通常は、SQL 文は開発者によって書かれません。代わりに SQL は DAL によって動的に生成され、すべての挿入データが

適切にエスケープされるようにします。

- 「悪意のあるファイルの実行：脆弱なコードによるリモートファイルのインクルード (Remote File Inclusion, RFI) は、悪意のあるコードやデータを持ち込ませることを許し、サーバー全体を危険にさらすような破壊的な攻撃につながります。」*web2py* は公開するコードのみ実行を許可し、悪意のあるファイルの実行を防止します。インポートされた関数は決して公開されません。つまり、アクションのみが公開されます。Web ベースの管理インターフェースを用いると、何が公開されているのか、いないのかを把握するのが容易になります。
- 「安全でない直接オブジェクト参照：直接オブジェクト参照とは、開発者が内部実装のオブジェクト、たとえば、ファイル、ディレクトリ、データベースのレコード、キーなどへの参照を、URL やフォームのパラメータとして公開したときに起こるものです。攻撃者は、認証なしに他のオブジェクトにアクセスし、それらの参照を操作できます。」*web2py* はいかなる内部オブジェクトも公開しません。さらに、すべての URL を検証し、ディレクトリ走査の攻撃を防ぎます。*web2py* はまた、すべての入力値を自動的に検証するフォームを作成するメカニズムを提供します。
- 「クロスサイトリクエストフォージェリ(Cross Site Request Forgery, CSRF)：CSRF 攻撃はログイン処理が行われた犠牲者のブラウザに、脆弱な Web アプリケーションに対して事前に認証されたリクエストを送信させます。これにより、犠牲者のブラウザは、攻撃者の利益となる悪意のある行為を実行させられます。CSRF は攻撃を行う Web アプリケーションと同じくらい強力です。」*web2py* は CSRF とともに、偶発的なフォームの二重投稿も防ぎます。これは、一度きりのランダムトークンを各フォームに対して割り当てることで実現しています。さらに、*web2py* は UUID をセッション Cookie として使用しています。
- 「情報漏洩と、不適切なエラー処理：アプリケーションは、設定や内部資料などの情報を意図せず漏洩したり、さまざまなアプリケーションの問題によりプライバシーの侵害を起こす恐れがあります。攻撃者はこの弱点を利用して機密データを盗み、より深刻な攻撃を実施します。」*web2py* はチケットシステムを持っています。どんなエラーでも、コードがユーザーにさらされることはありません。すべてのエラーはログに記録され、エラーの追跡を可能にするためのチケットがユーザーに発行されます。しかし、エラーとソースコー

ドに対しては管理者しかアクセスできません。

- 「不適切な認証やセッション管理：アカウントの認証情報やセッションのトークンは、しばしば適切に保護されません。攻撃者はパスワード、キー、認証トークンなどを侵害し、他のユーザーに成ります。」*web2py*では、管理者認証のための組み込み機構を提供します。そこでは、アプリケーションごとにセッションが独立に管理されます。管理インターフェイスでは、”localhost”からではないクライアントからの接続には、セキュアなセッションクッキーの使用を強制します。アプリケーションに対しては、強力なロールベースのアクセスコントロール API を用意しています。
- 「安全でない暗号保存：Web アプリケーションにおいて、データや認証情報を保護するための暗号化関数を適切に利用しているものはほとんどありません。攻撃者は、弱く保護されたデータから、身元情報の盗難や、クレジットカード詐欺などのその他犯罪を行います。」*web2py*では、保存したパスワードを保護するために、MD5 や HMAC+SHA-512 のハッシュアルゴリズムを使用しています。他のアルゴリズムも利用可能です。
- 「安全でない通信：アプリケーションは、機密情報の通信を保護する必要になったとき、ネットワークトラフィックの暗号化にしばしば失敗することがあります。」*web2py*は、SSL [21] が有効な Rocket WSGI サーバーを用意していますが、SSL の暗号化通信を提供する Apache や Lighttpd と mod\_ssl を用いることもできます。
- 「URL アクセス制限の失敗：よく見かけるのは、重要な機能を、非認証ユーザーに対して、リンクや URL の表示を防ぐだけで保護しているものです。攻撃者はこの弱点を利用ることができ、URL を直接叩くことで、認証されていない操作にアクセスし実行します。」*web2py*は URL リクエストを Python のモジュールと関数へ対応付けします。そして、どの関数が公開され、どれが認証や権限が必要かを宣言するメカニズムを用意しています。組み込みのロールベースのアクセスコントロール API は、任意の関数へのアクセスを、ログインやグループメンバーシップ、グループベースの権限に基づいて制限することを可能にします。その権限はとても細かい粒度で行われ、また、CRUD と組み合わせて、たとえば、特定のテーブル、及び / または、レコードへのアクセス権限を与えることができます。*web2py*は、デジタル署名された URL を可能にし、Ajax コールバックにデジタル署名するための API を提供します。

*web2py* はセキュリティに対する論評を受けています。その評価は参照 [20] で見

ることができます。

## 1.6 内部構造

web2py は公式サイトからダウンロードすることができます：

1 <http://www.web2py.com>

web2py は、以下のコンポーネントで構成されています：

- ライブラリ: web2py のコア機能を提供し、プログラムからアクセスできます。
- Web サーバー: Rocket WSGI Web サーバー。
- admin アプリケーション: 他の web2py アプリケーションを作成、設計、管理するために利用します。admin は web2py アプリケーションを構築するための、完結的な Web ベースの統合開発環境 (IDE) を提供します。また、Web ベースのテストや Web ベースのシェルなどの他の機能も用意されています。
- examples アプリケーション: ドキュメントとインタラクティブな例を含んでいます。examples は公式の [web2py.com](http://www.web2py.com) Web サイトのクローンで、epydoc ドキュメンテーションを盛り込んでいます。
- welcome アプリケーション: その他すべてのアプリケーションのため、基本的な雛形となるテンプレートです。デフォルトで、純粋な CSS によるカスケードメニューとユーザー認証（第 9 章で解説）が含まれます。

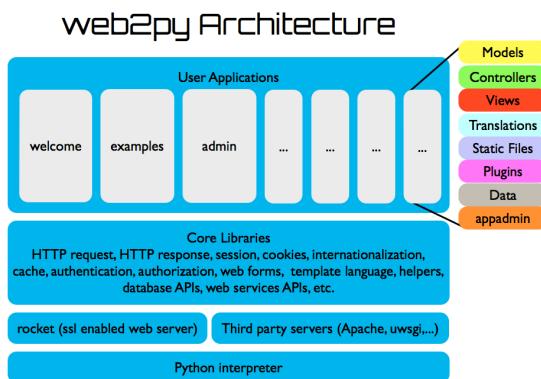
web2py はソースコード版のほか、Microsoft Windows 用や、Mac OS X 用のバイナリ形式で公開されています。

配布されたソースコードは、Python が動くプラットフォームならどこでも利用可能で、また、上記のコンポーネントを含んでいます。ソースコードを実行するには、Python2.5 があらかじめシステムにインストールされている必要があります。また、サポートされているデータベースエンジンのひとつがインストールされている必要があります。テスト目的や軽めの要望のアプリケーションには、Python 2.5 に含まれている SQLite データベースを使用するのもいいでしょう。

バイナリ版の web2py (Windows 用および Mac OS X 用) には、Python 2.5 インタプリタと SQLite データベースが含まれます。技術的には、これら 2 つは web2py のコンポーネントではありません。これらをバイナリ配布に含めること

で、web2py を難しい設定など無しにすぐに実行することが可能になります。

次の図は、全体的な web2py の構造を描いたものです：



## 1.7 ライセンス

(訳注：この節に関しては、原文をリンクしておきます。) web2py は LGPL のバージョン 3 でライセンスされています。ライセンスの全文は、参照 [31] に用意されています。

LGPL に従うならば、あなたは次の項目に当てはまる場合に：

- あなたのアプリケーションとともに、web2py（公式の web2py のバイナリバージョン含む）を再配布する場合
- 公式の web2py ライブラリを使うアプリケーションをあなたの望む任意のライセンスで公開する場合

次の項目に従う必要があります：

- あなたのアプリケーションが web2py を使っていることを明確に文書化する
- LGPL v3 ライセンスで、web2py ライブラリの修正箇所を公開する
- make clear in the documentation that your application uses web2py
- release any modification of the web2py libraries under the LGPLv3 license

ライセンスには、通常の免責条項が含まれています。( 訳注 : 訳は <http://sourceforge.jp/magazine/07/09/02/130237> より引用 )

『プログラム』には、適用可能な法で許可されている範囲において何の保証もない。書面で述べられていない限り、『コピーライト』保有者やその他の当事者は『プログラム』を「あるがまま (as is)」で、明示的、暗示的を問わず、いかなる種類の保証もなく提供する。この保証には、商用可能性や特定目的への適合性の暗黙的保証が含まれるが、これらに限定されない。『プログラム』の質や性能に関するリスクはすべてあなたに帰属する。『プログラム』に問題があると判明した場合、あなたは必要なすべての対応、補修、修正にかかる費用を負うものとする。

適用可能な法において義務づけられるか、書面による同意がない限り、『コピーライト』保有者あるいはその他『プログラム』を上記で許可された通りに改変あるいは伝達する当事者は、たとえそうした保有者や他の当事者が損害が発生する可能性について事前に通知されていたとしても、あなたに対して 損害賠償責任を有することはない。ここでいう損害には、『プログラム』の利用あるいは利用できないことから発生した一般的、特殊的、偶然的、必然的な 損害のすべてが含まれる (データの消失やデータの不正確な解釈、あなたや第三者によって被つた、あるいは『プログラム』が他のプログラムといっしょにうまく動作しなかつたために引き起こされた損害などが含まれるが、これらに限定されない)。

### 過去のバージョン

過去のバージョンの web2py(1.0.\* ~ 1.90.\* ) は、GPL2 ライセンスに加えて、現在の LGPLv3 によく似た、実用的な目的のための、例外的な商用ライセンスで公開されていました。

web2py とともに配布されるサードパーティ製のソフトウェア web2py には、gluon/contrib/ フォルダに含まれるさまざまな JavaScript、CSS ファイルなどのサードパーティ製のソフトウェアが含まれています。これらのファイルは、それらのファイルに記載されているように、それらのオリジナルなライセンスの下で web2py と一緒に配布されています。

## 1.8 謝辞

web2py は、Massimo Di Pierro によって元々作成され、著作権が取得されています。最初のバージョン（1.0）は 2007 年 10 月にリリースされました。それ以来多くのユーザーに受け入れられ、その中の幾人かは、バグレポート、テスト、デバッグ、パッチやこの本の校正に貢献してきました。

主な貢献者を、アルファベット順で紹介します：

Alexey Nezhdanov, Alvaro Justen, Andrew Willimott, Angelo Compagnucci, Anthony Bastardi, Antonio Ramos, Arun K. Rajeevan, Attila Csipa, Bill Ferret, Boris Manojlovic, Branko Vukelic, Brian Meredyk, Bruno Rocha, Carlos Galindo, Carsten Haese, Chris Clark, Chris Steel, Christian Foster Howes, Christopher Smiga, CJ Lazell, Cliff Kachinske, Craig Younkins, Daniel Lin, David Harrison, David Wagner, Denes Lengyel, Douglas Soares de Andrade, Eric Vicenti, Falko Krause, Farsheed Ashouri, Fran Boon, Francisco Gama, Fred Yanowski, Gilson Filho, Graham Dumpleton, Gyuris Szabolcs, Hamdy Abdel-Badeea, Hans Donner, Hans Murx, Hans C. v. Stockhausen, Ian Reinhart Geiser, Ismael Serratos, Jan Beilicke, Jonathan Benn, Jonathan Lundell, Josh Goldfoot, Jose Jachuf, Josh Jaques, Jose Vicente de Sousa, Keith Yang, Kenji Hosoda, Kyle Smith, Limodou, Lucas D'Avila, Marcel Leuthi, Marcel Hellkamp, Marcello Della Longa, Mariano Reingart, Mark Larsen, Mark Moore, Markus Gritsch, Martin Hufsky, Martin Mulone, Mateusz Banach, Miguel Lopez, Michael Willis, Michele Comitini, Nathan Freeze, Niall Sweeny, Niccolo Polo, Nicolas Bruxer, Olaf Ferger, Omi Chiba, Ondrej Such, Ovidio Marinho Falcao Neto, Pai, Paolo Caruccio, Patrick Breitenbach, Phy Arkar Lwin, Pierre Thibault, Ramjee Ganti, Robin Bhattacharyya, Ross Peoples, Ruijun Luo, Ryan Seto, Scott Roberts, Sergey Podlesnyi, Sharriff Aina, Simone Bizzotto, Sriram Durbha, Sterling Hankins, Stuart Rackham, Telman Yusupov, Thadeus Burgess, Tim Michelsen, Timothy Farrell, Yair Eshel, Yarko Tymciurak, Younghyun Jo, Vidul Nikolaev Petrov, Vinicius Assef, Zahariash.

この他で貢献者を載せ忘れていることは確かだと思うので、謝罪します。

私は特に、Jonathan, Mariano, Bruno, Martin, Nathan, Simone, Thadeus, Tim, Iceberg, Denes, Hans, Christian, Fran そして Patrick といった web2py の主だった

た貢献者、 Anthony, Alvaro, Bruno, Denes, Felipe, Graham, Jonathan, Hans, Kyle, Mark, Michele, Richard, Robin, Roman, Scott, Shane, Sharriff, Sriram, Sterling, Stuart, Thadeus、そしてその他の多くの方々が、この本のさまざまなバージョンを校正してくれたことに感謝します。彼らの貢献は非常に貴重でした。この本の中にあるすべての間違いについては、その責任はすべて私にあります。おそらく最後の編集によって混入されたのでしょう。また、この本の初版の発行を助けてくれた、Wiley Custom Learning Solutions の Ryan Steffen に感謝します。web2py は次の作者によるコードを含んでいます。彼らに感謝します：

Guido van Rossum for Python [2], Peter Hunt, Richard Gordon, Timothy Farrell for the Rocket [22] web server, Christopher Dolivet for EditArea [23], Bob Ippolito for simplejson [25], Simon Cusack and Grant Edwards for pyRTF [26], Dalke Scientific Software for pyRSS2Gen [27], Mark Pilgrim for feedparser [28], Trent Mick for markdown2 [29], Allan Saddi for fcgi.py, Evan Martin for the Python memcache module [30], John Resig for jQuery [32].

この本の表紙は、Young Designers に所属する Peter Kirchner がデザインしています。

Helmut Epp ( DePaul University の学長 ) , David Miller ( the College of Computing and Digital Media of DePaul University の学部長 ) , Estia Eichten ( MetaCryption LLC の会員 ) 彼らの継続的な信頼と支持に感謝します。

最後に、私が多くの時間を web2py の開発、ユーザーや協力者とのメールの交換、そしてこの本の執筆に費やしていることに忍耐強く我慢してくれている、妻の Claudia と息子の Marco に感謝しています。この本は彼らに捧げます。

## 1.9 本書について

### 1.10 *About this book*

この本はこの序章に加えて、次の章から成ります。

- 第 2 章は Python の最小限の入門です。ここでは、手続き型とオブジェクト指向プログラミングの両方のコンセプト、たとえば、ループや条件、関数呼び出しやクラスといった知識を前提に、基本的な Python の構文を説明します。

また、本書を通じて使用される Python モジュールの例を説明します。Python についてすでに知っている場合は、第 2 章は飛ばしてもかまいません。

- 第 3 章では、web2py を開始する方法を示し、管理インターフェースについて説明し、少しずつ複雑にしていきながら、さまざまな例を通して読者を案内します。文字列を返すアプリケーションから、数を数えるアプリケーション、画像付きブログ、そして、画像アップロードやコメントを可能にし、認証、権限、Web サービス、RSS フィードを提供する本格的な wiki アプリケーションまで例示します。この章を読むときは、一般的な Python の構文のために第 2 章を、利用されている機能についてのより詳細なレファレンスのために続きの章を参照する必要があるかもしれません。
- 第 4 章では、より体系的に中核となる構造とライブラリについて説明します。URL のマッピング、リクエスト、レスポンス、セッション、キャッシング、クーロン、国際化、一般的なワークフローについて説明します。
- 第 5 章は、ビューを構築するために用いられるテンプレート言語のレファレンスです。ここでは、Python コードをどのように HTML に埋め込むのか、またヘルパー（HTML を生成できるオブジェクト）を使用したデモを紹介します。
- 第 6 章では、データベース抽象化レイヤ、すなわち DAL を説明します。DAL の構文は、一連の例によって示されます。
- 第 7 章では、フォーム、フォームの検証、フォームの処理について説明します。FORM は、フォームを構築するための低レベルのヘルパーです。SQLFORM は、高レベルのフォームビルダーです。第 7 章ではまた、Create/Read/Update/Delete (CRUD) API について論じます。
- 第 8 章では、email や SMS を用いた送信によるコミュニケーションを扱います。
- 第 9 章では、web2p で使用可能な、認証、権限、拡張可能なロールベースのアクセス制御機構を扱います。メールの設定や CAPTCHA についても、認証でよく使われる所以、ここで議論します。本の第 3 版では、サードパーティー製の認証機構、たとえば、OpenID、OAuth、Google、Facebook、LinkedIn などとの統合に関する広範囲な説明も加えました。
- 第 10 章は、web2py において Web サービスを作成する方法についてです。Google Web Toolkit を Pyjamas から使う例や、Adobe Flash を PyAMF か

ら使う例を示します。

- 第 11 章は、web2py と jQuery のレシピについてです。web2py は、主にサーバーサイドのプログラミングとして設計されていますが、jQuery を内包しています。なぜならそれが、エフェクトや Ajax のために利用可能なベストなオープンソースの JavaScript ライブラリだと分かったからです。この章では、web2py とともに jQuery を効果的に使用する方法を説明します。
- 第 12 章では、モジュール化されたアプリケーションを作成する方法として、web2py のコンポーネントやプラグインを説明します。ここではよく使われる機能として、チャート、コメント、タグ、そして wiki などを実装したプラグインのサンプルを提供します。
- 第 13 章は、web2py アプリケーションの本番デプロイについてです。主に 3 つの可能な本番シナリオに取り組みます：Linux の Web サーバーまたはサーバーセットにデプロイし(主要となるデプロイの選択肢だと思います) Microsoft Windows 環境上にサービスとして動かし、Google Applications Engine (訳注：GAE) 上にデプロイします。この章ではまた、セキュリティやスケーラビリティの問題について議論します。
- 第 14 章は、特定のタスクを解決するさまざまなレシピを掲載します。アップグレードや、ジオコーディング、ページ処理、Twitter API などです。

この本では、web2py の基本的機能と web2py とともに公開される API しかカバーしません。web2py のアプライアンス(すなわち、既製アプリケーション)については説明しません。web2py のアプライアンスは、対応する Web サイト [34] からダウンロードできます。[34].

追加トピックに関しては、AlterEgo [35] や対話的な web2py FAQ で議論されていて、見つけることができます。

この本は markmin 構文を用いて書かれていて、自動的に HTML や LaTeX、PDF に変換されます。

## 1.11 要素のスタイル

PEP8 [36] は、Python でプログラミングをする時の良いスタイルのプラクティスが含まれています。web2py はこれらのプラクティスにすべて従ってるわけでは

ないことに気づくでしょう。これは省略や手抜きのためではありません。web2py の利用者はこれらのルールに従うべきであると信じていますし、それを奨励しています。我々は、これらのルールの内いくつかに従わないことを選択しました。それは web2py のヘルパー オブジェクトがユーザー定義オブジェクトに対して名前競合を起こす確率を最小限にしようとして定義したときです。

たとえば、`<div>`を表現するクラスは `DIV` ですが、Python のスタイルレファレンスによれば、`Div` とするべきです。我々は、この特定の例については、すべて大文字の”DIV”を使用するほうがより自然な選択だと考えています。さらに、このアプローチでは、必要とあればプログラマが”Div” クラスを自由に作ることができます。これらの構文はほとんどのブラウザの DOM 表記法に自然にマッピングされます（たとえば Firefox などを含みます）。

Python のスタイルガイドによると、すべて大文字の文字列は、変数ではなく定数として使用するべきとあります。先の例を続けると、`DIV` がクラスであることを考慮しても、それは二度と修正されるべきない特別なクラスになります。そうでないと、他の web2py アプリケーションが動かなくなるからです。このことから `DIV` クラスは定数のように扱われるべきものとして適格であり、我々はこの表記の選択が正当化されると信じています

要約すると、次のような慣例に従います：

- HTML ヘルパーとバリデータは、上記の議論の通りすべて大文字です（たとえば、`DIV`, `A`, `FORM`, `URL`）
- 翻訳オブジェクトの `T` は、クラス自身ではなくクラスのインスタンスであるという事実にもかかわらず、大文字で表されます。論理的に、翻訳オブジェクトは、HTML ヘルパーと似た動作を行い、プレゼンテーションの一部の表示に影響します。また、`T` はコード内に検しやすい必要があり、短かい名前を持つ必要があります。
- DAL のクラスは、Python のスタイルガイド（先頭文字を大文字にする）に従います。たとえば、`Table`、`Field`、`Query`、`Row`、`Rows` などがあります。

他のすべてのケースでは、我々は、可能な限り Python のスタイルガイド（PEP8）に従ってきたと信じています。たとえば、すべてのインスタンスオブジェクトは小文字（`request`、`response`、`session`、`cache`）であり、すべての内部クラスは大文字で始まっています。

この本のすべての例において、web2py のキーワードは太字で、文字列やコメン

トはイタリック体で記載されています。

第3版 - 翻訳: 中垣健志 レビュー: Omi Chiba

第4版 - 翻訳: 中垣健志 レビュー: Mitsuhiro Tsuda



# 2

## *Python* 言語

### 2.1 *Python* について

Python は高水準で凡用性の高いプログラミング言語です。開発者の生産性と可読性に重きを置いて設計されています。その設計思想は、プログラマの生産性とコードの読みやすさを強調しています。簡単なセマンティックと基礎的なコマンドを最小限の核となる構文を持ちます。一方で、大規模で総括的な標準ライブラリを有し、多くのオペレーティングシステム(OS)の機能を基礎としたアプリケーション・プログラム・インターフェース(API)を含んでいます。Python のコードは最小主義ですが、リスト(list)、タプル(tuple)、ハッシュテーブル(dict)、任意長の整数(long)などの組み込みオブジェクトを定義しています。

Python は、オブジェクト指向(class)、命令型(def)、関数型(lambda)などの複数のプログラミングパラダイムをサポートしています。動的型付けシステムと、参照カウントを利用した自動メモリ管理を有しています(Ruby、Perl、Scheme と同様です)。

Python は、1991 年に Guido Van Rossum によって初めてリリースされました。非営利の Python ソフトウェア財団が管理する、オープンなコミュニティベースの開発モデルになっています。Python 言語を実装している多くのインタプリタとコンパイラがあります。1 つは Java(Jython) によるものですが、ここでの簡単な説明では Guido によって開発された C 実装に言及します。

Python のオフィシャル Web サイト [2] では多くのチュートリアル、ライブラリーリファレンスや公式ドキュメントを見ることができます。

上記のドキュメントに加えて、参照 [37] や参照 [38] といった書籍も参考になるかもしれません。

すでに Python 言語に精通している場合は、本章を飛ばしてもかまわないでしょう。

## 2.2 Starting up

Microsoft Windows や Mac 版の web2py のバイナリ配布は、Python のインタープリターを同梱しています。

以下のように (DOS プロンプトで) 入力すると Windows 上で起動することができます。

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

Mac OSX では、ターミナルウィンドウで以下のコマンドを入力します (web2py.app と同じフォルダにいる必要があります)。

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

Linux や他の Unix コンピュータでは、Python がすでにインストールされているかを確認し、以下のようにシェルコマンドを入力してください。

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

もし Python2.5 以降がインストールされていない場合は、web2py を起動する前に python のダウンロードとインストールが必要です。

-S welcome は、welcome アプリケーションのコントローラー内でコマンドが実行されているかのように、対話型のシェルを動作することを web2py に指示します。web2py の対話型のコマンドラインと通常の Python コマンドラインの違いはこれだけです。

管理インターフェースはアプリケーション毎に Web ベースのシェルを提供します。次の URL で”welcome” アプリケーション用のシェルにアクセスできます。

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

本章におけるすべての例題は、通常のシェルか Web ベースのシェルで試すことができます。

### 2.3 help, dir

Python 言語には、組み込みおよびユーザ定義両方の現在のスコープにおいて、定義されたオブジェクトに関するドキュメントを取得する 2 つのコマンドが用意されています。

たとえば”1”というオブジェクトに関する help を尋ねることができます：

```

1 >>> help(1)
2 Help on int object:
3
4 class int(object)
5   |   int(x[, base]) -> integer
6   |
7   |   Convert a string or number to an integer, if possible. A floating
8   |   point
9   |   argument will be truncated towards zero (this does not include a
10  |    string
11  |    representation of a floating point number!) When converting a
12  |    string, use
13  |    the optional base. It is an error to supply a base when converting
14  |    a
15  |    non-string. If the argument is outside the integer range a long
16  |    object
17  |    will be returned instead.
18  |
19  |    Methods defined here:
20  |
21  |      __abs__(...)
22  |          x.__abs__() <==> abs(x)
23 ...

```

”1”は整数なので、int クラスとそのすべてのメソッドに関する説明が得られます。上記の例では出力結果が長いため、切り取られています。

同様に、dir コマンドを用いることで、”1”オブジェクトのメソッドのリストを得ることができます。

```

1 >>> dir(1)
2 ['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__',
3  '__delattr__', '__div__', '__divmod__', '__doc__', '__float__',

```

```

4 '__floordiv__', '__getattribute__', '__getnewargs__', '__hash__', '__
5   __hex__',
5 '__index__', '__init__', '__int__', '__invert__', '__long__', '__
6   __lshift__',
6 '__mod__', '__mul__', '__neg__', '__new__', '__nonzero__', '__oct__',
7 '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
8 '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__
9   __rfloordiv__',
9 '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rpow__', '__
10  __rrshift__',
10 '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
11 '__str__', '__sub__', '__truediv__', '__xor__']

```

## 2.4 型

Python は動的型付け言語です。つまり、変数に特定の型はなく、それゆえ宣言する必要がありません。一方で、値は特定の型を持っています。以下のようにして、変数に対して、そこに格納された値の型について問い合わせることができます。

```

1 >>> a = 3
2 >>> print type(a)
3 <type 'int'>
4 >>> a = 3.14
5 >>> print type(a)
6 <type 'float'>
7 >>> a = 'hello python'
8 >>> print type(a)
9 <type 'str'>

```

Python はまた、最初からリストや辞書などのデータ構造を内包しています。

### 2.4.1 str

Python は、ASCII 文字列と Unicode 文字列の 2 つの異なる型の文字列の使用をサポートしています。ASCII 文字列は...、...、..、""..."" で区切られたものです。トリプルクオートは、複数行の文字列を区切れます。Unicode 文字列は u で始まり、Unicode 文字列がその後に続きます。次のようにエンコーディングを選択することで、Unicode 文字列を ASCII 文字列に変換することができます。

```

1 >>> a = 'this is an ASCII string'

```

```

2 >>> b = u'This is a Unicode string'
3 >>> a = b.encode('utf8')

```

上記のコマンド実行により、`a` の結果は UTF8 でエンコードされた文字列を格納する ASCII 文字列になります。web2py は意図的に、UTF8 でエンコードされた文字列を内部で使用します。

また、さまざまな方法で、変数を文字列で記述することができます：

```

1 >>> print 'number is ' + str(3)
2 number is 3
3 >>> print 'number is %s' % (3)
4 number is 3
5 >>> print 'number is %(number)s' % dict(number=3)
6 number is 3

```

最後の表記はより明示的でエラーが起きにくいので、推奨される書き方です。

多くの Python オブジェクトは、たとえば数字は、`str` または `repr` を用いて文字列にシリアル化することができます。これら 2 つのコマンドは非常に似ていますが、若干異なる出力結果を生成します。例：

```

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

```

ユーザー定義クラスにおいて、`str` や `repr` は `__str__` と `__repr__` という特別な演算子を用いて定義/再定義できます。これらは後ほど簡単に説明します。より詳細については Python 公式ドキュメント [39] を参照してください。なお、`repr` は常に デフォルトの値を持っています。

Python 文字列のもう 1 つの重要な特徴は、リストのように反復可能なオブジェクトであるという点です。

```

1 >>> for i in 'hello':
2         print i
3 h
4 e
5 l
6 l
7 o

```

## 2.4.2 list

Python リストの主要なメソッドは追加 (append)、挿入 (insert)、削除 (delete) です：

```

1 >>> a = [1, 2, 3]
2 >>> print type(a)
3 <type 'list'>
4 >>> a.append(8)
5 >>> a.insert(2, 7)
6 >>> del a[0]
7 >>> print a
8 [2, 7, 3, 8]
9 >>> print len(a)
10 4

```

リストは次のようにスライスできます：

```

1 >>> print a[:3]
2 [2, 7, 3]
3 >>> print a[1:]
4 [7, 3, 8]
5 >>> print a[-2:]
6 [3, 8]

```

連結することも可能です：

```

1 >>> a = [2, 3]
2 >>> b = [5, 6]
3 >>> print a + b
4 [2, 3, 5, 6]

```

リストは反復可能です。つまり、それをループで回すことができます：

```

1 >>> a = [1, 2, 3]
2 >>> for i in a:
3         print i
4 1
5 2
6 3

```

リストの要素は同じ型にする必要はありません。任意の Python オブジェクトをとることができます。

*list comprehension* を使用できる典型的な状況があります。次のコードを考えて見ましょう。

```

1 >>> a = [1, 2, 3, 4, 5]
2 >>> b = []
3 >>> for x in a:
4         if x % 2 == 0:
5             b.append(x * 3)
6 >>> b
7 [6, 12]

```

このコードはアイテムリストを処理し、入力リストのサブセットを選択、編集し、新しい結果のリストを作成しています。このコードは次のようにリスト内包表記で完全に置き換えられます。

```

1 >>> a = [1, 2, 3, 4, 5]
2 >>> b = [x * 3 for x in a if x % 2 == 0]
3 >>> b
4 [6, 12]

```

### 2.4.3 tuple

タプルはリストに似ていますが、そのサイズと要素は、リストは変更可能なのにに対し、タプルは変更不可能です。タプルの要素がオブジェクトである場合、オブジェクトの属性は変更可能です。タプルは丸括弧で区切られます。

```

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

```

したがって、以下のコードはリストに対しては動作しますが、

```

1 >>> a = [1, 2, 3]
2 >>> a[1] = 5
3 >>> print a
4 [1, 5, 3]

```

タプルに対しては、要素の割り当てが機能しません。

```

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

```

タプルはリストと同様に反復可能なオブジェクトです。注意として、単一の要素でタプルを構成する際は、以下のように末尾にコンマをつけなければいけません：

```

1 >>> a = (1)
2 >>> print type(a)
3 <type 'int'>
4 >>> a = (1,)
5 >>> print type(a)
6 <type 'tuple'>

```

タプルはその不变性と括弧の利用が省略可能であることから、オブジェクトを効率的に格納するのにとても便利です。

```

1 >>> a = 2, 3, 'hello'
2 >>> x, y, z = a
3 >>> print x
4 2
5 >>> print z
6 hello

```

#### 2.4.4 dict

Python の dictionary(辞書) はキーオブジェクトを値オブジェクトにマッピングするハッシュテーブルです。例:

```

1 >>> a = { 'k': 'v', 'k2':3}
2 >>> a[ 'k']
3 v
4 >>> a[ 'k2']
5 3
6 >>> a.has_key('k')
7 True
8 >>> a.has_key('v')
9 False

```

キーは任意の型のハッシュ可能な型 (int、string、他、`__hash__`メソッド実装したクラスのオブジェクト) をとることができます。値は任意の型を使用できます。同じ辞書内の異なるキーと値は、同じ型にする必要はありません。キーが英数字の場合は、辞書は次のような代替の構文を用いて宣言することができます：

```

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

```

便利なメソッドは、`has_key`、`keys`、`values`、`items` です。

```

1 >>> a = dict(k='v', k2=3)
2 >>> print a.keys()
3 ['k', 'k2']
4 >>> print a.values()
5 ['v', 3]
6 >>> print a.items()
7 [('k', 'v'), ('k2', 3)]

```

`items` メソッドはタブルのリストを生成し、各タブルはキーとそれに対応付けられた値を格納しています。

辞書の要素とリストの要素は、`del` コマンドで削除することができます。

```

1 >>> a = [1, 2, 3]
2 >>> del a[1]
3 >>> print a
4 [1, 3]
5 >>> a = dict(k='v', h2=3)
6 >>> del a['h2']
7 >>> print a
8 {'k': 'v'}

```

内部的には、Python は、`hash` 演算子でオブジェクトを整数に変換し、その整数によってどこに値を格納するかを決めています。

```

1 >>> hash("hello world")
2 -1500746465

```

## 2.5 インデントについて

Python はインデントをコードブロックの区切りに利用しています。ブロック 1 つはコロンで終了する行から始まって、同じかそれより高いインデントが次の行として現れるまで続きます。例：

```

1 >>> i = 0
2 >>> while i < 3:
3 >>>     print i
4 >>>     i = i + 1
5 >>>
6 0
7 1
8 2

```

各インデントのレベルには 4 つのスペースを使うのが一般的です。 (不可視の) 混乱を招かないように、スペースとタブを混在しないことが得策です。

## 2.6 for...in

Python では反復可能なオブジェクトをループで回すことができます。

```

1 >>> a = [0, 1, 'hello', 'python']
2 >>> for i in a:
3     print i
4 0
5 1
6 hello
7 python

```

`xrange` は一般的なショートカットの 1 つで、すべてのリスト要素を格納せずに、反復可能な範囲オブジェクトを生成します。

```

1 >>> for i in xrange(0, 4):
2     print i
3 0
4 1
5 2
6 3

```

これは、次の C / C++ / C # の Java の構文と等価です：

```

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

```

もう 1 つの有用なコマンドは `enumerate` です。次のようにカウントしながらループ処理を行います：

```

1 >>> a = [0, 1, 'hello', 'python']
2 >>> for i, j in enumerate(a):
3     print i, j
4 0 0
5 1 1
6 2 hello
7 3 python

```

また、`range(a, b, c)` というキーワードがあります。これは、`a` で始まり `c` ずつ増加し、`b` より小さい最後の値で終わる整数のリストを返します。`a` はデフォルト値で 0 で、`c` はデフォルトで 1 です。`xrange` も似ていますが実際にリスト

を作ることはなく、そのリストに対するイテレータを生成します。これはループ処理にとってより良いです。

`break` を利用してループから抜けることができます。

```
1 >>> for i in [1, 2, 3]:
2         print i
3         break
4 1
```

`continue` を利用して、すべてのコードブロックを実行せずに、次のループ処理へ飛ぶことができます：

```
1 >>> for i in [1, 2, 3]:
2         print i
3         continue
4         print 'test'
5 1
6 2
7 3
```

## 2.7 while

Pythonにおいて `while` ループは他の多くのプログラミング言語と同じように、無限にループ処理を行い、また、各反復の前に条件のテストを行います。条件が `False`になると、ループは終わります。

```
1 >>> i = 0
2 >>> while i < 10:
3         i = i + 1
4 >>> print i
5 10
```

Pythonでは `loop...until` の構造は存在しません。

## 2.8 if...elif...else

Pythonにおける条件文の使用は直感的です：

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

```

4     elif i == 1:
5         print 'one'
6     else:
7         print 'other'
8 zero
9 one
10 other

```

”elif は””else if” の意味です。 elif 句と else 句はどちらも省略可能です。 elif は何回も利用可能ですが、 else は一度だけです。 複雑な条件文は not、 and、 or の演算子を利用し作成することができます。

```

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

```

## 2.9 try...except...else...finally

Python は、割り込みを投げる、例外を起こすことができます:

```

1 >>> try:
2     a = 1 / 0
3 >>> except Exception, e:
4     print 'oops: %s' % e
5 >>> else:
6     print 'no problem here'
7 >>> finally:
8     print 'done'
9 oops: integer division or modulo by zero
10 done

```

例外が発生したときは、 except 句で補足され、その句が実行されますが、 else 句は実行されません。例外が発生しなかった場合は、 except 句は実行されず、 else 句が実行されます。 finally 句は常に実行されます。

異なる例外が発生する可能性があるために複数の except 句をとることができます:

```

1 >>> try:
2     raise SyntaxError
3 >>> except ValueError:
4     print 'value error'
5 >>> except SyntaxError:
6     print 'syntax error'

```

```
7 syntax_error
```

`else` 句と `finally` 句は省略可能です。

以下は、組み込みの Python 例外のリストと (web2py で定義されている)HTTP 例外です。

```
1 BaseException
2     +-- HTTP (defined by web2py)
3     +-- SystemExit
4     +-- KeyboardInterrupt
5     +-- Exception
6         +-- GeneratorExit
7         +-- StopIteration
8         +-- StandardError
9             |     +-- ArithmeticError
10            |     +-- FloatingPointError
11            |     +-- OverflowError
12            |     +-- ZeroDivisionError
13            |     +-- AssertionError
14            |     +-- AttributeError
15            |     +-- EnvironmentError
16            |     +-- IOError
17            |     +-- OSError
18            |         +-- WindowsError (Windows)
19            |         +-- VMSError (VMS)
20         +-- EOFError
21         +-- ImportError
22         +-- LookupError
23             |     +-- IndexError
24             |     +-- KeyError
25             |     +-- MemoryError
26             |     +-- NameError
27                 |     +-- UnboundLocalError
28             |     +-- ReferenceError
29             |     +-- RuntimeError
30                 |     +-- NotImplemented
31             |     +-- SyntaxError
32                 |     +-- IndentationError
33                     |     +-- TabError
34             |     +-- SystemError
35             |     +-- TypeError
36             |     +-- ValueError
37                 |     +-- UnicodeError
38                     |     +-- UnicodeDecodeError
39                     |     +-- UnicodeEncodeError
40                     |     +-- UnicodeTranslateError
41     +-- Warning
```

```

42      +-- DeprecationWarning
43      +-- PendingDeprecationWarning
44      +-- RuntimeWarning
45      +-- SyntaxWarning
46      +-- UserWarning
47      +-- FutureWarning
48      +-- ImportWarning
49      +-- UnicodeWarning

```

各項目の詳細については、Python 公式ドキュメントを参照してください。web2py は、HTTP と呼ばれる新しい例外を 1 つだけ公開しています。その例外が発生すると、プログラムは HTTP エラーページを返すようになります（詳細は第 4 章を参照してください）。

任意のオブジェクトは例外として発生させることができます、それには組み込みの exception クラスを拡張したオブジェクトの利用を推奨します。

## 2.10 def...return

関数は `def` を使って宣言されます。以下は典型的な Python の関数です。

```

1 >>> def f(a, b):
2         return a + b
3 >>> print f(4, 2)
4 6

```

引数や戻り値の型を指定する必要（または方法）はありません。この例では関数 `f` は 2 つの引数を取るように定義されています。

関数はこの章で初めて `scope` や `namespace` という概念を紹介する記述です。上記の例で、`a` と `b` の変数は `f` 関数のスコープの外では未定義になります。

```

1 >>> def f(a):
2         return a + 1
3 >>> print f(1)
4 2
5 >>> print a
6 Traceback (most recent call last):
7   File "<pyshell#22>", line 1, in <module>
8     print a
9 NameError: name 'a' is not defined

```

関数のスコープの外で定義された変数は関数内でも使用可能です; 次の例で変数 `a` がどのように扱われているかを検証してみてください。

```

1 >>> a = 1
2 >>> def f(b):
3         return a + b
4 >>> print f(1)
5 2
6 >>> a = 2
7 >>> print f(1) # new value of a is used
8 3
9 >>> a = 1 # reset a
10 >>> def g(b):
11         a = 2 # creates a new local a
12         return a + b
13 >>> print g(2)
14 4
15 >>> print a # global a is unchanged
16 1

```

もし `a` が編集されると、後続の関数はグローバル変数 `a` の値を使用します。これは関数定義が宣言時の変数 `a` の値ではなく、変数 `a` の保管場所と紐づいていくためです。しかし、`g` 関数内部で `a` が割り振られた場合、新しいローカル変数 `a` がグローバルの値を隠すため、グローバル変数 `a` は影響を受けません。外部スコープ参照は *closures* の作成で実現できます。

```

1 >>> def f(x):
2         def g(y):
3             return x * y
4         return g
5 >>> doubler = f(2) # doubler is a new function
6 >>> tripler = f(3) # tripler is a new function
7 >>> quadrupler = f(4) # quadrupler is a new function
8 >>> print doubler(5)
9 10
10 >>> print tripler(5)
11 15
12 >>> print quadrupler(5)
13 20

```

関数 `f` が新しい関数を作成し、`g` のスコープは完全に `f` の内部にあります。Closures は非常に強力です。

関数の引数はデフォルト値を取ることができます、複数の結果を返すことができます。

```

1 >>> def f(a, b=2):
2     return a + b, a - b

```

```

3 >>> x, y = f(5)
4 >>> print x
5 7
6 >>> print y
7 3

```

引数を明示的に名称で渡すこともできます。これは呼び出し側で指定された引数の順序が関数で定義された引数の順序と異なってもよいことを意味します。

```

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

```

関数はまた、可変数の引数を取ることができます。

```

1 >>> def f(*a, **b):
2     return a, b
3 >>> x, y = f(3, 'hello', c=4, test='world')
4 >>> print x
5 (3, 'hello')
6 >>> print y
7 {'c':4, 'test':'world'}

```

ここで、名前付きで渡されなかった引数(3, 'hello')はリスト`a`に格納され、名前付きで渡された(`c`と`test`)は辞書`b`に格納されます。

逆に、リストやタプルは、展開しながら、個別の位置引数を要求する関数に渡すことができます：

```

1 >>> def f(a, b):
2     return a + b
3 >>> c = (1, 2)
4 >>> print f(*c)
5 3

```

辞書もまた、展開しながら、キーワード引数に引き渡すことができます。

```

1 >>> def f(a, b):
2     return a + b
3 >>> c = {'a':1, 'b':2}
4 >>> print f(**c)
5 3

```

### 2.10.1 lambda

`lambda` を使用することでとても簡単で簡潔な無名関数を作成することができます。

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

”`lambda [a] : [b]`” 式は、文字通り”引数 [a] を持ち [b] を返す関数”として読みます。`lambda` 式自体は無名関数ですが、変数に保存されることで名前を獲得しています。`def` のスコープの法則は `lambda` にも同様に適用され、上記の例にある `a` に関する `def` を使った関数宣言と全く同じです。

```
1 >>> def a(b):
2         return b + 2
3 >>> print a(3)
4 5
```

`lambda` の長所はその簡潔さだけです。しかし、簡潔さはある状況で非常に便利です。リスト上の要素全てに関数を適用し新しいリストを作成する `map` という名前の関数を考えて見ましょう。

```
1 >>> a = [1, 7, 2, 5, 4, 8]
2 >>> map(lambda x: x + 2, a)
3 [3, 9, 4, 7, 6, 10]
```

`lambda` の代わりに `def` を使用した場合、コード量は 2 倍になります。`lambda` の主な欠点（Python の実装で）は单一式しか許可しない点です。長い記述の関数の場合は関数名を指定する手間のある `def` 使用することで、関数の記述量が増えていっても対応できます。`def` や `lambda` が関数を *curry*（ネスト）することができるよう、既存の関数を包括した新しい関数を作成することで新しい関数が異なる変数セットを持つことができます。

```
1 >>> def f(a, b): return a + b
2 >>> g = lambda a: f(a, 3)
3 >>> g(2)
4 5
```

関数をネストすることが便利な状況はいろいろありますが、以下で挙げるキャッシュの例は web2py で特に便利な使用方法です。引数が素数であるかを確認する高負荷な関数を考えて見ましょう。

```

1 def isprime(number):
2     for p in range(2, number):
3         if (number % p) == 0:
4             return False
5     return True

```

この関数は明らかに多くの時間を必要とします。key、関数、秒数の3つの引数を受け取る `cache.ram` というキャッシュ関数があると仮定しましょう。

```

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

```

初回の実行時は、`f()` 関数を呼び出し、メモリ上の辞書（仮に”d”とします）に結果を保存し値を返します。このため値は以下のようになります。

```

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

```

2回目の実行時は、もし `key` が辞書に存在し、指定された秒数(60)よりデータが古くない場合、関数を実行しないで対応する値を返します。

```

1 value = d['key']

```

入力値に対応する `isprime` 関数の結果をキャッシュするにはどのようにすればよいでしょうか？これは以下のようになります：

```

1 >>> number = 7
2 >>> seconds = 60
3 >>> print cache.ram(str(number), lambda: isprime(number), seconds)
4 True
5 >>> print cache.ram(str(number), lambda: isprime(number), seconds)
6 True

```

結果は同じですが、初回に `cache.ram` が実行された場合だけ `isprime` が呼ばれ、2回目は呼ばれません。

`def` や `lambda` で作成された *Python* 関数は異なる引数の組み合わせの観点から既存の関数をリファクタリングすることができます。`cache.ram` と `cache.disk` は *web2py* のキャッシュ関数です。

## 2.11 class

*Python* は動的型付けなので、*Python* のクラスとオブジェクトは少し変わったものに見えるかもしれません。実際、クラスを宣言する際にメンバ変数（属性）

を必ずしも定義する必要はなく、同じクラスから作られた異なるインスタンスはそれぞれ違うメンバ変数(属性)を持つことができます。一般的に属性は、クラスではなくインスタンスに関連付けられます(ただし、クラス属性として宣言された場合は別です。これは C++/Java での”静的メンバ変数と同じです)。

例を示します：

```

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

```

ここで、`pass`は何もしないというコマンドであることに注意してください。この場合、何も含んでいない `MyClass` というクラスを定義するために用いられています。`MyClass()` はクラスのコンストラクタを呼び出し(この場合、デフォルトのコンストラクタ)、オブジェクト、つまり、このクラスのインスタンスを返します。クラス定義における(`object`)の部分は、このクラスが組み込みの `object` クラスを拡張したものであることを示しています。これは必須ではないですが、推奨される書き方です。

次により複雑なクラスの例を示します：

```

1 >>> class MyClass(object):
2 >>>     z = 2
3 >>>     def __init__(self, a, b):
4 >>>         self.x = a, self.y = b
5 >>>     def add(self):
6 >>>         return self.x + self.y + self.z
7 >>> myinstance = MyClass(3, 4)
8 >>> print myinstance.add()
9

```

クラスの内部で定義された関数はメソッドになります。いくつかのメソッドは、特別な予約済みの名前を持ちます。たとえば、`__init__` はコンストラクタです。すべての変数は、メソッドの外で定義されたものでない限り、メソッドのローカル変数です。たとえば、`z` はクラス変数です。これは、C++ の静的メンバ変数と同じで、そのクラスのすべてのインスタンスに対して同じ値を保持します。

ここで、`__init__` は 3 つの引数をとり、`add` は 1 つの引数をとっているのに対し、それらはそれぞれ 2 つの引数と、0 個の引数によって呼ばれている点に注意してください。最初の引数は、慣例的に、メソッド内で利用されるローカルな名前を示していて、現在のオブジェクトを参照します。ここでは、`self` を現在の

オブジェクトを参照するのに利用しています。ただし、任意の他の名前を用いることも可能です。`self` は C++ の `*this` や Java の `this` と同じ役割を担います。ただし、`self` は予約語ではありません。

この構文は、ネストしたクラス、たとえばあるクラス内のメソッドにおいてローカルなクラス、を宣言するときに、曖昧さを避けるために必要です。

## 2.12 特殊属性、メソッド、演算子

2つのアンダスコアから始まるクラス属性、メソッド、演算子は、一般にプライベート（クラス内でのみ使用し、クラス外から呼び出されない）であることを意図しますが、これはインタプリタによって強制される慣例ではありません。

そのうちのいくつかは予約されたキーワードで、特別な意味を持っています。

例として、そのなかの 3 つを示します。

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

これらは、たとえば、リストのように振舞うコンテナオブジェクトの作成に利用できます：

```

1 >>> class MyList(object):
2     def __init__(self, *a): self.a = list(a)
3     def __len__(self): return len(self.a)
4     def __getitem__(self, i): return self.a[i]
5     def __setitem__(self, i, j): self.a[i] = j
6 >>> b = MyList(3, 4, 5)
7 >>> print b[1]
8 4
9 >>> b.a[1] = 7
10 >>> print b.a
11 [3, 7, 5]
```

他の特殊演算子としては、クラスにおいて属性の取得と設定を定義する `__getattr__` と `__setattr__` や、算術演算子をオーバーロードする `__sum__` や `__sub__` などがあります。これらの演算子の利用についてもっと知りたい場合は、より高度な書籍を参照してください。また、`__str__` と `__repr__` 演算子に

についてはすでに言及してあります。

### 2.13 ファイル入力/出力

Python では、以下のようにしてファイルを開き、書き込むことができます。

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

同様に、以下のようにしてファイルを読み出すことができます。

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

もう一つの方法として、”rb”を用いてバイナリモードで読むことが可能で、”wb”を用いてバイナリモードで書き込むことができます。さらに、追記モード”a”においてファイルを開くこともできます。標準の C 言語表記と同じです。

`read` コマンドは省略可能な引数であるバイト数を取ります。また、ファイル内の任意の箇所に飛ぶ場合は、`seek` を利用します。

`read` を利用して飛んだ場所からファイルを読むことができます。

```
1 >>> print file.seek(6)
2 >>> print file.read()
3 world
```

ファイルを閉じるときは次のようにします。

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

*C*Python として知られる *Python* の標準実装では、変数は参照カウントを使用しており、使用しているファイルもこれで扱っています。このため、*C*Python は参照カウントの開いているファイル数がゼロになると、ファイルはクローズされていると判断し変数を破棄します。しかし *PyPy* などの別の実装では参照カウントの代わりにガベージコレクションが使用されます。これはもし大量のファイルが一度にオープンされた場合、*gc* (ガベージコレクション) がそれをクローズして破棄する前にはエラーが発生する可能性があります。このため、もう使用しない場合は明示的にファイルを

クローズするのが推奨されます。*web2py* は `read_file()` と `write_file()` の 2 つのヘルパ関数を提供します。これは `gluon.fileutils` 名前空間の内部でファイルアクセスをカプセル化し、使用されたファイルが適切にクローズされているかを確認します。

*web2py* を使用している場合、カレントディレクトリの場所を知る必要はありません。なぜなら、それは *web2py* の設定に依存するからです。`request.folder` 変数は、現在のアプリケーションへのパスを保持しています。パスは、後述する `os.path.join` コマンドによって連結することができます。

## 2.14 exec, eval

Java と異なり、Python は真のインタプリタ言語です。つまり、文字列として格納された Python コードを実行する能力があります。例:

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

何が起きたのでしょうか？ 関数 `exec` は、インタープリタに自分自身を呼び出すように命令し、引数として渡された文字列の中身を実行します。また、辞書のシンボルによって定義されたコンテキストの中で、文字列の中身を実行することも可能です：

```
1 >>> a = "print b"
2 >>> c = dict(b=3)
3 >>> exec(a, {}, c)
4 3
```

ここでは、インタプリタが、文字列 `a` を実行したときに、`c` で定義されたシンボル（この例では `b`）を参照しています。ただし、`c` や `a` 自身を参照することはできません。これは制限された環境とは大きく異なります。`exec` は内部コードができることに制限を加えないからです。コードが利用出来る変数セットを単に定義しているだけです。

関連する関数として `eval` があります。これは、`exec` と非常に似た動きをしますが、評価される引数が値になることを想定し、その値を返すという点で異なります。

```

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

```

## 2.15 import

Pythono が本当に強力なのは、そのライブラリモジュールがあるからです。それらは、大規模で一貫性のあるアプリケーション・プログラム・インターフェース (API) を多くの（大抵オペレーティング・システムから独立した）システムライブラリに提供します。

たとえば、乱数ジェネレーターを利用する必要がある場合、次のようにします：

```

1 >>> import random
2 >>> print random.randint(0, 9)
3 5

```

この例では、0 から 9 の間のランダムな整数を表示します（ここでは 5 が表示されています）。`randint` 関数は `random` モジュール内で定義されています。モジュールからオブジェクトを現在の名前空間にインポートすることも可能です：

```

1 >>> from random import randint
2 >>> print randint(0, 9)

```

もしくは、モジュールからすべてのオブジェクトを現在の名前空間にインポートすることも可能です：

```

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

```

さらに、すべてを新しく定義した名前空間にインポートすることも可能です：

```

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

```

本書の残りの部分では、`os`、`sys`、`datetime`、`time`、`cPickle` といったモジュールに定義されたオブジェクトをよく利用します。

すべての `web2py` オブジェクトは、`gluon` と呼ばれるモジュールを介してアクセスすることができます。これは、後述の章にて扱います。内部的には、`web2py` は多くの `Python` モジュール（たとえば `thread` など）を使用し

ています。しかし、利用者が直接それらにアクセスする必要はほとんどないでしょう。

以下の小節では、最も利用されるそれらのモジュールを考えます。

### 2.15.1 os

このモジュールは、オペレーティング・システム API へのインターフェイスを提供します。例：

```
1 >>> import os
2 >>> os.chdir('..')
3 >>> os.unlink('filename_to_be_deleted')
```

`chdir` のような `os` のいくつかの関数は、`web2py` のおいて使用しないでください。スレッドセーフではないからです。

`os.path.join` はとても便利で、OS に依存しない形でパスの連結が可能になります。

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

システム環境変数は `os.environ` を介してアクセスできます。

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

これは読み取り専用の辞書です。

### 2.15.2 sys

`sys` モジュールは多くの変数と関数を持ちますが、最も利用するのは `sys.path` です。これは、Python がモジュールを探すためのパスのリストを保持しています。モジュールをインポートしようとしたとき、Python は `sys.path` 内にリストされたすべてのフォルダーをチェックします。どこかの場所において追加のモジュールをインストールして、それを Python に探させたい場合、その場所へのパスを `sys.path` に追加しなければなりません。

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

web2py が実行されている時は、Python はメモリ内に常駐し、多くのスレッドが HTTP リクエストを処理している一方、`sys.path` は 1 つしかありません。メモリリークを回避するにためには、パスを追記する前に、それがすでに存在しているかを確認するのが最良です。

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

### 2.15.3 datetime

`datetime` モジュールの使い方を説明するいくつかの例を紹介します。

```
1 >>> import datetime
2 >>> print datetime.datetime.today()
3 2008-07-04 14:03:90
4 >>> print datetime.date.today()
5 2008-07-04
```

ローカルタイムではなく、UTC タイムに基づいたタイムスタンプデータが必要になるかもしれません。その場合、次のような関数を用いることができます：その場合には、次の関数を使用することができます：

```
1 >>> import datetime
2 >>> print datetime.datetime.utcnow()
3 2008-07-04 14:03:90
```

`datetime` のモジュールには、`date`、`datetime`、`time`、`timedelta` など、さまざまなクラスが含まれています。2つの`date`、2つの`datetime`、2つの`time`オブジェクト間の差は、`timedelta` になります：

```
1 >>> a = datetime.datetime(2008, 1, 1, 20, 30)
2 >>> b = datetime.datetime(2008, 1, 2, 20, 30)
3 >>> c = b - a
4 >>> print c.days
5 1
```

web2pyにおいて、`date` と `datetime` は、データベースへ渡すときや戻されるときに、対応する SQL の型を格納するのに利用されます。

### 2.15.4 time

time モジュールは date や datetime と異なり、(1970 年から始まる) エポックからの秒数として時間を表現します。

```
1 >>> import time
2 >>> t = time.time()
3 1215138737.571
```

秒の時間と datetime の時間とを変換する関数については Python ドキュメントを参照してください。

### 2.15.5 cPickle

cPickle はとても強力なモジュールです。cPickle は自己参照オブジェクトを含む、ほぼすべての Python オブジェクトをシリアル化することができる関数を提供します。たとえば、次のような奇妙なオブジェクトを構築します：

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

そして以下のようにします：

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

ここで、`b` は `a` の文字列表現で、`c` は `b` をデシリアル化して生成された `a` のコピーです。cPickle は、ファイルへシリアル化、ファイルからデシリアル化することも可能です：

```
1 >>> cPickle.dump(a, open('myfile.pickle', 'wb'))
2 >>> c = cPickle.load(open('myfile.pickle', 'rb'))
```

第 3 版 - 翻訳: 細田謙二 レビュー: Omi Chiba

第 4 版 - 翻訳: Omi Chiba レビュー: Fumito Mizuno

# 3

## 概要

### 3.1 はじめよう

web2py には Windows と Mac OS X 用のバイナリパッケージが提供されています。Python インタプリタが含まれているので、事前のインストールは不要です。また、Windows、Mac、Linux、その他の Unix システムで動作するソースコードもあります。Windows と OS X のバイナリ版には、動作させるのに必要な Python インタプリタが含まれます。ソースコードパッケージは、Python がすでにコンピュータにインストールされていることを前提としています。web2py はインストールする必要がありません。始めるには、利用するオペーティングシステム用にダウンロードした zip ファイルを解凍して、適切な `web2py` ファイルを実行します。

Windows の場合は、次のファイルを実行します：

```
1 web2py.exe
```

OS X の場合は、次のファイルを実行します：

```
1 open web2py.app
```

Unix および Linux では、以下のコマンドを入力してソースから実行します：

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

Windows 上でソースコードから web2py を実行するには、まず Mark Hammond's

"Python for Windows extensions からインストールをして、次のファイルを実行します。

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

web2py プログラムはさまざまなコマンドラインオプションを受け入れます。これは後ほど説明します。

デフォルトでは、起動時に、起動ウィンドウが表示されます。その後、画面には GUI ウィジェットが表示され、一度限りの管理パスワード、Web サーバーに利用されるネットワークインターフェイスの IP アドレス、リクエストを受けるポート番号を選択するように求められます。デフォルトでは、web2py は 127.0.0.1:8000 (ローカルホストのポート 8000 番) 上で動作しますが、任意の取りうる IP アドレスとポートでも動作させることができます。ネットワークインターフェイスの IP アドレスは、コマンドラインを開いたあと、Windows の場合は ipconfig コマンドを、OS X や Linux の場合は ifconfig コマンドを入力する事で確認することができます。これ以降、web2py はローカルホスト (127.0.0.1:8000) 上で実行しているものとします。任意のネットワークインターフェイス上に web2py を公開して動作させるときは、0.0.0.0:80 を使用してください。



管理者のパスワードを指定しない場合は、管理インターフェイスは無効になります。これは管理インターフェイスが公開されてしまう事を防ぐ、セキュリティ上の対策です。

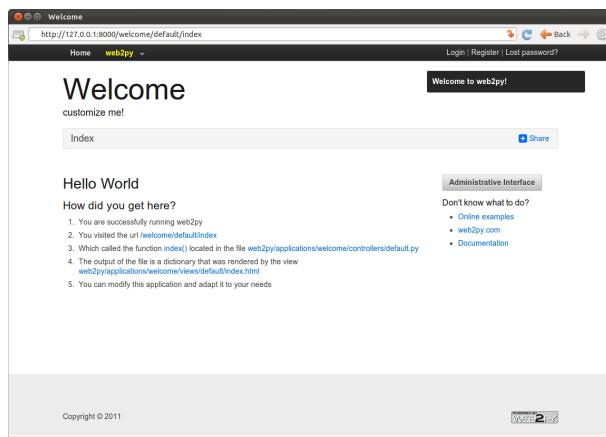
管理者インターフェイス admin は、web2py を Apache と mod\_proxy を組み合わせた環境で実行させない限り、ローカルホストからしかアクセスできません。もし管理インターフェイスがプロキシを検出した場合は、セッションクッキーは保護されることとなり、管理インターフェイスのログインは、クライアントとブ

ロキシが HTTPS 上で通信しない限り、機能しません。これはセキュリティ対策のためです。クライアントと管理者間のすべての通信は、常にローカルまたは暗号化されている必要があります。そうしないと、攻撃者は中間者攻撃やリプレイ攻撃を行うことができ、サーバ上で任意のコードを実行することが可能になるからです。

管理者用のパスワードが設定されたら、web2py は次のページから Web ブラウザを立ち上げます：

1 <http://127.0.0.1:8000/>

デフォルトのブラウザがない場合、Web ブラウザを開いて、URL を入力してください。

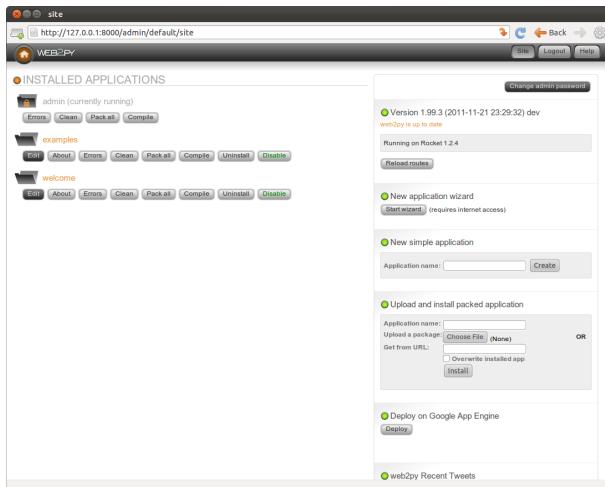


”administrative interface” をクリックすると、管理インターフェイス用のログインページが表示されます。



管理者パスワードは、起動時に指定したパスワードと同じです。なお、管理者は1人だけで、したがって、1つの管理パスワードしかありません。セキュリティ上の理由により、web2py が起動するたびに開発者は毎回新しいパスワードを尋ねられます。ただし、<recycle>オプションを指定するとその限りではありません。これは、web2py の認証機構とは区別されます。

管理者が web2py にログインすると、ブラウザは”site” ページへとリダイレクトされます。



このページではすべてのインストールされている web2py アプリケーションが列挙され、管理者はそれらを管理することができます。

- admin アプリケーション。これは現在あなたが利用しているものです。
- examples アプリケーション。これはオンラインの対話的なドキュメントと、web2py の公式サイトのレプリカ（複製）を持っています。
- welcome アプリケーション。これは、他の web2py アプリケーションのための基本的なテンプレートです。これは、離形となるアプリケーションとして参照されます。これは、起動時にユーザを迎えるアプリケーションです。

すぐに利用できる web2py アプリケーションは、web2py のアプライアンスとして参照されます。多くのフリーで利用可能な [?] からダウンロードすることができます。web2py のユーザは、オープンソースかクローズドソース（コンパイルされてパックされたもの）のいずれの形式でも、新しいアプライアンスを投稿

することが勧められています。

admin アプリケーションの *site* ページから、次の操作を行うことができます：

- **install** アプリケーションのインストールは、ページの右下にあるフォームを埋めて行います。アプリケーションの名前を入力し、パッケージ化されたアプリケーションを含むファイルを選択、または、アプリケーションが用意されている URL を指定して、”submit” ボタンをクリックします。
- **uninstall** アプリケーションのアンインストールは対応するボタンをクリックして行います。確認ページが用意されています。
- **create** 新しいアプリケーションの作成は、名前を入力して”create” ボタンをクリックして行います。
- **package** 配布用のアプリケーションのパッケージングは、対応するボタンをクリックして行います。ダウンロードされたアプリケーションは、データベースを含むすべてを保持する tar ファイルです。このファイルは untar してはいけません。admin でインストールしたときに web2py によって自動的にアンパッケージングされます。
- **clean up** セッションや、エラー、キャッシュファイルなどのアプリケーションの一時ファイルをクリーンアップします。
- **EDIT** アプリケーションを編集します。

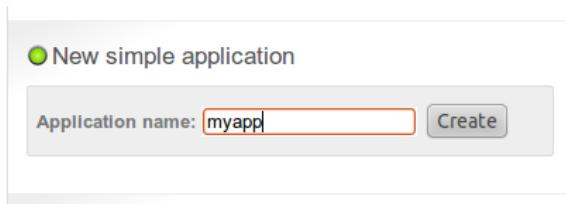
admin を用いて新規のアプリケーションを作成するときは、”welcome” 離形アプリのクローンから始まります。その中の ”models/db.py” は、SQLite データベースを作成、接続し、Auth、Crud、Service をインスタンス化し、設定します。そこにはまた、”controller/default.py” があり、”index”、”download”、ユーザー管理のための ”user”、サービスのための ”call” というアクションを提供しています。ここから先、これらのファイルが削除されていることを前提としています。つまり、アプリをスクラッチから作成していきます。

web2py には wizard も付属しています。これについては後の章で説明します。wizard は、web 用に準備されたレイアウトやプラグインと高いレベルのモデルの記述に基づいた足場となるコードを生成することができる、もう一つの仕組みです。

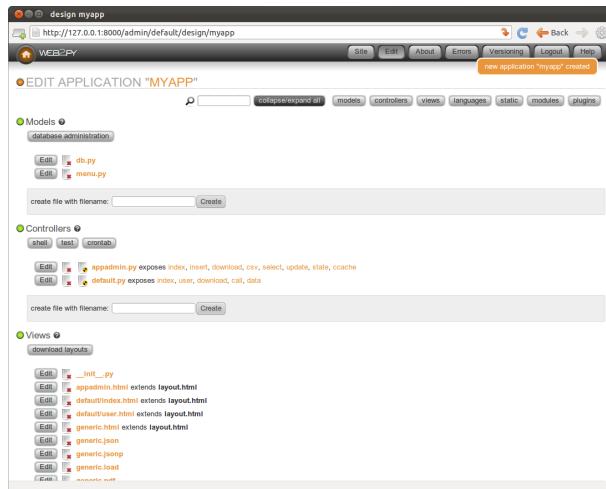
### 3.2 挨拶しよう

ここでは例として、ユーザーに”Hello from MyApp”というメッセージを表示する簡単な Web アプリケーションを作成します。このアプリケーションを”myapp”と呼びます。また、同じユーザがページを何回訪問したかをカウントするカウンタを追加します。

新しいアプリケーションは、admin の中の site ページの右上にあるフォームに、アプリケーション名を入れることで簡単に作成できます。



[create] ボタンを押すと、アプリケーションは組み込みの welcome アプリケーションのコピーとして作成されます。



新しいアプリケーションを実行するには、次の URL を開いてください：

1 <http://127.0.0.1:8000/myapp>

これで、welcome アプリケーションのコピーが作成できました。

アプリケーションを編集するには、新しく作成されたアプリケーションの `design` ボタンをクリックしてください。

EDIT ページは、アプリケーションの内部がどのようなものかを示しています。すべての web2py アプリケーションは一定のファイルから構成され、それらのほとんどは次の 6 つのカテゴリに分類されます：

- `models`: データ表現を記述します。
- `controllers`: アプリケーションのロジックとワークフローを記述します。
- `views`: データの表示方法を記述します。
- `languages`: アプリケーションで表示される内容を、他の言語に翻訳するための方法を記述します。
- `modules`: アプリケーションに属する Python モジュールです。
- `static files`: 静的画像ファイル、CSS ファイル [40, 41, 42]、JavaScript ファイル [43, 44]、などです。
- `plugins`: 一緒に動作するよう設計されたファイルの集合です。

すべてのファイルは、モデル - ビュー - コントローラのデザインパターンに沿ってきちんと構成されます。`edit` ページの各セクションは、アプリケーションフォルダ内のサブフォルダに対応します。

なお、セクションの見出をクリックすると、その中身の表示/非表示を切り替えることができます。同様に `static files` の下のフォルダ名も、折りたたむことができます。

セクション内の各ファイルは、サブフォルダにある物理的なファイルに対応しています。管理インターフェイスからファイルに対して行えるすべての操作 (`create`、`edit`、`delete`) は、好みのエディタを使用してシェルから実行することもできます。

アプリケーションは上記以外にもデータベース、セッションファイル、エラーファイルを含みますが、`edit` ページには掲載されません。これらのファイルは管理者ではなく、アプリケーション自身によって作成・編集されるためです。

コントローラは、アプリケーションのロジックやワークフローを含みます。すべての URL は、コントローラの関数（アクション）のいずれか 1 つに呼び出しにマッピングされます。`”appadmin.py”` と `”default.py”` という 2 つのデフォル

トコントローラが用意されています。appadmin は、データベース管理用のインターフェイスを提供しますが、ここでは必要ありません。”default.py” は、編集するべきファイルであり、URL に対応するコントローラが存在しない時にデフォルトで呼び出されます。次のように”index” 関数を編集してみましょう：

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

オンラインのエディタは次のような表示になります：

The screenshot shows the Web2py application management interface. The title bar says 'edit myapp/controllers/default.py'. The URL in the address bar is 'http://127.0.0.1:8000/admin/default/edit/myapp/controllers/default.py'. The main area is titled 'EDITING FILE "MYAPP/CONTROLLERS/DEFAULT.PY"'.

```
1 # -*- coding: utf-8 -*-
2 # this file is released under public domain and you can use without limitations
3 #
4 #####
5 # expose services controller
6 ## - index is the default action of any application
7 ## - user is required for authentication and authorization
8 ## - static files will be served at files directory (for files streaming)
9 ## - call exposes all registered services (none by default)
10 #####
11
12 def index():
13     """
14     example action using the internationalization operator T and flash
15     rendered by views/default/index.html or views/generic.html
16     """
17     response.flash = "Welcome to web2py!"
18     return dict(message=T("Hello World"))
19 
```

Below the code, there are buttons for 'back' and 'docs'. A status bar at the bottom shows 'Saved file hash: b6a6ed17e0faface75 Last saved on: Wed Nov 29 02:54:42 2011'.

それを保存し、*edit* ページに戻ってください。そして、index のリンクをクリックし、新しく作成されたページを表示してください。

次の URL を開くと、

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

myapp アプリケーションの default コントローラにある index アクションが呼び出されます。このメソッドは、ブラウザに表示される文字列を返します。下記のように表示されます：



それでは、”index” 関数を次のように編集しましょう：

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

また、edit ページから、”default/index.html” ビュー（アクションに関連付けられたビュー用のファイル）を編集し、すでに存在しているファイルの内容を、全て以下のものに置き換えてください：

```
1 <html>
2     <head></head>
3     <body>
4         <h1>{=message}</h1>
5     </body>
6 </html>
```

すると、アクションは message が定義された辞書を返すようになります。アクションが辞書を返すとき、web2py は下記の名前を持つビューを探します。

[controller]/[function].[extension]

そして、それを実行します。ここでは [extension] は、リクエストされた拡張子です。拡張子が指定されていない場合は、デフォルトは”html”で、ここではそのように想定しています。この場合、ビューは、Python のコードを特別な {{ }} タグを用いて埋め込んだ HTML ファイルとなります。この例では特に、{{=message}} の部分が、そのタグ付きのコードを、アクションから返された message の値に置き換えるように web2py に指示します。ただしここで、message は web2py のキーワードではなく、アクションで定義されたものです。ここまで、web2py のキーワードは使用されていません。

もし web2py がリクエストされたビューを見つけられなかった場合、すべてのアプリケーションで用意されている”generic.html” が使われます。

もし拡張子が ”html” 以外のもの（たとえば ”json”）が指定されて、かつ “[controller]/[function].json” というビューファイルが見つからなかつた場合、web2py は ”generic.json” というビューを探します。web2py では、generic.html、generic.json、generic.xml、generic.rss というファイルが用意されています。これらの汎用的なビューは、アプリケーションごとに個別に変更することができます。そして新しいビューも簡単に追加できます。

汎用的なビューは、開発用のツールです。リリースする製品では、全てのアクションは固有のビューを持つべきです。実のところ初期設定では、汎

用的なビューは *localhost* からのアクセス時のみ有効となっています。

次のようにビューを特定することもできます。 `response.view = 'default/something.html'`

このトピックに関する詳細は、第 10 章を読んでください。

”EDIT” ページに戻り `index` をクリックすると、次の HTML ページが表示されます：



デバッグ目的のために、次のコードを追加することができます。

```
1 { =response.toolbar() }
```

ビューの中のこのコードは、役に立つ情報を表示します。例えば、リクエスト、レスポンス、セッションに含まれるオブジェクト、そして同じタイミングで発行された全ての DB クエリなどです。

### 3.3 数えよう

今度は、同じ訪問者がこのページを何回表示したかを数えるカウンタを追加しましょう。web2py は、セッションとクッキーを使って自動的かつ透過的に訪問者を追跡します。新しい訪問者が来るたびに、セッションが作成されユニークな”`session_id`”が割り当てます。セッションはサーバ側に保存されている変数のためのコンテナです。一意の ID がクッキーを介してブラウザに送信されます。訪問者が同じアプリケーションから別のページをリクエストするとき、ブラウザはクッキーを送り戻し、そのクッキーは web2py によって取得され、対応するセッションが復元されます。

セッションを使用するには、デフォルトのコントローラを次のように変更します：

```
1 def index():
2     if not session.counter:
3         session.counter = 1
4     else:
5         session.counter += 1
```

```
6     return dict(message="Hello from MyApp", counter=session.counter)
```

counter は web2py のキーワードではなく、session に保存される変数であることに注意してください。ここでは、session の中に counter 変数が存在するかチェックするように web2py に求めます。存在しない場合は、それを作成し、1 に設定します。存在すれば、counter を 1 増加させるように web2py に求めます。最後に、ビューに counter の値を渡します。

同じ機能をコードするためのよりコンパクトな方法を以下に示します：

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

そして、ビューを変更し、counter の値を表示するための行を追加します：

```
1 <html>
2     <head></head>
3     <body>
4         <h1>{>message</h1>
5         <h2>Number of visits: {>counter</h2>
6     </body>
7 </html>
```

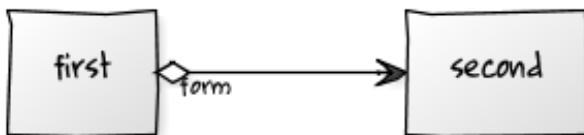
このページを再び(そして何回も)訪れると、次のような HTML のページが表示されます。



この counter は各訪問者と関連づけられ、訪問者がこのページをリロードするたびに増えています。異なる訪問者は異なるカウンタを見ることになります。

### 3.4 名前を名乗ろう

ここでは、2つのページ(firstとsecond)を作成します。firstページはフォームを作成し、訪問者の名前を尋ね、secondページへリダイレクトします。secondページは訪問者に名前で挨拶します。



デフォルトのコントローラに、対応するアクションを書きます：

```

1 def first():
2     return dict()
3
4 def second():
5     return dict()
  
```

次に、firstアクションに対する”default/first.html”ビューを作成し、以下を入力します：

```

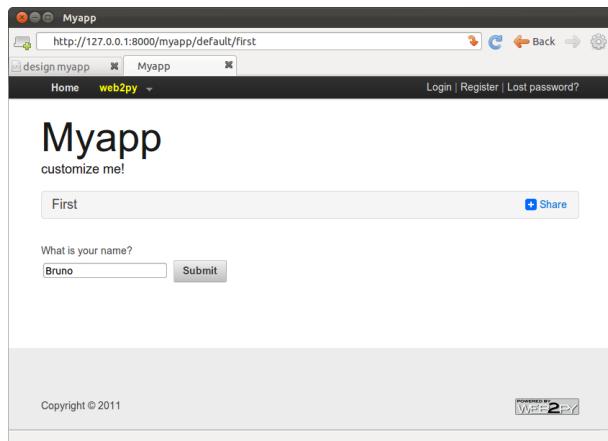
1 {{extend 'layout.html'}}
2 What is your name?
3 <form action="second">
4   <input name="visitor_name" />
5   <input type="submit" />
6 </form>
  
```

最後に、secondアクションに対する”default/second.html”ビューを作成します：

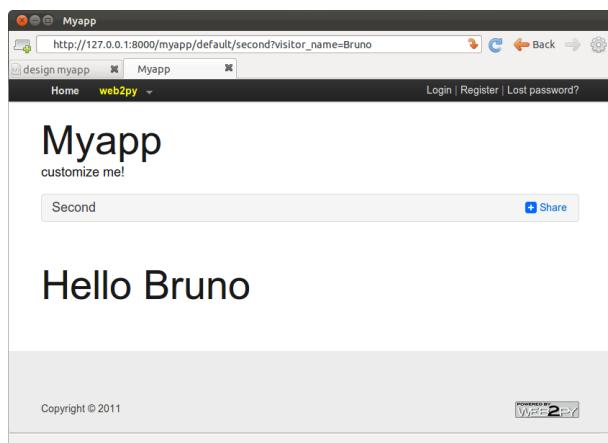
```

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

両方のビューにおいて、web2pyに用意されている基本的な”layout.html”ビューが拡張されています。このレイアウト・ビューは、2つのページのルック&フィールの一貫性を保ちます。レイアウト・ファイルは主にHTMLコードにより構成されているので、簡単に編集や置き換えができます。firstページを開いて、あなたの名前を入力してください：



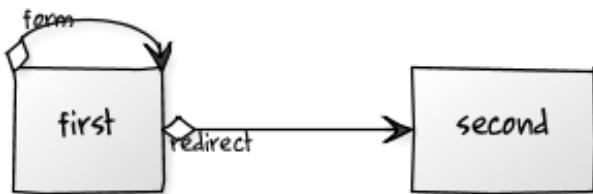
そして、フォームを送信( submit )してください。すると、挨拶が表示されます：



### 3.5 ポストバック ( Postbacks )

先に使用したフォームのサブミットに関するメカニズムはとても一般的なものです。しかし、これはあまり良いプログラミング練習ではありません。すべての入力は検証されるべきですが、上記の例では、検証の責任は second アクションに負っています。つまり、検証を行うアクションはフォームを生成したアクションと異なります。これは、コードの冗長性を引き起こしがちになります。

フォーム送信のためのより良いパターンは、フォームを生成したのと同じアクションに、今回の例では”first” にフォームを送信( submit )することです。 ”first” アクションは、変数を受け取り、処理し、サーバーサイドに保存し、訪問者を”second” ページにリダイレクトします。リダイレクト先でその変数を取得します。 このメカニズムは、ポストバック ( postback ) と呼ばれます。



デフォルトのコントローラを自己サブミット ( self-submission ) するように変更してみましょう :

```

1 def first():
2     if request.vars.visitor_name:
3         session.visitor_name = request.vars.visitor_name
4         redirect(URL('second'))
5     return dict()
6
7 def second():
8     return dict()
  
```

”default/first.html” ビューは次のように変更します :

```

1 {{extend 'layout.html'}}
2 What is your name?
3 <form>
4   <input name="visitor_name" />
5   <input type="submit" />
6 </form>
  
```

”default/second.html” ビューは、`request.vars` の代わりに `session` からデータを取得する必要があります :

```

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

訪問者から見ると、この自己サブミットは、前の実装と全く同じ挙動をしています。検証 ( validation ) はまだ加えていませんが、検証が first アクションで行われるようになることは明白です。

このアプローチはより優れています。なぜなら、訪問者の名前はセッション内に留まるようになり、明示的に渡されなくてもアプリケーションのすべてのアクションとビューからアクセスできるようになるからです。

訪問者の名前が設定される前に”second” アクションが呼び出された場合、画面上には”Hello anonymous” と表示されることに注意してください。これは、`session.visitor_name` が `None` を返すからです。もう一つの方法は、コントローラ (`second` 関数内) に次のコードを追加することです：

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

これはコントローラに認証を強制するために使用できる一般的なメカニズムです。ただし、より強力な方法のためには第 9 章を参照してください。

`web2py` ではもう一步先に進むことができ、検証を含むフォームを `web2py` に生成させることができます。`web2py` は HTML タグと同じ名前を持つヘルパー (FORM, INPUT, TEXTAREA, SELECT/OPTION) を提供します。これらを利用して、コントローラとビューのどちらにおいても、フォームを構築することができます。

例として、`first` アクションを書き換えるひとつの可能な方法を示します：

```
1 def first():
2     form = FORM(INPUT(_name='visitor_name', requires=IS_NOT_EMPTY()),
3                 INPUT(_type='submit'))
4     if form.process().accepted:
5         session.visitor_name = form.vars.visitor_name
6         redirect(URL('second'))
7     return dict(form=form)
```

ここで、`FORM` タグには 2 つの `INPUT` タグが含まれているのが分かります。`input` タグの属性は、アンダースコアで始まる名前付きの引数で指定されます。`requires` 引数はタグの属性ではありません (アンダースコアで始まってないからです)。これは `visitor_name` の値のためのバリデータ (validator) を設定します。

同じフォームを作成するもっと別の良い方法があります。

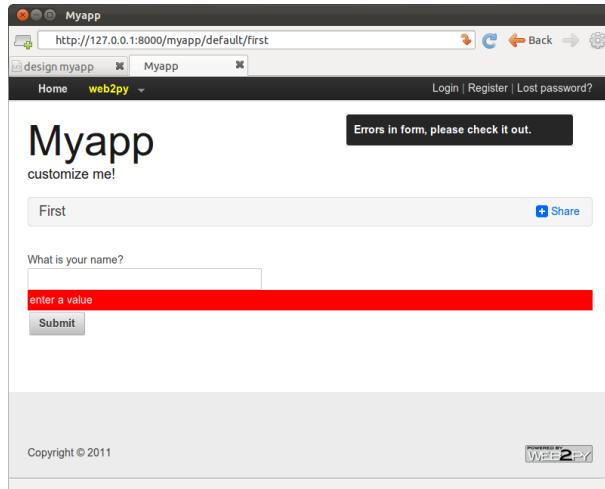
```
1 def first():
2     form = SQLFORM.factory(Field('visitor_name', requires=IS_NOT_EMPTY
3                                   ()))
4     if form.process().accepted:
5         session.visitor_name = form.vars.visitor_name
```

```
5     redirect(URL('second'))
6     return dict(form=form)
```

form オブジェクトは、”default/first.html” ビューにそれを埋め込むによって、簡単に HTML としてシリアル化することができます。

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

form.process() メソッドはバリデータを適用して自分自身のフォームに戻ってきます。form.accepts 変数は、フォームが処理されて検証を通った場合 True に設定されます。自己サブミットしたフォームが検証を通った場合、セッション内に変数が保存され、前と同じようにリダイレクトされます。フォームが検証に通らなかった場合、エラーメッセージがフォームに挿入され、次のようにユーザーに示されます：



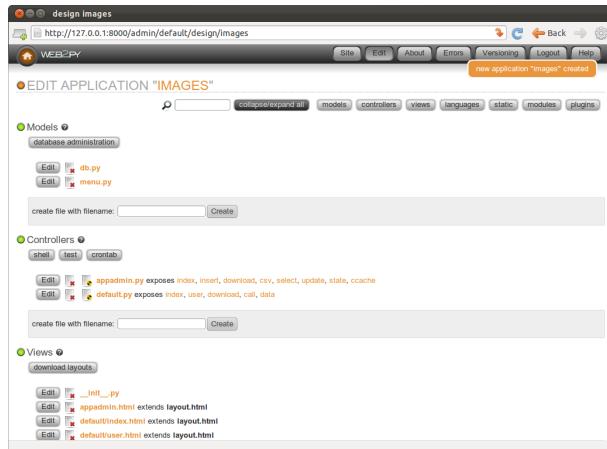
次節では、どのようにフォームがモデルから自動的に生成されるかを示します。

### 3.6 画像ブログ

ここでは、別の例として、管理者が画像を投稿して名前を付け、Web サイトの訪問者が画像を表示してコメントを送信できるような Web アプリケーションを

作成します。

前と同様に、admin にある site ページで images という名前の新しいアプリケーションを作成し、edit ページへ移ってください：



まずモデルを作成するところから始めます。モデルは、アプリケーション内の永続的なデータ（アップロードする画像、その名前、コメント）の表現です。初めに、モデルを作成/編集するためのファイルを作成します。余り深く考えず、このファイルは”db.py”とします。以下に示すコードは、db.py 内の全ての既存のコードを置き換えることを想定します。モデルとコントローラは、Python コードなので.py 拡張子を持つ必要があります。拡張子が指定されていない場合、web2py によって追加されます。ビューは.html 拡張子を代わりに持ちます。主に HTML コードで構成されるからです。

”db.py” ファイルを、対応する”edit” ボタンをクリックして編集します：



そして次のように入力してください：

```

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

```

行を一つ一つ分析してみましょう。

1行目は db と呼ばれるグローバル変数を定義します。db はデータベース接続を表します。この場合、”applications/images/databases/storage.db” ファイルに保存される SQLite データベースへの接続です。SQLite の場合は、データベースが存在しない場合は、新たに作成されます。このファイルの名前は、グローバル変数 db の名前と同じように、変更することができます。しかし、覚えやすくするために、同じ名前にしておいた方が便利です。

3~5 行目は、”image” テーブルを定義しています。define\_table は、db オブジェクトのメソッドです。最初の引数”image” は、定義したテーブルの名前です。他の引数はこのテーブルに属するフィールドです。このテーブルは、”title” というフィールド、”file” というフィールド、主キーとして機能する”id” というフィールドを持ちます (”id” は明示的に宣言されません。すべてのテーブルは id フィールドをデフォルトで持つからです)。”title” フィールドは文字列であり、”file” フィールドは upload 型です。upload は、web2py のデータ抽象化ライヤ (DAL) によって使用される特殊な型で、アップロードされたファイルの名前を保持します。web2py は、ファイルのアップロード (サイズが大きいとストリーミングを介します)、ファイルの安全なリネーム、ファイルの保存をうまく行うことができます。

テーブルが定義されるとき、web2py は以下に示すいくつかの可能なアクションのどれかひとつを取ります：

- テーブルが存在しない場合、テーブルが作成されます。
- テーブルが存在するが、その定義に対応していない場合、テーブルは定義に沿って変更されます。フィールドが異なる型を持つ場合、web2py はその内容を変更しようと試みます。
- テーブルが存在し、その定義に対応する場合、web2py は何もしません。

この挙動は、”マイグレーション( migration )”と呼ばれます。web2py ではマイグレーションは自動的に行われます。しかし `migrate=False` を `define_table` の最後の引数に渡すことによって、テーブル毎にこれを無効にすることができます。

6 行目はテーブルに対して文字列書式を定義しています。レコードが文字列としてどのように表現されるのかを決定します。`format` 引数には、レコードを受け取り文字列を返す関数を指定することもできます。次に例を示します。

```
1 format=lambda row: row.title
```

8 12 行目では、”comment” と呼ばれるテーブルを定義しています。コメントは、”author” フィールド、”email” フィールド (コメントの作者のメールアドレスを保存します)、”text” 型の”body” フィールド (その作者によって送信された実際のコメントを保存するために使用します)、`id` フィールドを介して `db.image` を指す参照型の”image\_id” フィールドを持ちます。

14 行目では、`db.image.title` は”image” テーブルの”title” フィールドであることを表します。`requires` 属性は、web2py フォームによって強制されることになる要求/制約を設定することを可能にします。ここでは、”title” は一意であることを要求します：

```
IS_NOT_IN_DB(db, db.image.title)
```

これは、`Field('title', unique=True)` の指定により自動的に設定されているので、任意指定となります。

これらの制約を表現するオブジェクトはバリデータと呼ばれます。複数のバリデータは、リストにおいてグループ化できます。バリデータは表示されている順序で実行されます。`IS_NOT_IN_DB(a, b)` は特殊なバリデータです。これは、

新規レコードに対する `b` フィールドの値が、`a` の中にすでにあっていいかをチェックします。

15 行目は、”comment” テーブルの”image\_id” フィールドが `db.image.id` に存在することを要求します。データベースに関する限り、”comment” テーブルを定義した時点で、これはすでに宣言されています。ここではさらに、明示的に、この制約が web2py によって強制されることをモデルに知らせています。この制約は、新規のコメントが送信されたとき、フォーム処理のレベルで強制されます。その結果、不正な値は入力フォームからデータベースへ伝搬しません。ここではまた、”image\_id” が対応するレコードの”title”、’%`(title)s`’ によって表現されるように要求しています。

20 行目は、`writable=False` で、”comment” テーブルの”image\_id” フィールドがフォームに表示されないように指示しています。さらに、`readable=False` で、読み取り専用フォームでも表示されないようにしています。

15 17 行目のバリデータの意味は明らかです。

なお、次のバリデータは、

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

次のように、image を表現するフォーマットを指定した場合、(自動的に) 無視されます：

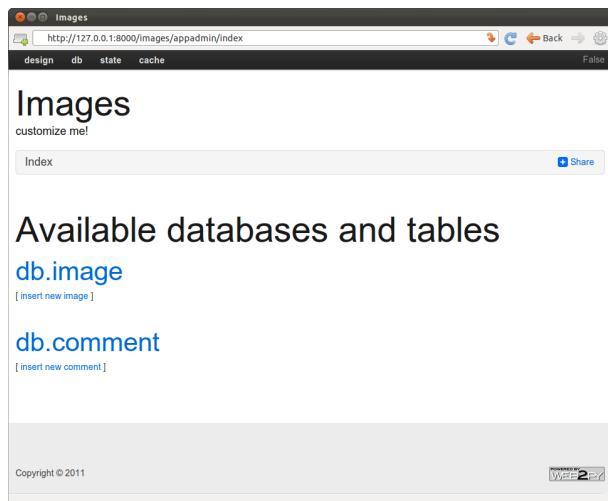
```
1 db.define_table('image', ..., format='%(title)s')
```

ここで、フォーマットは文字列、または、レコードを受け取り文字列を返す関数にすることができます。

一旦モデルが定義されると、エラーがない場合、web2py はデータベースを管理するためのアプリケーションの管理インターフェイスを作成します。このインターフェイスには、edit ページの”database administration” リンクからから、直接以下の URL からアクセスします。

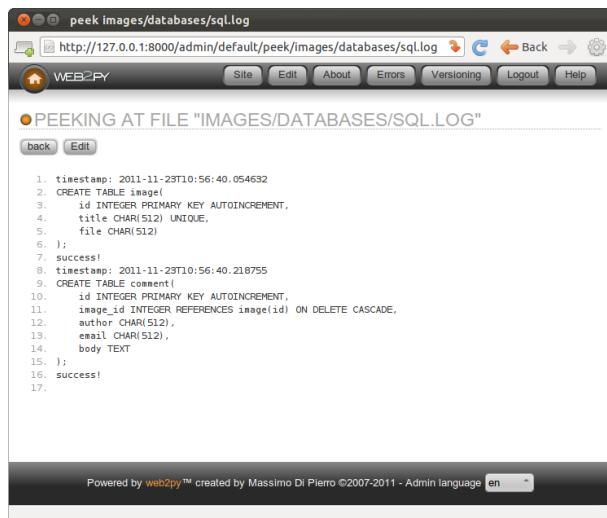
```
1 http://127.0.0.1:8000/images/appadmin
```

これは、appadmin インターフェイスのスクリーンショットです：



このインターフェイスは、”appadmin.py” というコントローラと対応する”appadmin.html” ビューにおいて実装されています。以降、このインターフェイスを単に `appadmin` と呼びます。これにより、管理者は新規のデータベースレコードを挿入し、既存のレコードを編集、削除し、テーブルを閲覧し、データベースの結合 (join) を行うことができるようになります。

`appadmin` に最初にアクセスしたときに、モデルが実行されテーブルが作成されます。web2py の DAL は、選択したデータベース・バックエンド (この例では SQLite) 固有の SQL 文に Python コードを変換します。生成された SQL は、edit ページから”models” の下にある”sql.log” リンクをクリックして、見ることができます。ただし、テーブルが作成されるまでリンクは現れません。



The screenshot shows a web browser window titled "peek images/databases/sql.log" with the URL "http://127.0.0.1:8000/admin/default/peek/images/databases/sql.log". The page displays the contents of the "sql.log" file, which contains SQL code for creating tables "image" and "comment". The log also shows timestamp entries and a success message at the end.

```

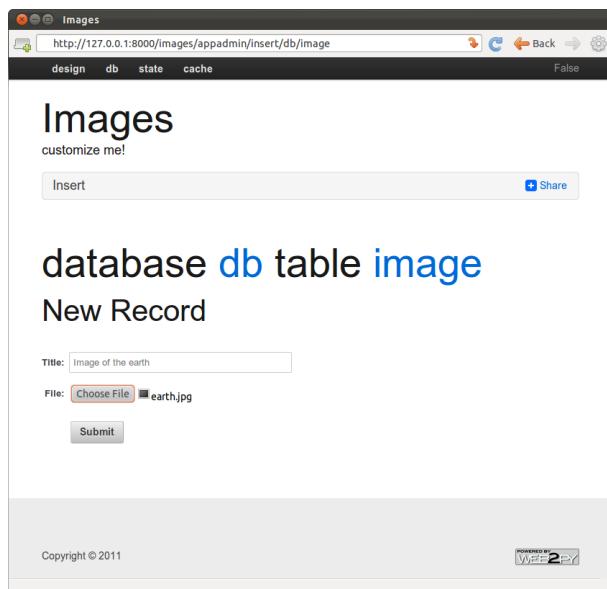
1. timestamp: 2011-11-23T10:56:40.054632
2. CREATE TABLE image(
3.     id INTEGER PRIMARY KEY AUTOINCREMENT,
4.     title CHAR(512) UNIQUE,
5.     file CHAR(512)
6. );
7. success!
8. timestamp: 2011-11-23T10:56:40.218755
9. CREATE TABLE comment(
10.    id INTEGER PRIMARY KEY AUTOINCREMENT,
11.    image_id INTEGER REFERENCES image(id) ON DELETE CASCADE,
12.    author CHAR(512),
13.    email CHAR(512),
14.    body TEXT
15. );
16. success!
17.

```

Powered by web2py™ created by Massimo Di Piero ©2007-2011 - Admin language en

モデルを編集し、再び appadmin にアクセスしようとする場合、web2py は既存のテーブル修正する SQL を生成します。生成された SQL は”sql.log” にログとして記録されます。

さて、appadmin に戻って、新しい画像レコードを挿入してみましょう：



The screenshot shows the "Images" appadmin interface with the URL "http://127.0.0.1:8000/images/appadmin/insert/db/image". The page title is "database db table image" and the sub-title is "New Record". It features an "Insert" button and a "Share" button. The form has fields for "Title" (set to "Image of the earth") and "File" (set to "Choose File" and "earth.jpg"). A "Submit" button is at the bottom.

Copyright © 2011

POWERED BY  WEB2PY

web2py は、`db.image.file` の”upload” フィールドを、ファイルをアップロードするためのフォームに変換します。フォームがサブミットされ、画像がアップロードされるとき、ファイルは、安全な方法で拡張子はそのままにリネームされ、アプリケーションの”uploads” フォルダの下に新しい名前で保存されます。新しい名前は `db.image.file` フィールドに保存されます。この処理は、ディレクトリトラバーサル攻撃を防ぐために設計されています。

なお、各フィールドの型は ウィジェット (widget) によってレンダリングされています。デフォルトの ウィジェット は オーバーライド することができます。

`appadmin` においてテーブル名をクリックすると、web2py は現在のテーブルのすべてのレコードの選択を実行します。これは、次の DAL クエリで特定されます

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

その結果は次のようにレンダリングされます。

SQL クエリを編集し [Submit] ボタンを押して、異なるレコードセットを選択することができます。

単一のレコードを編集、または、削除するには、レコードの id 番号をクリックします。

`IS_IN_DB` バリデータのおかげで、”image\_id” 参照フィールドはドロップダウンのメニューでレンダリングされます。ドロップダウンの項目はキー (`db.image.id`) として格納されますが、バリデータで指定したように、

`db.image.title` によって表現されます。

バリデータは強力なオブジェクトです。これは、どのようにフィールドを表現し、フィールドの値をフィルタし、エラーを生成し、フィールドから取り出した値をフォーマットするかを知っています。

次の図は、検証を通らないフォームをサブミットしたときに何が起こるかを示しています：

The screenshot shows a web browser window titled "Images" with the URL "http://127.0.0.1:8000/images/appadmin/insert/db/comment". The page title is "database db table comment" and the sub-section is "New Record". The form has five fields: "Image Id" (containing "value not in database"), "Author" (containing "enter a value"), "Email" (containing "enter a valid email address"), and "Body" (containing "enter a value"). Each of these fields has a red horizontal bar below it, indicating validation errors. A "Submit" button is at the bottom left.

`appadmin` によって自動生成されたものと同じフォームは、SQLFORM ヘルパーを介してプログラム的に生成し、ユーザのアプリケーションに埋め込むことができます。これらのフォームは、CSS フレンドリで、カスタマイズすることができます。

すべてのアプリケーションには `appadmin` が存在します。したがって、`appadmin` 自体、他のアプリケーションに影響を与えずに変更することができます。

ここまで、アプリケーションがデータを保存する方法、`appadmin` を介してどのようにデータベースにアクセスするかを見てきました。`appadmin` へのアクセスは管理者に対して制約されていて、アプリケーションのための本番用の web インターフェイスとして意図されたものではありません。したがって、このウォーターフォールの次のパートがあります。具体的には、次のものを作成します：

- ”index” ページ。これは、すべての利用可能な画像を title でソートして一覧表示します。そして、それらの画像に詳細ページへのリンクを張ります。
- ”show/[id]” ページ。これは、リクエストされた画像を訪問者に提示します。そして、コメントを見たり投稿したりできるようにします。
- ”download/[name]” アクション。アップロードした画像をダウンロードするために用いられます。

これはその図式です：



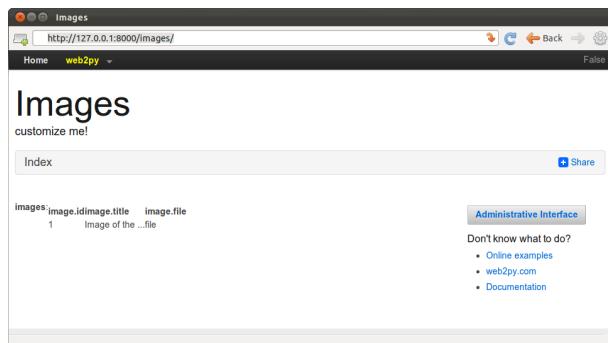
edit ページに戻り、”default.py” コントローラを編集し、その内容を次のものと入れ替えてください：

```

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

このアクションは、辞書を返します。辞書の項目のキーは、アクションに関連付けられたビューに渡される変数として解釈されます。開発中は、ビューが存在しない場合、アクションは”generic.html” ビューにより表示されます。これは、すべての web2py アプリケーションで用意されています。index アクションは、db.image.title のよってソートされた、image テーブルからのすべてのフィールド (db.image.ALL) の選択を実行します。選択の結果はレコードを格納する Rows オブジェクトです。これを、アクションによってビューへ返される images と呼ばれるローカル変数に割り当てます。images は、反復可能 (iterable) で、その要素は選択された行になります。各行において、カラムは辞書のように、つまり、images[0]['title'] のように、アクセスできます。また同様に、images[0].title のようにもアクセスできます。images[0]['title'] or equivalently as images[0].title.

ビューを記述しない場合、辞書は”views/generic.html” によってレンダリングされます。index アクションの呼び出しは次のように表示されます：



まだアクションのビューが作成されていないので、web2py はレコードをシンプルな表形式で表示しています。

では、index アクション用のビューを作成します。admin に戻り、“default/index.html” を編集して、その内容を次のように置き換えます：

```

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>
```

最初に注目する点は、ビューが特別な{{...}}タグを持つ純粋な HTML ということです。{{...}}に埋め込まれたコードは純粋な Python のコードです。ただし、インデントが無意味になるという注意があります。コードのブロックは、行末にコロン(:) がついた行で始まり、pass というキーワードで始まる行で終わります。ブロックの終わりが明らかな場合には、pass は不要です。

5~7行目は、images の行をループで回し、各行の画像に対し次のように表示します：

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

これは、image.title を含む<a href="...">...</a>タグを含んだ<li>...</li>タグになります。ハイパーテキスト参照 (href 属性) の値は次のようになります：

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

つまり、これは、”show”という関数を呼んでいる現在のリクエストと同じアプリケーションとコントローラの範囲内にある URL で、かつ、その関数に单一の引数 `args=images.id` を渡すような URL です。LI、A などは web2py のヘルパーで、対応する HTML タグをマッピングします。無名引数はシリアル化されるオブジェクトとして解釈され、そのタグの innerHTML にて挿入されます。アンダースコアで始まる名前付き引数（例えば `_href`）はそのタグの属性として解釈されます。ただし、アンダースコアは付かない属性です。たとえば、`_href` は `href` 属性、`_class` は `class` 属性、などになります。

例として、次の文は：

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

次のようにレンダリングされます：

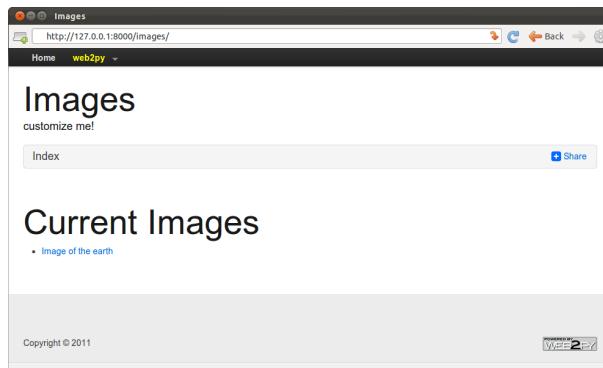
```
1 <li><a href="/images/default/show/123">something</a></li>
```

少数のヘルパ（INPUT, TEXTAREA, OPTION and SELECT）はまた、アンダースコアで始まらない特別な名前付き引数（`value` と `requires`）をサポートしています。これらはカスタムフォームを構築するために重要で、後で説明します。

`edit` ページに戻ってください。すると、”default.py exposes index” というものが示されます。”index” をクリックして、新しく作成したページを訪れることができます：

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

これは次のように表示されます：



画像名のリンクをクリックすると、次に遷移します：

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

これはエラーになります。なぜなら、”default.py” コントローラには”show” というアクションがまだ作成されていないからです。

”default.py” コントローラを編集して、その内容を次のものと置き換えてみましょう：

```
1 def index():
2     images = db().select(db.image.ALL, orderby=db.image.title)
3     return dict(images=images)
4
5 def show():
6     image = db(db.image.id==request.args(0)).select().first()
7     db.comment.image_id.default = image.id
8     form = SQLFORM(db.comment)
9     if form.process().accepted:
10         response.flash = 'your comment is posted'
11     comments = db(db.comment.image_id==image.id).select()
12     return dict(image=image, comments=comments, form=form)
13
14 def download():
15     return response.download(request, db)
```

このコントローラは、”show” と”download” の 2 つのアクションを保持しています。”show” アクションは、request.args から解析された `id` を持つ画像と、その画像に関連するすべてのコメントを選択します。そして、すべてを”default/show.html” ビューに渡します。

画像の `id` は、次のようにして：

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

”default/index.html” ビューにおいて参照され、”show” アクションにおいて：`request.args(0)` からアクセスすることができます。

”download” アクションは、`request.args(0)` にファイル名を要求します。そして、ファイルがあるはずの場所へのパスを構築し、クライアントにそのファイルを返します。ファイルが大きすぎる場合、いかなるメモリのオーバーヘッドも発生ないように、ファイルをストリーミングします。

以下の文 (statement) について注意してください：

- 7 行目は、指定したフィールドだけを用いて、`db.comment` テーブルに対するフォーム、SQLFORM を作成します。

- 8行目は、参照フィールドの値を設定します。これは入力フォームの一部ではありません。なぜなら、上で指定したフィールドのリストにないからです。
- 9行目は、現在のセッションの中で（セッションは二重送信の防止とナビゲーションの実施のために使われます）、サブミットされたフォーム（サブミットされたフォームの変数は `request.vars` にあります）を処理します。サブミットされたフォームの変数が検証を通った場合、`db.comment` テーブルに新規のコメントが挿入されます。そうでない場合、フォームはエラーメッセージを含むように修正されます（たとえば、作者のメールアドレスが不適当である場合）。これは、すべての9行目で行われます！
- 10行目は、フォームが受理された場合にのみ、データベース・テーブルにレコードが挿入された後に実行されます。`response.flash` は、web2py の変数で、ビューにおいて表示され、訪問者に何が行われたのかを通知するために使われます。
- 11行目は、現在の画像に関するすべてのコメントを選択します。

（訳注：この節は原文が次の第3版の古い *statement* に対応している。あえて第3版のコードをここに併記しておく。）

```

7.    form = SQLFORM(db.comment)
8.    form.vars.image_id = image.id
9.    if form.accepts(request.vars, session):
10.      response.flash = 'your comment is posted'
11.      comments = db(db.comment.image_id==image.id).select()

```

”download”アクションは、離形アプリケーションの ”default.py” コントローラにおいてすでに定義されています。

”download” アクションは辞書を返さないので、ビューは必要ありません。一方、”show” アクションはビューを持つべきです。したがって、admin に戻って”default/show.html” という新規のビューを作成してください。

この新規のファイルを編集し、その内容を次のものと置き換えてください：

```

1 {{extend 'layout.html'}}
2 <h1>Image: {{=image.title}}</h1>
3 <center>
4 
6 </center>
7 {{if len(comments):}}
8   <h2>Comments</h2><br /><p>
9   {{for comment in comments:}}
10    <p>{{=comment.author}} says <i>{{=comment.body}}</i></p>

```

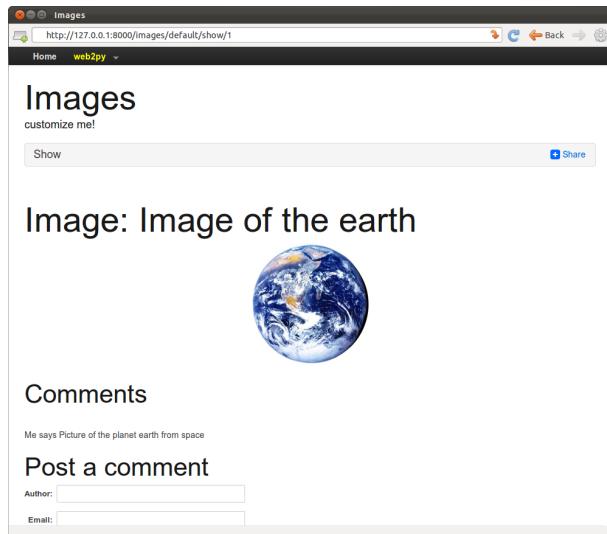
```

11 {{pass}}</p>
12 {{else:}}
13 <h2>No comments posted yet</h2>
14 {{pass}}
15 <h2>Post a comment</h2>
16 {{=form}}

```

このビューは、`<img ... />`タグ内において、”download”アクションを呼び出すことによって `image.file` を表示します。コメントがある場合は、それらに対してループを回し、一つ一つ表示します。

どのようにすべてが表示されるかを以下に示します：



訪問者がこのページでコメントを送信すると、コメントがデータベースに保存され、ページの下部に追加されます。

### 3.7 CRUDを追加しよう

web2py はまた、フォームをさらに単純化する CRUD(Create/Read/Update/Delete)API を提供します。CRUD を使用するには、”db.py” ファイルなど、どこかにそれを定義する必要があります：

```
1 from gluon.tools import Crud
```

```
2 crud = Crud(db)
```

これら 2 つの行は、すでに雛形アプリケーションに含まれています。

crud オブジェクトは高レベルのメソッドを提供します。例えば：

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

これは、次のプログラミング・パターンを置き換えるために使用することができます：

```
1 form = SQLFORM(table)
2 if form.process().accepted:
3     session.flash = '...'
4     redirect('...')
```

前回の”show” アクションを、crud を用いてより改良して書き換えてみます：

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

初めに、次のような構文を用いていることに注意してください

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

これにより、要求されたレコードが取り出されます。table(id) は、レコードが見つからない場合に None を返します。したがって、or redirect(...) を同じ行に書くようにします。

crud.create の next 引数は、フォームが受理された後にリダイレクトする先の URL です。message 引数は受理されたときに表示されるものです。CRUD については第 7 章でより詳しく説明します。

### 3.8 認証を追加しよう

web2py のロールベースのアクセス制御 API は非常に洗練されています。ただし、ここでは show アクションへのアクセスを認証されたユーザのみに限定する説明だけにし、残りの詳しい説明は第 9 章に委ねます。

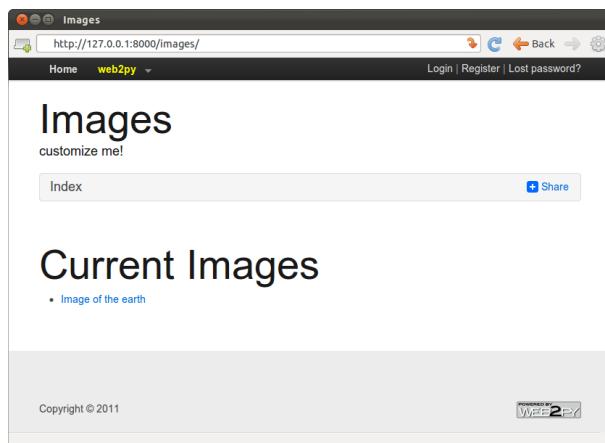
認証されたユーザにのみアクセスを限定するには、3つのステップを完了する必要があります。モデル（たとえば”db.py”）では、以下を追加する必要があります：

```
1 from gluon.tools import Auth
2 auth = Auth(db)
3 auth.define_tables()
```

コントローラでは、次の1つのアクションを追加する必要があります：

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

こうするだけで、ログイン、登録、ログアウトなどが可能になります。デフォルトのレイアウトはまた、対応するページへのオプションを右上隅に表示します。



こうすることで、制限したい関数を次のようにデコレート（decorate）することができます：

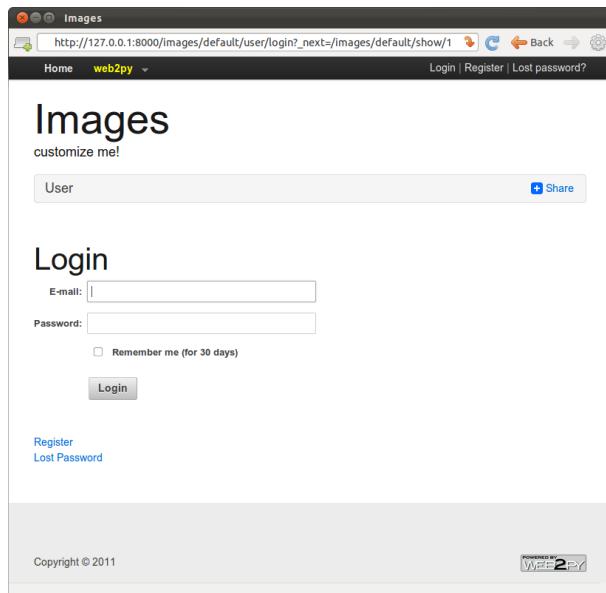
```
1 @auth.requires_login()
2 def show():
3     image = db.image(request.args(0)) or redirect(URL('index'))
4     db.comment.image_id.default = image.id
5     form = crud.create(db.comment, next=URL(args=image.id),
6                         message='your comment is posted')
7     comments = db(db.comment.image_id==image.id).select()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

次のアドレスへのいかなるアクセスも

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

ログインが要求されます。ユーザーがログインしていない場合は、次のアドレスにリダイレクトされます。

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



user 関数はまた、とりわけ次のようなアクションも公開します：

```
1 http://127.0.0.1:8000/images/default/user/logout
2 http://127.0.0.1:8000/images/default/user/register
3 http://127.0.0.1:8000/images/default/user/profile
4 http://127.0.0.1:8000/images/default/user/change_password
5 http://127.0.0.1:8000/images/default/user/request_reset_password
6 http://127.0.0.1:8000/images/default/user/retrieve_username
7 http://127.0.0.1:8000/images/default/user/retrieve_password
8 http://127.0.0.1:8000/images/default/user/verify_email
9 http://127.0.0.1:8000/images/default/user/impersonate
10 http://127.0.0.1:8000/images/default/user/notAuthorized
```

このとき、ログインして、コメントを読み、投稿するために、ユーザーは最初に登録する必要があります。

auth オブジェクトと user 関数の両方は雑形アプリケーションすでに定

義されています。`auth` オブジェクトは、高度にカスタマイズ可能で、`email`による照合、登録の承認、`CAPTHCA`、プラグインを介したログインメソッドの変更を行うことができます。

### 3.8.1 グリッドの追加

管理インターフェイスを作成するための `SQLFORM.grid` と `SQLFORM.smartgrid` の二つのガジェットは、作成したアプリケーションをさらに向上させます。

```
1 @auth.requires_membership('manager')
2 def manage():
3     grid = SQLFORM.smartgrid(db.image)
4     return dict(grid=grid)
```

関連する”views/default/manage.html”です。

```
1 {{extend 'layout.html'}}
2 <h2>Management Interface</h2>
3 {{=grid}}
```

`appadmin` を使用して”manager” グループを作成し、さらにそのグループにいくつかのユーザーメンバーを作成します。このメンバーは以下の URL にアクセスできます。( 訳注：この行の原文にある `not` は誤りと解釈して訳しています。)

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

ブラウジングや検索もできます。

The screenshot shows a web browser window titled "Images" at the URL <http://127.0.0.1:8000/images/default/manage/image>. The page header includes "Home", "web2py", "Welcome Bruno Logout | Profile | Password". Below the header, the title "Images" is displayed with the subtitle "customize me!". A "Manage" button and a "Share" button are visible. A search bar contains the query "image.title contains 'earth'" with "Search" and "Clear" buttons. Below the search bar, there is a search interface with dropdowns for "Title" and "contains", and buttons for "And" and "Or". A message "1 records found" is shown. A table lists one record: "Id: 1 Title: Image of the earth File: [Comments] View Edit Delete". At the bottom, a copyright notice "Copyright © 2011" and the "POWERED BY WEB2PY" logo are present.

画像の追加、更新、削除、そして画像へのコメントが行なえます。

The screenshot shows a web browser window titled "Images" at the URL <http://127.0.0.1:8000/images/default/manage/image/1?signature=61c4ad4749cabd0f258e>. The page header includes "Home", "web2py", "Welcome Bruno Logout | Profile | Password". Below the header, the title "Images" is displayed with the subtitle "customize me!". A "Manage" button and a "Share" button are visible. A navigation bar includes "Back", "View", and "Comments" buttons. The main form displays the details for an image with Id: 1, Title: "Image of the earth", and a file input field showing "(None)". A preview image of the Earth is displayed below the file input. A "Choose File" button and "[file] [delete]" buttons are present. A checkbox labeled "Check to delete:" is followed by a checkbox input. A "Submit" button is at the bottom. The "View" and "Comments" buttons in the navigation bar are highlighted.

### 3.9 レイアウトの設定

”views/layout.html” を編集してデフォルトのレイアウトを設定することができます。しかし、HTML を編集しなくても設定することができます。実際、”static/base.css” のスタイルシートは、よく文章化されていて、第 5 章において説明されています。HTML を編集せずに、色、カラム数、サイズ、枠線、背景を変更することができます。メニュー、タイトル、サブタイトルを編集したい場合は、任意のモデルファイルでそれを行うことができます。離形のアプリは、”models/menu.py” ファイルにおいて、以下のパラメータのデフォルト値を設定します：

```

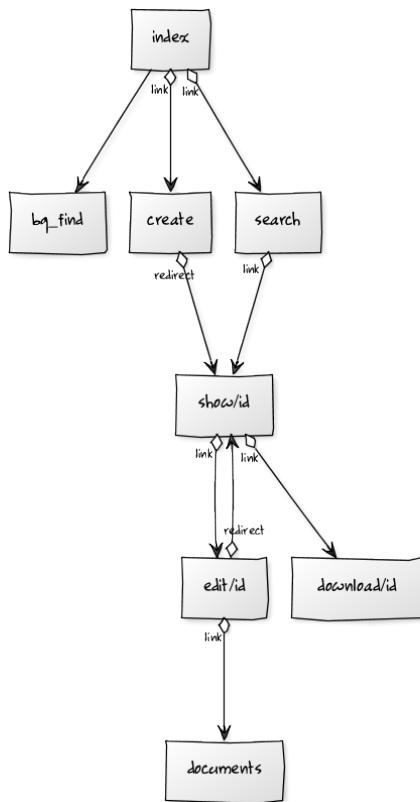
1 response.title = request.application
2 response.subtitle = T('customize me!')
3 response.meta.author = 'you'
4 response.meta.description = 'describe your app'
5 response.meta.keywords = 'bla bla bla'
6 response.menu = [ [ 'Index', False, URL('index') ] ]

```

### 3.10 Wiki

このセクションでは、wiki をスクラッチから構築します。第 13 章で説明する plugin\_wiki によって提供される拡張機能は使用しません。訪問者は、ページの作成、(タイトルによる)検索、編集を行うことができます。訪問者は、コメントを(前回のアプリケーションと全く同様に)投稿することができます、また、(ページへ添付する形で)文章も投稿することができます、ページからその文章にリンクを張ることができます。慣例として、ここでは Wiki 構文のために Markmin 構文を採用します。ここではまた、Ajax を用いた検索ページ、そのページに対する RSS フィード、XML-RPC [46] を介したページ検索用のハンドラ、を実装します。

次の略図は、実装が必要なアクションと、それらの間で構築すべきリンクを列挙しています。



”mywiki” という名の新規の雛形アプリを作成して始めましょう。

モデルはページ (page)、コメント (comment)、文章 (document) という 3 つのテーブルを持つ必要があります。comment と document の両者は page を参照します。それらは page に属しているからです。document は、前回の画像アプリケーションのように、upload 型のファイル・フィールドを持ちます。

以下にすべてのモデルを示します：

```

1 db = DAL('sqlite://storage.sqlite')
2
3 from gluon.tools import *
4 auth = Auth(db)
5 auth.define_tables()
6 crud = Crud(db)
7
8 db.define_table('page',
  
```

```

9   Field('title'),
10  Field('body', 'text'),
11  Field('created_on', 'datetime', default=request.now),
12  Field('created_by', db.auth_user, default=auth.user_id),
13  format='%(title)s')
14
15 db.define_table('comment',
16   Field('page_id', db.page),
17   Field('body', 'text'),
18   Field('created_on', 'datetime', default=request.now),
19   Field('created_by', db.auth_user, default=auth.user_id))
20
21 db.define_table('document',
22   Field('page_id', db.page),
23   Field('name'),
24   Field('file', 'upload'),
25   Field('created_on', 'datetime', default=request.now),
26   Field('created_by', db.auth_user, default=auth.user_id),
27   format='%(name)s')
28
29 db.page.title.requires = IS_NOT_IN_DB(db, 'page.title')
30 db.page.body.requires = IS_NOT_EMPTY()
31 db.page.created_by.readable = db.page.created_by.writable = False
32 db.page.created_on.readable = db.page.created_on.writable = False
33
34 db.comment.body.requires = IS_NOT_EMPTY()
35 db.comment.page_id.readable = db.comment.page_id.writable = False
36 db.comment.created_by.readable = db.comment.created_by.writable = False
37 db.comment.created_on.readable = db.comment.created_on.writable = False
38
39 db.document.name.requires = IS_NOT_IN_DB(db, 'document.name')
40 db.document.page_id.readable = db.document.page_id.writable = False
41 db.document.created_by.readable = db.document.created_by.writable =
        False
42 db.document.created_on.readable = db.document.created_on.writable =
        False

```

”default.py” コントローラを編集し、以下のアクションを作成してください：

- index: すべての wiki ページを列挙する
- create: 別の wiki ページを投稿する
- show: wiki ページとそのコメントを表示し、コメントを追加する
- edit: 既存のページを編集する
- documents: ページに添付された文書を管理する

- download: (訳注: この章で説明されたサンプル・アプリ) の例のように 文章をダウンロードする
- search: 検索用のポックスを表示し、Ajax コールバックを介して、訪問者が 入力したタイトルに該当するもの全てを返す
- callback: Ajax 用のコールバック関数。訪問者の入力に合わせて、検索ペー ジに埋め込まれる HTML を返す

これは”default.py” コントローラです：

```

1 def index():
2     """ this controller returns a dictionary rendered by the view
3         it lists all wiki pages
4     >>> index().has_key('pages')
5         True
6     """
7     pages = db().select(db.page.id, db.page.title, orderby=db.page.title
8                         )
9     return dict(pages=pages)
10
11 @auth.requires_login()
12 def create():
13     "creates a new empty wiki page"
14     form = crud.create(db.page, next=URL('index'))
15     return dict(form=form)
16
17 def show():
18     "shows a wiki page"
19     this_page = db.page(request.args(0)) or redirect(URL('index'))
20     db.comment.page_id.default = this_page.id
21     form = crud.create(db.comment) if auth.user else None
22     pagecomments = db(db.comment.page_id==this_page.id).select()
23     return dict(page=this_page, comments=pagecomments, form=form)
24
25 @auth.requires_login()
26 def edit():
27     "edit an existing wiki page"
28     this_page = db.page(request.args(0)) or redirect(URL('index'))
29     form = crud.update(db.page, this_page,
30                         next=URL('show', args=request.args))
31     return dict(form=form)
32
33 @auth.requires_login()
34 def documents():
35     "browser, edit all documents attached to a certain page"
36     page = db.page(request.args(0)) or redirect(URL('index'))
37     db.document.page_id.default = page.id

```

```

37     db.document.page_id.writable = False
38     grid = SQLFORM.grid(db.document.page_id==page.id, args=[page.id])
39     return dict(page=page, grid=grid)
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', _name='keyword',
51         _onkeyup="ajax('callback', ['keyword'], 'target');")),
52         target_div=DIV(_id='target'))
53
54 def callback():
55     "an ajax callback that returns a <ul> of links to wiki pages"
56     query = db.page.title.contains(request.vars.keyword)
57     pages = db(query).select(orderby=db.page.title)
58     links = [A(p.title, _href=URL('show', args=p.id)) for p in pages]
59     return UL(*links)

```

2、6行目は、index アクションのコメントを提供します。コメント内にある4、5行目は、テストコード (doctest) として python によって解釈されます。テストは管理インターフェイスから実行できます。この場合、テストは index アクションがエラーなしで実行されることを検証します。

18、27、35 行目は、request.args(0) の id を持つ page レコードを取り出そうと試みます。

13、20 行目は、それぞれ新規ページ、新規コメントのための作成フォームを定義し処理します。

28 行目は、wiki ページのための更新フォームを定義し処理します。

38 行目は、ブラウザ上でこのページにリンクしているコメントの追加と更新を行なうことを許可する grid の作成を行なっています。

51 行目はいくつかの魔法が起こっています。”keyword” という INPUT タグの onkeyup 属性が設定されます。訪問者がキーを放すたびに、onkeyup 属性内における JavaScript コードが、クライアント側で、実行されます。その JavaScript コードは次の通りです：

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

`ajax` は、”web2py.js” ファイルに定義された JavaScript 関数です。このファイルはデフォルトの”layout.html” によって組み込まれます。これは 3 つのパラメタをとります：同期的コールバックを実行するアクションの URL、コールバックに送る変数の ID リスト ([”keyword”])、そして、レスポンスが挿入される場所の ID(”target”) です。

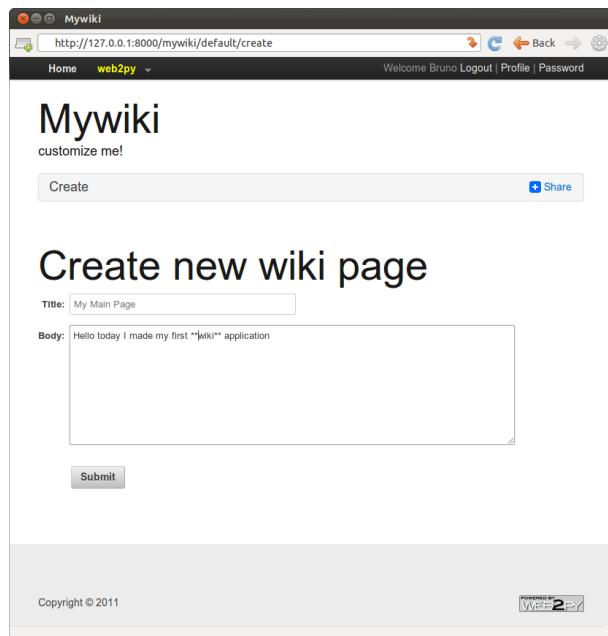
検索ボックスに何かを打ち込みキーを放すとすぐに、クライアントはサーバーを呼び出し、’keyword’ フィールドの内容を送信します。そして、サーバーが応答したら、そのレスポンスは’target’ タグの innerHTML としてページ自身に埋め込まれます。

’target’ タグは 52 行目で定義される DIV です。これはビューにおいても定義することができます。

これは”default/create.html” ビューのコードです：

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

create ページを訪れるとき、次のように表示されます：



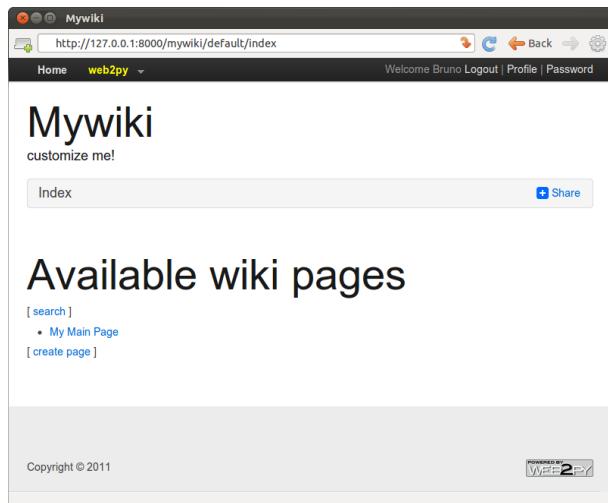
”default/index.html” ビューのコードです：

```

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

```

これは、次のページを生成します：



”default/show.html” ビューのコードです :

```

1 {{extend 'layout.html'}}
2 <h1>{{=page.title}}</h1>
3 [ {{=A('edit', _href=URL('edit', args=request.args))}}
4 | {{=A('documents', _href=URL('documents', args=request.args))}} ]<br
5 />
6 {{=MARKMIN(page.body)}}
7 <h2>Comments</h2>
8 {{for comment in comments:}}
9   <p>{{=db.auth_user[comment.created_by].first_name}} on {{=comment.
10    created_on}}
11   says <i>{{=comment.body}}</i></p>
12 {{/pass}}
13 <h2>Post a comment</h2>
14 {{=form}}

```

markimin 構文の代わりに markdown 構文を使用する場合、次のようにします :

```
1 from gluon.contrib.markdown import WIKI
```

そして、MARKMIN ヘルパの代わりに WIKI を使用してください。また、markmin の構文の代わりに生の HTML を受け入れるように選択することができます。この場合、次のものを :

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

次のものに置き換えます :

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

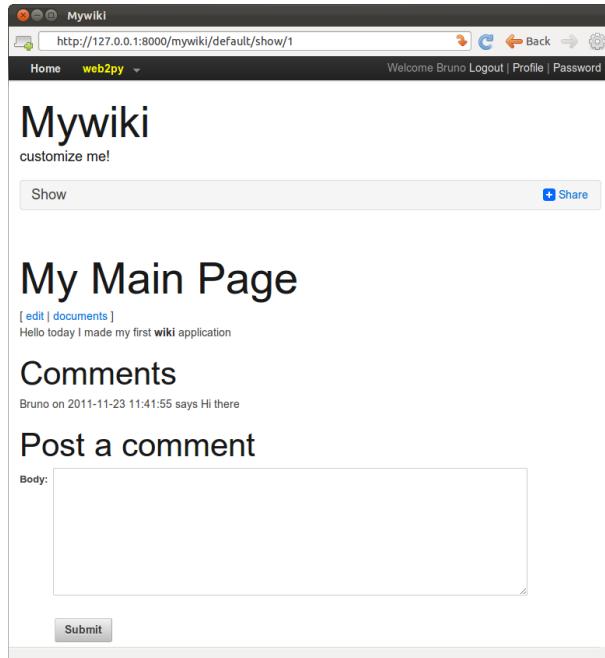
(デフォルトの web2py の挙動により、XML ではエスケープされません)

これは次のようにする方がより良いです：

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

`sanitize=True` と設定すると、”<script>” タグのような安全でない XML タグをエスケープするようにし、XSS の脆弱性を防ぎます。

これで、index ページからページタイトルをクリックすると、作成したページを見ることができます：



”default/edit.html” ビューのコードです：

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

これは、作成ページとほぼ同じページを生成します。

次は”default/documents.html” ビューのコードです：

```

1 {{extend 'layout.html'}}
2 <h1>Documents for page: {{=page.title}}</h1>
3 [ {{=A('show', _href=URL('show', args=request.args))}} ]<br />
4 <h2>Documents</h2>
5 {{=grid}}

```

”show” ページで documents ( 訳注：ページ上部にあるリンクのこと ) をクリックすると、ページに添付された文書を管理することができます。

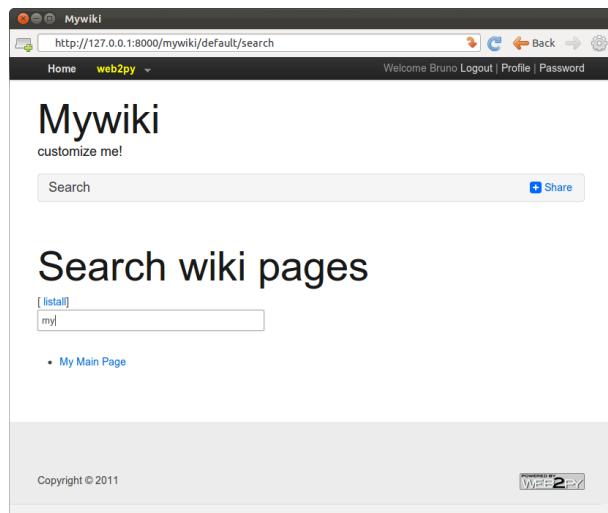
最後は”default/search.html” ビューのコードです：

```

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

```

これは、次のような Ajax 検索フォームを生成します：



たとえば次の URL に訪れることで、コールバックのアクションを直接呼び出すことも可能です：

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

このページのソースコードを見ると、コールバックによって返された次のような HTML を見ることができます：

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

web2py を用いて保存されたページから RSS フィードを生成することは簡単です。web2py には `gluon.contrib.rss2` があるからです。単に、次のアクションを default コントローラに追加してください：

```
1 def news():
2     "generates rss feed form the wiki pages"
3     response.generic_patterns = ['.rss']
4     pages = db().select(db.page.ALL, orderby=db.page.title)
5     return dict(
6         title = 'mywiki rss feed',
7         link = 'http://127.0.0.1:8000/mywiki/default/index',
8         description = 'mywiki news',
9         created_on = request.now,
10        items = [
11            dict(title = row.title,
12                 link = URL('show', args=row.id),
13                 description = MARKMIN(row.body).xml()),
```

```

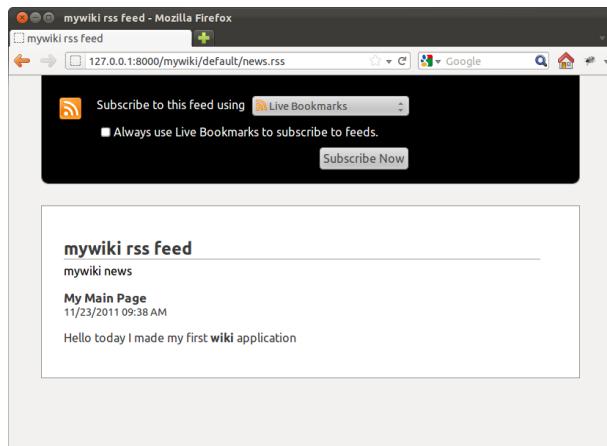
14     created_on = row.created_on
15 ) for row in pages])

```

そして、次のページに訪れると

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

フィードが表示されます(フィードリーダによって見た目は異なります)。なお、URL の拡張子が.rss なので、dict は自動的に RSS に変換されています。



web2py はまた、サードパーティのフィードを読むためのフィードパーサ(フィード解析)も含んでいます。

最後に、プログラム的に wiki を検索可能にする XML-RPC ハンドラを追加しましょう：

```

1 service = Service()
2
3 @service.xmlrpc
4 def find_by(keyword):
5     "finds pages that contain keyword for XML-RPC"
6     return db(db.page.title.contains(keyword)).select().as_list()
7
8 def call():
9     "exposes all registered services, including XML-RPC"
10    return service()

```

ハンドラのアクションは、このリストで指定された関数を(XML-RPC を介して)単純に公開しています。ここでは、find\_by です。find\_by はアクションではあ

りません(引数があるためです)。この関数は、`.select()` でデータベースに問い合わせ、`.response` でレコードをリストとして取り出し、そのリストを返します。

ここに、どのように外部の Python プラグラムから XML-RPC ハンドラにアクセスするかの例を示します。

```

1 >>> import xmlrpclib
2 >>> server = xmlrpclib.ServerProxy(
3     'http://127.0.0.1:8000/mywiki/default/call/xmlrpc')
4 >>> for item in server.find_by('wiki'):
5     print item['created_on'], item['title']

```

ハンドラは、XML-RPC を理解する多くのプログラミング言語(C、C++、C#、Java など)からアクセスすることができます。

### 3.10.1 date、datetime そして time の書式

`date`、`datetime` そして `time` のフィールドの型のそれに、三つの異なる表現があります。

- データベースでの表現
- web2py 内部での表現
- フォームやテーブルで使われる文字列表現

データベースの表現は、内部の問題であり、コードには影響しません。web2py の内部では、それぞれ `datetime.date`、`datetime.datetime` そして `datetime.time` というオブジェクトの表現で保存されています。そして次のように操作することができます。

```

1 for page in db(db.page).select():
2     print page.title, page.day, page.month, page.year

```

フォームの中で日付が文字列に変換される場合、次の ISO の表現に従って変換されます。

```
1 %Y-%m-%d %H:%M:%S
```

この表現は国際化が行なわれていますが、この表現を別のものに変更するために admin の翻訳ページを使用することもできます。

```
1 %m/%b/%Y %H:%M:%S
```

デフォルトでは、*web2py* はアプリケーションが既に英語で書かれていると想定しているため、英語は翻訳されません。英語用の国際化の機能を有効にしたい場合には、( *admin* を使用して ) 翻訳ファイルを作成する必要があります。そして、アプリケーションが現在の言語が英語以外であることを、次のように宣言する必要があります。

```
1 T.current_languages = ['null']
```

### 3.11 admin の追加情報

管理インターフェイスはさらなる機能を提供します。ここではそれを簡単に見ていきます。

#### 3.11.1 サイト

このページはインストールされたすべてのアプリケーションを列挙します。下部には 2 つのフォームがあります。

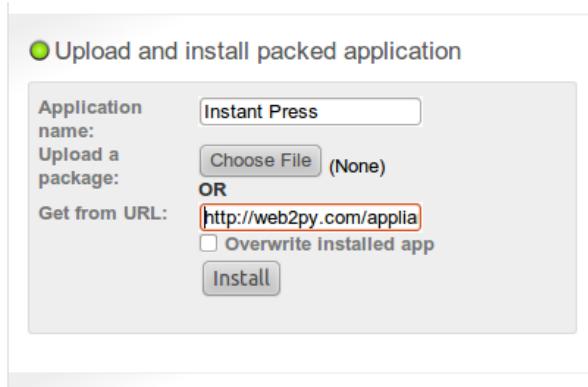
1 つ目のフォームでは、名前を指定して新しいアプリケーションを作成することができます。

2 つ目のフォームでは、ローカルファイルまたはリモート URL から、既存のアプリケーションをアップロードすることができます。アプリケーションをアップロードする場合は、その名前を指定する必要があります。元の名前と同じにすることができますが、同じである必要はありません。これにより、同じアプリケーションの複数のコピーをインストールすることができます。たとえば、Martin Mulone によって作られた Instant Press CMS を次の URL からアップロードすることができます :

```
1 http://code.google.com/p/instant-press/
```

*web2py* のファイルは .w2p ファイルにパッケージングされます。これは *gzip* 圧縮されたファイルを *tar* でまとめたです。*web2py* では、ブラウザでダウンロード時にファイルが解凍されるのを防ぐため、拡張子 .tgz の代わりに

拡張子.w2p を使います。これらは、`tar zxvf [ファイル名]` として、手動で解凍することができます。ただし、その必要は全くありません。



アップロードに成功すると、web2py はアップロードしたファイルの MD5 チェックサムを表示します。これにより、ファイルがアップロード中に破損していないことを確認することができます。InstantPress の名前がインストールされたアプリケーションのリストに表示されます。admin にある InstantPress の名前をクリックすると、起動して稼働します。

Instant Press に関するより詳しい情報は、以下の URL で読むことができます。

<sup>1</sup> <http://code.google.com/p/instant-press/>

各アプリケーションにおいて、*site* ページは次のことを可能にします：

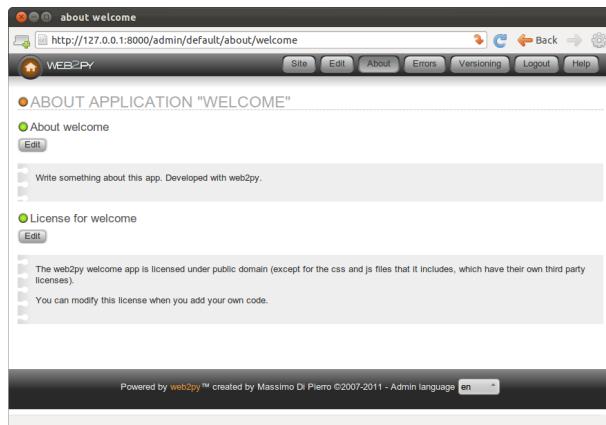
- アプリケーションのアンインストール

- *about* ページ (下で説明します) への遷移
- *edit* ページ (下で説明します) への遷移
- *errors* ページ (下で説明します) への遷移
- 一時ファイル (セッション、エラー、cache.disk ファイル) のクリーンアップ。
- すべてのパック。これは、アプリケーションの完全なコピーを保持する tar ファイルを返します。アプリケーションをパックする前に、一時ファイルをクリーンアップするべきです。
- アプリケーションのコンパイル。エラーがない場合は、このオプションはすべてのコントローラ、ビュー、モデルをバイトコードとしてコンパイルします。ビューは木のように他のビューの中で拡張や組み込みができるため、バイトコンパイルする前に、すべてのコントローラに対するビューの木構造は 1 つのファイルに折りたたまれます。この実質的な効果としてバイトコンパイルされたアプリケーションは、テンプレートの解析や文字列の置換が実行時に行われなくなるため、より高速に動作します。
- コンパイル結果のパック。このオプションは、バイトコードでコンパイルしたアプリケーションにのみ提示されます。これは、クローズドソースとしてソースコードを含まない状態でアプリケーションをパックできるようにします。ただし、Python は (他のプログラミング言語と同様に) 技術的には逆コンパイルすることができます。そのためコンパイル結果では、ソースコードの完全な保護は行えません。しかし、逆コンパイルは難しく、違法行為です。
- コンパイル結果の削除。これは単に、アプリケーションからバイトコードでコンパイルされた、モデル、ビュー、コントローラを削除します。アプリケーションがソースコードとともにパックされているか、ローカルで動くようになっている場合、バイトコードでコンパイルされたファイルを削除することはなんら弊害はなく、アプリケーションは動作し続けます。アプリケーションがコンパイル結果をパックしたファイルからインストールされた場合は、安全ではありません。元に戻すためのソースコードがないため、アプリケーションはもはや動作しなくなります。

*web2py* の *admin* サイトのページで利用できるすべての機能はまた、*gluon/admin.py* モジュールで定義されている API を介してプログラム的にアクセスできます。単に *Python* のシェルを開いて、このモジュールをインポートしてください。

### 3.11.2 about

`about` タブから、アプリケーションの説明とライセンス条項を編集することができます。これらは、それぞれに、`application` フォルダにある `ABOUT` と `LICENCE` ファイルに書き込まれます。



これらのファイルに対して `MARKMIN` か `gluon.contrib.markdown.WIKI` の構文を使用することができます。これは参照に説明されているとあります [29]。

### 3.11.3 edit

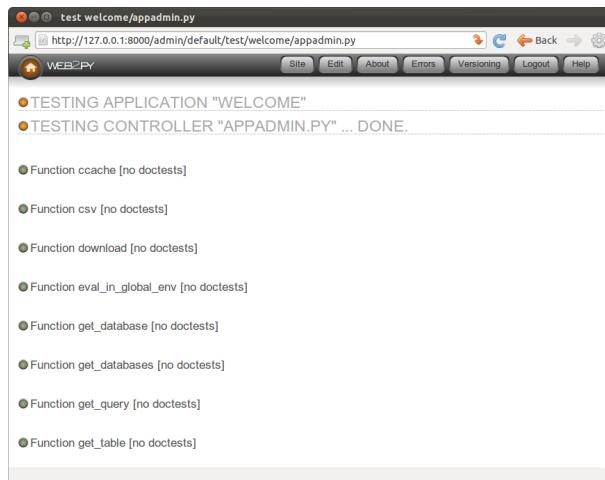
本章では、すでに `edit` ページを使用しています。ここでは、`edit` ページのさらなる数個の機能に注目します。

- 任意のファイル名をクリックすると、構文が強調表示された状態でファイルの内容を見ることができます。
- `edit` をクリックすると、Web インタフェースを介してファイルを編集することができます。
- `delete` をクリックすると、ファイルを(完全に)削除することができます。
- `test` をクリックすると、web2py はテストを走らせます。テストは Python の `doctest` を使って書かれます。そして、各関数は自分自身のテストを持ちます。
- Web インタフェースを介して、言語ファイルを追加し、全ての文字列を見

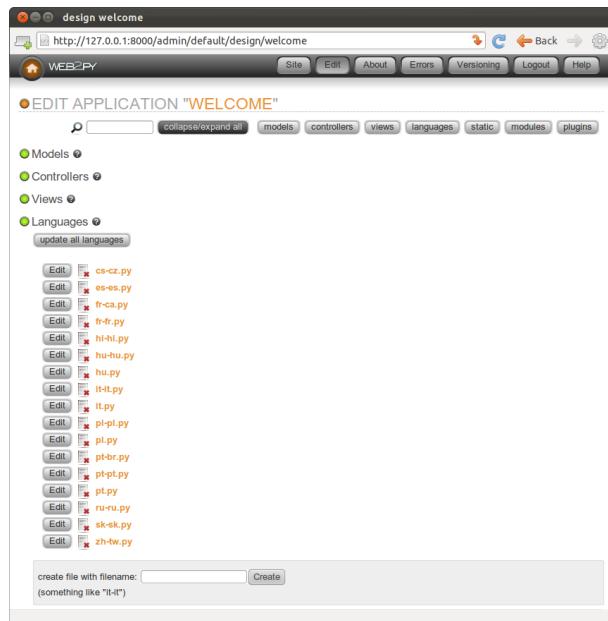
いだすためにアプリをスキャンし、翻訳文字列を編集することができます。

- 静的ファイルがフォルダとサブフォルダで構成されている場合、フォルダ階層はフォルダ名をクリックして切り替えることができます。

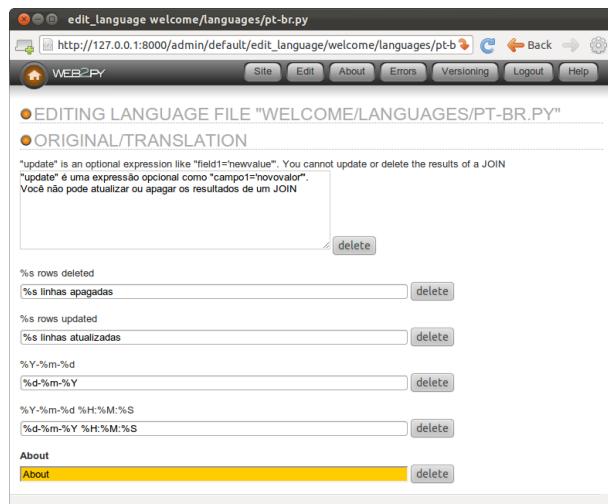
以下の画像は welcome アプリケーションのテストページの出力を示しています。



以下の画像は welcome アプリケーションの languages タブを示しています。

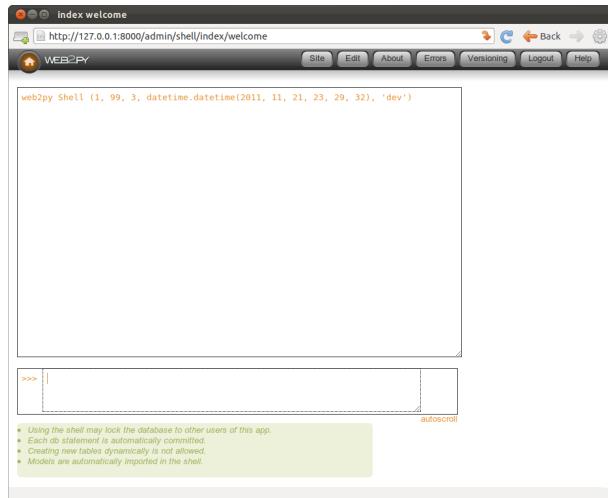


下の画像は、どのように言語ファイルを編集するかを示しています。この場合、welcome アプリケーションに対する "it"(イタリア) 語のファイルです。



*shell*

*edit* のコントローラタブの下にある”shell”リンクをクリックすると、web2py は Web ベースの Python のシェルを開き、現在のアプリケーションに対するモデルを実行します。これにより対話的にアプリケーションを実行することができます。



#### crontab

*edit* のコントローラタブの下にはまた、”crontab”のリンクがあります。このリンクをクリックすると、web2py の crontab ファイルを編集することができます。これは unix の crontab と同じ構文に従いますが、unix には依存していません。実際、この機能は web2py だけを必要とし、Windows 上でも動作します。これにより、スケジュールされた時間にバックグラウンドで起動する必要のあるアクションを登録することができます。詳細は、次章を参照してください。

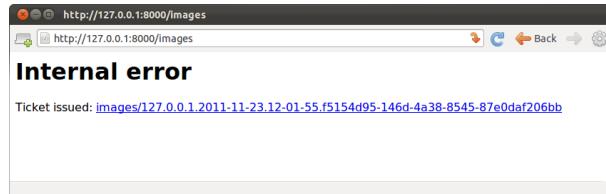
#### 3.11.4 errors

web2py をプログラムするときは、ミスを犯すことやバグを取り込むことは避けられません。web2py は 2 つの方法で支援します。1) すべての関数に対して、*edit* ページからブラウザで実行できるテストを作成することができます。2) エラーが明示されるとき、チケットが訪問者に発行され、エラーがログとして記録されます。

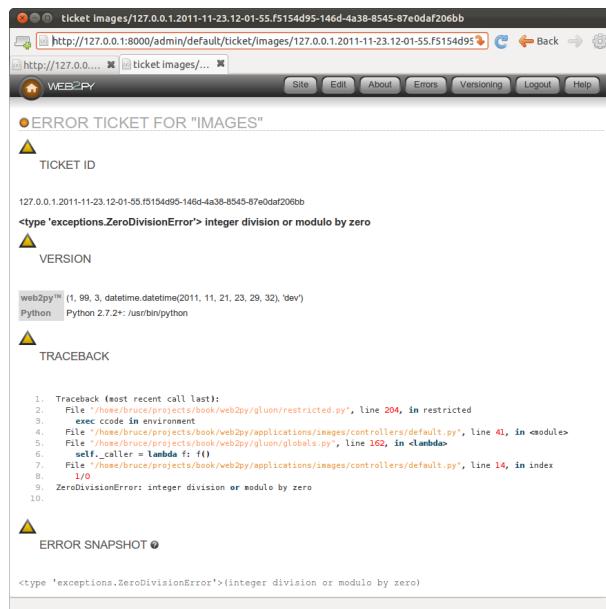
以下のように、故意に画像アプリケーションにエラーを挿入します：

```
1 def index():
2     images = db().select(db.image.ALL, orderby=db.image.title)
3     1/0
4     return dict(images=images)
```

index アクションにアクセスすると、次のようなチケットが表示されます：



管理者だけがこのチケットにアクセスすることができます：



チケットは、トレースバック、および、問題の原因となったファイルの内容、そして最終的なシステムの状態（変数、リクエスト、セッションなど）を示します。エラーがビューの中で発生した場合は、web2py はビューを HTML から Python コードに変換して表示します。これにより、簡単にファイルの論理構造を確認することができます。

デフォルトでは、チケットはトレースバックファイルシステムとグループに格納されています。管理インターフェイスでは、集計ビュー（発生のトレースと番号のタイプ）と詳細ビュー（すべてのチケットは、チケット ID により一覧表示されます）を提供しています。管理者は、2つのビューを切り替えることができます。

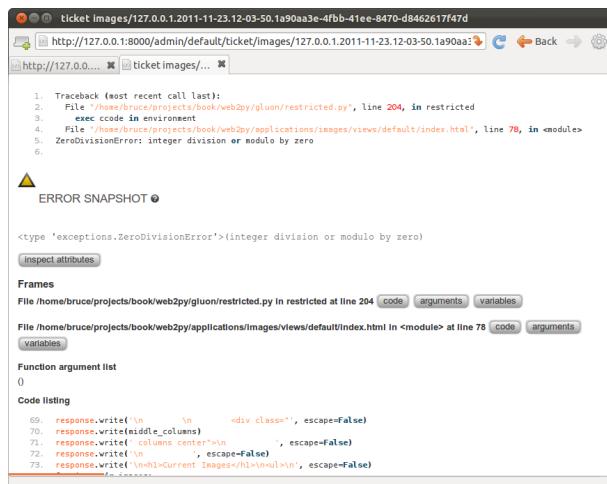
なお、admin は、すべての場所で、構文がハイライトされたコードを表示します（たとえば、エラーレポートでは、web2py のキーワードはオレンジ色で表示されます）。web2py のキーワードをクリックすると、キーワードに関するドキュメントページにリダイレクトされます。

アクションにある `1/0` というバグを修正し、そのバグを index ビューの中に埋め込んでみます：

```

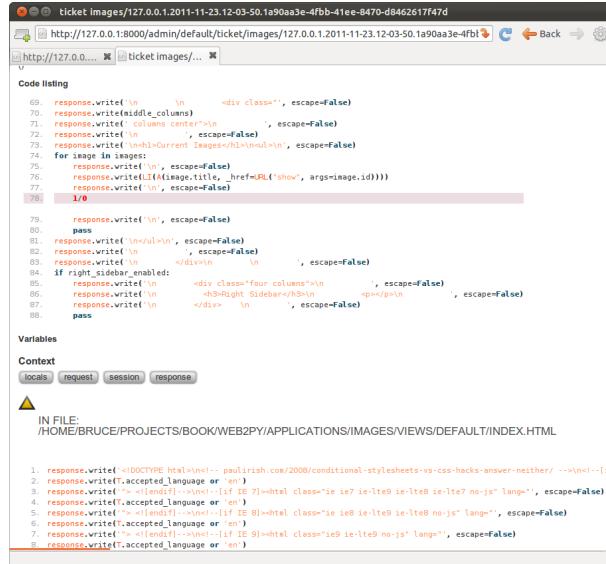
1 {{extend 'layout.html'}}
2
3 <h1>Current Images</h1>
4 <ul>
5 {{for image in images:}}
6 {{1/0}}
7 {{=LI(A(image.title, _href=URL("show", args=image.id)))}}
8 {{/pass}}
9 </ul>
```

すると、次のようなチケットを得ます：



ここで、web2py はビューを HTML から Python ファイルに変換します。チケッ

トに記述されるエラーは、元のビューファイルではなく、生成された Python コードを参照することに注意してください。



```

ticket images/127.0.0.1.2011-11-23.12-03-50.1a90aa3e-4fb-41ee-8470-d8462617f47d
http://127.0.0.1:8000/admin/default/ticket/images/127.0.0.1.2011-11-23.12-03-50.1a90aa3e-4fb...
http://127.0.0.1:8000/admin/default/ticket/images/127.0.0.1.2011-11-23.12-03-50.1a90aa3e-4fb...
V
Code listing
69.     response.write( '\n'           <div class="", escape=False)
70.     response.write(middle_columns)
71.     response.write( ' columns center=""', escape=False)
72.     response.write( ' style="text-align:center"', escape=False)
73.     response.write( '<h3>Current Images</h3>\n', escape=False)
74.     for image in images:
75.         response.write( ' ', escape=False)
76.         response.write( '<a href=%URL%(image.title, _href=URL("show", args=image.id))>', escape=False)
77.         response.write( '\n', escape=False)
78.     response.write( '1/0', escape=False)
79.     response.write( '\n', escape=False)
80.     pass
81.     response.write( '\n\n', escape=False)
82.     response.write( '\n'           <div>\n'           (right_sidebar, escape=False)
83.     response.write( '</div>\n'           (right_sidebar, escape=False)
84. if right_sidebar.enabled:
85.     response.write( '\n'           <div class="four columns"\n'           (right_sidebar, escape=False)
86.     response.write( '\n'           <h3>Right Sidebar</h3>\n'           (right_sidebar, escape=False)
87.     response.write( '\n'           </div>\n'           (right_sidebar, escape=False)
88.     pass
Variables
Context
locals request session response
⚠ IN FILE:
/HOME/BRUCE/PROJECTS/BOOK/WEB2PY/APPLICATIONS/IMAGES/VIEWS/DEFAULT/INDEX.HTML

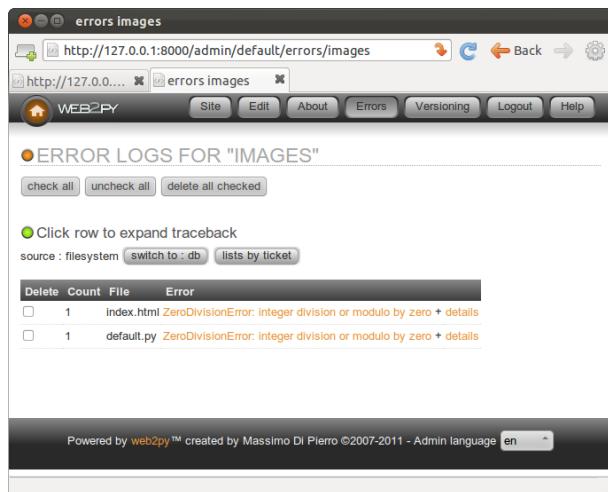
1. response.write( '<!DOCTYPE html>\n<!-- paulirish.com/2008/conditional-styleheets-vs-css-hacks-answer-neither/ -->\n<!--[if !IE]>\n2. response.write( accepted_language or 'en' )
3. response.write( '<![endif]--><!--[if IE 7]>\n4. response.write( accepted_language or 'en' )
5. response.write( '<![endif]--><!--[if IE 8]>\n6. response.write( accepted_language or 'en' )
7. response.write( '<![endif]--><!--[if IE 9]>\n8. response.write( accepted_language or 'en' )
B. response.write( accepted_language or 'en' )

```

最初は混乱するかもしれません。しかし、実践的にはこれはデバッグを容易にします。なぜなら、Python のインデントがビューに埋め込まれた論理構造を強調するからです。

コードは、同じページの下部に表示されます。

すべてのチケットは、各アプリケーションの *error* ページにおいて admin の下に列挙されます：



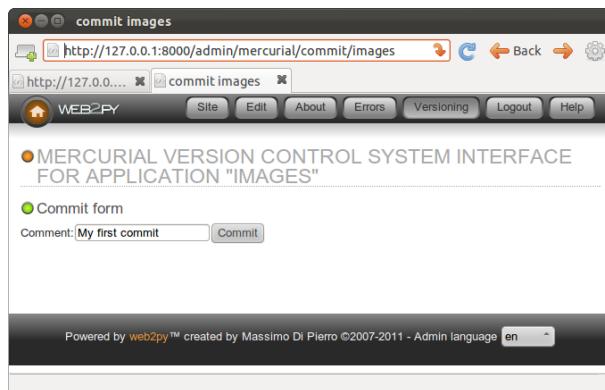
### 3.11.5 Mercurial

ソースコードから実行していく、Mercurial のバージョン・コントロールのライブラリをインストールしている場合：

```
1 easy_install mercurial
```

管理インターフェースは”mercurial”というメニュー項目を追加で表示します。これは、自動的にアプリケーションに対するローカルの Mercurial リポジトリを作成します。そのページにある”commit”ボタンを押すと、現在のアプリケーションがコミットされます。Mercurial は、コードに対して行った変更に関する情報を生成して格納するために、”.hg”という隠しフォルダを、アプリケーションのサブフォルダに作成します。すべてのアプリケーションは、独自の”.hg”フォルダと”.hgignore”ファイル（無視するファイルを Mercurial に伝えるファイル）があります。

Mercurial のウェブインターフェースを使用すると、前回のコミットや diff ファイルを閲覧することができます。しかし、より強力な機能を持つシェルまたは GUI ベースの Mercurial のクライアントを使って、直接 Mercurial を使用することを推奨します。たとえば、リモートソースリポジトリを使用してアプリを同期できるようになります。

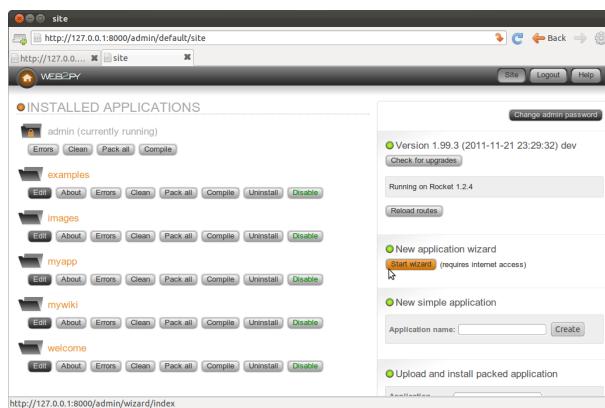


Mercurialについては、以下のURLでより詳しい情報を読むことができます。

1 <http://mercurial.selenic.com/>

### 3.11.6 管理ウィザード（実験的）

adminインターフェースでは、新しいアプリケーションを作成できるようにウィザードが含まれています。下の画像に示すように、”sites”ページからウィザードにアクセスできます。

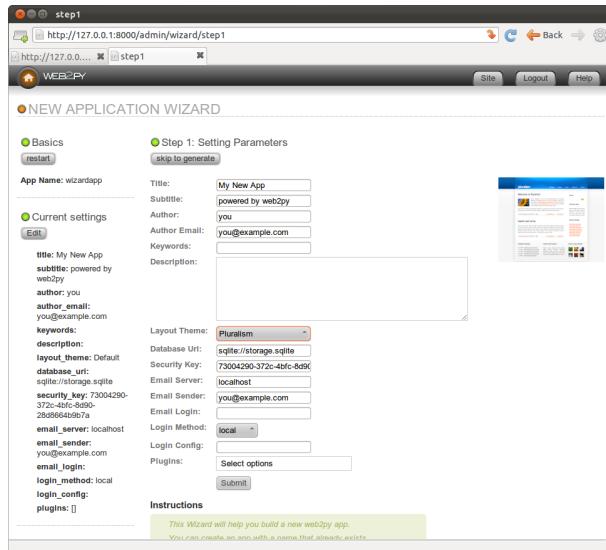


ウィザードは、新しいアプリケーションの作成に必要な一連の手順を示します。

- アプリケーション名の選択
- アプリケーションの構成と、必要なプラグインの選択

- 必要なモデルのビルト（各モデルのCRUDを行なうページが作成される）
- MARKMIN構文を使用して、ページのビューの編集を許可する

下の画像は、プロセスの2番目のステップを示しています。



(`web2py.com/layouts`から) レイアウトプラグインの選択を行なうためのドロップダウン、(`web2py.com/plugins`から) その他のプラグインを複数選択するためのドロップダウン、そして Janrain 用の”ドメイン : キー (domain:key)”をどこに配置するかを指定する”ログイン設定 (login config)”フィールドが用意されています。

他のステップは、ほぼ自明です。

このウィザードはよく機能していますが、二つの理由から実験的な機能となっています。

- ウィザードで作成したアプリケーションを手動で変更した場合、その後はウィザードで編集することができません。
- ウィザードのインターフェイスは、より多くの機能をサポートし、簡単にビジュアルな開発が行えるように、今後も変更されます。

いずれにしてもウィザードは、高速プロトタイピングのための便利なツールであり、別のレイアウトとオプションのプラグインを使用した新しいアプリケーションを作成するためのブートストラップとして使用することができます。

### 3.11.7 admin の設定

通常は、admin の構成を行う必要はありませんが、いくつかのカスタマイズが可能です。あなたは、admin にログインした後、次の URL を介して管理コンフィギュレーションファイルを編集することができます。

```
1 http://127.0.0.1:8000/admin/default/edit/admin/models/0.py
```

admin 自身も編集することができます。実際、admin も他と同じく一つのアプリケーションです。

”0.py” は自分自身が文書化されているので、したいことはどこを探せばいいかわかります。しかし、いくつか他に比べて重要なカスタマイズが存在します。

```
1 GAE_APPCFG = os.path.abspath(os.path.join('/usr/local/bin/appcfg.py'))
```

これは、Google App Engine の SDK に付属する”appcfg.py” ファイルの場所を指定している必要があります。SDK を使用している場合は、正しい値に設定パラメータを変更する必要があるかもしれません。正しく設定されていると、管理者インターフェースから GAE にデプロイすることができます。

web2py 管理者をデモモードで動かすことができます。

```
1 DEMO_MODE = True
2 FILTER_APPS = [ 'welcome' ]
```

フィルターアプリケーションに記載されているアプリだけがアクセスできるようになり、それらは読み取り専用モードでのみアクセスできるようになります。

あなたが教師であり、( バーチャルラボのような場所で ) 学生に対して教師側の管理インターフェイスを公開して共有できるようにする場合は、次のようにして行うことができます。

```
1 MULTI_USER_MODE = True
```

この場合、学生はログインする必要がありつつ学生自身の管理者インターフェイスにしかアクセスできませんが、教師は全ての学生の管理インターフェイスにアクセスすることができます。

このメカニズムは、まだすべてのユーザーが信頼されていると仮定している点に注意してください。admin の下に作成されたすべてのアプリケーションは、同じファイルシステム上に同じ資格情報で実行されます。他の学生が作成したデータやソースにアクセスできるアプリケーションを作成することもできます。

### 3.12 appadmin の追加情報

appadmin は一般に公開されることを意図されていません。これはデータベースへのアクセスを簡単に行えるようして開発者を支援するために設計されています。これには 2 つのファイルしか含まれていません：1 つは "appadmin.py" コントローラで、もう 1 つはそのコントローラのすべてのアクションから利用される "appadmin.html" ビューです。

appadmin コントローラは比較的小さく、読むことが可能です。ここでは、データベースのインターフェイス定義の例を提示しています。

appadmin はどのデータベースが利用可能なのか、どのテーブルが各データベースに存在しているかを示します。レコードを挿入することができ、また、各テーブルのレコードを個別に列挙することができます。appadmin は一度に 100 レコードを出力するようにページングします。

一旦、レコードのセットが選択されると、ページのヘッダが変化して、選択されたレコードを更新または削除することができるようになります。

レコードを更新するには、Query 文字列フィールドに SQL 文を入力します：

```
1 title = 'test'
```

ここで、文字列の値はシングルクオートで囲む必要があります。複数のフィールドはカンマで区切ることができます。

レコードを削除するには、対応するチェックボックスをクリックして、本当に行うか確認します。

appadmin はまた、SQL の FILTER が 2 つ以上のテーブルを含む SQL の条件を持つ場合、結合 (join) を実行することができます。たとえば、次の例を試してください：

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

web2py は、これを DAL に向かって渡します。DAL はそのクエリが 2 つのテーブルをリンクしていることを理解します。したがって、両者のテーブルは内部結合 (INNER JOIN) によって選択されます。これはその出力です：

The screenshot shows a web browser window with the URL `http://127.0.0.1:8000/appadmin/select/db`. The title bar says "Images". The main content area is titled "Rows in table" and contains a form with fields for "Query", "Update", and "Delete", each with a checkbox and a "submit" button. Below the form is a note about SQL JOINs. Underneath, it says "1 selected" and lists one row with columns: comment.id, comment.image\_id, comment.author, comment.email, comment.body, image.id, image.title, and image.file. The first column has a link labeled "1".

id フィールドの番号をクリックすると、対応する id を持つレコードに対する編集ページが得られます。

参照フィールドの数字をクリックすると、その参照されたレコードに対する編集ページが得られます。

結合 (join) によって選択された行を更新または削除することはできません。なぜなら、それらは複数のテーブルからのレコードを含み、対象が曖昧になるからです。

そのデータベース管理機能に加え、appadmin は、アプリケーションのキャッシュ (場所は `/yourapp/appadmin/ccache`) の内容の詳細を表示することができます。同様に、現在のリクエスト、レスポンスそしてセッションオブジェクト (場所は `/yourapp/appadmin/state`) の内容も表示できます。

appadmin は `response.menu` を自分自身のメニューに置き換えます。そこでは次のリンクが提供されます。admin の中にあるアプリケーションごとの edit ページ、db (データベース管理) ページ、state ページ、cache ページです。もし、アプリケーションのレイアウトで `response.menu` を使ったメニューの生成を行なわない場合、appadmin メニューは表示されません。この場合は、`appadmin.html` を修正して、メニューを表示するために `{=MENU(response.menu)}` を追加することができます。

第3版 - 翻訳: 細田謙二 レビュー: 中垣健志

第4版 - 翻訳: 中垣健志 レビュー: Mitsuhiro Tsuda



# 4

## コア

### 4.1 コマンドライン オプション

GUI を表示せずに web2py を開始するには、コマンドラインから次のように入力します：

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

web2py が起動すると、パスワードのハッシュ値を格納する場所として”parameters\_8000.py” ファイルが作成されます。”<ask>” をパスワードの代わりに入力すると、パスワードを入力するプロンプトが表示されます。for it. セキュリティ向上のため、次のように web2py を起動することができます：

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

この場合、web2py は以前に格納したパスワードのハッシュ値を再利用します。パスワードが提供されていない場合、つまり、”parameters\_8000.py” ファイルが削除されていた場合は、Web ベースの管理インターフェイスは無効になります。幾つかの Unix/Linux システム上で、パスワードが次の場合、

```
1 <pam_user:some_user>
```

web2py は、オペレーティングシステムのアカウント `some_user` の PAM パスワードを、PAM の設定でロックしていない限り、管理者認証に使用します。

*web2py* は通常 *CPython*(*Guido van Rossum*)によって作成された *C*言語で実装された *Python* で動作します。しかし *Jython*(*Java*による実装)でも動作します。後者によって、*J2EE*上で *web2py*を動かせることができるかもしれません。*Jython* を使うには、単に ”*python web2py.py...*” を ”*jython web2py.py*” と置き換えるだけです。*Jython* のインストールの詳細、及びデータベースのアクセスに必要な *zxJDBC* モジュールについては、第 14 章で説明します。

”*web2py.py*” スクリプトは、多くのコマンドライン引数を取ることができます。その中には、スレッドの最大数や SSL の有効化などがあります。以下は、完全なリストです：

```

1 >>> python web2py.py -h
2 Usage: python web2py.py
3
4 web2py Web Framework startup script. ATTENTION: unless a password
5 is specified (-a 'passwd'), web2py will attempt to run a GUI.
6 In this case command line options are ignored.オプション
7
8 :
9 Options:
10
11 --version           show program's version number and exit
12 -h, --help            show this help message and exit
13 -i IP, --ip=IP        ip address of the server (127.0.0.1)
14 -p PORT, --port=PORT  port of server (8000)
15 -a PASSWORD, --password=PASSWORD
16                           password to be used for administration (use -a
17                           "<recycle>" to reuse the last password))
18 -c SSL_CERTIFICATE, --ssl_certificate=SSL_CERTIFICATE
19                           file that contains ssl certificate
20 -k SSL_PRIVATE_KEY, --ssl_private_key=SSL_PRIVATE_KEY
21                           file that contains ssl private key
22 -d PID_FILENAME, --pid_filename=PID_FILENAME
23                           file to store the pid of the server
24 -l LOG_FILENAME, --log_filename=LOG_FILENAME
25                           file to log connections
26 -n NUMTHREADS, --numthreads=NUMTHREADS
27                           number of threads (deprecated)
28 --minthreads=MINTHREADS
29                           minimum number of server threads
30 --maxthreads=MAXTHREADS
31                           maximum number of server threads
32 -s SERVER_NAME, --server_name=SERVER_NAME
33                           server name for the web server
34 --request_queue_size=REQUEST_QUEUE_SIZE, --request_queue_size=REQUEST_QUEUE_SIZE

```

```

35                         max number of queued requests when server
36                         unavailable
37                         -o TIMEOUT, --timeout=TIMEOUT
38                         timeout for individual request (10 seconds)
39                         -z SHUTDOWN_TIMEOUT, --shutdown_timeout=SHUTDOWN_TIMEOUT
40                         timeout on shutdown of server (5 seconds)
41                         -f FOLDER, --folder=FOLDER
42                         folder from which to run web2py
43                         -v, --verbose           increase --test verbosity
44                         -Q, --quiet            disable all output
45                         -D DEBUGLEVEL, --debug=DEBUGLEVEL
46                         set debug output level (0-100, 0 means all, 100
47                         means
48                         none; default is 30)
49                         -S APPNAME, --shell=APPNAME
50                         run web2py in interactive shell or IPython (if
51                         installed) with specified appname (if app does
52                         not
53                         exist it will be created). APPNAME like a/c/f (
54                         c,f
55                         optional)
56                         -B, --bpython           run web2py in interactive shell or bpython (if
57                         installed) with specified appname (if app does
58                         not
59                         exist it will be created). Use combined with
60                         --shell
61                         -P, --plain             only use plain python shell; should be used
62                         with
63                         --shell option
64                         -M, --import_models     auto import model files; default is False;
65                         should be
66                         used with --shell option
67                         -R PYTHON_FILE, --run=PYTHON_FILE
68                         run PYTHON_FILE in web2py environment; should
69                         be used
70                         with --shell option
71                         -K SCHEDULER, --scheduler=SCHEDULER
72                         run scheduled tasks for the specified apps
73                         -K appl, app2, app3 requires a scheduler
74                         defined in the
75                         models of the respective apps
76                         -T TEST_PATH, --test=TEST_PATH
77                         run doctests in web2py environment; TEST_PATH
78                         like
79                         a/c/f (c,f optional)
80                         -W WINSERVICE, --winservice=WINSERVICE
81                         -W install|start|stop as Windows service
82                         -C, --cron              trigger a cron run manually; usually invoked
83                         from a

```

```

72          system crontab
73  --softcron      triggers the use of softcron
74  -N, --no-cron   do not start cron automatically
75  -J, --cronjob   identify cron-initiated command
76  -L CONFIG, --config=CONFIG
77          config file
78  -F PROFILER_FILENAME, --profiler=PROFILER_FILENAME
79          profiler filename
80  -t, --taskbar    use web2py gui and run in taskbar (system tray)
81  --nogui         text-only, no GUI
82  -A ARGS, --args=ARGS should be followed by a list of arguments to be
83          passed
84          to script, to be used with -S, -A must be the
85          last
86          option
87  --no-banner     Do not print header banner
88  --interfaces=INTERFACES
89          listen on multiple addresses:
                     "ip:port:cert:key;ip2:port2:cert2:key2;..." (:  

                     cert:key  

                     optional; no spaces)

```

小文字のオプションは、Web サーバーを設定するために使用されます。-L オプションは設定オプションをファイルから読み込むようにします。-W は、web2py を Windows サービスとしてインストールします。-S、-P、-M オプションは、インターフェイクティブな Python のシェルを開始します。-T オプションは、web2py の実行環境にあるコントローラの doctest を探し、実行します。次の例は、”welcome” アプリケーションの全コントローラの doctest を実行します：

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

web2py を Windows サービスとして実行している場合（-W オプション）コマンドライン引数を使用して設定を渡すのは簡単ではありません。このため web2py のフォルダに、内部の Web サーバー設定用として、”options\_std.py” がサンプルで用意されています：

```

1 import socket
2 import os
3
4 ip = '0.0.0.0'
5 port = 80
6 interfaces=[('0.0.0.0', 80)]
7 #interfaces.append(('0.0.0.0', 443, 'ssl_private_key.pem',
8 #                   'ssl_certificate.pem'))
8 password = '<recycle>' ## <recycle> means use the previous password

```

```
9 pid_filename = 'httpserver.pid'
10 log_filename = 'httpserver.log'
11 profiler_filename = None
12 minthreads = None
13 maxthreads = None
14 server_name = socket.gethostname()
15 request_queue_size = 5
16 timeout = 30
17 shutdown_timeout = 5
18 folder = os.getcwd()
19 extcron = None
20 nocron = None
```

このファイルには、web2py のデフォルト値が含まれています。このファイルを編集した場合、`-L` コマンドラインオプションを用いて明示的にインポートする必要があります。なおこれは、web2py を Windows サービスとして動作させた場合のみ機能します。

## 4.2 ワークフロー

web2py のワークフローは以下の通りです:

- HTTP リクエストが web サーバー（組み込みの Rocket サーバー、WSGI を経由して web2py に接続する異なるサーバー、もしくは別のアダプタ）に到達します。web サーバーは各リクエストを、各自のスレッドで並列に処理します。
- HTTP リクエストヘッダーが解析され、ディスパッチャー（この章で後ほど説明）に渡されます。
- ディスパッチャーは、どのインストールアプリケーションがリクエストを処理するかを決め、URL の PATH\_INFO を関数呼び出しにマッピングします。各 URL は、1 つの関数呼び出しに対応します。
- static フォルダのファイルに対するリクエストは、直接処理されます。また大きいファイルは自動的に、ストリーム処理されクライアントに送られます。
- static ファイル向けを除くどのリクエストも、1 つのアクションにマッピングされます（つまり、リクエストされたアプリケーションのコントローラファイルの 1 つの関数）。

- アクションが呼び出される前に、次のように幾つかのことが起こります。リクエストヘッダーがアプリのセッションクッキーを含んでいる場合、セッションオブジェクトが取り出されます。そうでない場合は、セッション id が生成されます(ただし、セッションファイルは後になるまで保存されません)。そしてリクエストに対する実行環境が作成されます。この実行環境でモデルが実行されます。
- 最後に、コントローラのアクションが事前に構築された環境で実行されます。
- アクションが文字列を返す場合、その文字列がクライアントに返されます(もしくは、アクションが web2py の HTML ヘルパー オブジェクトの場合、それがシリアル化されてクライアントに返されます)。
- アクションがインテレート可能オブジェクトを返す場合、クライアントへデータをループリストリームするために用いられます。
- アクションが辞書を返す場合、web2py はその辞書をレンダリングするためのビューを特定しようと試みます。ビューはアクションと同じ名前を持つ必要があります(特に指定がない限り)。また、リクエストページと同じ拡張子を持つ必要があります(デフォルトは.html です)。特定に失敗した場合、web2py は汎用ビュー(利用可能で有効な場合)を選択します。ビューはアクションによって返される辞書と同じように、モデルで定義した全ての変数を参照しますが、コントローラで定義したグローバル変数は参照しません。
- 全てのユーザーコードは、特に指定がない限り、単一のトランザクションで実行されます。
- ユーザーコードが成功すると、トランザクションがコミットされます。
- ユーザーコードが失敗すると、トレースバックがチケットに格納され、クライアントにチケット ID が発行されます。システム管理者だけが、チケットのトレースバックを検索し読むことができます。

念頭に置くべき、幾つかの注意事項があります:

- 同じフォルダ/サブフォルダ内のモデルは、アルファベット順に実行されます。
- モデルで定義した全ての変数は、アルファベット順の後続する他のモデルと、コントローラ、ビューで参照できます。
- サブフォルダのモデルは、条件付で実行されます。例えば、”a” はアプリケー

ション、”c” はコントローラ、”f” は関数 (アクション) で、ユーザーが”/a/c/f” をリクエストする場合、次のモデルが実行されます。

```
1 applications/a/models/*.py
2 applications/a/models/c/*.py
3 applications/a/models/c/f/*.py
```

- リクエストされたコントローラが実行され、リクエストされた関数が呼び出されます。これは、コントローラ内の全てのトップレベルのコードは、そのコントローラに対する全てのリクエストで、実行されることを意味しています。
- ビューは、アクションが辞書を返した場合にのみ呼び出されます。
- ビューが見つからない場合、web2py は汎用ビューを使用しようと試みます。デフォルトでは、汎用ビューは無効になっています。ただし’welcome’ アプリでは、ローカルホストにおいてのみ、有効なるコードを /models/db.py に含んでいます。それらは、拡張子のタイプ毎及び、アクション毎に (response.generic\_patterns を用いて) 有効にすることができます。一般に、汎用ビューは開発ツールです。そして普通は、本番環境では使用すべきではありません。汎用ビューを使用したい、幾つかのアクションがある場合、response.generic\_patterns にそれらのアクションをリストしてください (サービスに関する章で詳細に説明します)。

アクションの実行可能なふるまいは、以下の通りです:

文字列を返す

```
1 def index(): return 'data'
```

辞書をビューに対して返す

```
1 def index(): return dict(key='value')
```

全てローカル変数を返す

```
1 def index(): return locals()
```

ユーザーを他のページへリダイレクトさせる

```
1 def index(): redirect(URL('other_action'))
```

”200 OK” 以外の HTTP ページを返す

```
1 def index(): raise HTTP(404)
```

## ヘルパー (FORM など) を返す

```
1 def index(): return FORM(INPUT(_name='test'))
```

(これは Ajax のコールバックやコンポーネントで、主に使用されます。12 章を参照してください)

アクションが辞書を返す時、データベーステーブルに基づいたフォームやファクトリからのフォームなど、ヘルパーによって生成されたコードが含まれるかもしれません。例えば:

```
1 def index(): return dict(form=SQLFORM.factory(Field('name')).process())
```

(web2py によって生成されたすべてのフォームは、ポストバックを使用します。3 章を参照してください)

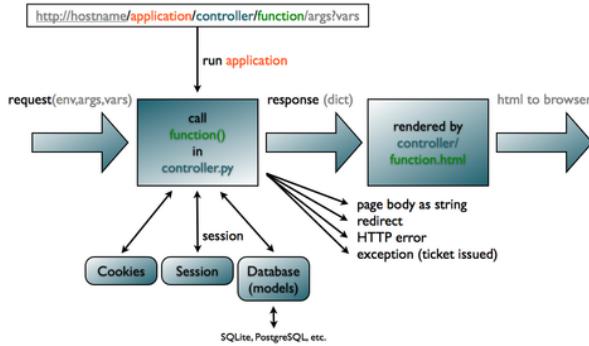
## 4.3 ディスパッチ

web2py は次のような形式の URL をマッピングします:

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

”a” アプリケーションの、”c.py” コントローラの、`f()` 関数に、マッピングします。もし `f` が存在しない場合、コントローラの `index` 関数がデフォルトとして使用されます。`c` が存在しない場合、”default.py” がデフォルトのコントローラとして使用されます。`a` が存在しない場合、`init` がデフォルトのアプリケーションとして使用されます。`init` アプリケーションがない場合に、web2py は `welcome` アプリケーションの実行を試みます。下のイメージは、これを図式化して表したものです：

(デフォルトのアプリケーション、コントローラ、関数の名前は、routes.py にて書き換えることができます。後述のデフォルトアプリケーション、コントローラ、関数を参照してください)。



デフォルトでは、どの新しいリクエストも、新しいセッションを作成します。さらに、セッションクッキーは、セッションを維持するためにクライアントのブラウザに返されます。

拡張子.html はオプションです。.html はデフォルトとして仮定されています。拡張子は、コントローラ関数 `f()` の出力をレンダリングする、ビューの拡張子を決定します。拡張子によって、同じコンテンツを複数の形式 (HTML、XML、JSON、RSS フィードなど) で提供することができます。

引数を取る関数や、`_` 2つのアンダースコアで始まる関数は一般には公開されず、他の関数からしか呼び出すことができません。

URL の形式では、次のような例外があります。

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

ここでは、”static” というコントローラは存在しません。web2py は、”a” アプリケーションの”static” サブフォルダにある、”filename” ファイルへのリクエストと解釈します。

静的ファイルをダウンロードする時、web2py はセッションを作成せず、クッキーを発行したりモデルを実行することもありません。web2py は常に、ストリームの静的ファイルを 1MB ごとに分割します。そして、クライアントが分割ファイルに対する範囲 (RANGE) のリクエストを送信した時に、分割コンテンツ (PARTIAL CONTENT) を送信します。web2py は、IF\_MODIFIED\_SINCE プロトコルもサポートしています。ファイルがすでにブラウザに格納されてい

て、そのファイルが現在のバージョンから変更されていない場合、ファイルは送信されません。static フォルダのオーディオやビデオファイルにリンクする時、メディアプレーヤーを使ってオーディオやビデオをストリーミングする代わりに、ブラウザにファイルダウンロードを強制させたい場合、URL に?attachment を加えてください。これは、HTTP レスポンスの Content-Disposition ヘッダーに”attachment” をセットすることを、web2py に指示します。例えば：

```
1 <a href="/app/static/my_audio_file.mp3?attachment">Download</a>
```

上記のリンクがクリックされると、すぐにオーディオをストリーミングするのではなく、MP3 ファイルをダウンロードするよう、ブラウザはユーザにプロンプトを表示します（後述で触れるように、ヘッダー名とその値からなる dict を response.headers に割り当てることにより、直接 HTTP レスポンスヘッダーをセットすることもできます）。

web2py はフォームの GET/POST リクエストを、次のようにマッピングします：

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

アプリケーションに a、コントローラに”c.py”、関数に f を割り当て、次のように request 变数に URL パラメータを格納します：

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

及び：

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

及び：

```
1 request.application = 'a'
2 request.controller = 'c'
3 request.function = 'f'
```

上記の例で、request.args[i] と request.args(i) はいずれも、request.args の i 番目の要素の取得に使用できます。しかし前者は、request.args に i 番目の要素がない場合に、例外が発生します。後者は None を返します。

```
1 request.url
```

これは、現在のリクエストに対する完全な URL(ただし GET 变数は含まれない)を格納しています。

```
1 request.ajax
```

これはデフォルトで False ですが、Ajax リクエストによってアクションが呼ばれたと web2py が判断した場合には、True になります。

リクエストが Ajax で、web2py のコンポーネントによって起動したものであれば、コンポーネントの名前は次によって知ることができます：

```
1 request.cid
```

コンポーネントについては、12 章で詳しく説明します。

HTTP リクエストが GET である場合、`request.env.request_method` は”GET” がセットされます。POST の場合、`request.env.request_method` は”POST” がセットされます。URL のクエリ変数は、`request.vars` の Storage 辞書に格納されます。これらの値は `request.get_vars`(GET リクエストの場合) または、`request.post_vars`(POST リクエストの場合) にも格納されます。web2py は、WSGI と web2py の環境変数を `request.env` に格納します。例えば：

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

また、HTTP ヘッダを環境変数に次のように格納します：

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

`web2py` は全ての URL に対して、ディレクトリトラバーサル攻撃を防ぐためにバリデータを行っています。

URL は、英数字、アンダースコア、スラッシュのみでしか構成されません。ただし、`args` は連続しないドットを含むこともあります。空白文字は、バリデーションの前にアンダースコアに置き換えられます。URL 構文が無効な場合は、HTTP 400 エラーメッセージ [47, 48] を返します。

URL が静的ファイルへのリクエストに一致する場合、`web2py` は要求されたファイルを単純に読み込んで返します(ストリーミングします)。

URL が静的ファイルへのリクエストでなかった場合、`web2py` はリクエストを次の順序で処理します：

- クッキーを解析します。
- 関数が実行される環境を作成します。

- `request`、`response`、`cache` を初期化します。
- 既存の `session` を開くか、新しいものを作成します。
- リクエストされたアプリケーションに属するモデルを実行します。
- リクエストされたコントローラのアクション関数を実行します。
- 関数が辞書を返す場合、関連付けられたビューを実行します。
- 成功した場合、すべての開いているトランザクションをコミットします。
- セッションを保存します。
- HTTP レスポンスを返します。

コントローラとビューは、同じ環境の異なるコピーで実行されることに注意してください。したがって、ビューはコントローラを参照しません。代わりにモデルと、コントローラのアクション関数によって返される変数を参照します。

(HTTP 以外の) 例外が発生した場合、web2py は次の処理を行います：

- トレースバックをエラーファイルに格納し、チケット番号をそれに割り当てます。
- すべてのオープンなトランザクションをロールバックします。
- チケット番号を報告するエラーページを返します。

例外が HTTP の例外の場合、意図した動作 (例えば HTTP のリダイレクト) と見なされ、すべてのオープンなデータベース・トランザクションがコミットされます。その後の動作は、HTTP 例外自身で指定されています。HTTP 例外クラスは、Python 標準の例外ではなく、web2py で定義されています。

#### 4.4 ライブライ

web2py のライブラリは、グローバルオブジェクトとしてユーザアプリケーションに公開されます。例えば、`request`、`response`、`session`、`cache`、クラス (ヘルパー、バリデータ、DAL の API)、関数 (`t` と `redirect`) などがあります。

これらのオブジェクトは、次のコアファイルで定義されています：

<sup>1</sup> web2py.py

```

2 gluon/__init__.py      gluon/highlight.py    gluon/restricted.py   gluon/
   streamer.py
3 gluon/admin.py        gluon/html.py       gluon/rewrite.py     gluon/
   template.py
4 gluon/cache.py        gluon/http.py      gluon/rocket.py     gluon/
   storage.py
5 gluon/cfs.py          gluon/import_all.py  gluon/sanitizer.py  gluon/
   tools.py
6 gluon/compileapp.py   gluon/languages.py  gluon/serializers.py gluon/
   utils.py
7 gluon/contenttype.py  gluon/main.py      gluon/settings.py   gluon/
   validators.py
8 gluon/dal.py          gluon/myregex.py   gluon/shell.py     gluon/
   widget.py
9 gluon/decoder.py      gluon/newcron.py  gluon/sql.py      gluon/
   winservice.py
10 gluon/fileutils.py   gluon/portalocker.py gluon/sqlhtml.py   gluon/
   xmlrpc.py
11 gluon/globals.py    gluon/reserved_sql_keywords.py

```

次の雑形アプリが、tar+gzip で圧縮されて web2py に同梱しています。

```
1 welcome.w2p
```

このアプリは、インストール時に作成され、アップグレード時に上書きされます。

最初に *web2py* を起動すると、*deposit* と *applications* という 2 つの新しいフォルダが作成されます。"welcome" アプリは "welcome.w2p" ファイルに圧縮され、雑形アプリとして利用されます。*deposit* フォルダはアプリケーションのインストール、アンインストールのための一時的な格納場所として使用されます。

web2py のユニットテストは、次にあります。

```
1 gluon/tests/
```

さまざまな Web サーバーと接続するための、ハンドラがあります：

```

1 cghandler.py      # discouraged
2 gaehandler.py    # for Google App Engine
3 fcgihandler.py   # for FastCGI
4 wsgihandler.py   # for WSGI
5 isapiwsgihandler.py # for IIS
6 modpythonhandler.py # deprecated

```

("fcgihandler" は、Allan Saddi によって開発された "gluon/contrib/gateways/fcgi.py" を呼び出します)、そして、

1 anyserver.py

これは、多くの異なるウェブサーバーのインターフェースになるスクリプトです。13章で説明します。

3つのサンプルファイルがあります:

1 options\_std.py  
2 routes.example.py  
3 router.example.py

1番目は、オプションの設定ファイルで、-L オプションで web2py.py に渡されます。2番目は、URL マッピングファイルのサンプルです。”routes.py” にリネームすると、自動で読み込まれます。3番目は、URL マッピングのための代替構文です。”routes.py” にリネーム（またはコピー）することができます。

次のファイルは

1 app.yaml  
2 index.yaml  
3 queue.yaml

Google App Engine 上にデプロイするために必要な設定ファイルです。詳しくは、デプロイレシピの章と Google のドキュメンテーションのページを参照ください。

一般に、サードパーティによって開発された、追加のライブラリもあります：

**feedparser** [28] は、Mark Pilgrim によって作されました。RSS と Atom のフィードの読み取りに使用します：

1 gluon/contrib/\_\_init\_\_.py  
2 gluon/contrib/feedparser.py

**markdown2** [29] は、Trent Mick によって作らました。wiki マークアップに使用します：

1 gluon/contrib/markdown/\_\_init\_\_.py  
2 gluon/contrib/markdown/markdown2.py

**markmin** のマークアップです：

1 gluon/contrib/markmin.py

**pyfpdf** は Mariano Reingart によって作成された、PDF ドキュメント生成ツールです：

```
1 gluon/contrib/pyfpdf
```

この本には記載されていませんが、次の場所にドキュメントがあります：

```
1 http://code.google.com/p/pyfpdf/
```

**pysimplesoap** は Mariano Reingart によって作成された、軽量な SOAP サーバーの実装です。

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

**simplejsonrpc** は Mariano Reingart によって作成された、軽量な JSON-RPC クライアントです：

```
1 gluon/contrib/simplejsonrpc.py
```

**memcache** [30] 用の Python API です。Evan Martin によって作成されました：

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

**redis\_cache** は、redis データベースにキャッシュを格納するモジュールです：

```
1 gluon/contrib/redis_cache.py
```

**gql** は、DAL を Google App Engine 用に移植したものです。

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

**memdb** は、DAL を memcache 上に移植したものです：

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

**gae\_memcache** は、Google App Engine 上で memcache を使うための API です。

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

**pyrtf** [26] は、リッチテキストフォーマット (RTF) を生成するためのライブラリです。Simon Cusack によって開発され、Grant Edwards により改定されました。

```
1 gluon/contrib/pyrtf  
2 gluon/contrib/pyrtf/__init__.py  
3 gluon/contrib/pyrtf/Constants.py  
4 gluon/contrib/pyrtf/Elements.py
```

```

5 gluon/contrib/pyrtf/PropertySets.py
6 gluon/contrib/pyrtf/README
7 gluon/contrib/pyrtf/Renderer.py
8 gluon/contrib/pyrtf/Styles.py

```

**PyRSS2Gen** [27] は、RSS フィードを生成するためのものです。Dalke Scientific Software によって開発されました：

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

**simplejson** [25] は、JSON オブジェクトを解析し、書き込みのための標準ライブラリです。Bob Ippolito によって作成されました：

```

1 gluon/contrib/simplejson/__init__.py
2 gluon/contrib/simplejson/decoder.py
3 gluon/contrib/simplejson/encoder.py
4 gluon/contrib/simplejson/jsonfilter.py
5 gluon/contrib/simplejson/scanner.py

```

**Google Wallet** [96] は、支払い処理のために、Google にリンクする”pay now”ボタンを提供します。

```
1 gluon/contrib/google_wallet.py
```

**Stripe.com** [98] は、クレジットカード決済を受理するためのシンプルな API を提供します。

```
1 gluon/contrib/stripe.py
```

**AuthorizeNet** [99] は、Authorize.net ネットワークを介して、クレジットカード決済を受理するための API を提供します

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

**Dowcommerce** [100] は、もう 1 つのクレジットカード処理 API です：

```
1 gluon/contrib/DowCommerce.py
```

**PAM** [75] は、Chris AtLee によって作られた認証 API です：

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

テスト用のダミーデータをデータベースに投入する、ペイズ分類器です：

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

web2py がサービスとして稼働している時、windows のタスクバーでインタラクティブな操作を行うためのファイルです：

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

オプションの `login_methods` と `login_forms` は、認証で使用されます：

```
1 gluon/contrib/login_methods/__init__.py
2 gluon/contrib/login_methods/basic_auth.py
3 gluon/contrib/login_methods/cas_auth.py
4 gluon/contrib/login_methods/dropbox_account.py
5 gluon/contrib/login_methods/email_auth.py
6 gluon/contrib/login_methods/extended_login_form.py
7 gluon/contrib/login_methods/gae_google_account.py
8 gluon/contrib/login_methods/ldap_auth.py
9 gluon/contrib/login_methods/linkedin_account.py
10 gluon/contrib/login_methods/loginza.py
11 gluon/contrib/login_methods/oauth10a_account.py
12 gluon/contrib/login_methods/oauth20_account.py
13 gluon/contrib/login_methods/openid_auth.py
14 gluon/contrib/login_methods/pam_auth.py
15 gluon/contrib/login_methods/rpx_account.py
16 gluon/contrib/login_methods/x509_auth.py
```

web2py にはまた、以下のような便利なスクリプトが収められているフォルダがあります。

```
1 scripts/setup-web2py-fedora.sh
2 scripts/setup-web2py-ubuntu.sh
3 scripts/setup-web2py-nginx-uwsgi-ubuntu.sh
4 scripts/update-web2py.sh
5 scripts/make_min_web2py.py
6 ...
7 scripts/sessions2trash.py
8 scripts/sync_languages.py
9 scripts/tickets2db.py
10 scripts/tickets2email.py
11 ...
12 scripts/extract_mysql_models.py
13 scripts/extract_pgsql_models.py
14 ...
15 scripts/access.wsgi
16 scripts/cpdb.py
```

最初の 3 つは特に便利で、web2py の本番環境の完全なインストールとセットアップを、スクラッチで行うよう試みます。これらの幾つかは 14 章にて説明さ

れます。しかし、その目的と使用方法を説明するドキュメントの文字列が、スクリプトファイル中に書かれています。

最後に web2py は、バイナリ・ディストリビューションを作成するために必要なファイルを含んでいます。

```
1 Makefile
2 setup_exe.py
3 setup_app.py
```

これらは py2exe と py2app ための、それぞれのセットアップスクリプトです。 web2py のバイナリ・ディストリビューションを作成するためだけに必要です。あなたが、それらを実行する必要は全くありません。

要約すると、 web2py のライブラリは次の機能を提供します：

- URL を関数呼び出しにマッピングします。
- HTTP 経由のパラメータの受け渡しを行います。
- それらのパラメータのバリデーションを行います。
- ほとんどのセキュリティ問題からアプリケーションを保護します。
- データの永続性(データベース、セッション、キャッシュ、クッキー)を扱います。
- 各種のサポートされた言語用に文字列の翻訳を行います。
- HTML をプログラム的に生成します(データベーステーブルからなど)。
- データベース抽象化レイヤ(DAL)を介して SQL を生成します。
- リッチテキストフォーマット(RTF)出力を生成します。
- Comma-Separated Value(CSV) 形式の出力をデータベースのテーブルから生成します。
- Really Simple Syndication(RSS) フィードを生成します。
- Ajax 用に、JavaScript Object Notation(JSON) のシリアル化文字列を生成します。
- wiki マークアップ(Markdown)を HTML に変換します。
- XML-RPC の Web サービスを公開します。

- 大きいファイルのアップロードとダウンロードをストリーミングを介して行います。

web2py アプリケーションには、追加のファイルが含まれています。特に、jQuery、calendar、EditArea、nicEdit などの、サードパーティ製の JavaScript ライブライアリが含まれています。作者の情報はファイル自体に記載されています。

## 4.5 アプリケーション

web2py で開発されたアプリケーションは、以下のパートから構成されています：

- `models` は、データベースのテーブルとテーブル間のリレーションのようなデータの表現を記述します。
- `controllers` は、アプリケーションのロジックとワークフローを記述します。
- `views` は、JavaScript と HTML を使用して、ユーザにデータをどのように表示するかを記述します。
- `languages` は、アプリケーションの文字列を、サポートされている各種の言語にどのように翻訳するかを記述します。
- `static files` は、処理を必要としません (例えば画像や CSS スタイルシート等)。
- `ABOUT` と `README` は、その名の通りの文書です。
- `errors` は、アプリケーションで発生したエラーのレポートを格納します。
- `sessions` は、各ユーザー固有の情報を格納します。
- `databases` は、SQLite データベースと、付加的なテーブル情報を格納します。
- `cache` は、キャッシュされたアプリケーションの項目を格納します。
- `modules` は、その他の追加 Python モジュールです。
- `private` ファイルは、コントローラからアクセスされますが、開発者から直接アクセスされません。
- `uploads` ファイルは、モデルからアクセスされますが、開発者から直接アクセスされません (ユーザによりアップロードされたファイルなど)。

- tests は、テスト用のスクリプト、フィクスチャやモックを格納するためのディレクトリです。

models、views、controllers、languages、static files は、web の管理インターフェイス [デザイン] を介してアクセス可能です。ABOUT、README、errors も、管理インターフェイスを介して対応するメニュー項目を経てアクセスできます。sessions、cache、modules、private files は、アプリケーションからはアクセスできますが、管理インターフェイスからはアクセスできません。

ユーザがファイルシステムに直接アクセスする必要はないですが、全ては明確なディレクトリ構造にきれいに構造化されています。そしてこれらは、インストールした各 web2py のアプリケーションに複製されています。

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

”\_\_init\_\_.py” は空のファイルです。これは Python (そして web2py も) が、modules ディレクトリ内のモジュールを、インポートするために必要となります。

なお、admin アプリケーションはサーバー・ファイルシステム上で、web2py アプリケーションのための、シンプルな web インターフェイスを提供します。web2py アプリケーションは、コマンドラインから作成と開発を行うこともできます。このため、ブラウザの admin インターフェイスを、使用しなければならないというわけではありません。また新規のアプリケーションは、上記のディレクトリ構造を、例えば、”applications/newapp/” の下に複製し、手動で作成することができます (もしくは、新規アプリケーションのディレクトリ上で、welcome.w2p を単純に untar で展開することでも可能です)。admin インターフェイスを使わずに、アプリケーションのファイルをコマンドラインで、作成と編集を行うこともできます。

## 4.6 API

models、controllers、views は、次のオブジェクトがすでにインポートされている環境で実行されます：

グローバルオブジェクト：

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

国際化:

```
1 T
```

ナビゲーション:

```
1 redirect, HTTP
```

ヘルパー:

```
1 XML, URL, BEAUTIFY
2
3 A, B, BODY, BR, CENTER, CODE, COL, COLGROUP,
4 DIV, EM, EMBED, FIELDSET, FORM, H1, H2, H3, H4, H5, H6,
5 HEAD, HR, HTML, I, IFRAME, IMG, INPUT, LABEL, LEGEND,
6 LI, LINK, OL, UL, META, OBJECT, OPTION, P, PRE,
7 SCRIPT, OPTGROUP, SELECT, SPAN, STYLE,
8 TABLE, TAG, TD, TEXTAREA, TH, THEAD, TBODY, TFOOT,
9 TITLE, TR, TT, URL, XHTML, xmlescape, embed64
10
11 CAT, MARKMIN, MENU, ON
```

\*\*フォームとテーブル\*

```
1 SQLFORM (SQLFORM.factory, SQLFORM.grid, SQLFORM.smartgrid)
```

バリデータ:

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

データベース:

```
1 DAL, Field
```

下位互換性のために、SQLDB=DAL と SQLField=Field となっています。新しい構文である DAL と Field を、古い構文の代わりに使うことを推奨します。

他のオブジェクトとモジュールは、ライブラリで定義されます。しかし、頻繁に使用されるわけではないので、自動的にはインポートされません。web2py の実行環境におけるコアとなる API の実体は、後述する、request、response、session、cache、URL、HTTP、redirect、T です。

Auth、Crud、Serviceなどの幾つかのオブジェクトや関数は、”gluon/tools.py”で定義されています。そして、それらは必要な時に import する必要があります：

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

#### 4.6.1 Python モジュールからの API アクセス

モデルやコントローラが python モジュールをインポートし、これらが web2py の API の幾つかを使用する必要があるかもしれません。これを実現する方法は、次のようにインポートすることです：

```
1 from gluon import *
```

実際、例え web2py アプリケーションによってインポートされていないくとも、web2py が `sys.path` に存在さえすれば、どの Python モジュールも web2py の API をインポートすることができます。

しかし、1つの注意点があります。web2py は HTTP リクエストが出現した（もしくは偽装された）時にのみ、存在する幾つかのグローバルオブジェクト (`request`, `response`, `session`, `cache`, `T`) を定義しています。したがって、モジュールはアプリケーションから呼び出された場合にのみ、それにアクセスできます。このため、それらのオブジェクトは、コンテナの呼び出し元という意味の `current` と呼ばれる、スレッドローカル・オブジェクトに配置されます。ここに例を示します。

次のコードを含む、”/myapp/modules/test.py” モジュールを作成します：

```
1 from gluon import *
2 def ip(): return current.request.client
```

すると、”myapp” のコントローラーから次のことが可能になります：

```
1 import test
2 def index():
3     return "Your ip is " + test.ip()
```

幾つかの点に注意してください：

- `import test` は最初に、現在のアプリのモジュールフォルダで、モジュールを探します。次に、`sys.path` のリストにあるフォルダを探します。したがって、アプリレベルのモジュールは、Python モジュールよりも優先されま

す。これにより、異なるアプリで衝突することなく、異なるバージョンのモジュールを使用して出荷できるようになります。

- 異なるユーザーが、モジュールの関数を呼び出す同じアクション `index` を同時に呼び出すことができますが、衝突は起きません。なぜなら、`current.request` は、異なるスレッドで異なるオブジェクトだからです。ただし、モジュールの関数やクラスの外で（つまりトップレベルで）、`current.request` にアクセスしないように注意してください。
- `import test` は `from applications.appname.modules import test` のショートカットです。長い構文を使うと、他のアプリケーションのモジュールをインポートすることが可能になります。

変更が行われた場合、通常の Python の挙動と一緒に、デフォルトの web2py はモジュールをリロードしません。しかし、これは変更できます。モジュールの自動リロードを有効にするには、次のような `track_changes` 関数を（通常はモデルファイルに、インポートの前で）使用してください。

```
1 from gluon.custom_import import track_changes; track_changes(True)
```

こうすると、モジュールがインポートされる度に、インポーターは Python のソースファイル (.py) に変更がなかったをチェックします。変更があると、モジュールはリロードされます。これは全ての Python モジュールに、例え web2py の外にある Python モジュールでも適用されます。このモードはグローバルで、全てのアプリケーションに適用されます。モデル、コントローラ、ビューの変更は、このモードとは関係なく常にリロードされます。モードを無効にするには、引数を `False` にした同じ関数を使用してください。現在のトラッキング状態を知るには、同じ `gluon.custom_import` の `is_tracking_changes()` を使用してください。

`current` をインポートしたモジュールは、次のものにアクセスできます：

- `current.request`
- `current.response`
- `current.session`
- `current.cache`
- `current.T`

さらに、アプリケーションで `current` に格納することを決めた、どの他の変数にもアクセスできます。例えば、モデルを次のようにすると

```
1 auth = Auth(db)
2 from gluon import current
3 current.auth = auth
```

インポートした全てのモジュールで、次にアクセス可能です:

- `current.auth`

`current` と `import` はアプリケーションに対して、拡張性かつ再利用性のあるモジュールを構築するための、強力なメカニズムを作成します。

1つの大きな注意点があります。`from gluon import current` が与えられた時、`current.request` 及び、他のどのスレッドローカルのオブジェクトを使用することも、正しいです。しかしこのように、モジュールのグローバル変数に割り当てるには、すべきではありません。

```
1 request = current.request # WRONG! DANGER!
```

また、クラス属性に割り当てるにも、しないでください。

```
1 class MyClass:
2     request = current.request # WRONG! DANGER!
```

なぜなら、スレッドローカルのオブジェクトは、実行時に値を引き出す必要があるからです。反対にグローバル変数は、最初にモデルがインポートされた時に、1度だけ定義されます。

## 4.7 request

`request` オブジェクトは、Python の `dict` クラスを拡張した `gluon.storage.Storage` という、ユピキタンスな web2py のクラスのインスタンスです。基本的には辞書ですが、項目の値は属性としてアクセスすることができます :

```
1 request.vars
```

これは次と同じです :

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

辞書とは異なり、属性（またはキー）がない場合、例外を発生させずに代わりに `None` を返します。

独自のストレージオブジェクトを、作成しておくと便利な時があります。  
次のように行うことができます：

```
1 from gluon.storage import Storage
2 my_storage = Storage() # empty storage object
3 my_other_storage = Storage(dict(a=1, b=2)) # convert dictionary to
   Storage
```

`request` は以下の項目/属性を持ちます。その内の幾つかはまた、`Storage` クラスのインスタンスです：

- `request.cookies`: HTTP リクエストで渡されたクッキーを含む、`Cookie.SimpleCookie()` オブジェクトです。クッキーの辞書のように動作します。各クッキーは、`Morsel` オブジェクトです。
- `request.env`: コントローラに渡される環境変数を含む、`Storage` オブジェクトです。HTTP リクエストからの HTTP ヘッダ変数と、標準の WSGI パラメータを含みます。環境変数は全て小文字に変換され、記憶しやすいようにドットはアンダースコアに変換されます。
- `request.application`: リクエストされたアプリケーションの名前です (`request.env.path_info` から解析)。
- `request.controller`: リクエストされたコントローラの名前です (`request.env.path_info` から解析)。
- `request.function`: リクエストされた関数の名前です (`request.env.path_info` から解析)。
- `request.extension`: リクエストアクションの拡張子です。デフォルトは”html”です。もしコントローラの関数が辞書を返し、さらにビューが指定されなかった場合、辞書をレンダリングするビューファイルの拡張子を決めるために利用されます (`request.env.path_info` から解析)。
- `request.folder`: アプリケーションのディレクトリです。例えばアプリケーションが”welcome”的の場合、`request.folder` には”/path/to/welcome”と絶対パスがセットされます。プログラムにて、アクセスが必要なファイルへのパスを作成するには、この変数と `os.path.join` 関数を常に使用すべきです。`web2py` は常に絶対パスを使用していますが、スレッドセーフの方法ではない

ですので、現在の作業フォルダは(何であれ)決して変更しないということが良いルールです。

- `request.now`: 現在のリクエストの日時を保存した `datetime.datetime` オブジェクトです。
- `request.utcnow`: 現在のリクエストの UTC 日時を保存した `datetime.datetime` オブジェクトです。
- `request.args`: コントローラの関数名の後に続く、URL パスの構成要素のリストです。`request.env.path_info.split('/')[3:]` と等しいです。
- `request.vars`: HTTP GET と HTTP POST のクエリ変数を含む `gluon.storage.Storage` オブジェクトです。
- `request.get_vars`: GET のクエリ変数のみを含む `gluon.storage.Storage` オブジェクトです。
- `request.post_vars`: POST のクエリ変数のみを含む `gluon.storage.Storage` オブジェクトです。
- `request.client`: クライアントの IP アドレスです。存在する場合、`request.env.http_x_forwarded_for` から、そうでない場合は、`request.env.remote_addr` から決定します。これは便利ですが、`http_x_forwarded_for` は偽装することができるため、信頼すべきものではありません。
- `request.is_local`: `True` の場合、クライアントがローカルホストです。`False` なら他の状態です。プロキシが `http_x_forwarded_for` をサポートしていれば、プロキシを通して機能します。
- `request.body` リクエストのボディが含まれている、読み取り専用ファイルストリームです。これは `request.post_vars` を取得するために自動で解析され、解析後はストリームが巻き戻ります。`request.body.read()` で読み取ることができます。
- `request.ajax` は呼び出された関数が、Ajax リクエストを介したものだと `True` になります。
- `request.cid` は Ajax リクエストを生成した、(もしあれば) コンポーネントの `id` です。コンポーネントの詳細については、12 章を参照してください。

- `request.restful` これは新しく便利なデコレータです。リクエストを GET/POST/PUSH/DELETE に分離することによって、web2py のアクションのデフォルトの挙動を変更するために使用できます。詳細は 10 章で説明されています。
- `request.user_agent()` はクライアントの `user_agent` フィールドを解析し、辞書の形式でその情報を返します。モバイルデバイスを検出するのに便利です。Ross Peoples によって作成された”`gluon/contrib/user_agent_parser.py`”を利用しています。何をしているかを確認するには、次のコードをビューに埋め込んでみてください：

```
1 {{=BEAUTIFY(request.user_agent())}}}
```

- `request.wsgi` はアプリケーション内部から、サードパーティの WSGI アプリケーションを呼び出せるようにするフックです。

最後のものは、以下のものも含みます：

- `request.wsgi.environ`
- `request.wsgi.start_response`
- `request.wsgi.middleware`

これらの使用法は、この章の最後に説明します。

例として、典型的なシステム上の次の呼び出しは：

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

次のような `request` オブジェクトになります：

```
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.is_local / False request.is_https / False request.ajax / False request.cid / None 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.web2py_version   / Version
1.99.1 request.env.web2py_runtime_gae / (optional, defined only if
GAE detected) 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

```

どのような環境変数が実際に定義されているかは、web サーバーによって異なります。ここでは、組み込み Rocket wsgi のサーバーを想定しています。変数のセットは Apache の web サーバーを使用した場合でも、大きな違いはありません。

`request.env.http_*`変数は、HTTP リクエストヘッダから解析されたものです。

`request.env.web2py_*`変数は、web サーバー環境から解析されたものではありません。web2py の実行場所やバージョン、あるいは Google App Engine 上で動いているかどうか（個別の最適化が必要かもしれないため）ということをアプリケーションが知るために、web2py によって作成されます。

また、`request.env.wsgi_*`変数もあります。これらは WSGI アダプタに固有です。

#### 4.8 response

`response` は、もう 1 つの `Storage` インスタンスです。以下のものを格納しています：

`response.body`: web2py が出力ページの `body` を書き込む、`StringIO` オブジェクトです。この変数は決して変更しないでください。

`response.cookies`: `request.cookies` と似ていますが、後者はクライアントからサーバーに送られるクッキーを格納するのに対し、前者はサーバーからクライアントに送られるクッキーを格納します。セッションクッキーは自動的に処理されます。

`response.download(request, db)`: アップロードされたファイルのダウンロードを可能にする、コントローラ関数の実装に使用するメソッドです。`request.download` は、`request.args` の最後の `arg` を、エンコードされたファイル名(すなわち、アップロード時に生成され、アップロードフィールドに保存されるファイル名)として受け取ります。エンコードされたファイル名から、アップロードフィールド名とテーブル名がオリジナルのファイル名と同様に抽出されます。`response.download` は 2 つのオプション引数を取ります: `chunk_size` は、ストリームをチャンクするためのバイトサイズをセットします(デフォルトは 64K)。`attachments` は、ダウンロードファイルが添付ファイルとして扱われるか、そうでないかを決めます(デフォルトは `True`)。なお、`response.download` は、`db` のアップロードフィールドに連携しているファイルをダウンロードするのに特化しています。他のタイプのファイルをダウンロードおよびストリーミングをするには、`response.stream`(後述) を用いてください。また、/static フォルダにアップロードされたファイルにアクセスするのに、`response.download` を必ずしも使う必要はありません。静的ファイルは、URL(/app/static/files/myfile.pdf など) を介して直接アクセスすることができます(一般にそうすべきです)。

`response.files`: ページに必要な CSS と JS の一覧です。これらは、インクルードした”web2py\_ajax.html”を介して、標準の”layout.html”のヘッダーに自動でリンクされます。新しい CSS や JS ファイルを含めるには、このリストに追加するだけ十分です。重複を正しく処理します。順序は重要です。

`response.include_files()` は、全ての `response.files` をインクルードする、html の `head` タグを生成します(”views/web2py\_ajax.html”で使われています)。

`response.flash:` ビューに含まれるかもしれないオプションのパラメータです。通常は、何かが発生したことをユーザに通知するために使用します。

`response.headers:` HTTP レスポンスヘッダのための dict(辞書)です。web2py はデフォルトで、"Content-Length"、"Content-Type"、"X-Powered-By"(web2py に等しくセットする)などを含む、幾つかのヘッダをセットします。web2py はまた、クライアント側のキャッシュが有効になっている静的ファイルへのリクエストを除いて、クライアント側のキャッシュを防ぐために、"Cache-Control"、"Expires" および "Pragma" をヘッダにをセットします。web2py がセットしたヘッダは、上書きもしくは削除ができます。さらに新しいヘッダを追加することも可能ですが(例えば、`response.headers['Cache-Control'] = 'private'`)。

`response.menu:` ビューに含まれるかもしれないオプションのパラメータです。通常は、ビューにナビゲーションメニューツリーを渡すために使用します。これは、MENU ヘルパーでレンダリングすることができます。

`response.meta:` `response.meta.author`、および/または、`response.meta.description`、`response.meta.keywords` のような、オプションのメタ情報からなる(辞書のような)ストレージオブジェクトです。各々のメタ変数の内容は、自動的に正しい META タグに入れられます。これはデフォルトの"views/layout.html" に含まれている、"web2py\_ajax.html" のコードによって行われます。

`response.include_meta()` は、シリализされた全ての `response.meta` ヘッダを、含んだ文字列を生成します("views/web2py\_ajax.html" で使用されます)。

`response.postprocessing:` これは関数のリストです。デフォルトは空です。これらの関数は、ビューによって出力がレンダリングされる前に、アクション出力でのレスポンスオブジェクトを、フィルタするために使用されます。これは、他のテンプレート言語に対するサポートを実装するために使用可能です。

`response.render(view, vars):` コントローラ内のビューを、明示的に呼び出すために使用するメソッドです。`view` はオプションのパラメータで、ビューファイルの名前を指定します。`vars` は、ビューに渡される名前付きの値の辞書です。

`response.session_file:` セッションを含むファイルストリームです。

`response.session_file_name`: セッションが保存されるファイルの名前です。

`response.session_id`: 現在のセッションの ID です。ID は自動で決定されます。この変数は決して変更しないでください。

`response.session_id_name`: このアプリケーションのセッションクッキーの名前です。この変数は決して変更しないでください。

`response.status`: レスポンスに渡される HTTP ステータスコードの数字です。デフォルトは 200(OK) です。

`response.stream(file, chunk_size, request=request)`: コントローラがこれを返す時、web2py はファイルの内容ストリームし、`chunk_size` のプロックごとにクライアントに戻します。`request` パラメタは、HTTP ヘッダでチャンクの開始に使用するため必要です。前述したように `response.download` は、アップロードフィールドを介して、保存されたファイルを取り出すために使用されるべきです。`response.stream` は他のケース、例えば、一時ファイルや、コントローラで作成された `StringIO` オブジェクトを返すために使用されます。`response.download` と異なり、`response.stream` は `Content-Disposition` ヘッダを自動的にセットしません。したがって、それは手動で行う必要があります(例、添付ファイルとしてダウンロードすることを指定し、ファイル名を付与するなど)。ただし、`Content-Type` は(ファイル名の拡張子に従って)自動的にセットされます。

`response.subtitle`: ビューへの組み込みを行うことができる、オプションパラメータです。ページのサブタイトルを設定します。

`response.title`: ビューへの組み込みを行うことができる、オプションパラメータです。ページのタイトルを設定し、ヘッダの HTML タイトル TAG によってレンダリングされます。

`response.toolbar`: デバッグ用のツールバーをページに埋め込む関数です  
`\{{=response.toolbar()}\}`。ツールバーは、`request`、`response`、`session` の変数や、各クエリのデータベースアクセス時間を表示します

`response._vars`: この変数は、アクションではなく、ビューにおいてのみアクセス可能です。アクションがビューに対して返す値が入ります。

`response.optimize_css`: "concat,minify,inline" とセットすると、web2py に含まれている CSS ファイルを、連結・圧縮・インライン化できます

`response.optimize_js: "concat,minify,inline"` と設定すると、web2py に含まれている JS ファイルを連結・圧縮・インライン化できます。

`response.view`: ビューのテンプレートの名前です。このテンプレートはページを必ずレンダリングします。デフォルトでは次のように設定されています：

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

上記のファイルがない場合は、次のようになります。

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

特定のアクションに関連付けられたビューファイルを変更するには、この変数の値を変更します。

`response.xmlrpc(request, methods)`: コントローラがこれを返す時、この機能は、XML-RPC [46] を介してメソッドを公開します。10 章で説明するより良いメカニズムが使用可能ですが。このため、この機能の利用は推奨されていません。

`response.write(text)`: 出力ページのボディに、テキストを書き込むメソッドです。

`response.js` は、Javascript のコードを含めることができます。このコードは 13 章で説明するように、レスポンスが、web2py コンポーネントによって受信される場合にのみ実行されます。

`response` は `gluon.storage.Storage` オブジェクトなので、ビューに渡したい他の属性も格納することができます。技術的な制約はありませんが、全ページによってレンダリングされる全体用のレイアウト ("layout.html")、上の変数だけを格納することを推奨します。

いずれにせよ、次のリストの変数を利用することを強くお勧めします：

```
1 response.title
2 response.subtitle
3 response.flash
4 response.menu
5 response.meta.author
6 response.meta.description
7 response.meta.keywords
8 response.meta.*
```

これにより、web2py に同梱された標準の”layout.html” ファイルを、同じ変数セットを利用する別のレイアウトファイルに置き換えることが容易になります。

古いバージョンの web2py は `response.meta.author` の代わりに `response.author` を使っていました。他の meta 属性も同様です。

#### 4.9 session

`session` は、もう 1 つの `Storage` インスタンスです。`session` に格納したものは何でも、例えば次のようにセットした場合：

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

後で読み出すことができます。

```
1 a = session.myvariable
```

同じユーザの同じセッションの範囲内である限り、コードが実行されます（ユーザがセッションクッキーを削除せず、セッション期限が切れていない条件下です）。`session` は `Storage` オブジェクトですので、存在しない属性/キーのセットにアクセスしても例外は発生しません。代わりに `None` が返されます。

セッションオブジェクトには、3 つの重要なメソッドがあります。1 つは `forget` です：

```
1 session.forget(response)
```

これは、web2py にセッションを保存しないよう指示します。アクションが頻繁に呼ばれ、さらにユーザの活動を追跡する必要がないコントローラで使用されます。`session.forget()` は、例えセッションオブジェクトが変更されたとしても、セッションファイルに対する書き込みを防ぎます。`session.forget(response)` はさらに、セッションファイルをアンロックしクローズします。セッションは変更されない限り保存されないので、このメソッドを呼び出すことはほとんどありません。しかし、ページが同時に複数の Ajax リクエストを行う場合、Ajax 経由で呼び出されたアクションが、`session.forget(response)` を呼び出すことは良いアイデアです（このアクションはセッションを必要としていないと仮定しています）。そうでない場合、個々の Ajax のアクションは処理開始前に、前のアクションが完了するまで（そしてセッションファイルをアンロックするまで）

待つ必要があり、ページのローディングが遅くなります。ただし、セッションがデータベースに格納されている場合はロックされません。

もう1つはメソッドは:

```
1 session.secure()
```

です。これは web2py に、セッションクッキーをセキュアクッキーにセットするよう指示します。アプリが https で動いている場合は、これをセットすべきです。セッションクッキーをセキュアにすることで、https 接続でない場合にサーバーは、ブラウザがセッションクッキーをサーバーに送信しないように求めます。

もう1つのメソッドは、connect です :

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

ここで、db は (DAL によって返される) 開いているデータベース接続の名前です。これは web2py に、ファイルシステムではなくデータベースにセッションを保存するよう指示します。session.connect は、db=DAL(...) の後に呼び出す必要がありますが、セッションを要求する全てのロジックの前、例えば Auth の設定前に、呼び出す必要があります。web2py は、次のテーブルを作成します：

```
1 db.define_table('web2py_session',
2                 Field('locked', 'boolean', default=False),
3                 Field('client_ip'),
4                 Field('created_datetime', 'datetime', default=now),
5                 Field('modified_datetime', 'datetime'),
6                 Field('unique_key'),
7                 Field('session_data', 'text'))
```

そして、cPickle したセッションを session\_data フィールドに格納します。

masterapp=None オプションはデフォルトでは、動作中のアプリケーション、つまり request.application の名前を持つアプリケーションに対し、既存のセッションを取得するよう web2py に指示します。

複数のアプリケーションでセッションを共有したい場合は、masterapp にマスターとなるアプリケーションの名前をセットします。

アプリケーションの状態は、request、session、response のシステム変数をプリントすることで、いつでもチェックすることができます。それを行う1つの方法は、次のような専用アクションを作成することです：

```
1 def status():
```

```
2     return dict(request=request, session=session, response=response)
```

#### 4.9.1 セッションの分割

ファイルシステム上にセッションを格納しており、それらの多くを使用している場合、ファイルシステムのアクセスがボトルネックになる可能性があります。解決策の1つは次のようにすることです：

```
1 session.connect(request, response, separate=True)
```

`separate=True` にすると、”sessions/” フォルダではなく、”sessions/” フォルダのサブフォルダにセッションが格納されるようになります。サブフォルダは自動で作成されます。同じプレフィックスのセッションは、同じサブフォルダに格納されます。またこれは、セッションを要求するどのロジックよりも前に、呼び出す必要があることを注意してください。

#### 4.10 cache

`cache` は、web2py の実行環境でも使用可能なグローバルオブジェクトです。これは次の2つの属性を持っています：

- `cache.ram`: メインメモリ上のアプリケーションキャッシュです。
- `cache.disk`: ディスク上のアプリケーションキャッシュです。

`cache` は呼び出し可能であり、アクションやビューをキャッシュするための、デコレータとして使うことができます。

次の例では、`time.ctime()` 関数を RAM 上でキャッシュしています：

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

`lambda: time.ctime()` の出力結果は、RAM 上で5秒間キャッシュされます。文字列 `'time'` はキャッシュのキーとして使用されます。

次の例は、`time.ctime()` 関数をディスク上でキャッシュします：

```

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

```

lambda: time.ctime() の出力結果は、(shelve モジュールを用いて) ディスク上で 5 秒間キャッシュされます

cache.ram と cache.disk の第 2 引数は、関数もしくは呼び出し可能オブジェクトでなければなりません。関数の出力ではなく、既存のオブジェクトをキャッシュしたい場合は、ラムダ関数からそのオブジェクトを返すようにしてください:

```
1 cache.ram('myobject', lambda: myobject, time_expire=60*60*24)
```

次の例は、time.ctime() 関数を RAM とディスクの両方でキャッシュします:

```

1 def cache_in_ram_and_disk():
2     import time
3     t = cache.ram('time', lambda: cache.disk('time',
4                     lambda: time.ctime(), time_expire=5),
5                     time_expire=5)
6     return dict(time=t, link=A('click me', _href=request.url))

```

lambda: time.ctime() の出力結果は 5 秒間、(shelve モジュールを用いて) ディスク上にキャッシュされ、続いて RAM 上でもキャッシュされます。web2py は最初に RAM を検索し、見つからない場合はディスクを検索します。RAM とディスクのいずれにも存在しない場合、lambda: time.ctime() が実行され、キャッシュが更新されます。この手法はマルチプロセス環境で有用です。2 つの time は同じである必要はありません。

次の例は、コントローラの関数の結果を RAM 上にキャッシュします(ビューはキャッシュしません)。

```

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

```

cache\_controller\_in\_ram によって返される辞書は、RAM 上に 5 秒間キャッシュされます。注意として、データベースの選択結果は最初に、シリアル化されなければキャッシュすることができます。より良い方法は、select メソッドの cache 引数を使用し、データベースのセレクト結果を直接キャッシュすることです。

次の例は、コントローラの関数の結果をディスクにキャッシュします（ビューはキャッシュしません）。

```
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))
```

`cache_controller_in_disk` によって返される辞書は、ディスク上に 5 秒間キャッシュされます。なお、pickle 化できないオブジェクトを含む辞書は、キャッシュできないことに注意してください。

ビューをキャッシュすることも可能です。仕組みは、コントローラで文字列を返すようにするために、コントローラ関数でビューをレンダリングします。これは、`response.render(d)` を返すことで実行されます。ここで `d` は、ビューに渡す予定であった辞書です。以下の例では、コントローラ関数の出力（レンダリングされたビューを含む）を RAM でキャッシュしています：

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

`response.render(d)` はレンダリングされたビューを文字列で返し、それは 5 秒間キャッシュされます。これは最も最良で最速のキャッシュ方法です。

なお `time_expire` は、リクエストされたオブジェクトが最後にキャッシュに保存された時刻と、現在の時刻を比較するために使用されます。これは将来のリクエストに影響を与えません。またこれは、オブジェクトが保存された時に定めるのではなく、オブジェクトがリクエストされた時に `time_expire` を動的にセットすることを可能にします。例えば：

```
1 message = cache.ram('message', lambda: 'Hello', time_expire=5)
```

この時、次のような呼び出しが、上記の呼び出しの 10 秒後に行われることを想像してください：

```
1 message = cache.ram('message', lambda: 'Goodbye', time_expire=20)
```

`time_expire` は第 2 の呼び出しにおいて 20 秒間と設定され、なおかつ、メッセージが最初に保存されてから 10 秒間しか経過していないので、"Hello" という値はキャッシュから取り出され、さらに "Goodbye" には更新されません。最初の呼び出しでの `time_expire` の値である 5 秒は、第 2 の呼び出しでは何の影響も与えません。

`time_expire=0`(もしくは負の値) にセットすると、キャッシュされた項目を強制的にリフレッシュすることができます(最後に保存してからの経過時間は常に  $> 0$  のため)。また `time_expire=None` とセットすると、保存してからの経過時間に関係なく、常にキャッシュの値を取り出すようにします(`time_expire` が常に `None` ならば、キャッシュされた項目は実質的に期限切れになりません)。

次のようにすると、複数のキャッシュ変数を削除することができます：

```
1 cache.ram.clear(regex='...')
```

ここで、`regex` は、キャッシュから削除したい全てのキーにマッチする正規表現です。また、次のようにして単一の項目をクリアすることができます:

```
1 cache.ram(key, None)
```

ここで、`key` はキャッシュした項目のキーです。

また memcache のような、他のキャッシュメカニズムを定義することも可能でです。Memcache は `gluon.contrib.memcache` を介して利用可能で、14 章で詳しく説明されています。

キャッシュは通常ユーザーレベルではなく、アプリレベルで行われることに注意してください。もし必要なら、例えばユーザー固有の内容をキャッシュするには、ユーザー `ID` を含むキーを選んでください。

## 4.11 URL

URL 関数は、web2py において最も重要な関数の 1 つです。これは、アクションと静的ファイルのための内部 URL のパスを生成します。

例えば、これは:

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

次のようにマッピングされます

```
1 / [application] / [controller] / f
```

ただし、URL 関数の出力は現在のアプリケーションの名前や、呼び出したコントローラ、その他のパラメータに依存します。web2py は、URL マッピングや、URL マッピングのリバースもサポートしています。URL マッピングによって、外部 URL のフォーマットを再定義することができます。URL 関数を全ての内部 URL の生成に使用する場合、URL マッピングに追加や変更を行うことで、web2py アプリケーション内のリンク切れを予防します。

URL 関数に追加のパラメータを渡すことができます。すなわち、URL の特別な項目のパス (args) や、URL のクエリ変数 (vars) といったものです：

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

これは、次のようにマッピングされます

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

args 属性は、web2py によって自動で解析・デコードされ、最後に request.args に格納されます。同様に vars は、解析・デコードされ、request.vars に格納されます。args と vars は、web2py がクライアントのブラウザと情報の交換をするための基本的なメカニズムを提供します。args が 1 つの要素しか含まない場合、リストにして渡す必要はありません。

URL 関数は、他のコントローラやアプリケーションの URL を生成するためにも使用することができます：

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

これは、次のようにマッピングされます

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

アプリケーション、コントローラ、関数を、名前付き引数で指定することも可能です：

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

アプリケーションの名前がない場合は、現在のアプリを想定します。

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

コントローラの名前がない場合は、現在のものを想定します。

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

コントローラ関数の名前を渡す代わりに、関数自身を渡すことも可能です。

```
1 URL(f)
```

上記の理由から、アプリケーションの静的ファイルの URL を生成するために、常に URL 関数を使用する必要があります。静的ファイルは、アプリケーションの static サブフォルダに保存されています（管理インターフェースを使ってアップロードすることができる場所です）。web2py は仮想的な、'static' コントローラを提供しています。それによって、static サブフォルダからファイルが取り出され、content-type が決められ、クライアントにファイルがストリームされます。次の例では、"image.png" という静的ファイルに対する URL を生成しています：

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

これは、次のようにマッピングされます

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

静的ファイルが static フォルダのサブフォルダに入っている場合、サブフォルダをファイル名の一部として含むことができます。例えば、次のように生成します：

```
1 /[application]/static/images/icons/arrow.png
```

次のように利用してください：

```
1 URL('static', 'images/icons/arrow.png')
```

args と vars の引数は、エンコード/エスケープする必要はありません。自動的で行われます。

デフォルトでは、現在のリクエストに対応する拡張子（request.extension で見つかります）は、request.extension がデフォルトの html でない限り、関数に追加されます。これを書き換えるには、URL(f='name.ext') のように関数名の一部として拡張子を明示的に含めるか、次のように拡張子の引数を指定します：

```
1 URL(..., extension='css')
```

現在の拡張子を、明示的に抑制するには次のようにします：

```
1 URL(..., extension=False)
```

#### 4.11.1 絶対 URL

デフォルトでは、URL は相対 URL を生成します。しかし scheme や host 引数を指定することで、絶対 URL を生成することができます（これは、例えば、email のメッセージに URL を挿入する時などに便利です）。

```
1 URL(..., scheme='http', host='www.mysite.com')
```

引数を True にセットすることで簡単に、現在のリクエストのスキーマとホストを自動的に含めることができます。

```
1 URL(..., scheme=True, host=True)
```

URL 関数はサーバーのポートを指定する port 引数も、必要な場合は受け取ることができます。

#### 4.11.2 デジタル署名つき URL

URL を生成時に、デジタル署名を利用することも可能です。これによりサーバーで検証することができる、\_signature という GET 変数が追加されます。またこれを実施するには、2通りの方法が存在します。

URL 関数に次の引数を渡すことができます：

- hmac\_key: URL に署名するためのキー（文字列）です。
- salt: 署名前のデータをソルト（salt）するための、オプションの文字列です。
- hash\_vars: 署名に含まれるクエリ文字列（つまり GET 変数）からの、オプションの変数名のリストです。True（デフォルト）をセットすると全ての変数を含むことができ、False だと変数を全く含まないようになります。

以下は使用例です：

```
1 KEY = 'mykey'
2
3 def one():
4     return dict(link=URL('two', vars=dict(a=123), hmac_key=KEY)
5
6 def two():
7     if not URL.verify(hmac_key=KEY): raise HTTP(403)
8     # do something
9     return locals()
```

こうすることで `two` アクションは、デジタル署名つきの URL からしかアクセスできなくなります。署名つき URL は次のようにになります:

```
1 '/welcome/default/two?a=123&_signature=4981
bc70e13866bb60e52a09073560ae822224e9'
```

ここでデジタル署名は、`URL.verify` 関数によって、認証されていることに注目してください。`URL.verify` はまた、上述の `hmac_key`、`salt`、`hash_vars` 引数を取ります。それらの値は、URL 確認用のデジタル署名を作成した時に、`URL` 関数に渡した値と一致しなければなりません。

2番目の、より洗練されており、しかしながら、より一般的なデジタル署名の URL は、Auth と一緒に使用するものです。これには、次の例を使って説明するのがベストです:

```
1 @auth.requires_login()
2 def one():
3     return dict(link=URL('two', vars=dict(a=123), user_signature=True)
4
5 @auth.requires_signature()
6 def two():
7     # do something
8     return locals()
```

この場合、`hmac_key` は自動で生成され、セッションにおいて共有されます。これにより、`two` アクションが `one` アクションに、アクセスコントロールを委ねることを可能にします。リンクが生成され署名されているならば有効で、そうでないならば無効です。リンクが他のユーザーに盗まれている場合は、このリンクは無効になります。

デジタル署名つきの Ajax コールバックを、常に利用することは良いプラクティスです。LOAD 関数を利用する場合も、`user_signature` 引数があり、この目的のために使用することができます:

```
1 {{=LOAD('default', 'two', vars=dict(a=123), ajax=True, user_signature=True)}}
```

#### 4.12 HTTP と redirect

`web2py` は `HTTP` という、新しい例外を 1つだけ定義しています。この例外は、モデル、コントローラ、ビューのどこでも、次のコマンドで発生させることができます:

きます：

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

これによりフロー制御は、ユーザのコードからジャンプして web2py に戻り、次のような HTTP レスポンスを返します：

```
1 HTTP/1.1 400 BAD REQUEST
2 Date: Sat, 05 Jul 2008 19:36:22 GMT
3 Server: Rocket WSGI Server
4 Content-Type: text/html
5 Via: 1.1 127.0.0.1:8000
6 Connection: close
7 Transfer-Encoding: chunked
8
9 my message
```

HTTP の最初の引数は、HTTP ステータスコードです。2 番目の引数は、レスポンスのボディとして返される文字列です。その他のオプションの名前付き引数は、HTTP レスポンスヘッダを作成するために使用されます。例えば：

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

これは以下を生成します：

```
1 HTTP/1.1 400 BAD REQUEST
2 Date: Sat, 05 Jul 2008 19:36:22 GMT
3 Server: Rocket WSGI Server
4 Content-Type: text/html
5 Via: 1.1 127.0.0.1:8000
6 Connection: close
7 Transfer-Encoding: chunked
8 test: hello
9
10 my message
```

もし開いているデータベーストランザクションをコミットしたくなければ、例外が発生する前にロールバックしてください。

HTTP 以外のどの例外でも、全ての開いているデータベーストランザクションをロールバックし、エラーのトレースバックをログに保存し、訪問者にチケットを発行し、標準のエラーページが返すよう、web2py を動作させます。

これは HTTP のみが、クロスページのフロー制御に使用できることを意味します。他の例外は、アプリケーションによって捕捉されなければならず、そうしないと web2py によってチケットが発行されます。次のコマンドは：

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

単に次のショートカットです：

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

この `HTTP` を初期化するメソッドの名前付き引数は、HTTP ヘッダのディレクティブに変換されます。この場合は、リダイレクト先になります。`redirect` は、リダイレクトのための HTTP ステータスコード (デフォルトは 303) を、オプションの 2 番目の引数にて受け取ります。この数字を 307 に変更すると一時的なリダイレクトになり、301 に変更すると永久的なリダイレクトになります。

ユーザリダイレクトの最も一般的な使い方は、次のように同じアプリ内の他のページにリダイレクトし、(オプションで) パラメータを渡すことです：

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

#### 4.13 T と国際化

`T` オブジェクトは、言語のトランслレータです。これは `web2py` のクラスである、`gluon.language.translator` の單一のグローバルインスタンスから構成されています。全ての文字列定数は (文字列定数のみが)、次の例のように `T` によってマークされるべきです：

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

`T` によってマークされた文字列は、言語の翻訳が必要なものとして `web2py` によって特定され、(モデル、コントローラ、ビューの) コードが実行された時に翻訳されます。翻訳する文字列が定数でなく変数の場合は、実行時に (GAE を除く)、後で翻訳するため翻訳ファイルにその文字列が追加されます。

`T` オブジェクトは、補間される変数を含む、複数の同等の構文をサポートすることができます。

```
1 a = T("hello %s", ('Tim',))
2 a = T("hello %(name)s", dict(name='Tim'))
3 a = T("hello %s") % ('Tim',)
4 a = T("hello %(name)s") % dict(name='Tim')
```

後の構文は翻訳がより簡単になるので、推奨されています。最初の文字列はリクエストされた言語のファイルに従って翻訳され、`name` 変数が言語とは独立して置換されます。

翻訳された文字列と通常の文字列を、連結することは可能です：

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

しかし、逆は不可能です：

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

次のようなコードも可能で、多くの場合、望ましいです：

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

もしくは代替構文で

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

両者ともに、変数名が”%(name)s”の位置で置換される前に、翻訳が行われます。次の代替は使用すべきではありません：

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

なぜなら、翻訳が置換の後に行われるからです。

リクエストされる言語は、HTTP ヘッダにある”Accept-Language” フィールドによって決められます。この選択は、次のように特定のファイルを要求することによって、プログラムで上書きされる可能性があります。

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

これは、”languages/it-it.py” の言語ファイルを読み込みます。言語ファイルは管理インターフェースを介して、作成及び編集することができます。

文字列単位で言語を強制することも可能です：

```
1 T("Hello World", language="it-it")
```

次のようにして、翻訳を完全に無効にすることもできます。

```
1 T.force(None)
```

通常での文字列の変換は、ビューがレンダリングされる時に遅延評価されます。したがって、翻訳オブジェクトの `force` メソッドは、ビュー内で呼び出してもいけません。

遅延評価は、次のように無効化することが可能です

```
1 T.lazy = False
```

これにより文字列は、現在の容認されている、もしくは強制された言語に基づいて、`T` 演算子により直ちに翻訳されます。

個々の文字列に対して、遅延評価を無効にすることも可能です。

```
1 T("Hello World", lazy=False)
```

次のような、一般的な問題があります。元のアプリケーションが、英語で書かれていたとします。翻訳ファイル(例えばイタリア語、”it-it.py”)があり、かつ、HTTP クライアントが英語(en)とイタリア語(it-it)の順序で、受け入れることを宣言していると想定してください。この場合、次のような望ましくない状況が発生します。つまり、web2py はデフォルトが英語で書かれていることは知りません。したがって、イタリア語の翻訳ファイルしか見つからないので、全てをイタリア語(it-it)に翻訳するようにします。もし”it-it.py”ファイルが見つからなかったら、デフォルト言語の文字列(英語)が使用されたはずです。

この問題には 2 つの解決方法があります。一つは英語の翻訳ファイルを作成することですが、しかしファイル自体は、冗長で不需要です。より良い方法は web2py に、どの言語を使用すべきか、デフォルト言語の文字列を指示することです。これは次のようにして行うことができます：

```
1 T.set_current_languages('en', 'en-en')
```

これは `T.current_languages` に、翻訳が必要でない言語のリストを格納し、そして言語ファイルのリロードを強制します。

なお”it”と”it-it”は、web2py のビューの観点からすると違う言語になります。それらの両方をサポートするためには、常に小文字の名前を持つ、2 つの翻訳ファイルが必要となります。他の全ての言語についても同様です。

現在受け入れている言語は、次に格納されています

```
1 T.accepted_language
```

`T(...)` は、単に文字列を変換するだけでなく、変数を翻訳できることに注意してください：

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

この場合、翻訳されるのは単語”test”です。しかし、もし翻訳語がファイルに見つからず、ファイルシステムが書き込み可能であれば、言語ファイルにある翻訳対象の単語リストに追加されます。

#### 4.14 クッキー

web2py は、Python のクッキー・モジュールを、クッキー処理のために使用します。

ブラウザからのクッキーは `request.cookies` にあり、サーバーから送られるクッキーは `response.cookies` にあります。

クッキーは次のようにセットすることができます：

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

2 行目は、ブラウザにクッキーを 24 時間保持するように伝えます。3 行目は、現在のドメインの任意のアプリケーション (URL パス) にクッキーを返送するように、ブラウザに指示します。

クッキーは次のように、セキュリティで保護することができます：

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

こうすることで、ブラウザはクッキーを HTTP ではなく HTTPS 経由でのみ返送するようになります。

クッキーは次のように、取り出すことができます：

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

セッションが無効になってない限り、web2py の内部では、次のようにクッキーをセットし、セッション処理のために使用します：

```

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

```

ただし、単一のアプリケーションが複数のサブドメインを含み、セッションをそれらのサブドメインで共有したい場合（例えば、sub1.yourdomain.com, sub2.yourdomain.com など）、セッションクッキーのドメインを次のように明示的にセットしてください：

```

1 if not request.env.remote_addr in ['127.0.0.1', 'localhost']:
2     response.cookies[response.session_id_name]['domain'] = ".yourdomain
        .com"

```

上記の方法は例えば、サブドメイン間でログイン状態を保持したい場合に便利です。

#### 4.15 init アプリケーション

web2py をデプロイする時、デフォルトアプリケーションをセットしたい場合があります。すなわち、次のように URL のパスが空の時に、起動するアプリケーションのことです：

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

デフォルトで空のパスに出くわしたら、web2py は `init` という名のアプリケーションを探します。もし `init` アプリケーションがなかったら、`welcome` と呼ばれるアプリケーションを探します。

`routes.py` の `default_application` の設定で、デフォルトアプリケーションの名前を、`init` から別の名前にすることで変更できます。

```
1 default_application = "myapp"
```

注： `default_application` は web2py のバージョン 1.83 において最初に登場しました。

ここでは、デフォルトアプリケーションを設定する方法は 4 通りあります：

- 作成するデフォルトアプリケーションの名前を `init` とします。
- `routes.py` の `default_application` を、作成するアプリケーションの名前にセットします。

- ”applications/init” から、作成するアプリケーションのフォルダへのシンボリックリンクを作成します。
- 次のセクションで説明する、 URL リライトを使用します。

## 4.16 URL リライト

web2py では、コントローラのアクションを呼び出す前に、着信リクエストの URL パスを書き換えることができます (URL マッピング)。逆に、URL 関数によって生成された URL パスも書き換えることができます (リバース URL マッピング)。これを行う理由の 1 つは、古い仕様の URL を扱うためです。もう 1 つはパスを単純化し、短くするためです。web2py には、2 つの異なる URL リライトシステムを組み込んでいます。多くのユースケースに対して簡単に利用できるパラメタベースのシステムと、より複雑なケースのための柔軟な、パターンベースのシステムです。URL のリライトルールを指定するためには、”web2py” フォルダで、`routes.py` という名の新しいファイルを作成してください (`routes.py` の内容は、次の 2 つのセクションで説明する 2 つのリライトシステムのどちらかを選択したほうに依存します)。2 つのシステムを混在させることはできません。

`routes.py` を編集する場合、リロードする必要があります。これは次の 2 つの方法で行われます。web サーバーをリスタートするか、管理画面のルーティングの再読み込み (*Reload routes*) ボタンをクリックするかです。`routers` にバグがある場合、リロードされません。

### 4.16.1 パラメタベースのシステム

パラメタベース (パラメトリック) のルーターは、幾つかの”予め準備された” の URL リライトメソッドへの、簡単なアクセスを用意します。その機能は次の通りです:

- \* 外部から URL(これらは URL() 関数から作成されたものです) が見えないように、デフォルトのアプリケーション、コントローラ、関数の名前を取り除きます
- \* ドメイン (やポート) を、アプリケーションやコントローラにマッピングします
- \* URL 中に、言語セレクタを埋め込みます

\* 着信 URL から固定のプレフィックスを取り除き、送出 URL に再び付け加えます

\* /robots.txt のようなルートファイルを、アプリケーションの静的ディレクトリの 1 つにマッピングします

パラメトリックのルーターは、着信 URL へのより柔軟なバリデーションを提供します。

myapp アプリケーションを作成し、アプリケーション名をユーザーが参照する URL の一部にしないように、デフォルトにすることを希望しているとします。デフォルトのコントローラはまだ default で、それもユーザーが参照する URL から、同様に取り除きたいとします。この場合、routes.py に次の記述を入れてください:

```
1 routers = dict(
2     BASE  = dict(default_application='myapp'),
3 )
```

これだけです。パラメトリックルーターは、次のような URL に対して何が正しいのか知っており、適切に対応します:

```
1 http://domain.com/myapp/default/myapp
```

もしくは

```
1 http://domain.com/myapp/myapp/index
```

これらは、通常の省略では曖昧さが残るものです。myapp と myapp2 という 2 つのアプリケーションがある場合、同様の効果が得られます。さらに、myapp2 のデフォルトのコントローラは安全な時(大抵の場合)は、いつでも URL から取り除かれます。

もう 1 つの例を示します。URL は次のように、URL ベースの言語をサポートするとします:

```
1 http://myapp/en/some/path
```

もしくは(書き換え)

```
1 http://en/some/path
```

これは次のようにします:

```

1 routers = dict(
2     BASE = dict(default_application='myapp'),
3     myapp = dict(languages=['en', 'it', 'jp'], default_language='en'),
4 )

```

この時、着信 URL が次のような場合:

```
1 http://domain.com/it/some/path
```

/myapp/some/path にルーティングされます。そして、request.uri\_language は'it' にセットされ、翻訳を強制することができます。言語固有の静的ファイルを持つことも可能です。

```
1 http://domain.com/it/static/filename
```

これは次にマッピングされます:

```
1 applications/myapp/static/it/filename
```

ただし、ファイルが存在する場合に限ります。存在しない場合、次のような URL は:

```
1 http://domain.com/it/static/base.css
```

今までどおり、次にマッピングされます:

```
1 applications/myapp/static/base.css
```

(なぜなら static/it/base.css が存在しないからです)

したがって必要なら、画像などの言語固有の静的ファイルをもつことができます。ドメインのマッピングも同様にサポートしています:

```

1 routers = dict(
2     BASE = dict(
3         domains = {
4             'domain1.com' : 'app1',
5             'domain2.com' : 'app2',
6         }
7     ),
8 )

```

これは期待通りに動作します。

```
1 routers = dict(
2     BASE = dict(
```

```

3     domains = {
4         'domain.com:80' : 'app/insecure',
5         'domain.com:443' : 'app/secure',
6     }
7 ),
8 )

```

これは、`http://domain.com`へのアクセスを、`insecure`という名のコントローラにマッピングする一方、`HTTPS`でのアクセスは、`secure`というコントローラにアクセスさせます。また同様に、異なるポートを異なるアプリケーションへマッピングすることも可能です。

詳細な情報は、標準の web2py 配布のベースフォルダにある `router.example.py` ファイルを調べてください。

注: *parameter-based* システムは、web2py のバージョン 1.92.1において最初に登場しました。

#### 4.16.2 パターンベースのシステム

先に説明したパラメタベースのシステムは、ほとんどの場合で十分なものです。しかし代わりにパターンベースシステムを使用する場合は、より複雑なケースに対する幾つかの追加の柔軟性を提供します。パラメータベースのシステムを利用するには、ルーターをルーティングパラメタの辞書として定義する代わりに、タプルの組からなる `routes_in` と `routes_out` という 2 つのリスト(もしくはタプル)を定義します。各タプルは 2 つの要素を保持します。これは、置換されるパターンとそれを置換する文字列です。例えば:

```

1 routes_in = (
2     ('/testme', '/examples/default/index'),
3 )
4 routes_out = (
5     ('/examples/default/index', '/testme'),
6 )

```

これらのルーティングによって、次の URL は :

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

次にマッピングされます :

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

訪問者には、ページの URL への全リンクは /testme のように見えます。

パターンは Python の正規表現と同じ構文を持っています。例えば：

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

これは ".php" で終わる全ての URL が、index ページにマッピングされます。

1 つのアプリケーションしか公開予定がない場合、時にはアプリケーションのプレフィックスを URL から取り除きたい場合があります。これは次のように、実現できます：

```
1 routes_in = (
2   ('/(?P<any>.* )', '/init/\g<any>'),
3 )
4 routes_out = (
5   ('/init/(?P<any>.* )', '/\g<any>'),
6 )
```

上記の正規表現と混ぜることができます。もう 1 つの別の構文があります。これは、 (?P<name>\w+) や \g<name> の代わりに \$name を使用します。例えば：

```
1 routes_in = (
2   ('/$c/$f', '/init/$c/$f'),
3 )
4 routes_out = (
5   ('/init/$c/$f', '/$c/$f'),
6 )
7 )
```

これは、 "/example" アプリケーションのプレフィックスを、全ての URL で取り除きます。

\$name 表記法を使用し、 routes\_in から routes\_out へ自動的にマッピングすることができます。ただしこの場合、正規表現を使用することはできません。例えば次のようになります：

```
1 routes_in = (
2   ('/$c/$f', '/init/$c/$f'),
3 )
4 routes_out = [(x, y) for (y, x) in routes_in]
```

ここで複数のルーティングがある場合、最初にマッチした URL が実行されます。もしマッチするパターンがない場合、パスはそのままになります。

`$anything` を使うと、行の最後までに何か `(.*)` をマッチさせることができます。

ここでは、favicon と robots リクエストを処理するための最小限の”routes.py”を示します：

```
1 routes_in = (
2     ('/favicon.ico', '/examples/static/favicon.ico'),
3     ('/robots.txt', '/examples/static/robots.txt'),
4 )
5 routes_out = ()
```

さらに複雑な例を示します。”myapp” という単一のアプリを、不必要的プレフィックスなしに公開しますが、同時に、`admin` と `appadmin` 及び `static` も公開するという例です。

```
1 routes_in = (
2     ('/admin/$anything', '/admin/$anything'),
3     ('/static/$anything', '/myapp/static/$anything'),
4     ('/appadmin/$anything', '/myapp/appadmin/$anything'),
5     ('/favicon.ico', '/myapp/static/favicon.ico'),
6     ('/robots.txt', '/myapp/static/robots.txt'),
7 )
8 routes_out = [(x, y) for (y, x) in routes_in[:-2]]
```

ルーティングの一般的な構文は、これまで見てきた簡単な例よりも複雑です。ここでは、より一般的で代表的な例を示します：

```
1 routes_in = (
2     ('140\.191\.\\d+\\.\\d+:https://www.web2py.com:POST /(?P<any>.*\\.php',
3      '/test/default/index?vars=\\g<any>'),
4 )
```

これは、`https` の `POST` リクエストを、次の正規表現にマッチしたリモート IP から、`www.web2py.com` というホストにマッピングします。

```
1 '140\.191\.\\d+\\.\\d+'
```

そして、次の正規表現にマッチしたページを

```
1 '/(?P<any>.*\\.php'
```

次へリクエストします。

```
1 '/test/default/index?vars=\\g<any>'
```

ここで `\g<any>` は、マッチした正規表現によって置換されます。

全般的な構文は以下の通りです：

```
1 '[remote address]:[protocol]://[host]:[method] [path]'
```

式全体は正規表現としてマッチするので、””は常にエスケープされ、マッチしたなどの部分式もPythonの正規表現の構文に従って、(?)P<....>....を使用し捉えることができます。

これにより、クライアントのIPアドレス、ドメイン、リクエストのタイプ、メソッド、パス、などに基づいて、リクエストを再ルーティングすることが可能になります。また、異なるバーチャルホストを、異なるアプリケーションにマッピングすることも可能です。マッチした部分式はターゲットのURLを構築するために使用することができ、結果的に、GET変数に渡すことができます。

Apacheやlighttpdなどの全ての主要なWebサーバーは、URLをリライトする機能を持っています。本番環境では、それらはroutes.pyに代わる選択肢になります。いずれにせよ、アプリの内部URLをハードコーディングせず、URL関数で生成することを強く推奨します。これにより、必要であればroutesを変更することで、アプリケーションはよりポータブルなものになります。

### アプリケーション固有のURLリライト

パターンベースのシステムを使用する場合、アプリケーションのベースフォルダにある固有のroutes.pyファイルに、アプリケーションの独自のルーティングを設定することができます。これは、着信URLを元にベースのroutes.pyで確定した、アプリケーション名のroutes\_appを構成することによって有効になります。これが起こると、アプリケーション固有のroutes.pyは、ベースのroutes.pyの代わりに使用されます。

routes\_appのフォーマットは、置換パターンが単純にアプリケーションの名前になることを除いて、routes\_inと全く同じです。routes\_appを着信URLに適用してアプリケーション名にならない場合、または、アプリケーション固有のroutes.pyが見つからない場合、ベースのroutes.pyがこれまで通り使用されます。

注：routes\_appはweb2pyのバージョン1.83で初めて登場しました。

### デフォルトのアプリケーション、コントローラ、関数

パターンベースのシステムを使用する場合、デフォルトのアプリケーション、コントローラ、関数は、routes.pyで適切な値をセットすることで、init、default、

index から違う名前にそれぞれ変更することができます：

```
1 default_application = "myapp"
2 default_controller = "admin"
3 default_function = "start"
```

注：これらの項目は、web2py のバージョン 1.83 で初めて登場しました。

#### 4.17 エラーのルーティング

routes.py を使用し、サーバーにエラーが起きた時、特定のアクションにリクエストを再ルーティングすることも可能です。このようなマッピングを、各アプリケーション、各エラーコード、各アプリケーションのエラーコード、に対してグローバルに指定することができます。以下はその例です：

```
1 routes_onerror = [
2     ('init/400', '/init/default/login'),
3     ('init/*', '/init/static/fail.html'),
4     ('*/404', '/init/static/cantfind.html'),
5     ('*/*', '/init/error/index')
6 ]
```

各タプルに対して、最初の文字列は”[app name]/[error code]” に照合されます。一致した場合、失敗したリクエストは 2 番目の文字列にある URL に再ルーティングされます。エラーハンドリングの URL が静的ファイルでない場合、次のような GET 変数がエラーのアクションに渡されます：

- code: HTTP のステータスコードです (例、404 や 500 など)
- ticket: ”[app name]/[ticket number]” 形式のチケットです (チケットがない場合は”None” になります)
- requested\_uri: request.env.request\_uri と等価です。
- request\_url: request.url と等価です。

これらの変数はエラーを処理するアクションで、request.vars を介して利用可能です。そして、エラーのレスポンスを生成するために使用することができます。とりわけ、デフォルトの 200(OK) ステータスコードの代わりに、元の HTTP エラーコードを返すのは良いアイデアです。これは response.status = request.vars.code とすることによって、実現することができます。エラー

アクションから管理者に、`admin` のチケットへのリンクを含んだメールを送信させることも可能です。

照合できなかったエラーは、デフォルトのエラーページを表示します。デフォルトのエラーページは、カスタマイズすることができます (web2py のルートフォルダにある `router.example.py` と `routes.example.py` を参照してください)

```

1 error_message = '<html><body><h1>%s</h1></body></html>'
2 error_message_ticket = '''<html><body><h1>Internal error</h1>
3     Ticket issued: <a href="/admin/default/ticket/%(ticket)s"
4         target="_blank">%(ticket)s</a></body></html>'''
```

最初の変数は、無効なアプリケーションや関数がリクエストされていた時のエラーメッセージです。2番目の変数は、チケットが発行された時のエラーメッセージが入っています。

`routes_onerror` は両方のルーティングメカニズムで機能します

#### 4.18 バックグラウンドでのタスク実行

web2py では、各々の http リクエストは、それぞれのスレッドで扱われます。スレッドは web サーバーによって、効率化のためにリサイクルされ管理されています。セキュリティのために、web サーバーは各リクエストにタイムアウトを設けています。これはアクションが、時間のかかるタスクを処理すべきでないこと、新規のスレッドを作成すべきでないこと、プロセスをフォーク (これは可能ですが推奨されません) すべきでないこと、を意味します。

時間のかかるタスクを実行する正しい方法は、バックグラウンドで行うことです。その方法は一通りではありませんが、しかしここでは、web2py に組み込まれている 3 つの機構を説明します。これは、`cron`、`homemade task queues`、`scheduler` です。

またここでは、`cron` を、Unix のクーロンシステムではなく、web2py の 1 つの機能として言及します。web2py のクーロンは windows でも動作します。web2py のクーロンは、スケジューリングされた時刻にバックグラウンドでのタスクを必要とし、さらにそれらのタスクが、2 つの呼び出しの間隔時間に比べて、短い時間で処理する場合に有効な方法です。各タスクはそれぞれ独自のプロセスで実行され、複数のタスクは同時に実行されます。しかし、実行するタスクの数を制御する方法はありません。誤ってタスクが自身をオーバーラップすると、データ

ベースロックやメモリの激しい消費を引き起こします。web2py のスケジューラは、別のアプローチを取ります。実行するプロセスの数は固定され、それらは異なるメカニズムで動作します。各プロセスはワーカーと呼ばれます。各ワーカーは、タスクが実行可能な時にピックアップし、スケジュールされた時刻のなるべくすぐ後に実行されます。ただし、必ずしも正確な時刻に実行されるとは限りません。スケジュールされたタスクの数以上に、実行プロセスの数が多くなることはありません。したがって、メモリの激しい消費は起こりません。スケジューラーのタスクはモデルで定義でき、データベースに保存されます。web2py のスケジューラは、分散キューを実装しません。なぜなら、タスクの実行時間に比べ、タスクを分散する時間は無視できると想定しているからです。ワーカーはデータベースからタスクをピックアップします。

ホームメードのタスクキューは、幾つかのケースでスケジューラの単純な代替になります。

#### 4.18.1 クーロン

web2py のクーロンは、アプリケーションが予め設定された時刻に、プラットフォームに依存しない形で、タスクを実行できるように提供されています。

各アプリケーションでの、クーロンの機能は次のようなクーロンタブ・ファイルで定義されます:

```
1 app/cron/crontab
```

これは、[45] で定義されている構文に従っています (web2py 独自の拡張が幾つあります)。

このことは、ホスト OS に影響されることなく、全てのアプリケーションが個別にクーロンの設定を持つことができ、クーロンの設定が web2py から変更できることを意味します。

次の例を見てください:

```
1 0-59/1 * * * * root python /path/to/python/script.py
2 30      3 * * * root *applications/admin/cron/db_vacuum.py
3 */30    * * * * root **applications/admin/cron/something.py
4 @reboot root      *mycontroller/myfunction
5 @hourly root     *applications/admin/cron/expire_sessions.py
```

この例の最後の 2 行は、web2py の追加機能を提供するために、標準のクーロン構文を拡張したものを利用しています。

*”applications/admin/cron/expire\_sessions.py”* というファイルは実際に存在し、admin アプリとともに配布されています。これは有効期限の切れたセッションをチェックし、削除します。*”applications/admin/cron/crontab”* は、1 時間毎に実行されます。

もしスクリプト/関数の名前がアスタリスク (\*) で始まり、”.py” で終わる場合、それは web2py の環境で実行されます。これは、全てのコントローラとモデルを自由に使えることを意味します。2 つのアスタリスク (\*\*) を使用した場合、モデルは実行されません。これはオーバーヘッドが少なく、ロックの可能性の問題を回避する、推奨される呼び出し方法です。

ただし、web2py 環境で実行されるスクリプト/関数は、関数の最後にマニュアルでの db.commit() が必要です。そうでない場合、トランザクションが戻されます。

クーロンが動作するシェルモードにおいて、web2py はチケットや意味のあるトレースバックを生成しません。したがって web2py コードが、エラーなしで動作することを、クーロンタスクを設定する前に確認してください。さらに、どのようにモデルを使用しているかに注意してください。これは、タスクの実行は別プロセスで行われますが、データベースのロックを考慮する必要があります。つまり、データベースをロックするクーロンタスクのために、ページが待機するのを避けるためです。クーロンタスクでデータベースを使わない場合は、\*\* 構文を使用してください。

また、コントローラの関数を呼び出すことができます。この場合、パスを指定する必要はありません。コントローラと関数は、呼び出し側のアプリケーションのものです。さらに、上に列挙した注意事項に特に注意を払ってください。例：

```
1 */30 * * * * root *mycontroller/myfunction
```

クーロンタブの最初のフィールドで、@reboot と明記すると、所定のタスクは web2py 起動時に一度だけ実行されます。この特徴を利用して、web2py 起動時にアプリケーションのデータを、事前にキャッシュ、検証、初期化したりすることができます。なお、クーロンタスクはアプリケーションとは並行して実行されます。したがって、もしアプリケーションがクーロンタスクが終わるまで、リクエストを受け取る準備ができる場合、タスクが反映されたかチェックするよう

な実装をするべきです。例：

```
1 @reboot * * * * root *mycontroller/myfunction
```

どのように web2py を起動しているかに応じて、web2py クーロンには 4 つの動作モードがあります。

- ソフトクーロン：全ての実行モード下で利用可能です。
- ハードクーロン：組み込みサーバーを使用している場合に利用可能（直接もしくは Apache の mod\_proxy を介して）です。
- 外部クーロン：システム自身のクーロンサービスにアクセスできる場合に利用可能です。
- クーロンなし、です。

組み込みのウェブサーバーを利用している場合、デフォルトはハードクーロンです。他の全てでは、デフォルトはソフトクーロンです。ソフトクーロンは CGI、FASTCGI、WSGI を使用している場合、デフォルトになります（ただし、ソフトクーロンは web2py が提供する標準の `wsgihandler.py` では、デフォルトでは `enabled` にはなりません）。

タスクは、クーロンタブで指定した時刻の後の、最初の web2py に対する呼び出し（ページロード）で実行されます。ただし、ユーザーへの遅延を発生させないために、ページの処理が終わった後に実行されます。明らかに、タスク実行に関する正確な時刻は、サイトが受け取るトラフィックに依存するため不確実性があります。また、web サーバーがページロードのタイムアウトをセットしている場合、クーロンタスクは中断されるおそれがあります。これらの制限が許容できない場合は、”外部クーロン”を参照してください。ソフトクーロンは妥当で、最後の手段ですが、web サーバーが他のクーロン方式を利用できる場合には、ソフトクーロンよりそちらを利用したほうがよいです。

ハードクーロンは、（直接または Apache の mod\_proxy を介して）組み込みの web サーバーを使用している場合、デフォルトになります。ハードクーロンは並列スレッドによって実行されるので、ソフトクーロンとは異なり、実行に要する時間や、実行する時間の精度に関して制約はありません。

外部クーロンはどのような場合でもデフォルトではありませんが、しかし、システムクーロン機能へのアクセスを持っている必要があります。これは並列プロセスで動作するため、ソフトクーロンで適用されるような制限はありません。

WSGI や FastCGI の下で、クーロンを利用する場合に推奨される方法です。

次のサンプル行は、システムクーロン（通常は /etc/crontab）に追加されるものです：

```
1 0-59/1 * * * * web2py cd /var/www/web2py/ && python web2py.py -J -C -D
   1 >> /tmp/cron.output 2>&1
```

外部クーロンで動すならば、-N コマンドラインパラメータを、web2py の起動スクリプトまたは設定に追加していることを確認してください。複数のタイプのクーロンによる、衝突を回避するためです。また外部のクーロンを利用する場合、上で示したように-J（または--cronjob、でも同じ）を加えていることを確認してください。これにより web2py が、クーロンによって実行されていることを検知することができるからです。ソフトまたはハードクーロンの場合、web2py はこれを内部でセットしています。

特定のプロセスでどんなクーロン機能も必要ない場合は、-N コマンドラインパラメータを使用し、それを無効にすることができます。ただしこれは、幾つかのメンテナンスタスク（自動的なセッションディレクトリのクリーンなど）も、無効にするかもしれないことに注意してください。この機能の最も一般的な用途は以下の通りです：

- すでにシステムから起動される外部クーロンを開始している場合（WSGI の設定では最も一般的）
- アプリケーションをデバッグする際に、アクションや出力をクーロンに干渉されたくない場合

#### 4.18.2 ホームメード・タスクキュー

クーロンは一定の時間間隔で実行するタスクには便利ですが、バックグラウンドタスクを実行するための解決策として常に最良というわけではありません。このため web2py は、どの python スクリプトもコントローラのように実行する機能を提供しています：

```
1 python web2py.py -S app -M -N -R applications/app/private/myscript.py -
   A a b c
```

ここで、-S app は”myscript.py”を”app”として実行することを web2py に指示します。-M はモデルを実行することを指示し、-N はクーロンを動作さ

せないことを指示します。`-A a b c` は、オプションのコマンドライン引数 `sys.argv=['a', 'b', 'c']` を、”myscript.py” に渡します。

このようなタイプのバックグラウンド・プロセスは、クーロンを介して実行すべきではありません（たぶん、`@reboot` を除いて）。なぜなら、同時に 1 つのインスタンスしか実行しないことを、保証する必要があるからです。クーロンでは、あるプロセスがクーロンのイテレーション 1 で始まり、完了する前にクーロンのイテレーション 2 が来る可能性があるからです。さらにクーロンは次々にそれを開始しするので、メールサーバーのようなものを妨害する可能性があります。

8 章において、どのように上記の方法を使ってメール送信するか、例を使って説明します。

#### 4.18.3 スケジューラー（実験的）

`web2py` のスケジューラは前のサブセクションで説明した、タスクキューとともによく似た方法で動作しますが、幾つか違いがあります：

- タスクを作成し、スケジューリングするための標準的なメカニズムを提供します
- 単一のバックグラウンド・プロセスではなく、複数のワーカープロセスからなります
- ワーカーノードのジョブは、監視することができます。なぜなら、それらの状態はタスクの状態とともに、データベースに格納されるからです。
- `web2py` 抜きで動作しますが、ここでは説明しません。

スケジューラはクーロンを使用しませんが、`@reboot` クーロンでワーカーのノードを起動することができます。

スケジューラでは、タスクはモデルに定義された（もしくはモデルからインポートされたモジュールの）単純な関数です。例えば：

```
1 def task_add(a, b):
2     return a+b
```

タスクは、コントローラによって参照されるものと同じ環境下で、呼び出されます。したがって、モデルで定義された全てのグローバル変数、例えばデータベース接続 (`db`) など、を参照することができます。タスクは、HTTP リクエストと

関連付けされていないという点で、コントローラのアクションとは異なります。このため、`request.env` がありません。

タスクが定義されていたら、モデルに次のコードを追加して、スケジューラを有効にする必要があります:

```
1 myscheduler = Scheduler(db, dict(task_add=task_add))
```

`Scheduler` クラスの最初の引数は、スケジューラがワーカーとやり取りするために使用する、データベースにする必要があります。これはアプリケーションの `db`、もしくは、複数のアプリで共有するの専用の `db`、にすることができます。スケジューラは必要なテーブルを作成します。第 2 の引数は `key:value` となる、ペアの Python の辞書です。`key` はタスクを公開するために使用する名前で、`value` はタスクを定義する関数の実際の名前です。

タスクを定義し、`Scheduler` がインスタンス化されたなら、後はワーカーを起動するだけです:

```
1 python web2py.py -K myapp
```

`-K` オプションは、1 つのワーカーを起動します。`-K` オプションの引数は、カンマで区切られたアプリの名前のリストです。これらは、ワーカーによって提供されるアプリです。多数のワーカーを起動することも可能です。

このようにして、所定のインフラを持つことができました。つまり、タスクを定義し、スケジューラにそれらを伝え、ワーカーを起動しました。残りの作業は、実際にタスクをスケジューリングすることです。

タスクはプログラムによって、もしくは `appadmin` を介して、スケジューリングすることができます。実際、タスクは単純に”scheduler\_task” のテーブルに、1 つのエントリーを加えることでスケジューリングされます。これは `appadmin` を介してアクセスできます:

```
1 http://127.0.0.1:8000/scheduler/appadmin/insert/db/scheduler_task
```

このテーブルのフィールドの意味は明白です。”args” と”vars” フィールドは、JSON フォーマットでタスクに渡される値です。上記の”task\_add” の場合、”args” と”vars” の例は、以下のようになります:

```
1 args = [3, 4]
2 vars = {}
```

または

```
1 args = []
2 vars = { 'a':3, 'b':4}
```

タスクは、次の状態の 1 つを取ることができます:

```
1 QUEUED, RUNNING, COMPLETED, FAILED, TIMEOUT
```

タスクが存在し ("scheduler\_task" テーブルにレコードがあれば)、QUEUED の状態で、準備が整っているならば (レコードにある全ての条件が満たされていれば)、タスクはワーカーによってピックアップされるようになります。ワーカーが利用可能になるとすぐに、実行するためにスケジュールされた最初の準備済みタスクをピックアップします。ワーカーは、(スケジューラによって作成された) もう 1 つのテーブル "scheduler\_run" に、エントリーを作成します。

"scheduler\_run" テーブルは、全ての実行タスクの状態を保存します。各レコードはワーカーによってピックアップされた、タスクを参照します。1 つのタスクは、複数のランを持つことができます。例えば、1 時間に 10 回繰り返すようにスケジュールされたタスクは、10 個のランを持つでしょう (1 つが失敗する場合や 1 時間以上かかる場合でない限り)。

取ることのできるランの状態は、次の通りです:

```
1 RUNNING, COMPLETED, FAILED, TIMEOUT
```

QUEUED タスクがピックアップされた場合、それは RUNNING タスクになり、そのランのステータスも RUNNING になります。もしランが完了し、何の例外もなく、タスクのタイムアウトもない場合、ランは COMPLETED にマークされ、タスクも後で実行されるか否かに応じて、QUEUED もしくは COMPLETED にマークされます。

RUNNING タスクに例外を発生した時は、ランは FAILED にマークされ、タスクも FAILED にマークされます。トレースバックはランのレコードに格納されます。

同様にランがタイムアウトを超える場合、それは停止させられ TIMEOUT にマークされ、タスクもまた TIMEOUT にマークされます。

どのような場合でも、stdout はキャプチャされ、ランのレコードにログが保存されます。appadmin を使用すると、全ての RUNNING タスク、COMPLETED タスクの出力、FAILED タスクのエラーなどを、チェックすることができます。

スケジューラは、さらにもう 1 つの”scheduler\_worker” というテーブルを作成します。これはワーカーのハートビートとステータスを保存します。取ることのできるワーカーのステータスは、次の通りです:

```
1 ACTIVE, INACTIVE, DISABLED
```

appadmin を使用し、ワーカーのステータスを変えてワーカーを無効にすることができます。appadmin を介してできることは全て、これらのテーブルにレコード挿入・更新することにより、プログラムからでも可能です。

いずれにせよ、RUNNING のタスクのレコードを変更すべきではありません。予期せぬ挙動をする可能性があります。ベストプラクティスは、”insert” を使用しタスクをキューイングすることです。例えば:

```
1 db.scheduler_task.insert(
2     status='QUEUED',
3     application_name='myapp',
4     task_name='my first task',
5     function_name='task_add',
6     args='[]',
7     vars="{ 'a':3, 'b':4 }",
8     enabled=True,
9     start_time = request.now,
10    stop_time = request.now+datetime.timedelta(days=1),
11    repeats = 10, # run 10 times
12    period = 3600, # every 1h
13    timeout = 60, # should take less than 60 seconds
14 )
```

なお、”times\_run”、”last\_run\_time”、”assigned\_worker\_name” のフィールドは、スケジュールした時点では提供されず、ワーカーによって自動で設定されます。

完了したタスクの出力を、次のようにして取り出すことができます:

```
1 completed_runs = db(db.scheduler_run.status='COMPLETED').select()
```

スケジューラは実験的です。なぜなら、より徹底的なテストが必要で、さらに機能追加される時にテーブルの構造が変更されるかもしれないからです。

- タスクを定義し、(タスクを定義した後に)Scheduler をインスタンス化するために、個別のモデルファイルを持つことを推奨します。
- アプリケーションごとに、少なくとも 1 つのワーカーを持つことを推奨しま

す。より詳細に制御できるからです。ただし、これは厳密に必須というわけではありません。

- タスクを(モデルではなく)モジュールに定義する場合、ワーカーをリストアートする必要があります。

#### 4.19 サードパーティのモジュール

web2pyはPythonで書かれていますので、サードパーティのものも含め、任意のPythonモジュールをインポートして使うことができます。これには、モジュールを見つけるようにすることだけが必要です。他のPythonアプリケーションと同様に、モジュールは公式のPythonの”site-packages”ディレクトリにインストールすることができます。インストールしたモジュールは、コード上の任意の場所からインポートすることができます。

”site-packages”にあるモジュールは、その名が示すように、サイトレベルのパッケージです。モジュールが個別にインストールできないのであれば、site-packagesを必要とするアプリケーションはポータブルではありません。”site-pacakages”にモジュールを持つ利点は、複数のアプリケーションがそれを共有できることです。例えば、”matplotlib”というグラフ描画用のパッケージを考えましょう。これはPEAKのeasy\_installコマンドを使用して、シェルからインストールすることができます：

```
1 easy_install py-matplotlib
```

そうすると、全てのモデル/コントローラ/ビューでインポートすることができます：

```
1 import matplotlib
```

web2pyのソースディストリビューションとWindows版のバイナリディストリビューションは、1つのsite-packagesをトップレベルのフォルダに持っています。Mac版のバイナリディストリビューションは、次のフォルダ内にsite-packagesがあります：

web2py.app/Contents/Resources/site-packages site-packagesを使用する時の問題は、1つのモジュールで複数のバージョンを同時に使用することが難しいことです。例えば、2つのアプリケーションが同じファイルの異なるバージョ

ンを使用する、といった場合です。この例では、`sys.path` はどちらのアプリケーションにも影響するため、変更することができません。

このような状況に対応するため、web2py はグローバルな `sys.path` を変更しないで、モジュールをインポートする別の方法を提供しています。これは、アプリケーションの”modules” フォルダに、モジュールを置くことです。利点の 1 つは、アプリケーションとともに、モジュールが自動的にコピーされ配布されることです。

”`mymodule.py`” モジュールがアプリの ”`modules/`” フォルダに置かれた場合、(`sys.path` を変える必要なく) `web2py` アプリケーションの任意の場所からインポートできます：

```
1 import mymodule
```

#### 4.20 実行環境

ここで議論されてる全てのものは正しく動きますが、代わりに、12 章で説明するコンポーネントを使用してアプリケーションを構築することをお勧めします。

web2py のモデルとコントローラのファイルは、Python の `import` 文を使用してインポートできないという点で Python モジュールではありません。この理由は、モデルとコントローラが、用意された環境で実行されるように設計されているためです。その環境では web2py のグローバルオブジェクト (`request`、`response`、`session`、`cache`、`T`) と、ヘルパー関数が予め公開されています。これは、web2py の環境は動的に作られるのに関わらず、Python が静的（レキシカル）スコープの言語であるため、必要になります。web2py は `exec_environment` 関数を用意し、モデルとコントローラに直接アクセスすることを許しています。`exec_environment` は、web2py の実行環境を作り出し、ファイルをロードし、その環境を含む `Storage` オブジェクトを返します。`Storage` オブジェクトはまた、名前空間のメカニズムとして機能します。この実行環境で実行されるように設計された任意の Python ファイルは、`exec_environment` を使ってロードすることができます。`exec_environment` には、次のような利用方法が含まれます：

- 他のアプリケーションからのデータ（モデル）にアクセスします。
- 他のモデルやコントローラからグローバルオブジェクトにアクセスします。

- 他のコントローラからコントローラの関数を実行します。
- サイト全体のヘルパライブラリをロードします。

次の例では、cas アプリケーションの user テーブルから、rows を読み出します：

```
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)
```

もう 1 つの例で、次のコードを含む”other.py” コントローラを仮定します：

```
1 def some_action():
2     return dict(remote_addr=request.env.remote_addr)
```

このアクションを、他のコントローラから（または web2py のシェルから）呼び出す方法は以下の通りです：

```
1 from gluon.shell import exec_environment
2 other = exec_environment('applications/app/controllers/other.py',
3                           request=request)
3 result = other.some_action()
```

2 行目の `request=request` は省略可能です。これは現在のリクエストを、”other” の環境に渡す効果があります。この引数の指定がない時は、新規の空の（ただし `request.folder` を除く）リクエストオブジェクトを含む環境になります。レスポンスやセッションオブジェクトも、`exec_environment` に渡すことが可能です。ただし、リクエスト、レスポンス、セッションオブジェクトを渡す時は注意してください。呼び出されたアクションによる修正や、呼び出されたアクションでのコードの依存性は、予期せぬ副作用につながる可能性があります。

3 行目で呼ぶ出す関数は、ビューを実行しません。つまり”some\_action” で明示的に、`response.render` を呼び出ししない限り、単純に辞書を返します。

最後の注意：`exec_environment` を不適切に使用しないでください。他のアプリケーションでのアクションの結果が必要な場合、おそらく XML-RPC API で実装すべきです（web2py で、XML-RPC API を実装するのは容易です）。そして `exec_environment` を、リダイレクトの仕組みとして使用しないでください。代わりに `redirect` ヘルパーを使用してください。

## 4.21 協調

複数のアプリケーションを協調させる方法は多数あります：

- アプリケーションは同じデータベースに接続することができ、テーブルを共有することができます。データベースの全てのテーブルを、全てのアプリケーションで定義する必要はありません。しかし、それらを使用するアプリケーションでは、定義しなければなりません。同じテーブルを使用する全てのアプリケーションは1つを除いて、`migrate=False`としてテーブルを定義しなければなりません。
- アプリケーションは、(12章で説明する)LOADヘルパーを使って、他のアプリケーションのコンポーネントを埋め込むことができます。
- アプリケーションは、セッションを共有することができます。
- アプリケーションは、XML-RPCを介して、リモートで互いのアクションを呼び出すことができます。
- アプリケーションは、ファイルシステム(それらが同一のファイルシステムを共有すると仮定)を介して、互いのファイルにアクセスすることができます。
- アプリケーションは、互いのアクションを、前述のように`exec_environment`を用いて、ローカルに呼び出すことができます。
- アプリケーションは、次の構文を使って、互いのモジュールをインポートすることができます：

```
1 from applications.appname.modules import mymodule
```

- アプリケーションは、`PYTHONPATH`の検索パス、つまり`sys.path`にある、任意のモジュールをインポートすることができます。

アプリは他のアプリのセッションを、次のコマンドで読み込むことができます：

```
1 session.connect(request, response, masterapp='appname', db=db)
```

ここで”appname”は、クッキーに最初の`session_id`を設定する、マスター-applicationの名前です。`db`はセッションテーブル(`web2py_session`)を含む、データベースに対するデータベース接続です。セッションを共有する全てのアプリは、セッションストレージとして同じデータベースを使用する必要があります。

アプリケーションは次のように、他のアプリのモジュールをロードすることができます。

```
1 import applications.otherapp.modules.othermodule
```

## 4.22 ロギング

Python はロギング用の API を提供しています。Web2py は、アプリが利用できるように、それを設定するメカニズムを提供します。

アプリケーションでは、ロガーを作成することができます。モデルで次のように記述します:

```
1 import logging
2 logger = logging.getLogger("web2py.app.myapp")
3 logger.setLevel(logging.DEBUG)
```

これを用いて、様々な重要度でメッセージのログを取ることができます。

```
1 logger.debug("Just checking that %s" % details)
2 logger.info("You ought to know that %s" % details)
3 logger.warn("Mind that %s" % details)
4 logger.error("Oops, something bad happened %s" % details)
```

ロギングは、次で説明されている python 標準のモジュールです:

```
1 http://docs.python.org/library/logging.html
```

”web2py.app.myapp” 文字列は、アプリレベルのロガーを定義します。

これが正しく動作するために、このロガーに対する設定ファイルが必要となります。その 1 つは、web2py のルートフォルダの”logging.example.conf” で提供されています。このファイルを”logging.conf” にリネームし、必要ならカスタマイズしてください。

このファイルは自己ドキュメント化されています。ファイルを開いて、参照してください。

”myapp” アプリケーションに対する設定可能なロガーを作成するためには、[loggers] キーのリストに myapp を加える必要があります:

```
1 [loggers]
2 keys=root, rocket, markdown, web2py, rewrite, app, welcome, myapp
```

そして、[logger\_myapp] セクションを加え、[logger\_welcome] を参考に設定してください。

```
1 [logger_myapp]
2 level=WARNING
3 qualname=web2py.app.myapp
4 handlers=consoleHandler
5 propagate=0
```

”handlers” ディレクティブは、ロギングのタイプを指定します。ここでは”myapp”を、コンソールへロギングしています。

#### 4.23 WSGI

web2py と WSGI は愛憎の関係にあります。私たちの観点では、WSGI は、ポータブルな方法で Web サーバーが Web アプリケーションに接続するためのプロトコルとして開発されたものと見ていて、私たちはその目的で使用しています。web2py はコア部分において、1 つの WSGI アプリケーションです (gluon.main.wsgibase)。一部の開発者は、WSGI をミドルウェア通信プロトコルとしての限界まで推し進め、Web アプリケーションを多数の層からなる、たまねぎのように開発しています (各層は、全体的なフレームワーク上で独立に開発された WSGI ミドルウェアからなります)。web2py はこのような構造を内部で採用していません。これは、フレームワークのコアとなる機能 (クッキー セッション、エラー、トランザクション、ディスパッチの処理) は、1 つの包括的な層で扱ったほうが、速度とセキュリティの面でより最適化できると、私たちは感じているからです。

それでもなお web2py では、サードパーティの WSGI アプリケーションとミドルウェアを、次の 3 通りの方法で (もしくはそれらの組み合わせで) 使用することができます。

- ”wsgihandler.py” ファイルを編集し、任意のサードパーティの WSGI ミドルウェアを取り込むことができます。
- アプリケーション内の任意の指定したアクションに対し、サードパーティの WSGI ミドルウェアを接続することができます。
- アクションから、サードパーティの WSGI アプリを呼び出すことができます。

唯一の制限は、サードパーティ製のミドルウェアを使用しても、web2py のコア

機能を置き換えることはできないことです。

#### 4.23.1 外部ミドルウェア

”wsgibase.py” ファイルを、次のように見てみます：

```

1 #...
2 LOGGING = False
3 #...
4 if LOGGING:
5     application = gluon.main.appfactory(wsgiapp=gluon.main.wsgibase,
6                                         logfilename='httpserver.log',
7                                         profilerfilename=None)
8 else:
9     application = gluon.main.wsgibase

```

LOGGING が True にセットされている時、gluon.main.wsgibase はミドルウェアの関数 gluon.main.appfactory によってラップされます。それにより ”httpserver.log” ファイルへのロギング機能が提供されます。同じようにして、任意のサードパーティ製のミドルウェアを追加することができます。詳細については、公式の WSGI ドキュメントを参照してください。

#### 4.23.2 内部ミドルウェア

コントローラのアクション (例えば index) と、サードパーティ製のミドルウェアのアプリケーション (例えば、出力を大文字に変換する MyMiddleware) があるとします。この時、web2py のデコレータを使って、ミドルウェアをアクションに適用することができます。以下はその例です:

```

1 class MyMiddleware:
2     """converts output to upper case"""
3     def __init__(self, app):
4         self.app = app
5     def __call__(self, environ, start_response):
6         items = self.app(environ, start_response)
7         return [item.upper() for item in items]
8
9 @request.wsgi.middleware(MyMiddleware)
10 def index():
11     return 'hello world'

```

全てのサードパーティのミドルウェアが、このメカニズムで動作することは保障できません。

#### 4.23.3 WSGI アプリケーションの呼び出し

WSGI アプリを web2py アクションから呼び出すことは簡単です。以下はその例です：

```
1 def test_wsgi_app(environ, start_response):
2     """this is a test WSGI app"""
3     status = '200 OK'
4     response_headers = [ ('Content-type', 'text/plain'),
5                          ('Content-Length', '13') ]
6     start_response(status, response_headers)
7     return ['hello world!\n']
8
9 def index():
10    """a test action that calls the previous app and escapes output"""
11    items = test_wsgi_app(request.wsgi.environ,
12                          request.wsgi.start_response)
13    for item in items:
14        response.write(item, escape=False)
15    return response.body.getvalue()
```

この場合、index アクションは test\_wsgi\_app を呼び出し、値を返す前にエスケープします。また、index 自身は WSGI アプリではなく、通常の web2py の API(ソケットに書き込む response.write など)を使用する必要があることに、注意してください。

第3版 - 翻訳: 細田謙二 レビュー: 中垣健志

第4版 - 翻訳: 細田謙二 レビュー: Hitoshi Kato



# 5

## ビュー

web2py はモデル、コントローラ、ビューのために Python を使用しています。ただし、ビューにおいては若干修正した Python 構文を用いています。これにより、適切な python の利用に制約をかけることなく、より可読性のあるコードが書けるようになります。

ビューの目的は、HTML ドキュメント内に Python コードを埋め込むことです。これは一般的に、幾つかの問題を引き起こします：

- 埋め込まれたコードは、エスケープすべきか？
- インデントは Python か HTML のどちらのルールに基づいてされるべきか？

web2py は`{{...}}`を HTML に埋め込んだ Python コードをエスケープするために使用しています。中括弧を山括弧の代わりに用いる利点は、全ての一般的な HTML エディタにおいて分かりやすいからです。これによりそれらのエディタで web2py のビューを作成することが可能になります。

開発者は HTML に Python コードを埋め込むので、ドキュメントは Python のルールではなく、HTML のルールに従ってインデントされる必要があります。そのため、web2py では、`{{ ... }}`のタグの中でインデント不要の Python を書けるようにしています。Python は通常、コードのブロックを区切るためにインデントを使用していますが、ここでは別の方針が要求されます。このような理由で、web2py のテンプレート言語は Python キーワードの `pass` を活用しています。

コードブロックは、コロンで終わる行から始まり、`pass` で始まる行で終わ

ります。キーワード `pass` は、ブロックの終わりがコンテキストで明白な場合は必要ありません。

これはその例です：

```

1 { {
2 if i == 0:
3 response.write('i is 0')
4 else:
5 response.write('i is not 0')
6 pass
7 } }
```

`pass` は Python のキーワードで、web2py のキーワードではないことに注意してください。Emacs などの幾つかの Python エディタは、`pass` キーワードを用いてブロックの区切りを示し、自動的にコードを再インデントします。web2py のテンプレート言語は、Python と同じ動きをします。次のような記述は：

```

1 <html><body>
2 {{for x in range(10):}}{{=x}}hello<br />{{pass}}
3 </body></html>
```

テンプレート言語により、次のようなプログラムに変換されます：

```

1 response.write("""<html><body>""", escape=False)
2 for x in range(10):
3     response.write(x)
4     response.write("""hello<br />""", escape=False)
5 response.write("""</body></html>""", escape=False)
```

`response.write` は、`response.body` に書き込みます。web2py のビューにエラーがある場合、エラーレポートには、開発者が書いた実際のビューではなく、生成されたコードを表示します。これにより、実行された実際のコードをハイライトするため、コードをデバッグすることが容易になります（そのコードは HTML エディタやブラウザの DOM インスペクタを使用して、デバッグすることができます）。

また注意する点として：

```
1 {{=x}}
```

これは以下のものを生成します

```
1 response.write(x)
```

このように HTML に注入された変数は、デフォルトでエスケープされます。もし `x` が XML オブジェクトなら、`escape` を `True` にしても、エスケープは無視されます。

ここで示すのは、`H1` ヘルパーの利用例です：

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

これは、以下のものに変換されます：

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

上記の解釈は、`H1` オブジェクトとその要素は再帰的にシリアル化され、さらにエスケープし、`response.body` に書かれます。`H1` と内部 HTML によって生成されたタグはエスケープされません。この仕組みによって、Web ページ上に表示される全てのテキスト—そしてテキストのみ—が、常にエスケープされ、それにより XSS 脆弱性を防ぐことを保証します。同時に、コードはシンプルでデバッグし易いものとなります。

`response.write(obj, escape=True)` メソッドは、記述するオブジェクトとエスケープするかどうか（デフォルトは `True`）という 2 つの引数を取ります。`obj` が `.xml()` メソッドを持つ場合、メソッドが呼び出され、結果が `response` の `body` に書かれます（エスケープ引数は無視されます）。それ以外の場合は、オブジェクトの `_str_` メソッドがシリアル化するのに用いられ、エスケープ引数が `True` ならばエスケープされます。全ての組み込みのヘルパーオブジェクト（`H1` など）は、`.xml()` メソッドを介して自分自身をシリアル化する方法を知っています。

これは、全て透過的に行われます。`response.write` メソッドを明示的に呼び出す必要は決してありません（呼び出すべきではないです）。

## 5.1 基本構文

web2py のテンプレート言語は、全ての Python の制御構造をサポートしています。ここでは、それぞれの例を示します。それらは通常のプログラミングのプラクティスに準じてネストさせることができます。

### 5.1.1 for...in

テンプレートでは、どの反復可能オブジェクトに対してループを回すことができます：

```

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

これは次のようにになります：

```

1 <ul>
2 <li>a</li>
3 <li>b</li>
4 <li>c</li>
5 </ul>
```

ここで `item` は、任意の反復可能オブジェクトです。これは、Python のリスト、Python のタプル、Rows オブジェクト、イテレータとして実装された任意のオブジェクトなどです。表示されるオブジェクトは初めにシリアル化され、エスケープされます。

### 5.1.2 while

`while` キーワードを用いてループを作成できます：

```

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

これは次のようにになります：

```

1 <ul>
2 <li>3</li>
3 <li>2</li>
4 <li>1</li>
5 </ul>
```

### 5.1.3 if...elif...else

条件句を使用できます：

```

1  {{
2 import random
3 k = random.randint(0, 100)
4 }
5 <h2>
6 {{=k}}
7 {{if k % 2:}}is odd{{else:}}is even{{pass}}
8 </h2>
```

これは次のようにになります：

```

1 <h2>
2 45 is odd
3 </h2>
```

`else` が最初の `if` ブロックを閉じることは明らかなので、ここでは `pass` 文は必要なく、使用は不適切です。しかし `else` ブロックは、`pass` 文とともに明示的に閉じなければなりません。

Python では”`else if`” を `elif` と書くことに留意して、次の例を見てください：

```

1  {{
2 import random
3 k = random.randint(0, 100)
4 }
5 <h2>
6 {{=k}}
7 {{if k % 4 == 0:}}is divisible by 4
8 {{elif k % 2 == 0:}}is even
9 {{else:}}is odd
10 {{pass}}
11 </h2>
```

これは次のようにになります：

```

1 <h2>
2 64 is divisible by 4
3 </h2>
```

#### 5.1.4 try...except...else...finally

`try...except` 文をビューの中で使用することもできます。ただし一つ注意があります。次の例を考えてください：

```

1 {{try:}}
```

```

2 Hello {{= 1 / 0}}
3 {{except:}}
4 division by zero
5 {{else:}}
6 no division by zero
7 {{finally}}
8 <br />
9 {{pass}}

```

これは次のような出力を生成します：

```

1 Hello
2 division by zero
3 <br />

```

この例は、例外が発生する前の全ての出力が、try ブロック内でレンダリングされること示しています（例外の前の出力もレンダリングされます）。“Hello”は例外の前にあるので出力されます。

### 5.1.5 def...return

web2py のテンプレート言語では、任意の Python オブジェクトや text/html 文字列を返す関数を定義し実装することができます。ここでは、2つの例を考えます：

```

1 {{def itemize1(link): return LI(A(link, _href="http://" + link))}}
2 <ul>
3 {{=itemize1('www.google.com')}}
4 </ul>

```

これは次のような出力を生成します：

```

1 <ul>
2 <li><a href="http://www.google.com">www.google.com</a></li>
3 </ul>

```

関数 itemize1 は、関数が呼び出された箇所に挿入するヘルパー オブジェクトを返します。

次のコードを考えてみてください：

```

1 {{def itemize2(link):}}
2 <li><a href="http://{{=link}}">{{=link}}</a></li>
3 {{return}}

```

```

4 <ul>
5 {{itemize2('www.google.com')}}
6 </ul>
```

これは上記と全く同じ出力を生成します。このケースの関数 `itemize2` は、関数が呼ばれた場所で web2py のタグが置換されることになる HTML の一部を表現します。`itemize2` の呼び出しの手前に '=' がないことに注意してください。これは、関数が文字列を返すのではなく、直接レスポンスに書き込んでいるからです。

1つ注意点として、ビュー内で定義された関数は `return` 文で終了しなければなりません。そうでないと、自動インデントが失敗します。

## 5.2 HTML ヘルパー

次のようなビューのコードを考えます：

```

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

これは次のようにレンダリングされます：

```

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

`DIV` はヘルパークラスです。つまり、HTML をプログラムで構築するために使われるものです。これは、HTML の`<div>`タグに対応します。

固定引数は、開始タグと終了タグの間に含まれるオブジェクトとして解釈されます。アンダースコアで始まる名前付き引数は、HTML タグの (アンダースコアなしの) 属性として解釈されます。幾つかのヘルパーは、アンダースコアで始まらない名前付き数を持ち、それらの引数はタグ固有のものです。

名前なし引数のセットの代わりに、ヘルパーは \* 表記を用いた、单一のリストまたはタプルをコンポーネントのセットとして受け取ることもできます。さらに \*\* を用いて、单一の辞書を属性のセットとして受け取ることも可能です。以下はその例です：

```

1 {{
2 contents = ['this', 'is', 'a', 'test']
3 attributes = {'_id': '123', '_class': 'myclass'}
4 =DIV(*contents, **attributes)
5 }}
```

(これは前述したものと同じ出力を生成します)

以下のヘルパーのセットは :

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

XML [51] [52] へとシリアル化できる複雑な表現を構築するのに使用することができます。例えば :

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

これは次のようにレンダリングされます :

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

ヘルパーはまた、`__str__`または`xml`メソッドを用いて、同様に文字列にシリアル化することができます。

```
1 >>> print str(DIV("hello world"))
2 <div>hello world</div>
3 >>> print DIV("hello world").xml()
4 <div>hello world</div>
```

web2py のヘルパーの仕組みは、文字列の連結なしに HTML を生成するシステム以上のものです。これは、サーバーサイドのドキュメントオブジェクトモデル(DOM) 表現を提供します。

ヘルパーのコンポーネントはその位置によって参照できます。またヘルパーは、それらコンポーネントに対してリストのように機能します。

```
1 >>> a = DIV(SPAN('a', 'b'), 'c')
2 >>> print a
3 <div><span>ab</span>c</div>
4 >>> del a[1]
5 >>> a.append(B('x'))
6 >>> a[0][0] = 'y'
7 >>> print a
8 <div><span>yb</span><b>x</b></div>
```

ヘルパーの属性は名前で参照することができ、ヘルパーはそれら属性に対して辞書のように機能します：

```
1 >>> a = DIV(SPAN('a', 'b'), 'c')
2 >>> a['_class'] = 's'
3 >>> a[0]['_class'] = 't'
4 >>> print a
5 <div class="s"><span class="t">ab</span>c</div>
```

全てのコンポーネントのセットは、`a.components` というリストを介してアクセスすることができます。また、全ての属性のセットは `a.attributes` という辞書を介してアクセスすることができます。そうすると、`a[i]` は `i` が整数のとき `a.components[i]` と等価になります。また、`a[s]` は `s` が文字列のとき `a.attributes[s]` と等価になります。

なお、ヘルパーの属性は、キーワード引数としてヘルパーに渡すことができます。しかし場合によっては、属性の名前が Python の識別子で許可されていない特殊文字を含むことがあります（例えば、ハイフンなど）、これらはキーワード引数の名前として利用することができません。例えば、：

```
1 DIV('text', _data-role='collapsible')
```

この場合、”\_data-role” がハイフンを含むため動作せず、Python のシンタックエラーが発生します。

このような場合、辞書と Python の\*\*関数引数の表記を、代わりに利用できます。この表記は、(キー:バリュー) ペアの辞書をキーワード引数のセットにマッピングします。

```
1 >>> print DIV('text', **{'_data-role': 'collapsible'})
2 <div data-role="collapsible">text</div>
```

また、特殊な TAG を動的に作成することもできます。

```
1 >>> print TAG['soap:Body']('whatever', **{'_xmlns:m': 'http://www.example
2 .org'})
<soap:Body xmlns:m="http://www.example.org">whatever</soap:Body>
```

### 5.2.1 XML

XML はエスケープされるべきでないテキストを、カプセル化するために使用するオブジェクトです。テキストは有効な XML が含まれる場合もあれば、そうでな

い場合もあります。例えば、JavaScript を含むことができます。

次の例のテキストはエスケープされますが：

```
1 >>> print DIV("<b>hello</b>")
2 &lt;b&gt;hello&lt;/b&gt;
```

次のように XML を使用してエスケープを防ぐことができます：

```
1 >>> print DIV(XML("<b>hello</b>"))
2 <b>hello</b>
```

時々、変数に格納された HTML をレンダリングしたい場合があります。しかし、HTML はスクリプトなどの安全でないタグが含まれている可能性があります：

```
1 >>> print XML('<script>alert ("unsafe!")</script>')
2 <script>alert ("unsafe!")</script>
```

このようなエスケープされてない実行可能な入力(例えば、ブログのコメント本文への入力)は、安全ではありません。これをを利用して、そのページの他の訪問者に対してクロスサイトスクリプティング(XSS)攻撃を生成し、利用することができるからです。

web2py の XML ヘルパーはインジェクションを防ぐためにテキストを無害化することができ、明示的に許可しているものを除き全てのタグをエスケープします。これはその例です：

```
1 >>> print XML('<script>alert ("unsafe!")</script>', sanitize=True)
2 &lt;script&gt;alert("unsafe!")&lt;/script&gt;
```

XML のコンストラクタはデフォルトでは、幾つかのタグの中身とそれらの幾つかの属性が安全であると想定します。このデフォルトは、permitted\_tags と allowed\_attributes というオプション引数で上書きすることができます。以下に示すのは、XML ヘルパーのオプション引数のデフォルトの値です。

```
1 XML(text, sanitize=False,
2     permitted_tags=['a', 'b', 'blockquote', 'br/',
3                      'i', 'li',
4                      'ol', 'ul', 'p', 'cite', 'code', 'pre', 'img/'],
5     allowed_attributes={'a': ['href', 'title'],
6                         'img': ['src', 'alt'],
7                         'blockquote': ['type']})
```

## 5.2.2 組み込みヘルパー

このヘルパーは、リンクを生成するために使用されます。

```
1 >>> print A('<click>', XML('<b>me</b>'),
2           _href='http://www.web2py.com')
3 <a href='http://www.web2py.com'>&lt;click&gt;<b>me</b></a>
```

\_href の代わりに、callback 引数を用いて URL を渡すことができます。例え  
ばビューの中で:

```
1 {{=A('click me', callback=URL('myaction'))}}
```

とすると、リンクを押したときの挙動はリダイレクトではなく”myaction”への  
ajax 呼び出しになります。この場合、target と delete という 2 つの引数を任  
意に指定することができます:

```
1 {{=A('click me', callback=URL('myaction'), target="t")}}
2 <div id="t"><div>
```

こうすると、ajax コールバックのレスポンスは、”t” と一致する id の DIV に格  
納されます。

```
1 <div id="b">{{=A('click me', callback=URL('myaction'), delete='div#b')}}
2 }</div>
```

また上記のレスポンスでは、”div#b” にマッチする最も近いタグが削除されま  
す。この例では、ボタンが消されます。典型的なアプリケーションは次のような  
ものです:

```
1 {{=A('click me', callback=URL('myaction'), delete='tr')}}
```

これはテーブル内です。ボタンを押すと、コールバックが実行され、テーブルの  
行が消されます。

callback と delete は組み合わせることができます。

A ヘルパーは cid という特殊な引数を取ります。これは次のように動作します:

```
1 {{=A('linked page', _href='http://example.com', cid='myid')}}
2 <div id="myid"></div>
```

リンクをクリックすると、その div に中身がロードされるようになります。これ  
は同様なものですか、ページのコンポーネントをリフレッシュするように設計さ  
れているため、上の構文よりも強力です。cid の応用については第 12 章のコン  
ポーネントで、詳しく説明します。

上記の ajax の機能を利用するには jQuery と ”static/js/web2py\_ajax.js” が必要です。これらは {{include 'web2py\_ajax.html'}} をレイアウトの head に置くことで自動的にインクルードされます。”views/web2py\_ajax.html” は、request に基づいて幾つかの変数を定義し、必要な全ての js と css ファイルをインクルードします。

B

このヘルパーは、その中身を太字にします。

```
1 >>> print B('<hello>', XML('<i>world</i>'), _class='test', _id=0)
2 <b id="0" class="test">&lt;hello&gt;<i>world</i></b>
```

BODY

このヘルパーは、ページの body を作成します。

```
1 >>> print BODY('<hello>', XML('<b>world</b>'), _bgcolor='red')
2 <body bgcolor="red">&lt;hello&gt;<b>world</b></body>
```

BR

このヘルパーは改行を作成します。

```
1 >>> print BR()
2 <br />
```

CAT (1.98.1 and up)

このヘルパーは他のヘルパーを連結します。これは、TAG[”] 同じです。

```
1 >>> print CAT('Here is a ', A('link', _href=URL()), ', and here is some
2   ', B('bold text'), '.')
2 Here is a <a href="/app/default/index">link</a>, and here is some <b>
   bold text</b>.
```

CENTER

このヘルパーは、その中身を中央に配置します。

```
1 >>> print CENTER('<hello>', XML('<b>world</b>'),
2 >>>           _class='test', _id=0)
3 <center id="0" class="test">&lt;hello&gt;<b>world</b></center>
```

CODE

このヘルパーは、Python、C、C++、HTML、そして、web2py のコードでの構文のハイライトを実現します。これはコードリストに対して、PRE を使うよりも

望ましいです。CODE はまた、web2py の API ドキュメントへのリンクを作成する機能があります。

ここでは、Python コードのセクションをハイライトする例を示します：

```

1 >>> print CODE('print "hello"', language='python').xml()
2 <table><tr valign="top"><td style="width:40px; text-align: right;"><pre
3   style="
4     font-size: 11px;
5     font-family: Bitstream Vera Sans Mono,monospace;
6     background-color: transparent;
7       margin: 0;
8       padding: 5px;
9       border: none;
10      background-color: #E0E0E0;
11      color: #A0A0A0;
12    ">1.</pre></td><td><pre style="
13   font-size: 11px;
14   font-family: Bitstream Vera Sans Mono,monospace;
15   background-color: transparent;
16     margin: 0;
17     padding: 5px;
18     border: none;
19     overflow: auto;
20   "><span style="color:#185369; font-weight: bold">print </span>
21   <span style="color: #FF9966">"hello"</span></pre></td></tr>
22 </table>
```

HTML に対して同様の例を示します。

```

1 >>> print CODE(
2 >>>   '<html><body>{{=request.env.remote_add}}</body></html>',
3 >>>   language='html')
```

```

1 <table>...<code>...
2 <html><body>{{=request.env.remote_add}}</body></html>
3 ...</code>...</table>
```

以下に示すのは CODE ヘルパーのデフォルトの引数です：

```

1 CODE("print 'hello world'", language='python', link=None, counter=1,
      styles={})
```

language 引数でサポートされている値は、"python"、"html\_plain"、"c"、"cpp"、"web2py"、"html" です。"html" 言語では、{{及び}}のタグは "web2py" のコードとして解釈されます。一方、"html\_plain" ではそのように解釈されません。

`link` の値が指定される場合、例えば”/examples/global/vars/”とされている場合、コード内の web2py の API レファレンスがそのリンク URL のドキュメントへ関係付けられます。例えば”request”は、”/examples/global/vars/request”へ関係付けられます。上記の例ではリンク URL は、”global.py” コントローラの”vars”アクションによって処理されます。”global.py”は、web2py の”examples” アプリケーションの一部として配布されています。

`counter` 引数は行番号のために使用されます。これは 3 種類の値の、いずれかを設定できます。`None` になると行番号は表示されません。数値で開始番号を指定できます。`counter` に文字列を設定した場合、プロンプトとして解釈され、行番号は表示されません。

`styles` 引数は少し変わっています。上記のように生成された HTML を見ると、2 つの列からなるテーブルがあり、各列は CSS を用いてインラインで宣言された独自のスタイルを持つことがあります。`styles` 属性は、これら 2 つの CSS スタイルを上書きできるようにします。例えば次のようにします：

```
1 {{=CODE(..., styles={ 'CODE': 'margin: 0; padding: 5px; border: none; ' })}}
```

`styles` 属性は辞書である必要があり、2 つのキーを取りえます：`CODE` は実際のコードのスタイル、`LINENUMBERS` は行番号を格納する左の列のスタイルです。これらのスタイルは、単純に足すのではなく、デフォルトのスタイルを完全に置き換えることに留意してください。

COL

```
1 >>> print COL('a', 'b')
2 <col>ab</col>
```

COLGROUP

```
1 >>> print COLGROUP('a', 'b')
2 <colgroup>ab</colgroup>
```

DIV

XML 以外の全てのヘルパーは `DIV` から派生し、その基本メソッドを継承しています。

```
1 >>> print DIV('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <div id="0" class="test">&lt;hello&gt;<b>world</b></div>
```

EM

中身を強調します。

```
1 >>> print EM('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <em id="0" class="test">&lt;hello&gt;<b>world</b></em>
```

FIELDSET

入力フィールドをラベルと共に作成するために使用されます。

```
1 >>> print FIELDSET('Height:', INPUT(_name='height'), _class='test')
2 <fieldset class="test">Height:<input name="height" /></fieldset>
```

FORM

これは最も重要なヘルパーの1つです。単純なフォームでは<form>...</form>タグを作り出すだけですが、ヘルパーはオブジェクトであり、そして含まれているもの情報を持っているので、フォームの投稿処理を行うことができます(例えば、フィールドの検証など)。詳細は第7章で説明します。

```
1 >>> print FORM(INPUT(_type='submit'), _action='', _method='post')
2 <form enctype="multipart/form-data" action="" method="post">
3 <input type="submit" /></form>
```

”enctype”はデフォルトでは”multipart/form-data”です。

FORMとSQLFORMのコンストラクタは、hiddenという特別な引数を取ることができます。辞書がhiddenとして渡される場合、その項目は”隠し”INPUTフィールドへと変換されます。例えば次のようになります:

```
1 >>> print FORM(hidden=dict(a='b'))
2 <form enctype="multipart/form-data" action="" method="post">
3 <input value="b" type="hidden" name="a" /></form>
```

H1, H2, H3, H4, H5, H6

これらのヘルパーは、段落の見出しと小見出しに利用します：

```
1 >>> print H1('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <h1 id="0" class="test">&lt;hello&gt;<b>world</b></h1>
```

HEAD

HTMLページのHEADをタグ付けに利用します。

```
1 >>> print HEAD(TITLE('<hello>', XML('<b>world</b>'))))
2 <head><title>&lt;hello&gt;<b>world</b></title></head>
```

## HTML

このヘルパーは少し異なります。<html>タグの作成に加えて、doctype 文字列をタグの前に付加します [54, 55, 56]。

```
1 >>> print HTML(BODY('<hello>', XML('<b>world</b>')))  
2 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  
3 "http://www.w3.org/TR/html4/loose.dtd">  
4 <html><body>&lt;hello&gt;<b>world</b></body></html>
```

HTML ヘルパーは、幾つかの追加のオプション引数を取り、次のようなデフォルト値を持ちます：

```
1 HTML(..., lang='en', doctype='transitional')
```

doctype は 'strict'、'transitional'、'frameset'、'html5'、または、完全な doctype 文字列にすることができます。

## XHTML

XHTML は HTML に似ていますが、代わりに XHTML の doctype を作成します。

```
1 XHTML(..., lang='en', doctype='transitional', xmlns='http://www.w3.org  
/1999/xhtml')
```

doctype は、'strict'、'transitional'、'frameset'、または、完全な doctype 文字列にすることができます。

## HR

このヘルパーは、水平ラインを HTML ページに作成します。

```
1 >>> print HR()  
2 <hr />
```

## I

このヘルパーはその中身をイタリックにします。

```
1 >>> print I('<hello>', XML('<b>world</b>'), _class='test', _id=0)  
2 <i id="0" class="test">&lt;hello&gt;<b>world</b></i>
```

## INPUT

<input.../>タグを作成します。input タグは、他のタグを含まないので、>ではなく、/>で閉じられます。input タグは、オプション属性\_typeを持ち、"text"（デフォルト）や、"submit"、"checkbox"、"radio" といった値を設定できます。

```

1 >>> print INPUT(_name='test', _value='a')
2 <input value="a" name="test" />

```

これはまた、”value” という ”\_value” とは異なる特殊なオプション引数を取ります。後者は、input フィールドに対してデフォルト値を設定します。一方、前者は現在の値を設定します。”text” タイプの入力の場合、前者は後者を上書きします：

```

1 >>> print INPUT(_name='test', _value='a', value='b')
2 <input value="b" name="test" />

```

ラジオボタンでは INPUT は、選択の”checked” 属性を設定します：

```

1 >>> for v in ['a', 'b', 'c']:
2     >>>     print INPUT(_type='radio', _name='test', _value=v, value='b'),
3             v
4 <input value="a" type="radio" name="test" /> a
5 <input value="b" type="radio" checked="checked" name="test" /> b
5 <input value="c" type="radio" name="test" /> c

```

チェックボックスでも同様です：

```

1 >>> print INPUT(_type='checkbox', _name='test', _value='a', value=True)
2 <input value="a" type="checkbox" checked="checked" name="test" />
3 >>> print INPUT(_type='checkbox', _name='test', _value='a', value=False)
4 <input value="a" type="checkbox" name="test" />

```

## IFRAME

このヘルパーは、現在のページに別の Web ページを取り込みます。他のページの URL は”\_src” 属性で指定します。

```

1 >>> print IFRAME(_src='http://www.web2py.com')
2 <iframe src="http://www.web2py.com"></iframe>

```

## IMG

HTML に画像を埋め込むために使用できます：

```

1 >>> IMG(_src='http://example.com/image.png', _alt='test')
2 

```

次に示すのは、A と IMG と URL のヘルパーを組み合わせて、リンク付きの静止画像を取り込んだものです：

```

1 >>> A(IMG(_src=URL('static', 'logo.png'), _alt="My Logo"),
2      _href=URL('default', 'index'))
3 <a href="/myapp/default/index">
4   
5 </a>
```

## LABEL

INPUT フィールド用の LABEL タグを作成するために使用されます。

```

1 >>> print LABEL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <label id="0" class="test">&lt;hello&gt;<b>world</b></label>
```

## LEGEND

フォームの中のフィールドに対する LEGEND タグを作成するために使用されます。

```

1 >>> print LEGEND('Name', _for='myfield')
2 <legend for="myfield">Name</legend>
```

## LI

リスト項目を作成します。UL または OL タグの中に含めるべきです。

```

1 >>> print LI('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <li id="0" class="test">&lt;hello&gt;<b>world</b></li>
```

## META

HTML の head で、META タグを作成するために使用します。例:

```

1 >>> print META(_name='security', _content='high')
2 <meta name="security" content="high" />
```

MARKMIN markmin の wiki 構文を実装します。下の例では markmin のルールに従って、記述されたテキスト入力を html 出力に変換します :

```

1 >>> print MARKMIN("this is **bold** or ''italic'' and this [[a link
2      http://web2py.com]]")
3 <p>this is <b>bold</b> or <i>italic</i> and
3 this <a href="http://web2py.com">a link</a></p>
```

markmin の構文は、web2py に付属している次のファイルで説明しています :

```
1 http://127.0.0.1:8000/examples/static/markmin.html
```

幾つかの例は、第 12 章の MARKMIN を多用する plugin\_wiki の説明で示します。markmin を用いて、HTML、LaTeX、PDF ドキュメントを生成することができます。

```
1 m = "Hello **world** [[link http://web2py.com]]"
2 from gluon.contrib.markmin.markmin2html import markmin2html
3 print markmin2html(m)
4 from gluon.contrib.markmin.markmin2latex import markmin2latex
5 print markmin2latex(m)
6 from gluon.contrib.markmin.markmin2pdf import markmin2pdf
7 print markmin2pdf(m) # requires pdf2tex
```

(MARKMIN ヘルパーは markmin2html のショートカットです)

以下に基本構文の初步を示します:

```
SOURCE / OUTPUT # title / title ## section / section ###
subsection / subsection **bold** / bold ''italic'' / italic
``verbatim`` / verbatim http://google.com / http://google.com
http://... / <a href="http://">http://...</a> http://...png
/  http://...mp3 / <audio
src="http://...mp3"></audio> http://...mp4 / <video
src="http://...mp4"></video> qr:http://... / <a
href="http://"></a> embed:http://...
/ <iframe src="http://"></iframe> [[click me #myanchor]]
/ click me $$\int_a^b \sin(x) dx$$ / $\int_a^b \sin(x) dx$
```

画像やビデオ、音声ファイルへのマークアップなしのリンクを単純に含めると、対応する画像、ビデオ、音声ファイルが自動的に組み込まれます(音声とビデオには HTML の<audio>と<video>タグを使用します)。

リンクに qr: プレフィックスを次のように追加すると、

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

対応する QR コードが埋め込まれ、指定した URL ハリンクするようになります。

リンクに embed: プレフィックスを次のように追加すると、

```
1 embed:http://www.youtube.com/embed/x1w8hKTJ2Co
```

埋め込まれたページが表示されます。この場合は、youtube のビデオが埋め込まれます。

画像はまた、次のような構文で埋め込むことができます。

```
1 [[image-description http://.../image.png right 200px]]
```

順序がないリストは次のように書きます:

```
1 - one
2 - two
3 - three
```

順序付きリストは次のように書きます:

```
1 + one
2 + two
3 + three
```

テーブルは次のように書きます:

```
1 -----
2   X | 0 | 0
3   0 | X | 0
4   0 | 0 | 1
5 -----
```

MARKMIN 構文はまた、blockquote や、HTML5 の audio、video タグ、画像の位置調整、カスタム CSS をサポートし、拡張することができます:

```
1 MARKMIN("``abab``:custom", extra=dict(custom=lambda text: text.replace(
    'a', 'c')))
```

これは次のものを生成します。

```
'cbcb'
```

カスタムブロックは ``...``:<key>によって区切られ、そして MARKMIN の特別な辞書引数の対応するキーの値として、渡される関数によってレンダリングされます。関数は、XSS を防ぐために出力をエスケープする必要があるかもしれないことに注意してください。

OBJECT

HTML にオブジェクト (例えば、flash プレーヤーなど) を埋め込むために使用します。

```
1 >>> print OBJECT('<hello>', XML('<b>world</b>'),
2 >>>                      _src='http://www.web2py.com')
3 <object src="http://www.web2py.com">&lt;hello&gt;<b>world</b></object>
```

## OL

順序付きリストを表します。リストは LI タグを含める必要があります。LI オブジェクトでない OL の引数は、自動的に<li>...</li>タグで囲まれます。

```
1 >>> print OL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <ol id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ol>
```

## ON

これは後方互換のためにあり、単に `True` の別名となります。チェックボックスで利用されますが、`True` の方がより Python 的ですので、非推奨になっています。

```
1 >>> print INPUT(_type='checkbox', _name='test', _checked=ON)
2 <input checked="checked" type="checkbox" name="test" />
```

## OPTGROUP

SELECT 内の複数のオプションをグループ化することを可能にします。CSS でフィールドをカスタマイズするのに便利です。

```
1 >>> print SELECT('a', OPTGROUP('b', 'c'))
2 <select>
3   <option value="a">a</option>
4   <optgroup>
5     <option value="b">b</option>
6     <option value="c">c</option>
7   </optgroup>
8 </select>
```

## OPTION

SELECT/OPTION の組み合わせの一部としてのみ使用されるものです。This should only be used as part of a SELECT/OPTION combination.

```
1 >>> print OPTION('<hello>', XML('<b>world</b>'), _value='a')
2 <option value="a">&lt;hello&gt;<b>world</b></option>
```

INPUT の場合と同じように web2py は、”\_value”(OPTION の値)と”value”(SELECT ボックスの現在の値)を区別することができます。もしそれらが等しい場合、オプションは”selected”になります。

```
1 >>> print SELECT('a', 'b', value='b'):
2 <select>
```

```

3 <option value="a">a</option>
4 <option value="b" selected="selected">b</option>
5 </select>
```

P

段落のタグ付けに利用します。

```

1 >>> print P('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <p id="0" class="test">&lt;hello&gt;<b>world</b></p>
```

PRE

整形済みテキストを表示するための<pre>...</pre>タグを生成します。CODE ヘルパーのほうが、コードリストには一般に望ましいです。

```

1 >>> print PRE('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <pre id="0" class="test">&lt;hello&gt;<b>world</b></pre>
```

SCRIPT

JavaScriptなどのスクリプトを組み込む、または、リンクします。タグに包まれる内容は、実際に古いブラウザのために、HTMLのコメントとしてレンダリングされます。

```

1 >>> print SCRIPT('alert("hello world");', _language='javascript')
2 <script language="javascript"><!--
3 alert("hello world");
4 //--></script>
```

SELECT

<select>...</select>タグを作成します。これは、OPTION ヘルパーとともに使用されます。OPTION オブジェクト以外の SELECT 引数は、自動的に option に変換されます。

```

1 >>> print SELECT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <select id="0" class="test">
3     <option value="&lt;hello&gt;">&lt;hello&gt;</option>
4     <option value="&lt;b&gt;world&lt;/b&gt;"><b>world</b></option>
5 </select>
```

SPAN

DIVと似ていますが、(ブロックよりも) インラインのコンテンツをタグ付けするのに使用されます。

```

1 >>> print SPAN('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <span id="0" class="test">&lt;hello&gt;<b>world</b></span>

```

### STYLE

script と似ていますが、CSS コードを組み込む、もしくは、リンクするために使用されます。次に示すのは、CSS を組み込んだ例です：

```

1 >>> print STYLE(XML('body {color: white}'))
2 <style><!--
3 body { color: white }
4 //--></style>

```

また、リンクする例です：

```

1 >>> print STYLE(_src='style.css')
2 <style src="style.css"><!--
3 //--></style>

```

### TABLE, TR, TD

これらのタグは (オプション的な THEAD、TBODY、TFOOTER ヘルパーとともに)HTML テーブルを作成するために使用されます。

```

1 >>> print TABLE(TR(TD('a'), TD('b')), TR(TD('c'), TD('d')))
2 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>

```

TR は TD が来ることを想定しています。つまり、TD オブジェクトでない引数は自動的に変換されます：

```

1 >>> print TABLE(TR('a', 'b'), TR('c', 'd'))
2 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>

```

Python 配列から HTML テーブルへは Python の\*関数引数の表記を使用すると、リスト要素が関数引数の位置にマッピングされるため、容易に変換することができます。

次の例では、行単位で実行しています：

```

1 >>> table = [['a', 'b'], ['c', 'd']]
2 >>> print TABLE(TR(*table[0]), TR(*table[1]))
3 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></
    table>

```

次の例では、一度に全ての行で実行しています：

```

1 >>> table = [[ 'a', 'b'], [ 'c', 'd']]
2 >>> print TABLE(*[TR(*rows) for rows in table])
3 <table><tr><td>a</td><td>b</td></tr><tr><td>c</td><td>d</td></tr></table>

```

## TBODY

テーブルのボディーに含む行集合を、タグ付けするのに使用されます。ヘッダー やフッターの行集合と対称です。これは任意なものです。

```

1 >>> print TBODY(TR('<hello>'), _class='test', _id=0)
2 <tbody id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tbody>

```

## TEXTAREA

<textarea>...</textarea>タグを作成するヘルパーです。

```

1 >>> print TEXTAREA('<hello>', XML('<b>world</b>'), _class='test')
2 <textarea class="test" cols="40" rows="10">&lt;hello&gt;<b>world</b></textarea>

```

唯一の注意点は、オプションの”value”が、その内容(内部HTML)を上書きすることです。

```

1 >>> print TEXTAREA(value=<hello world>, _class="test")
2 <textarea class="test" cols="40" rows="10">&lt;hello world&gt;</textarea>

```

## TFOOT

テーブルのフッターの行集合をタグ付けするのに使用されます。

```

1 >>> print TFOOT(TR(TD('<hello>')), _class='test', _id=0)
2 <tfoot id="0" class="test"><tr><td>&lt;hello&gt;</td></tr></tfoot>

```

## TH

テーブルのヘッダーにおいて、TD の代わりに使用されます。

```

1 >>> print TH('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <th id="0" class="test">&lt;hello&gt;<b>world</b></th>

```

## THEAD

テーブルのヘッダーの行集合をタグ付けするために使用されます。

```

1 >>> print THEAD(TR(TH('<hello>')), _class='test', _id=0)
2 <thead id="0" class="test"><tr><th>&lt;hello&gt;</th></tr></thead>

```

## TITLE

HTML ヘッダーにて、ページのタイトルをタグ付けするのに使用されます。

```
1 >>> print TITLE('<hello>', XML('<b>world</b>'))
2 <title>&lt;hello&gt;<b>world</b></title>
```

## TR

テーブルの行をタグ付けします。テーブルの内部でレンダリングされ、かつ、  
`<td>...</td>` タグを含む必要があります。TD オブジェクトでない TR の引数  
は、自動で変換されます。

```
1 >>> print TR('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <tr id="0" class="test"><td>&lt;hello&gt;<b>world</b></td></tr>
```

## TT

タイプライタ (固定幅) テキストとしてのテキストをタグ付けします。

```
1 >>> print TT('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <tt id="0" class="test">&lt;hello&gt;<b>world</b></tt>
```

## UL

順序のないリストを示します。LI の項目を含める必要があります。中身が LI と  
してタグ付けされていない場合は、UL が自動で変換します。

```
1 >>> print UL('<hello>', XML('<b>world</b>'), _class='test', _id=0)
2 <ul id="0" class="test"><li>&lt;hello&gt;</li><li><b>world</b></li></ul>
```

## embed64

embed64 (filename=None, file=None, data=None,  
extension='image/gif') は与えられた (バイナリ) データを base64 に  
エンコードします。filename: 指定された場合、このファイルを'rb' モードで開  
き、読み出します。file: 指定された場合、このファイルを読みだします。data:  
指定された場合、指定されたデータを使用します。

## xmlescape

xmlescape (data, quote=True) は指定されたデータのエスケープされた文字  
列を返します。

```

1 >>> print xmlescape('<hello>')
2 &lt;hello&gt;
```

### 5.2.3 カスタム・ヘルパー

#### TAG

独自の XML タグを生成したい場合があるかもしれません。web2py は、普遍的なタグ生成機能として TAG を提供しています。

```

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

これは、次のような XML を生成します。

```

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

引数”a”、”b”、”d”は自動的にエスケープされます。この挙動を抑えるには XML ヘルパーを用いてください。TAG を使用し、API に提供されていない HTML/XML タグを生成することができます。TAG はネストさせることができます。str() によってシリアル化することができます。同等の構文は次の通りです：

```

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

TAG オブジェクトが空の名前で作成された場合、囲むためのタグを挿入せずに、複数文字列及び HTML ヘルパーを連結するために使用することができます。しかしこのような用途は、非推奨になりました。代わりに CAT を使用してください。

なお TAG はオブジェクトで、TAG.name または TAG['name'] は、一時的なヘルパークラスを返す関数です。

#### MENU

MENU ヘルパーは、(第4章で説明している)response.menu の、リストのリストまたはリストのタプルを取り、メニューを表現する順序のないリストを用いてツリー型の構造を生成します。例えば次のようになります:

```

1 >>> print MENU([['One', False, 'link1'], ['Two', False, 'link2']])
2 <ul class="web2py-menu web2py-menu-vertical">
3   <li><a href="link1">One</a></li>
4   <li><a href="link2">Two</a></li>
5 </ul>
```

各メニュー項目は、ネストするサブメニューを第4引数に持つことができます（同じように再帰的に続きます）：

```

1 >>> print MENU([['One', False, 'link1', [['Two', False, 'link2']]]])
2 <ul class="web2py-menu web2py-menu-vertical">
3   <li class="web2py-menu-expand">
4     <a href="link1">One</a>
5     <ul class="web2py-menu-vertical">
6       <li><a href="link2">Two</a></li>
7     </ul>
8   </li>
9 </ul>
```

MENU ヘルパーは次のオプション引数を取ります：

- `_class`: 外側の UL 要素のクラスを設定します。デフォルトは”web2py-menu web2py-menu-vertical”です。
- `ul_class`: 内側の UL 要素のクラスを設定します。デフォルトは”web2py-menu-vertical”です。
- `li_class`: 内側の LI 要素のクラスを設定します。デフォルトは”web2py-menu-expand”です。

MENU は `mobile` というオプション引数を取ります。True が設定された場合、再帰的な UL メニュー構造を生成するのに代わり、SELECT ドロップダウンを返します。これは、全てのメニュー オプションを持ち、選択オプションに対応したページにリダイレクトする `onchange` 属性を持ちます。これは電話のような小さなモバイルデバイスで、ユーザービリティを向上させる代替メニュー表示のために設計されています。

通常メニューは、次の構文でレイアウトに使用されます。

```
1 { {=MENU(response.menu, mobile=request.user_agent().is_mobile) } }
```

このようにモバイルデバイスは自動で検知され、それに従ってメニューはレンダリングされます。

### 5.3 BEAUTIFY

BEAUTIFY は、リストやタブル、辞書を含む、複合的なオブジェクトの HTML 表現を構築するために使用されます：

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

BEAUTIFY は XML へとシリアル化される XML のようなオブジェクトを、コンストラクタの引数の見栄えの良い表現とともに返します。この場合、次のような XML 表現は：

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

次のようにレンダリングされます：

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

## 5.4 サーバーサイドの DOM と構文解析

### 5.4.1 elements

DIV ヘルパーと全ての派生ヘルパーは、検索メソッドの `element` と `elements` を提供します。

`element` は、指定した条件にマッチする最初の子供要素を返します（マッチするものがない場合、`None` を返します）。

`elements` は、マッチした全ての子供のリストを返します。

`element` と `elements` はマッチ条件を指定するのにも同じ構文を用います。この構文は、次のように使用の許された 3 種のタイプがあり、混合してマッチングに使用可能です：jQuery のような表現と、正確に属性値と照合するもの、正規表現を使用して照合するものです。

ここでは簡単な例を示します：

```
1 >>> a = DIV(DIV(DIV('a', _id='target', _class='abc')))
2 >>> d = a.elements('div#target')
3 >>> d[0][0] = 'changed'
4 >>> print a
5 <div><div><div id="target" class="abc">changed</div></div></div>
```

無名引数の `elements` は、次のものを含む文字列です：タグの名前、ポンド記号

が前に付くタグの ID、ドットが前に付くクラス、角括弧内にある属性の明示的な値です。

ここでは、前述のタグを ID で検索するための 4 つの同等な方法を示します：

```
1 >>> d = a.elements('#target')
2 >>> d = a.elements('div#target')
3 >>> d = a.elements('div[id=target]')
4 >>> d = a.elements('div', _id='target')
```

ここでは、前述のタグをクラスで検索するための 4 つの同等な方法を示します：

```
1 >>> d = a.elements('.abc')
2 >>> d = a.elements('div.abc')
3 >>> d = a.elements('div[class=abc]')
4 >>> d = a.elements('div', _class='abc')
```

任意の属性は、要素の場所を特定するのに用いることができます (`id` や `class` だけではありません)。複数の属性 (`element` 関数は複数の名前付き引数を取ることができます) を用いることができますが、最初にマッチした要素だけを返します。jQuery の構文”`div#target`”を用いて、スペースで区切られた複数の検索条件を指定することも可能です：

```
1 >>> a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
2 >>> d = a.elements('span#t1', 'div#c2')
```

または次のように書けます

```
1 >>> a = DIV(SPAN('a', _id='t1'), DIV('b', _class='c2'))
2 >>> d = a.elements('span#t1', 'div#c2')
```

属性の値が名前付き引数で指定されている場合、文字列か正規表現を取ることができます：

```
1 >>> a = DIV(SPAN('a', _id='test123'), DIV('b', _class='c2'))
2 >>> d = a.elements('span', _id=re.compile('test\d{3}'))
```

`DIV`(とその派生) ヘルパーの特別な名前付き引数は `find` です。これは、タグのテキスト内容に対する、検索値、または、検索の正規表現を指定するために用いられます。例えば次のようになります:

```
1 >>> a = DIV(SPAN('abcde'), DIV('fghij'))
2 >>> d = a.elements(find='bcd')
3 >>> print d[0]
4 <span>abcde</span>
```

または

```

1 >>> a = DIV(SPAN('abcde'), DIV('fghij'))
2 >>> d = a.elements(find=re.compile('fg\w{3}'))
3 >>> print d[0]
4 <div>fghij</div>

```

#### 5.4.2 components

html 文字列の全ての要素を列挙する例を示します：

```

1 html = TAG('<a>xxx</a><b>yyy</b>')
2 for item in html.components: print item

```

#### 5.4.3 parent

`parent` は、現在の要素の親を返します。

```

1 >>> a = DIV(SPAN('a'), DIV('b'))
2 >>> d = a.element('a').parent()
3 >>> d['_class']='abc'
4 >>> print a
5 <div class="abc"><span>a</span><div>b</div></div>

```

#### 5.4.4 flatten

`flatten` メソッドは、再帰的に、与えられた要素の子要素の中身を標準の(タグなし)テキストにシリアル化します：

```

1 >>> a = DIV(SPAN('this', DIV('is', B('a'))), SPAN('test'))
2 >>> print a.flatten()
3 thisisatest

```

`flatten` では、オプション引数の `render` を渡すことができます。すなわち、異なるプロトコルを使用して中身をレンダリング/フラット化する関数です。次に示すのは、幾つかのタグを Markmin の wiki 構文へとシリアル化する例です：

```

1 >>> a = DIV(H1('title'), P('example of a ', A('link', _href='#test')))
2 >>> from gluon.html import markmin_serializer
3 >>> print a.flatten(render=markmin_serializer)
4 ## titles

```

```

5
6 example of [[a link #test]]

```

書き込む際には、`markmin_serializer` と `markdown_serializer` が用意されています。

#### 5.4.5 構文解析

`TAG` オブジェクトは XML/HTML の構文解析器でもあります。テキストを読み込んで、ヘルパーのツリー構造へと変換することができます。これにより、上記の API を利用した操作を可能にします：

```

1 >>> html = '<h1>Title</h1><p>this is a <span>test</span></p>'
2 >>> parsed_html = TAG(html)
3 >>> parsed_html.element('span')[0] = 'TEST'
4 >>> print parsed_html
5 <h1>Title</h1><p>this is a <span>TEST</span></p>

```

### 5.5 ページレイアウト

ビューはツリー状の構造中に、拡張や他のビューを取り込むことができます。

例えば、”layout.html” を拡張し、”body.html” を取り込んだ”index.html” というビューを考えることができます。同時に、”layout.html” は、”header.html” や”footer.html” を取り込むこともあります。

ツリーのルートは、レイアウトビューと呼ばれます。他の THML テンプレートファイルと同様に、web2py の管理インターフェースを用いてそれを編集することができます。ファイル名”layout.html” は単なる慣例です。

ここでは、”layout.html” ビューを拡張し、”page.html” ビューを取り込んだ最小のページを示します：

```

1 {{extend 'layout.html'}}
2 <h1>Hello World</h1>
3 {{include 'page.html'}}

```

この拡張したレイアウトファイルは、次のように、`{{include}}` ディレクティブを必ず含まなければなりません：

```

1 <html>
2   <head>
3     <title>Page Title</title>
4   </head>
5   <body>
6     {{include}}
7   </body>
8 </html>
```

ビューが呼び出されると、拡張した（レイアウト）ビューが呼び出され、呼び出したビューはレイアウト内部の{{include}}ディレクティブを置換します。この処理は再帰的に、全ての extend と include ディレクティブの処理が終わるまで行われます。そして、結果のテンプレートは、Python コードに変換されます。ただし、アプリケーションがバイトコードにコンパイルされている場合、コンパイルされているのは Python コードであり、オリジナルのビューファイル自身ではありません。したがって、提供されたビューのバイトコードにコンパイルされたバージョンは、オリジナルのビューファイルだけでなく、拡張とインクルードしたビューの全体ツリーの、Python コードを含む単一の.pyc ファイルになります。

extend、include、block、super、はテンプレート固有のディレクティブで、Python のコマンドではありません。

{{extend ...}} ディレクティブの前にある、どのコンテンツまたはコードも、拡張したビューのコンテンツ/コードの始まりの前に挿入（したがって実行）されます。実際、HTML コンテンツを、拡張ビューのコンテンツの前に挿入することは稀です。しかし拡張したビューで利用可能にする、変数や関数を定義するのに便利です。例えば、次のような”index.html”ビューを考えます：

```

1 {{sidebar_enabled=True}}
2 {{extend 'layout.html'}}
3 <h1>Home Page</h1>
```

そして、次のような”layout.html”の抜粋を考えます。

```

1 {{if sidebar_enabled:}}
2   <div id="sidebar">
3     Sidebar Content
4   </div>
5 {{pass}}
```

”index.html”における extend の前の sidebar\_enabled の指定によって、”layout.html”が始まる前にその行が挿入され、”layout.html”のコード内

のどの場所でも `sidebar_enabled` が有効になります（この洗練されたバージョンが `welcome` アプリにおいて使用されています）。

なおコントローラの関数から返された変数は、その関数のメインのビューだけではなく、全ての拡張あるいはインクルードしたビュー中でも利用可能です。

`extend` または `include` の引数（つまり拡張もしくはインクルードしたビューの名前）は、python 変数（python 式ではありません）にすることができます。しかし、これには制約があります。`- extend` または `include` 命令文に変数を使用するビューは、バイトコードにコンパイルすることはできません。前述したとおり、バイトコードにコンパイルされたビューは、拡張及びインクルードしたビューの全体ツリーを含んでいます。このため拡張及びインクルードされた指定のビューは、コンパイル時点で判明している必要があります。ビューの名前が変数の場合、これが不可能です（その値はランタイム時まで決定できないからです）。バイトコードにコンパイルされているビューは大幅な速度向上が期待できるため、`extend` や `include` に変数を使用するのは、一般に可能な限り避けたほうがよいでしょう。

場合によっては、`include` の変数を用いる代わりに、通常の`{{include ...}}` ディレクティブを `if...else` ブロックの中に単に置くこともできます。

```

1 {{if some_condition:}}
2 {{include 'this_view.html'}}
3 {{else:}}
4 {{include 'that_view.html'}}
5 {{pass}}

```

上記のコードでは変数を含んでいないので、バイトコードのコンパイルに関する問題は生じません。ただしバイトコードにコンパイルしたビューは、”`this_view.html`” と ”`that_view.html`” の両方の Python コードを、`some_condition` の値によって片方しか実行されない場合でも含んでいます。

注意として、これは `include` に対してのみ機能します。`- {{extend ...}}` 指示文は `if...else` ブロックの中に置くことはできません。

レイアウトは、ページの共通性（ヘッダー、フッター、メニュー）をカプセル化するために使用されます。それらは必須ではないですが、アプリケーションを書きやすく、保守しやすくします。とりわけコントローラで設定することのできる、次の変数を活用するレイアウトを書くことを推奨します。これらによく知られた変数を用いることは、レイアウトを交換可能にするのに役立ちます：

```

1 response.title
2 response.subtitle
3 response.meta.author
4 response.meta.keywords
5 response.meta.description
6 response.flash
7 response.menu
8 response.files

```

`menu` と `files` を除いて、これらの全ての文字列と意味は明らかです。

`response.menu` のメニューは、3-タプル、または、4-タプルからなるリストです。その3つの要素は、リンクの名前、リンクがアクティブ(現在のリンク)かどうかを示すブール値、リンク先のページのURLです。例えば次のようにります:

```

1 response.menu = [ ('Google', False, 'http://www.google.com', []),
2                   ('Index', True, URL('index'), [])]

```

第4のタプル要素は、オプションのサブメニューです。

`response.files` は、ページで必要とする CSS と JS ファイルのリストです。

同様に、次のものを HTML の head で使用することを推奨しています :

```

1 {{include 'web2py_ajax.html'}}

```

これによって特殊効果や Ajax のための、jQuery ライブラリと幾つかの後方互換がある JavaScript 関数の定義が組み込まれます。”`web2py_ajax.html`” は、ビューの `response.meta` タグや、jQuery のベース、カレンダーの日付選択機能、そして必要な全ての `response.files` の CSS と JS を含みます。

### 5.5.1 デフォルトのページのレイアウト

ここで示すのは、`web2py` の雛形アプリケーション `welcome` に付属する最小の”`views/layout.html`” です(付属のパーティは省いています)。新規のアプリケーションはどれも、同様のデフォルトレイアウトを持っています：

```

1 <!DOCTYPE html>
2 <head>
3   <meta charset="utf-8" />
4   <title>{{=response.title or request.application}}</title>
5

```

```
6 <!-- http://dev.w3.org/html5/markup/meta.name.html -->
7 <meta name="application-name" content="{{=request.application}}" />
8
9 <script src="{{=URL('static', 'js/modernizr.custom.js')}}"></script>
10
11 <!-- include stylesheets -->
12 {{{
13     response.files.append(URL('static', 'css/skeleton.css'))
14     response.files.append(URL('static', 'css/web2py.css'))
15     response.files.append(URL('static', 'css/superfish.css'))
16     response.files.append(URL('static', 'js/superfish.js'))
17 }}}
18
19 {{include 'web2py_ajax.html'}}
20
21 <script type="text/javascript">
22     jQuery(function() { jQuery('ul.sf-menu').supersubs({minWidth:12,
23         maxWidth:30, extraWidth:3}).superfish(); });
24 </script>
25
26 {{{
27     # using sidebars need to know what sidebar you want to use
28     left_sidebar_enabled = globals().get('left_sidebar_enabled',False)
29     right_sidebar_enabled = globals().get('right_sidebar_enabled',False)
30     middle_columns = {0:'sixteen',1:'twelve',2:'eight'}[
31         (left_sidebar_enabled and 1 or 0)+(right_sidebar_enabled and 1 or
32         0)]
33 }}}
34
35 </head>
36 <body>
37     <div class="wrapper"><!-- for sticky footer -->
38
39         <div class="topbar">
40             <div class="container">
41                 <div class="sixteen columns">
42                     <div id="navbar">
43                         {{='auth' in globals() and auth.navbar(separators=( ' ', ' | '
44                         ' ', ''))}}
45                     </div>
46                     <div id="menu">
47                         {{=MENU(response.menu,
48                             _class='mobile-menu' if is_mobile else 'sf-menu',
49                             mobile=request.user_agent().is_mobile)}}
50                     </div>
51                 </div>
52             </div>
53         </div>
54     </div><!-- topbar -->
```

```
52     <div class="flash">{{=response.flash or ''}}</div>
53
54     <div class="header">
55         <div class="container">
56             <div class="sixteen columns">
57                 <h1 class="remove-bottom" style="margin-top: .5em;">
58                     {{=response.title or request.application}}
59                 </h1>
60                 <h5>{{=response.subtitle or ''}}</h5>
61             </div>
62
63             <div class="sixteen columns">
64                 <div class="statusbar">
65                     {{block statusbar}}
66                     <span class="breadcrumbs">{{=request.function}}</span>
67                     {{end}}
68                 </div>
69             </div>
70         </div>
71     </div>
72
73     <div class="main">
74         <div class="container">
75             {{if left_sidebar_enabled:}}
76             <div class="four columns left-sidebar">
77                 {{block left_sidebar}}
78                 <h3>Left Sidebar</h3>
79                 <p></p>
80                 {{end}}
81             </div>
82             {{pass}}
83
84             <div class="{{=middle_columns}} columns center">
85                 {{block center}}
86                 {{include}}
87                 {{end}}
88             </div>
89
90             {{if right_sidebar_enabled:}}
91             <div class="four columns">
92                 {{block right_sidebar}}
93                 <h3>Right Sidebar</h3>
94                 <p></p>
95                 {{end}}
96             </div>
97             {{pass}}
98
99             </div><!-- container -->
100            </div><!-- main -->
```

```
101 <div class="push"></div>
102 </div><!-- wrapper -->
103
104
105 <div class="footer">
106   <div class="container header">
107     <div class="sixteen columns">
108       {{block footer}} <!-- this is default footer -->
109       <div class="footer-content" >
110         {{=T('Copyright')}} &#169; 2011
111         <div style="float: right;">
112           <a href="http://www.web2py.com/">
113             
115           </a>
116         </div>
117       </div>
118       {{end}}
119     </div>
120   </div><!-- container -->
121 </div><!-- footer -->
122
123 </body>
124 </html>
```

このデフォルトのレイアウトにはちょっとした特徴があり、非常に使いやすく、カスタマイズしやすいようになっています：

- HTML5 で書かれ、`modernizr` を用いてます。 [49] は後方互換性のためのライブラリです。実際のレイアウトでは IE で必要とされる追加の条件文を含みますが、簡潔にするため無視します。
- モデルで設定することのできる `response.title` と `response.subtitle` を表示します。設定されていない場合は、アプリケーションの名前がタイトルとなります。
- `web2py_ajax.html` ファイルをヘッダーに組み込みます。これは、全てのリンクおよびスクリプトのインポート命令文を生成します。
- ”skeleton” の修正版を使用しています。 [50] 柔軟なレイアウトのためのライブラリで、モバイルデバイス上で機能し、小さなスクリーンにフィットするようにコラム（列）を再配置します。
- ”superfish.js” を動的なカスケードメニューのために使用します。superfish のカスケードメニューを有効にするための明示的なスクリプトが用意されお

り、不要な場合は削除することができます。

- `{=auth.navbar(...)}` は、現在のユーザーへのあいさつ文を表示すると共に、login、logout、register、change password などの auth 関数へのリンクを、内容に応じて表示します。これはヘルパーのファクトリーで、その出力は他のヘルパーと同様に操作することができます。auth が定義されていないときのために、 `{{try:}}...{{except:pass}}` の中に設置されています。
- `{=MENU(response.menu)` は、`<ul>...</ul>` のようにメニュー構造を表示します。
- `{{include}}` は、ページがレンダリングされるときに、拡張先のビューの内容に置き換えられます。
- デフォルトでは、条件付きの 3 カラムです（左、右のサイドバーは拡張先のビューで消すことができます）。
- 次のようなクラスを用いています： header, main, footer
- 次のようなブロックを用いています： statusbar, left\_sidebar, center, right\_sidebar, footer

ビューは次のようにサイドバーを有効にし、中身を定義することができます。

```

1 {{left_sidebar_enable=True}}
2 {{extend 'layout.html'}}
3
4 This text goes in center
5
6 {{block left_sidebar}}
7 This text goes in sidebar
8 {{end}}

```

### 5.5.2 デフォルトレイアウトのカスタマイズ

編集なしにデフォルトレイアウトをカスタマイズするのは、とても簡単です。CSS ファイルが、よくドキュメント化されているからです。

- ”skeleton.css” はリセット、グリッドレイアウト、フォームのスタイルを含みます。
- ”web2py.css” は web2py 固有のスタイルを含みます。

- ”superfish.css” はメニューのスタイルを含みます。

色や背景画像を変更するには、単に次のようなコードを”web2py.css” に追加してください:

```

1 body { background: url('images/background.png') repeat-x #3A3A3A; }
2 a { color: #349C01; }
3 .header h1 { color: #349C01; }
4 .header h2 { color: white; font-style: italic; font-size: 14px; }
5 .statusbar { background: #333333; border-bottom: 5px solid #349C01; }
6 .statusbar a { color: white; }
7 .footer { border-top: 5px solid #349C01; }
```

メニューでは色の指定は特に行っていませんが、同様に変更することができます。

もちろん、”layout.html” や”web2py.css” ファイルを、独自のものに完全に置き換えることも可能です。

### 5.5.3 モバイル開発

デフォルトの layout.html はモバイルデバイスに親和性があるように設計されていますが、十分とは言えません。一つは、ページがモバイルデバイスによって訪問されたときに、異なるビューを使用する必要があるかもしれません。

デスクトップとモバイルデバイスの開発を容易にするため、web2py は@mobilize デコレータを用意しています。このデコレータは、通常のビューとモバイルビューを持たなければいけないアクションに対して適用されます。ここでは、その利用例を示します:

```

1 from gluon.contrib.user_agent_parser import mobilize
2 @mobilize
3 def index():
4     return dict()
```

デコレータはコントローラで使用する前に、インポートされている必要があります。”index” 関数が通常のブラウザ（デスクトップコンピュータ）から呼ばれるときは、web2py は”[controller]/index.html” のビューを用いて、返される辞書をレンダリングします。しかしモバイルデバイスから呼ばれるときは、その辞書は”[controller]/index.mobile.html” によってレンダリングされます。すなわち、モバイルのビューは”mobile.html” 拡張子を持つことになります。

代わりに、次のようなロジックを適用して、全てのビューをモバイルフレンドリーにすることができます。

```
1 if request.user_agent().is_mobile:
2     response.view.replace('.html', '.mobile.html')
```

”\*.mobile.html” ビューを作成するタスクは開発者に残されますが、そのタスクをとても簡単にする”jQuery Mobile” プラグインを使用することを強く推奨します。

## 5.6 ビュー内の関数

次の”layout.html”を考えます：

```
1 <html>
2     <body>
3         {{include}}
4         <div class="sidebar">
5             {{if 'mysidebar' in globals():}}{{mysidebar()}}{{else:}}
6                 my default sidebar
7             {{pass}}
8         </div>
9     </body>
10    </html>
```

そして、次の拡張ビューも考えます。

```
1 {{def mysidebar():}}
2 my new sidebar!!!
3 {{return}}
4 {{extend 'layout.html'}}
5 Hello World!!!
```

ここで、関数が{{extend...}}文の前に定義されていることに注意してください。 – これによりその関数は”layout.html” コードが実行される前に作成され、その関数は”layout.html” 内の任意の場所、例えば{{include}}の前でも呼び出することができます。また、関数が、拡張元のビューにおいて = プレフィックスが付かずに組み込まれていることに注意してください。

このコードは次のような出力を生成します：

```
1 <html>
2     <body>
```

```

3     Hello World!!!
4     <div class="sidebar">
5         my new sidebar!!!
6     </div>
7     </body>
8 </html>

```

この関数が HTML(これはまた Python コードを含むことができます) の中で定義されることで、`response.write` が内容を書くために使用されていることに注意してください(関数は内容を返しません)。レイアウトで`{{=mysidebar()}}`ではなく`{{mysidebar()}}`を用いてビュー関数を呼び出している理由は、ここにあります。このような方法で定義された関数は、引数を取ることができます。

## 5.7 ビュー内のブロック

ビューをよりモジュール化するためのもう 1 つの方法は`{{block...}}`を用いることです。この仕組は、上記のセクションで説明した仕組みの代替案になります。

次の”layout.html”を考えます：

```

1 <html>
2   <body>
3     {{include}}
4     <div class="sidebar">
5       {{block mysidebar}}
6         my default sidebar
7       {{end}}
8     </div>
9   </body>
10 </html>

```

そして、次の拡張先ビューも考えます。

```

1 {{extend 'layout.html'}}
2 Hello World!!!
3 {{block mysidebar}}
4 my new sidebar!!!
5 {{end}}

```

これは次のような出力を生成します：

```

1 <html>
2   <body>
3     Hello World!!!

```

```

4   <div class="sidebar">
5     my new sidebar!!!
6   </div>
7   </body>
8 </html>

```

ブロックは複数定義することができます。拡張元のビューにブロックがあるが、拡張先のビューにはない場合、拡張元のビューの内容が使用されます。また関数と異なり、`{{extend ...}}`の前にブロックを定義する必要はありません。 – `extend` の後で定義されても、拡張元ビュー中の任意の場所で置換できます。

ブロックの内部では、`{{super}}`という式を使用して、親の内容を組み込むことができます。例えば、上記の拡張先ビューを次のように置換する場合:

```

1 {{extend 'layout.html'}}
2 Hello World!!!
3 {{block mysidebar}}
4 {{super}}
5 my new sidebar!!!
6 {{end}}

```

以下が得られます:

```

1 <html>
2   <body>
3     Hello World!!!
4     <div class="sidebar">
5       my default sidebar
6       my new sidebar!!!
7     </div>
8   </body>
9 </html>

```

第3版 - 翻訳: 細田謙二 レビュー: Omi Chiba

第4版 - 翻訳: 細田謙二 レビュー: Hitoshi Kato

# 6

## データベース抽象化レイヤ

### 6.1 依存関係

web2py は、データベース抽象化層 (DAL) を備えています。これは、Python オブジェクトを、クエリやテーブル、レコードなどのデータベース・オブジェクトに対応付けする API です。DAL は、データベース・バックエンド固有の方言を用いて SQL をリアルタイムで動的に生成します。そのため、開発者は SQL コードを書く必要がなく、また、異なる SQL 方言を学ぶ必要もありません (SQL という言葉は総称的に用いています)。そして、アプリケーションは異なるタイプのデータベース間でポータブルになります。この文書の執筆時点では、サポートされているデータベースは、SQLite (Python に備わっています)、PostgreSQL、MySQL、Oracle、MSSQL、FireBird、DB2、Informix、Ingres、(部分的に)the GoogleApp Engine (SQL 及び NoSQL) です。実験的に他の多くのデータベースもサポートしています。web2py のウェブサイトやメーリングリストで最新の適応状況を確認してください。Google NoSQL は、第 13 章で具体的な事例として扱います。

Windows のバイナリ・ディストリビューションは、SQLite と MySQL と一緒にすぐ使えます。Mac のバイナリ・ディストリビューションは、SQLite と一緒にすぐに使えます。他のデータベースバックエンドを使うには、ソース・ディストリビューションから実行し、バックエンドに必要な適切なドライバをインストールしてください。

適切なドライバをインストールしたら、web2py をソースから起動してください。  
web2py はドライバを見つけます。以下はドライバのリストです：

*database / driver (source) SQLite / sqlite3 or pymysql2 or zxJDBC [58] (on Jython) PostgreSQL / psycopg2 [59] or zxJDBC [58] (on Jython) MySQL / pymysql [60] or MySQLdb [61] Oracle / cx\_Oracle [62] MSSQL / pyodbc [63] FireBird / kinterbasdb [64] DB2 / pyodbc [63] Informix / informixdb [65] Ingres / ingresdbi [66]*

(pymysql ships with web2py) web2py は、DAL を構成する次のクラスを定義しています：

DAL は、データベース接続を表します。例：

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

Table はデータベースのテーブルを表します。Table を直接インスタンス化するのではなく、代わりに、`DAL.define_table` によってインスタンス化します。

```
1 db.define_table('mytable', Field('myfield'))
```

Table の中で最も重要なメソッドは以下のものです： `.insert`, `.truncate`, `.drop`, and `.import_from_csv_file`.

Field はデータベースのフィールドを表します。インスタンス化し、`DAL.define_table` へ引数として渡すことができます。

DAL Rows はデータベースの選択によって返されるオブジェクトです。`Row` の行からなるリストとして考えることができます：

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

Row はフィールドの値を保持します。

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

Query は SQL の”where” 句を表現するオブジェクトです：

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

Set はレコードのセットを表します。最も重要なメソッドは、`count`、`select`、`update`、`delete` です。例：

```

1 myset = db(myquery)
2 rows = myset.select()
3 myset.update(myfield='somevalue')
4 myset.delete()

```

**Expression** は `orderby` や `groupby` 式のようなものです。Field クラスは、`Expression` から派生しています。下に例を示します。

```

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

```

## 6.2 接続文字列

データベースとの接続は、DAL のオブジェクトのインスタンスを作成することによって確立されます：

```
>>> db = DAL('sqlite://storage.db', pool_size=0)
```

`db` はキーワードではありません。それはローカルな変数で、接続オブジェクト `DAL` を格納します。違う名前を付けても問題ありません。`DAL` のコンストラクタは 1 つの引数、すなわち接続文字列を必要とします。接続文字列は、特定のバックエンドのデータベースに依存する唯一の web2py コードです。ここでは、サポートされているバックエンドのデータベースの具体的な接続文字列の例を示します(全てのケースで、データベースは、localhost のデフォルトポート上で動作し、“test”という名前だと仮定しています)：

<b>SQLite</b>	/	sqlite://storage.db	<b>MySQL</b>	/
mysql://username:password@localhost/test			<b>PostgreSQL</b>	
/		postgres://username:password@localhost/test		
<b>MSSQL</b>	/	mssql://username:password@localhost/test		
<b>FireBird</b>	/	firebird://username:password@localhost/test		
<b>Oracle</b>	/	oracle://username/password@test		
<b>DB2</b>	/	db2://username:password@test	<b>Ingres</b>	/
ingres://username:password@localhost/test			<b>Informix</b>	/
informix://username:password@test			<b>Google App Engine/SQL</b>	/
google:sql			Google App Engine/NoSQL	/ google: datastore

SQLite ではデータベースが、単一のファイルからなることに注意してください。ファイルが存在しない場合は作成されます。このファイルはアクセスするたび

にロックされます。MySQL、PostgreSQL、MSSQL。FireBird、Oracle、DB2、Ingres、Informix の場合、”test” データベースは web2py の外部で作成される必要があります。接続が確立されると、web2py は、テーブルを適切に作成、変更、削除します。

接続文字列を `None` に設定することも可能です。この場合、DAL はいかなるバックエンド・データベースにも接続しませんが、テスト用途として API にはアクセス可能です。この例は、第 7 章で説明します。

### 6.2.1 接続プール

DAL のコンストラクタの 2 番目の引数は `pool_size` です。デフォルトでは 0 になります。

各リクエストに対して新規のデータベース接続を確立するのはそれなりに遅いので、web2py は接続プール機構を実装しています。接続が確立し、ページが処理され、トランザクションが完了すると、その接続は閉じず、プールにされます。次の HTTP リクエストが来ると、web2py はそのプールから接続を取得して、新規のトランザクションに利用しようと試みます。もしプールに利用可能な接続が存在しないと、新しい接続を確立します。

`pool_size` パラメタは、SQLite と GAE では無視されます。

プール内の接続は、スレッド間で順番に共有されます。つまり、それらは 2 つの異なるスレッドによって利用されますが、同時には利用されません。また、それぞれの web2py のプロセスには、1 つのプールしかありません。web2py の起動時は、プールは常に空です。プールは、`pool_size` の値か最大同時リクエスト数の、どちらか少ない方まで増えます。つまり、`pool_size=10` でも、サーバーが 5 より多いリクエスト数を受け付けない場合、実際のプールサイズは 5 までしか成長しません。`pool_size=0` の場合は、接続プールは使用されません。

接続プールは SQLite の場合は無視されます。特に恩恵がないからです。

### 6.2.2 接続の失敗

web2py がデータベースへの接続に失敗した場合、1 秒間待機し、さらに接続の試みを 5 回、失敗の確定までに行います。接続プールではプールされている接

続の中で、オープンになっているが暫く使用していないものを、データベース側でクローズにすることが可能です。再試行の機能により web2py は、切断した接続に対する再確立を試みます。

接続プールを使って接続を行うと、プールに戻されリサイクルされます。プール中の使用されていない接続は、データベースサーバーによってクローズされることもあります。これは、障害やタイムアウトによって発生します。web2py はその発生を検知し、接続を再確立します。

### 6.2.3 複製されたデータベース

DAL(...) の最初の引数には、URI のリストをとることもできます。この場合、web2py はそれぞれに接続しようと試みます。その主な目的は、複数のデータベースサーバーに対応し、それらの間で負荷を分散させることです。ここでは典型的なユースケースを示します：

```
1 db = DAL(['mysql://...1', 'mysql://...2', 'mysql://...3'])
```

この場合、DAL は最初のものに接続しようと試み、失敗したら、第 2、第 3 のものに試みます。これは、マスター-スレーブ構成のデータベースにおいて負荷を分散するためにも利用できます。詳細は、第 13 章のスケーラビリティの中で説明します。

## 6.3 予約キーワード

DAL のコンストラクタに渡すことのできる、もう 1 つの引数があります。これは、対象となるバックエンドのデータベースにおける予約された SQL のキーワードに対してテーブルの名前や、カラムの名前をチェックすることができます。

その引数は、`check_reserved` です。デフォルトは `None` です。

これは、データベース・バックエンドのアダプタの名前を含む文字列のリストです。

アダプタの名前は、DAL の接続文字列において使用されているものと同じです。例えば、PostgreSQL と MSSQL に対してチェックしたい場合は、次のような接続文字列になります：

```

1 db = DAL('sqlite://storage.db',
2           check_reserved=['postgres', 'mssql'])

```

DAL はリストと同じ順番で、キーワードを走査します。

”all” と”common” という 2 つの追加オプションがあります。all を指定すると、全ての知られている SQL キーワードに対してチェックされます。common を指定すると、SELECT、INSERT、UPDATE などの一般的な SQL のキーワードだけがチェックされます。

サポートされるバックエンドに対して、非予約語の SQL キーワードをチェックするかどうかを指定することも可能です。この場合、\_nonreserved をその名前に追加してください。例：

```
1 check_reserved=['postgres', 'postgres_nonreserved']
```

以下のデータベース・バックエンドは、予約語のチェックをサポートしています。

**PostgreSQL / postgres (\_nonreserved) MySQL / mysql FireBird  
/ firebird(\_nonreserved) MSSQL / mssql Oracle / oracle**

#### 6.4 DAL, Table, Field

DAL の API を理解する最良の方法は、それぞれの関数を自分で試してみることです。これは、web2py のシェルを介してインタラクティブで実行することができます。とはいっても最終的に、DAL コードはモデルとコントローラに設置します。

接続を作成して始めましょう。例なので、SQLite を使用してもよいでしょう。バックエンドのエンジンを変更したとしても、この記事では他に何も変更するものはありません。

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

データベースは今接続されて、その接続はグローバル変数 db に格納されます。

いつでも、接続文字列を取り出すことができます。

```

1 >>> print db._uri
2 sqlite://storage.db

```

データベース名も取り出せます。

```

1 >>> print db._dbname
2 sqlite

```

接続文字列は`_uri`と呼ばれます。これは、Uniform Resource Identifier のインスタンスだからです。

DAL では、同じデータベースや異なるデータベース、さらに、異なる種類のデータベースに対する複数の接続が可能です。ここでは、最も一般的な状況として、単一のデータベースを想定します。

DAL の最も重要なメソッドは `define_table` です：

```

1 >>> db.define_table('person', Field('name'))

```

これは、”name” フィールド（カラム）を持つ”person” という `Table` オブジェクトを定義し、格納し、返しています。このオブジェクトはまた、`db.person` に関連付けられているので、その戻り値を捉える必要はありません。web2py が何れにせよ作成するため、”id” というフィールドは宣言しないでください。全てのテーブルは、”id” というフィールドをデフォルトで持っています。これは、(1 から始まる) 自動インクリメントした整数のフィールドで、相互参照や、各レコードをユニークにするために用いられます。すなわち、”id” はプライマリーキーです。（注：id が 1 から始まるかはバックエンドによります。例えばこれは、Google App Engine NoSQL では適用されません。）

オプション的に、`type='id'` とするフィールドを定義することができます。web2py はこのフィールドを自動インクリメントした `id` フィールドとして使用します。これは、レガシーなデータベーステーブルにアクセスする時以外には推奨されません。いくつかの制約がありますが、複数の異なるプライマリキーを使用することもできます。これについては、” レガシー・データベースとキー付きテーブル ” のセクションで後述します。

## 6.5 レコードの表現

これはオプションですが、レコードの書式表現を指定するのを推奨されています：

```

1 >>> db.define_table('person', Field('name'), format='%(name)s')

```

または

```

1 >>> db.define_table('person', Field('name'), format='%(name)s %(id)s')

```

より複雑なものは関数を用いて次のように書きます：

```
1 >>> db.define_table('person', Field('name'),
2                      format=lambda r: r.name or 'anonymous')
```

format 属性は、2つの目的のために使用されます：

- セレクト/オプションのドロップダウンにおいて、参照先のレコードを表現するため。
- このテーブルを参照する全てのフィールドに対して、`db.othertable.person.represent` 属性を設定するためです。これは、SQLTABLE が id によって参照を表示するのではなく、代わりに好ましい書式表現を用いることを意味します。

以下に示すのは Field コンストラクタのデフォルトの値です：

```
1 Field(name, 'string', length=None, default=None,
2        required=False, requires='<default>',
3        ondelete='CASCADE', notnull=False, unique=False,
4        uploadfield=True, widget=None, label=None, comment=None,
5        writable=True, readable=True, update=None, authorize=None,
6        autodelete=False, represent=None, compute=None,
7        uploadfolder=os.path.join(request.folder, 'uploads'),
8        uploadseparate=None)
```

全てのものが各フィールドに関連しているというわけではありません。”length” は、”string” 型のフィールドに対してのみ関連しています。”uploadfield” と”authorize” は、”upload” 型のフィールドに対してのみ関連しています。”ondelete” は”reference” と”upload” 型のフィールドに対してのみ関連しています。

- `length` は”string” や、”password”、”upload” フィールドの最大長を設定します。`length` が指定されていない場合は、デフォルト値が使用されます。ただし、デフォルト値に関しては後方互換は保証されていません。意図しないマイグレーションやアップグレードを避けるため、`string`、`password`、`upload` フィールドに対して、常に `length` を指定することを推奨します。
- `default` はフィールドのデフォルト値を設定します。デフォルト値は値が明示的に指定されていない場合において、挿入を実行したときに使用されます。また、SQLFORM を用いてテーブルから構築されたフォームで、事前入力のために使用されます。注意点として、固定値を指定する代わりに、適切な値を返す関数 (lambda を含む) を使用することができます。この場合、複数の

レコードが一つのトランザクションで挿入されても、各レコードが挿入されるたびに関数が呼び出されます。

- `required` は DAL に、このフィールドの値が明示的に指定されていない場合、挿入することを許さないようにします。
- `requires` はバリデータ、または、バリデータのリストです。これは、DAL によっては使用されず、SQLFORM によって使用されます。以下に、与えられた型に対するデフォルトのバリデータの一覧を示します：

```
field type / default field validators string / IS_LENGTH(length)
default length is 512 text / IS_LENGTH(65536) 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 <table> / IS_IN_DB(db,table.field,format)
list:string / None list:integer / None list:reference <table> /
IS_IN_DB(db,table.field,format,multiple=True)
```

`Decimal` は、Python の `decimal` モジュールに定義されているような、`Decimal` オブジェクトを要求し返します。SQLite では `decimal` 型は処理されないので、`double` として扱われます。`(n, m)` はそれぞれ、合計の桁数と小数点以下の桁数です。

`list:` フィールドは特殊です。なぜなら、NoSQL 上の特定の非正規化の特徴 (Google App Engine NoSQL では、`ListProperty` や `StringListProperty` といったフィールド型) に対して有利になるように、そして、それらを他のサポートされたリレーションナル・データベースに移植できるように設計されているからです。リレーションナルデータベースでは、リストは `text` フィールドとして格納されます。項目は、`|` によって区切られ、文字列項目の各 `|` は `||` にエスケープされます。詳細は、`list` フィールドのセクションで説明します。

`requires=...` は、フォーム・レベルで強制され、`required=True` は DAL(挿入) レベルで強制されることに注意してください。一方、`notnull` や `unique`、`onDelete` はデータベース・レベルで強制されます。それらは時として冗長に見えるかもしれません、DAL を用いたプログラミングにおいて、その区別を管理することは重要です。

- `onDelete` は”ON DELETE”SQL 文へと変換されます。デフォルトでは、”CASCADE” に設定されています。これは、レコードを削除する時に、それを参照している全てのレコードを削除するようにデータベースに指示します。この機能を無効にするには、`onDelete` を”NO ACTION”、または”SET NULL” に設定してください。
- `notnull=True` は”NOT NULL”SQL 文へと変換されます。これにより、データベースから、このフィールドに `null` 値が挿入されることを防ぎます。
- `unique=True` は”UNIQUE”SQL 文へと変換され、フィールドの値が、そのテーブル内でユニークであることを保証します。これはデータベース・レベルで強制されます。
- `uploadfield` は ”upload” 型のフィールドに対してのみ適用されます。”upload” 型のフィールドはどこか他に保存されたファイル、デフォルトではアプリケーションの”uploads/” フォルダ以下のファイルシステム上に保存されたファイルの名前を格納します。`uploadfield` が設定されている場合、ファイルは同じテーブルの `blob` フィールドに格納され、`uploadfield` の値はその `blob` フィールドの名前になります。この点に関しては SQLFORM のコンテキストでより詳しく後述します。
- `uploadfolder` はデフォルトではアプリケーションの”uploads/” フォルダになります。別のパスが設定されている場合、ファイルは別のフォルダにアップロードされます。例えば、`uploadfolder=os.path.join(request.folder, 'static/temp')` とすると、ファイルは `web2py/applications/myapp/static/temp` フォルダにアップロードされます。
- `uploadseparate` は、`True` に設定されていると、ファイルは `uploadfolder` フォルダの異なるサブフォルダの下にアップロードされます。これは、非常に多くのファイルを同じフォルダ/サブフォルダに置くことを回避するために最適化されています。注意：システムの中止なしに、`uploadseparate` の値を `True` から `False` に変更することはできません。web2py は分解したサブフォルダを使用するかしないかのどちらかしかとることはできません。ファイルをアップロードしてからこの挙動を変更すると、web2py はそれらのファイルを取り出すことができなくなります。これが起こった場合は、ファイルを移動し、問題を解決することは可能ですが、ここでは説明しません。

- `widget` は、利用可能なウィジェット・オブジェクトの 1 つである必要があります。カスタム・ウィジェットや `SQLFORM.widgets.string.widget` などです。利用可能なウィジェットのリストは後述します。各フィールドの型は、デフォルトのウィジェットを持ちます。
- `label` は自動生成されるフォームで、このフィールドに使用するラベルを保持する文字列（または文字列にシリアル化できるもの）です。
- `comment` は、このフィールドに関連付けられたコメントを保持し、自動生成されるフォームにおいて入力フィールドの右側に表示される文字列（または文字列にシリアル化できるもの）です。
- `writable` もしフィールドが `writable` だと、自動生成される作成と更新フォームにおいて編集可能になります。
- `readable` もしフィールドが `readable` だと、読み取り専用フォームにおいて表示されます。もしフィールドが `readable` でも `writable` でもない場合、作成と更新フォームにおいてフィールドは表示されません。
- `update` は、レコード更新時の、このフィールドに対するデフォルト値になります。
- `compute` は、オプション的な関数です。レコードが挿入、または、更新されたとき、`compute` 関数が実行され、フィールドには戻り値が設定されます。レコードは `compute` 関数に `dict` として渡されます。その `dict` には、そのフィールドの現在の値や、他のどの `compute` フィールドの値も含まれていません。
- `authorize` は、”upload” フィールドのみで、対応するフィールドのアクセスコントロール要求に使用できます。詳細は、認証と承認のコンテキストにて後述します。
- `autodelete` は、アップロードされたファイルを参照するレコードが削除された場合、対応するファイルを削除するかどうかを決定します。”upload” フィールドに対してのみ有効です。
- `represent` は、`None`、または、フィールド値を受け取りフィールド値の代替表現として値を返す関数を指定できます。例:

```
1 db.mytable.name.represent = lambda name, row: name.capitalize()
2 db.mytable.other_id.represent = lambda id, row: row.myfield
3 db.mytable.some_uploadfield.represent = lambda value, row: \
4     A('get it', _href=URL('download', args=value))
```

”blob” フィールドもまた特別です。デフォルトでは、バイナリデータは、実際のデータベースフィールドに格納される前に、base64 でエンコードされ、抽出時にデコードされます。これには、blob フィールドに必要なものより 25%余分に記憶領域を使用するというマイナスの効果がありますが、2 つの利点があります。平均で web2py とデータベースサーバー間のデータ通信量を削減します。そして、通信をバックエンド固有のエスケープ規則から独立させます。

多くのフィールドやテーブル属性は定義された後でも変更可能です：

```
1 db.define_table('person', Field('name', default=''), format='%(name)s')
2 db.person._format = '%(name)s/%(id)s'
3 db.person.name.default = 'anonymous'
```

(テーブルの属性はフィールド名との衝突を避けるために下線を接頭文字として使用している点に注意してください)。

データベース接続に対して、定義されているテーブル一覧を問い合わせることができます。

```
1 >>> print db.tables
2 ['person']
```

定義されているフィールド一覧に関しても、テーブルに問い合わせることができます：

```
1 >>> print db.person.fields
2 ['id', 'name']
```

テーブルの型を問い合わせることができます：

```
1 >>> print type(db.person)
2 <class 'gluon.sql.Table'>
```

また DAL 接続から、次のようにテーブルにアクセスすることができます：

```
1 >>> print type(db['person'])
2 <class 'gluon.sql.Table'>
```

同様にフィールドの名前から、同等な複数の方法でフィールドにアクセスすることができます：

```
1 >>> print type(db.person.name)
2 <class 'gluon.sql.Field'>
3 >>> print type(db.person['name'])
4 <class 'gluon.sql.Field'>
```

```
5 >>> print type(db['person']['name'])
6 <class 'gluon.sql.Field'>
```

フィールド名を指定して、定義で設定された属性にアクセスすることができます：

```
1 >>> print db.person.name.type
2 string
3 >>> print db.person.name.unique
4 False
5 >>> print db.person.name.notnull
6 False
7 >>> print db.person.name.length
8 32
```

親のテーブルやテーブル名、親の接続にもアクセスできます：

```
1 >>> db.person.name._table == db.person
2 True
3 >>> db.person.name._tablename == 'person'
4 True
5 >>> db.person.name._db == db
6 True
```

フィールドはメソッドを持っています。後述しますがクエリーを作成する際に使用する場合があります。フィールドオブジェクトの特別なメソッドは `validate` で、そのフィールドに対するバリデータを呼び出します。

```
1 print db.person.name.validate('John')
```

これは `(value, error)` のタプルを返します。入力値がバリデータを通った場合、`error` は `None` になります。

## 6.6 マイグレーション

`define_table` は、対応するテーブルが存在するかどうかをチェックします。存在しない場合は、それを作成する SQL を生成し、その SQL を実行します。テーブルが存在しても定義されているものと違うものであれば、そのテーブルを変更する SQL を生成し実行します。フィールドの型を変更し名前は変更しない場合、データを変更しようと試みます（そうしたくない場合は、テーブルを二度、定義し直す必要があります。一度目はフィールドを除くことによって、そのフィールドを削除するように `web2py` に指示します。二度目は、新規に定義したフィー

ルドを加えて、web2py に作らせます)。テーブルが存在し現在の定義と一致する場合は、そのままになります。全ての場合において、そのテーブルを表現する db.person オブジェクトが作られます。

このような挙動を、ここでは”マイグレーション”と言います。web2py は全てのマイグレーションとマイグレーションの試みを、”databases/sql.log” ファイルにログとして記録します。

define\_table の最初の引数は常にテーブルの名前です。他の無名引数はフィールド (Field) です。この関数はまた、”migrate” という省略可能な最後の引数をとることができます。これは、次のように名前によって明示的に参照されなければなりません：

```
1 >>> db.define_table('person', Field('name'), migrate='person.table')
```

migrate の値は、(アプリケーションの”database” フォルダ内の) ファイル名です。このファイルには、このテーブルの内部的なマイグレーション情報が web2py によって格納されています。これらのファイルはとても重要で、データベース全体を削除するとき以外には、削除すべきではありません。削除する場合は、”.table” ファイルを手動で削除する必要があります。デフォルトでは、migrate は True に設定されています。こうすると、web2py は接続文字列のハッシュからファイル名を生成します。migrate が False に設定されていると、マイグレーションは実行されません。web2py は、データベースにテーブルが存在し、define\_table に列挙されたフィールドを含んでいると想定します。ベストプラクティスは、明示的な名前をこの migrate テーブルに与えることです。

同じアプリケーションに、同じ migrate ファイルを持つ 2 つのテーブルが存在することはありません。

DAL クラスはまた、”migrate” 引数をとります。これは、define\_table が呼び出されたときの migrate のデフォルト値を設定します。例：

```
1 >>> db = DAL('sqlite://storage.db', migrate=False)
```

このようにすると、db.define\_table が migrate 引数なしに呼び出されたときは常に、migrate のデフォルト値が False に設定されます。

接続時にマイグレーションを全てのテーブルに対して無効にすることもできます。

```
1 db = DAL(...,migrate_enabled=False)
```

これは 2 つのアプリケーションが同じデータベースを共有する場合に推奨されます。2 つの内、1 つのアプリケーションでマイグレーションを実行し、もう一方は無効にすることです。

## 6.7 壊れたマイグレーションの修復

マイグレーションには一般的に 2 つの問題があり、それらを修復する方法があります。

1 つの問題は、SQLite 固有のものです。SQLite は、カラムの型を強制せず、また、カラムを削除することができません。したがって、文字列型のカラムを持っている、それを削除した場合、それは実際には削除されません。異なる型のカラムを再び加えよう（例えば datetime）とした場合、（実質的にゴミとなる）文字列が含まれる datetime カラムを作ってしまうことになります。web2py はこれに対してエラーを出しません。なぜならレコードを取得しようとして失敗するまでは、データベースに何が入っているか分からないからです。

もし web2py が、レコード選択時に gluon.sql.parse 関数でエラーを返す場合、これは上記の問題による、カラム内の壊れたデータの問題になります。

解決方法は、テーブルの全てのレコードを更新し、問題となっているカラムの値を None に更新することです。

もう 1 つの問題は、より一般的ですが、MySQL で典型的に見られるものです。MySQL は、トランザクション中に複数の ALTER TABLE を許可しません。これは、web2py が、複雑なトランザクションを小さなもの（一度に 1 つの ALTER TABLE）に分解しなければならず、一つ一つコミットしなければならないことを意味します。したがって、複雑なトランザクションの一部がコミットされ、別の部分が失敗して、web2py を壊れた状態にしてしまう可能性があります。なぜトランザクションの一部が失敗するでしょうか？なぜなら、例えば、テーブルを変更し、文字列カラムを日付カラムに変更しようとしたとき、web2py がそれらのデータの変換を試みますが、しかしデータ変換ができない場合があるからです。web2py はどうなるのでしょうか？データベースに実際に格納したテーブル構造は正確に何なのか、ということについて混乱します。

解決策は、全てのテーブルに対する migration を無効にし、次のように、fake migration を有効にすることです：

```
1 db.define_table(...,migrate=False,fake_migrate=True)
```

これにより、テーブルに関する web2py のメタデータは、テーブル定義に従って再構築されます。(マイグレーションが失敗する前のものと後のものの) どれが機能するか、複数のテーブル定義で試してみてください。一旦成功した後は、`fake_migrate=True` 属性を削除してください。

**マイグレーションの問題を修復しようとする前に**、”`applications/yourapp/databases/*.table`” ファイルのコピーをとっておくのが賢明です。

全テーブルを一度に、マイグレーション問題の修復を行うこともできます。

```
1 db = DAL(...,fake_migrate_all=True)
```

しかし失敗した場合、原因を突き詰めていく手助けにはなりません。

## 6.8 挿入

テーブルを指定して、レコードを挿入することができます

```
1 >>> db.person.insert(name="Alex")
2 1
3 >>> db.person.insert(name="Bob")
4 2
```

挿入は、挿入したそれぞれのレコードのユニークな”`id`” 値を返します。

テーブルを切捨てるることができます。つまり、全てのレコードを削除し、`id` のカウンタを元に戻します。

```
1 >>> db.person.truncate()
```

このとき、もう一度レコードを挿入した場合、カウンタは 1 から始まります(これはバックエンド固有で、Google NoSQL には適用されません) :

```
1 >>> db.person.insert(name="Alex")
2 1
```

web2py は `bulk_insert` メソッドも用意しています。

```
1 >>> db.person.bulk_insert([{'name': 'Alex'}, {'name': 'John'}, {'name': 'Tim'}])
2 [3, 4, 5]
```

これは、挿入されるフィールドの辞書のリストを受け取り、複数の挿入を一度に実行します。そして挿入された複数のレコードの ID を返します。サポートされているリレーションナルデータベースでは、この関数を使用した場合と、ループさせて個別に挿入をした場合を比べても、特に利点はありません。しかし、Google App Engine では、大幅な高速化が見込めます。

## 6.9 コミットとロールバック

いかなる作成、削除、挿入、切捨て、削除、更新操作も、コミットコマンドが発行されるまでは、実際にはコミットされません。

```
1 >>> db.commit()
```

確認のため、新規のレコードを挿入してみましょう：

```
1 >>> db.person.insert(name="Bob")
2
```

そしてロールバックします。つまり、最後にコミットした時点からの全ての操作を無効にします：

```
1 >>> db.rollback()
```

再び挿入すると、前回の挿入はロールバックされたので、カウンタは再び 2 に設定されます。

```
1 >>> db.person.insert(name="Bob")
2
```

モデル、ビュー、コントローラ内のコードは、web2py のコードで次のように囲まれます：

```
1 try:
2     execute models, controller function and view
3 except:
4     rollback all connections
5     log the traceback
6     send a ticket to the visitor
7 else:
8     commit all connections
9     save cookies, sessions and return the page
```

web2pyにおいてコミットやロールバックを明示的に呼び出すことは、より細かい制御を望まない限り、必要ありません。

## 6.10 生の SQL

### 6.10.1 クエリ実行時間の計測

全てのクエリは、web2py によって実行時間を自動計測します。`db._timings` 変数はタプルのリストです。それぞれのタプルはデータベース・ドライバに渡された生の SQL とその実行時間を秒数で持っています。この変数は toolbar を使用して表示できます。

```
1 {{=response.toolbar()}}
```

### 6.10.2 executesql

DAL は、SQL 文を明示的に発行することを可能にします。

```
1 >>> print db.executesql('SELECT * FROM person;')
2 [(1, u'Massimo'), (2, u'Massimo')]
```

この場合、戻り値は、DAL によって構文解析や変換されることはなく、そのフォーマットは特定のデータベース・ドライバに依存します。select での使用は通常は必要ありませんが、インデックスの使用ではより一般的です。executesql は 2 つのオプション引数をとります：`:placeholders` と `as_dict` です。`:placeholders` は、SQL で置換されるオプションの値の配列、もしくは DB ドライバでサポートされいれば、SQL の名前付きのプレースホルダーに一致するキーを持つ辞書です。

`as_dict` が True に設定されていると、DB ドライバによって返される結果のカーソルは、db フィールド名をキーとして持つ辞書の配列に変換されます。`as_dict = True` で返された結果は、通常の select に `.as_list()` を適用した時に返されるものと同様のものになります。

```
1 [{field1: value1, field2: value2}, {field1: value1b, field2: value2b}]
```

### 6.10.3 \_lastsql

SQL が executesql を用いて手動で実行されても、DAL によって生成された SQL でも、`db._lastsql` で SQL のコードを常に見ることができます。これは、デバッグに便利です：

```

1 >>> rows = db().select(db.person.ALL)
2 >>> print db._lastsql
3 SELECT person.id, person.name FROM person;

```

*web2py* は “\*” 演算子を使ったクエリを生成することはありません。*web2py* では常に、明示的にフィールドを選択します。

### 6.11 drop

最後に、テーブルを削除することができ、全てのデータは失われます：

```

1 >>> db.person.drop()

```

### 6.12 インデックス

現在 DAL の API は、テーブルにインデックスを作成するコマンドを提供していませんが、これは `executesql` コマンドによって行うことができます。その理由は、既存のインデックスではマイグレーションが複雑になり、それを明示的に扱ったほうが良いからです。インデックスは、クエリで頻繁に使用されているフィールドに対して必要になります。

次に示すのは、SQLiteにおいてSQLを使用してインデックスを作成する例です：

```

1 >>> db = DAL('sqlite://storage.db')
2 >>> db.define_table('person', Field('name'))
3 >>> db.executesql('CREATE INDEX IF NOT EXISTS myidx ON person (name);')

```

他のデータベースの方言は非常に似た構文を持っていますが、オプション的な “IF NOT EXISTS” 宣言をサポートしていないことがあります。

### 6.13 レガシー・データベースとキー付きテーブル

*web2py* は、いくつかの条件の下で、レガシー・データベースに接続することができます。

最も簡単な方法は、以下の条件を満たしている時です：

- 各テーブルは、必ず”id”と呼ばれる一意で自動インクリメントした整数フィールドを持つ
- レコードは、必ず”id”フィールドを用いてのみ参照される

既存のテーブルにアクセスする時、つまり、テーブルが現在のアプリケーションの *web2py* によって作成されていない場合、常に `migrate=False` としてください。

レガシー・テーブルが自動インクリメントした整数フィールドを持つが、それが”id”と呼ばれていない場合でも、*web2py* はアクセス可能です。しかしこの場合、テーブル定義に、`Field('....', 'id')` として明示的に含めなければなりません。ここで…は、自動インクリメントした整数フィールドの名前です。

最後に、レガシー・テーブルが自動インクリメント `id` でないプライマリキーを使用していた場合、次の例のように、キー付きテーブルを用いてアクセスすることが可能です：

```

1 db.define_table('account',
2     Field('accnum', 'integer'),
3     Field('acctype'),
4     Field('accdesc'),
5     primarykey=['accnum', 'acctype'],
6     migrate=False)

```

- `primarykey` はプライマリキーを構成するフィールド名のリストです。
- 指定されなくても全ての `parimarykey` フィールドは `NOT NULL` がセットされています。
- キー付きテーブルは他のキー付きテーブルだけを参照できます。
- 参照するフィールドは `reference tablename.fieldname` フォーマットを使用しなければなりません。
- `update_record` 関数を、キー付きテーブルの `Rows` に使用することはできません。

ただし現在、これは *DB2* と *MS-SQL*、*Ingres*、*Informix* に対してのみ利用可能です。しかし、他のデータベースにも簡単に加えることができます。

執筆時点で、`primarykey` 属性が、全てのレガシー・テーブルと、サポートされたデータベース・バックエンドに対して機能することは保障されていません。シ

ンプルにするため可能なら、自動インクリメントした id フィールドを持つデータベースのビューを作成することを、お勧めします。

### 6.14 分散トランザクション

執筆時点では、この機能は *PostgreSQL*、*MySQL*、*Firebird* に対してのみサポートされています。これらは 2相コミットの API を公開しているためです。

個別の PostgreSQL データベースに接続する 2つ（またはそれ以上）の接続を持っていると仮定します：

```
1 db_a = DAL('postgres://...')  
2 db_b = DAL('postgres://...')
```

モデルやコントローラにおいて、それらを同時にコミットすることが可能です：

```
1 DAL.distributed_transaction_commit(db_a, db_b)
```

失敗した場合は、この関数はロールバックして、`Exception` を発生させます。

コントローラで 1つのアクションが返された時、もし 2つの別個の接続を持ち、かつ、上記の関数を呼び出していない場合は、web2py はそれらを個別にコミットします。これは、1つのコミットが成功し、もう一つが失敗するという可能性があることを意味します。分散トランザクションはこのようなことが起こるのを防ぎます。

### 6.15 手動アップロード

次のモデルを考えてください：

```
1 >>> db.define_table('myfile', Field('image', 'upload'))
```

通常、挿入は、SQLFORM や crud フォーム (SQLFORM の 1つ) を介して自動的に処理されます。しかし場合によっては、ファイルシステム上にすでにファイルがあり、プログラムでアップロードしたいことがあります。これは次のような方法で行うことができます：

```
1 >>> stream = open(filename, 'rb')  
2 >>> db myfile.insert(image=db myfile.image.store(stream, filename))
```

upload フィールドオブジェクトの `store` メソッドは、ファイルストリームとファイル名を受け取ります。ファイル名はファイルの拡張子(型)を決めるのに使用され、(web2py のアップロード機構に従って)そのファイルのための新しい仮の名前を作成し、(特に指定がなければ uploads フォルダの下)その新しい仮のファイルにファイルの内容をロードします。そして、新しい仮のファイル名が返され、`db myfile` テーブルの `image` フィールドに格納されます。

`.store` の逆は `.retrieve` になります:

```
1 >>> row = db(db myfile).select().first()
2 >>> (filename, stream) = db myfile.image.retrieve(row.image)
3 >>> import shutil
4 >>> shutil.copyfileobj(stream, open(filename, 'wb'))
```

### 6.16 Query, Set, Rows

再び、先ほど定義した(削除した)テーブルを考え、3つのレコードを挿入してみます:

```
1 >>> db.define_table('person', Field('name'))
2 >>> db.person.insert(name="Alex")
3 1
4 >>> db.person.insert(name="Bob")
5 2
6 >>> db.person.insert(name="Carl")
7 3
```

テーブルは変数に格納することができます。例えば、`person` 変数として利用することができます:

```
1 >>> person = db.person
```

フィールドも、`name` のように変数に格納することができます。例えば次のようにすることができます:

```
1 >>> name = person.name
```

クエリを (`==`, `!=`, `<`, `>`, `<=`, `>=`, `like`, `belongs` のような演算子を用いて) 構築し、そのクエリを次のように変数 `q` に格納することもできます:

```
1 >>> q = name == 'Alex'
```

`db` をクエリとともに呼び出すと、レコードセットを定義していることになります。次のように書いて、それを変数 `s` に格納することができます：

```
1 >>> s = db(q)
```

ここまでデータベースクエリが実行されていないことに注意してください。DAL+クエリは、単純にクエリにマッチする `db` 内のレコードセットを定義するだけです。`web2py` はクエリからどのテーブル（もしくは複数のテーブル）が該当しているかを決めるので、テーブルを実際に指定する必要はありません。

### 6.17 select

Sets に対して、`select` コマンドを用いてレコードを取得することができます：

```
1 >>> rows = s.select()
```

これは、`Row` オブジェクトを要素とする `gluon.sql.Rows` クラスの反復可能なオブジェクトを返します。`gluon.sql.Row` オブジェクトは辞書のように振舞いますが、`gluon.storage.Storage` と同様、その要素は属性に関連付けられています。前者は、その値が読み取り専用であるということで後者とは異なります。

`Rows` オブジェクトは、`select` の結果に対しループを回して、各行の選択したフィールドをプリントできるようにすることができます：

```
1 >>> for row in rows:
2     print row.id, row.name
3 Alex
```

上の一連の手順は、次のように 1 つの文で行うことができます：

```
1 >>> for row in db(db.person.name=='Alex').select():
2     print row.name
3 Alex
```

`select` コマンドは引数をとることができます。全ての無名引数は、取得したいフィールド名として解釈されます。例えば、”`id`” と”`name`” フィールドを明示的に取得することができます：

```
1 >>> for row in db().select(db.person.id, db.person.name):
2     print row.name
3 Alex
4 Bob
5 Carl
```

テーブルの ALL 属性によって、全てのフィールドを指定することができます：

```

1 >>> for row in db().select(db.person.ALL):
2     print row.name
3 Alex
4 Bob
5 Carl

```

db にはクエリ文字列が何も渡されていないことに注目してください。web2py は、person テーブルの全てのフィールドが追加情報なしに要求された場合、person テーブルの全てのレコードが要求されていることを理解しています。

同等の代替構文は以下の通りです：

```

1 >>> for row in db(db.person.id > 0).select():
2     print row.name
3 Alex
4 Bob
5 Carl

```

そして、`person(id > 0)` テーブルの全てのレコードが追加情報なしに要求された場合、web2py は person テーブルの全てのフィールドが要求されていることを理解しています。

ひとつの row に対して

```
1 row = rows[0]
```

値をいくつかの同等な式で取得できます：

```

1 >>> row.name
2 Alex
3 >>> row['name']
4 Alex
5 >>> row('person.name')
6 Alex

```

後者の構文は、カラムではなく式で選択したい場合に特に便利です。これについては後述します。

また以下のように

```
1 rows.compact = False
```

表記法を無効にしたり

```
1 row[i].name
```

有効にして長い表記法にしたり：

```
1 row[i].person.name
```

できますが、実際にはこのようにする必要はありません。

#### 6.17.1 ショートカット

DAL はコードを簡素化するさまざまなショートカットをサポートしています。具体例を示します：

```
1 myrecord = db.mytable[id]
```

これは、`id` を持つレコードを存在すれば返します。`id` が存在しない場合は、`None` を返します。上記の文は以下と等価です：

```
1 myrecord = db(db.mytable.id==id).select().first()
```

次のようにして、`id` でレコードを削除することができます：

```
1 del db.mytable[id]
```

これは以下と等価です

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

これは、`id` を持つレコードが存在すれば削除します。

次のようにして、レコードを挿入することができます：

```
1 db.mytable[0] = dict(myfield='somevalue')
```

これは以下と等価です

```
1 db.mytable.insert(myfield='somevalue')
```

これは、右側の辞書で指定したフィールド値を持つ新規レコードを作成します。

次のようにしてレコードを更新することができます：

```
1 db.mytable[id] = dict(myfield='somevalue')
```

これは以下と等価です

```
1 db(db.mytable.id==id).update(myfield='somevalue')
```

これは、右側の辞書で指定したフィールド値で既存のレコードを更新します。

### 6.17.2 Fetching a Row

もう一つの便利な構文は次のとあります:

```
1 record = db.mytable(id)
2 record = db.mytable(db.mytable.id==id)
3 record = db.mytable(id,myfield='somevalue')
```

上記の構文は、明らかに `db.mytable[id]` と似ていますが、より柔軟性が高く安全です。まず初めに、これは `id` が `int`(または `str(id)` が `int`) であることを確認し、そうでない場合は `None` を返します(例外を発生させることはできません)。レコードが満たす必要のある複数の条件を指定することも可能です。条件に合わない場合は、同様に `None` を返します。

### 6.17.3 再帰的な selects

前述の `person` テーブルと、”`person`”を参照する新規の”`dog`”テーブルを考えます:

```
>>> db.define_table('dog', Field('name'), Field('owner', db.person))
```

このテーブルの単純な `select` は次のようにになります:

```
1 >>> dogs = db(db.dog).select()
```

これは以下と等価です

```
1 >>> dogs = db(db.dog._id>0).select()
```

`._id` はテーブルのプライマリキーを参照しています。通常、`db.dog._id` は `db.dog.id` と同じで、この本でもこれを前提にしています。`dogs` の各 Row に対して、選択したテーブル(`dog`)のフィールドだけでなく、リンクしたテーブルのフィールドを(再帰的に)取り出すことが可能です:

```
1 >>> for dog in dogs: print dog.name, dog.owner.name
```

ここでは、`dog.owner.name` は、`dogs` の個々の `dog` に対して、1回のデータベースの `select` を要求するので非効率です。利用可能な時は、再帰的な `select` の代わりに `join` を用いることを推奨します。とはいえ、これは個々のレコードにアクセスするときに便利で実用的です。`person` によって参照された `dogs` を、逆方向で `select` することも可能です:

```

1 person = db.person(id)
2 for dog in person.dog.select(orderby=db.dog.name):
3     print person.name, 'owns', dog.name

```

この最後の式において、`person.dog` は次のものに対するショートカットになります：

```
1 db(db.dog.owner==person.id)
```

つまり、現在の `person` によって参照される（複数の）`dog` の Set になります。この構文では参照しているテーブルが、参照されたテーブルへ複数の参照を持つ場合は破綻します。そのような時は、より明確に完全なクエリを使用する必要があります。

#### 6.17.4 ビューにおける `Rows` のシリアルライズ

クエリーを含む次の関数がある場合、

```

1 def index():
2     return dict(rows = db(query).select())

```

`select` の結果は、次の構文を使用してビューに表示することができます：

```

1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=rows}}

```

これは以下と等価です：

```

1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=SQLTABLE(rows)}}

```

`SQLTABLE` は `rows` を、HTML のテーブルに変換します。テーブルは、ヘッダにカラム名を、一行毎に各レコードを含んでいます。行は”even” と”odd” クラスで交互にマークされます。内部では、`Rows` は最初に `SQLTABLE` オブジェクト（テーブルとは混同しないでください）へと変換され、シリアルライズされます。データベースから抽出した値はまた、そのフィールドに関連付けられたバリデータによってフォマットされ、エスケープされます。（注：ビュー内で `db` をこのような方法で使用することは、普通は良い MVC のプラクティスとして考えられていません）

また、SQLTABLE を明示的に呼び出すことも可能で、便利な時があります。

SQLTABLE のコンストラクタは次のようなオプション引数をとります：

- `linkto` URL、または、参照フィールドをリンクするために使用される関数 (デフォルトは `None`)
- `upload` URL、または、アップロードしたファイルのダウンロードを許可するダウンロード関数 (デフォルトは `None`)
- `headers` ヘッダーに使用するフィールド名とそのラベルをマッピングする辞書 (デフォルトは `{}`)。一種の命令を指定することもできます。現在は、`headers='fieldname:capitalize'` をサポートしています。
- `truncate` テーブルの長い値を切り捨てる文字数 (デフォルトは 16)
- `columns` カラムとして表示するフィールド名のリスト (`tablename.fieldname` のフォーマット)。リストされていないものは表示されません (デフォルトは全て)。
- `**attributes` 最外部の TABLE オブジェクトに渡される汎用的なヘルパ属性です。

以下が具体例です：

```

1 {{extend 'layout.html'}}
2 <h1>Records</h1>
3 {{=SQLTABLE(rows,
4     headers='fieldname:capitalize',
5     truncate=100,
6     upload=URL('download'))}}
7 }}
```

`SQLTABLE` は便利ですが、より多くの機能を必要とする場合があります。`SQLFORM.grid` は `SQLTABLE` の拡張で、検索、ページング、詳細レコードの表示、作成、編集、削除機能を持ったテーブルを作成します。`SQLFORM.smartgrid` は上記の全ての機能に加えて、参照レコードへアクセスするためのボタンを作成します。

以下は `SQLFORM.grid` の使用例です:

```

1 def index():
2     return dict(grid=SQLFORM.grid(query))
```

そして対応するビューは:

```

1 {{extend 'layout.html'}}
2 {{=grid}}

```

`SQLFORM.grid` と `SQLFORM.smartgrid` は制約を受けますが、それ以上に強力であるため `SQLTABLE` より優れているといえます。第 8 章で詳細を説明します。

#### 6.17.5 orderby, groupby, limitby, distinct

`select` コマンドは 5 つのオプション引数をとります: `orderby`、`groupby`、`limitby`、`left`、`cache` です。ここでは、最初の 3 つについて説明します。

次のように、`name` でソートされたレコードを取り出すことができます：

```

1 >>> for row in db().select(
2         db.person.ALL, orderby=db.person.name):
3     print row.name
4 Alex
5 Bob
6 Carl

```

`name` の逆順でソートされたレコードを取り出すことができます（チルダに注意してください）：

```

1 >>> for row in db().select(
2         db.person.ALL, orderby=~db.person.name):
3     print row.name
4 Carl
5 Bob
6 Alex

```

ランダムな順番で取り出したレコードを得ることができます：

```

1 >>> for row in db().select(
2         db.person.ALL, orderby='<random>'):
3     print row.name
4 Carl
5 Alex
6 Bob

```

`orderby='<random>'` の使用は *Google NoSQL* 上ではサポートされません。しかし、同じ状況や同様にビルトインが不十分な他の多くの場合、次のようにインポートを使うことができます：

```

1 import random
2 rows=db(...).select().sort(lambda row: random.random())

```

複数のフィールドに対して、レコードをソートすることができます。これはフィールドを”|”によって連結することで可能です：

```

1 >>> for row in db().select(
2         db.person.ALL, orderby=db.person.name|db.person.id):
3     print row.name
4 Carl
5 Bob
6 Alex

```

orderbyと一緒に groupby を用いて、指定したフィールドの同じ値を持つレコードをグループ化することができます（これはバックエンドに依存します、Google NoSQL では利用できません）：

```

1 >>> for row in db().select(
2         db.person.ALL,
3         orderby=db.person.name, groupby=db.person.name):
4     print row.name
5 Alex
6 Bob
7 Carl

```

`distinct=True` の引数を用いて、重複のないレコードだけを選択することができます。フィールドを全て指定して、グループ化するのと同じ効果を持ちます。ただしこの場合、ソートは必要ありません。`distinct` を用いるとき、全フィールドを選択をしないことは重要です。特に、”id” フィールドを選択しないでください。選択した場合、全レコードが常に重複なしの状態になってしまいます。

以下に例を示します：

```

1 >>> for row in db().select(db.person.name, distinct=True):
2     print row.name
3 Alex
4 Bob
5 Carl

```

`distinct` は式にすることもできます。例：

```

1 >>> for row in db().select(db.person.name, distinct=db.person.name):
2     print row.name
3 Alex
4 Bob
5 Carl

```

`limitby` を使用すると、レコードの一部を選択することができます（以下の例では、0 から始まる最初の 2 つが選択されます）：

```

1 >>> for row in db().select(db.person.ALL, limitby=(0, 2)):
2     print row.name
3 Alex
4 Bob

```

### 6.17.6 論理演算子

クエリは、AND の二項演算子”&”を使用し、組み合わせることができます：

```

1 >>> rows = db((db.person.name=='Alex') & (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 4 Alex

```

OR の二項演算子”|”も同様です：

```

1 >>> rows = db((db.person.name=='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 1 Alex

```

”!=” の二項演算子によってクエリ（またはサブクエリ）を否定できます：

```

1 >>> rows = db((db.person.name!='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 2 Bob
4 3 Carl

```

または、”” 単項演算子による明示的な否定によっても可能です：

```

1 >>> rows = db((~db.person.name=='Alex') | (db.person.id>3)).select()
2 >>> for row in rows: print row.id, row.name
3 2 Bob
4 3 Carl

```

Python における”and” と”or” 演算子のオーバーロード制約により、これらはクエリ生成には使用できません。代わりに二項演算子を使用する必要があります。in-place 論理演算子（累積代入文）を使用し、クエリを構築することもできます。

```

1 >>> query = db.person.name!='Alex'
2 >>> query &= db.person.id>3
3 >>> query |= db.person.name=='John'

```

### 6.17.7 count, isempty, delete, update

セット内のレコードをカウントすることができます：

```
1 >>> print db(db.person.id > 0).count()
2 3
```

`count` はオプションで、デフォルトが `False` の `distinct` 引数を指定することができます。これは `select` で同じ引数を指定した場合と、非常によく似た動作をします。

テーブルのレコードが空かどうかをチェックしたい場合があります。この場合、カウントするよりも `isempty` メソッドを使うほうがより有効です。

```
1 >>> print db(db.person.id > 0).isempty()
2 False
```

または、等価の式で：

```
1 >>> print db(db.person).isempty()
2 False
```

セット内のレコードを削除することができます：

```
1 >>> db(db.person.id > 3).delete()
```

セット内の全てのレコードを更新することができます。更新が必要なフィールドに対応する、名前付き引数を渡します。

```
1 >>> db(db.person.id > 3).update(name='Ken')
```

### 6.17.8 式

更新文で割り当てる値は、式でも可能です。例えば、次のようなモデルで考えてみます

```
1 >>> db.define_table('person',
2     Field('name'),
3     Field('visits', 'integer', default=0))
4 >>> db(db.person.name == 'Massimo').update(
5         visits = db.person.visits + 1)
```

クエリで使用される値もまた、式にすることができます

```

1 >>> db.define_table('person',
2     Field('name'),
3     Field('visits', 'integer', default=0),
4     Field('clicks', 'integer', default=0))
5 >>> db(db.person.visits == db.person.clicks + 1).delete()

```

### 6.17.9 update\_record

web2py では `update_record` を用いて、すでにメモリ上にある単一のレコードを更新することも可能です。

```

1 >>> row = db(db.person.id==2).select().first()
2 >>> row.update_record(name='Curt')

```

`update_record` を次のものと混同しないでください

```

1 >>> row.update(name='Curt')

```

その理由は、単一の `row` に対して、`update` メソッドは `row` オブジェクトを更新しますが、`update_record` のようにデータベースのレコードを更新することはしないからです。`row` の属性を変更（一度に一つずつ）し、それを保存するため引数指定なしで `update_record()` を実行することもできます。

```

1 >>> row = db(db.person.id > 2).select().first()
2 >>> row.name = 'Curt'
3 >>> row.update_record() # saves above change

```

### 6.17.10 first と last

レコードを保持した `Rows` オブジェクトが与えられた時、次のようにすることができます：

```

1 >>> rows = db(query).select()
2 >>> first_row = rows.first()
3 >>> last_row = rows.last()

```

これは以下のものに相当します。

```

1 >>> first_row = rows[0] if len(rows)>0 else None
2 >>> last_row = rows[-1] if len(rows)>0 else None

```

### 6.17.11 as\_dict と as\_list

Row オブジェクトは、`as_dict()` メソッドを用いて標準の辞書にシリアル化することができます。Rows オブジェクトは、`as_list()` メソッドを用いて辞書のリストにシリアル化することができます。例をいくつか示します：

```
1 >>> rows = db(query).select()
2 >>> rows_list = rows.as_list()
3 >>> first_row_dict = rows.first().as_dict()
```

これらのメソッドは、Rows を汎用的なビューに渡したり、Rows をセッションに格納したりするのに便利です (Rows オブジェクト自体は、開いている DB 接続への参照があるのでシリアル化できません) :

```
1 >>> rows = db(query).select()
2 >>> session.rows = rows # not allowed!
3 >>> session.rows = rows.as_list() # allowed!
```

### 6.17.12 find, exclude, sort

2 つの select を実行する必要があり、また前回の select のサブセットを保持するしている状況がよくあります。この場合、再度データベースにアクセスするのは冗長なことです。`find`、`exclude`、`sort` オブジェクトは、Rows オブジェクトを操作し、データベースアクセスなしで別の Rows オブジェクト生成を可能にします。具体的に次のようになります:

- `find` は、条件でフィルタされた新規の Rows セットを返します。元のものはそのままです。
- `exclude` は、条件でフィルタされた新規の Rows セットを返します。それらは元のものから取り除かれます。
- `sort` は、条件でソートされた新規の Rows セットを返します。元のものはそのままです。

これら全てのメソッドは、単一の引数として、各々の row に作用する関数をとります。

これはその使用例です：

```
1 >>> db.define_table('person', Field('name'))
2 >>> db.person.insert(name='John')
```

```

3 >>> db.person.insert(name='Max')
4 >>> db.person.insert(name='Alex')
5 >>> rows = db(db.person).select()
6 >>> for row in rows.find(lambda row: row.name[0]== 'M'):
7     print row.name
8 Max
9 >>> print len(rows)
10 3
11 >>> for row in rows.exclude(lambda row: row.name[0]== 'M'):
12     print row.name
13 Max
14 >>> print len(rows)
15 2
16 >>> for row in rows.sort(lambda row: row.name):
17     print row.name
18 Alex
19 John

```

これらは組み合わせることができます：

```

1 >>> rows = db(db.person).select()
2 >>> rows = rows.find(
3     lambda row: 'x' in row.name).sort(
4         lambda row: row.name)
5 >>> for row in rows:
6     print row.name
7 Alex
8 Max

```

## 6.18 その他のメソッド

### 6.18.1 update\_or\_insert

同じ値が存在しない場合だけ、挿入したい時があります。これは以下のように実現できます

```

1 db.define_table('person',Field('name'),Field('birthplace'))
2 db.person.update_or_insert(name='John',birthplace='Chicago')

```

Chicago で生まれた John が他に存在しない場合だけ挿入されます。

レコードの存在チェックにどのキーを使用するかを指定することができます。例：

```

1 db.person.update_or_insert(db.person.name== 'John',
2     name='John',birthplace='Chicago')

```

John が存在する場合は birthplace が更新され、それ以外は挿入されます。

### 6.18.2 validate\_and\_insert, validate\_and\_update

この関数は The function

```
1 ret = db.mytable.validate_and_insert(field='value')
```

以下とほぼ同じ様に動作します。works very much like

```
1 id = db.mytable.insert(field='value')
```

違いは、挿入の前にバリデータが実行され、通らなかった場合は挿入されないと  
いう点です。バリデータを通らなかった場合は `ret.error` にエラー内容が存在  
します。通った場合は、新規レコードの `id` は `ret.id` に存在します。通常、バ  
リデータはフォームの処理ロジックで実装されるので、この関数を使用する機会  
はほとんどないはずです。

同様に

```
1 ret = db(query).validate_and_update(field='value')
```

以下とほぼ同じ様に動作します。

```
1 num = db(query).update(field='value')
```

違いは、更新の前にバリデータが実行される点です。单一のテーブルでだけ動作  
する点に注意してください。更新されたレコード数は `res.updated` に、エラー  
は `ret.errors` に存在します。

### 6.18.3 smart\_query (実験的)

以下のような自然言語を使ったクエリを解析したい場合があります

```
1 name contain m and age greater than 18
```

DAL はこのようなタイプのクエリを解析するメソッドを提供します：

```
1 search = 'name contain m and age greater than 18'
2 rows = db.smart_query([db.person], search).select()
```

最初の引数はテーブルのリストか検索可能なフィールドでないといけません。検索文字列が無効な場合は `RuntimeError` が発生します。この機能は RESTful なインターフェース（10章を参照）を構築する場合や、`SQLFORM.grid` と `SQLFORM.smartgrid` の内部で使用されています。`smartquery` 検索文字列では、フィールドはフィールド名だけかテーブル名. フィールド名で指定できます。空白を含む文字はダブルクォーテーションで区切られます。

## 6.19 計算されたフィールド

DAL のフィールドは `compute` フィールドを持つことがあります。これは、Row オブジェクトを引数にとり、そのフィールドに対する値を返す関数（またはラムダ）である必要があります。新規のレコードが挿入や更新などで変更される時、そのフィールドの値が用意されていない場合、web2py は `compute` 関数を用いて他のフィールドの値から計算しようとします。以下がその例です。

```

1 >>> db.define_table('item',
2     Field('unit_price', 'double'),
3     Field('quantity', 'integer'),
4     Field('total_price',
5         compute=lambda r: r['unit_price']*r['quantity']))
6 >>> r = db.item.insert(unit_price=1.99, quantity=5)
7 >>> print r.total_price
8 9.95

```

なお、計算された値は db に格納され、後述する仮想フィールドの場合のように再取得時に計算されることはありません。計算されたフィールドの 2 つの典型的な活用方法は：

- wiki アプリケーションにおいて、HTML に加工された wiki の入力テキストを、リクエスト毎の加工を避けるために格納する
- 検索用に、フィールドの正規化した値を計算し、検索時に使用する

## 6.20 仮想フィールド

### 6.20.1 古い形式の仮想フィールド

仮想フィールドもまた、(前節のように) 計算されたフィールドですが、それらは異なります。なぜなら、データベースには格納されず、また、データベースからレコードが取り出されるたびに計算されるという点で仮想であるからです。追加の格納先なしに単純にユーザーコードを用いることができますが、それを用いて検索することはできません。

1つ以上の仮想フィールドを定義するためには、コンテナクラスを定義し、インスタンス化し、テーブルまたは選択に対してリンクさせる必要があります。例えば、次のようなテーブルを考えてください：

```
1 >>> db.define_table('item',
2                     Field('unit_price', 'double'),
3                     Field('quantity', 'integer'),
```

この時、total\_price という仮想フィールドを次のように定義できます

```
1 >>> class MyVirtualFields(object):
2         def total_price(self):
3             return self.item.unit_price * self.item.quantity
4
>>> db.item.virtualfields.append(MyVirtualFields())
```

单一の引数 (self) をとるクラスの各メソッドが、新規の仮想フィールドになることに注意してください。フィールドの値は self.item.unit\_price のように完全パスで参照されます。テーブルは、このクラスのインスタンスをテーブルの virtualfields 属性に追加することによって、この仮想フィールドにリンクされます。

仮想フィールドも同様に、次のように再帰的なフィールドにアクセスできます

```
1 >>> db.define_table('item',
2                     Field('unit_price', 'double'))
3 >>> db.define_table('order_item',
4                     Field('item', db.item),
5                     Field('quantity', 'integer'))
6 >>> class MyVirtualFields(object):
7         def total_price(self):
8             return self.order_item.item.unit_price \
9                   * self.order_item.quantity
10 >>> db.order_item.virtualfields.append(MyVirtualFields())
```

再帰的なフィールドは self.order\_item.item.unit\_price にアクセスしていますが、ここで、self はループで回されているレコードであることに注意して

ください。

これらは、結合 (JOIN) の結果に対しても作用することができます

```

1 >>> db.define_table('item',
2                     Field('unit_price', 'double'))
3 >>> db.define_table('order_item',
4                     Field('item', db.item),
5                     Field('quantity', 'integer'))
6 >>> rows = db(db.order_item.item==db.item.id).select()
7 >>> class MyVirtualFields(object):
8     def total_price(self):
9         return self.item.unit_price \
10            * self.order_item.quantity
11 >>> rows.setvirtualfields(order_item=MyVirtualFields())
12 >>> for row in rows: print row.order_item.total_price

```

この場合、どのように構文が異なっているかに注意してください。仮想フィールドは、join 選択に属している `self.item.unit_price` と `self.order_item.quantity` の両方にアクセスしています。仮想フィールドは `rows` オブジェクトの `setvirtualfields` メソッドを用いてテーブルの `rows` に付け加えられます。このメソッドは任意の数の名前付き引数をとります。そして次のように、複数のクラスで定義された複数の仮想フィールドを設定し、それらを複数のテーブルに付け加えることができます：

```

1 >>> class MyVirtualFields1(object):
2     def discounted_unit_price(self):
3         return self.item.unit_price*0.90
4 >>> class MyVirtualFields2(object):
5     def total_price(self):
6         return self.item.unit_price \
7            * self.order_item.quantity
8     def discounted_total_price(self):
9         return self.item.discounted_unit_price \
10            * self.order_item.quantity
11 >>> rows.setvirtualfields(
12     item=MyVirtualFields1(),
13     order_item=MyVirtualFields2())
14 >>> for row in rows:
15     print row.order_item.discounted_total_price

```

仮想フィールドは遅延 (lazy) することができます。そのためにやることは、関数を返すようにし、その関数を呼び出すことによってアクセスすることです：

```

1 >>> db.define_table('item',
2                     Field('unit_price', 'double'),

```

```

3     Field('quantity','integer'),
4 >>> class MyVirtualFields(object):
5     def lazy_total_price(self):
6         def lazy(self=self):
7             return self.item.unit_price \
8                 * self.item.quantity
9         return lazy
10    >>> db.item.virtualfields.append(MyVirtualFields())
11    >>> for item in db(db.item).select():
12        print item.lazy_total_price()

```

または、ラムダ関数を用いてより短くします：

```

1 >>> class MyVirtualFields(object):
2     def lazy_total_price(self):
3         return lambda self=self: self.item.unit_price \
4             * self.item.quantity

```

### 6.20.2 新しい形式の仮想フィールド（実験的）

web2py は新しくて簡単な仮想フィールドと遅延 (lazy) フィールドの定義方法を用意しています。この節の見出しが（実験的）となっているのは、API がここで述べられているものから一部変更されている可能性があるからです。

前節と同じサンプルで考えてみます。具体的には以下のモデルになります：

```

1 >>> db.define_table('item',
2                     Field('unit_price','double'),
3                     Field('quantity','integer'),

```

この時、total\_price という仮想フィールドを次のように定義できます

```

1 >>> db.item.total_price = Field.Virtual(lambda row: row.unit_price*row.
2                                         quantity)

```

つまり、Field.Virtual として total\_price という新しいフィールドを定義しています。コンストラクタ引数は row を受け取る関数だけで、計算されたフィールドを返します。

上記で定義された仮想フィールドは、レコードが選択された時点で全てのレコードの値が自動で計算されます。

```

1 >>> for row in db(db.item).select(): print row.total_price

```

呼び出された時に随時計算される遅延 (lazy) フィールドを定義することもできます。例：

```
1 >>> db.item.total_price = Field.Lazy(lambda row, discount=0.0: \
2     row.unit_price*row.quantity*(1.0-discount/100))
```

この場合 `row.total_price` は値ではなく関数です。この関数は暗黙の型宣言 (`row` オブジェクトの `self` のように) である `row` を除いて、Lazy コンストラクタに渡されます。

上記の例にある遅延 (lazy) フィールドは、それぞれの `item` の合計金額を計算します。

```
1 >>> for row in db(db.item).select(): print row.total_price()
```

そして、discount 率 (15%) をオプションで渡します。

```
1 >>> for row in db(db.item).select(): print row.total_price(15)
```

仮想フィールドは他のフィールドと同様の属性(`default`, `readable`, `requires`, `etc`)を持たず、`db.table.fields` のリストには表示されず、テーブル (`TABLE`) やグリッド (`SQLFORM.grid`, `SQLFORM.smartgrid`) ではデフォルトで表示されない点に気をつけてください。

## 6.21 1対多のリレーション

web2py の DAL を用いて 1 対多のリレーションをどのように実装するかを説明するために、”person” テーブルを参照するもう 1 つの”dog” テーブルを定義します。”person” もここで再定義します：

```
1 >>> db.define_table('person',
2                     Field('name'),
3                     format='%(name)s')
4 >>> db.define_table('dog',
5                     Field('name'),
6                     Field('owner', db.person),
7                     format='%(name)s')
```

”dog” テーブルは、犬の名前 (`name`) と犬の飼い主 (`owner`) という 2 つのフィールドを持ちます。フィールドの型が他のテーブルの時、そのフィールドが他のテーブルをその `id` によって参照することを意図しています。実際、実在の型の値を出力及び取得することができます：

```

1 >>> print db.dog.owner.type
2 reference person

```

ここで、Alex の飼っている 2 匹と、Bob が飼っている 1 匹の計 3 匹の犬を挿入します：

```

1 >>> db.dog.insert(name='Skipper', owner=1)
2 1
3 >>> db.dog.insert(name='Snoopy', owner=1)
4 2
5 >>> db.dog.insert(name='Puppy', owner=2)
6 3

```

ほかのテーブルと同じように、テーブルを select することができます：

```

1 >>> for row in db(db.dog.owner==1).select():
2     print row.name
3 Skipper
4 Snoopy

```

犬は飼い主の参照を持っているため、飼い主は複数の犬を持つことができます。したがって、person テーブルのレコードはこの時、dog という新規の属性を取得します。これにより、全ての飼い主に対してループを回して、それらの犬を取得することができるようになります：

```

1 >>> for person in db().select(db.person.ALL):
2     print person.name
3     for dog in person.dog.select():
4         print '    ', dog.name
5 Alex
6     Skipper
7     Snoopy
8 Bob
9     Puppy
10 Carl

```

### 6.21.1 内部結合 (*Inner Joins*)

同様の結果を得るために別の方法は、join、具体的には、INNER JOIN を用いることです。web2py は、次の例のようにクエリが 2 つ以上のテーブルをリンクする時に、join を自動で透過的に実行します。

```

1 >>> rows = db(db.person.id==db.dog.owner).select()
2 >>> for row in rows:

```

```

3     print row.person.name, 'has', row.dog.name
4 Alex has Skipper
5 Alex has Snoopy
6 Bob has Puppy

```

web2py が join を行って、rows が、一緒にリンクされた各テーブル由来の 2 つのレコードを含んでいることを見てください。2 つのレコードは競合する名前のフィールドを持つ可能性があるので、row からのフィールド値を取り出すときに、テーブルを指定する必要があります。つまり、次のことをする前は：

```
1 row.name
```

これが飼い主の名前なのか、犬の名前なのかは明らかでした。join の結果においては、次のようにより明示的なものにする必要があります：

```
1 row.person.name
```

or:

```
1 row.dog.name
```

INNER JOIN には別の構文もあります：

```

1 >>> rows = db(db.person).select(join=db.dog.on(db.person.id==db.dog.
    owner))
2 >>> for row in rows:
3     print row.person.name, 'has', row.dog.name
4 Alex has Skipper
5 Alex has Snoopy
6 Bob has Puppy

```

出力結果は同じですが、生成される SQL は異なる可能性があります。後者の構文は同じテーブルが 2 回結合されて別名が使用された場合に、それぞれのテーブルを明確に指定できます。

```

1 >>> db.define_table('dog',
2     Field('name'),
3     Field('owner1',db.person),
4     Field('owner2',db.person))
5 >>> rows = db(db.person).select(
6     join=[db.person.with_alias('owner1').on(db.person.id==db.dog.owner1
7         ),
8           db.person.with_alias('owner2').on(db.person.id==db.dog.owner2
9             )])

```

join の値には、`db.table.on(...)` のリストを使用できます。

### 6.21.2 左外部結合 (Left Outer Join)

Carl は犬を飼っていないので、上記の表には現れませんでした。飼い主 (犬を飼っている、いないに関わらず) と犬 (もし飼われていれば) を選択しようとした場合、LEFT OUTER JOIN を実行する必要があります。これは、select コマンドの”left”引数を用いて行うことができます。以下がその例です：

```

1 >>> rows=db().select(
2         db.person.ALL, db.dog.ALL,
3         left=db.dog.on(db.person.id==db.dog.owner))
4 >>> for row in rows:
5         print row.person.name, 'has', row.dog.name
6 Alex has Skipper
7 Alex has Snoopy
8 Bob has Puppy
9 Carl has None

```

ここで：

```
1 left = db.dog.on(...)
```

これは left join クエリを行います。db.dog.on の引数は、join に必要な条件になります (上述の inner join で使用したものと同じです)。left join の場合、どのフィールドを選択するかは明示的にする必要があります。

複数の left join をする場合は、db.mytable.on(...) のリストかタプルを left 属性に渡すことで組み合わせることができます。

### 6.21.3 グループ化とカウント

結合 (join) を行うとき、特定の条件に従って行をグループ化し、カウントしたい場合があります。例えば、各飼い主が飼っている犬の数をカウントする場合です。web2py ではこれも同様に行うことができます。初めに、カウント演算子が必要になります。第 2 に、person テーブルと dog テーブルをその飼い主 (owner) によって join します。第 3 に、全ての行 (person + dog) を選択し、person 毎にそれらをグループ化して、グループ化している最中にカウントします：

```

1 >>> count = db.person.id.count()
2 >>> for row in db(db.person.id==db.dog.owner).select(
3             db.person.name, count, groupby=db.person.name):
4             print row.person.name, row[count]
5 Alex 2

```

```
6 Bob 1
```

(組み込みの) カウント演算子がフィールドのように用いられていることに注意してください。ここでの唯一の問題は、どのように情報を取り出すかにあります。各行は明らかに 1 人の person とカウントを含んでいます。しかしカウントは、person のフィールドではなく、テーブルでもありません。これ（カウントオブジェクト）はどこに行くのでしょうか？。クエリ式自体と同じキーを持つ、レコードを表現するストレージオブジェクトになります。

### 6.22 Many to many

前述の例では、1 匹の犬は 1 人の飼い主 (owner) を持つけれど、1 人の飼い主は多くの犬を飼えるようにしました。Alex と Curt によって飼われている Skipper はどうなるのでしょうか？ これには多対多のリレーションが必要です。そしてこれは、所有 (ownership) 関係で 1 人の飼い主と 1 匹の犬をリンクする中間テーブルを介して実現されます。

どのようにそれを行うかを以下に示します：

```
1 >>> db.define_table('person',
2                         Field('name'))
3 >>> db.define_table('dog',
4                         Field('name'))
5 >>> db.define_table('ownership',
6                         Field('person', db.person),
7                         Field('dog', db.dog))
```

これまでの所有 (ownership) 関係は次のように書き換えることができます：

```
1 >>> db.ownership.insert(person=1, dog=1) # Alex owns Skipper
2 >>> db.ownership.insert(person=1, dog=2) # Alex owns Snoopy
3 >>> db.ownership.insert(person=2, dog=3) # Bob owns Puppy
```

今度は、Curt が Skipper を一緒に飼っているという、新しいリレーションを加えることができます：

```
1 >>> db.ownership.insert(person=3, dog=1) # Curt owns Skipper too
```

3 方向のテーブル間のリレーションを持っているので、操作を実行する上で、次のような新規の Set を定義することは便利です：

```

1 >>> persons_and_dogs = db(
2         (db.person.id==db.ownership.person) \
3         & (db.dog.id==db.ownership.dog))

```

これで新規の Set から、全ての飼い主と彼らの犬を簡単に選択できます：

```

1 >>> for row in persons_and_dogs.select():
2     print row.person.name, row.dog.name
3 Alex Skipper
4 Alex Snoopy
5 Bob Puppy
6 Curt Skipper

```

同様に、Alex が飼っている全ての犬を検索することもできます：

```

1 >>> for row in persons_and_dogs(db.person.name=='Alex').select():
2     print row.dog.name
3 Skipper
4 Snoopy

```

そして、Skipper の飼い主も検索することができます：

```

1 >>> for row in persons_and_dogs(db.dog.name=='Skipper').select():
2     print row.person.name
3 Alex
4 Curt

```

多対多の簡易な代替案はタグ付けです。タグ付けは `IS_IN_DB` のコンテキストで後述します。タグ付けは、Google App Engine NoSQL のような JOIN をサポートしていないデータベース・バックエンドでも機能します。

### 6.23 多対多、`list:<type>`、`contains`

web2py は、以下の特別なフィールド型を用意しています：

```

1 list:string
2 list:integer
3 list:reference <table>

```

これらはそれぞれ、文字列、整数、参照のリストを収容します。

Google App Engine NoSQL では、`list:string` は `StringListProperty` にマッピングされ、他の 2 つは、`ListProperty(int)` にマッピングされます。リレーションナル・データベースでは、|によって区切られた項目のリストを持つテ

キストフィールドにマッピングされます。例えば、`[1, 2, 3]` は `|1|2|3|` にマッピングされます。

文字列のリストでは、項目内の任意の `|` が `||` に置換されるように項目はエスケープされます。いずれにせよ、これは内部表現でありユーザーに対しては透過的です。

次の例のように `list:string` を用いることができます：

```

1 >>> db.define_table('product',
2     Field('name'),
3     Field('colors', 'list:string'))
4 >>> db.product.colors.requires=IS_IN_SET(('red', 'blue', 'green'))
5 >>> db.product.insert(name='Toy Car', colors=['red', 'green'])
6 >>> products = db(db.product.colors.contains('red')).select()
7 >>> for item in products:
8     print item.name, item.colors
9 Toy Car ['red', 'green']

```

`list:integer` も同様に機能します。ただし、項目は整数でなければなりません。例のごとく、この要求は、`insert` レベルではなく、フォームレベルで強制されます。

`list:<type>` フィールドにおいて、`contains(value)` 演算子は `value` が含まれているかをリストに対してチェックする、通常でないクエリにマッピングされます。`contains` 演算子は、標準の `string` と `text` フィールドでも機能し、`LIKE '%value%` にマッピングされます。

`list:reference` と `contains(value)` 演算子は、多対多リレーションの非正規化にとって特に有用です。以下がその例です。

```

1 >>> db.define_table('tag', Field('name'), format='%(name)s')
2 >>> db.define_table('product',
3     Field('name'),
4     Field('tags', 'list:reference tag'))
5 >>> a = db.tag.insert(name='red')
6 >>> b = db.tag.insert(name='green')
7 >>> c = db.tag.insert(name='blue')
8 >>> db.product.insert(name='Toy Car', tags=[a, b, c])
9 >>> products = db(db.product.tags.contains(b)).select()
10 >>> for item in products:
11     print item.name, item.tags
12 Toy Car [1, 2, 3]
13 >>> for item in products:
14     print item.name, db.product.tags.represent(item.tags)

```

```
15 Toy Car red, green, blue
```

`list:reference` の `tag` フィールドは、次のようなデフォルトの制約を取得することに注意してください

```
1 requires = IS_IN_DB(db, 'tag.id', db.tag._format, multiple=True)
```

これは、フォームにおいて `SELECT/OPTION` の複数ドロップボックスを生成します。

また、このフィールドはデフォルトで、フォーマットした参照のカンマ区切りリストのように、参照リストを表現する `represent` 属性を取得することにも注意してください。これは、フォームと `SQLTABLE` の読み込み時に利用されます。

`list:reference` はデフォルトのバリデータとデフォルトの表現を持つ一方、`list:integer` と `list:string` は持ちません。したがって、これら 2 つをフォームで利用する場合、`IS_IN_SET` か `IS_IN_DB` バリデータが必要になります。

## 6.24 他の演算子

web2py には、同等な SQL 演算子にアクセスするための API を提供する演算子があります。”log” という別のテーブルを定義してみます。そのテーブルでは、セキュリティ・イベントとその `event_time` と重大度 (`severity`) を格納するようにします。ここで重大度 (`severity`) は整数です。

```
1 >>> db.define_table('log', Field('event'),
2                               Field('event_time', 'datetime'),
3                               Field('severity', 'integer'))
```

前回と同様、イベントとして、”port scan” と ”xss injection” と ”unauthorized login” を数個挿入します。例として、同じ `event_time` を持つが重大度 (それぞれ 1,2,3) は異なるイベントをログとして記録します。

```
1 >>> import datetime
2 >>> now = datetime.datetime.now()
3 >>> print db.log.insert(
4             event='port scan', event_time=now, severity=1)
5 1
6 >>> print db.log.insert(
7             event='xss injection', event_time=now, severity=2)
```

```

8 2
9 >>> print db.log.insert(
10     event='unauthorized login', event_time=now, severity=3)
11 3

```

### 6.24.1 like, startswith, contains, upper, lower

フィールドは、文字列を照合するための like 演算子を持っています：

```

1 >>> for row in db(db.log.event.like('port%')).select():
2     print row.event
3 port scan

```

ここで、”port%” は”port” から始まる文字列を示しています。パーセント記号文字”%” は、”任意の文字列” を意味するワイルドカード文字です。web2py はまた、いくつかのショートカットを提供しています：

```

1 db.mytable.myfield.startswith('value')
2 db.mytable.myfield.contains('value')

```

これは、それぞれ以下に相当します

```

1 db.mytable.myfield.like('value%')
2 db.mytable.myfield.like('%value%')

```

contains は、前節で説明したように、list:<type> フィールドに対して特別な意味を持つことに注意してください。

contains メソッドは値をリストで渡すことや、ブーリアン型の変数 all をオプションで指定し全ての値を含むレコードだけを検索することもできます。

```
1 db.mytable.myfield.contains(['value1', 'value2'], all=True)
```

または、リストのどちらかの値を含む場合は次の通りです。

```
1 db.mytable.myfield.contains(['value1', 'value2'], all=False)
```

同様に、フィールドの値を大文字、または、小文字に変換するために upper と lower メソッドを使用することができます。さらに、like 演算子と組み合わせることができます。

```

1 >>> for row in db(db.log.event.upper().like('PORT%')).select():
2     print row.event
3 port scan

```

#### 6.24.2 year, month, day, hour, minutes, seconds

date と datetime フィールドは day、month、year メソッドを持ちます。datetime および time フィールドは、hour、minutes、seconds メソッドを持ちます。以下がその例です：

```

1 >>> for row in db(db.log.event_time.year() == 2009).select():
2     print row.event
3 port scan
4 xss injection
5 unauthorized login

```

#### 6.24.3 belongs

SQL の IN 演算子は、belongs メソッドを介して実現されます。このメソッドは、フィールドの値が指定したセット（タプルのリスト）に所属している時に true を返します。

```

1 >>> for row in db(db.log.severity.belongs((1, 2))).select():
2     print row.event
3 port scan
4 xss injection

```

DAL はまた、belongs 演算子の引数にネストした select を許しています。唯一の注意点は、ネストした select は select ではなく \_select でなければならず、フィールドは 1 つだけ、明示的にセットを定義するものを選択する必要があります。

```

1 >>> bad_days = db(db.log.severity == 3)._select(db.log.event_time)
2 >>> for row in db(db.log.event_time.belongs(bad_days)).select():
3     print row.event
4 port scan
5 xss injection
6 unauthorized login

```

#### 6.24.4 sum, min, max and len

前回は、カウント演算子をレコードのカウントに使用しました。同様にサム (sum) 演算子を、レコードのグループから特定のフィールドの値を足す (sum) ことに

使用することができます。カウントの場合と同様に、サムの結果は格納オブジェクトから取り出すことができます：

```
1 >>> sum = db.log.severity.sum()
2 >>> print db().select(sum).first()[sum]
3 6
```

同様に `min` と `max` で、選択されたレコードの最小値と最大値を取り出せます。

```
1 >>> max = db.log.severity.max()
2 >>> print db().select(max).first()[max]
3 3
```

`.len()` は文字、テキスト、またはブーリアン型のフィールドの長さを計算します。

式を組み合わせてより複雑な式を作ることができます。この例では、`log` テーブルの `severity` 文字フィールドの長さに 1 を加えた結果を合計しています。

```
1 >>> sum = (db.log.severity.len() + 1).sum()
2 >>> print db().select(sum).first()[sum]
```

## 6.24.5 サブストリング

サブストリングの値を参照した式を作成することができます。例えば、最初の 3 文字の名前が同じ犬をグループ化でき、各グループから 1 つだけ選択します。

```
1 db(db.dog).select(distinct = db.dog.name[:3])
```

## 6.24.6 `coalesce` と `coalesce_zero` によるデフォルト値

データベースから値を取得したいが、NULL の時はデフォルトの値が必要な場合があります。SQL ではこの目的のために `COALESCE` が提供されています。web2py では等価の `coalesce` メソッドを提供します。

```
1 >>> db.define_table('sysuser', Field('username'), Field('fullname'))
2 >>> db.sysuser.insert(username='max', fullname='Max Power')
3 >>> db.sysuser.insert(username='tim', fullname=None)
4 print db(db.sysuser).select(db.sysuser.fullname.coalesce(db.sysuser.
    username))
5 "COALESCE(sysuser.fullname, sysuser.username)"
6 Max Power
7 tim
```

数学的な式を計算したいが、本来ゼロであるべきフィールドに None がセットされている場合もあります。`coalesce_zero` はクエリで None のデフォルト値にゼロをセットする手助けをします。

```

1 >>> db.define_table('sysuser',Field('username'),Field('points'))
2 >>> db.sysuser.insert(username='max',points=10)
3 >>> db.sysuser.insert(username='tim',points=None)
4 >>> print db(db.sysuser).select(db.sysuser.points.coalesce_zero().sum()
5     )
5 "SUM(COALESCE(sysuser.points, 0)) "
6 10

```

## 6.25 生 SQL の生成

SQL は生成したいが実行したくないことがあります。web2py でこれを行うのは簡単です。なぜならデータベースの IO を実行する全てのコマンドは、単純に実行しようとした SQL を実行せずに返す、同等のコマンドを持つからです。これらのコマンドは、機能するものと同じ名前と構文を持ちますが、アンダースコアで始まります：

これは`_insert` です：

```

1 >>> print db.person._insert(name='Alex')
2 INSERT INTO person(name) VALUES ('Alex');

```

これは`_count` です

```

1 >>> print db(db.person.name=='Alex')._count()
2 SELECT count(*) FROM person WHERE person.name='Alex';

```

これは`_select` です

```

1 >>> print db(db.person.name=='Alex')._select()
2 SELECT person.id, person.name FROM person WHERE person.name='Alex';

```

これは`_delete` です

```

1 >>> print db(db.person.name=='Alex')._delete()
2 DELETE FROM person WHERE person.name='Alex';

```

最後に、これは`_update` です

```

1 >>> print db(db.person.name=='Alex')._update()
2 UPDATE person SET WHERE person.name='Alex';

```

さらに、`db._lastsql` を用いて、直近の *SQL* コードを返すことができます。これは、`executesql` を用いて手動で実行された *SQL* でも、*DAL* によって生成された *SQL* でも可能です。

## 6.26 データのエクスポートとインポート

### 6.26.1 CSV(一度に 1 つのテーブル)

*DAL* の `Rows` オブジェクトが文字列に変換される時、自動的に CSV 形式にシリアル化されます：

```
1 >>> rows = db(db.person.id==db.dog.owner).select()
2 >>> print rows
3 person.id,person.name,dog.id,dog.name,dog.owner
4 1,Alex,1,Skipper,1
5 1,Alex,2,Snoopy,1
6 2,Bob,3,Puppy,2
```

単一のテーブルを CSV 形式にシリアル化して、”test.csv” ファイルに格納することができます：

```
1 >>> open('test.csv', 'w').write(str(db(db.person.id).select()))
```

そして、次のようにして簡単にそれを読み取ることができます：

```
1 >>> db.person.import_from_csv_file(open('test.csv', 'r'))
```

インポートする時に、`web2py` は CSV のヘッダにあるフィールド名を探します。この例では、”person.name” と ”person.id” という 2 つのカラムを見つけます。”person” という接頭辞と、”id” というフィールドは無視されます。そして全てのレコードは追加され、新しい ID が割り当てられます。これら両方の操作は `appadmin` の Web・インターフェースを介して行うことができます。

### 6.26.2 CSV(全てのテーブルを一度に)

`web2py` では、次の 2 つのコマンドでデータベース全体をバックアップ / 復元することができます：

エクスポートするには：

```
1 >>> db.export_to_csv_file(open('somefile.csv', 'wb'))
```

インポートするには：

```
1 >>> db.import_from_csv_file(open('somefile.csv', 'rb'))
```

このメカニズムは、インポートしたデータベースがエクスポートするデータベースと異なるタイプのものでも使用することができます。データは”somefile.csv”に CSV ファイルとして格納されます。このファイルでは、各テーブルは、テーブル名を示す一つの行と、フィールド名を持つもう一つの行から始まります：

```
1 TABLE tablename
2 field1, field2, field3, ...
```

2 つのテーブルは\r\n\r\nで区切られます。ファイルは次の行で終わります

```
1 END
```

このファイルには、アップロードファイルがデータベースに格納されていない限り含まれません。どのような場合でも、”uploads” フォルダを個別に圧縮することは十分に簡単です。

インポートする時、新規のレコードはデータベースが空でない場合に、データベースに追加されます。一般に新しくインポートしたレコードは、元の(保存した)レコードと同じレコード id を持つことはありません。しかし web2py は参照も復元するので、id の値が変化しても参照が機能しなくなることはありません。

もしテーブルに”uuid”と呼ばれるフィールドが含まれる場合、そのフィールドは重複を識別するために使用されます。また、インポートしたレコードが既存のレコードと同じ”uuid”を持つ場合、既存のレコードは更新されます。

### 6.26.3 CSV とリモート・データベースの同期

次のモデルを考えてください：

```
1 db = DAL('sqlite:memory:')
2 db.define_table('person',
3     Field('name'),
4     format='%(name)s')
5 db.define_table('dog',
6     Field('owner', db.person),
7     Field('name'))
```

```

8     format='%(name)s')
9
10 if not db(db.person).count():
11     id = db.person.insert(name="Massimo")
12     db.dog.insert(owner=id, name="Snoopy")

```

各レコードは、ID によって識別され、その ID によって参照されます。別々にインストールした web2py によって利用されるデータベースの 2 つのコピーを持っているなら、ID は各データベースにおいてのみユニークで、データベース全体ではユニークではありません。これは、異なるデータベースからレコードをマージする時に問題になります。

データベース全体でレコードを一意に識別できるようにするには、レコードを次のようにする必要があります：

- 一意の ID(UUID) を持たせる
- event\_time を持たせる (複数のコピーがある場合、より最近のものを判別するため)
- id の代わりに UUID で参照する

これは web2py を変更することなく実現できます。以下にどのようにするかを示します：

### 1. 上記のモデルを次のように変更します：

```

1 db.define_table('person',
2     Field('uuid', length=64, default=lambda:str(uuid.uuid4())),
3     Field('modified_on', 'datetime', default=now),
4     Field('name'),
5     format='%(name)s')
6
7 db.define_table('dog',
8     Field('uuid', length=64, default=lambda:str(uuid.uuid4())),
9     Field('modified_on', 'datetime', default=now),
10    Field('owner', length=64),
11    Field('name'),
12    format='%(name)s')
13
14 db.dog.owner.requires = IS_IN_DB(db, 'person.uuid', '%(name)s')
15
16 if not db(db.person.id).count():
17     id = uuid.uuid4()
18     db.person.insert(name="Massimo", uuid=id)
19     db.dog.insert(owner=id, name="Snoopy")

```

上記のテーブル定義では、2つの *uuid* フィールドのデフォルト値が(文字に変換された) *UUID* を返すラムダ関数によってセットされています。ラムダ関数はそれぞれのレコードが挿入される際に呼び出され、複数のレコードが一つのトランザクションで挿入された場合でも、ユニークな *UUID* を取得するようにします。

## 2. データベースをエクスポートするコントローラの関数を作成します：

```

1 def export():
2     s = StringIO.StringIO()
3     db.export_to_csv_file(s)
4     response.headers['Content-Type'] = 'text/csv'
5     return s.getvalue()
```

## 3. 他のデータベースが保存したコピーをインポートし、レコードを同期するコントローラの関数を作成します：

```

1 def import_and_sync():
2     form = FORM(INPUT(_type='file', _name='data'), INPUT(_type='submit'))
3     if form.process(session=None).accepted:
4         db.import_from_csv_file(form.vars.data.file, unique=False)
5         # for every table
6         for table in db.tables:
7             # for every uuid, delete all but the latest
8             items = db(db[table]).select(db[table].id,
9                                           db[table].uuid,
10                                         orderby=db[table].modified_on,
11                                         groupby=db[table].uuid)
12             for item in items:
13                 db((db[table].uuid==item.uuid)& \
14                     (db[table].id!=item.id)).delete()
15     return dict(form=form)
```

この URL は外部から接続されることを想定しているので、`session=None` が CSRF 保護を無効にしている点に注意してください。

## 4. *uuid* による検索を高速化するためにインデックスを手動で作成します。

ただし、ステップ 2 と 3 は全てのデータベースモデルで機能します。この例では固有のものはないからです。

別の方針として、XML-RPC を用いてファイルをエクスポート/インポートすることができます。

レコードがアップロードしたファイルを参照する場合、uploads フォルダの中身もまたエクスポート/インポートする必要があります。ただし、ファイルは UUID で既にラベル付けされているので、名前の衝突と参照を心配する必要はありません。

#### 6.26.4 HTML/XML の(一度に一つのテーブル)

DAL の Rows オブジェクトはまた、(ヘルパのように) 自身を XML/HTML へとシリアル化する `xml` メソッドを持ちます：

```

1 >>> rows = db(db.person.id > 0).select()
2 >>> print rows.xml()
3 <table>
4   <thead>
5     <tr>
6       <th>person.id</th>
7       <th>person.name</th>
8       <th>dog.id</th>
9       <th>dog.name</th>
10      <th>dog.owner</th>
11    </tr>
12  </thead>
13  <tbody>
14    <tr class="even">
15      <td>1</td>
16      <td>Alex</td>
17      <td>1</td>
18      <td>Skipper</td>
19      <td>1</td>
20    </tr>
21    ...
22  </tbody>
23 </table>
```

DAL の Rows を、カスタムタグを持った他の XML フォーマットへとシリアル化したい場合は、普遍的なタグヘルパや\*表記を使用して簡単に行うことができます：

```

1 >>> rows = db(db.person.id > 0).select()
2 >>> print TAG.result(*[TAG.row(*[TAG.field(r[f], _name=f) \
3           for f in db.person.fields]) for r in rows])
4 <result>
5   <row>
6     <field name="id">1</field>
7     <field name="name">Alex</field>
```

```

8   </row>
9   ...
10 </result>
```

### 6.26.5 データ表現

`export_to_csv_file` 関数はキーワード引数 `represent` を持ちます。True の場合、データのエクスポート中に、生のデータの代わりに、カラムの `represent` 関数を用います。

この関数はまた、エクスポートしたいカラムの名前のリストを保持するキーワード引数 `colnames` を持ちます。デフォルトでは全てのカラムになります。

`export_to_csv_file` と `import_from_csv_file` の両方とも、CSV の構文解析機に保存/読み込み先のファイルのフォーマットを知らせる次のキーワード引数を持ちます：

- `delimiter`: 値の区切り文字の指定 (デフォルトは`,`)
- `quotechar`: 文字列値を引用符で囲むために使用する文字 (デフォルトはダブルクオート)
- `quoting`: 引用符の体系 (デフォルトは `csv.QUOTE_MINIMAL`)

ここでは、いくつか使用例を示します：

```

1 >>> import csv
2 >>> db.export_to_csv_file(open('/tmp/test.txt', 'w'),
3     delimiter='|',
4     quotechar='"',
5     quoting=csv.QUOTE_NONNUMERIC)
```

これは以下のようなレンダリングになります

```
1 "hello"|35|"this is the text description"| "2009-03-03"
```

より詳細な情報は公式の Python ドキュメントを参照してください。 [67]

### 6.27 選択のキャッシュ

`select` メソッドでは `cache` 引数を取ります。これはデフォルトでは `None` です。キャッシュの利用の際は、ここにタプルを設定する必要があります。このタプルの最初の要素はキャッシュモデルで (`cache.ram`、`chace.disk` など)、第 2 の要素は秒単位の有効期限です。

次の例では、前に定義した `db.log` テーブルに対する `select` をキャッシュするコントローラを設定しています。実際の `select` では 60 秒間隔より頻繁に、バックエンドのデータベースからデータを取り出すことはなく、`cache.ram` に結果を格納します。このコントローラへの次の呼び出しが、最終のデータベース IO から 60 秒以内に発生する場合、`cache.ram` から前回のデータが単純に取り出されます。

```
1 def cache_db_select():
2     logs = db().select(db.log.ALL, cache=(cache.ram, 60))
3     return dict(logs=logs)
```

`select` の結果は複雑で、*pickle* 化できないオブジェクトです。したがって、これらは `session` に格納することはできず、ここで説明したもの以外はどの方法でもキャッシュすることはできません。

### 6.28 自己参照と別名

自分自身を参照するフィールドを持つテーブルを定義することも可能ですが、通常の表記方法ではうまくいきません。次のコードは、`db.person` 変数を定義する前に使用しているので間違っています：

```
1 db.define_table('person',
2     Field('name'),
3     Field('father_id', db.person),
4     Field('mother_id', db.person))
```

解決策は次のような代替表記を使用することです

```
1 db.define_table('person',
2     Field('name'),
3     Field('father_id', 'reference person'),
4     Field('mother_id', 'reference person'))
```

実際、`db.tablename` と "reference tablename" は同じフィールドの型になります。

テーブルが自分自身を参照する場合、SQL の”AS” キーワードの使用なしに、JOIN を実行して、person とその親 (parents) を選択することは不可能です。これは、web2pyにおいて `with_alias` を用いて実現されます。以下がその例です。

```

1 >>> Father = db.person.with_alias('father')
2 >>> Mother = db.person.with_alias('mother')
3 >>> db.person.insert(name='Massimo')
4 1
5 >>> db.person.insert(name='Claudia')
6 2
7 >>> db.person.insert(name='Marco', father_id=1, mother_id=2)
8 3
9 >>> rows = db().select(db.person.name, Father.name, Mother.name,
10      left=(Father.on(Father.id==db.person.father_id),
11            Mother.on(Mother.id==db.person.mother_id)))
12 >>> for row in rows:
13     print row.person.name, row.father.name, row.mother.name
14 Massimo None None
15 Claudia None None
16 Marco Massimo Claudia

```

以下のものを区別して選択していることに注意してください：

- ”father\_id”: ”person” テーブルにおいて使用されるフィールド名
- ”father”: 上記のフィールドによって参照されるテーブルのために使用する別名。これはデータベースとやり取りされます。
- ”Father”: その別名を参照するための web2py によって使用される変数

僅かな違いなので、それら 3 つに同じ名前をつけても間違いではありません：

```

1 db.define_table('person',
2     Field('name'),
3     Field('father', 'reference person'),
4     Field('mother', 'reference person'))
5 >>> father = db.person.with_alias('father')
6 >>> mother = db.person.with_alias('mother')
7 >>> db.person.insert(name='Massimo')
8 1
9 >>> db.person.insert(name='Claudia')
10 2
11 >>> db.person.insert(name='Marco', father=1, mother=2)
12 3
13 >>> rows = db().select(db.person.name, father.name, mother.name,
14      left=(father.on(father.id==db.person.father),
15            mother.on(mother.id==db.person.mother)))
16 >>> for row in rows:

```

```

17     print row.person.name, row.father.name, row.mother.name
18 Massimo None None
19 Claudia None None
20 Marco Massimo Claudia

```

しかし、正しいケアリを構築するには、この区別を明確にすることが重要です。

## 6.29 高度な機能

### 6.29.1 テーブル継承

他のテーブルの全てのフィールドを含んだテーブルを、作成することが可能で  
す。これは、他のテーブルを `define_table` に置くだけで十分です。例えば次の  
ようになります。

```

1 db.define_table('person', Field('name'))
2 db.define_table('doctor', db.person, Field('specialization'))

```

データベースに格納されないダミーテーブルを定義して、他の複数の場所で再利  
用することも可能です。例:

```

1 signature = db.Table(db, 'signature',
2     Field('created_on', 'datetime', default=request.now),
3     Field('created_by', db.auth_user, default=auth.user_id),
4     Field('updated_on', 'datetime', update=request.now),
5     Field('updated_by', db.auth_user, update=auth.user_id))
6
7 db.define_table('payment', Field('amount', 'double'), signature)

```

この例は、標準の web2py 認証が有効になっていることを前提としています。

もし `Auth` を利用している場合は、このようなテーブルを web2py が既に作成済  
みです。

```

1 auth = Auth(db)
2 db.define_table('payment', Field('amount', 'double'), auth.signature)

```

テーブル継承を使用する際に、バリデータも継承したい場合は、継承テーブルを  
定義する前に継承元のバリデータを定義しておく必要があります。

### 6.29.2 コモンフィールドとマルチテナント

`db._common_fields` は全てのテーブルに属するフィールドのリストです。このリストにテーブルを含むこともでき、その場合は該当テーブルの全てのフィールドがリストされたとして理解します。例えば、`auth` を除く全てのテーブルに `signature` を追加したい場合があります。この場合、`db.define_tables()` の後で、且つ、それ以外のテーブルを定義する前に、以下を挿入します。

```
1 db._common_fields.append(auth.signature)
```

”`request_tenant`” は特別なフィールドです。このフィールドは（標準での定義が）存在しませんが、自分で作成して、いくつか（または全てのテーブル）のテーブルに追加できます。

```
1 db._common_fields.append(Field('request_tenant',
2     default=request.env.http_host,writable=False))
```

`db.request_tenant` というフィールドを持っているテーブルは、全てのクエリの全てのレコードが、常に自動でフィルタされます。

```
1 db.table.request_tenant == db.table.request_tenant.default
```

そしてレコードの挿入のたびに、デフォルトの値がセットされます。上記の例では以下がセットされます。

```
1 default = request.env.http_host
```

つまり、アプリ上の全てのテーブルとクエリを以下でフィルタするという選択をしたことになります。

```
1 db.table.request_tenant == request.env.http_host
```

この簡単なトリックで、アプリケーションを複数のテナントに対応したアプリケーションにすることができます。つまりアプリを、单一インスタンス上の单一データベースで運用しているおり、複数ドメイン（上記の例の場合、ドメイン名は `request.env.http_host` から取得）でのアクセスがアプリに対してある場合、訪問者が接続したドメイン名によって参照するデータが異なることになります。複数のオンラインストアを、單一アプリとデータベースを使用し異なるドメイン下で運用する場合などを、想像してみてください。

複数のテナントによるフィルタを無効にできます。

```
1 rows = db(query, ignore_common_filters=True).select()
```

### 6.29.3 コモンフィルタ

コモンフィルタは上記の複数のテナントという考え方を一般化したものです。同じクエリを繰り返し使用させない簡単な方法を提供します。次のテーブル例を考えてください。

```

1 db.define_table('blog_post',
2     Field('subject'),
3     Field('post_text', 'text'),
4     Field('is_public', 'boolean'),
5     common_filter = lambda query: db.blog_post.is_public==True
6 )

```

このテーブルに対する選択、削除、更新は公開されたブログ記事だけを含みます。その属性はコントローラで変更できます。

```

1 db.blog_post._common_filter = lambda query: db.blog_post.is_public ==
2     True

```

それぞれのブログ記事検索に”db.blog\_post.is\_public==True”という条件を繰り返し使用する必要がなくなり、非公開ブログ記事の参照不可の設定忘れ、といったセキュリティ面も向上もできます。

もし意図的にコモンフィルタを外したい（例、管理者は非公開のブログ記事を参照できる）場合は、フィルタを削除することができます。

```

1 db.blog_post._common_filter = None

```

もしくは、無視するには次のようにします。

```

1 db(query, ignore_common_filters=True).select(...)

```

### 6.29.4 カスタム Field型 (実験的)

新しい/カスタムフィールド型を定義することが可能です。圧縮されたバイナリデータを含むフィールドの例です。

```

1 from gluon.dal import SQLCustomType
2 import zlib
3
4 compressed = SQLCustomType(
5     type ='text',
6     native='text',

```

```

7     encoder = (lambda x: zlib.compress(x or '')),
8     decoder = (lambda x: zlib.decompress(x))
9 )
10
11 db.define_table('example', Field('data', type=compressed))

```

`SQLCustomType` はフィールド型のファクトリです。その `type` 引数は web2py の標準型の一つでなければなりません。web2py レベルで、そのフィールド値をどのように扱うべきか指示します。`native` はデータベースが接続されているかぎり使用できるフィールドの名前です。データベースエンジン特有の型名も許可します。`encoder` はデータ格納時に適用されるオプションの変換関数で、`decoder` は逆変換の関数です。

この機能は実験的とされています。現実的に長い間使用されて動作していますが、コードがポータブルでなくなります。例として、データベース特有のフィールド型を使用した場合に Google App Engine NoSQL で動作しなくなります。

### 6.29.5 テーブル定義なしで DAL を使用

DAL は以下のようにすることで、どのような Python プログラムからでも使用できます。

```

1 from gluon import DAL, Field
2 db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases')

```

つまり、DAL と Field をインポートし、接続、.table ファイル（app/databases フォルダ）を含むフォルダを指定すればよいです。

データやその属性にアクセスするには、`db.define_tables(...)` で接続する全てのテーブルを定義する必要があります。

もしデータにだけアクセスし、web2py テーブル属性は必要がない場合は、.table ファイルにあるメタデータから必要な情報を読み込むように web2py に指示するだけです。テーブルを再定義する必要はないです。

```

1 from gluon import DAL, Field
2 db = DAL('sqlite://storage.sqlite', folder='path/to/app/databases',
3           auto_import=True)

```

これによって再定義せずに `db.table` に接続できます。

### 6.29.6 異なる db からデータをコピー

以下のデータベースを使用している場合を考えてください。

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

そして異なる接続文字で、別のデータベースに移動したいとします：

```
1 db = DAL('postgresql://username:password@localhost/mydb')
```

切り替える前に、新しいデータベースへデータを移動してメタデータを再構築したいです。新しいデータベースは存在するが空であると想定します。web2pyはこれを実現するスクリプトを提供します：

```
1 cd web2py
2 python scripts/cpdb.py \
3   -f applications/app/databases \
4   -y 'sqlite://storage.sqlite' \
5   -Y 'postgresql://username:password@localhost/mydb'
```

スクリプト実行後、単にモデルの接続文字を切り替えるだけで、全て動きます。新しいデータも存在しています。

このスクリプトはひとつのアプリケーションから別のアプリケーションへデータを移動できる様々なコマンドラインオプションを提供し、全てのテーブルを移動したり、一部だけ移動したり、テーブルのデータをクリアしたりします。詳しくは次を実行してみてください。

```
1 python scripts/cpdb.py -h
```

### 6.29.7 新しい DAL とアダプタの注意点

データベース抽象化レイヤのソースコードは 2010 年に全て書き換えられました。後方互換性を保ちながら、よりモジュール化し拡張性しやすくすることができました。ここで主要なロジックについて説明します。

”gluon/dal.py” ファイルはとりわけ次のクラスを定義します。

```
1 ConnectionPool
2 BaseAdapter extends ConnectionPool
3 Row
4 DAL
5 Reference
```

```

6 Table
7 Expression
8 Field
9 Query
10 Set
11 Rows

```

`BaseAdapter` を除き、使用方法については前節で説明しました。`Table` や `Set` オブジェクトがデータベースと通信する必要がある場合、アダプタに SQL を生成したり、関数を呼び出すメソッドを委譲します。

例：For example:

```
1 db.myable.insert(myfield='myvalue')
```

以下を呼び出します calls

```
1 Table.insert(myfield='myvalue')
```

これは以下を返すことでアダプタに委譲します：

```
1 db._adapter.insert(db.mytable, db.mytable._listify(dict(myfield='myvalue
    ')))
```

ここで `db.mytable._listify` は引数の辞書を `(field, value)` のリストに変換し、`adapter` の `insert` メソッドを呼びます。`db._adapter` は大体次のようなことをしています。

```
1 query = db._adapter._insert(db.mytable, list_of_fields)
2 db._adapter.execute(query)
```

最初の行はクエリを作り、2行目で実行しています。

`BaseAdapter` は全てのアダプタのインターフェースを定義します。

執筆時点で、”gluon/dal.py” は次のアダプタを含みます。

```

1 SQLiteAdapter extends BaseAdapter
2 JDBCSQLiteAdapter extends SQLiteAdapter
3 MySQLAdapter extends BaseAdapter
4 PostgreSQLAdapter extends BaseAdapter
5 JDBCPostgreSQLAdapter extends PostgreSQLAdapter
6 OracleAdapter extends BaseAdapter
7 MSSQLAdapter extends BaseAdapter
8 MSSQL2Adapter extends MSSQLAdapter
9 FireBirdAdapter extends BaseAdapter
10 FireBirdEmbeddedAdapter extends FireBirdAdapter

```

```

11 InformixAdapter extends BaseAdapter
12 DB2Adapter extends BaseAdapter
13 IngresAdapter extends BaseAdapter
14 IngresUnicodeAdapter extends IngresAdapter
15 GoogleSQLAdapter extends MySQLAdapter
16 NoSQLAdapter extends BaseAdapter
17 GoogleDatastoreAdapter extends NoSQLAdapter
18 CubridAdapter extends MySQLAdapter (experimental)
19 TeradataAdapter extends DB2Adapter (experimental)
20 SAPDBAdapter extends BaseAdapter (experimental)
21 CouchDBAdapter extends NoSQLAdapter (experimental)
22 MongoDBAdapter extends NoSQLAdapter (experimental)

```

これは `BaseAdapter` をオーバーライドします。

それぞれのアダプタは、おおよそ次のような構造です。

```

1 class MySQLAdapter(BaseAdapter):
2
3     # specify a diver to use
4     driver = globals().get('pymysql',None)
5
6     # map web2py types into database types
7     types = {
8         'boolean': 'CHAR(1)',
9         'string': 'VARCHAR(%(length)s)',
10        'text': 'LONGTEXT',
11        ...
12    }
13
14     # connect to the database using driver
15     def __init__(self,db,uri,pool_size=0,folder=None,db_codec ='UTF-8',
16                  credential_decoder=lambda x:x, driver_args={},adapter_args={}):
17
18         # parse uri string and store parameters in driver_args
19         ...
20
21         # define a connection function
22         def connect(driver_args=driver_args):
23             return self.driver.connect(**driver_args)
24
25         # place it in the pool
26         self.pool_connection(connect)
27         # set optional parameters (after connection)
28         self.execute('SET FOREIGN_KEY_CHECKS=1;')
29         self.execute("SET sql_mode='NO_BACKSLASH_ESCAPES';")
30
31     # override BaseAdapter methods as needed
32     def lastrowid(self,table):
33         self.execute('select last_insert_id();')
34         return int(self.cursor.fetchone()[0])

```

様々なアダプタの例を参考にすれば、新しいアダプタを記述するのも簡単です。

db インスタンスが作られると：

```
1 db = DAL('mysql://...')
```

uri 文字列の接頭辞はアダプタを定義しています。マッピングは”gluon/dal.py”で、次のように辞書型で定義されています。

```
1 ADAPTERS = {
2     'sqlite': SQLiteAdapter,
3     'sqlite:memory': SQLiteAdapter,
4     'mysql': MySQLAdapter,
5     'postgres': PostgreSQLAdapter,
6     'oracle': OracleAdapter,
7     'mssql': MSSQLAdapter,
8     'mssql2': MSSQL2Adapter,
9     'db2': DB2Adapter,
10    'teradata': TeradataAdapter,
11    'informix': InformixAdapter,
12    'firebird': FireBirdAdapter,
13    'firebird_embedded': FireBirdAdapter,
14    'ingres': IngresAdapter,
15    'ingresu': IngresUnicodeAdapter,
16    'sapdb': SAPDBAdapter,
17    'cubrid': CubridAdapter,
18    'jdbc:sqlite': JDBCSQLiteAdapter,
19    'jdbc:sqlite:memory': JDBCSQLiteAdapter,
20    'jdbc:postgres': JDBCPostgreSQLAdapter,
21    'gae': GoogleDatastoreAdapter, # discouraged, for backward
22        compatibility
23    'google:datastore': GoogleDatastoreAdapter,
24    'google:sql': GoogleSQLAdapter,
25    'couchdb': CouchDBAdapter,
26    'mongodb': MongoDBAdapter,
27 }
```

uri 文字列はアダプタ自身で、より詳細に解析されます。

どのようなアダプタでも異なるドライバに置き換えることができます：

```
1 from gluon.dal import MySQLAdapter
2 MySQLAdapter.driver = mysqldb
```

オプションのドライバ引数やアダプタ引数を指定することもできます。

```
1 db = DAL(..., driver_args={}, adapter_args={})
```

第3版 - 翻訳: 細田謙二 レビュー: Omi Chiba

第4版 - 翻訳: Omi Chiba レビュー: Hitoshi Katoweb2py は 2007 年に立ち上げられ、4 年間の継続的な開発を経た現在、私たちは待望の第4版を完成了。この間に web2py は、数千人の知識豊富なユーザと百人以上の開発者の愛情を得てきました。私たちの努力の結果、実存する中で一番多機能なオープンソースの web フレームワークの一つが完成了。

当初、私は教材として web2py を作成しました。なぜなら、良質な web アプリケーションを構築することは、自由でオープンな社会の成長に非常に重要だと信じているからです。そして大手企業や団体による情報の独占を防止します。今もその思いに変わりはなく、むしろさらに重要性を増しています。

一般的にどの web フレームワークの目的も、簡単に、早く、特にセキュリティに関する開発者の間違いを防いで、web 開発を行うことです。web2py では、私たちは三つの主要な目標で、これらの問題に取り組んでいます。

使いやすさは、web2py の一番の目標です。私たちにとって、学習と開発に関わる時間を減らすという意味があります。またこれは web2py が、他に依存する物を持たないフルスタックフレームワークである理由です。インストールも設定ファイルも必要ありません。web サーバー、データベース、全ての機能にアクセスできる web ベースの IDE を含む全てが標準で動作します。API は 12 のコアオブジェクトに単純化されており、簡単に覚えて使うことができます。大部分の web サーバーやデータベース、そして全ての Python ライブラリと互換性を持ちます。

開発の高速化は二つ目の目標です。web2py のどの関数も(上書き可能な)デフォルトの振る舞いを持っています。例えば、データモデルを指定するとすぐに、web ベースのデータベース管理パネルにアクセスできます。また、web2py はデータのフォームを自動で作成し、HTML、XML、JSON、RSS などのデータを簡単に公開できます。

セキュリティは web2py の心臓部で、システムとデータを安全に保つために全てを保護することが目標です。このためデータベースレイヤは、SQL インジェクションを取り除きます。テンプレート言語は、クロスサイトスクリプティングの脆弱性を妨ぎます。web2py によって生成されるフォームは、フィールドのバリデーションを提供し、クロスサイトリクエストフォージェリをブロックします。パスワードは常にハッシュ化して保存されます。セッションはクッキーの改ざ

んを防ぐために、デフォルトでサーバーサイドに保存され、セッションクッキーはクッキーの盗難を防ぐために uuid を使用します。

web2py は常にユーザ視点で開発され、何時でも後方互換を保ちながら、より速くよりスリムになるように常に内部最適化が続けられます。

web2py は無償で使用できます。もしあなたが web2py で利益を得ることができた場合には、どのような形でもいいので社会に恩返しをする心を持っていただけることを希望します。

2011 年の InforWorld magazine で、最も人気のあるフルスタックで Python ベースの web フレームワークの 6 つに選ばれ、その中でも最高ランクでした。また同じく 2011 年に、Bossie Award の最も優れたオープンソース開発ソフトを受賞しました。

第 3 版 - 翻訳: Omi Chiba レビュー: 中垣健志

第 4 版 - 翻訳: Omi Chiba レビュー: Hitoshi Kato

## 6.30 翻訳

この本は web2py Japan のメンバーによって翻訳されました。

名前のアルファベット順:

- Fumito MIZUNO (GitHub <https://github.com/ounziw> )
- Hitoshi Kato (blog : <http://todayspython.blogspot.com> )
- Kenji Hosoda ([hosoda@s-cubism.jp](mailto:hosoda@s-cubism.jp))
- Kenji Nakagaki (blog 「IT Virtuoso」)
- Mitsuhiro Tsuda ([mtsuda@ipallet.org](mailto:mtsuda@ipallet.org))
- Omi Chiba (blog 「Python Roll」)
- Yota Ichino (GitHub <https://github.com/nus> )## コア

# 7

## フォームとバリデータ

web2py でフォームを構築するには以下の 4 通りの方法があります :

- FORM は HTML ヘルパに関して低レベルの実装を提供します。FORM オブジェクトは HTML へシリアル化することができ、そこに含まれるフィールドについて把握しています。FORM オブジェクトは送信フォームの値を検証することができます。
- SQLFORM は、作成、更新、削除のフォームを、既存のデータベーステーブルから構築するための高レベルの API を提供します。

-SQLFORM.factory は SQLFORM の上にある抽象化レイヤです。データベースが用意されてない場合でもフォーム生成機能を活用できるようにしています。これは、テーブルの記述から SQLFORM とともに良く似たフォームを生成します。ただしデータベース・テーブルを作成する必要はありません。

- CRUD メソッド。SQLFORM と同等で SQLFORM に基づく関数が用意されていますが、よりコンパクトな表記が可能になります。

これらすべてのフォームは自分自身を把握しており、入力が検証を通らなかった場合、自分自身を修正してエラーメッセージを加えることができます。フォームでは、検証によって生成された検証済み変数とエラーメッセージを問い合わせることができます。

ヘルパを用いて、任意の HTML コードをフォームへ挿入、また、フォームから抽出することができます。

FORM と SQLFORM はヘルパで DIV のように扱えます。例えば、フォームスタイルを指定できます：

```
1 form = SQLFORM(...)
2 form['_style']='border:1px solid black'
```

## 7.1 FORM

次のような”default.py” コントローラを持つ test アプリケーションを考えます：

```
1 def display_form():
2     return dict()
```

関連付けるビュー”default/display\_form.html” は以下のようにします：

```
1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 <form enctype="multipart/form-data"
4     action="{{=URL()}}" method="post">
5 Your name:
6 <input name="name" />
7 <input type="submit" />
8 </form>
9 <h2>Submitted variables</h2>
10 {{=BEAUTIFY(request.vars)}}
```

これは、ユーザー名を問い合わせる一般的な HTML フォームです。このフォームを入力しサブミット・ボタンをクリックすると、フォームは自分自身をサブミットし、request.vars.name 変数とその値が下部に表示されます。

同じフォームをヘルパを用いて生成することができます。これは、ビュー、または、アクションにおいて行うことができます。web2py はフォームの処理をアクションにおいて行うので、フォームをアクションで定義しても差し支えありません。

これが新しいコントローラです：

```
1 def display_form():
2     form=FORM('Your name:', INPUT(_name='name'), INPUT(_type='submit'))
3     return dict(form=form)
```

関連付けるビュー”default/desplay\_form.html” は以下のようにします：

```

1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 {{=form}}
4 <h2>Submitted variables</h2>
5 {{=BEAUTIFY(request.vars)}}

```

以前のコードは上のコードと等しいです。しかし、フォームは、`FORM`オブジェクトをシリアル化する`{{=form}}`という文によって生成されています。

次に、フォームの検証と処理を追加して、一段複雑なものを加えます。

コントローラを以下のように変更します：

```

1 def display_form():
2     form=FORM('Your name:',
3               INPUT(_name='name', requires=IS_NOT_EMPTY()),
4               INPUT(_type='submit'))
5     if form.accepts(request, session):
6         response.flash = 'form accepted'
7     elif form.errors:
8         response.flash = 'form has errors'
9     else:
10        response.flash = 'please fill the form'
11
12     return dict(form=form)

```

関連付けるビュー”default/display\_form.html”は以下のようにします：

```

1 {{extend 'layout.html'}}
2 <h2>Input form</h2>
3 {{=form}}
4 <h2>Submitted variables</h2>
5 {{=BEAUTIFY(request.vars)}}
6 <h2>Accepted variables</h2>
7 {{=BEAUTIFY(form.vars)}}
8 <h2>Errors in form</h2>
9 {{=BEAUTIFY(form.errors)}}

```

以下のことに注意してください：

- アクションでは、入力フィールド”name”にに対して`requires=IS_NOT_EMPTY()`バリデータを加えています。
- アクションでは、`form.accepts(...)`の呼び出しを加えています。
- ビューでは、フォームと`request.vars`とともに`form.vars`と`form.errors`を表示しています。

すべての作業は、`form` オブジェクトの `accepts` メソッドによって行われます。これは、`request.vars` を、(バリデータによって表現された) 宣言された要求に従って、フィルタします。`accepts` は、検証を通ったこれらの変数を `form.vars` に格納します。フィールドの値が何かしら要求を満たさない場合は、失敗したバリデータがエラーを返し、そのエラーが `form.errors` に格納されます。`form.vars` と `form.errors` は両方とも、`request.vars` に似た `gluon.storage.Storage` オブジェクトです。前者は、次のように検証を通った値を保持します：

```
1 form.vars.name = "Max"
```

後者は、次のようにエラーを保持します：

```
1 form.errors.name = "Cannot be empty!"
```

`accepts` メソッドのすべての用法は以下の通りです：

```
1 form.accepts(vars, session=None, formname='default',
2               keepvalues=False, onvalidation=None,
3               dbio=True, hideerror=False):
```

これらオプション・パラメータの意味は、次の小節で説明します。

最初の引数には `request.vars`、`request.get_vars`、`request.post_vars` や、単に `request` と指定できます。後者は `request.post_vars` と入力したのと等しく処理されます。

`accepts` 関数はフォームが受理されたときに `True` を返し、そうでない場合は `False` を返します。フォームは、エラーがある場合か、サブミットされてない場合（たとえば、最初に表示されるとき）には受理されません。

次に示すのは、このページが最初に表示されたときの様子です：

The screenshot shows a web browser window with the URL `http://127.0.0.1:8000/test/default/display_form`. The page title is "test". A green banner at the top says "customize me!". On the right, there is a button labeled "please fill the form". Below the banner, there are two tabs: "Index" and "Edit".

**Input form**

Your name:

**Submitted variables**

**Accepted variables**

**Errors in form**

Copyright © 2010 - Powered by web2py

A red error message "form has errors" is displayed above the "Submitted variables" section. The "Your name:" field contains "Max".

無効なサブミットをしたときの様子です：

The screenshot shows a web browser window with the URL `http://127.0.0.1:8000/test/default/display_form`. The page title is "test". A green banner at the top says "customize me!". On the right, there is a button labeled "form has errors". Below the banner, there are two tabs: "Index" and "Edit".

**Input form**

Your name:

**Submitted variables**

name :

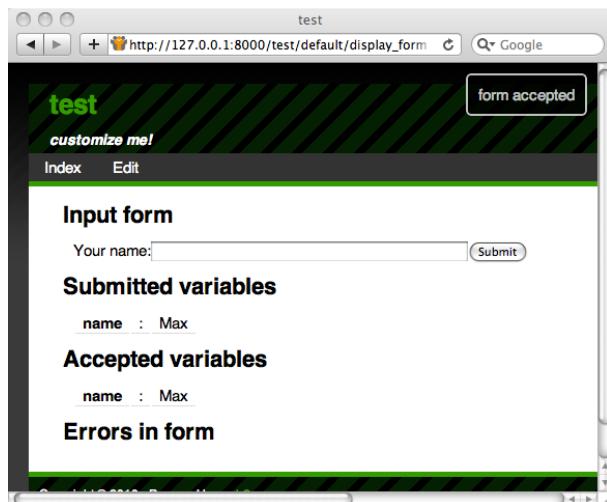
**Accepted variables**

name :

**Errors in form**

A red error message "enter a value" is displayed above the "Submitted variables" section. The "Your name:" field is empty.

有効なサブミットをしたときの様子です：



### 7.1.1 process と validate メソッド

以下のショートカットは

```
1 form.accepts(request.post_vars, session, ...)
```

このようになります

```
1 form.process(...).accepted
```

後者は `request` と `session` 引数を必要としません（任意で指定は可能ですが）、またフォーム自身を返すため、`accepts` とは異なります。内部的には、`process` が `accepts` を呼び出し、その引数を渡します。`accepts` によって返された値が `form.accepted` に保存されます。`process` 関数は `accepts` に存在しない追加の引数を受け取れます。

- `message_onsuccess`
- `onsuccess: 'flash'` (既定) に相当し、フォームが受理されると上記の '`message_onsuccess`' を表示します。 - `message_onfailure`

```
1 -
```

`onfailure: 'flash'` (既定) に相当し、フォームが検証を通らなかった場合、上記の '`message_onfailure`' を表示します。

- `next` フォームが受理されるとユーザーはリダイレクトされます。
- `onsuccess` と `onfailure` には `lambda form: do_something(form)` のような関数も使用できます。

```
1 form.validate(...)
```

は以下のショートカットです。

```
1 form.process(..., dbio=False).accepted
```

### 7.1.2 隠しフィールド

上記のフォーム・オブジェクトが`{=form}`によってシリアル化されたとき、`accepts` メソッドに対する前述の呼び出しがあるために、フォームは次のようにになります：

```
1 <form enctype="multipart/form-data" action="" method="post">
2 your name:
3 <input name="name" />
4 <input type="submit" />
5 <input value="783531473471" type="hidden" name="_formkey" />
6 <input value="default" type="hidden" name="_formname" />
7 </form>
```

注意する点は、2つの隠しフィールド”`_formkey`”と”`_formname`”があることです。これらの存在は、`accepts` の呼び出しによって引き起こされたもので、2つの異なる重要な役割を果たします：

- ”`_formkey`”という隠しフィールドは一度限りのトーケンで、web2pyがフォームの二重投稿を防ぐために用いられます。このキーの値はフォームがシリアル化されたときに生成され、`session` に保存されます。フォームがサブミットされたときに、この値が一致する必要があります。そうでないと `accepts` は、フォームが全くサブミットされてないかのように、エラーなしで `False` を返します。これは、フォームが正しくサブミットされたかどうかを web2py が判断できないためです。
- ”`_formname`”という隠しフィールドは、フォームの名前として web2py によって生成されますが、その名前は上書きすることができます。このフィールドは、ページが複数のフォームを含んで処理することを可能にするために

必要です。web2py は、この名前によって異なるサブミットされたフォームを区別します。

- オプション的な隠しフィールドは `FORM(..., hidden=dict(...))` のように指定します。

これらの隠しフィールドの役割と、カスタムフォームと複数のフォームを持つページにおける使用方法は、本章の後半で詳しく説明します。

上記のフォームを空の”name” フィールドでサブミットした場合、フォームは検証を通過しません。フォームが再びシリアル化されるときは、次のように表示されます：

```

1 <form enctype="multipart/form-data" action="" method="post">
2 your name:
3 <input value="" name="name" />
4 <div class="error">cannot be empty!</div>
5 <input type="submit" />
6 <input value="783531473471" type="hidden" name="_formkey" />
7 <input value="default" type="hidden" name="_formname" />
8 </form>
```

シリアル化したフォームにある DIV のクラス”error” の存在に注意してください。web2py はこのエラーメッセージをフォームに挿入し、検証を通過しなかったフィールドについて訪問者に知らせます。サブミットの際の `accepts` メソッドは、フォームがサブミットされたかどうかを判断し、フィールド”name” が空でないか、また、それが要求されているかをチェックし、最終的に、バリデータからフォームにエラーメッセージを挿入します。

基底の”layout.html” ビューは、DIV クラスの”error” を処理することが想定されています。デフォルトのレイアウトは jQuery のエフェクトを使用して、エラーを可視化し、赤い背景とともにスライドダウンさせます。詳細は第 11 章を参照してください。

### 7.1.3 keepvalues

オプション引数 `keepvalues` は、フォームが受理され、かつ、リダイレクトがないときに、web2py に何をするか知らせ、同じフォームが再び表示されるようにします。デフォルトではすべてクリアされます。`keepvalues` が `True` の場合、フォームは前回挿入した値を事前に入力します。これは、複数の似たレコードを

繰り返し挿入するために使用することを想定したフォームがあるときに便利です。`dbio` 引数が `False` の場合、`web2py` は、フォームを受理した後、いかなる DB の挿入/更新も行いません。`hideerror` が `True` でフォームにエラーが含まれている場合、フォームがレンダリングされたときにエラーは表示されません (`form.errors` をどのように表示するかは開発者次第です)。`onvalidation` 引数は以下に説明します。

#### 7.1.4 onvalidation

`onvalidation` 引数は `None` もしくは、フォームを受け取り何も返さない関数をとることができます。そのような関数は、検証(が通った)直後に、かつ、それ以外のことが発生する前に呼ばれ、フォームを渡します。この関数の目的は複数あります。これは、たとえば、追加的なフォームのチェックを実行したり、最終的にフォームにエラーを加えたりすることができます。これはまた、いくつかのフィールドの値を、他のフィールドの値に基づいて計算するのに使用することもできます。これを用いて、レコードが作成/更新される前にいくつかのアクション(`email` の送信など)を引き起こすことも可能です。

以下がその例です。

```
1 db.define_table('numbers',
2     Field('a', 'integer'),
3     Field('b', 'integer'),
4     Field('c', 'integer', readable=False, writable=False))
5
6 def my_form_processing(form):
7     c = form.vars.a * form.vars.b
8     if c < 0:
9         form.errors.b = 'a*b cannot be negative'
10    else:
11        form.vars.c = c
12
13 def insert_numbers():
14     form = SQLFORM(db.numbers)
15     if form.process(onvalidation=my_form_processing).accepted:
16         session.flash = 'record inserted'
17         redirect(URL())
18     return dict(form=form)
```

### 7.1.5 レコード変更の検知

レコードを編集するためにフォームを入力している際に、わずかですが他のユーザーが同じレコードを同時に変更している可能性があります。そこでレコードを保存する際に競合していないかチェックしたいです。これは以下のように実施できます。

```

1 db.define_table('dog', Field('name'))
2
3 def edit_dog():
4     dog = db.dog(request.args(0)) or redirect(URL('error'))
5     form=SQLFORM(db.dog,dog)
6     form.process(detect_record_change=True)
7     if form.record_changed:
8         # do something
9     elif form.accepted:
10        # do something else
11    else:
12        # do nothing
13    return dict(form=form)

```

### 7.1.6 フォームとリダイレクト

フォームを使用する最も一般的な方法は、自己サブミットを介して、サブミットされたフィールドの変数が、フォームを生成したものと同じアクションによって処理されるようにすることです。フォームが一旦受理されれば、現在のページを再び表示することはあまりありません（ここでは説明を単純にするためいくつか行っています）。訪問者を”next”ページへリダイレクトさせるのがより一般的です。

ここでは新しいコントローラの例を示します：

```

1 def display_form():
2     form = FORM('Your name:',
3                 INPUT(_name='name', requires=IS_NOT_EMPTY()),
4                 INPUT(_type='submit'))
5     if form.process().accepted:
6         session.flash = 'form accepted'
7         redirect(URL('next'))
8     elif form.errors:
9         response.flash = 'form has errors'
10    else:
11        response.flash = 'please fill the form'

```

```

12     return dict(form=form)
13
14 def next():
15     return dict()

```

現在のページの代わりに next ページで flash を設定するために、`session.flash` を `response.flash` の代わりに設定する必要があります。web2py はリダイレクト後、前者を後者に移します。`session.flash` は `session.forget()` を使用していないことが前提であることに注意してください。

### 7.1.7 ページ毎に複数のフォーム

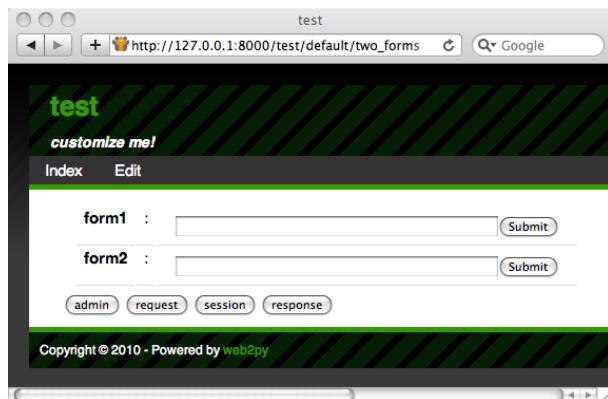
この節の内容は、FORM と SQLFORM オブジェクトどちらにも適用されます。ページ毎に複数のフォームを持つことが可能です。しかし、web2py にそれらを区別できるようにしなければなりません。異なるテーブルの SQLFORM によって生成されたものならば、web2py は異なる名前を自動的にそれらに与えます。それ以外の場合は、異なるフォームの名前を明示的に与えなければなりません。以下がその例です。

```

1 def two_forms():
2     form1 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
3                  INPUT(_type='submit'))
4     form2 = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
5                  INPUT(_type='submit'))
6     if form1.process(formname='form_one').accepted:
7         response.flash = 'form one accepted'
8     if form2.process(formname='form_two').accepted:
9         response.flash = 'form two accepted'
10    return dict(form1=form1, form2=form2)

```

ここに生成された出力を示します：



訪問者が空の form1 をサブミットした場合、form1 のみがエラーを表示します。一方、訪問者が空の form2 をサブミットした場合、form2 のみがエラーメッセージを表示します。

### 7.1.8 フォームの共有

この節の内容は、FORM と SQLFORM オブジェクトどちらにも適用されます。ここで説明することは可能ですが推奨されません。自己サブミットするフォームを持つことがベストプラクティスだからです。しかし、場合によっては、フォームを送受信するアクションが異なるアプリケーションに属していて、選択肢がないことがあります。

異なるアクションへサブミットするフォームを生成することは可能です。これは、FORM または SQLFORM オブジェクトの属性において、処理するアクションの URL を指定することで行われます。例:

```

1 form = FORM(INPUT(_name='name', requires=IS_NOT_EMPTY()),
2             INPUT(_type='submit'), _action=URL('page_two'))
3
4 def page_one():
5     return dict(form=form)
6
7 def page_two():
8     if form.process(session=None, formname=None).accepted:
9         response.flash = 'form accepted'
10    else:
11        response.flash = 'there was an error in the form'
12    return dict()

```

”page\_one” と ”page\_two” は両者とも同じ `form` を利用しているので、同じこの繰り返しを避けるために、そのフォームをすべてのアクションの外側で一度だけ定義していることに注意してください。コントローラの冒頭にある共通のコード部分は、アクションの呼び出しに制御を渡す前に毎回実行されます。

”page\_one” は `process(または、accepts)` を呼び出さないため、`form` には名前がなくキーもありません。そのため、`session=None` を渡し、`formname=None` を設定する必要があります。そうでないと、フォームは ”page\_two” に受け取られたときに検証を行いません。

## 7.2 SQLFORM

次の段階に進んで、以下のようなアプリケーションのモデルファイルを用意します：

```
1 db = DAL('sqlite://storage.sqlite')
2 db.define_table('person', Field('name', requires=IS_NOT_EMPTY()))
```

コントローラを以下のように変更します：

```
1 def display_form():
2     form = SQLFORM(db.person)
3     if form.process().accepted:
4         response.flash = 'form accepted'
5     elif form.errors:
6         response.flash = 'form has errors'
7     else:
8         response.flash = 'please fill out the form'
9     return dict(form=form)
```

ビューを変更する必要はありません。

コントローラでは、`FORM` を構築する必要はありません。なぜなら、`SQLFORM` は、モデルに定義された `db.person` テーブルからそれを構築するからです。この新しいフォームがシリアル化されると次のように表示されます：

```
1 <form enctype="multipart/form-data" action="" method="post">
2     <table>
3         <tr id="person_name_row">
4             <td><label id="person_name_label"
5                 for="person_name">Your name: </label></td>
6             <td><input type="text" class="string"
7                 name="name" value="" id="person_name" /></td>
```

```

8      <td></td>
9  </tr>
10 <tr id="submit_record__row">
11   <td></td>
12   <td><input value="Submit" type="submit" /></td>
13   <td></td>
14 </tr>
15 </table>
16 <input value="9038845529" type="hidden" name="_formkey" />
17 <input value="person" type="hidden" name="_formname" />
18 </form>

```

この自動的に生成されたフォームは、前述の低レベルのフォームよりも複雑です。第1に、これはテーブルの行を含み、各行は3つのカラムを持っています。最初のカラムはフィールドの名前を保持しています (`db.person` から決定されます)。第2のカラムは入力フィールドを保持します (最終的にエラーメッセージも保持します)。第3のカラムは省略可能で空になっています (`SQLFORM` のコンストラクタにおいてフィールドを用いて入力される可能性があります)。

フォームのすべてのタグには、テーブルとフィールドの名前に由来する名前が付けられています。これにより CSS と JavaScript を用いてフォームをカスタマイズするのが容易になります。この機能については、第11章で詳しく説明します。

ここでより重要なのは、`accepts` メソッドがより多くの仕事をすることです。前回の場合と同様、入力の検証を行いますが、加えて、入力が検証を通ったら、データベースに対して新規レコードの挿入を実行し、`form.vars.id` に新規レコードのユニークな”id”を格納します。

`SQLFORM` オブジェクトはまた、自動的に”upload”フィールドを処理し、アップロードしたファイルを”uploads”フォルダに保存します (競合を避けるため安全にリネームして、ディレクトリ・トラバーサル攻撃を防いだ後に保存します)。そして、(新しい) ファイル名をデータベースの適切なフィールドに保存します。フォームが処理されると、新しいファイル名は `form.vars.fieldname` (つまり、`cgi.FieldStorage` オブジェクトの値を `request.vars.fieldname` に置き換える) で参照可能なので、更新後、簡単にその名前を参照することができます。

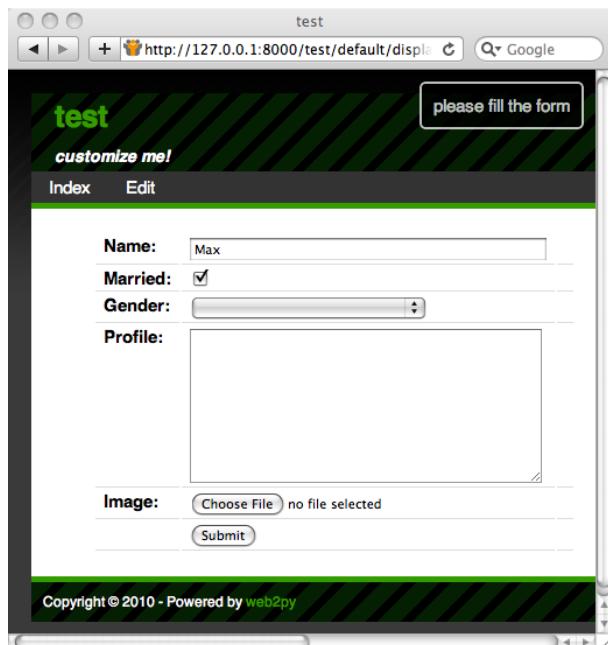
`SQLFORM` は、”boolean”の値をチェックボックスで、”text”の値をテキストエリアで、限定したセットまたはデータベース内に含まれることを要求された値をドロップボックスで、”upload”フィールドをアップロードしたファイルをダウンロードできるようにしたリンクで、表示します。”blob”フィールドは非表示し

ます。後述しますが、異なる方法で処理されることになるからです。

たとえば、次のモデルを考えてください：

```
1 db.define_table('person',
2     Field('name', requires=IS_NOT_EMPTY()),
3     Field('married', 'boolean'),
4     Field('gender', requires=IS_IN_SET(['Male', 'Female', 'Other'])),
5     Field('profile', 'text'),
6     Field('image', 'upload'))
```

この場合、SQLFORM(db.person) は次に表示されるようなフォームを生成します：



SQLFORM のコンストラクタは、さまざまなカスタマイズを可能にします。たとえば、フィールドの一部のみを表示したり、ラベルを変更したり、オプション的な第 3 のカラムに値を加えたり、現在の INSERT フォームとは対照的に UPDATE と DELETE フォームを作成したりすることができます。SQLFORM は web2py において、最も大きく時間を節約できる単一のオブジェクトです。

SQLFORM クラスは "gluon/sqlhtml.py" に定義されています。これは、xml メソッドをオーバーライドして簡単に拡張することができます。このメソッドはこのオ

プロジェクトをシリアル化して、その出力を変更します。

SQLFORM のコンストラクタの用法は以下の通りです：

```

1 SQLFORM(table, record = None,
2         deletable = False, linkto = None,
3         upload = None, fields = None, labels = None,
4         col3 = {}, submit_button = 'Submit',
5         delete_label = 'Check to delete:',
6         showid = True, readonly = False,
7         comments = True, keepopts = [],
8         ignore_rw = False, record_id = None,
9         formstyle = 'table3cols',
10        buttons = ['submit'], separator = ': ',
11        **attributes)

```

- オプション的な第 2 の引数は、INSERT フォームから、指定したレコードに対する UPDATE フォームに切り替えます(次の小節を参照してください)。

`deletable` が `True` の場合、UPDATE フォームは”Chcek to delete” というチェックボックスを表示します。フィールドがある場合、このラベルの値は `delete_label` 引数を介して設定されます。

- `submit_button` はサブミット・ボタンの値を設定します。
- `id_label` はレコードの”id” のラベルを設定します。
- `showid` が `False` の場合、レコードの”id” は表示されません。

`fields` は表示したいフィールド名のオプション的なリストです。リストが提供されている場合、リスト内のフィールドしか表示されません。例:

```
1 fields = ['name']
```

- `labels` はフィールドラベルの辞書です。辞書のキーはフィールド名で、対応する値はラベルとして表示されるものです。ラベルが提供されていない場合、web2py はラベルをフィールド名から生成します(フィールド名を大文字で書き始めアンダースコアをスペースに置換します)。例:

```
1 labels = {'name': 'Your Full Name:'}
```

- `col3` は第 3 のカラム用の辞書の値です。例:

```

1 col3 = {'name': A('what is this?',
2                   _href='http://www.google.com/search?q=define:name') }
```

- `linkto` と `upload` は、ユーザー定義コントローラへのオプション的な URL です。これにより、フォームで参照フィールドを扱うことが可能になります。これについては、後述の節で詳説します。
- `readonly`。True の場合、読み取り専用のフォームを表示します。
- `comments`。False の場合、`col3` のコメントを表示しません。
- `ignore_rw`。通常は、作成/更新フォームに対して、`writable=True` でマークされたフィールドしか表示されず、読み取り専用フォームに対しては、`readable=True` でマークしたフィールドしか表示されません。`ignore_rw=True` に設定すると、これらの制約は無視され、すべてのフィールドが表示されます。これは主に、appadmin インターフェースにおいて、モデルの意図を覆して、各テーブルのすべてのフィールドを表示するために使用されます。
- `formstyle` フォームを `html` にシリアル化するときに使用されるスタイルを決めます。次の値をとることができます：“table3cols”(デフォルト)、“table2cols”(一行にラベルとコメントを、もう1つの行に入力を表示します)、“ul”(入力フィールドの順序なしリストを作成します)、“divs”(フォームを css フレンドリな `div` で表現します)。`formstyle` はまた、(`record_id`, `field_label`, `field_widget`, `field_comment`) を属性として受け取り、`TR()` オブジェクトを返す関数をとることもできます。
- `buttons` は `INPUT` や `TAG.BUTTON`(技術的にはあらゆるヘルパの組み合わせを使用できるが) のリストで、`submit` ボタンが配置される `DIV` に追加されます。
- `separator` はフォームのラベルと入力フィールドの間に区切り文字を設定します。
- オプション的な `attributes` は、`SQLFORM` オブジェクトをレンダリングする `FORM` タグに対して渡したいアンダースコアで始まる引数群です。たとえば次のようにあります：

```
1 _action = '.'
2 _method = 'POST'
```

特別な `hidden` 属性があります。辞書が `hidden` として渡されたとき、その項目は “hidden” `INPUT` フィールドに変換されます(第5章の `FORM` ヘルパの例を参照してください)。

```
1 form = SQLFORM(...,hidden=...)
```

隠しフィールドはサブミットによって渡されるため、`form.accepts(...)` は受信した隠しフィールドを読み込み、`form.vars` に値を渡すといつと動作は行いません。これはセキュリティ上の理由です。隠しフィールドは改ざんされる可能性があるからです。このため、隠しフィールドを `request` からフォームに明示的に移動する必要があります。

```
1 form.vars.a = request.vars.a
2 form = SQLFORM(..., hidden=dict(a='b'))
```

### 7.2.1 SQLFORM と insert/update/delete

`SQLFORM` はフォームが受理されると新しいレコードを作成します。以下の例を考えてみると、`form=SQLFORM(db.test)` このとき、最後に作成されたレコードの `id` は `form.vars.id` で参照できます。

レコードを `SQLFORM` コンストラクタのオプション的な第 2 の引数に渡した場合、フォームはそのレコードに対する UPDATE フォームになります。つまり、フォームがサブミットされると既存のレコードが更新され新しいレコードは挿入されません。`deletable=True` 属性を設定して場合は、UPDATE フォームは”Check to delete” というチェックボックスを表示します。チェックされると、レコードは削除されます。

フォームがサブミットされ、削除チェックボックスがチェックされている場合は、`form.deleted` 属性に `True` が設定されます。

たとえば、前述の例のコントローラを修正し、次のように、URL のパスに追加の整数引数を渡すことができます。

```
1 /test/default/display_form/2
```

これに対応する `id` を持つレコードがあると、`SQLFORM` は、このレコードのための UPDATE/DELETE フォームを生成します：

```
1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     form = SQLFORM(db.person, record)
4     if form.process().accepted:
5         response.flash = 'form accepted'
6     elif form.errors:
```

```

7     response.flash = 'form has errors'
8     return dict(form=form)

```

2行目でレコードを見つけ、3行目で UPDATE/DELETE フォームを作り、4行目はすべての対応するフォームの処理を行います。

更新フォームは作成フォームにとても似ていますが、現在のレコードによつて事前入力され、プレビュー画像も表示します。デフォルトでは、`deletable = True` になっていて、”*delete record*” オプションが更新フォームに表示されます。

編集フォームはまた、`name="id"` という隠れ INPUT フィールドを保持しています。これによりレコードが特定できるようになります。この `id` はまた、追加のセキュリティのためサーバーサイドで保存され、訪問者がフィールドの値を改ざんした場合、UPDATE は実行されず、web2py は”user is tampering with form” となる SyntaxError を発生させます。

`Field` が `writable=False` としてマークされていると、フィールドは作成フォームに表示されず、読み込み専用の更新フォームで表示されます。フィールドが `writable=False`、かつ、`readable=False` としてマークされている場合、フィールドは、更新フォームを含むすべてのフォーム上で表示されません。

次のように作成されたフォームは、

```
1 form = SQLFORM(..., ignore_rw=True)
```

`readable` と `writable` の属性を無視して、常にすべてのフィールドを表示します。`appadmin` のフォームはデフォルトでそれらを無視します。

次のように作成されたフォームは、

```
1 form = SQLFORM(table, record_id, readonly=True)
```

常に、読み取り専用モードですべてのフィールドを表示し、フォームが受理することはありません。

### 7.2.2 HTML における SQLFORM

SQLFORM のフォームをその生成と処理機能から利便を享受するために使用したいが、SQLFORM オブジェクトのパラメタではできなくらいの HTML のカスタマ

イズが必要で、HTML を用いてフォームを設計しなければならないことがあります。

では、前回のコントローラを編集し新しいアクションを追加してみます：

```

1 def display_manual_form():
2     form = SQLFORM(db.person)
3     if form.process(session=None, formname='test').accepted:
4         response.flash = 'form accepted'
5     elif form.errors:
6         response.flash = 'form has errors'
7     else:
8         response.flash = 'please fill the form'
9     # Note: no form instance is passed to the view
10    return dict()

```

そして、関連付けられたビュー”default/display\_manual\_form.html”にフォームを挿入します：

```

1 {{extend 'layout.html'}}
2 <form>
3 <ul>
4     <li>Your name is <input name="name" /></li>
5 </ul>
6     <input type="submit" />
7     <input type="hidden" name="_formname" value="test" />
8 </form>

```

ここで、アクションはフォームを返していないことに注意してください。なぜならビューに渡す必要がないからです。ビューはHTMLにおいて手動で作成されたフォームを含んでいます。フォームは、隠れフィールド”\_formname”を含んでいて、それはアクションの accepts の引数で指定された formname と必ず同じにしなければなりません。web2py は同じページに複数のフォームがある場合に、どちらがサブミットされたかを判断するのに、フォームの名前を使用します。ページが単一のフォームからなる場合、formname=None と設定して、ビューにおける隠れフィールドを見送ることができます。

form.accepts はデータベースステータブル db.person のフィールドに適合するデータを response.vars 内から検索します。これらのフィールドは以下のように HTML で宣言することができます。<input name="field\_name\_goes\_here" />

上記の例では、フォーム変数が URL の引数として渡される点に注意してください。そうしたくない場合、POST プロトコルが指定される必要があります。さら

に、upload フィールドが指定された場合は、フォーム上でそれを許可するよう設定しなければなりません。両方のオプションを以下に示します。

```
1 <form enctype="multipart/form-data" method="post">
```

### 7.2.3 SQLFORM とアップロード

”upload”型のフィールドは特殊です。それらは、`type="file"` の INPUT フィールドでレンダリングされます。特に指定がない限り、アップロードしたファイルはバッファにストリームされ、自動的に割り当てられる新しい安全な名前を用いて、アプリケーションの”upload” フォルダに保存されます。このファイルの名前はこのとき、アップロード型のフィールドに保存されます。

例として、次のモデルを考えてください：

```
1 db.define_table('person',
2     Field('name', requires=IS_NOT_EMPTY()),
3     Field('image', 'upload'))
```

先ほどのコントローラアクション”display\_form”と同じものを利用することができます。

新規のレコードを挿入するとき、フォームはファイルに対する閲覧を可能にします。たとえば、jpg 画像を選択してください。このファイルはアップロードされ、次のように保存されます：

```
1 applications/test/uploads/person.image.XXXXXX.jpg
```

”XXXXXX”は、web2py によってこのファイルに割り当てられるランダムな識別子になります。

デフォルトでは、アップロードしたファイルの元のファイル名は `b16` エンコードされ、そのファイルに対する新しい名前の構築に使用されることに注意してください。この名前は、デフォルトの”download” アクションによって取り出され、元のファイルへの *Content-Disposition* ヘッダを設定するのに使用されます。

拡張子だけがそのままになります。これはセキュリティ上の理由です。なぜなら、ファイル名は特別な文字列を含む可能性があり、訪問者にディレクトリ・トラバーサル攻撃や他の悪意のある操作を許してしまうからです。

新しいファイル名は `form.vars.image` にも格納されます。

UPDATE フォームを使用してレコードを編集するとき、既存のアップロードしたファイルへのリンクを表示するのは便利で、web2py はその方法を提供しています。

URL を `upload` 引数を介して `SQLFORM` のコンストラクタに渡す場合、web2py は、ファイルをダウンロードするために、その URL のアクションを用います。次のアクションを考えてください：

```

1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     form = SQLFORM(db.person, record, deletable=True,
4                     upload=URL('download'))
5     if form.process().accepted:
6         response.flash = 'form accepted'
7     elif form.errors:
8         response.flash = 'form has errors'
9     return dict(form=form)
10
11 def download():
12     return response.download(request, db)
```

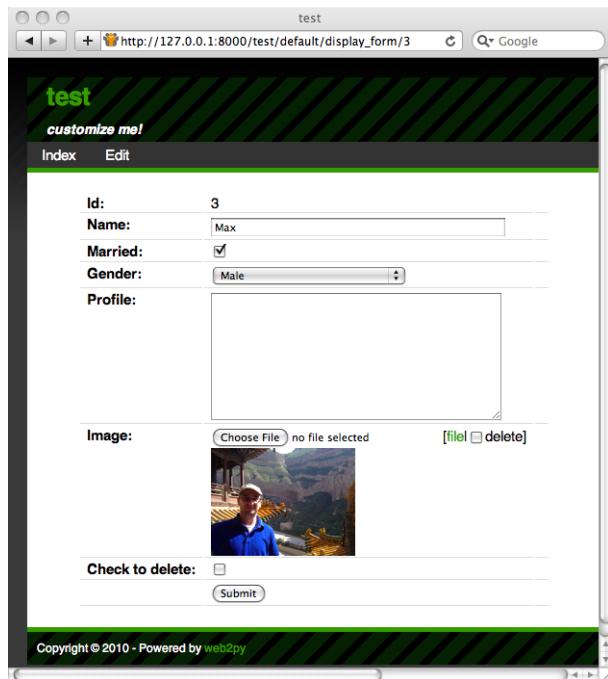
さて、次の URL にて新規のレコードを挿入してみます：

```
1 http://127.0.0.1:8000/test/default/display_form
```

画像をアップロード、フォームをサブミットして、次の URL を訪れて新しく作られたレコードを編集します：

```
1 http://127.0.0.1:8000/test/default/display_form/3
```

(ここでは最新のレコードが `id=3` をもつと仮定します)。フォームは以下に示すように画像のプレビューを表示します：



このフォームは、シリアル化されるときに、次のような HTML を生成します：

```

1 <td><label id="person_image_label" for="person_image">Image: </label>
2 </td>
3 <td><div><input type="file" id="person_image" class="upload" name="image">
4 [<a href="/test/default/download/person.image.0246683463831.jpg">file</a> | 
5 <input type="checkbox" name="image_delete" />delete]</div></td><td></td></tr>
6 <tr id="delete_record_row"><td><label id="delete_record_label" for="delete_record">
7 Check to delete:</label></td><td><td><input type="checkbox" id="delete_record" class="delete" name="delete_this_record" /></td>
```

これは、アップロードしたファイルをダウンロードするリンクと、データベースのレコードからこのファイルを削除するためのチェックボックスを含んでいます。したがって、"image" フィールドには NULL が格納されています。

なぜ、このような機構が公開されているのでしょうか？ なぜ、ダウンロード関

数を書く必要があるのでしょうか？なぜなら、いくつかの認証メカニズムをダウンロード関数に課すことが必要になるかもしれませんからです。例は、第9章を参照してください。

通常アップロードファイルは”app/uploads”の中に保存されますが、別の場所を指定することもできます。

```
1 Field('image', 'upload', uploadfolder='...')
```

多くのオペレーティングシステムにおいて、同一のフォルダ内に大量のファイルがある場合はファイルシステムへの接続が遅くなる場合があります。もし1000以上のファイルをアップロードする予定があるならば、web2pyにサブフォルダでアップロードファイルを整理するように指示できます。

```
1 Field('image', 'upload', uploadseparate=True)
```

#### 7.2.4 元のファイル名の保存

web2pyは自動的に元のファイル名を新しいUUIDのファイル名の中に保存し、ファイルがダウンロードされたときにそれを取り出します。ダウンロードの際、オリジナルのファイル名は、HTTPレスポンスのContent-Dispositionヘッダに格納されます。これはすべて、プログラミングの必要なしに透過的に行われます。

時には、オリジナルのファイル名をデータベースのフィールドに保存したい場合もあります。この場合、モデルを修正し、それを保存するフィールドを加える必要があります：

```
1 db.define_table('person',
2     Field('name', requires=IS_NOT_EMPTY()),
3     Field('image_filename'),
4     Field('image', 'upload'))
```

そして、それを処理するようにコントローラーを修正する必要があります：

```
1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     url = URL('download')
4     form = SQLFORM(db.person, record, deletable=True,
5                   upload=url, fields=['name', 'image'])
6     if request.vars.image!=None:
7         form.vars.image_filename = request.vars.image.filename
8     if form.process().accepted:
```

```

9     response.flash = 'form accepted'
10    elif form.errors:
11        response.flash = 'form has errors'
12    return dict(form=form)

```

SQLFORM は”image\_filename” フィールドを表示していないことに注意してください。”display\_form” アクションは、request.vars.image のファイル名を form.vars.image\_filename に移動します。これにより、accepts においてファイル名を処理し、データベースに保存することができるようになります。ダウンロード関数は、そのファイルを配信する前に、元のファイル名をデータベースにおいてチェックし、Content-Disposition ヘッダにおいて使用します。

### 7.2.5 autodelete

レコードを削除する際に、SQLFORM はレコードによって参照された物理的なアップロード・ファイルを削除することはありません。その理由は、web2py がそのファイルが他のテーブルによって使用/リンクされているか、また、他の目的で使用されているかどうか知ることができないからです。対応するレコードが削除されたとき、実際のファイルを削除しても安全だと判断できる場合は、次のようにして削除できます：

```

1 db.define_table('image',
2     Field('name', requires=IS_NOT_EMPTY()),
3     Field('file', 'upload', autodelete=True))

```

autodelete 属性はデフォルトでは False です。True に設定すると、レコードが削除されるとき、ファイルも削除されるようになります。

### 7.2.6 参照レコードへのリンク

今度は、参照フィールドによってリンクされた 2 つのテーブルを考えます。

```

1 db.define_table('person',
2     Field('name', requires=IS_NOT_EMPTY()))
3 db.define_table('dog',
4     Field('owner', db.person),
5     Field('name', requires=IS_NOT_EMPTY()))
6 db.dog.owner.requires = IS_IN_DB(db, db.person.id, '%(name)s')

```

飼い主は犬を飼い、犬は所有者 (owner)、つまり、飼い主に所属しています。犬の所有者 (owner) には、有効な db.person.id を '%(name)s' を用いて参照することが要求されます。

このアプリケーションの appadmin インターフェースを用いて、何人かの飼い主と彼らの犬を加えましょう。

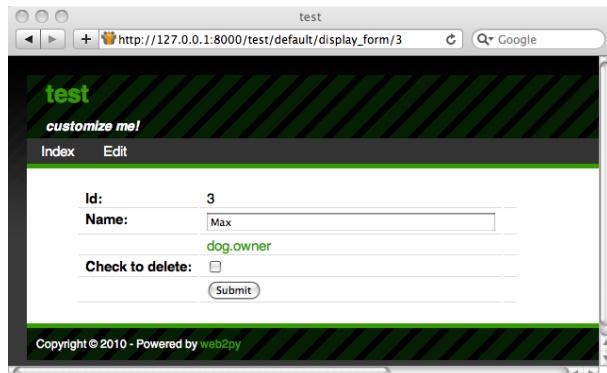
既存の飼い主を編集するとき、appadmin の UPDATE フォームは、この飼い主に属する犬の一覧を表示するページへのリンクを表示します。この挙動は、SQLFORM の linkto 引数を用いて真似することができます。linkto は、SQLFORM からのクエリ文字列を受け取って対応するレコードを返す新規のアクションの URL を指す必要があります。以下がその例です。

```

1 def display_form():
2     record = db.person(request.args(0)) or redirect(URL('index'))
3     url = URL('download')
4     link = URL('list_records', args='db')
5     form = SQLFORM(db.person, record, deletable=True,
6                     upload=url, linkto=link)
7     if form.process().accepted:
8         response.flash = 'form accepted'
9     elif form.errors:
10        response.flash = 'form has errors'
11    return dict(form=form)

```

これがそのページです：



"dog.owner" というリンクがあります。このリンクの名前は、次のような SQLFORM の labels 引数を介して変更することができます：

```

1 labels = { 'dog.owner': "This person's dogs" }

```

リンクをクリックすると、次の場所に向かいます：

```
1 /test/default/list_records/dog?query=dog.owner%3D5
```

”list\_records” は指定されたアクションで、`request.args(0)` に参照するテーブルの名前が設定され、`request.vars.query` に SQL のクエリ文字列が設定されています。URL のクエリ文字列は適切に url エンコードされた”dog.owner=5” の値を含んでいます (web2py は URL を解析するときに自動的にこれをデコードします)。

とても汎用的な”list\_records” アクションを次のように簡単に実装することができます：

```
1 def list_records():
2     table = request.args(0)
3     query = request.vars.query
4     records = db(query).select(db[table].ALL)
5     return dict(records=records)
```

関連付けられるビュー”default/list\_records.html” は次のようにします：

```
1 {{extend 'layout.html'}}
2 {{=records}}
```

選択によってレコードセットが返され、ビューでシリアル化されるとき、これは最初に SQLTABLE オブジェクト (Table と同じではありません) に変換された後、各フィールドがテーブルのカラムと対応する HTML テーブルへとシリアル化されます。

### 7.2.7 フォームの事前入力

次の構文を用いて、フォームを事前入力することは常に可能です：

```
1 form.vars.name = 'fieldvalue'
```

上記のような文は、フィールド (この例では”name”) が明示的にフォームで表示されているかにかかわらず、フォームの宣言の後、かつ、フォームの受理の前に挿入される必要があります。

### 7.2.8 SQLFORM に要素の追加

フォーム作成後に要素を追加したい場合があります。例えば、あなたのウェブサイトの会員規約に同意するかどうかのチェックボックスを追加したい場合です。

```
1 form = SQLFORM(db.yourtable)
2 my_extra_element = TR(LABEL('I agree to the terms and conditions'), \
3                         INPUT(_name='agree', value=True, _type='checkbox'))
4 form[0].insert(-1,my_extra_element)
```

`my_extra_element` 变数はフォームスタイルに適合している必要があります。この例では、デフォルトの `formstyle='table3cols'` を想定しています。

サブミット後、`form.vars.agree` はチェックボックスのステータス値を持ち、`onvalidation` 関数などで使用できます。

### 7.2.9 データベース IO なしの SQLFORM

SQLFORM を使用してデータベーステーブルからフォームを生成し、サブミットしたフォームをそのまま検証するが、データベースにおいて自動的な INSERT/UPDATE/DELETE を行いたくない場合があります。たとえば、1つのフィールドが他の入力フィールドから計算される必要がある場合です。また、挿入されたデータに対して標準の検証では達成できない追加の検証を行いたい場合です。

これは、次のものを：

```
1 form = SQLFORM(db.person)
2 if form.process().accepted:
3     response.flash = 'record inserted'
```

以下のように分解して簡単に行うことができます。

```
1 form = SQLFORM(db.person)
2 if form.validate():
3     ### deal with uploads explicitly
4     form.vars.id = db.person.insert(**dict(form.vars))
5     response.flash = 'record inserted'
```

同じことは UPDATE/DELETE フォームでも行うことができます。次のものを：

```
1 form = SQLFORM(db.person, record)
2 if form.process().accepted:
3     response.flash = 'record updated'
```

以下のように分解します。

```

1 form = SQLFORM(db.person, record)
2 if form.validate():
3     if form.deleted:
4         db(db.person.id==record.id).delete()
5     else:
6         record.update_record(**dict(form.vars))
7     response.flash = 'record updated'

```

”upload”型フィールドを持つテーブルの場合でも、process(dbio=False) と validate() はどちらもアップロードファイルの保存とリネームを、dbio=True のように、つまりデフォルトのシナリオのように、処理します。

アップロードされたファイル名は以下にあります：

```
1 form.vars.fieldname
```

### 7.3 SQLFORM.factory

データベース・テーブルを持っているかのようにフォームを生成したいが、データベース・テーブルはいらないような場面があります。見栄えのよい CSS フレンドリなフォームの生成や、ファイルのアップロードとリネームの実行のために、SQLFORM の能力を単純に活用したい場面です。

これは form\_factory を介して行うことができます。ここに、フォームを生成し、検証を行い、ファイルをアップロードし、すべてを session に保存するような例を示します。

```

1 def form_from_factory():
2     form = SQLFORM.factory(
3         Field('your_name', requires=IS_NOT_EMPTY()),
4         Field('your_image', 'upload'))
5     if form.process().accepted:
6         response.flash = 'form accepted'
7         session.your_name = form.vars.your_name
8         session.filename = form.vars.your_image
9     elif form.errors:
10         response.flash = 'form has errors'
11     return dict(form=form)

```

次にビュー”default/form\_from\_factory.html”を示します：

```

1 {{extend 'layout.html'}}
2 {{=form}}

```

フィールドのラベルにおいてスペースの代わりにアンダースコアを使用するか、SQLFORMで行ったのと同様に、labelsの辞書をform\_factoryに明示的に渡す必要があります。デフォルトでは、SQLFORM.factoryは、あたかも”no\_table”というテーブルから生成されたフォームのように生成されたhtmlの”id”属性を用いてフォームを生成します。このダミーテーブルの名前を変更したいときは、factoryのtable\_name属性を用いてください：

```

1 form = SQLFORM.factory(...,table_name='other_dummy_name')

```

factoryから生成された2つのフォームを同じテーブルに配置する必要があり、かつ、CSSの競合を避けたい場合、table\_nameの変更が必要になります。

### 7.3.1 複数テーブルでひとつのフォーム

例えば、参照によってリンクされた’client’と’address’というふたつのテーブルが存在し、顧客とその住所レコードをひとつのフォームで挿入したいとします。これは以下のようにできます：model:

```

1 db.define_table('client',
2     Field('name'))
3 db.define_table('address',
4     Field('client',db.client,writable=False,readable=False),
5     Field('street'),Field('city'))

```

controller:

```

1 def register():
2     form=SQLFORM.factory(db.client,db.address)
3     if form.process().accepted:
4         id = db.client.insert(**db.client._filter_fields(form.vars))
5         form.vars.client=id
6         id = db.address.insert(**db.address._filter_fields(form.vars))
7         response.flash='Thanks for filling the form'
8     return dict(form=form)

```

SQLFORM.factory(両方のテーブルで公開されたフィールドからひとつのフォームを作成しバリデータも継承している)に注意してください。ひとつのフォームの受理で、あるデータはひとつ目のテーブル、残りは別のテーブルからと、ふたつの挿入を実施しています。

複数テーブル間で共通のフィールド名が存在しない場合のみ動作します。

## 7.4 CRUD

web2py に最近追加されたものの 1 つは、SQLFORM の上にある Create/Read/Update/Delete (CRUD) API です。CRUD は SQLFORM を作成しますが、フォームの作成、フォームの処理、通知、リダイレクトを、すべて 1 つの関数において補完することによって、コーディングを単純化します。

初めに注意する点は、CRUD が他のこれまで使用してきた web2py の API と異なり、API がすでに公開されていないことです。これはインポートしなければなりません。また、特定のデータベースにリンクする必要があります。例：

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

上で定義した crud オブジェクトは次のような API を提供します：

- crud.tables() は、データベースに定義されているテーブルのリストを返します。
- crud.create(db.tablename) は、テーブルの tablename に対する作成フォームを返します。
- crud.read(db.tablename, id) は、tablename とレコード id に対する読み取り専用のフォームを返します。
- crud.update(db.tablename, id) は、tablename とレコード id に対する更新フォームを返します。
- crud.delete(db.tablename, id) は、レコードを削除します。
- crud.select(db.tablename, query) は、テーブルから選択されたレコードのリストを返します。
- crud.search(db.tablename) は、(form, records) のタプルを返します。form は検索フォームで、records はサブミットされた検索フォームに基づくレコードのリストです。
- crud() は、request.args() に基づいて、上記のうちの 1 つを返します。

たとえば、次のアクションは：

```
1 def data(): return dict(form=crud())
```

次のような URL を公開します：

```
1 http://.../[app]/[controller]/data/tables
2 http://.../[app]/[controller]/data/create/[tablename]
3 http://.../[app]/[controller]/data/read/[tablename]/[id]
4 http://.../[app]/[controller]/data/update/[tablename]/[id]
5 http://.../[app]/[controller]/data/delete/[tablename]/[id]
6 http://.../[app]/[controller]/data/select/[tablename]
7 http://.../[app]/[controller]/data/search/[tablename]
```

しかし、次のアクションは：

```
1 def create_tablename():
2     return dict(form=crud.create(db.tablename))
```

以下の作成メソッドしか公開しません

```
1 http://.../[app]/[controller]/create_tablename
```

また、次のアクションは：

```
1 def update_tablename():
2     return dict(form=crud.update(db.tablename, request.args(0)))
```

以下の更新メソッドしか公開しません

```
1 http://.../[app]/[controller]/update_tablename/[id]
```

他も同様です。

CRUD の挙動は二通りの方法でカスタマイズできます。1 つは、`crud` オブジェクトにいくつかの属性を設定することです。もう 1 つは、各メソッドに追加のパラメータを渡すことです。

#### 7.4.1 設定

ここに、現在の CRUD 属性と、そのデフォルトの値と、意味のリスト一式を示します：

すべての crud のフォームに認証をかけます：

```
1 crud.settings.auth = auth
```

利用方法は第 9 章で説明します。

crud オブジェクトを返す data 関数を定義しているコントローラを指定します

```
1 crud.settings.controller = 'default'
```

レコードの”create” が成功した後のリダイレクト先の URL を指定します：

```
1 crud.settings.create_next = URL('index')
```

レコードの”update” が成功した後のリダイレクト先の URL を指定します：

```
1 crud.settings.update_next = URL('index')
```

レコードの”delete” が成功した後のリダイレクト先の URL を指定します：

```
1 crud.settings.delete_next = URL('index')
```

アップロードされたファイルへリンクするために使用する URL を指定します：

```
1 crud.settings.download_url = URL('download')
```

crud.create フォームに対する標準の検証処理の後に実行される追加の関数を指定します：

```
1 crud.settings.create_onvalidation = StorageList()
```

StorageList は Storage オブジェクトと同様で、両者”gluon/storage.py” ファイルに定義されていますが、そのデフォルトは None ではなく [] になります。これにより、次の構文が使用できます：

```
1 crud.settings.create_onvalidation.mytablename.append(lambda form:....)
```

crud.update フォームに対する標準の検証処理の後に実行される追加の関数を指定します：

```
1 crud.settings.update_onvalidation = StorageList()
```

crud.create フォームの完了後に実行される追加の関数を指定します：

```
1 crud.settings.create_onaccept = StorageList()
```

crud.update フォームの完了後に実行される追加の関数を指定します：

```
1 crud.settings.update_onaccept = StorageList()
```

レコードが削除される場合において、`crud.update` の完了後に実行される追加の関数を指定します：

```
1 crud.settings.update_onDelete = StorageList()
```

`crud.delete` の完了後に実行される追加の関数を指定します：

```
1 crud.settings.delete_onaccept = StorageList()
```

”update” フォームが”delete” ボタンを持つかどうかを決めます：

```
1 crud.settings.update_deletable = True
```

”update” フォームが編集レコードの”id” を表示するかどうかを決めます：

```
1 crud.settings.showid = False
```

フォームが前回挿入された値を維持するか、サブミット成功後デフォルトにリセットするかどうかを決めます：

```
1 crud.settings.keepvalues = False
```

`crud` は編集されているレコードがフォーム表示時からサブミットの間に第 3 者によって修正されていないかを検知します。これは以下と等しいです。

```
1 form.process(detect_record_change=True)
```

そしてこのように設定します：

```
1 crud.settings.detect_record_change = True
```

変数の値を `False` にすることで無効にすることができます。

フォームのスタイルは次のようにして変更することができます

```
1 crud.settings.formstyle = 'table3cols' or 'table2cols' or 'divs' or 'ul'
      '
```

全ての `crud` フォームに区切り文字を設定できます。

```
1 crud.settings.label_separator = ':'
```

`auth` で説明されるのと同じやり方で、フォームにキャプチャを加えることができます：

```
1 crud.settings.create_captcha = None  
2 crud.settings.update_captcha = None  
3 crud.settings.captcha = None
```

#### 7.4.2 メッセージ

カスタマイズ可能なメッセージのリストを以下に示します：

```
1 crud.messages.submit_button = 'Submit'
```

これは、create、update フォーム両方に対して”submit”ボタンのテキストを設定します。

```
1 crud.messages.delete_label = 'Check to delete:'
```

これは、”update” フォームにおいて”delete” ボタンのラベルを設定します。

```
1 crud.messages.record_created = 'Record Created'
```

これは、レコードの作成が成功した際の flash メッセージを設定します。

```
1 crud.messages.record_updated = 'Record Updated'
```

これは、レコードの更新が成功した際の flash メッセージを設定します。

```
1 crud.messages.record_deleted = 'Record Deleted'
```

これは、レコードの削除が成功した際の flash メッセージを設定します。

```
1 crud.messages.update_log = 'Record %(id)s updated'
```

これは、レコードの更新が成功したときのログメッセージを設定します。

```
1 crud.messages.create_log = 'Record %(id)s created'
```

これは、レコードの作成が成功したときのログメッセージを設定します。

```
1 crud.messages.read_log = 'Record %(id)s read'
```

これは、レコードの読み取りアクセスが成功したときのログメッセージを設定します。

```
1 crud.messages.delete_log = 'Record %(id)s deleted'
```

これは、レコードの削除が成功したときのログメッセージを設定します。

なお、`crud.messages` は `gluon.storage.Message` クラスに所属しています。これは、`gluon.storage.Storage` に似ていますが、`T` 演算子の必要なしに、自動的にその値を翻訳します。

ログメッセージは、CRUD が第 9 章で説明する Auth に接続された場合のみ使用されます。イベントは、Auth テーブルの”auth\_events” にログとして記録されます。

### 7.4.3 メソッド

CRUD のメソッドの挙動は、呼び出し毎の原則でカスタマイズすることができます。ここにその用法を示します：

```

1 crud.tables()
2 crud.create(table, next, onvalidation, onaccept, log, message)
3 crud.read(table, record)
4 crud.update(table, record, next, onvalidation, onaccept, ondelete, log,
             message, deletable)
5 crud.delete(table, record_id, next, message)
6 crud.select(table, query, fields, orderby, limitby, headers, **attr)
7 crud.search(table, query, queries, query_labels, fields, field_labels,
              zero, showall, chkall)

```

- `table` は、DAL のテーブル、または、テーブル名です。メソッドはその上で動作します。
- `record` と `record_id` は、レコードの id です。メソッドはその上で動作します。
- `next` は、成功後にリダイレクトする先の URL です。URL が部分文字列”[id]” を含む場合、これは、現在作成/更新されたレコードの id によって置換されます。
- `onvalidation` は、SQLFORM(..., onvalidation) と同じ機能を持ちます。
- `onaccept` は、フォームのサブミットが受理された後に呼ばれ、そこで、リダイレクトする前に、動作する関数です。
- `log` はログのメッセージです。CRUD におけるログのメッセージは、`form.vars` の辞書変数を”%(id)s” のように参照します。

- `message` はフォームが受理されたときの flash メッセージです。
- `ondelete` は、”update” フォームを介してレコードが削除されるときに、`onaccept` の場所で呼ばれます。
- `deletable` は、”update” フォームが delete オプションを持つかどうかを決めます。
- `query` は、レコードを選択するために使用するクエリです。
- `fields` は、レコードを選択するために使用するクエリです。
- `orderby` は、選択したレコードの順序を決めます（第 6 章を参照してください）。
- `limitby` は、表示される選択レコードの範囲を決めます（第 6 章を参照してください）。
- `headers` は、テーブルのヘッダの名前からなる辞書です。
- `queries` は、`['equals', 'not equal', 'contains']` のようなリストです。検索フォームにおける使用可能なメソッドを保持します。
- `query_labels` は、`query_labels=dict>equals='Equals'` のような辞書です。検索メソッドに対する名前を与えます。
- `fields` は、検索ウィジェットにおいて列挙されるフィールドのリストです。
- `field_labels` は、フィールド名をラベルにマッピングする辞書です。
- `zero` は、デフォルトでは”choose one” で、検索ウィジェットのドロップダウンのためのデフォルトのオプションとして使用されます。
- `showall` は最初の呼び出し時に `query` で選択された `rows` を返したい場合は `True` を設定しておきます。（1.98.2 以降に追加）
- `chkall` は検索フォームのチェックボックスを全てチェックしたい時に `True` を設定しておきます。（1.98.2 以降に追加）

ここでは、単一のコントローラ関数における使用例を示します：

```

1 ## assuming db.define_table('person', Field('name'))
2 def people():
3     form = crud.create(db.person, next=URL('index'),
4                        message=T("record created"))
5     persons = crud.select(db.person, fields=['name'],

```

```

6     headers={'person.name': 'Name'})
7     return dict(form=form, persons=persons)

```

もう 1 つのとても汎用的なコントローラ関数を示します。これにより、任意のテーブルから任意のレコードを検索、作成、編集することができます。このとき、テーブル名は `request.args(0)` によって渡されます：

```

1 def manage():
2     table=db[request.args(0)]
3     form = crud.update(table, request.args(1))
4     table.id.represent = lambda id, row: \
5         A('edit:', id, _href=URL(args=(request.args(0), id)))
6     search, rows = crud.search(table)
7     return dict(form=form, search=search, rows=rows)

```

なお、`table.id.represent=...` の行は、web2py に対して、id フィールドの表現を変更し、代わりに、自分自身のページへのリンクを表示し、作成ページを更新ページに切り替えるために id を `request.args(1)` として渡すように指示します。

#### 7.4.4 レコードのバージョニング

SQLFORM と CRUD は共にデータベースレコードのバージョニングを行うユーティリティを提供しています：

すべての改訂履歴を必要とするテーブル (`db.mytable`) を持つたいなら、次のようにするだけです：

```

1 form = SQLFORM(db.mytable, myrecord).process(onsuccess=auth.archive)
1 form = crud.update(db.mytable, myrecord, onaccept=auth.archive)

```

`auth.archive` は `db.mytable_archive` という新規のテーブルを定義します（この名前は参照するテーブルの名前に由来します）。そして、更新時に、（更新前の）レコードのコピーを、作成した記録用のテーブルに保存します。そのレコードへの参照も含まれます。

レコードは実際に更新されるので（その前回の状態のみが記録されます）、参照が壊れることはあります。

これはすべて内部で行われます。記録テーブルにアクセスしたいならば、モデルにおいてそれを定義しておく必要があります：

```

1 db.define_table('mytable_archive',
2     Field('current_record', db.mytable),
3     db.mytable)

```

なお、テーブルは `db.mytable` を拡張し（そのすべてのフィールドを含み）、`current_record` へ参照を追加しています。

`auth.archive` は、次のように元のテーブルが timestamp フィールドを持たない限り、保存したレコードのタイムスタンプを取りません。

```

1 db.define_table('mytable',
2     Field('created_on', 'datetime',
3         default=request.now, update=request.now, writable=False),
4     Field('created_by', auth.user,
5         default=auth.user_id, update=auth.user_id, writable=False),

```

これらのフィールドに関して何ら特別なことはなく、好きな名前を付けることが可能です。これらはレコードが記録される前に入力され、各レコードのコピーと共に記録されます。記録テーブルの名前、または/かつ、フィールドの名前は次のように変更することができます：

```

1 db.define_table('myhistory',
2     Field('parent_record', db.mytable),
3     db.mytable)
4 ## ...
5 form = SQLFORM(db.mytable, myrecord)
6 form.process(onsuccess = lambda form:auth.archive(form,
7             archive_table=db.myhistory,
8             current_record='parent_record'))

```

## 7.5 カスタムフォーム

フォームが SQLFORM や SQLFORM.factory、CRUD を利用して作られている場合、それをビューに埋め込む方法は複数あり、複数の度合いのカスタマイズができるようになります。たとえば次のモデルを考えてみます：

```

1 db.define_table('image',
2     Field('name'),
3     Field('file', 'upload'))

```

また、次のアップロード・アクションも考えます：

```

1 def upload_image():
2     return dict(form=crud.create(db.image))

```

最も簡単に、`upload_image`に対するビューにおいてフォームを埋め込む方法は次の通りです：

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

これは標準のテーブル・レイアウトになります。別のレイアウトを使用したい場合、フォームを要素に分解することができます

```
1 {{=form.custom.begin}}
2 Image name: <div>{{=form.custom.widget.name}}</div>
3 Image file: <div>{{=form.custom.widget.file}}</div>
4 Click here to upload: {{=form.custom.submit}}
5 {{=form.custom.end}}
```

ここで、`form.custom.widget[fieldname]` は、そのフィールドに対して適切な ウィジェットにシリアル化されます。フォームがサブミットされてエラーを含む場合、そのエラーは従来通り ウィジェット の下に追加されます。

上記のサンプルフォームは下図のように表示されます。

The screenshot shows a web page with a form. The first field is labeled "Image name:" followed by a text input field. The second field is labeled "Image file:" followed by a file input field containing the text "no file selected". Below these is a button labeled "Choose File". Underneath the file input is a button labeled "Click here to upload:". To the right of this button is another button labeled "Submit".

ただし、同様の結果は次のようにしても得られます：

```
1 crud.settings.formstyle='table2cols'
```

この場合、カスタムフォームを使用していません。他の可能な `formstyle` は、"table3cols" (デフォルト)、"divs"、"ul" です。web2py によってシリアル化された ウィジェット を使用したくない場合は、それを HTML で置き換えることができます。このために有用ないくつかの変数があります：

- `form.custom.label[fieldname]` はフィールドのラベルを含みます。
- `form.custom.comment[fieldname]` はフィールドのコメントを含みます。
- `form.custom.dspval[fieldname]` はフィールドの表示方法に関する `form-type` と `field-type` を含みます。
- `form.custom.inpval[fieldname]` はフィールドの値に関する `form-type` と `field-type` を含みます。

以下に説明する慣例に従うことは重要です。

### 7.5.1 CSS の慣例

SQLFORM、SQLFORM.factory、CRUD によって生成されたフォーム内のタグは、フォームのさらなるカスタマイズに使用することができる厳密な CSS の命名規則に従っています。

”mytable” テーブルと”string” 型の”myfield” フィールドが与えられたとき、次のものによってレンダリングされます。

```
1 SQLFORM.widgets.string.widget
```

これは次のようにになります：

```
1 <input type="text" name="myfield" id="mytable_myfield"
2   class="string" />
```

以下のことに注意してください：

- INPUT タグのクラスはフィールドの型と同じです。これは”web2py\_ajax.html”におけるjQueryのコードが機能するのに非常に重要です。これは、”integer” か”double” のフィールドにおいて数値しか持たないようにし、”time”、”date”、”datetime” のフィールドではポップアップのカレンダーが表示されるようにします。
- id は、クラスの名前とフィールドの名前をアンダースコアで結合したものです。これにより、たとえば `jQuery('#mytable_myfield')` のようにして一意に参照することができ、フィールドのスタイルシートを操作したり、フィールドのイベントに関連付けられたアクション (focus、blur、keyup など) をバインドすることができるようになります。
- name は、想像通り、フィールド名になります。

### 7.5.2 エラーの非表示

場合によっては、自動的なエラー配置を無効にして、フォームのエラーメッセージをデフォルトではないどこか別の場所に表示したいことがあります。これを行うのは簡単です。

- FORM または SQLFORM の場合は、`hideerror=True` を `accepts` メソッドに渡してください。
- CRUD の場合は、`crud.settings.hideerror=True` に設定してください。エラーを表示するビューを変更したくなることもあります（もはや自動的に表示されないので）。

次の例では、エラーをフォームの中ではなく、フォームの上に表示されるようにしています。

```

1 {{if form.errors:}}
2   Your submitted form contains the following errors:
3   <ul>
4     {{for fieldname in form.errors:}}
5       <li>{{=fieldname}} error: {{=form.errors[fieldname]}}</li>
6     {{pass}}
7   </ul>
8   {{form.errors.clear()}}
9 {{pass}}
10 {{=form}}

```

エラーは下図のように表示されます：

Your submitted form contains the following errors:  
 ■ name error: enter a value

Name:	<input type="text"/>
File:	<input type="button" value="Choose File"/> no file selected
<input type="button" value="Submit"/>	

このメカニズムはカスタムフォームでも動作します。

## 7.6 バリデータ

バリデータは入力フィールド（データベース・テーブルから生成されたフォームを含む）を検証するために使用されるクラスです。

FORM とともにバリデータを使用する例です：

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

どのようにテーブルのフィールドに対するバリデータを要求するかの例です：

```
1 db.define_table('person', Field('name'))
2 db.person.name.requires = IS_NOT_EMPTY()
```

バリデータは常にフィールドの `requires` 属性を用いて割り当てられます。フィールドは、單一もしくは複数のバリデータを持つことができます。複数のバリデータはリストの一部になります：

```
1 db.person.name.requires = [IS_NOT_EMPTY(),
2                             IS_NOT_IN_DB(db, 'person.name')]
```

バリデータは FORM 上の `accepts` と `process` 関数や、フォームを含む他の HTML ヘルパーオブジェクトによって呼ばれます。それらは、列挙されている順序で呼ばれます。

あるフィールドに対して明示的にバリデータを呼ぶこともできます。

```
1 db.person.name.validate(value)
```

これは `(value, error)` のタプルを返し、検証する値が無い場合は `error` が `None` を返します。

組み込みのバリデータはオプション引数を取るコンストラクタを持っています：

```
1 IS_NOT_EMPTY(error_message='cannot be empty')
```

`error_message` は、任意のバリデータに対してデフォルトのエラーメッセージをオーバーライドするようにします。

データベース・テーブルに対するバリデータの例です：

```
1 db.person.name.requires = IS_NOT_EMPTY(error_message=T'fill this!')
```

ここで、国際化対応のため `T` という翻訳演算子を使用しています。なお、デフォルトのエラーメッセージは翻訳されません。

`list`:型のフィールドに対して使用されるバリデータは以下ののみとなります。

- `IS_IN_DB(..., multiple=True)`
- `IS_IN_SET(..., multiple=True)`
- `IS_NOT_EMPTY()`
- `IS_LIST_OF(...)`

一番最後のバリデータはリスト中の個別要素に対してバリデータを適用します。

### 7.6.1 バリデータ

IS\_ALPHANUMERIC

このバリデータは、フィールドの値が a ~ z、A ~ Z、0 ~ 9 の範囲にある文字しか含まれていないことをチェックします。

```
1 requires = IS_ALPHANUMERIC(error_message='must be alphanumeric!')
```

IS\_DATE

このバリデータは、指定したフォーマットで有効な日付がフィールドの値に入っていることをチェックします。異なる場所の異なるフォーマットをサポートするために、翻訳演算子を用いてフォーマットを指定するのは良いプラクティスです。

```
1 requires = IS_DATE(format=T('%Y-%m-%d'),  
2 error_message='must be YYYY-MM-DD!')
```

%ディレクティブの詳細な説明は IS\_DATETIME バリデータの項目を参照してください。

IS\_DATE\_IN\_RANGE

前のバリデータと非常に似ていますが、範囲を指定することができます：

```
1 requires = IS_DATE_IN_RANGE(format=T('%Y-%m-%d'),  
2 minimum=datetime.date(2008, 1, 1),  
3 maximum=datetime.date(2009, 12, 31),  
4 error_message='must be YYYY-MM-DD!')
```

%ディレクティブの詳細な説明は IS\_DATETIME バリデータの項目を参照してください。

IS\_DATETIME

このバリデータは、指定したフォーマットで有効な日時がフィールドの値に入っていることをチェックします。異なる場所の異なるフォーマットをサポートするために、翻訳演算子を用いてフォーマットを指定するのは良いプラクティスです。

```
1 requires = IS_DATETIME(format=T('%Y-%m-%d %H:%M:%S'),  
2 error_message='must be YYYY-MM-DD HH:MM:SS!')
```

以下のシンボルをフォーマット文字列に対して使用することができます（シンボルと例となる文字列を示します）：

```

1 %Y  '1963'
2 %y  '63'
3 %d  '28'
4 %m  '08'
5 %b  'Aug'
6 %B  'August'
7 %H  '14'
8 %I  '02'
9 %P  'PM'
10 %M  '30'
11 %S  '59'
```

#### IS\_DATETIME\_IN\_RANGE

前のバリデータと非常に似ていますが、範囲を指定することができます：

```

1 requires = IS_DATETIME_IN_RANGE(format=' %Y-%m-%d %H:%M:%S'),
2                         minimum=datetime.datetime(2008, 1, 1, 10, 30),
3                         maximum=datetime.datetime(2009, 12, 31, 11, 45),
4                         error_message='must be YYYY-MM-DD HH:MM::SS!')
```

%ディレクティブの詳細な説明は IS\_DATETIME バリデータの項目を参照してください。

#### IS\_DECIMAL\_IN\_RANGE

```

1 INPUT(_type='text', _name='name', requires=IS_DECIMAL_IN_RANGE(0, 10,
2 dot=".") )
```

入力を Python の Decimal へと変換します。もしくは、数値が指定した範囲に収まっている場合はエラーを生成します。比較は Python の Decimal の算術で行われます。

最小値と最大値には None を設定することができ、それぞれ。下限なし、上限なしを意味します。

dot 引数はオプションで小数を区切る記号を国際化できます。

#### IS\_EMAIL

フィールドの値が email のアドレスのようになっているかをチェックします。確認のため email を送信することは試みません。

```

1 requires = IS_EMAIL(error_message='invalid email!')
```

IS\_EQUAL\_TO

検証された値が与えられた値(変数にすることもできます)と等しいかチェックします:

```
1 requires = IS_EQUAL_TO(request.vars.password,
2                         error_message='passwords do not match')
```

## IS\_EXPR

最初の引数は、変数に値に関する論理的な表現を保持する文字列です。フィールドの値を、その式が True に評価されるかどうかで検証します。例:

例外が発生しないように、初めに整数であることをチェックしたほうがよいでしょう。

```
1 requires = [IS_INT_IN_RANGE(0, 100), IS_EXPR('value%3==0')]
```

#### IS\_FLOAT\_IN\_RANGE

フィールドの値が範囲内の浮動小数点になっていることをチェックします。次の例では、`0 <= value <= 100` の範囲をチェックしています：

```
1 requires = IS_FLOAT_IN_RANGE(0, 100, dot=".",
2                               error_message='too small or too large!')
```

`dot` 引数はオプションで小数を区切る記号を国際化できます。

### IS INT IN RANGE

フィールドの値が範囲内の整数になっていることをチェックします。次の例では、`0 <= value <= 100` の範囲をチェックしています：

```
1 requires = IS_INT_IN_RANGE(0, 100,  
2                             error_message='too small or too large!')
```

IS\_IN\_SET

フィールドの値がセットに含まれていることをチェックします:

`zero` 引数は省略可能で、デフォルトで選択されたオプション、つまり、`IS_IN_SET` バリデータ自身によって受理されないオプション、のテキストを決めます。“choose one” オプションを望まない場合は、`zero=None` としてください。

`zero` オプションはリビジョン (1.67.1) において導入されました。これは、アプリケーションを壊さないという意味で後方互換性を破りませんでした。しかし、以前は `zero` オプションがなかったので、その挙動は変化しました。

セットの要素は常に文字列でなければなりません。ただし、このバリデータが `IS_INT_IN_RANGE`(値を int に変換) か `IS_FLOAT_RANGE`(値を float に変換) の後に続く場合はその限りではありません。例:

```
1 requires = [IS_INT_IN_RANGE(0, 8), IS_IN_SET([2, 3, 5, 7],  
2     error_message='must be prime and less than 10')]
```

辞書型やタプルのリストを使ってより記述的なドロップダウンリストを作成することもできます。

```
1 ##### Dictionary example:  
2 requires = IS_IN_SET({'A': 'Apple', 'B': 'Banana', 'C': 'Cherry'}, zero=None)  
3 ##### List of tuples example:  
4 requires = IS_IN_SET([('A', 'Apple'), ('B', 'Banana'), ('C', 'Cherry')])
```

### IS\_IN\_SET とタグ付け

`IS_IN_SET` バリデータは `multiple=False` というオプション属性を持ちます。これが `True` に設定されている場合、複数の値を 1 つのフィールドに格納することができます。フィールドの型は、`list:integer` か `list:string` にしてください。`multiple` 参照は、作成と更新フォームにおいて自動的に処理されます。しかし、DAL に対して透過的ではありません。`multiple` フィールドをレンダリングするためには、jQuery の `multiselect` プラグインを使用することを強く勧めます。

`multiple=True` の場合、`IS_IN_SET` は `zero` や他の値を許可します。つまり、何も選択していないフィールドを許可するということになります。`multiple` はそれぞれ選択できる最小と最大（排他的）である `a` と `b` を持つ、フォーム (`a, b`) のタプルを使用することもできます。

### IS\_LENGTH

フィールドの値の長さが与えられた境界の間に収まることをチェックします。テキストとファイルの入力の両方で機能します。

引数は次の通りです :

- maxsize: 最大許容の長さ/サイズ (デフォルトは 255)
- minsize: 最小許容の長さ/サイズ

例 : テキストの文字列が 33 文字よりも短いかをチェックします :

```
1 INPUT(_type='text', _name='name', requires=IS_LENGTH(32))
```

例 : テキストの文字列が 5 文字よりも長いかをチェックします :

```
1 INPUT(_type='password', _name='name', requires=IS_LENGTH(minsize=6))
```

アップロードされたファイルのサイズが 1KB と 1MB の間にあるかをチェックします :

```
1 INPUT(_type='file', _name='name', requires=IS_LENGTH(1048576, 1024))
```

ファイルを除くすべてのフィールドの型に対して、値の長さをチェックします。ファイルの場合は、値は `cookie.FieldStorage` になります。したがって、直感的に予想できる挙動であるファイルのデータ長をチェックすることになります。

#### IS\_LIST\_OF

これは正確にはバリデータではありません。その使用目的は、複数の値を返すフィールドの検証を可能にすることです。フォームが同じ名前の複数のフィールドや複数選択ボックスを含む場合といった稀なケースにおいて使用されます。唯一の引数は、別のバリデータです。別のバリデータをリストの各要素に適用することしかしません。たとえば、次の式はリストの各項目が 0 ~ 10 の範囲にある整数であることをチェックします :

```
1 requires = IS_LIST_OF(IS_INT_IN_RANGE(0, 10))
```

これは、エラーを返すことではなく、エラーメッセージも含まれません。内部のバリデータがエラーの発生を制御します。

#### IS\_LOWER

このバリデータは決してエラーを返しません。単に、値を小文字に変換します。

```
1 requires = IS_LOWER()
```

#### IS\_MATCH

このバリデータは、値を正規表現と照合し、一致してない場合はエラーを返します。米国の郵便番号を検証する使用例を示します：

```
1 requires = IS_MATCH('^\d{5}(-\d{4})?$',  
2 error_message='not a zip code')
```

IPv4 アドレスを検証する使用例です（ただし、IS\_IPV4 バリデータのほうがこの目的のためにはより妥当です）：

```
1 requires = IS_MATCH('^\d{1,3}(\.\d{1,3}){3}$',  
2 error_message='not an IP address')
```

米国の電話番号を検証するための使用例です：

```
1 requires = IS_MATCH('^1?((-)\d{3}-?|(\(\d{3}\)))\d{3}-?\d{4}$',  
2 error_message='not a phone number')
```

Python の正規表現の詳細については、公式の Python のマニュアルを参照してください。

IS\_MATCH はデフォルトで `False` に設定されている `strict` というオプションの引数を受け取ります。`True` が設定されている場合は、最初の文字だけを照合します。

```
1 >>> IS_MATCH('a')('ba')  
2 ('ba', <lazyT 'invalid expression'>) # no pass  
3 >>> IS_MATCH('a', strict=False)('ab')  
4 ('a', None) # pass!
```

IS\_MATCH はデフォルトで `False` に設定されている `search` というオプションの引数を受け取ります。`True` が設定されている場合は、`match` の代わりに正規表現の `search` を利用して文字を検証します。

#### IS\_NOT\_EMPTY

このバリデータは、フィールドの値が空の文字列ではないことをチェックします。

```
1 requires = IS_NOT_EMPTY(error_message='cannot be empty!')
```

#### IS\_TIME

このバリデータは、指定したフォーマットでの有効な時間がフィールドの値に入力されていることをチェックします。

```
1 requires = IS_TIME(error_message='must be HH:MM:SS!')
```

### IS\_URL

次のいずれかに該当する URL 文字列を拒否します：

- 文字列が空または None
- 文字列が URL で許可されていない文字を使用する
- 文字列が HTTP 構文規則のいずれかを破る
- (指定した場合)URL のスキームが 'http' か 'https' でない
- (ホスト名を指定した場合) トップレベルのドメインが存在しない

(これらの規則は、RFC 2616 [68] に基づいています)

この関数は URL の構文をチェックすることしかしません。たとえば、URL が実際の文章を指しているか、語義的に理にかなっているかはチェックしません。この関数は、省略 URL('google.ca' など) の場合、自動的に URL の前に 'http://' を追加します。mode='generic' というパラメータが使用されている場合、この関数の挙動は変化します。このときは、次のいずれかに該当する URL 文字列を拒否します：

- 文字列が空または None
- 文字列が URL で許可されていない文字を使用する
- (指定した場合)URL のスキームが有効でない

(これらの規則は、RFC 2396 [69] に基づいています)

許可されたスキーマのリストは allowed\_schemes パラメータを使用してカスタマイズすることができます。リストから None を取り除いた場合、('http' のようなスキームを欠く) 省略 URL は拒否されます。

デフォルトで先頭に追加されるスキーマは prepend\_scheme パラメータでカスタマイズすることができます。prepend\_scheme を None に設定した場合、先頭への追加は無効になります。それでも、解析のために先頭への追加が必要な URL は受け入れられますが、戻り値は変更されません。

IS\_URL は、RFC3490 [70] で指定されている国際化ドメイン名 (IDN) の標準と互換性があります。その結果、URL には、通常の文字列または Unicode 文字列を指定できます。URL のドメイン・コンポーネント (たとえば、google.ca) が非 US-ASCII 文字を含んでいる場合、ドメインは (RFC3492 [71] で定義され

た)Punycode に変換されます。IS\_URL は標準を少しだけ越えて、非 US-ASCII 文字が URL のパスとクエリのコンポーネントにも提示されることを許可しています。これらの非 US-ASCII 文字はエンコードされます。たとえば、スペースは、' % 20' にエンコードされます。16 進数の Unicode 文字 0x4e86 は'%4e%86' になります。

例：

```
1 requires = IS_URL()
2 requires = IS_URL(mode='generic')
3 requires = IS_URL(allowed_schemes=['https'])
4 requires = IS_URL(prepend_scheme='https')
5 requires = IS_URL(mode='generic',
6                     allowed_schemes=['ftps', 'https'],
7                     prepend_scheme='https')
```

IS\_SLUG

```
1 requires = IS_SLUG(maxlen=80, check=False, error_message='must be slug'
    )
```

check が True に設定されている場合、検証される値が (英数字と繰り返しなしのダッシュのみ許可する) スラグかどうかをチェックします。If check is set to True it check whether the validated value is a slug (allowing only alphanumeric characters and non-repeated dashes).

check が False の場合 (デフォルト)、入力値をスラグに変換します。If check is set to False (default) it converts the input value to a slug.

IS\_STRONG

フィールド (通常はパスワードフィールド) の複雑さの要求を強制します。

例：

```
1 requires = IS_STRONG(min=10, special=2, upper=2)
```

ここで、

- min は値の最小の長さです
- special は要求される特殊文字の最小数です。特殊文字は、次のいずれかなります @#\$%^&\*(){}[]-+
- upper は大文字の最小数です

### IS\_IMAGE

このバリデータは、ファイル入力からアップロードされたファイルが選択した画像のフォーマットの1つで保存されているか、また、与えられた制約内の寸法(幅と高さ)を持っているかどうかをチェックします。

これは、最大ファイルサイズはチェックしていません(そのためにはIS\_LENGTHを使用してください)。何もデータがアップロードされていない場合は、検証エラーを返します。BMP、GIF、JPEG、PNGのファイル形式をサポートしています。Python Imaging Libraryは必要ありません。

コードの一部は参照 [72] から取っています。

次の引数を取ります：

- extensions: 許可する小文字の画像ファイル拡張子を保持する反復可能オブジェクト
- maxsize: 画像の最大の幅と高さを保持する反復可能オブジェクト
- minsize: 画像の最小の幅と高さを保持する反復可能オブジェクト

画像サイズのチェックを回避するには、minsizeとして(-1, -1)を使用してください。

いくつかの例を示します：

- アップロードされたファイルがサポートされている画像フォーマットのいずれかに含まれるかをチェックします：

```
1 requires = IS_IMAGE()
```

- アップロードされたファイルがJPEGまたはPNG形式かをチェックします：

```
1 requires = IS_IMAGE(extensions=('jpeg', 'png'))
```

- アップロードされたファイルが最大200x200ピクセルのサイズのPNGであるかをチェックします：

```
1 requires = IS_IMAGE(extensions=('png'), maxsize=(200, 200))
```

- 注記: requires = IS\_IMAGE()を含むテーブルの編集フォームを表示している場合、ファイルが削除されると検証を通らないので delete チェックボックスが表示されません。delete チェックボックスを表示したい場合は次のバリデータを使用してください。

```
1 requires = IS_EMPTY_OR(IS_IMAGE())
```

### IS\_UPLOAD\_FILENAME

このバリデータは、ファイル入力からアップロードされたファイルの名前と拡張子が与えられた条件に一致するかをチェックします。

どのような方法であれ、これはファイルの型を保証するものではありません。何もデータがアップロードされていない場合は、検証エラーを返します。

引数は次の通りです：

- filename: (ドットの前の) ファイル名の正規表現です。
- extension: (ドットの後の) 拡張子の正規表現です。
- lastdot: どのドットがファイル名/拡張子の区分に使用されるか : `True` の場合、最後のドットとなります(たとえば、”file.tar.gz” は”file.tar”+”gz” に分解されます)。一方 `False` の場合、最初のドットとなります(たとえば、”file.tar.gz” は”file”+”tar.gz” に分解されます)。
- case: 0 は大文字小文字を維持します。1 は文字列を小文字に変換します(デフォルト)。2 は文字列を大文字に変換します。

`dot` が存在しない場合、拡張子は空の文字列に対してチェックされ、ファイル名はすべての文字列に対してチェックされます。

例：

ファイルが `pdf` の拡張子を持つかをチェックします(大文字小文字は区別しません)：

```
1 requires = IS_UPLOAD_FILENAME(extension='pdf')
```

ファイルが `tar.gz` 拡張子を持ち、かつ、`backup` で始まる名前を持つかをチェックします：

```
1 requires = IS_UPLOAD_FILENAME(filename='backup.*', extension='tar.gz',
                                lastdot=False)
```

ファイルが、拡張子を持たず、かつ、名前が `README` に一致するかをチェックします(大文字小文字を区別します)：

```
1 requires = IS_UPLOAD_FILENAME(filename='^README$', extension='^$', case
                                =0)
```

**IS\_IPV4**

このバリデータは、フィールドの値が 10 進数形式の IP バージョン 4 のアドレスかをチェックします。特定の範囲のアドレスに強制するように設定できます。

IPv4 の正規表現は参照 [73] から取っています。引数は以下の通りです。

- `minip` 許容する最下限のアドレス。str(例 : 192.168.0.1) や、反復可能な数字(例 : [192, 168, 0, 1]) や、int(例 : 3232235521) を受け入れます。
- `maxip` 許容する最上限のアドレス。上と同様に受け入れます。

ここにある 3 つの例の値は同じです。アドレスは、次の関数で包含チェックをするために、整数に変換されるからです :

```
1 number = 16777216 * IP[0] + 65536 * IP[1] + 256 * IP[2] + IP[3]
```

例 :

有効な IPv4 のアドレスに対するチェックをします :

```
1 requires = IS_IPV4()
```

有効なプライベートネットワークの IPv4 のアドレスに対するチェックをします :

```
1 requires = IS_IPV4(minip='192.168.0.1', maxip='192.168.255.255')
```

**IS\_LOWER**

このバリデータはエラーを返すことはありません。値を小文字に変換します。

```
1 requires = IS_LOWER()
```

**IS\_UPPER**

このバリデータはエラーを返すことはありません。値を大文字に変換します。

```
1 requires = IS_UPPER()
```

**IS\_NULL\_OR**

現在のバージョンでは使用されておらず、下に記述する `IS_EMPTY_OR` の別名です。

**IS\_EMPTY\_OR**

他の要求を満たしつつフィールドに空の値を許可したい場合があります。たとえば、フィールドは日付だが、空の値にもなりうるという場合です。`IS_EMPTY_OR` バリデータはこれを可能にします：

```
1 requires = IS_EMPTY_OR(IS_DATE())
```

CLEANUP

これはフィルタです。失敗することはありません。単に、[10、13、32~127] のリストに含まれない 10 進数の ASCII コードを持つすべての文字を削除します。

```
1 requires = CLEANUP()
```

CRYPT

これもフィルタです。入力に対して安全なハッシュを実行します。パスワードがデータベースにそのまま渡されるのを防ぐのに使用されます。

```
1 requires = CRYPT()
```

`key` が指定されていない場合、MD5 アルゴリズムが使用されます。`key` が指定されている場合、CRYPT は HMAC アルゴリズムを用います。`key` には、HMAC とともに使用するアルゴリズムを決める接頭辞を含めることも可能です。たとえば、SHA512 は次のようにになります：

```
1 requires = CRYPT(key='sha512:thisisthekey')
```

これは、推奨される構文です。`key` は、使用されるデータベースに関連付けられた一意の文字列でなければなりません。`key` は変更することはできません。`key` を失うと、それ以前にハッシュ化された値は使用できなくなります。

CRYPT バリデータは入力をハッシュ化するので、ある意味特別な処理と言えます。ハッシュ化される前にパスワードフィールドを検証したい場合は、CRYPT をバリデータのリストに入れることで可能ですが、最後に呼ばれるようにするために、リストの一番最後に追加する必要があります。例：

```
1 requires = [IS_STRONG(), CRYPT(key='sha512:thisisthekey')]
```

CRYPT はデフォルト値がゼロである `min_length` 引数を取ることもできます。

## 7.6.2 データベースのバリデータ

`IS_NOT_IN_DB`

次の例を考えます：

```
1 db.define_table('person', Field('name'))
2 db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

これは、新規の person を挿入したとき、彼/彼女の名前がデータベース db のフィールド person.name にすでに存在していないことを要求します。他のバリデータと同様、この要求はフォーム処理のレベルで強制され、データベースレベルではされません。これには次のわずかな可能性があります。2人の訪問者が同時に、同じ person.name を持つレコードを挿入しようとした場合、競合状態を引き起こし、両者のレコードが受け入れられてしまうことです。したがって、データベースに対しても、このフィールドが一意の値を持つということを知らせるほうが安全です：

```
1 db.define_table('person', Field('name', unique=True))
2 db.person.name.requires = IS_NOT_IN_DB(db, 'person.name')
```

このとき、競合状態が発生した場合、データベースは OperationalError を発生させ、2つのうちの1つの挿入が拒否されます。

IS\_NOT\_IN\_DB の最初の引数は、データベース接続か Set にすることができます。後者の場合、Set で定義されたセットのみをチェックするようになります。

例えば次のコードは、10日以内に同じ名前を持つ2人の persons の登録を許可しません：

```
1 import datetime
2 now = datetime.datetime.today()
3 db.define_table('person',
4     Field('name'),
5     Field('registration_stamp', 'datetime', default=now))
6 recent = db(db.person.registration_stamp>now-datetime.timedelta(10))
7 db.person.name.requires = IS_NOT_IN_DB(recent, 'person.name')
```

IS\_IN\_DB

次のテーブルと要求を考えてください：

```
1 db.define_table('person', Field('name', unique=True))
2 db.define_table('dog', Field('name'), Field('owner', db.person))
3 db.dog.owner.requires = IS_IN_DB(db, 'person.id', '%(name)s',
4                                     zero=T('choose one'))
```

これは、dog の挿入/更新/削除フォームのレベルで強制されます。これは、dog.owner が db データベースの person.id フィールドにおいて有効な id に

なっていることを要求します。このバリデータのおかげで、`dog.owner` フィールドはドロップボックスによって表現されます。バリデータの 3 番目の引数は、ドロップボックス内の要素を説明する文字列です。この例では、`person` の`% (id)s` の代わりに、`person` の`% (name)s` を見たいことになります。`% (...)s` は、各レコードに対して括弧内においてフィールドの値によって置き換えられます。

`zero` オプションは `IS_IN_SET` バリデータに対するものと非常によく似た動作をします。

バリデータの最初の引数は `IS_NOT_IN_DB` のようにデータベース接続や DAL セットも使用できます。これはドロップボックスのレコードを制限したい場合などに活用できます。次の例では、コントローラが呼ばれるたびに動的にレコードを制限するように `IS_IN_DB` を使用しています。

```
1 def index():
2     (...)  

3     query = (db.table.field == 'xyz') #in practice 'xyz' would be a  

4         variable  

5     db.table.field.requires=IS_IN_DB(db(query), ...)  

6     form=SQLFORM(...)  

7     if form.process().accepted: ...  

8     (...)
```

フィールドの検証はしたいが、ドロップボックスを表示したくない場合、バリデータをリストの中に置いてください。

```
1 db.dog.owner.requires = [IS_IN_DB(db, 'person.id', '% (name)s')]
```

場合によっては、ドロップボックスは使用したい(上のようにリスト構文を用いたくない)が、追加のバリデータを使用したいときがあります。この目的のために、`IS_IN_DB` バリデータは`_and` という追加の引数をとります。これは、検証値が `IS_IN_DB` の検証を通った場合に適用される他のバリデータのリストを指します。たとえば、`db` 内のすべての `dog` の `owners` において、あるサブセットにはないことを検証するためには次のようにします：

```
1 subset=db(db.person.id>100)
2 db.dog.owner.requires = IS_IN_DB(db, 'person.id', '% (name)s',
3     _and=IS_NOT_IN_DB(subset, 'person.id'))
```

`IS_IN_DB` は `select` の `cache` 引数と同じような `cache` 引数も取ります。

`IS_IN_DB` とタグ付け

`IS_IN_DB` バリデータは、`multiple=False` というオプション属性を持ちます。これが `True` に設定されている場合、複数の値が 1 つのフィールドに保存されます。このフィールドは、第 6 章で説明した `list:reference` 型にする必要があります。そこでは、明示的なタグ付けの例が説明されています。`multiple` の参照は作成と更新フォームにおいて自動的に処理されます。しかし、DAL に対して透過的ではありません。`multiple` フィールドをレンダリングするためには、jQuery の `multiselect` プラグインを使用することを強く勧めます。

### 7.6.3 カスタムバリデータ

すべてのバリデータは、以下のプロトタイプに従っています：

```

1 class sample_validator:
2     def __init__(self, *a, error_message='error'):
3         self.a = a
4         self.e = error_message
5     def __call__(self, value):
6         if validate(value):
7             return (parsed(value), None)
8         return (value, self.e)
9     def formatter(self, value):
10        return format(value)

```

すなわち、値を検証するために呼ばれたとき、バリデータは `(x, y)` というタプルを返します。`y` が `None` の場合、値は検証を通過し、`x` は通過した値を保持します。たとえば、バリデータが値に整数であることを要求する場合、`x` は `int(value)` に変換されます。値が検証を通過しない場合は、`x` は入力値を保持し、`y` は検証の失敗を説明するエラーメッセージを保持します。このエラーメッセージは、値が妥当でないフォームにエラーを報告するために使用されます。

バリデータは `formatter` メソッドも持つことが可能ですが。これは、`__call__` が行うものと逆の変換を行う必要があります。たとえば、`IS_DATE` に対するソースコードを考えてみます：

```

1 class IS_DATE(object):
2     def __init__(self, format='%Y-%m-%d', error_message='must be YYYY-
3                 MM-DD !'):
4         self.format = format
5         self.error_message = error_message
6     def __call__(self, value):
7         try:

```

```

7         y, m, d, hh, mm, ss, t0, t1, t2 = time.strptime(value, str(
8             self.format))
9         value = datetime.date(y, m, d)
10        return (value, None)
11    except:
12        return (value, self.error_message)
13    def formatter(self, value):
14        return value.strftime(str(self.format))

```

成功した場合、`__call__`メソッドはフォームからデータ文字列を読み取り、コンストラクタで指定したフォーマット文字列を用いてそれを `datetime.date` オブジェクトに変換します。`formatter` オブジェクトは、`datetime.date` オブジェクトを受け取り、同じフォーマットを用いてそれを文字列表現に変換します。`formatter` はフォームによって自動的に呼び出されます。しかし、明示的に使用して、オブジェクトを適切な表現に変換することもできます。例:

```

1 >>> db = DAL()
2 >>> db.define_table('atable',
3     Field('birth', 'date', requires=IS_DATE('%m/%d/%Y')))
4 >>> id = db.atable.insert(birth=datetime.date(2008, 1, 1))
5 >>> row = db.atable[id]
6 >>> print db.atable.formatter(row.birth)
7 01/01/2008

```

複数のバリデータが要求（リストに格納）されたとき、それらは順序通りに実行され、1つの出力は入力として次のものへ渡されます。この連鎖は、バリデータのいずれかが失敗したときに中断されます。

反対に、フィールドの `formatter` メソッドを呼ぶとき、複数のバリデータに関連付けられた `formatters` もまた連鎖されますが、逆順になります。

**カスタムバリデータの代わりに** `form.accepts(...)`、  
`form.process(...)`、`form.validate(...)` の要素である `onvalidate` を使用することもできます。

#### 7.6.4 依存関係のバリデータ

通常、バリデータは全てのモデル内で一度だけ設定されます。

フィールドを検証する必要があり、その検証が別のフィールドの値に依存することがあります。これはいろいろな方法で実現できます。モデルで実現する方法とコントローラで実現する方法があります。

例として、ユーザー名と2度のパスワードを尋ねる登録フォームを生成するページを示します。どのフィールドも空にすることはできず、両者のパスワードは一致しなければなりません：

```

1 def index():
2     form = SQLFORM.factory(
3         Field('username', requires=IS_NOT_EMPTY()),
4         Field('password', requires=IS_NOT_EMPTY()),
5         Field('password_again',
6             requires=IS_EQUAL_TO(request.vars.password)))
7     if form.process().accepted:
8         pass # or take some action
9     return dict(form=form)

```

同じメカニズムは、FORM と SQLFORM オブジェクトに適用することができます。

## 7.7 Widgets

以下に利用可能な web2py のウィジェットの一覧を示します：

```

1 SQLFORM.widgets.string.widget
2 SQLFORM.widgets.text.widget
3 SQLFORM.widgets.password.widget
4 SQLFORM.widgets.integer.widget
5 SQLFORM.widgets.double.widget
6 SQLFORM.widgets.time.widget
7 SQLFORM.widgets.date.widget
8 SQLFORM.widgets.datetime.widget
9 SQLFORM.widgets.upload.widget
10 SQLFORM.widgets.boolean.widget
11 SQLFORM.widgets.options.widget
12 SQLFORM.widgets.multiple.widget
13 SQLFORM.widgets.radio.widget
14 SQLFORM.widgets.checkboxes.widget
15 SQLFORM.widgets.autocomplete

```

最初の 10 個は対応するフィールド型のデフォルトになります。”options” ウィジェットは、フィールドの要求が IS\_IN\_SET か IS\_IN\_DB で multiple=False(デフォルトの挙動) のときに使用されます。”multiple” ウィジェットはフィールドの要求が IS\_IN\_SET か IS\_IN\_DB で multiple=True のときに使用されます。”radio” と”checkboxes” ウィジェットはデフォルトでは決して使用されませ

んが、手動で設定することができます。autocomplete ウィジェットは特別で、それ自身のセクションで説明します。

たとえば、textarea で表示される”文字列”フィールドを持つには以下のようにします：

```
1 Field('comment', 'string', widget=SQLFORM.widgets.text.widget)
```

ウィジェットをフィールドに帰納的に割り当てることもできます：

```
1 db.mytable.myfield.widget = SQLFORM.widgets.string.widget
```

ウィジェットは値が指定される必要がある追加の引数を取る場合があります。この場合、lambda を使用できます。

```
1 db.mytable.myfield.widget = lambda field,value: \
2     SQLFORM.widgets.string.widget(field,value,_style='color:blue')
```

ウィジェットはヘルパファクトリで、最初の2つの引数は常に field と value です。他の引数には通常のヘルパの属性である \_style や \_class 等を含みます。特別な引数を取るウィジェットもあります。具体的にいうと、SQLFORM.widgets.radio や SQLFORM.widgets.checkboxes はそれを格納するフォームの formstyle と適合するために、”table”、“ul”、“divs”を指定できる style 引数 (\_style と混同しないでください) を取ることができます。

新しいウィジェットを作成したり、既存のウィジェットを拡張したりすることができます。

SQLFORM.widgets[type] はクラスで、SQLFORM.widgets[type].widget は対応するクラスの静的メンバ関数です。各ウィジェット関数は2つの引数をとります。フィールドオブジェクトと現在のフィールドの値です。これは、ウィジェットの表現を返します。例として、string ウィジェットは次のように再コード化することができます：

```
1 def my_string_widget(field, value):
2     return INPUT(_name=field.name,
3                 _id="%s_%s" % (field._tablename_, field.name),
4                 _class=field.type,
5                 _value=value,
6                 requires=field.requires)
7
8 Field('comment', 'string', widget=my_string_widget)
```

`id` と `class` の値は、本章の後半で説明されている慣例に従う必要があります。ウィジェットは独自のバリデータを持つことが可能ですが、バリデータをフィールドの”`requires`” 属性に関連付け、ウィジェットがそこからそれらを得るようにするのが良いプラクティスです。

### 7.7.1 Autocomplete widget

autocomplete ウィジェットには 2 つの使い道があります：リストから値を受けてフィールドを自動補完するためと、参照フィールドを自動補完するためです（ここで自動補完される文字列は `id` のように実装された参照の表現です）。

最初のケースは簡単です：

```
1 db.define_table('category', Field('name'))
2 db.define_table('product', Field('name'), Field('category'))
3 db.product.category.widget = SQLFORM.widgets.autocomplete(
4     request, db.category.name, limitby=(0,10), min_length=2)
```

ここで、`limitby` は一度に 10 個までの候補しか表示しないようにウィジェットに指示します。`min_length` は、ユーザーが検索ボックスにおいて少なくとも 2 文字をタイプした後のみ、候補を取得する Ajax コールバックを実行するようにウィジェットに指示します。

2 番目のケースはより複雑になります：

```
1 db.define_table('category', Field('name'))
2 db.define_table('product', Field('name'), Field('category'))
3 db.product.category.widget = SQLFORM.widgets.autocomplete(
4     request, db.category.name, id_field=db.category.id)
```

この場合、`id_field` の値は、自動補完される値が `db.category.name` でも、保存される値は対応する `db.category.id` になるようにウィジェットに指示します。オプションのパラメタ `orderby` は、候補をどのようにソートするかをウィジェットに指示します（デフォルトはアルファベット順です）。

このウィジェットは、Ajax を介して動作します。ここで、Ajax コールバックはどこにあるのでしょうか？ いくつかの魔法が、このウィジェットで起こっています。コールバックはウィジェットオブジェクトのメソッドそのものです。どのように公開されているのでしょうか？ web2py において、任意のコード断片は HTTP 例外を発生させることによってレスポンスを生成することができます。

このウィジェットは次の方法でこの可能性を利用しています： ウィジェットは Ajax 呼び出しを最初にウィジェットを生成した URL と同じところに送ります。そして、request.vars において特別なトークンを置きます。 ウィジェットは再びインスタンス化されるはずで、 ウィジェットはそのトークンを見つけ、リクエストに応答する HTTP 例外を発生させます。これらすべては内部で行われ、開発者に対して隠されています。

## 7.8 SQLFORM.grid と SQLFORM.smartgrid (実験的)

複雑な CRUD コントロールを作成する 2 つの高機能なガジェットがあります。レコードのページ送り、表示、検索、ソート、作成、更新、削除を 1 つのガジェットから実行する機能を提供します。

`SQLFORM.grid` の方がシンプルです。ここに使用方法の例を挙げます：

```
1 @auth.requires_login()
2 def manage_users():
3     grid = SQLFORM.grid(db.auth_user)
4     return locals()
```

これは次のページを作成します。

Id	First name	Last name	E-mail			
1	Sotoposo	Codadupo	sacelapa@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
2	Pasatopo	Madutosa	satatoso@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
3	Cepapoco	Masatopa	somapada@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Modusoco	Topacopo	modataso@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Páducodu	Tasadisa	cotoresa@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
6	Tacesada	Tapacoco	tocecoso@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
7	Cosamamo	Copoduta	tosadadu@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
8	Somamadu	Cotacepo	ducopoco@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
9	Cotaposo	Pacesoto	popatoco@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
10	Satadace	Cotasoda	tacopasa@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
11	Dupatapa	Daduduso	comopomati@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
12	Ducemomo	Pomotoso	daramamomo@example.com	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>

`SQLFORM.grid` の最初の引数はテーブル名かクエリです。 `grid` ガジェットはクエリに適合したレコードに対する接続を提供します。`grid` ガジェットの膨大な引数のリストの説明の前に、どのように動作するかを理解する必要があります。

ガジェットは `request.args` を参照し何の動作（表示、検索、作成、更新、削除等）をするか決定します。ガジェットで作成されたそれぞれのボタンは同じ関数（上記の場合は `manage_users`）にリンクされますが、異なる `request.args` を渡します。grid によって作成された URL は全てデフォルトで電子署名され認証されています。これはユーザーがログインしていない場合、一部の機能（作成、更新、削除）が実行できないことを意味します。この制限については緩和することができます。

```
1 def manage_users():
2     grid = SQLFORM.grid(db.auth_user, user_signature=False)
3     return locals()
```

しかし、これは推奨されません。

`LOAD` を利用してコンポーネントとして埋め込みでもしない限り、`grid` はコントローラの関数につきひとつしか使用できないからです。

`grid` を含む関数自体がコマンドライン引数を操作する場合があるので、`grid` はどの引数を `grid` で処理し、どの引数を `grid` 以外で処理するかを把握する必要があります。例えば、これは任意のテーブルを処理できるコードの例です。

```
1 @auth.requires_login()
2 def manage():
3     table = request.args(0)
4     if not table in db.tables(): redirect(URL('error'))
5     grid = SQLFORM.grid(db[table], args=request.args[:1])
6     return locals()
```

`grid` の `args` 引数はどの `request.args` がガジェットに渡されるか若しくは無視されるかを指定します。私たちの例では `request.args[:1]` は処理したいテーブル名を表し、ガジェットではなく、`manage` 関数自体で扱われています。`grid` の完全な用法は次のようになります：

```
1 SQLFORM.grid(query,
2               fields=None,
3               field_id=None,
4               left=None,
5               headers={},
6               orderby=None,
7               searchable=True,
8               sortable=True,
9               deletable=True,
10              editable=True,
11              details=True,
```

```
12         create=True,
13         csv=True,
14         paginate=20,
15         selectable=None,
16         links=None,
17         upload = '<default>',
18         args=[],
19         user_signature = True,
20         maxtextlengths={},
21         maxtextlength=20,
22         onvalidation=None,
23         oncreate=None,
24         onupdate=None,
25         ondelete=None,
26         sorter_icONS=( '[^]', '[v]' ),
27         ui = 'web2py',
28         showbuttonontext=True,
29         search_widget='default',
30         _class="web2py_grid",
31         formname='web2py_grid',
32         ignore_rw = False,
33         formstyle = 'table3cols'):
```

- `fields` はデータベースから取得されるフィールドのリストです。grid ビューにどのフィールドを表示するかを決定するためにも使用されます。
- `field_id` は`db.mytable.id`のように、ID として使用されるテーブルのフィールドである必要があります。
- `headers` は`'tablename.fieldname'` を対応するヘッダーラベルにマッピングする辞書です。
- `left` は`...select(left=...)` によって左外部結合を定義したい場合に利用する追加の記述です。
- `orderby` は `rows` のデフォルトでの表示順序に使用します。
- `searchable,sortable,deletable,details,create` は検索、ソート、削除、詳細表示、新規レコード作成を実行できるかどうかそれぞれ決定します。
- `csv true` の場合は CSV 形式で grid をダウンロードできます。
- `paginate` は各ページの `rows` の最大値を指定します。
- `links` は異なるページにリンクできる新しい項目を表示するのに使用されます。`links` 変数は `dict(header='name', body=lambda row: A(...))` の

リストを取る必要があります。`header` は新しい項目のヘッダーで、`body` は `row` を取得し値を返します。この例でいうとその値は `A(...)` ヘルパになります。

- `maxtextlength` は grid ビュー上で表示される各フィールドの文字の最大長を設定します。この値は`'tablename.fieldname':length` の辞書として `maxtextlengths` を利用し上書きすることができます。
- `onvalidation, oncreate, onupdate, ondelete` はコールバック関数です。`onDelete` はフォームオブジェクトを入力値として受け取ります。
- `sorter_icons` は各フィールドの昇順、降順ソートオプションを表示するふたつの文字（またはヘルパ）のリストです。
- `ui` は`'web2py'` と同様に web2py 形式のクラス名を設定します。`jquery-ui` は jquery UI 形式のクラス名を設定しますが、様々な grid コンポーネントにおいて独自のクラスも設定されます。

```

1 ui = dict(widget='',
2             header='',
3             content='',
4             default='',
5             cornerall='',
6             cornertop='',
7             cornerbottom='',
8             button='button',
9             buttontext='buttontext button',
10            buttonadd='icon plus',
11            buttonback='icon leftarrow',
12            buttonexport='icon downarrow',
13            buttondelete='icon trash',
14            buttonedit='icon pen',
15            buttontable='icon rightarrow',
16            buttonview='icon magnifier')
```

- `search_widget` はデフォルトの検索ウィジェットを上書きすることができます。詳細は”gluon/sqlhtml.py” のソースコードを参照してください。
- `showbutton` 全てのボタンを非表示にできます。
- `_class` は grid コンテナのクラスです。
- `formname, ignore_rw, formstyle` は作成/更新フォーム用の grid で使用される SQLFORM オブジェクトに渡されます。

`deletable`, `editable`, `details` は一般的にブーリアンの値ですが、`row` オブジェクトを取得し対応するボタンを表示・非表示するか決定するといった関数も使用できます。

`SQLFORM.smartgrid` は `grid` に非常によく似ています。実際に `grid` を含みますが、クエリではなくひとつのテーブルを入力として受け取るように設計されています。そして参照先のテーブルも表示します。

以下のテーブル構造を考えてみましょう:

```
1 db.define_table('parent', Field('name'))
2 db.define_table('child', Field('name'), Field('parent', 'reference parent'))
```

`SQLFORM.grid` で全ての親を一覧表示できます:

```
1 SQLFORM.grid(db.parent)
```

全ての子供:

```
1 SQLFORM.grid(db.child)
```

全ての親と子供をひとつのテーブル:

```
1 SQLFORM.grid(db.parent, left=db.child.on(db.child.parent==db.parent.id))
```

`SQLFORM.smartgrid` で全てのデータをまとめたひとつのガジェットを作成し、両方のテーブルを出力できます。

```
1 @auth.requires_login():
2 def manage():
3     grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
4     return locals()
```

以下のようになります:

Id	Name	Children	View	Edit	Delete
1	Dumomapo	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
2	Sasaceso	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
3	Dudasoma	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
4	Tasdudua	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
5	Dutomato	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
6	Cocoduda	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
7	Satosomo	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
8	Codupoco	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
9	Daducedu	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>
10	Momotodu	<a href="#">Children</a>	<a href="#">View</a>	<a href="#">Edit</a>	<a href="#">Delete</a>

”children” リンクが追加されています。通常の grid を利用して追加の links を作成することができますがその場合は異なる機能を示しています。smartgrid を利用することで自動的に作成され同じガジェットで処理されます。

また、ある親の”children” リンクをクリックするとその親に紐づく（これは明らかですが）子供だけのリストが取得できます。しかし、新しい子供を追加しようとすると、その子供の親の値は選択された親（ガジェットに関連するパンくずリストで表示）から自動的に設定されます。そのフィールドの親の値は上書きくこともできます。読み取り専用にすることで上書きを防げます。

```

1 @auth.requires_login():
2 def manage():
3     db.child.parent.writable = False
4     grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
5     return locals()

```

linked\_tables 引数が指定されていない場合は全ての参照テーブルが自動的にリンクされます。どちらにせよ、誤ってデータが公開されないように、リンクされるべきテーブルの一覧を明示的に指定することを推奨します。

次のコードはシステム内の全てのテーブルの非常に強力な管理インターフェースを作成します。

```

1 @auth.requires_membership('managers'):
2 def manage():
3     table = request.args(0) or 'auth_user'
4     if not table in db.tables(): redirect(URL('error'))
5     grid = SQLFORM.smartgrid(db[table], args=request.args[:1])
6     return locals()

```

`smartgrid` は `grid` と同じ引数を取り、条件付で追加の引数も取ります。

- 最初の引数はテーブルで、クエリではありません
- 'tablename':query の辞書である `constraints` という追加の引数があります。これは'tablename'grid で表示されるレコードに対してさらなるアクセス制限をかけることができます。
- `smartgrid` から接続できるテーブルの名称リストである `linked_tables` という追加の引数があります。
- 以下で説明するようにテーブル、`args`、`linked_tables`、`user_signatures` に辞書型を使うことができます。

前回の `grid` を考えて見ましょう:

```
1 grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'])
```

`db.parent` と `db.child` の両方に接続できます。ナビゲーションコントロールにとって、それぞれのテーブルは、スマートテーブルではなくただの `grid` です。この場合、これはひとつの `smartgrid` が親と子供の `grid` をひとつずつ作成できることを意味します。これらの `grid` に異なるパラメタのセットを渡すこともできます。例えば異なる `searchable` パラメタのセットです。`grid` ではブーリアン型を渡せます:

```
1 grid = SQLFORM.grid(db.parent, searchable=True)
```

`smartgrid` ではブーリアン型の辞書を渡せます:

```
1 grid = SQLFORM.smartgrid(db.parent, linked_tables=['child'],
2                           searchable= dict(parent=True, child=False))
```

このように親は検索可能だが子供は検索不可（検索ウィジェットが必要な場合はあまり多くないです）にできます。

`grid` と `smartgrid` は今後も残りますが実験的としています。これは新たな機能追加があった場合に、返される実際の `html` レイアウトやパラメータのセットが変更される可能性があるからです。

`grid` と `smartgrid` は crud のように自動でアクセス権を強制しませんが、`auth` と統合して明示的にパーミッションを確認することができます:

```
1 grid = SQLFORM.grid(db.auth_user,
2                      editable = auth.has_membership('managers'),
3                      deletable = auth.has_membership('managers'))
```

または

```
1 grid = SQLFORM.grid(db.auth_user,
2     editable = auth.has_permission('edit', 'auth_user'),
3     deletable = auth.has_permission('delete', 'auth_user'))
```

`smartgrid` は単数形と複数形の両方のテーブル名を表示する唯一の web2py の ガジェットです。例えば `parent` は一人の”Child” や、たくさんの”Children” を持てます。それゆえ、テーブルオブジェクトは自身の単数形と複数形の名称を知る必要があります。web2py は通常これを予測しますが明示的に設定することもできます:

```
1 db.define_table('child', ..., singular="Child", plural="Children")
```

または:

```
1 db.define_table('child', ...)
2 db.child._singular = "Child"
3 db.child._plural = "Children"
```

演算子を使用して国際化対応することもできます。

そして複数形と単数形の値は `smartgrid` で使用されるヘッダーとリンクの正しい名前として提供されます；。

第3版 - 翻訳: 細田謙二 レビュー: Omi Chiba

第4版 - 翻訳: Omi Chiba レビュー: Fumito Mizuno

# 8

## EmailとSMS

### 8.1 メールの設定

web2py は、 web2py を使用して簡単にメール送信を行うために、 gluon.tools.Mail クラスを提供しています。メールは次のように定義できます。

```
1 from gluon.tools import Mail
2 mail = Mail()
3 mail.settings.server = 'smtp.example.com:25'
4 mail.settings.sender = 'you@example.com'
5 mail.settings.login = 'username:password'
```

アプリケーションが Auth を使用している場合 (Authについて次章で取り上げます) auth オブジェクトは専用メール auth.settings.mailer を持っています。このため代わりに、次のように定義することも可能です。

```
1 mail = auth.settings.mailer
2 mail.settings.server = 'smtp.example.com:25'
3 mail.settings.sender = 'you@example.com'
4 mail.settings.login = 'username:password'
```

使用するSMTPサーバ用に、mail.settingsで適切なパラメータ値を設定します。SMTPサーバが認証を必要としない場合は、mail.settings.login=Falseと設定します。

デバッグ用には次のように設定します。

```
1 mail.settings.server = 'logging'
```

この場合メールは送信しません。その代わりコンソールにログが出力されます。

### 8.1.1 Google App Engine でのメール設定

Google App Engine でのメール送信設定は次のようにになります。

```
1 mail.settings.server = 'gae'
```

現時点での web2py は、Google App Engine 上での添付ファイルと暗号化メールはサポートしていません。

### 8.1.2 x509 と PGP 暗号化

x509 (SMIME) 暗号化メールを送信するには、次のように設定します。

```
1 mail.settings.cipher_type = 'x509'
2 mail.settings.sign = True
3 mail.settings.sign_passphrase = 'your passphrase'
4 mail.settings.encrypt = True
5 mail.settings.x509_sign_keyfile = 'filename.key'
6 mail.settings.x509_sign_certfile = 'filename.cert'
7 mail.settings.x509_crypt_certfiles = 'filename.cert'
```

PGP 暗号化メールを送信するには、次のように設定します。

```
1 from gpgme import pgp
2 mail.settings.cipher_type = 'gpg'
3 mail.settings.sign = True
4 mail.settings.sign_passphrase = 'your passphrase'
5 mail.settings.encrypt = True
```

後者の設定 (PGP 暗号化メール送信) では、python-pyme パッケージが必要です。

## 8.2 メール送信

mail の設定が完了している場合、次のようにメールの送信が可能です。

```

1 mail.send(to=['somebody@example.com'],
2           subject='hello',
3           # If reply_to is omitted, then mail.settings.sender is used
4           reply_to='us@example.com',
5           message='hi there')

```

メール送信が成功すると Mail は True を返します。そうでなければ、False を返します。 mail.send() の全ての引数は次の通りです。

```

1 send(self, to, subject='None', message='None', attachments=1,
2      cc=1, bcc=1, reply_to=1, encoding='utf-8', headers={})

```

to cc そして bcc はメールアドレスのリスト値を指定します。

headers は送信するメールのヘッダー情報を、次のように辞書型で指定します。

```

1 headers = {'Return-Path' : 'bounces@example.org'}

```

以下、 mail.send() の追加の使用例です。

### 8.2.1 テキストメール

```

1 mail.send('you@example.com',
2           'Message subject',
3           'Plain text body of the message')

```

### 8.2.2 HTML メール

```

1 mail.send('you@example.com',
2           'Message subject',
3           '<html>html body</html>')

```

メール本文が <html> で始まり </html> で終わった場合、HTML メールとして送信します。

### 8.2.3 テキストと HTML の混在メール

メール本文をタプルで指定できます（テキスト、html）

```

1 mail.send('you@example.com',
2           'Message subject',
3           ('Plain text body', '<html>html body</html>'))

```

### 8.2.4 cc と bcc を使用したメール

```

1 mail.send('you@example.com',
2   'Message subject',
3   'Plain text body',
4   cc=['other1@example.com', 'other2@example.com'],
5   bcc=['other3@example.com', 'other4@example.com'])

```

### 8.2.5 添付ファイル

```

1 mail.send('you@example.com',
2   'Message subject',
3   '<html></html>',
4   attachments = Mail.Attachment('/path/to/photo.jpg', content_id='photo'
      '))

```

### 8.2.6 複数の添付ファイル

```

1 mail.send('you@example.com',
2   'Message subject',
3   'Message body',
4   attachments = [Mail.Attachment('/path/to/fist.file'),
5                  Mail.Attachment('/path/to/second.file')])

```

## 8.3 SMS メッセージの送信

web2py アプリケーションから SMS メッセージを送るには、受信者にメッセージを中継することのできるサードパーティのサービスが必要です。通常はこのサービスは無料ではありません。しかしこれは国によって違います。幾つかのサービスを試してみましたが、ほとんど成功しませんでした。電話会社は結局スパムと判断し、これらのサービスからのオリジナルのメールをブロックしました。

より良い方法は、SMS を中継する電話会社のサービスを使用することです。各電話会社が独自に携帯電話番号に関連付けられたメールアドレスを持っているので、SMS メッセージを電話番号へ、メールとして送信することができます。web2py には、この処理を支援するモジュールが付属しています。

```

1 from gluon.contrib.sms_utils import SMSCODES, sms_email
2 email = sms_email('1 (111) 111-1111', 'T-Mobile USA (tmail)')
3 mail.sent(to=email, subject='test', message='test')

```

SMSCODES は、主要な電話会社のメールアドレスの後に付加する名前のマッピング辞書です。 `sms_email` 関数は、電話番号（文字列）と電話会社の名前を受け取り、携帯電話のメールアドレスを返します。

#### 8.4 メッセージ作成で使用するテンプレートシステム

メール作成にテンプレートシステムを使用することが可能です。例えば、次のようなデータベーステーブルを使用します。

```
1 db.define_table('person', Field('name'))
```

データベースに登録されている人に、ビュー・ファイル "message.html" に保存した次のメッセージを送信します。

```
1 Dear {{=person.name}},
2 You have won the second prize, a set of steak knives.
```

次のように実行します。

```
1 for person in db(db.person).select():
2     context = dict(person=person)
3     message = response.render('message.html', context)
4     mail.send(to=['who@example.com'],
5               subject='None',
6               message=message)
```

大抵の処理は、次の命令で行われます。

```
1 response.render('message.html', context)
```

"context" 辞書に定義されている変数を使用し、ビュー "message.html" をレンダリングします。そしてレンダリングしたメールテキストを文字列で返します。context は、テンプレートファイルで表示する変数を含んだ辞書オブジェクトです。

メッセージが `<html>` で始まり `</html>` で終わる場合、HTML メールになります。

HTML メールにウェブサイトのリンクを入れる時は、`URL` 関数を利用できます。しかしデフォルトの `URL` 関数が生成する相対 URL は、メールからは動作しません。絶対 URL を生成するには、`URL` 関数の引数 `scheme` と `host` を指定することが必要です。この設定例です。

```
1 <a href="{=URL(..., scheme=True, host=True)}">Click here</a>
```

もしくは

```
1 <a href="{=URL(..., scheme='http', host='www.site.com')}>Click here
   </a>
```

SMS メッセージや他のテンプレートベースのメッセージ生成でも、メールテキスト生成で使われるメカニズムと同じものが使用されています。

## 8.5 バックグラウンドタスクを使用したメッセージ送信

接続の可能性があるリモート SMTP サーバへのログインと通信処理のため、メールメッセージ送信の動作に数秒かかることがあります。この間、ユーザは処理が完了するまで待機状態になるため、バックグラウンドタスクによって、メールを後で送信するキューを使う方が望ましい場合があります。4章での説明にのよう、手製のタスクキューの設定や、web2py スケジューラを使用することが可能です。ここではタスクキューを使用した例を紹介します。

最初に、メールキューを格納するデータベースモデルを、アプリケーションのモデルファイルに定義します。

```
1 db.define_table('queue',
2     Field('status'),
3     Field('email'),
4     Field('subject'),
5     Field('message'))
```

コントローラからメッセージを送信するため、次のようにキューに入れます。

```
1 db.queue.insert(status='pending',
2                  email='you@example.com',
3                  subject='test',
4                  message='test')
```

次に、バックグラウンドプロセスのスクリプトで、キューを読むと共にメール送信を行うことが必要です。

```
1 ## in file /app/private/mail_queue.py
2 import time
3 while True:
4     rows = db(db.queue.status=='pending').select()
```

```

5      for row in rows:
6          if mail.send(to=row.email,
7              subject=row.subject,
8              message=row.message):
9              row.update_record(status='sent')
10         else:
11             row.update_record(status='failed')
12         db.commit()
13     time.sleep(60) # check every minute

```

最後に、第4章で説明したように、アプリケーションの環境下で mail\_queue.py スクリプトを実行する必要があります

```
1 python web2py.py -S app -M -N -R applications/app/private/mail_queue.py
```

ここで、-S app は web2py の”app”環境下で”mail\_queue.py”を動かす、-M は web2py のモデル定義を読み込む、-N は web2py のクーロンを動かさない、という意味です。mail\_queue.py”で参照している mail オブジェクトは、モデルファイルで定義しています。これは -M オプションにより、”mail\_queue.py”スクリプトで参照可能になる、ということが前提になっています。さらにまた、他のプロセスに対してデータベースロックの影響を及ぼさないよう、できるだけ早く変更をコミットすることも重要です。

第4章でも説明しましたが、このタイプのバックグラウンドプロセスはクーロンでは動かすべきではありません（@reboot クーロンは除きます）。理由は、同時に複数のインスタンスが実行されていないことを確認する必要があるからです。

バックグラウンドプロセスでメール送信することの欠点は、メール送信が失敗した場合にユーザへのフィードバックが困難なことです。メールをコントローラ機能から直接送信した場合は、各種エラーを受け取り、すぐにユーザに返すことが可能です。バックグラウンドプロセスを使用した場合は、コントローラ機能がレスポンスを返した後に、メールは非同期で送信されます。これによりエラーをユーザに知らせるのは、より複雑になります。

第3版 - 翻訳: 中垣健志 レビュー: 細田謙二

第4版 - 翻訳: Hitoshi Kato レビュー: Mitsuhiro Tsuda



# 9

## アクセス制御

web2py には、パワフルでカスタマイズ可能なロールベースのアクセス制御メカニズム (RBAC) が含まれています。

Wikipedia での定義を以下に示します：

「ロールベースアクセス制御 (英: Role-based access control, RBAC) は、権限のあるユーザーに対してシステムアクセスを制限する手法の一種。強制アクセス制御 (MAC) や任意アクセス制御 (DAC) に対するより新しい代替手法である。RBAC はロールベースのセキュリティとして参照されることもある。」

RBAC は、ポリシーが中立で、柔軟にアクセス制御できる技術であり、DAC や MAC もシミュレート可能である。逆に MAC は、ロールの構造が半順序ロール階層 (Partially ordered role Hierarchy) ではなく、ツリーに制限されている場合に限り、RBAC をシミュレートできる。

RBAC が開発されるまで、アクセス制御のモデルとして知られていたのは MAC と DAC しかなかった。従って、モデルが MAC でないなら DAC を、DAC でないなら MAC であると考えられていた。90 年代後半の研究では、RBAC はそのどちらにも分類されていない。

組織において、ロール (役割) は仕事上の機能のために作られる。ある操作を実行する許可 (パーミッション) は、特定のロールに対して割り当てられる。従業員 (またはシステムユーザー) には特定のロールが割り当てられ、そのロールの割り当てを通して特定のシステム機能を実行するパーミッションが与えられる。コンテキストベースアクセス制御 (CBAC) とは異なり、RBAC はメッセージの

コンテキスト（接続がどこから起動されているかなど）を考慮しない。ユーザーに対して直接パーミッションが与えられるわけではなく、ロールを通して与えられるため、各人のアクセス権の管理はユーザーへのロールの適切な割り当てに単純化される。つまり、これによりユーザーの追加やユーザーの部門の異動などの共通の操作が単純化される。

RBAC は従来の任意アクセス制御システムで用いられているアクセス制御リスト (ACL) とも異なる。ACL は低レベルのデータオブジェクトに対してパーミッションを与えるが、RBAC は組織において意味のある特定の操作に対してパーミッションを与える。例えば、一つのアクセス制御リストは特定のシステムファイルへの書き込みを許可/拒否するために用いられるが、そのファイルがどのように変更できるかは規定しない。」web2py には、RBAC を実装した Auth クラスがあります。

Auth は下記のテーブルが必要です（そして定義を行います）。

- `auth_user` ユーザーの名前、メールアドレス、パスワード、ステータス（登録、保留、許可、禁止）が保存される
- `auth_group` 多対多の構造においてユーザーに対するグループまたはロールが保存される。デフォルトでは各ユーザーは自分自身のグループに属している。しかし、1人のユーザーを複数のグループに所属させることも、各グループに複数のユーザーを含めることもできる。グループは、ロールと説明文で識別される
- `auth_membership` 多対多の構造においてユーザーとグループを結ぶ
- `auth_permission` グループとパーミッションを関連づける。パーミッションは名前（テーブルとレコードが含まれることもある）で識別される。例えば、特定のグループのメンバーは、特定のテーブルの特定のレコードの”更新”のパーミッションが与えられる
- `auth_event` これ以外の他のテーブルに対する変更や、RBAC により制御されたオブジェクトへの CRUD を介して成立したアクセスを記録する

原則として、ロールの名前とパーミッションの名前に制限はありません。つまり、開発者は組織内のロールやパーミッションに合わせてそれらを作成することができます。一度これらのテーブルが作成されると、web2py は、ユーザーがログインしているかどうか、グループに属しているかどうか、および／または必要

なパーミッションを持つグループの一つに属しているかどうかを確認する API を提供します。web2py は、ログイン、メンバシップ、パーミッションに基づいて任意の関数へのアクセスを制限するデコレータも提供しています。web2py はまた、いくつかの特別なパーミッション、すなわち、CRUD メソッド(削除、読み込み、更新、作成)に対応した名前を持つパーミッションに対応しており、デコレータを使用しなくても自動的にそのパーミッションを強制することができます。

この章では、RBAC のそれぞれの構成要素を一つ一つ解説していきます。

## 9.1 認証

RBAC を使用するためには、ユーザーが識別されている必要があります。これは、ユーザーが登録を行い(あるいは事前に登録されている状態で)、ログインする必要があることを意味しています。

Auth は複数のログイン用メソッドを提供します。デフォルトのメソッドは、ローカルの auth\_user テーブルに基づいてユーザーを識別します。代わりに、サードパーティの認証システムやシングルサインオンを提供するプロバイダ(Google、PAM、LDAP、Facebook、LinkedIn、Dropbox、OpenID、OAuth、その他)に対してユーザーをログインさせることができます。

Auth の使用を開始するには、少なくとも以下のコードを model ファイルに用意する必要があります。このコードはまた、web2py の”welcome” アプリケーションで提供され、db 接続オブジェクトを利用しています：

```
1 from gluon.tools import Auth
2 auth = Auth(db, hmac_key=Auth.get_or_create_key())
3 auth.define_tables()
```

db.auth\_user の password フィールドには CRYPT バリデータがデフォルト指定されており、それは hmac\_key と共に必要な設定です。Auth.get\_or\_create\_key() はアプリケーションフォルダ内の ”private/auth.key” ファイルから hmac キーを読み出す機能です。もしファイルが存在しない場合は、ランダムな hmac\_key を生成します。また複数のアプリケーションで同じ auth(アクセス制御) データベースを共有する場合は、同一の hmac\_key が使われているか確認してください。

デフォルトでは web2py はログインに email(メールアドレス) を使用します。もし代わりに username(ユーザ名)を使用したい場合は、auth.define\_tables(username=True) と設定します。

もし複数のアプリケーションで auth データベースを共有する場合は、auth.define\_tables(username=True) のようにマイグレーションを無効に設定してください。

Auth を公開するには、次のような関数をコントローラに用意する必要があります ("default.py" にサンプルがあります)。

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

auth オブジェクトと user 関数の両方は離形アプリケーションすでに定義されています。

web2py にはまた、適切にこの関数をレンダリングするための次のようなサンプルビュー "welcome/views/default/user.html" が含まれています：

```
1 {{extend 'layout.html'}}
2 <h2>{{=T( request.args(0).replace('_', ' ').capitalize() )}}</h2>
3 <div id="web2py_user_form">
4     {{=form}}
5     {{if request.args(0)=='login':}}
6         {{if not 'register' in auth.settings.actions_disabled:}}
7             <br/><a href="{{=URL(args='register')}}">register</a>
8         {{pass}}
9         {{if not 'request_reset_password' in auth.settings.actions_disabled:}}
10            :}}
11            <br/>
12            <a href="{{=URL(args='request_reset_password')}}">lost password</
13                a>
14        {{pass}}
15    {{pass}}
16 </div>
```

この関数は form を単純に表示しています。ですので、標準のフォームカスタマイズ用の構文を使ってカスタマイズすることができます。唯一の注意点は form=auth() によって表示されているフォームが request.args(0) に依存していることです。つまりデフォルトの auth() ログインフォームをカスタムログインフォームに置き換える場合、以下のビューのように if 文を利用する必要があります。

```
1 {{if request.args(0)=='login':}}...custom login form...{{pass}}
```

上記のコントローラは、複数の機能を公開しています。

```
1 http://.../[app]/default/user/register
2 http://.../[app]/default/user/login
3 http://.../[app]/default/user/logout
4 http://.../[app]/default/user/profile
5 http://.../[app]/default/user/change_password
6 http://.../[app]/default/user/verify_email
7 http://.../[app]/default/user/retrieve_username
8 http://.../[app]/default/user/request_reset_password
9 http://.../[app]/default/user/reset_password
10 http://.../[app]/default/user/impersonate
11 http://.../[app]/default/user/groups
12 http://.../[app]/default/user/not_authorized
```

- register はユーザーの登録を行います。CAPTCHA も統合されていますが、デフォルトでは無効になっています。
- login は事前に登録されたユーザーのログインを許可します（検証が通るか必要ない場合に、承認が通るか必要ない場合に、ブロックされていない場合に許可します）。
- logout は期待通りの動きをしますが、その他のメソッドのように、イベントのログを記録し、他のイベントのトリガーとして使用することもできます。
- profile は、ユーザーにプロファイルを編集することを許可します。プロファイルとは auth\_user テーブルに登録された情報です。このテーブルは固定された構造を持っておらず、カスタマイズができることに注意してください。
- change\_password は、ユーザーにフェイルセーフな方法でパスワードを変更させることができます。
- verify\_email。E メール検証が有効な場合、登録処理を行ったユーザーは、その E メール情報を検証するためのリンクを含むメールを受け取ります。このリンクはこの機能を指し示します。
- retrieve\_username。デフォルトでは、Auth は E メールとパスワードを使用してログインしますが、電子メールの代わりにユーザー名を使うこともできます。後者のケースでは、もしユーザーがユーザー名を忘れた場合、retrieve\_username メソッドによって、ユーザーにメールアドレスを入力させ、そのアドレスに送られたメールからユーザー名を取得することが可能になります。

- `request_reset_password`。このメソッドでは、自分のパスワードを忘れてしまったユーザーが新しいパスワードを要求できます。ユーザーは、`reset_password` を指し示す確認メールを受け取ることになります。
- `impersonate` は、あるユーザーを別の”偽装ユーザー”にすることができます。これは、デバッグと、サポートにとって重要な機能です。`request.args[0]` が偽装されたユーザー ID になります。このメソッドは、`has_permission('impersonate', db_auth_user, user_id)` として指定されたユーザーでログインした時のみ有効となります。
- `groups` は、現在ログオンしているユーザーが所属しているグループが一覧表示されます。
- `not_authorized` は、ユーザーが権限のないページを表示しようとした時に、エラーメッセージを表示します。
- `navbar` は、ログインやユーザー登録リンクなどのバー(ナビゲーションバー)を生成するヘルパーです。

`Logout`、`profile`、`change_password`、`impersonate`、`groups` は、ログインしている必要があります。

デフォルトではこれらはすべて公開されますが、それらの機能の一部のみにアクセスを制限することも可能です。

上記のすべてのメソッドは、`Auth` のサブクラスを作成することで、拡張したり置き換えることが可能です。

また次の例のように、すべてのメソッドは別々の機能として使用することも可能です。

```

1 def mylogin(): return dict(form=auth.login())
2 def myregister(): return dict(form=auth.register())
3 def myprofile(): return dict(form=auth.profile())
4 ...

```

ログインした訪問者のみ関数にアクセスできるよう制限するためには、以下のサンプルのように関数にデコレータを指定します。

```

1 @auth.requires_login()
2 def hello():
3     return dict(message='hello %(first_name)s' % auth.user)

```

全ての関数はデコレータを指定できます。公開されているものだけではありません。もちろんこれは、アクセスコントロールの非常に単純な例です。より複雑な例については後述します。

`auth.user` は、現在ログインしているユーザーに該当する `db.auth_user` のレコードのコピーか、`None` を格納しています。また `auth.user_id` は `auth.user.id` と同じ値(すなわち、現在ログインしているユーザーの `ID`) か `None` になります。

### 9.1.1 登録の制限

訪問者が登録を行うことはできるが、管理者によって承認されるまでログインできないようにする場合：

```
1 auth.settings.registration_requires_approval = True
```

`appadmin` インターフェイスを介して、登録を承認することができます。`auth_user` テーブルを見てください。保留中の登録情報は、`registration_key` フィールドに ”pending” が設定されています。登録を承認するには、このフィールドを空白に設定します。`appadmin` インターフェイスを介して、ユーザーをログインできないようにすることもできます。`auth_user` から該当のユーザーを見つけて `registration_key` を ”blocked” に設定します。”blocked” となったユーザーは、ログインができません。ただし、ログインしていないユーザーからのログインを防ぐことははできますが、既にログインしているユーザーを強制的にログアウトさせることはできません。”disabled” を”blocked” の代わりに使用することもできます。動きは全く同じです。

また以下のステートメントを使用して完全に ”登録” ページへのアクセスをブロックすることができます：

```
1 auth.settings.actions_disabled.append('register')
```

訪問者が登録し登録完了後に、自動ログインを行うことも可能です。その場合に確認用メールを送信し、メールを使った確認を完了しない限り、ログアウト後のログインを禁止するには次の設定を行います。

```
1 auth.settings.registration_requires_approval = True
2 auth.settings.login_after_registration = True
```

`Auth` の他のメソッドも同じ方法で制限することができます。

### 9.1.2 OpenID, Facebook などとの統合

web2py のロールベースのアクセス制御を使用して、OpenID、Facebook、LinkedIn、Google、Dropbox、MySpace、Flickr、などの外部サービスによる認証ができます。最も簡単な方法は、Janrain Engage (旧 RPX) (Janrain.com) を使用することです。

Dropbox はログインしたユーザへの、ストレジサービスプロバイダです。14 章(その他のレシピ)で、ログイン以外のケースについても特別に触れます。

Janrain Engage はミドルウェアの認証を提供するサービスです。Janrain.com への登録、そして使用するドメイン(あなたのアプリケーションの名前)と利用する URL の登録を行うことができます。そして API キーが提供されます。

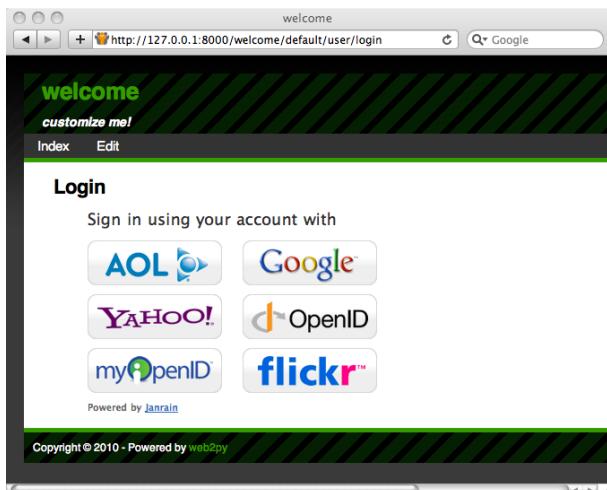
では、あなたの web2py アプリケーションのモデルを編集して、以下の行を auth オブジェクトの定義より後ろに追加してください:

```

1 from gluon.contrib.login_methods.rpx_account import RPXAccount
2 auth.settings.actions_disabled=['register', 'change_password',
3     'request_reset_password']
4 auth.settings.login_form = RPXAccount(request,
5     api_key='...',
6     domain='...',
7     url = "http://localhost:8000/%s/default/user/login" % request.
8         application)

```

最初の行は新しいログインメソッドをインポートしています。2 行目は、ローカルユーザーの登録を無効にしています。3 行目は web2py に対して RPX のログインメソッドを使用するように指示しています。Janrain.com によって提供される独自の `api_key`、登録する際に選択したドメイン、アプリケーションのログインページの `url` を設定する必要があります



新しいユーザーが最初にログインした時に、web2py はそのユーザーに関連づけられた新規の `db.auth_user` のレコードを作成します。`registration_id` フィールドが、ユーザーを一意に識別する ID として設定されます。ほとんどの認証方法は、ユーザー名、電子メール、名前と名字を提供しますが、必ず提供されるとは限りません。どのフィールドを提供しているかは、ユーザーによって選択された login メソッドに依存します。もし同じユーザーが異なる認証のメカニズムを使ってログインした場合（例えば、OpenID でログインしたあとに Facebook でログインし直した）Janrain は、それらを同じユーザーとしては認識せず、それぞれの `registration_id` を発行します。

Janrain によって提供されるデータと `db.auth_user` に保存されるデータとのマッピングは、カスタマイズすることができます。ここでは Facebook を例として示します：

```

1 auth.settings.login_form.mappings.Facebook = lambda profile:\\
2     dict(registration_id = profile["identifier"],\\
3         username = profile[ "preferredUsername"],\\
4         email = profile[ "email"],\\
5         first_name = profile[ "name"] [ "givenName"],\\
6         last_name = profile[ "name"] [ "familyName"])

```

ディクショナリ内のキーは、`db.auth_user` のフィールドです。そして値は、Janrain によって提供されるプロファイルオブジェクトに存在するデータエントリです。後者の詳細については、オンライン Janrain のマニュアルをご覧ください。

さい。

Janrain はまた、ユーザーのログインに関する統計情報を保持します。

このログインフォームは、web2py のロールベースのアクセス制御と完全に統合されており、グループの作成、ユーザーのグループへの割り当て、権限の割り当て、ユーザーのブロック等を行うことができます。

*Janrain* のフリー *Basic* サービスは、年間 2500 ユニークユーザーのサインインが可能です。さらに多くのユーザーの利用には、有料サービスのどれかにアップグレードする必要があります。

*Janrain* ではなく、他のログインメソッド( *LDAP*、*PAM*、*Google*、*OpenID*、*OAuth/Facebook*、*LinkedIn*、など)を必要とする場合は、それを利用するともできます。そのための API は、本章の後半で記載します。

### 9.1.3 CAPTCHA と reCAPTCHA

スパマーやボットによるあなたのサイトへの不要な登録を避けるため、登録用の CAPTCHA が必要になることがあります。web2py はすぐ使える reCAPTCHA [74] をサポートしています。reCAPTCHA は、よく設計されていて、無料で、利便性がよく（訪問者が簡単に単語を読むことができる）簡単に導入できて、その他のサードパーティー製のライブラリを必要としないからです。

以下は、reCPATCHA を使う時に必要となることです。

- reCPATCHA [74] に登録を行い、登録アカウント用のパブリックキー、プライベートキー入手してください。これらは単に二つの文字列です。
- 次のコードを auth オブジェクトを定義した後にモデルに追加してください。

```
1 from gluon.tools import Recaptcha
2 auth.settings.captcha = Recaptcha(request,
3     'PUBLIC_KEY', 'PRIVATE_KEY')
```

reCAPTCHA は、'localhost' や '127.0.0.1' というアドレスのサイトにアクセスした場合はうまく動かないかもしれません。外部に公開されたサイトでのみ動作するように登録されているからです。

Recaptcha のコンストラクタは、いくつかの任意の引数をとります。

```
1 Recaptcha(..., use_ssl=True, error_message='invalid', label='Verify:',  
    options='')
```

`use_ssl=False` が初期値となっています。

`options` は設定用文字列などです。例 `options="theme:'white', lang:'fr'"`

詳細: reCAPTCHA [?] と customizing [?].

もし、reCAPTCHA を必要としない場合は、“gluon/tools.py”の中の `Recaptcha` クラスを見てください。他の CAPTCHA システムを利用するのも簡単です。

注意 `Recaptcha` は `DIV` を拡張したヘルパーがあります。これは `reCaptcha` や他のサービスを使うことのできる、バリデートのダミーのフィールドを生成します。これは定義し使用したフォームを含む、どのフォームでも使うことができます。

```
1 form = FORM(INPUT(...), Recaptcha(...), INPUT(_type='submit'))
```

`SQLFORM` の全てのタイプにも挿入可能です。

```
1 form = SQLFORM(...) or SQLFORM.factory(...)  
2 form.element('table').insert(-1, TR(' ', Recaptcha(...), ' '))
```

### 9.1.4 Auth のカスタマイズ

以下の呼び出しについて

```
1 auth.define_tables()
```

この呼び出しは、まだ登録されていない `Auth` テーブルの定義を行います。つまり、独自の `auth_user` テーブルを定義することができるのです。`auth` のカスタマイズはいくつもの方法があります。最も簡単な方法は、拡張フィールドを追加することです。

```
1 ## after auth = Auth(db)  
2 auth.settings.extra_fields['auth_user']= [  
3     Field('address'),  
4     Field('city'),  
5     Field('zip'),  
6     Field('phone')]  
7 ## before auth.define_tables(username=True)
```

”auth\_user” テーブルだけでなく拡張フィールドは、他の ”auth\_” テーブルでも定義することができます。拡張フィールドを使う方法は、内部メカニズムを停止させることがないため、推奨している方法です。

他に（これはエキスパート用です！）auth テーブル自身を定義する方法です。もしテーブルを auth.define\_tables() の前に定義すれば、デフォルトテーブルの代わりに使われます。以下方法を示します。

```

1 ## after auth = Auth(db)
2 db.define_table(
3     auth.settings.table_user_name,
4     Field('first_name', length=128, default=''),
5     Field('last_name', length=128, default=''),
6     Field('email', length=128, default='', unique=True), # required
7     Field('password', 'password', length=512,           # required
8           readable=False, label='Password'),
9     Field('address'),
10    Field('city'),
11    Field('zip'),
12    Field('phone'),
13    Field('registration_key', length=512,             # required
14        writable=False, readable=False, default=''),
15    Field('reset_password_key', length=512,           # required
16        writable=False, readable=False, default=''),
17    Field('registration_id', length=512,               # required
18        writable=False, readable=False, default=''))
19
20 ## do not forget validators
21 custom_auth_table = db[auth.settings.table_user_name] # get the
22     custom_auth_table
22 custom_auth_table.first_name.requires = \
23     IS_NOT_EMPTY(error_message=auth.messages.is_empty)
24 custom_auth_table.last_name.requires = \
25     IS_NOT_EMPTY(error_message=auth.messages.is_empty)
26 custom_auth_table.password.requires = [IS_STRONG(), CRYPT()]
27 custom_auth_table.email.requires = [
28     IS_EMAIL(error_message=auth.messages.invalid_email),
29     IS_NOT_IN_DB(db, custom_auth_table.email)]
30
31 auth.settings.table_user = custom_auth_table # tell auth to use
32     custom_auth_table
32
33 ## before auth.define_tables()

```

フィールドはいくつでも追加することは可能です。またバリデータも変更可能です。しかし、このサンプルにある ”required” のコメントが入ったフィールド

は削除することはできません。

”password”, ”registration\_key”, ”reset\_password\_key”, ”registration\_id” フィールドを `readable=False` と `writable=False` に設定することは重要です。訪問者が不正にこれらの値を改竄できるようなことがあってはならないからです。

”username” というフィールドを追加した場合、これを”email” フィールドの代わりにログイン時に利用することができます。この場合、以下のようなバリデータを追加する必要があります。

```
1 auth_table.username.requires = IS_NOT_IN_DB(db, auth_table.username)
```

### 9.1.5 Auth テーブルの名前変更

Auth テーブルの名前は、次のように定義されています。

```
1 auth.settings.table_user_name = 'auth_user'
2 auth.settings.table_group_name = 'auth_group'
3 auth.settings.table_membership_name = 'auth_membership'
4 auth.settings.table_permission_name = 'auth_permission'
5 auth.settings.table_event_name = 'auth_event'
```

これらの名前は `auth` オブジェクトの定義を行ってから Auth テーブルの定義を行うまでの間で再定義することにより変更できます。以下は設定例です。

```
1 auth = Auth(db)
2 auth.settings.table_user_name = 'person'
3 #...
4 auth.define_tables()
```

テーブルは次の属性値により、その名前とは無関係に参照することが可能です。

```
1 auth.settings.table_user
2 auth.settings.table_group
3 auth.settings.table_membership
4 auth.settings.table_permission
5 auth.settings.table_event
```

### 9.1.6 その他のログイン方式とログインフォーム

Auth は多数のログイン方式の提供と、新しいログイン方式作成のためのフックを提供します。サポートしている各ログイン方式は、次のフォルダ内の各ファイルに対応しています。

```
1 gluon/contrib/login_methods/
```

各ログイン方式に関しては、ファイル自身に記述された説明を参照してください。ここではいくつかの例だけを説明します。

まず、以下の二つの異なるタイプのログイン方式の違いを理解する必要があります。

- web2py のログインフォームを使ったログイン方式（ただし認証は web2py の外で行われる）例：LDAP。
- 外部のシングルサインオン用フォームを必要とするログイン方式（例：Google や Facebook）

後者の場合は、web2py はログイン用の認証情報を取得せず、サービスプロバイダから発行されたログイン用のトークンのみを取得します。このトークンは、`db.auth_user.registration_id` に保存されます。

それでは、最初の例を見てみましょう。

### ベーシック認証

認証サービスがあります。例えば次の URL が

```
1 https://basic.example.com
```

ベーシック認証を実施するようにします。これはこのサーバが以下のようなヘッダ情報を持つフォームの、HTTP リクエストを受けることを意味します。

```
1 GET /index.html HTTP/1.0
2 Host: basic.example.com
3 Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

後半の文字列は base64 でエンコーディングされた、ユーザー名:パスワード の文字列です。このサービスはユーザーが認証に成功した場合は 200 OK を、それ以外の場合は 400, 401, 402, 403 あるいは 404 を返します。

Auth 標準のログインフォームを使用しユーザー名とパスワードを入力させ、そのサービスに対しての認証をしたいとします。この場合は、単にアプリケーションに以下のコードを挿入するだけで十分です。

```
1 from gluon.contrib.login_methods.basic_auth import basic_auth
2 auth.settings.login_methods.append(
3     basic_auth('https://basic.example.com'))
```

`auth.settings.login_methods` は、順番に実行される認証方式のリストです。デフォルトでは以下のように設定されています。

```
1 auth.settings.login_methods = [auth]
```

代替の認証方式がリストに追加された場合（例 `basic_auth`）、`Auth` はまず `auth_user` に登録された情報に基づいて訪問者の認証を試みます。認証に失敗した場合は、リストに載っている次的方式で認証を試みます。もし認証に成功しなおかつ `auth.settings.login_methods[0]==auth` となるとき、`Auth` は以下の処理を実行します：

- `auth_user` にユーザーが存在しない場合、新しいユーザーが作成され、`username/email` と `password` を保存します。
- `auth_user` にユーザーが存在するが、入力された新しいパスワードが登録されたパスワードと一致しない場合、登録パスワードを新しいものに置き換えます（特に指定しない限り、パスワードは常にハッシュ化されたものを保存します）。

新しいパスワードを `auth_user` に保存したくない場合は、ログイン方法の順番を変更すれば十分です。もしくは、`auth` をリストから削除します。以下は設定例です。

```
1 from gluon.contrib.login_methods.basic_auth import basic_auth
2 auth.settings.login_methods = \
3     [basic_auth('https://basic.example.com')]
```

ここで記載する他のログイン方法についても同様です。

## SMTP と Gmail

### SMTP and Gmail

ログイン用の認証情報をリモートの SMTP サーバ（例えば Gmail）を用いて検証することができます。すなわちユーザーの提供する `email` とパスワードが、Gmail の SMTP サーバ（`smtp.gmail.com:587`）に正常にアクセスできるものであればユーザーをログインさせることができます。これに必要なのは以下のコードだけです。

```
1 from gluon.contrib.login_methods.email_auth import email_auth
2 auth.settings.login_methods.append(
3     email_auth("smtp.gmail.com:587", "@gmail.com"))
```

`email_auth` の最初の引数は、SMTP サーバの ”address:port” です。二つ目の引数はメールのドメインです。

これは、TLS 認証を必要とする SMTP サーバに対しても動作します。

## PAM

Pluggable Authentication Modules(PAM) を使った認証は、前のケースと同様に機能します。これにより、オペレーティングシステムのアカウントを用いてユーザーの認証ができるようになります。

```
1 from gluon.contrib.login_methods.pam_auth import pam_auth
2 auth.settings.login_methods.append(pam_auth())
```

## LDAP

LDAP を使った認証は、前のケースと全く同様に機能します。

LDAP を MS Active Directory とともに使用する方法です。：

```
1 from gluon.contrib.login_methods.ldap_auth import ldap_auth
2 auth.settings.login_methods.append(ldap_auth(mode='ad',
3     server='my.domain.controller',
4     base_dn='ou=Users, dc=domain, dc=com'))
```

LDAP を Lotus Notes とともに使用する方法です。：

```
1 auth.settings.login_methods.append(ldap_auth(mode='domino',
2     server='my.domino.server'))
```

LDAP を OpenLDAP ( UID による ) とともに使用する方法です。：

```
1 auth.settings.login_methods.append(ldap_auth(server='my.ldap.server',
2     base_dn='ou=Users, dc=domain, dc=com'))
```

LDAP を OpenLDAP ( CN による ) とともに方法です。

```
1 auth.settings.login_methods.append(ldap_auth(mode='cn',
2     server='my.ldap.server', base_dn='ou=Users, dc=domain, dc=com'))
```

## Google App Engine

Google App Engine 上で動作しているアプリケーションで Google による認証を行う場合には、web2py のログインフォームによる認証は行わず、Google のログインページにリダイレクトを行い、そして成功したら元のページに戻るようにします。前述の例とは動作が異なるため、API は少し異なります。

```

1 from gluon.contrib.login_methods.gae_google_login import
   GaeGoogleAccount
2 auth.settings.login_form = GaeGoogleAccount()

```

## OpenID

(OpenIDのサポートがある)Janrainの組み込みについて説明しましたが、これが最も簡単なOpenIDを使う方法です。しかし、サードパーティ製のサービスに頼るのではなく、OpenIDプロバイダにその利用者(あなたのアプリケーション)から直接アクセスしたいこともあります。

設定例です。

```

1 from gluon.contrib.login_methods.openid_auth import OpenIDAuth
2 auth.settings.login_form = OpenIDAuth(auth)

```

OpenIDAuthは`python-openid`モジュールを別途インストールする必要があります。内部では、このログイン方式は次のようなテーブルを定義します。

```

1 db.define_table('alt_logins',
2                 Field('username', length=512, default=''),
3                 Field('type', length=128, default='openid', readable=False),
4                 Field('user', self.table_user, readable=False))

```

このテーブルには、各ユーザーのopenidのユーザー名が保存されます。現在ログインしているユーザーのopenidを表示したい場合は、次のようにします。

```

1 {{=auth.settings.login_form.list_user_openids()}}

```

## OAuth2.0とFacebook

(Facebookのサポートがある)Janrainの組み込みについて説明しましたが、サードパーティ製のサービスに頼るのではなく、OAuth2.0プロバイダ(例えばFacebook)にその利用者(あなたの作成するアプリケーション)から直接アクセスしたいこともあります。方法は以下の通りです。

```

1 from gluon.contrib.login_methods.oauth20_account import OAuthAccount
2 auth.settings.login_form=OAuthAccount(YOUR_CLIENT_ID, YOUR_CLIENT_SECRET
)

```

自分のアプリケーションではなく、特定のFacebookアプリケーションのAPIにアクセスするためにFacebook OAuth2.0を用いる場合は、少し複雑になります。Facebook Graph APIにアクセスする例を示します。

初めに、Facebook Python SDK をインストールする必要があります。

そして、モデルに以下のコードを追加します。

```

1 ## import required modules
2 from facebook import GraphAPI
3 from gluon.contrib.login_methods.oauth20_account import OAuthAccount
4 ## extend the OAuthAccount class
5 class FaceBookAccount(OAuthAccount):
6     """OAuth impl for Facebook"""
7     AUTH_URL="https://graph.facebook.com/oauth/authorize"
8     TOKEN_URL="https://graph.facebook.com/oauth/access_token"
9     def __init__(self, g):
10         OAuthAccount.__init__(self, g,
11                               YOUR_CLIENT_ID,
12                               YOUR_CLIENT_SECRET,
13                               self.AUTH_URL,
14                               self.TOKEN_URL)
15         self.graph = None
16     # override function that fetches user info
17     def get_user(self):
18         "Returns the user using the Graph API"
19         if not self.accessToken():
20             return None
21         if not self.graph:
22             self.graph = GraphAPI((self.accessToken()))
23         try:
24             user = self.graph.get_object("me")
25             return dict(first_name = user['first_name'],
26                         last_name = user['last_name'],
27                         username = user['id'])
28         except GraphAPIError:
29             self.session.token = None
30             self.graph = None
31             return None
32     ## use the above class to build a new login form
33 auth.settings.login_form=FaceBookAccount()

```

## LinkedIn

( LinkedIn のサポートがある ) Janrain の組み込みについて説明しましたが、これが最も簡単な OAuth を使った方法です。しかし、サードパーティー製のサービスに頼るのではなく、Janrain プロバイダが提供するよりも多くの情報を得たいために LinkedIn に直接アクセスしたい場合もあります。

方法は以下の通りです。

```

1 from gluon.contrib.login_methods.linkedin_account import
   LinkedInAccount
2 auth.settings.login_form=LinkedInAccount (request,KEY,SECRET,RETURN_URL)

```

LinkedInAccount は”python-linkedin” モジュールを別途インストールする必要があります。

X509 x509 証明書と証明書から取得した認証をページに渡すことによって、ログインがすることが可能です。これには M2Crypto を次のサイトからインストールする必要があります。

```
1 http://chandlerproject.org/bin/view/Projects/MeTooCrypto
```

M2Crypton がインストールされたら、使用可能です。

```

1 from gluon.contrib.login_methods.x509_auth import X509Account
2 auth.settings.actions_disabled=['register', 'change_password',
   'request_reset_password']
3 auth.settings.login_form = X509Account ()

```

web2py 内部で x509 証明書を渡して認証します。どのようにするかはプラウザによって違いますが Web サービス用の証明書使用する場合がほとんどです。次の例は curl を使って認証を試したものです。

```

1 curl -d "firstName=John&lastName=Smith" -G -v --key private.key \
2     --cert server.crt https://example/app/default/user/profile

```

この機能は Rocket ( web2py 組み込みの Web サーバ ) では、すぐ使えます。しかし違う Web サーバを使用する場合は、サーバ側で何らかの特別な設定が必要です。証明書をローカルホスト上のどこに置くか、クライアントから来る証明書を検証する必要があるなど、個別に Web サーバに指定する必要があります。これらは Web サーバに依存するため、説明は省略します。

### Multiple login フォーム

ログイン方法の中には、login\_form を変更するものとしないものがあります。変更する場合は、他と共に存することができないかもしれません。ただし、同じページに複数のログインフォームを表示することで共存させることができます。web2py は、そのための方法を提供しています。ここに、通常のログイン (auth) と RPX ログイン (janrain.com) を共存させる例を示します。

```

1 from gluon.contrib.login_methods.extended_login_form import
   ExtendedLoginForm

```

```

2 other_form = RPXAccount(request, api_key='...', domain='...', url='...')
3 auth.settings.login_form = ExtendedLoginForm(request,
4     auth, other_form, signals=['token'])

```

もしシグナルが設定され、リクエストに含まれるパラメータがいずれかのシグナルにマッチした場合、代わりに `other_form.login_form` への呼び出しを返します。`other_form` は、個別な状態をハンドルできます。例えば `other_form.login_form` 内での、複数のステップによる OpenID ログインです。

そうでない場合は、`other_form` と共に通常のログインフォームが表示されます。

## 9.2 Mail と Auth

メールを次のように設定できます。

```

1 from gluon.tools import Mail
2 mail = Mail()
3 mail.settings.server = 'smtp.example.com:25'
4 mail.settings.sender = 'you@example.com'
5 mail.settings.login = 'username:password'

```

もしくは簡単に `auth` を使い、メールプロバイダの設定ができます。

```

1 mail = auth.settings.mailer
2 mail.settings.server = 'smtp.example.com:25'
3 mail.settings.sender = 'you@example.com'
4 mail.settings.login = 'username:password'

```

`mail.settings` を使用する SMTP サーバの正しいパラメータに置き換える必要があります。`mail.settings.login=False` と設定すると、SMTP サーバは認証を要求しません。web2py API のメールとメール設定に関しては、8章でさらに触っています。ここでは `Mail` と `Auth` の相互作用に関して、限定的に説明します。

`Auth` でのメールによる確認は、デフォルトでは無効になっています。メールを有効にするには、モデルの `auth` 定義の後に次のコードを追加します。

```

1 auth.settings.registration_requires_verification = False
2 auth.settings.registration_requires_approval = False
3 auth.settings.reset_password_requires_verification = True

```

```

4 auth.messages.verify_email = 'Click on the link http://' + \
5   request.env.http_host + \
6   URL(r=request, c='default', f='user', args=['verify_email']) + \
7   '/%(key)s to verify your email'
8 auth.messages.reset_password = 'Click on the link http://' + \
9   request.env.http_host + \
10  URL(r=request, c='default', f='user', args=['reset_password']) + \
11  '/%(key)s to reset your password'

```

次の文字列は、

```
1 'Click on the link ...'
```

`auth.messages.verify_email` にありますが、URL の設定を `verify_email` が動く正しい URL に変更する必要があります。なぜなら、web2py はプロキシの背後にインストールされているかもしれません、自身の公開 URL を確実に決定することができないからです。

### 9.3 認可

新しいユーザーが登録されると、そのユーザーを含む新しいグループが作成されます。新しいユーザーのロールは、慣例的に”`user_[id]`”となります。ここで、`[id]` の部分は新しいユーザーの id です。自動的にグループを作成したくない場合は、以下のようにします。

```
1 auth.settings.create_user_groups = False
```

ただし、この設定は推奨していません。

ユーザーはグループのメンバーシップを保持します。それぞれのグループは名前かロール名で特定されます。グループはパーミッションを保持します。従って、ユーザーも所属するグループのパーミッションを保持するになります。

グループの作成とグループへのメンバーシップとパーミッションの割り当ては、`appadmin` を利用するか、以下のメソッドを使ってプログラム的に行うことができます。

```
1 auth.add_group('role', 'description')
```

```
1 auth.add_group('role', 'description')
```

これは、新しく作成したグループの id を返します。

```
1 auth.del_group(group_id)
```

これは、group\_idを持つグループを削除します。

```
1 auth.del_group(auth.id_group('user_7'))
```

これは、"user\_7"のロールを持つグループ、つまり7番のユーザーに一意に関連付けられたグループを削除します。

```
1 auth.user_group(user_id)
```

これは、user\_idで特定されるユーザーに一意に関連付けられたグループのidを返します。

```
1 auth.add_membership(group_id, user_id)
```

これは、group\_id グループへのメンバーシップを user\_id 与えます。  
user\_id が指定されない場合、現在のログインユーザーの id が使用されます。

```
1 auth.del_membership(group_id, user_id)
```

これは、group\_id グループへの user\_id のメンバーシップを取り消します。  
user\_id が指定されない場合、現在のログインユーザーの id が使用されます。

```
1 auth.has_membership(group_id, user_id, role)
```

これは user\_id が、group\_id グループか、もしくは特定のロールを持つグループに属するメンバーシップかどうかを確認します。group\_id と role の両方ではなく、どちらかを指定してください。user\_id が指定されない場合、現在のログインユーザーの id が使用されます。

```
1 auth.add_permission(group_id, 'name', 'object', record_id)
```

これは、(ユーザー定義の)"object"というオブジェクトに対する(ユーザー定義の)"name"というパーミッションを、group\_id グループのメンバーに与えます。"object"がテーブル名の場合、record\_id がゼロならばテーブル全体に対して、record\_id がゼロより大きい場合は特定のレコードにだけにパーミッションを与えます。パーミッションをテーブルに与えた場合は、('create', 'read', 'update', 'delete', 'select')の中からパーミッション名を利用するのが一般的です。CRUD はこれらのパーミッション名を元に、認可を制御します。

group\_id の指定がゼロの場合、現在のログインユーザーに関連付けられているユニークなグループが使用されます。

また、`auth.id_group(role="...")` を使用すると、指定したロール名からグループ id を取得できます。

```
1 auth.del_permission(group_id, 'name', 'object', record_id)
```

これは、パーミッションを取り消します。

```
1 auth.has_permission('name', 'object', record_id, user_id)
```

これは、`user_id` で特定されるユーザーが、指定したパーミッションを持つグループのメンバーかどうかを確認します。

```
1 rows = db(auth.accessible_query('read', db.mytable, user_id)) \
2     .select(db.mytable.ALL)
```

これは、”mytable” テーブルの中の `user_id` ユーザーが”read” パーミッションを持つ行を、全て返します。`user_id` の指定がない場合、現在のログインユーザーの id が使用されます。`accessible_query(...)` は他のクエリと組み合って、より複雑なものを作ることができます。`accessible_query(...)` は JOIN を使う唯一の Auth のメソッドです。そのため、Google App Engine では動作しません。

以下はコード例で使用する定義です。

```
1 >>> from gluon.tools import Auth
2 >>> auth = Auth(db)
3 >>> auth.define_tables()
4 >>> secrets = db.define_table('document', Field('body'))
5 >>> james_bond = db.auth_user.insert(first_name='James',
6                                         last_name='Bond')
```

認可に関するコード例です。

```
1 >>> doc_id = db.document.insert(body = 'top secret')
2 >>> agents = auth.add_group(role = 'Secret Agent')
3 >>> auth.add_membership(agents, james_bond)
4 >>> auth.add_permission(agents, 'read', secrets)
5 >>> print auth.has_permission('read', secrets, doc_id, james_bond)
6 True
7 >>> print auth.has_permission('update', secrets, doc_id, james_bond)
8 False
```

### 9.3.1 デコレータ

パーミッションを確認する最も一般的な方法は、上記のメソッドを明示的に呼び出すではありません。関数にデコレータを指定し、ログインしている訪問者に対してパーミッションのチェックをするようにします。以下、いくつか例を示します。

```

1 def function_one():
2     return 'this is a public function'
3
4 @auth.requires_login()
5 def function_two():
6     return 'this requires login'
7
8 @auth.requires_membership('agents')
9 def function_three():
10    return 'you are a secret agent'
11
12 @auth.requires_permission('read', secrets)
13 def function_four():
14    return 'you can read secret documents'
15
16 @auth.requires_permission('delete', 'any file')
17 def function_five():
18     import os
19     for file in os.listdir('./'):
20         os.unlink(file)
21     return 'all files deleted'
22
23 @auth.requires(auth.user_id==1 or request.client=='127.0.0.1',
24               requires_login=True)
25 def function_six():
26     return 'you can read secret documents'
27
28 @auth.requires_permission('add', 'number')
29 def add(a, b):
30     return a + b
31
32 def function_seven():
33     return add(3, 4)
```

`@auth.requires(condition)` の condition 引数は、呼び出し可能にすることができます。また、`@auth.requires` は、デフォルト値が `True` のオプションの引数 `requires_login` を取ります。もし `False` を指定した場合、`true / false` のように condition 引数を評価する前に、ログインを要求しません。condition 引

数はブール値か、ブール値を返す関数を指定します。

最初の一つ以外の全ての関数は、訪問者がパーミッションを持っているか持っていないかによってアクセスを制限することを、注意してください。

訪問者がログインしていない場合、パーミッションの確認はできません。このためログインページにリダイレクトされ、ログイン後にパーミッションが必要なページに戻ります。

### 9.3.2 権限要求の組み合わせ

権限要求を組み合わせることが必要になる場合があります。これは真か偽かの条件を一つの引数で設定する、汎用的な `requires` デコレータによって行うことができます。例えば、`agents` にアクセス権限を与えるが、火曜日のみにする場合は次のようにします：

```
1 @auth.requires(auth.has_membership(group_id=agents) \
2                 and request.now.weekday() == 1)
3 def function_seven():
4     return 'Hello agent, it must be Tuesday!'
```

次の書き方もできます。

```
1 @auth.requires(auth.has_membership(role='Secret Agent') \
2                 and request.now.weekday() == 1)
3 def function_seven():
4     return 'Hello agent, it must be Tuesday!'
```

### 9.3.3 権限と CRUD

デコレータと `and/or` を使って明示的なチェックすることは、アクセス制御の一つの実装方法です。

もう一つの実装方法は、データベースアクセスに常に CRUD を使う（SQLFORM ではなく）ことによって、データベーステーブルやレコードに対するアクセス制御を CRUD から実行することです。これは次の命令文を用いて、Auth と CRUD を紐づけることで実施します。

```
1 crud.settings.auth = auth
```

これは、訪問者が明示的なアクセス権を持ってログインしていない限り、いかなる CRUD 関数へのアクセスも防ぎます。例えば、訪問者はコメントを投稿できるが、自分自身のコメントしか更新できないようにするには、次のようにします（crud、auth および db.comment が定義されているものとします）：

```

1 def give_create_permission(form):
2     group_id = auth.id_group('user_%s' % auth.user.id)
3     auth.add_permission(group_id, 'read', db.comment)
4     auth.add_permission(group_id, 'create', db.comment)
5     auth.add_permission(group_id, 'select', db.comment)
6
7 auth.settings.register_onaccept = give_create_permission
8 crud.settings.auth = auth
9
10 def post_comment():
11     form = crud.create(db.comment, onaccept=give_update_permission)
12     comments = db(db.comment).select()
13     return dict(form=form, comments=comments)

```

これにより訪問者がログインし明示的なアクセス権を持たない限り、どの CRUD 関数もアクセスできなくなります。例えば、コメントを作成できるが、自分が作成したコメントしか更新できません（crud、auth、db.comment は宣言済みとする）：

```

1 def give_create_permission(form):
2     group_id = auth.id_group('user_%s' % auth.user.id)
3     auth.add_permission(group_id, 'read', db.comment)
4     auth.add_permission(group_id, 'create', db.comment)
5     auth.add_permission(group_id, 'select', db.comment)
6
7 auth.settings.register_onaccept = give_create_permission
8 crud.settings.auth = auth
9
10 def post_comment():
11     form = crud.create(db.comment, onaccept=give_update_permission)
12     comments = db(db.comment).select()
13     return dict(form=form, comments=comments)

```

（'read' 権限を持つ）特定のレコードを選択することもできます。

```

1 def post_comment():
2     form = crud.create(db.comment, onaccept=give_update_permission)
3     query = auth.accessible_query('read', db.comment, auth.user.id)
4     comments = db(query).select(db.comment.ALL)
5     return dict(form=form, comments=comments)

```

以下で有効なパーミッション名は

```
1 crud.settings.auth = auth
```

are "read", "create", "update", "delete", "select", "impersonate" になります。

### 9.3.4 認可とダウンロード

デコレータや `crud.settings.auth` を使用しても、通常のダウンロード関数を使ったファイルダウンロードの認可は実施されません。

```
1 def download(): return response.download(request, db)
```

ファイルダウンロードでの認可を実施する場合、どの"upload" フィールドがアクセス制御が必要なファイルと関連があるかを、明示的に宣言しなければなりません。以下、設定例です。

```
1 db.define_table('dog',
2     Field('small_image', 'upload'),
3     Field('large_image', 'upload'))
4
5 db.dog.large_image.authorization = lambda record: \
6     auth.is_logged_in() and \
7     auth.has_permission('read', db.dog, record.id, auth.user.id)
```

`upload` フィールドの `authorization` 属性は、`None` とするか(デフォルト) 関数を指定することができます。関数は、ユーザーがログインしているかどうか、そして現在の行に対する'read' パーミッションを持っているかどうかを判断します。この例では、"small\_image" フィールドにリンクされた画像のダウンロードには何の制限もかけられていません。しかし"large\_image" フィールドにリンクされた画像には、アクセス制御がかけられています。

### 9.3.5 アクセス制御とベーシック認証

アクセス制御を行うデコレータが設定されている機能を、サービスとして公開しなければならない場合があります。すなわち、プログラムやスクリプトから呼ばれても、認証と権限チェックを行なえることが必要な場合です。

`Auth` は、ベーシック認証によるログインを可能にできます。

```
1 auth.settings.allow_basic_login = False
```

上記の設定とともに、以下のような機能を定義すると

```
1 @auth.requires_login()
2 def give_me_time():
3     import time
4     return time.ctime()
```

次のシェルコマンドから、機能を呼び出すことができます。

```
1 wget --user=[username] --password=[password]
2 http://.../[app]/[controller]/give_me_time
```

ベーシック認証によるログインは、しばしば（次章で説明する）サービスに対する唯一のオプションとなります。しかしデフォルトは無効になっています。

### 9.3.6 手動認証

独自ロジックの実装や”手動”によるログインを行いたい場合が稀にあります。このような時、次の関数を呼び出すことができます。

```
1 user = auth.login_bare(username, password)
```

`login_bare` はユーザーが存在しパスワードが正しい時、ユーザー（レコード）を返します。そうでない場合、`None` を返します。また、”auth\_user” テーブルに”username” フィールドがない場合、`username` は `email` になります。

### 9.3.7 設定とメッセージ

`Auth` のカスタマイズに関する、全てのパラメタの一覧を以下に示します。

次のパラメタには、`gluon.tools.Mail` オブジェクトを指定してください。これにより、`auth` によるメールの送信が可能になります。

```
1 auth.settings.mailer = None
```

次のパラメタには、`user` の機能を定義するコントローラの名前を指定してください。

```
1 auth.settings.controller = 'default'
```

次のパラメタは、とても重要な設定です。

```
1 auth.settings.hmac_key = None
```

これには、”sha512:a-pass-phrase” のような設定をしてください。これは、auth\_user テーブルの”password” フィールドに対する CRYPT バリデータへ渡されます。パスワードをハッシュ化するために使われるアルゴリズムとパスフレーズになります。

最小パスワード文字数のデフォルト値は 4 です。これは変更可能です。

```
1 auth.settings.password_min_length = 4
```

user 関数の特定機能を無効にするのは、機能名(パラメータ文字列)を次のリストに追加します。

```
1 auth.settings.actions_disabled = []
```

設定例です。

```
1 auth.settings.actions_disabled.append('register')
```

ユーザー登録を無効にします。

ユーザー登録確認のためのメールを受け取る設定にするには、次を True にします。

```
1 auth.settings.registration_requires_verification = False
```

ユーザー登録後自動でログインする設定は、次を True にします。この場合、メールによるユーザー登録確認プロセス中であっても自動ログインします。

```
1 auth.settings.login_after_registration = False
```

新規のユーザーに対しては、管理者の承認を待ってからログインする設定は、次を True にします。

```
1 auth.settings.registration_requires_approval = False
```

管理者が承認するには appadmin を使用するかプログラムなどを使って、registration\_key=='' に変更します。

新規のユーザーに、ユーザー専用のユニークグループを作成しないようにするには、次を False にします

```
1 auth.settings.create_user_groups = True
```

次の設定は前述のように、代替のログイン方法とログインフォームを決定します。

```
1 auth.settings.login_methods = [auth]
2 auth.settings.login_form = auth
```

ベーシック認証を使用するときは、次を `True` にします。

```
1 auth.settings.allows_basic_login = False
```

次の設定は、`login` 機能の URL です。

```
1 auth.settings.login_url = URL('user', args='login')
```

既にログインしている状態でユーザー登録用のページを開いた時は、次の URL にリダイレクトします。

```
1 auth.settings.logged_url = URL('user', args='profile')
```

プロファイル(ユーザ登録情報)に画像がある場合に指定する、ダウンロード用の URL です。

```
1 auth.settings.download_url = URL('download')
```

以下は `auth` の様々な機能が実行された後に、(参照先が無い場合の) リダイレクト先の URL を指定しています。

```
1 auth.settings.login_next = URL('index')
2 auth.settings.logout_next = URL('index')
3 auth.settings.profile_next = URL('index')
4 auth.settings.register_next = URL('user', args='login')
5 auth.settings.retrieve_username_next = URL('index')
6 auth.settings.retrieve_password_next = URL('index')
7 auth.settings.change_password_next = URL('index')
8 auth.settings.request_reset_password_next = URL('user', args='login')
9 auth.settings.reset_password_next = URL('user', args='login')
10 auth.settings.verify_email_next = URL('user', args='login')
```

呼び出し関数では認証が必要なのに訪問者がログインしていない場合、`auth.settings.login_url` にリダイレクトします。このデフォルト値は `URL('default', 'user/login')` になります。再定義することにより、この動作を置き換えることが可能です。

```
1 auth.settings.on_failed_authentication = lambda url: redirect(url)
```

これは認証を失敗した場合のリダイレクト時に呼び出される関数です。`url` 引数は、この関数にログインページの URL を渡します。

訪問者がアクセスに必要な十分な権限を所持していない場合、次に指定している URL にリダイレクトされます。

```
1 auth.settings.on_failed_authorization = \
2     URL('user', args='on_failed_authorization')
```

ユーザーが他の場所にリダイレクトするように、この変数を変更することも可能です。

多くの場合、`on_failed_authorization` は URL です。しかしアクセスが失敗した場合に呼び出される、URL を返す関数にすることも可能です。

以下は、バリデーション実施後かつデータベース IO 処理前に実行される各機能のコールバック関数の一覧です。

```
1 auth.settings.login_onvalidation = []
2 auth.settings.register_onvalidation = []
3 auth.settings.profile_onvalidation = []
4 auth.settings.retrieve_password_onvalidation = []
5 auth.settings.reset_password_onvalidation = []
```

各コールバックは `form` オブジェクトを引数とすることが必要です。これにより、データベースの IO 処理前に、そのフォームオブジェクトの属性を修正することができます。

以下は、データベース IO 処理後かつリダイレクト前に実行する、コールバック関数の一覧です。

```
1 auth.settings.login_onaccept = []
2 auth.settings.register_onaccept = []
3 auth.settings.profile_onaccept = []
4 auth.settings.verify_email_onaccept = []
```

設定例です。

```
1 auth.settings.register_onaccept.append(lambda form:\n2     mail.send(to='you@example.com', subject='new user',\n3                 message="new user email is %s'%form.vars.email))
```

いくつかの `auth` 機能に対して、キャプチャを有効にできます。

```
1 auth.settings.captcha = None
2 auth.settings.login_captcha = None
3 auth.settings.register_captcha = None
4 auth.settings.retrieve_username_captcha = None
5 auth.settings.retrieve_password_captcha = None
```

.captcha が gluon.tools.Recaptcha に設定されている場合、(.login\_captcha などの) 個別オプションが None と指定している機能はすべて、キャプチャが有効になります。もし個別オプションの値が、False だったら、その機能のキャプチャは有効になりません。.captcha が None に設定されている場合、個別オプションに gluon.tools.Recaptcha を指定している機能のみがキャプチャが有効になり、それ以外の指定が無い機能ではキャプチャは有効にはなりません。

次は、ログインセッションの有効期間です。

```
1 auth.settings.expiration = 3600 # seconds
```

パスワードフィールド名は変更することができます（例えば、Firebird では”password”は予約語です。このため、フィールド名として利用できません）

```
1 auth.settings.password_field = 'password'
```

通常、ログインフォームはメールアドレスでのバリデーションを試みます。次の設定を False にすれば無効にできます。

```
1 auth.settings.login_email_validate = True
```

ユーザプロファイルの編集ページでレコード ID を表示するには、次の設定を True にします。

```
1 auth.settings.showid = False
```

フォームのカスタマイズにおいて、自動的でエラー通知を行いたくない場合は次の設定を False にします。

```
1 auth.settings.hideerror = False
```

フォームのスタイルを変更したい場合は次のようにします。

```
1 auth.settings.formstyle = 'table3cols'
```

（この値は”table2cols”、“divs”及び”ul”を指定することが可能です。）auth が生成するフォームのラベル区切り記号を設定できます。

```
1 auth.settings.label_separator = ':'
```

”remember me” オプションによって、自動でログイン状態を維持することが可能です。この機能はデフォルトでは有効になっています。次の設定によって、自

動ログインの有効期限の変更や、自動ログインのオプション自体を無効にすることができます。

```
1 auth.settings.long_expiration = 3600*24*30 # one month  
2 auth.settings.remember_me_form = True
```

次のメッセージをカスタマイズすることができます。その用途や場面は明らかなため、特に説明はしません。

```
1 auth.messages.submit_button = 'Submit'  
2 auth.messages.verify_password = 'Verify Password'  
3 auth.messages.delete_label = 'Check to delete:'  
4 auth.messages.function_disabled = 'Function disabled'  
5 auth.messages.access_denied = 'Insufficient privileges'  
6 auth.messages.registration_verifying = 'Registration needs verification'  
  
7 auth.messages.registration_pending = 'Registration is pending approval'  
8 auth.messages.login_disabled = 'Login disabled by administrator'  
9 auth.messages.logged_in = 'Logged in'  
10 auth.messages.email_sent = 'Email sent'  
11 auth.messages.unable_to_send_email = 'Unable to send email'  
12 auth.messages.email_verified = 'Email verified'  
13 auth.messages.logged_out = 'Logged out'  
14 auth.messages.registration_successful = 'Registration successful'  
15 auth.messages.invalid_email = 'Invalid email'  
16 auth.messages.unable_send_email = 'Unable to send email'  
17 auth.messages.invalid_login = 'Invalid login'  
18 auth.messages.invalid_user = 'Invalid user'  
19 auth.messages.is_empty = "Cannot be empty"  
20 auth.messages.mismatched_password = "Password fields don't match"  
21 auth.messages.verify_email = ...  
22 auth.messages.verify_email_subject = 'Password verify'  
23 auth.messages.username_sent = 'Your username was emailed to you'  
24 auth.messages.new_password_sent = 'A new password was emailed to you'  
25 auth.messages.password_changed = 'Password changed'  
26 auth.messages.retrieve_username = 'Your username is: %(username)s'  
27 auth.messages.retrieve_username_subject = 'Username retrieve'  
28 auth.messages.retrieve_password = 'Your password is: %(password)s'  
29 auth.messages.retrieve_password_subject = 'Password retrieve'  
30 auth.messages.reset_password = ...  
31 auth.messages.reset_password_subject = 'Password reset'  
32 auth.messages.invalid_reset_password = 'Invalid reset password'  
33 auth.messages.profile_updated = 'Profile updated'  
34 auth.messages.new_password = 'New password'  
35 auth.messages.old_password = 'Old password'  
36 auth.messages.group_description = \  
    'Group uniquely assigned to user %(id)s'  
38 auth.messages.register_log = 'User %(id)s Registered'
```

```

39 auth.messages.login_log = 'User %(id)s Logged-in'
40 auth.messages.logout_log = 'User %(id)s Logged-out'
41 auth.messages.profile_log = 'User %(id)s Profile updated'
42 auth.messages.verify_email_log = 'User %(id)s Verification email sent'
43 auth.messages.retrieve_username_log = 'User %(id)s Username retrieved'
44 auth.messages.retrieve_password_log = 'User %(id)s Password retrieved'
45 auth.messages.reset_password_log = 'User %(id)s Password reset'
46 auth.messages.change_password_log = 'User %(id)s Password changed'
47 auth.messages.add_group_log = 'Group %(group_id)s created'
48 auth.messages.del_group_log = 'Group %(group_id)s deleted'
49 auth.messages.add_membership_log = None
50 auth.messages.del_membership_log = None
51 auth.messages.has_membership_log = None
52 auth.messages.add_permission_log = None
53 auth.messages.del_permission_log = None
54 auth.messages.has_permission_log = None
55 auth.messages.label_first_name = 'First name'
56 auth.messages.label_last_name = 'Last name'
57 auth.messages.label_username = 'Username'
58 auth.messages.label_email = 'E-mail'
59 auth.messages.label_password = 'Password'
60 auth.messages.label_registration_key = 'Registration key'
61 auth.messages.label_reset_password_key = 'Reset Password key'
62 auth.messages.label_registration_id = 'Registration identifier'
63 auth.messages.label_role = 'Role'
64 auth.messages.label_description = 'Description'
65 auth.messages.label_user_id = 'User ID'
66 auth.messages.label_group_id = 'Group ID'
67 auth.messages.label_name = 'Name'
68 auth.messages.label_table_name = 'Table name'
69 auth.messages.label_record_id = 'Record ID'
70 auth.messages.label_time_stamp = 'Timestamp'
71 auth.messages.label_client_ip = 'Client IP'
72 auth.messages.label_origin = 'Origin'
73 auth.messages.label_remember_me = "Remember me (for 30 days)"

```

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".

## 9.4 Central Authentication Service

web2py は、サードパーティ認証やシングルサインオンのサポートをしています。ここでは業界標準で、クライアント及びサーバ共に web2py に組み込まれてい

る、Central Authentication Service (CAS) について取り上げます。

CAS は分散認証を行うためのオープンなプロトコルです。次のように動作します。訪問者が Web サイトにアクセスした時、アプリケーションはユーザーが既に認証されているかどうかをセッションを使ってチェックします（例、`session.token` オブジェクトを経由）。もし認証されていない場合、コントローラは CAS アプライアンスからユーザーをリダイレクトします。リダイレクト先は、ユーザーがログイン、ユーザー登録、認証情報の管理（名前、メールアドレス、パスワード）ができるところです。ユーザー登録をした場合は、そのユーザーが登録確認メールに対応するまで登録を保留します。登録が完了しログインすると、CAS アプライアンスはキーとともにユーザーをアプリケーションへリダイレクトします。アプリケーションではこのキーを使って、ユーザーの認証情報を取得します。これはバックグラウンドで、CAS サーバとの HTTP リクエストを介して実行します。

この仕組みを使うと、複数のアプリケーションが一つの CAS サーバを通してシングルサインオンを使用することができます。認証を提供するサーバは、サービスプロバイダと呼びます。訪問者を認証しようとするアプリケーションは、サービスコンシューマと呼びます。

CAS は OpenID と似ていますが、大きな違いが一つあります。OpenID の場合、訪問者はサービスプロバイダを選択することができます。CAS の場合は、アプリケーションがサービスプロバイダを選択します。そのため、CAS はより安全です。web2py の CAS プロバイダを実行するのはひな形アプリケーションをコピーするのと同じくらい簡単です。実際に機能を公開した web2py アプリケーションを次に示します。

```
1 ## in provider app
2 def user(): return dict(form=auth())
```

CAS 2.0 プロバイダです。そのサービスは URL でアクセスできます。

```
1 http://.../provider/default/user/cas/login
2 http://.../provider/default/user/cas/validate
3 http://.../provider/default/user/cas/logout
```

（アプリケーションは “provider” と呼ばれると仮定しています）

プロバイダに対するシンプルな代理認証によって、他の Web アプリケーション（コンシューマ）からサービスにアクセス可能です。

```

1 ## in consumer app
2 auth = Auth(db, cas_provider = 'http://127.0.0.1:8000/provider/default/
    user/cas')

```

コンシューマアプリケーションのログイン URL にアクセスすると、認証の働きをするプロバイダアプリケーションにリダイレクトし、さらにコンシューマにリダイレクトで戻ります。ユーザ登録、ログアウト、パスワード変更、パスワードの再確認などのすべての処理は、プロバイダアプリケーション上で完結します。拡張フィールドやローカルプロファイルのために、コンシューマにログインユーザのエントリが作成されます。幸いのこと CAS 2.0 は、すべてのフィールドはプロバイダ上で読み取り可能です。そして コンシューマの `auth_user` テーブルに相当するフィールドは、自動でコピーされます。

`Auth(..., cas_provider='...')` はサードパーティのプロバイダと共に動作し、CAS 1.0 及び 2.0 をサポートします。バージョンの検出は自動で行います。デフォルトでは追加することにより、ベース（上記の `cas_provider` URL）からプロバイダの URL を生成します。

```

1 /login
2 /validate
3 /logout

```

次の設定は、コンシューマとプロバイダで変更できます。

```

1 ## in consumer or provider app (must match)
2 auth.settings.cas_actions['login']='login'
3 auth.settings.cas_actions['validate']='validate'
4 auth.settings.cas_actions['logout']='logout'

```

別ドメインから web2py CAS プロバイダに接続する場合は、許可ドメインのリストに追加する必要があります。

```

1 ## in provider app
2 auth.settings.cas_domains.append('example.com')

```

#### 9.4.1 web2pyを使用した、非 web2py アプリケーションの認可

これは可能ですが、Web サーバに依存します。前提として、2つのアプリケーションは同じ `mod_wsgi` を使った Apache サーバで動作するとします。一方のアプリケーションは `Auth` を使用したアクセス制御を行う web2py アプリケー-

ションです。もう一方は CGI スクリプトの PHP プログラムなどです。後者のアプリケーションにアクセス要求があった時、Web サーバは前者にパーミッションを問い合わせるようにします。

最初に web2py アプリケーションを修正して、次のコントローラを追加する必要があります。

```
1 def check_access():
2     return 'true' if auth.is_logged_in() else 'false'
```

このコントローラから `true` が返ってくる時はユーザはログインしています。`false` の時はそうでない時です。この web2py のプロセスはバックグラウンドで動作しています。

```
1 nohup python web2py.py -a '' -p 8002
```

ポート 8002 は必須です。admin の有効化と admin パスワードは必要ありません。

ここで Apache の設定ファイル（例 “`/etc/apache2/sites-available/default`”）を修正します。非 web2py プログラムの動作時に上記のチェックプログラムを呼び出し、`true` だった場合はレクエストに対するレスポンスを返し、他の場合はアクセス拒否を行なうようにします。web2py と非 web2py はアプリケーションは同一ドメインで実行されます。このためユーザーが web2py アプリケーションにログインしている場合は、他のアプリケーションからの要求であっても web2py のセッションクッキーを Apache に渡すため、認証情報の確認が可能になります。

これを実施するためには、“`web2py/scripts/access.wsgi`” スクリプトが必要になります。web2py では、このスクリプトが付属しています。アクセス制御が必要なアプリケーションの URL とスクリプトの位置を Apache に知らせて、このスクリプトを呼び出す必要があります。

```
1 <VirtualHost *:80>
2   WSGIDaemonProcess web2py user=www-data group=www-data
3   WSGIProcessGroup web2py
4   WSGIScriptAlias / /home/www-data/web2py/wsgihandler.py
5
6   AliasMatch ^myapp/path/needng/authentication/myfile /path/to/myfile
7   <Directory /path/to/>
8     WSGIAccessScript /path/to/web2py/scripts/access.wsgi
9   </Directory>
10 </VirtualHost>
```

ここで “`myapp/path/needng/authentication/myfile`” は、受信レクエストと

web2py フォルダの絶対パス ”/path/to/” をマッチさせる正規表現になっています。

”access.wsgi” スクリプトは次の行を含んでいます。

```
1 URL_CHECK_ACCESS = 'http://127.0.0.1:8002/%(app)s/default/check_access'
```

これは、web2py アプリケーションを指定します。しかし、8002 以外のポートで実行している特定のアプリケーションを指定するよう、編集することが可能です。

また、`check_access()` 機能を変更し、ロジックをより複雑にすることができます。この機能は環境変数を使用し、最初のリクエストの URL を取得することができます

```
1 request.env.request_uri
```

より複雑なルールに実装できます。

```
1 def check_access():
2     if not auth.is_logged_in():
3         return 'false'
4     elif not user_has_access(request.env.request_uri):
5         return 'false'
6     else:
7         return 'true'
```

第3版 - 翻訳: 中垣健志 レビュー: 細田謙二

第4版 - 翻訳: Hitoshi Kato レビュー: Omi Chiba

# 10

## サービス

W3C ではウェブサービスを ”ネットワークを経由したマシン同士の相互通信をサポートするために設計されたソフトウェアシステム” と定義しています。これは広義の意味であり、マシンと人間の通信のために設計されたプロトコルを除く、XML、JSON、RSS などのマシン同士の通信の多くのプロトコルを含んでいます。

この章では web2py を使用したサービスを、どのように公開するか解説します。もしサードパーティサービス (Twitter, Dropbox など) を活用した例に興味がある場合は、9 章及び 14 章を参照してください。web2py は標準で XML、JSON、RSS、CSV、XMLRPC、JSONRPC、AMFRPC、SOAP を含む多くのプロトコルをサポートしています。また、web2py はプロトコルを追加して拡張することもできます。

それぞれのプロトコルは複数の方法でサポートがされており、以下のように区別することにします：

- 指定されたフォーマットで関数の出力をレンダリング（例：XML、JSON、RSS、CSV）
- リモートプロシージャコール（例：XMLRPC、JSONRPC、AMFRPC）

### 10.1 辞書のレンダリング

### 10.1.1 HTML, XML, そして JSON

次のコードを考えてみてください。

```
1 def count():
2     session.counter = (session.counter or 0) + 1
3     return dict(counter=session.counter, now=request.now)
```

このコードは、訪問者がページをリロードするたびに一つずつ増加する counter 値と、現在のページリクエストのタイムスタンプを返します。

このページは、次の URL からリクエストされます。

```
1 http://127.0.0.1:8000/app/default/count
```

ページは、HTML でレンダリングされます。URL に拡張子を追加することで一行のコードも書かずに、web2py に違うプロトコルのページのレンダリングを指示することができます。

```
1 http://127.0.0.1:8000/app/default/count.html
2 http://127.0.0.1:8000/app/default/count.xml
3 http://127.0.0.1:8000/app/default/count.json
```

これらにより返されるレンダリング済みの辞書データは、それぞれ HTML、XML、JSON になります。

XML でのアウトプットです。

```
1 <document>
2     <counter>3</counter>
3     <now>2009-08-01 13:00:00</now>
4 </document>
```

JSON でのアウトプットです。

```
1 { 'counter':3, 'now':'2009-08-01 13:00:00' }
```

date、time、dateime オブジェクトが ISO フォーマットの文字列でレンダリングされている点に注意してください。これは JSON の標準機能というより web2py の仕様です。

### 10.1.2 汎用 (generic) ビュー

例えば、”.xml”拡張子が呼び出された場合、web2pyは”default/count.xml”という名称のテンプレートファイルを探します。もし見つからない場合は、”generic.xml”というテンプレートを参照します。”generic.html”、“generic.xml”、“generic.json”というファイルは、ひな形アプリケーションで用意しています。それ以外の拡張子についても、ユーザーによって簡単に定義可能です。

セキュリティ上の理由から、汎用 (generic) ビューはローカルホスト上のみにアクセスを許可しています。リモートクライアントからのアクセスを有効にするためには、`response.generic_patterns` を設定する必要があります。

ひな形アプリケーションのコピーを使用している場合は、models/db.py にある次の行を編集してください。

- ローカルホストのみにアクセスを制限

```
1 response.generic_patterns = ['*'] if request.is_local else []
```

- 全ての汎用 (generic) ビューへのアクセスを許可

```
1 response.generic_patterns = ['*']
```

- .json のみへアクセスを許可

```
1 response.generic_patterns = ['*.json']
```

`generic_patterns` は glob パターンです（訳注：詳しくは python glob モジュールを参照）。これはアプリケーションのコードとマッチする任意のパターンを使用するか、パターンのリストを渡すことができる意味しています。

```
1 response.generic_patterns = ['*.json', '*.xml']
```

古い web2py アプリケーションでこの機能を利用するには、（バージョン 1.6 以降の）ひな形アプリケーションから、”generic.\*” ファイルをコピーする必要があります。

以下は ”generic.html” のコードです。

```
1 {{extend 'layout.html'}}
2
3 {{=BEAUTIFY(response._vars)}}
```

```

4 <button onclick="document.location='{{=URL("admin", "default", "design", args=request.application)}}'">admin</button>
5 <button onclick="jQuery('#request').slideToggle()">request</button>
6 <div class="hidden" id="request"><h2>request</h2>{{=BEAUTIFY(request)}}
7   }}</div>
8 <button onclick="jQuery('#session').slideToggle()">session</button>
9 <div class="hidden" id="session"><h2>session</h2>{{=BEAUTIFY(session)}}
10  }}</div>
11 <button onclick="jQuery('#response').slideToggle()">response</button>
12 <div class="hidden" id="response"><h2>response</h2>{{=BEAUTIFY(response)}}
13 }}</div>
14 <script>jQuery('.hidden').hide();</script>

```

以下は "generic.xml" のコードです。

```

1 {{{
2 try:
3     from gluon.serializers import xml
4     response.write(xml(response._vars), escape=False)
5     response.headers['Content-Type']='text/xml'
6 except:
7     raise HTTP(405, 'no xml')
8 }}}

```

以下は "generic.json" のコードです。

```

1 {{{
2 try:
3     from gluon.serializers import json
4     response.write(json(response._vars), escape=False)
5     response.headers['Content-Type']='text/json'
6 except:
7     raise HTTP(405, 'no json')
8 }}}

```

python の基本データ型（整数、浮動小数、文字、リスト、タプル、辞書）だけで構成される限り、どのような辞書データでも HTML、XML、JSON でレンダリングできます。response.\_vars には、この機能によって返される辞書データが含まれます。

ユーザー定義データや web2py 独自のオブジェクトが辞書データに含まれる場合、カスタムビューを使ってレンダリングする必要があります。

### 10.1.3 Rows のレンダリング

Rows オブジェクトは select で生成します。生成した Rows を XML、JSON、またはそれ以外のフォーマットでレンダリングするには、最初に `as_list()` メソッドを使用して辞書型のリスト型データ（訳注：リスト型データでフィールド値は辞書型データ）に変換します。

次の例を考えてみます。

```
1 db.define_table('person', Field('name'))
```

次のコードは HTML にレンダリングできますが、XML や JSON にはできません：

```
1 def everybody():
2     people = db().select(db.person.ALL)
3     return dict(people=people)
```

しかしこのコードでは、XML と JSON のレンダリングが可能です。

```
1 def everybody():
2     people = db().select(db.person.ALL).as_list()
3     return dict(people=people)
```

### 10.1.4 カスタムフォーマット

例えば、Python pickle 形式でコードをレンダリングする場合は、

```
1 http://127.0.0.1:8000/app/default/count.pickle
```

単に次のコードを含んだ”default/count.pickle”という、新しいビューファイルを作成するだけです。

```
1 {{{
2 import cPickle
3 response.headers['Content-Type'] = 'application/python.pickle'
4 response.write(cPickle.dumps(response._vars), escape=False)
5 }}}
```

もし任意の関数を Pickle 化ファイルとしてレンダリングしたい場合、上記のファイルを”generic.pickle”という名前で保存するだけです。

ただし、全てのオブジェクトが Pickle 化できるわけではなく、また全ての Pickle 化したオブジェクトを元に戻せるわけではありません。Python の基本データ型

とその組み合わせたものは、問題ありません。また、ファイルストリームやデータベース接続への参照を持たないオブジェクトも、通常は Pickle 化できます。しかし、事前に全ての Pickle 化したオブジェクトのクラスが定義されている状況下でしか、Pickle 化したオブジェクトを元に戻すことができません。

### 10.1.5 RSS

web2py には関数が返した辞書データを RSS フィードとしてレンダリングする、"generic.rss" ビューがあります。

RSS フィードは動作時に固定のデータ構造（タイトル、リンク、説明、アイテム、等々）を持つため、関数によって返される辞書データの値は適切な構造を持つ必要があります。

```

1 { 'title'      : '',
2  'link'       : '',
3  'description': '',
4  'created_on' : '',
5  'entries'    : [] }
```

RSS フィードの entries 項目内の各エントリも、同じような構造を持つ必要があります。

```

1 { 'title'      : '',
2  'link'       : '',
3  'description': '',
4  'created_on' : '' }
```

例えば、次のコードは RSS フィードとしてレンダリングできます。

```

1 def feed():
2     return dict(title="my feed",
3                 link="http://feed.example.com",
4                 description="my first feed",
5                 entries=[
6                     dict(title="my feed",
7                         link="http://feed.example.com",
8                         description="my first feed")
9                 ])
```

単に次の URL にアクセスすることによって動作します。

```
1 http://127.0.0.1:8000/app/default/feed.rss
```

別の方法として、まず次のモデルを想定してみます。

```

1 db.define_table('rss_entry',
2     Field('title'),
3     Field('link'),
4     Field('created_on', 'datetime'),
5     Field('description'))
```

そして次のコードを使えば、RSS フィードとしてレンダリング可能です。

```

1 def feed():
2     return dict(title="my feed",
3                 link="http://feed.example.com",
4                 description="my first feed",
5                 entries=db().select(db.rss_entry.ALL).as_list())
```

Rows オブジェクトの `as_list()` メソッドは、rows を辞書データのリストに変換します。

ここでは明記されていないキー名の追加辞書フィールドがある場合は、無視されます。

次は、web2py の”generic.rss” ビューのコードです。

```

1 {{{
2 try:
3     from gluon.serializers import rss
4     response.write(rss(response._vars), escape=False)
5     response.headers['Content-Type'] = 'application/rss+xml'
6 except:
7     raise HTTP(405, 'no rss')
8 }}}}
```

もう一つの RSS アプリケーションの例として、”slashdot” フィードからデータを収集し新しい web2py RSS フィードを返す、RSS アグリゲータ（RSS リーダ）を考えてみましょう。

```

1 def aggregator():
2     import gluon.contrib.feedparser as feedparser
3     d = feedparser.parse(
4         "http://rss.slashdot.org/Slashdot/slashdot/to")
5     return dict(title=d.channel.title,
6                 link=d.channel.link,
7                 description=d.channel.description,
8                 created_on=request.now,
9                 entries=[
10                     dict(title=entry.title,
```

```

11         link = entry.link,
12         description = entry.description,
13         created_on = request.now) for entry in d.entries])

```

以下からアクセスできます。

```
1 http://127.0.0.1:8000/app/default/aggregator.rss
```

### 10.1.6 CSV

Comma Separated Values (CSV) フォーマットは表形式のデータを表します。

次のモデルを考えてみます。

```

1 db.define_model('animal',
2     Field('species'),
3     Field('genus'),
4     Field('family'))

```

そして次のコード、

```

1 def animals():
2     animals = db().select(db.animal.ALL)
3     return dict(animals=animals)

```

web2py には”generic.csv” がありません。このため animals を CSV にシリアル化するカスタムビューとして、”default/animals.csv” を定義する必要があります。実装例を示します。

```

1 {{{
2 import cStringIO
3 stream=cStringIO.StringIO()
4 animals.export_to_csv_file(stream)
5 response.headers['Content-Type']='application/vnd.ms-excel'
6 response.write(stream.getvalue(), escape=False)
7 }}}

```

”generic.csv” を定義しておくことは可能ですが、シリアル化するプロジェクト名（例では”animals”）を明示する必要があります。このため、”generic.csv” ファイルは web2py では提供しません。

## 10.2 リモートプロシージャコール

web2py はどのような関数でもウェブサービスにする仕組みがあります。ここで言う仕組みとは前述した仕組みと以下の場合で異なります。

- 関数が引数を持つ場合
- 関数がコントローラではなくモデルやモジュールで指定されている場合
- サポートされるべき RPC メソッドを詳細に指定したい場合
- より厳格な URL 命名規則を強制する場合
- 固定プロトコルの組み合わせで動くことで、拡張性は良くないが、以前より高い機能を実現する場合

これらの機能を使うために、

まず最初に、サービスオブジェクトをインポートしてインスタンス化します。

```
1 from gluon.tools import Service
2 service = Service()
```

これはひな形アプリケーション内にある、”db.py” モデルファイル中で既に定義しています。

二番目に、コントローラー内でサービスハンドラを公開します。

```
1 def call():
2     session.forget()
3     return service()
```

これはひな形アプリケーションの ”default.py” コントローラーで既に定義しています。セッションクッキーを使用する場合は、`session.forget()` を削除します。

三番目に、サービスとして公開する関数にデコレータをつける必要があります。次のものが、現在サポートされているデコレータのリストです。

```
1 @service.run
2 @service.xml
3 @service.json
4 @service.rss
5 @service.csv
6 @service.xmlrpc
7 @service.jsonrpc
```

```

8 @service.amfrpc3('domain')
9 @service.soap('FunctionName', returns={'result':type}, args={'param1':
    type, })

```

例として、以下のデコレータ関数を考えてみます。

```

1 @service.run
2 def concat(a,b):
3     return a+b

```

この関数はモデルもしくは、call関数が定義されているコントローラで定義可能です。関数は次の2つの方法で、リモートから実行できます。

```

1 http://127.0.0.1:8000/app/default/call/run(concat?a=hello&b=world)
2 http://127.0.0.1:8000/app/default/call/run(concat/hello/world)

```

httpリクエストは、どちらの方法でも以下の値を返します：

```
1 helloworld
```

@service.xml デコレータを使用する場合は、次のURL経由で実行します。

```

1 http://127.0.0.1:8000/app/default/call/xml(concat?a=hello&b=world)
2 http://127.0.0.1:8000/app/default/call/xml(concat/hello/world)

```

出力は XML で返します。

```

1 <document>
2   <result>helloworld</result>
3 </document>

```

これが DAL Rows オブジェクトだとしても、関数の出力はシリализされます。このケースでは as\_list() を自動で呼び出します。

@service.json デコレータを使用する場合は、次のURL経由で実行します。

```

1 http://127.0.0.1:8000/app/default/call/json(concat?a=hello&b=world)
2 http://127.0.0.1:8000/app/default/call/json(concat/hello/world)

```

出力は JSON で返します。

@service.csv デコレータを使用する場合にサービスハンドラは、リストを含むリストのように、反復可能( iterable )オブジェクトを含む反復可能( iterable )オブジェクトを返す必要があります。例を挙げます。

```

1 @service.csv
2 def table1(a,b):
3     return [[a,b], [1,2]]

```

このサービスは次のどちらかの URL で実行可能です。

```
1 http://127.0.0.1:8000/app/default/call/csv/table1?a=hello&b=world
2 http://127.0.0.1:8000/app/default/call/csv/table1/hello/world
```

そして次の値を返します。

```
1 hello,world
2 1,2
```

`@service.rss` デコレータは、前章で説明した”generic.rss” と同じフォーマットの返り値を期待できます。

それぞれの関数に対して、複数デコレータの設定が可能です。

今までのところ、この章で説明した内容は前章の単なる代替手段でしかありません。サービスオブジェクトの本当の力を発揮するのは、これから説明する XMLRPC、JSONRPC、AMFRPC です。

### 10.2.1 XMLRPC

”default.py” コントローラで、例えば次のコードを考えてみます。

```
1 @service.xmlrpc
2 def add(a,b):
3     return a+b
4
5 @service.xmlrpc
6 def div(a,b):
7     return a/b
```

Python のシェルで実行します。

```
1 >>> from xmlrpclib import ServerProxy
2 >>> server = ServerProxy(
3         'http://127.0.0.1:8000/app/default/call/xmlrpc')
4 >>> print server.add(3,4)
5 7
6 >>> print server.add('hello', 'world')
7 'helloworld'
8 >>> print server.div(12,4)
9 3
10 >>> print server.div(1,0)
11 ZeroDivisionError: integer division or modulo by zero
```

Python xmlrpclib モジュールは、クライアントに XMLRPC プロトコルを提供します。この場合 web2py は、リモートサーバーとして動作します。

クライアントは ServerProxy 経由でサーバーに接続し、サーバー上のデコレータ関数をリモートから実行できます。データ (a, b) は GET/POST をを使った変数を経由するのではなく、XMLRPC プロトコルにより適切にエンコードされて関数に渡されます。そうすることで、データタイプ (int, String、その他) が保持されます。戻り値についても同様です。さらに、サーバー上で発生した例外もクライアントに戻されます。

多くのプログラミング言語 (C、C++、Java、C#、Ruby、Perl) には XMLRPC ライブライアリがあり、これらは互いに相互運用可能です。これは異なるプログラム言語間で、相互に通信をするアプリケーションを作成する場合の最適な方法のひとつです。

XMLRPC クライアントは web2py 内に実装することもできます。そうすることで、ある機能が別の web2py アプリケーション (インストール先が同一であっても) と、XMLRPC を使用し通信することが可能になります。この場合、セッションのデッドロックに注意してください。同一アプリケーション内で XMLRPC を利用したコードを実行する場合は、実行前にセッションロックを開放する必要があります。

```
1 session.forget(response)
```

### 10.2.2 JSONRPC

この章では XMLRPC と同じコード例を使用します。しかしサービスは JSONRPC を代わりに使用します。

```
1 @service.jsonrpc
2 def add(a,b):
3     return a+b
4
5 def call():
6     return service()
```

JSONRPC は XMLRPC と非常に似ています。しかしシリアル化プロトコルは XML の代わりに JSON を使用します。

このサービスはもちろん、どの言語のどのプログラムからも呼び出し可能です。しかしここでは、Python を使用します。web2py は Mariano Reingart によって作成された、”gluon/contrib/simplejsonrpc.py” モジュールを同梱しています。上記のサービス呼び出しをどのように使うか、以下、使用例を示します。

```
1 >>> from gluon.contrib.simplejsonrpc import
2 >>> URL = "http://127.0.0.1:8000/app/default/call/jsonrpc"
3 >>> service = ServerProxy(URL, verbose=True)
4 >>> print service.add(1, 2)
```

### 10.2.3 JSONRPC と Pyjamas

JSONRPC は XMLRPC にとてもよく似ていますが、XML の代わりに JSON ベースのプロトコルを利用してデータをエンコードします。このアプリケーション例として、Pyjamas を使った手法を説明します。Pyjamas は Google Web Toolkit( 当初は Java で書かれていた )の Python 用移植版です。Pyjamas によって Python でクライアントアプリケーションを書くことができます。Pyjamas はコードを JavaScript に変換します。web2py は JavaScript を配信すると共に、クライアントやユーザ操作によるトリガーから発生したリクエストを、AJAX で通信します。

ここでは Pyjamas を web2py 上で、どのように動作するかを説明します。web2py と Pyjamas 以外のライブラリは特に必要ありません。

JSONRPC を使って排他的にサーバーと通信する Pyjamas クライアント( 全て JavaScript )を利用した、シンプルな”todo” アプリケーションを作成していきます。

ステップ1、”todo” アプリケーションを作成します。

ステップ2、”models/db.py” に次のコードを記述します。

```
1 db=DAL('sqlite://storage.sqlite')
2 db.define_table('todo', Field('task'))
3 service = Service()
```

(注意: Service クラスは *gluon.tools* モジュールにあります)

ステップ3、”controllers/default.py” に次のコードを記述します。

```
1 def index():
2     redirect(URL('todoApp'))
```

```

3      @service.jsonrpc
4      def getTasks():
5          todos = db(db.todo).select()
6          return [(todo.task,todo.id) for todo in todos]
7
8
9      @service.jsonrpc
10     def addTask(taskFromJson):
11         db.todo.insert(task= taskFromJson)
12         return getTasks()
13
14     @service.jsonrpc
15     def deleteTask (idFromJson):
16         del db.todo[idFromJson]
17         return getTasks()
18
19     def call():
20         session.forget()
21         return service()
22
23     def todoApp():
24         return dict()

```

それぞれの関数の意味は明らかだと思います。

ステップ4、”views/default/todoApp.html”に次のコードを記述します。

```

1 <html>
2   <head>
3     <meta name="pygwt:module"
4       content="{ {=URL('static', 'output/TodoApp') } }" />
5     <title>
6       simple todo application
7     </title>
8   </head>
9   <body bgcolor="white">
10    <h1>
11      simple todo application
12    </h1>
13    <i>
14      type a new task to insert in db,
15      click on existing task to delete it
16    </i>
17    <script language="javascript"
18      src="{ {=URL('static', 'output/pygwt.js') } }">
19    </script>
20  </body>
21</html>

```

このビューは、まだ作成していない”static/output/todoapp” の Pyjamas コードを実行するだけです。

ステップ5、”static/TodoApp.py”( todoApp でなく TodoApp なことを注意してください) に次のクライアントコードを記述します。

```
1 from pyjamas.ui.RootPanel import RootPanel
2 from pyjamas.ui.Label import Label
3 from pyjamas.ui.VerticalPanel import VerticalPanel
4 from pyjamas.ui.TextBox import TextBox
5 import pyjamas.ui.KeyboardListener
6 from pyjamas.ui.ListBox import ListBox
7 from pyjamas.ui.HTML import HTML
8 from pyjamas.JSONService import JSONProxy
9
10 class TodoApp:
11     def onModuleLoad(self):
12         self.remote = DataService()
13         panel = VerticalPanel()
14
15         self.todoTextBox = TextBox()
16         self.todoTextBox.addKeyboardListener(self)
17
18         self.todoList = ListBox()
19         self.todoList.setVisibleItemCount(7)
20         self.todoList.setWidth("200px")
21         self.todoList.addClickListener(self)
22         self.Status = Label("")
23
24         panel.add(Label("Add New Todo:"))
25         panel.add(self.todoTextBox)
26         panel.add(Label("Click to Remove:"))
27         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
47     if keyCode == KeyboardListener.KEY_ENTER and \
48         sender == self.todoTextBox:
49         id = self.remote.addTask(sender.getText(),self)
50         sender.setText(" ")
51         if id<0:
52             RootPanel().add(HTML("Server Error or Invalid Response"
53                               ))
54
55     def onClick(self, sender):
56         id = self.remote.deleteTask(
57             sender.getValue(sender.getSelectedIndex()),self)
58         if id<0:
59             RootPanel().add(
60                 HTML("Server Error or Invalid Response"))
61
62     def onRemoteResponse(self, response, request_info):
63         self.todoList.clear()
64         for task in response:
65             self.todoList.addItem(task[0])
66             self.todoList.setValue(self.todoList.getItemCount()-1,
67                                   task[1])
68
69     def onRemoteError(self, code, message, request_info):
70         self.Status.setText("Server Error or Invalid Response: " \
71                           + "ERROR " + code + " - " + message)
72
73 class DataService(JSONProxy):
74     def __init__(self):
75         JSONProxy.__init__(self, ".../default/call/jsonrpc",
76                            ["getTasks", "addTask", "deleteTask"])
77
78 if __name__ == '__main__':
79     app = TodoApp()
80     app.onModuleLoad()

```

ステップ6、アプリケーションを実行する前に Pyjamas を起動します：

```

1 cd /path/to/todo/static/
2 python /python/pyjamas-0.5p1/bin/pyjsbuild TodoApp.py

```

これにより Python コードが JavaScript に変換され、ブラウザで実行できるようになります。

このアプリケーションへは、次の URL よりアクセスします。

```
1 http://127.0.0.1:8000/todo/default/todoApp
```

この小節は、Luke Kenneth Casson Leighton ( Pyjamas 開発者) の助けの下、Chris Piron が作成し、Alexei Vinidiktov が更新しました。Pyjamas 0.5p1 でテストされています。このサンプルは Django の [77] を参考にしています。

#### 10.2.4 Amfrpc

AMFRPC は Flash ライアントがサーバーと通信するために使用するリモートプロシージャコール・プロトコルです。web2py は AMFRPC をサポートしていますが、PyAMF ライブラリ導入済みの web2py ソース版が必要です。Linux や Windows のシェルから、次のコマンドでインストールできます。

```
1 easy_install pyamf
```

( 詳細については PyAMF ドキュメントを参考にしてください )

この小節では読者が既に、ActionScript 言語をよく理解していることを前提とします。

2 つの数値を引数とし、それらの足した結果を返すシンプルなサービスを作成します。この web2py アプリケーション名を”pyamf\_test” として、サービス addNumbers を呼び出します。

ステップ 1、Adobe Flash ( MX2004 以降のいずれかのバージョン ) を利用して、Flash FLA ファイルの新規作成からフラッシュクライアント・アプリケーションを作成します。ファイルの最初のフレームに、以下のコードを記述します。

```
1 import mx.remoting.Service;
2 import mx.rpc.RelayResponder;
3 import mx.rpc.FaultEvent;
4 import mx.rpc.ResultEvent;
5 import mx.remoting.PendingCall;
6
7 var val1 = 23;
8 var val2 = 86;
9
10 service = new Service(
11     "http://127.0.0.1:8000/pyamf_test/default/call/amfrpc3",
12     null, "mydomain", null, null);
13
14 var pc:PendingCall = service.addNumbers(val1, val2);
15 pc.responder = new RelayResponder(this, "onResult", "onFault");
```

```

16
17 function onResult(re:ResultEvent):Void {
18     trace("Result : " + re.result);
19     txt_result.text = re.result;
20 }
21
22 function onFault(fault:FaultEvent):Void {
23     trace("Fault: " + fault.fault.faultstring);
24 }
25
26 stop();

```

このコードで Flash クライアントは、”/pyamf\_test/default/gateway” ファイルにある”addNumbers” 関数に対応するサービスに接続することが許されます。Flash のリモート処理を有効にするために、ActionScript バージョン 2 MX リモートクラス をインポートする必要があります。Adobe Flash IDE のクラスパス設定にこれらのクラスのパスを追加するか、単純に新規作成ファイルに”mx” フォルダを追加してください。

サービス構造の引数に注意してください。最初の引数はこれから作成するサービスに対応する URL です。三つ目の引数はサービスのドメイン名です。このドメイン名は”mydomain” とします。

ステップ 2、”txt\_result” というダイナミックフィールド( 訳注: Flash の dynamic text field ) を作成し、ステージ上に配置します。

ステップ 3、上記の Flash クライアントと通信する web2py ゲートウェイをセットアップします。

新しいサービスと Flash クライアント用の AMF ゲートウェイをホストする、 pyamf\_test という web2py アプリケーションを新規作成します。”default.py” コントローラーを編集し、次のコードが含まれるようにします。

```

1 @service.amfrpc3('mydomain')
2 def addNumbers(val1, val2):
3     return val1 + val2
4
5 def call(): return service()

```

ステップ 4、コンパイルし、 pyamf\_test.swf という名称で SWF Flash クライアントをパブリッシュ( export/publish ) します。さらに、新規作成した”pyamf\_test” アプリケーションの”static” フォルダーに、”pyamf\_test.amf”、”pyamf\_test.html”、”AC\_RunActiveContent.js”、”crossdomain.xml”

の各ファイルを設置します。

次の URL で、クライアントのテストができます。

```
1 http://127.0.0.1:8000/pyamf_test/static/pyamf_test.html
```

ゲートウェイはクライアントが addNumbers に接続した際に、バックグラウンドで呼び出されます。

AMF3 の代わりに AMF0 を使用している場合は、同様に次のデコレータを使用できます。

```
1 @service.amfrpc
```

これは次のデコレータの代わりです。

```
1 @service.amfrpc3('mydomain')
```

代わりのデコレータを使用した場合は、サービスの URL も変更になります。

```
1 http://127.0.0.1:8000/pyamf_test/default/call/amfrpc
```

### 10.2.5 SOAP

web2py は、Mariano Reingart が作成した SOAP クライアントとサーバーを含んでいます。XML-RPC とほとんど同じように使うことができます。

”default.py” コントローラーで、次のコードを記述した場合を考えてみます。

```
1 @service.soap('MyAdd', returns={'result':int}, args={'a':int, 'b':int,})
2 def add(a,b):
3     return a+b
```

python シェルで、以下のように実行可能です。

```
1 >>> from gluon.contrib.pysimplesoap.client import SoapClient
2 >>> client = SoapClient(wsdl="http://localhost:8000/app/default/call/
3 >>> soap?WSDL")
4 >>> print client.MyAdd(a=1, b=2)
{'result': 3}
```

返り値が文字列の時に適切なエンコーディングで取得するには、u'proper utf8 text' を指定してください( 訳注 : utf-8 を指定するため文字列の前に u を付ける )

サービスの WSDL は、次の URL で取得可能です。

```
1 http://127.0.0.1:8000/app/default/call/soap?WSDL
```

公開されているメソッドのドキュメントは、次の URL で取得可能です。

```
1 http://127.0.0.1:8000/app/default/call/soap
```

## 10.3 低レベル API とその他のレシピ

### 10.3.1 simplejson

web2py には、Bob Ippolito が開発した gluon.contrib.simplejson が含まれています。このモジュールは最も標準的な、Python-JSON のエンコーダ・デコーダを提供します。

SimpleJSON は、二つの機能から構成されます。

- `gluon.contrib.simplejson.dumps(a)` は Python オブジェクト `a` を JSON にエンコードします。
- `gluon.contrib.simplejson.loads(b)` は JavaScript オブジェクト `b` を Python オブジェクトにデコードします。

シリアル化されるオブジェクト型には、基本型、リスト型、辞書型があります。複合オブジェクトは、ユーザ定義クラスを除きシリアル化可能です。

これは低レベル API を使った、曜日を含む Python のリスト型データをシリアル化する簡単な関数の例（コントローラーは”default.py”）です。

```
1 def weekdays():
2     names=[ 'Sunday', 'Monday', 'Tuesday', 'Wednesday',
3            'Thursday', 'Friday', 'Saturday']
4     import gluon.contrib.simplejson
5     return gluon.contrib.simplejson.dumps(names)
```

下は Ajax リクエストを上記の関数に送信し、JSON メッセージを受信し対応する JavaScript 変数のリストに格納する、HTML ページのサンプルです。

```
1 {{extend 'layout.html'}}
2 <script>
3 $getJSON('/application/default/weekdays',
4           function(data){ alert(data); });
5 </script>
```

このコードは、jQuery の `$.getJSON` 関数を使用しています。この関数は Ajax の呼び出し及び応答時に、ローカルの JavaScript 変数 `data` に曜日名を格納し、その変数をコールバック関数に返します。この例ではコールバック関数がデータを受信したことを、訪問者に単純に知らせます。

### 10.3.2 PyRTF

ウェブサイトでよく必要となる他の機能として、Word で読み取ることが可能なテキスト文書の作成があります。一番簡単な方法は、Rich Text Format (RTF) フォーマットを使用することです。このフォーマットは Microsoft によって開発され、標準フォーマットになりました。web2py には、Simon Cusack によって開発され Grant Edwards によって改良された `gluon.contrib.pyrtf` が含まれます。このモジュールは、色付きのテキストや画像を含む RTF 文書をプログラム的に作成することができます。

次の例では、二つの基本的な RTF クラス `Document` と `Section` をインスタンス化し、後者を前者に追加し、後者にダミーテキストを挿入しています。

```

1 def makertf():
2     import gluon.contrib.pyrtf as q
3     doc=q.Document()
4     section=q.Section()
5     doc.Sections.append(section)
6     section.append('Section Title')
7     section.append('web2py is great. '*100)
8     response.headers['Content-Type']='text/rtf'
9     return q.dumps(doc)

```

`Document` の最後は、`q.dumps(doc)` によってシリアル化されます。RTF 文書を返す前にヘッダーに `content-type` を指定する必要があることに注意してください、そうしないとブラウザはファイルをどのように処理していいか分かりません。

ブラウザの設定に依存しますが、ファイルを保存するかテキストエディタで開くかを聞かれます。

### 10.3.3 ReportLab と PDF

web2py は”ReportLab” [76] という追加ライブラリで、PDF ドキュメントを作成することもできます。web2py ソース版を実行しているのであれば、ReportLab が既にインストールされています。Windows バイナリディストリビューションの場合は、ReportLab を”wb2py/” フォルダで解凍する必要があります。Mac バイナリディストリビューションの場合は、以下のフォルダで解凍することが必要です。

```
1 web2py.app/Contents/Resources/
```

ReportLab がインストールされ、web2py がそれを実行できる状態であるとします。PDF ドキュメントを作成する、”get\_me\_a\_pdf” という簡単な関数を作成してみます。

```
1 from reportlab.platypus import *
2 from reportlab.lib.styles import getSampleStyleSheet
3 from reportlab.rl_config import defaultPageSize
4 from reportlab.lib.units import inch, mm
5 from reportlab.lib.enums import TA_LEFT, TA_RIGHT, TA_CENTER,
6     TA_JUSTIFY
7 from reportlab.lib.colors import colors
8 from uuid import uuid4
9 from cgi import escape
10 import os
11
12 def get_me_a_pdf():
13     title = "This The Doc Title"
14     heading = "First Paragraph"
15     text = 'bla '* 10000
16
17     styles = getSampleStyleSheet()
18     tmpfilename=os.path.join(request.folder, 'private', str(uuid4()))
19     doc = SimpleDocTemplate(tmpfilename)
20     story = []
21     story.append(Paragraph(escape(title), styles["Title"]))
22     story.append(Paragraph(escape(heading), styles["Heading2"]))
23     story.append(Paragraph(escape(text), styles["Normal"]))
24     story.append(Spacer(1,2*inch))
25     doc.build(story)
26     data = open(tmpfilename, "rb").read()
27     os.unlink(tmpfilename)
28     response.headers['Content-Type']='application/pdf'
29     return data
```

`tmpfilename` という仮のファイル名で PDF を作成し、そのファイルから生成した PDF データを読み出し、次にファイルを削除している点に注意してください。ReportLab API についての詳細は、ReportLab ドキュメントを参照してください。段落 や 空白 などを利用できる、ReportLab の Platypus API は特にお勧めです。

## 10.4 Restful Web サービス

REST は ”REpresentational State Transfer” の略語です。Web サービスアーキテクチャの一種であり、SOAP のようなプロトコルのことではありません。実際、REST には標準がありません。

大まかに REST を言い表せば、リソースの集合体によるサービスを考えることができます。各リソースは URL によって識別されます。リソースには 4 つのメソッドがあります。それが、POST (create)、GET (read)、PUT (update)、DELETE です。これらの頭文字を取って、CRUD (生成-読み取り-更新-削除) と略します。クライアントは、リソースを識別する URL と、リソース処理命令である POST/PUT/GET/DELETE といった HTTP メソッドを使用し、HTTP リクエストを組立ててリソースと通信を行います。URL は、指定したプロトコルでエンコーディングするための、例えば `json` といった拡張子を持っていることがあります。

次の URL に対する、POST リクエスト例です。

<sup>1</sup> `http://127.0.0.1/myapp/default/api/person`

これは新しい `person` を作成します。`person` は、`person` テーブルのレコードのことです。しかし他のタイプのリソース（例えばファイル）であってもよいです。

同様に GET リクエストです。

<sup>1</sup> `http://127.0.0.1/myapp/default/api/persons.json`

`json` フォーマットで `persons` のリスト（`person` データのレコード）をリクエストしています。

次の URL 対する、GET リクエストです。

```
1 http://127.0.0.1/myapp/default/api/person/1.json
```

json フォーマットで、`person/1` (`id==1` のレコード) に関連する情報をリクエストしています。

この web2py のリクエストは、次の 3 つのパートに分割することが可能です。

- 最初のパートは、サービスのロケーションを識別。すなわち、サービスを公開している機能。

```
1 http://127.0.0.1/myapp/default/api/
```

- リソースの名前 (`person`、`persons`、`person/1` など)
- 拡張子で指定する通信プロトコル

常にルータ（訳注：web2py の router）を使用し、URL の不要なプレフィックスの除去が可能なことに注意してください。次は簡素化した例です。

```
1 http://127.0.0.1/myapp/default/api/person/1.json
```

これが次のようにになります。

```
1 http://127.0.0.1/api/person/1.json
```

これはテストの問題であり、すでに第 4 章で論じました。

使用例では `api` 機能をコールしています。しかし、これは必ずしも必要ではありません。実際、RESTful サービスで公開する機能は、好きな名前を付けることや、一つだけでなく複数作成することさえ可能です。しかし同様に論じるため、RESTful の機能名は `api` と、継続していくことにします。

同様に次の 2 つのテーブルを前提とします。

```
1 db.define_table('person', Field('name'), Field('info'))
2 db.define_table('pet', Field('owner', db.person), Field('name'), Field('
    info'))
```

これらのリソースを公開します。

最初に RESTful 機能を作成します。

```
1 def api():
2     return locals()
```

拡張子が `request.args` (`request.args` はリソースの識別に使用可能) から除外されているのを除外しないように、また個別のメソッドでハンドルできるよう に、コードを修正します。

```

1 @request.restful()
2 def api():
3     def GET(*args, **vars):
4         return dict()
5     def POST(*args, **vars):
6         return dict()
7     def PUT(*args, **vars):
8         return dict()
9     def DELETE(*args, **vars):
10        return dict()
11    return locals()
```

次に、GET http リクエストを生成します。

```
1 http://127.0.0.1:8000/myapp/default/api/person/1.json
```

呼び出し及び返り値になるのが `GET('person', '1')` です。これは機能中に定義されている GET 関数 (メソッド) です。以下、注意点となります。

- 4つ全てのメソッドを定義する必要はありません。公開したいメソッドだけ定義すればよいです。
- メソッド関数は、名前付き引数を取ることが可能です。
- 拡張子は `request.extension` に格納されます。またコンテンツタイプは自動で設定されます。

`@request.restful()` デコレータは、パスの拡張子が `request.extension` に格納されるのを確認します。さらに、リクエストを機能 (`POST`, `GET`, `PUT`, `DELETE`) の中の一致する関数にマッピングし、`request.args` と `request.vars` をマッピングした関数に渡します。

個々のレコードで、POST 及び GET を行うサービスを次に作成します。

```

1 @request.restful()
2 def api():
3     response.view = 'generic.json'
4     def GET(tablename, id):
5         if not tablename=='person': raise HTTP(400)
6         return dict(person = db.person(id))
7     def POST(tablename, **fields):
```

```

8         if not tablename=='person': raise HTTP(400)
9             return db.person.validate_and_insert(**fields)
10            return locals()

```

注意点：

- GET 及び POST は、別々の関数によって処理します。
- 関数は、正しい引数が渡されることを予想しています（名前付きでない引数は `request.args` で、名前付き引数は `request.vars` で解析されます）
- 入力が正しいことをチェックし、正しくない場合は例外を発生させます。
- GET では `db.person(id)` で Select を実行し、返り値としてレコードを返します。出力は汎用 (generic) ビューが呼び出されるため、自動で JSON に変換します。
- POST では `validate_and_insert(..)` を実行し、返り値として新しいレコードの `id` か、バリデーションエラーを返します。また、POST 関数の `**fields` 変数は、呼び出し時に渡す、post パラメータのことです。

#### 10.4.1 parse\_as\_rest (実験的試み)

ここまで説明したロジックで、 RESTful のどのタイプの Web サービス作成に關しても十分です。しかし web2py には、更に役立つ機能があります。

事実、web2py はデータベーステーブルの公開と、リソースの URL へのマッピング、及びその逆の方法についての記述構文を提供しています。

この機能は、URL パターンを使用します。パターンは、URL からデータベースクエリーにマッピングするリクエスト変数の文字列です。4 種類の核となるパターンタイプがあります。

- 文字列定数。例えば、”friend”
- テーブルに対応する文字列定数。例えば、”friend[person]” は、”person” テーブルを指す URL 下の”friends” にマッチします。
- 変数は条件に使用します。例えば、”{person.id}” は `db.person.name=={person.id}` という条件を適用します。
- フィールド名は ”:field” で表します。

核となるパターンは次のように、”/”を使って複雑な URL パターンにまとめることが可能です。

```
1 "/friend[person]/{person.id}/:field"
```

これにフォームの URL を与えます。

```
1 http://..../friend/1/name
```

person の名前を返す person.id に対するクエリが入っています。”friend[person]” は、”friend” にマッチし、条件としては”person” テーブル。”{person.id}” は ”1” にマッチし、条件として”person.id==1” になります。”:field” は ”name” にマッチします。合わせると次の値を返します。

```
1 db(db.person.id==1).select().first().name
```

複数の URL パターンを、さまざまなタイプのリクエストに対するサービスを提供できる、一つの RESTful 機能のリストにまとめることができます。

DAL はパターンリスト機能を付加する、`parse_as_rest(pattern, args, vars)` メソッドを持っています。`request.args` と `request.vars` はパターンにマッチし、レスポンスを返します（GET のみ）。

次にもっと複雑な例を示します。

```
1
2 @request.restful()
3 def api():
4     response.view = 'generic.'+request.extension
5     def GET(*args, **vars):
6         patterns = [
7             '/friends[person]',
8             '/friend/{person.name.startswith}',
9             '/friend/{person.name}/:field',
10            '/friend/{person.name}/pets[pet.owner]',
11            '/friend/{person.name}/pet[pet.owner]/{pet.name}',
12            '/friend/{person.name}/pet[pet.owner]/{pet.name}/:field'
13        ]
14        parser = db.parse_as_rest(patterns, args, vars)
15        if parser.status == 200:
16            return dict(content=parser.response)
17        else:
18            raise HTTP(parser.status, parser.error)
19    def POST(table_name, **vars):
20        if table_name == 'person':
21            return db.person.validate_and_insert(**vars)
```

```

22     elif table_name == 'pet':
23         return db.pet.validate_and_insert(**vars)
24     else:
25         raise HTTP(400)
26     return locals()

```

これはリスト化パターンに一致する以下の URL を解釈します。

- GET 全ての person レコード

```
1 http://.../api/friends
```

- GET 名前が ”t” で始まる一つの person レコード

```
1 http://.../api/friend/t
```

- GET 名前が ”Tim” で始まる最初の person レコードの ”info” フィールドの値

```
1 http://.../api/friend/Tim/info
```

- GET 上と同じ person レコードの pet のリスト

```
1 http://.../api/friend/Tim/pets
```

- GET person の名前が”Tim” で、 pet の名前が”Snoopy”

```
1 http://.../api/friend/Tim/pet/Snoopy
```

- GET 上と条件の pet の ”info” フィールドの値

```
1 http://.../api/friend/Tim/pet/Snoopy/info
```

この機能では、 2 つの POST の url も同様に公開しています。

- POST 新規 friend
- POST 新規 pet

以下、 ”curl” ユーティリティをインストールしている場合の使用例です。

```

1 $ curl -d "name=Tim" http://127.0.0.1:8000/myapp/default/api/friend.
      json
2 { "errors": {}, "id": 1 }
3 $ curl http://127.0.0.1:8000/myapp/default/api/friends.json
4 { "content": [ { "info": null, "name": "Tim", "id": 1 } ] }
5 $ curl -d "name=Snoopy&owner=1" http://127.0.0.1:8000/myapp/default/api
      /pet.json

```

```

6 {"errors": {}, "id": 1}
7 $ curl http://127.0.0.1:8000/myapp/default/api/friend/Tim/pet/Snoopy.
     json
8 {"content": [{"info": null, "owner": 1, "name": "Snoopy", "id": 1}]}

```

等価かといった単純なクエリではなく、さらに複雑な URL 値を使ったクエリを宣言することも可能です。例えば、次のパターンを設定した場合、

```
patterns = ['friends/{person.name.contains}'
```

次の URL は、

```
1 http://..../friends/i
```

次のクエリと同等になります。

```
1 db.person.name.contains('i')
```

同様に次のパターンを設定した場合、

```
patterns = ['friends/{person.name.ge}/{person.name.gt.not}'
```

次の URL は、

```
1 http://..../friends/aa/uu
```

次のクエリと同等になります。

```
1 (db.person.name>='aa') & (~ (db.person.name>'uu'))
```

パターンでのフィールドに有効な属性は、`contains`, `startswith`, `le`, `ge`, `lt`, `gt`, `eq` (等価, デフォルト), `ne` (非等価) があります。この他、`date` 及び `datetime` フィールド用の特別な属性として、`day`, `month`, `year`, `hour`, `minute`, `second` があります。

注意点として、パターン構文は汎用にデザインしていません。全タイプのクエリをパターンに記述することはできませんが、しかし多くは可能です。構文は将来的に拡張することができます。

条件クエリを実行する、RESTful の URL を公開したい時があります。この場合、特別な引数 `queries` を `parse_as_rest` メソッドに渡すことで実現できます。`queries` は、辞書型の `(tablename, query)` で指定します。`query` には `tablename` テーブルに条件をつけてアクセスするための、DAL のクエリ式を指定します。`order` という GET 変数を使用することで、結果順を変えることもできます。

```
1 http://....api/friends?order=name|~info
```

`order` 変数によって、`name` がアルファベット順、かつ、`info` が逆ソート順になります。

GET 変数の `limit` と `offset` を指定することによって、レコード数の制限も可能です。

```
1 http://....api/friends?offset=10&limit=1000
```

最初の 10 件は飛ばして、1000 件の `friends(persons)` を返します。`limit` のデフォルト値は 1000 です。`offset` のデフォルト値は 0 です。

極端なケースについて考えてみます。全てのテーブル (`auth_` 関係のテーブルは除く) の可能な限り全てのパターンを構築したい。任意のテキストフィールド、整数フィールド、浮動小数点フィールド (範囲による)、日付 (これも範囲による) で、検索できるようにしたい。また、任意のテーブルに POST できるようにしたい、とします。

一般的なケースでは、たくさんのパターンが必要です。しかし web2py は簡単に、これを作成します。

```
1 @request.restful()
2 def api():
3     response.view = 'generic.'+request.extension
4     def GET(*args,**vars):
5         patterns = 'auto'
6         parser = db.parse_as_rest(patterns,args,vars)
7         if parser.status == 200:
8             return dict(content=parser.response)
9         else:
10            raise HTTP(parser.status,parser.error)
11    def POST(table_name,**vars):
12        return db[table_name].validate_and_insert(**vars)
13    return locals()
```

`patterns='auto'` の設定で、web2py は `auth` 関係のテーブル以外の全てのパターンを生成します。また、パターンを照会するパターンもあります。

```
1 http://....api/patterns.json
```

これは次のように、`person` と `pet` テーブルの結果を返します。

```
1 { "content": [
2     "/person[person]",
```

```

3   "/person/{id}/{person.id}",
4   "/person/{id}/{person.id}/:field",
5   "/person/{id}/{person.id}/pet[pet.owner]",
6   "/person/{id}/{person.id}/pet[pet.owner]/id/{pet.id}",
7   "/person/{id}/{person.id}/pet[pet.owner]/id/{pet.id}/:field",
8   "/person/{id}/{person.id}/pet[pet.owner]/owner/{pet.owner}",
9   "/person/{id}/{person.id}/pet[pet.owner]/owner/{pet.owner}/:field",
10  "/person/name/pet[pet.owner]",
11  "/person/name/pet[pet.owner]/id/{pet.id}",
12  "/person/name/pet[pet.owner]/id/{pet.id}/:field",
13  "/person/name/pet[pet.owner]/owner/{pet.owner}",
14  "/person/name/pet[pet.owner]/owner/{pet.owner}/:field",
15  "/person/info/pet[pet.owner]",
16  "/person/info/pet[pet.owner]/id/{pet.id}",
17  "/person/info/pet[pet.owner]/id/{pet.id}/:field",
18  "/person/info/pet[pet.owner]/owner/{pet.owner}",
19  "/person/info/pet[pet.owner]/owner/{pet.owner}/:field",
20  "/pet[pet]",
21  "/pet{id}/{pet.id}",
22  "/pet{id}/{pet.id}/:field",
23  "/pet/owner/{pet.owner}",
24  "/pet/owner/{pet.owner}/:field"
25 ] }

```

次のように、特定のテーブルのみ auto パターンを指定することも可能です。

```
1 patterns = [':auto[person]', ':auto[pet]']
```

#### 10.4.2 smart\_query (実験的試み)

RESTful サービスに次のような、もっと柔軟で自由なクエリーを渡したい時は、

```
1 http://.../api.json?search=person.name starts with 'T' and person.name
   contains 'm'
```

次の設定で可能です。

```

1 @request.restful()
2 def api():
3     response.view = 'generic.'+request.extension
4     def GET(search):
5         try:
6             rows = db.smart_query([db.person, db.pet], search).select()
7             return dict(result=rows)
8         except RuntimeError:
9             raise HTTP(400, "Invalid search string")

```

```

10     def POST(table_name, **vars):
11         return db[table_name].validate_and_insert(**vars)
12     return locals()

```

db.smart\_query メソッドは、2つの引数を取ります。

- クエリの対象のフィールドかテーブルのリスト
- 自然言語で表現したクエリを含む文字列

これによって、該当するレコードの db.set オブジェクトを返します。

検索文字列は評価及び実行するのではなく構文解析されます。このためセキュリティ上のリスクがないことに、注意してください。

#### 10.4.3 アクセス制御

APIへのアクセスはデコレータを使用して、通常と同じように制限することができます。例を示します。

```

1 auth.settings.allow_basic_login = True
2
3 @auth.requires_login()
4 @request.restful()
5 def api():
6     def GET(s):
7         return 'access granted, you said %s' % s
8     return locals()

```

次のようにアクセスが可能です。

```

1 $ curl --user name:password http://127.0.0.1:8000/myapp/default/api/
   hello
2 access granted, you said hello

```

#### 10.5 サービスとアクセス制御

前の章で、次のようなデコレータの使用について説明をしました。

```

1 @auth.requires_login()
2 @auth.requires_membership(...)
3 @auth.requires_permission(...)

```

通常の動作（サービスとしてのデコレータではない）では、出力が HTML 以外の形式でレンダリングされている場合でも、これらのデコレータを使用することができます。

サービスとして定義された関数と、`@service...` や `@auth...` デコレータは使用すべきではありません。異なる二つのタイプのデコレータを、混在させることはできません。もし認証を実施する場合、その `call` 動作ではデコレートする必要があります。

```
1 @auth.requires_login()
2 def call(): return service()
```

複数のサービスオブジェクトをインスタンス化し、同じような別の関数に登録して、そのいくつかを認証付きで、他を認証なしで公開可能であることに注意してください。

```
1 public_services=Service()
2 private_services=Service()
3
4 @public_service.jsonrpc
5 @private_service.jsonrpc
6 def f(): return 'public'
7
8 @private_service.jsonrpc
9 def g(): return 'private'
10
11 def public_call(): return public_service()
12
13 @auth.requires_login()
14 def private_call(): return private_service()
```

これは呼び出し元が、（前章で触れた有効なセッションクッキーや Basic 認証を利用して）HTTP ヘッダーに証明書を渡すことを前提にしています。クライアントはそれをサポートしている必要があります。しかし、一部のクライアントはサポートしていません。

第 3 版 - 翻訳: Omi Chiba レビュー: Yota Ichino

第 4 版 - 翻訳: Hitoshi Kato レビュー: Mitsuhiro Tsuda



# 11

## *jQuery & Ajax*

web2py は主にサーバーサイド開発向けですが、welcome 雜形アプリは基本的な jQuery ライブリ [32]、jQuery カレンダー（日付ピッカー（date picker）、日付時間ピッカー（datetime picker）とクロック（clock））、”superfish.js” メニュー、そしていくつかの jQuery に基づく追加の JavaScript を利用することができます。

Prototype や ExtJS、YUI といったそれ以外の Ajax [78] ライブリも web2py で使うことができますが、jQuery をパッケージすることに決めた理由は、他の同等なライブラリに比べて利用が簡単でかつ強力だからです。また機能的で簡潔であれという web2py 精神をよくとらえていると思うからです。

### 11.1 web2py\_ajax.html

雑形アプリの web2py アプリケーション”welcom” は次のファイルを含みます。

```
1 views/web2py_ajax.html
```

ファイルの中身はこのようになります：

```
1 {{  
2     response.files.insert(0,URL('static','js/jquery.js'))  
3     response.files.insert(1,URL('static','css/calenadar.css'))  
4     response.files.insert(2,URL('static','js/calendar.js'))  
5     response.include_meta()  
6     response.include_files()  
7 }}  
8 <script type="text/javascript"><!--
```

```

9   // These variables are used by the web2py_ajax_init
10  // function in web2py.js (which is loaded below).
11  var w2p_ajax_confirm_message =
12    "{{{=T('Are you sure you want to delete this object?')}}}";
13  var w2p_ajax_date_format = "{{{=T('%Y-%m-%d')}}}";
14  var w2p_ajax_datetime_format = "{{{=T('%Y-%m-%d %H:%M:%S')}}}";
15 //--></script>
16 <script src="{{=URL('static', 'js/web2py.js')}}" type="text/javascript"></script>
17

```

このファイルはデフォルトの”layout.html”のHEADで付け加えられ、次のようなサービスを提供します：

- ”static/jquery.js”を付け加える。
- ”static/calendar.js”と”static/calendar.css”を付け加える。これらはポップアップ・カレンダーで使われます。
- すべての `response.meta` ヘッダを付け加える。
- すべての `response.files`(要求する CSS と JS、コードで宣言されたもの)を付け加える。
- フォーム変数の設定と”static/js/web2y.js”を付け加える

”web2py.js”は次のことを行います：

- (jQuery の`$.ajax`に基づく) `ajax` 関数の定義。
- クラス”error”のDIVの作成、あるいはクラス”flash”のタグ・オブジェクトのスライドダウン(slide down)。
- クラス”integer”のINPUTフィールドで無効な整数入力をエラーにする。
- クラス”doubles”のINPUTフィールドで無効な浮動少数入力をエラーにする。
- タイプ”date”タイプのINPUTフィールドと popup date picker を関連づける。
- タイプ”datetime”タイプのINPUTフィールドと popup datetime picker を関連づける。
- タイプ”time”のINPUTフィールドと popup time picker を関連づける。
- `web2py_ajax_component`を定義する。これはとても重要なツールで、12章で説明します。

- web2py\_comet の定義。この関数は HTML5 の websocket(ウェブソケット)のために用いられます。(この本では説明しませんが、”gluon/contrib/comet\_messaging.py”のソースコードの例をみてください。)

また下位互換性のために popup、collapse、fade 関数を付け加えます。

これはその他のエフェクトがどのように上手く連携しているかの例です。

次のモデルの test アプリケーションを考えてみましょう：

```

1 db = DAL("sqlite://db.db")
2 db.define_table('child',
3     Field('name'),
4     Field('weight', 'double'),
5     Field('birth_date', 'date'),
6     Field('time_of_birth', 'time'))
7
8 db.child.name.requires=IS_NOT_EMPTY()
9 db.child.weight.requires=IS_FLOAT_IN_RANGE(0,100)
10 db.child.birth_date.requires=IS_DATE()
11 db.child.time_of_birth.requires=IS_TIME()
```

この”default.py” コントローラーは：

```

1 def index():
2     form = SQLFORM(db.child)
3     if form.process().accepted:
4         response.flash = 'record inserted'
5     return dict(form=form)
```

そして”default/index.html” ビューは：

```

1 {{extend 'layout.html'}}
2 {{=form}}
```

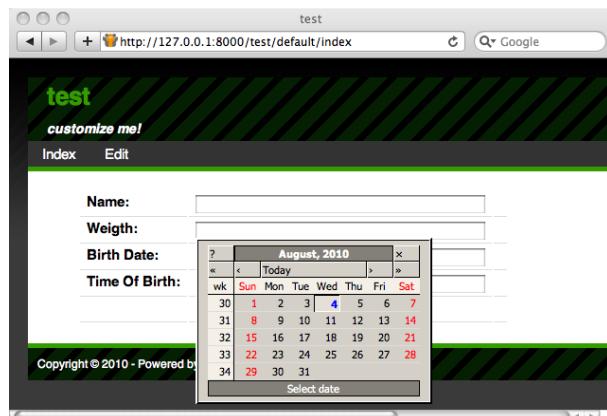
この”index” アクションは次のフォームを生成します：

Name:	<input type="text"/>
Weigh:	<input type="text"/>
Birth Date:	<input type="text"/>
Time Of Birth:	<input type="text"/>
<input type="button" value="Submit"/>	

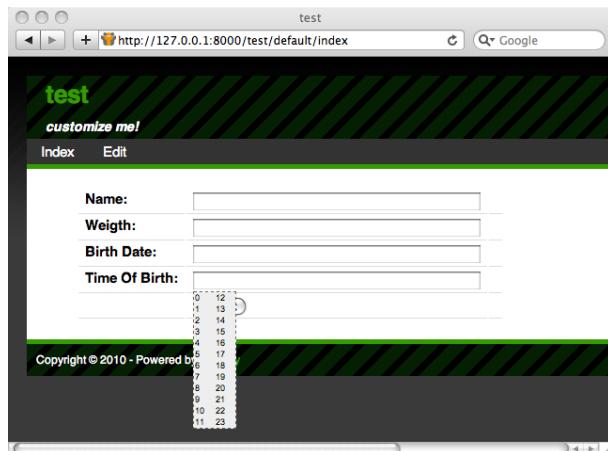
もし無効なフォームが送信されると、サーバーはエラー・メッセージを含む修正されたフォームのページを返します。エラー・メッセージはクラス”error”のDIVです。上記の web2py.jp コードによって、エラーがスライドダウン・エフェクトで表示されます。：

Name:	<input type="text"/>
	<small>enter a value</small>
Weigh:	<input type="text"/>
	<small>enter a number between 0.0 and 100.0</small>
Birth Date:	<input type="text"/>
	<small>enter date and time as 1963-08-28 14:30:59</small>
Time Of Birth:	<input type="text"/>
	<small>enter time as hh:mm:ss (seconds, am, pm optional)</small>
<input type="button" value="Submit"/>	

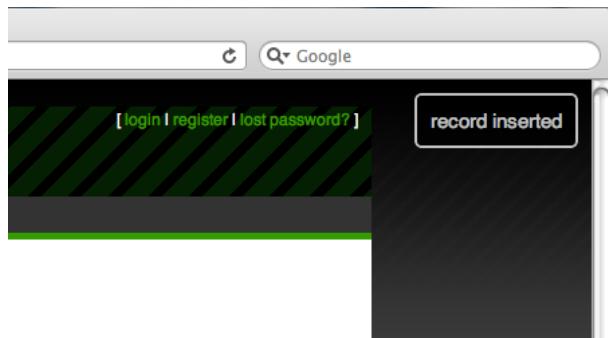
エラーの色は”layout.html” の CSS コードで指定されています。web2py.js コードは入力フィールドへの無効な値の入力を防ぎます。これはサーバーサイドの検証の前に行われます。付け加えますが、サーバーサイドの検証を置き換えるものではありません。web2py.js コードはクラス”date” の INPUT フィールドに入力したとき日付ピッカー (date picker) を表示し、そしてクラス”datetime” の INPUT フィールドに入力したとき日付時間ピッcker (datetime picker) を表示します。例を挙げると：



また web2py.js コードはクラス”time” の INPUT フィールドの編集時に次のような時間ピッcker (time picker) を表示します：



送信時に、コントローラ・アクションは response flash にメッセージ”record inserted”をセットします。デフォルト・レイアウトは id=”flash” の DIV にこのメッセージをレンダリングします。web2py.js コードはページ送信時の DIV の表示、非表示の役割を担っています：



これらの、あるいはそれ以外のエフェクトはビューの中で、またコントローラーの中でヘルパを経由してプログラム的にアクセスできます。

## 11.2 jQuery エフェクト

ここで述べている基本的なエフェクトは追加のファイルを特に必要としません；必要なものは既に web2py\_ajax.html に含まれています。

HTML/XHTML オブジェクトはそれらの要素（例：DIV）クラス、id によって識別することができます。例を挙げると：

```
1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
```

これらはそれぞれ”one”と”two”のクラスに属しています。また、id はそれぞれ”a”と”b”という値です。jQuery では前者を次のような CSS に似た表記で参照できます。

```
1 jQuery('.one') // クラスが".one"の要素を示す
2 jQuery('#a') // id が#a"の要素を示す
3 jQuery('DIV.one') // クラスが".one"の要素 "DIV"を示す
4 jQuery('DIV #a') // がid".one"の要素 "DIV"を示す
```

後者は次のように参照できます

```
1 jQuery('.two')
2 jQuery('#b')
3 jQuery('DIV.two')
4 jQuery('DIV #b')
```

両者を参照するには次のようにします

```
1 jQuery('DIV')
```

タグオブジェクトは”onclick”などのイベントに連携してます。jQuery はこれらのイベントとエフェクトを接続できます。”slideToggle” の例を挙げます：

```
1 <div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
2 <div class="two" id="b">World</div>
```

このとき、”Hello”をクリックすると”World”が消えます。もう一度クリックすると、”World”が再表示されます。hidden クラスを付与するとタグをデフォルトで非表示にできます：

```
1 <div class="one" id="a" onclick="jQuery('.two').slideToggle()">Hello</div>
2 <div class="two hidden" id="b">World</div>
```

タグ自身の外部でアクションとイベントを接続することもできます。前のコードは次のように書き換えることができます：

```

1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
3 <script>
4 jQuery('.one').click(function() {jQuery('.two').slideToggle() });
5 </script>

```

エフェクトは呼び出しオブジェクトを返すので、それらをチェイン(訳注:method chain)させることができます。

`click` にはクリック時に呼び出されるコールバック関数をセットします。`change`、`keyup`、`keydown`、`mouseover`などでも同様です。

ドキュメント全体がロードされた後にだけ JavaScript を実行したいことがよくあります。これは通常は BODY の `onload` 属性で実現されますが、jQuery はレイアウトを編集する必要のない別の方法を提供します：

```

1 <div class="one" id="a">Hello</div>
2 <div class="two" id="b">World</div>
3 <script>
4 jQuery(document).ready(function() {
5     jQuery('.one').click(function() {jQuery('.two').slideToggle() });
6 });
7 </script>

```

この無名関数の本体はドキュメントが完全にロードされ準備ができた時にだけ実行されます。

よく使うイベント名の一覧です：

#### フォームイベント

- `onchange`: 要素の変更時にスクリプトが実行される
- `onsubmit`: フォーム送信時にスクリプトが実行される
- `onreset`: フォームリセット時にスクリプトが実行される
- `onselect`: 要素選択時にスクリプトが実行される
- `onblur`: 要素がフォーカスを失った時にスクリプトが実行される
- `onfocus`: 要素がフォーカスされた時にスクリプトが実行される

#### キーボードイベント

- `onkeydown`: キーが押された時にスクリプトが実行される

- onkeypress: キーが押されて離された時にスクリプトが実行される
- onkeyup: キーが離された時にスクリプトが実行される

### マウスイベント

- onclick: マウスクリック時にスクリプトが実行される
- ondblclick: マウスダブルクリック時にスクリプトが実行される
- onmousedown: マウスボタンが押された時にスクリプトが実行される
- onmousemove: マウスポインタが動いたときにスクリプトが実行される
- onmouseout: マウスポインタが要素の外に動いた時にスクリプトが実行される
- onmouseover: マウスポインタが要素上を動いた時にスクリプトが実行される
- onmouseup: マウスボタンが離された時にスクリプトが実行される

jQuery で定義されているよく使うエフェクトの一覧です：

### エフェクト

- jQuery(...).attr(name): 属性名の値を返します（訳注：原文に誤り？）
  - jQuery(...).attr(name, value): 属性名に値をセットします
  - jQuery(...).show(): オブジェクトを表示します
  - jQuery(...).hide(): オブジェクトを非表示にします
  - jQuery(...).slideToggle(speed, callback): オブジェクトをスライドアップ、あるいはダウンします（訳注：トグル）
  - jQuery(...).slideUp(speed, callback): オブジェクトをスライドアップします
  - jQuery(...).slideDown(speed, callback): オブジェクトをスライドダウンします
  - jQuery(...).fadeIn(speed, callback): オブジェクトをフェードインします
  - jQuery(...).fadeOut(speed, callback): オブジェクトをフェードアウトします
- speed 引数は通常”slow”、”fast” もしくは省略（デフォルト）です。callback はエフェクトが完了した際に実行されるオプション関数です：jQuery エフェクト

は簡単にヘルパに埋め込むこともできます。ビューの例を挙げます：

```
1 { {=DIV('click me!', _onclick="jQuery(this).fadeOut()")}}
```

jQuery はとてもコンパクトで簡潔な Ajax ライブラリです。そのため web2py は（後述する ajax 関数を除き）jQuery の上に追加の抽象化レイヤを必要としません。jQuery API はそれらにネイティブな形で必要な時にアクセスでき手軽に利用できます。

エフェクトやその他の jQuery API についての詳細はドキュメンテーションを参考にしてください。jQuery ライブラリはプラグインとユーザーインターフェースのウィジェットを使って拡張することができます。このトピックはここでは割愛します；参照 [80] に詳細があります。

### 11.2.1 フォームの条件付フィールド

典型的な jQuery エフェクトのアプリケーションはフィールドの値によってその見た目を変えるフォームです。

これは web2py では簡単です。なぜなら、SQLFROM ヘルパが”CSS フレンドリー”なフォームを作成してくれるからです。フォームには列をともなうテーブルがあります。それぞれの列はラベル、入力フィールド、オプション的な第 3 のカラムからなります。フォームの項目はテーブル名とフィールド名で厳密に指定された id を持ります。

慣例として、全ての入力フィールドはテーブル名\_フィールド名という id を持ち、テーブル名\_フィールド名\_列という id を持つ列 (row) の一部になります。

例として、既婚者の場合のみ、納税者の名前とその扶養者の名前を入力するフォームを作成します。

次のモデルを持つテストアプリケーションを作成します：

```
1 db = DAL('sqlite://db.db')
2 db.define_table('taxpayer',
3     Field('name'),
4     Field('married', 'boolean'),
5     Field('spouse_name'))
```

コントローラー”default.py”：

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

```

2     form = SQLFORM(db.taxpayer)
3     if form.process().accepted:
4         response.flash = 'record inserted'
5     return dict(form=form)

```

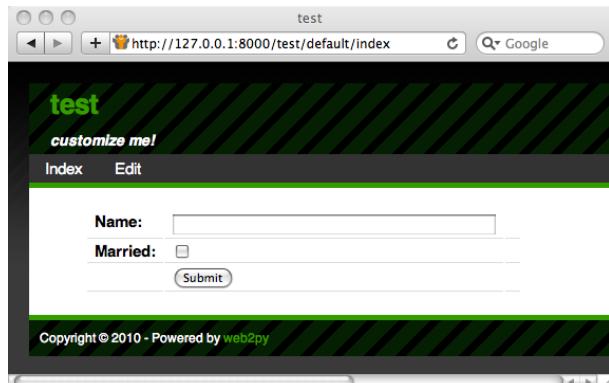
そしてビュー”default/index.html”:

```

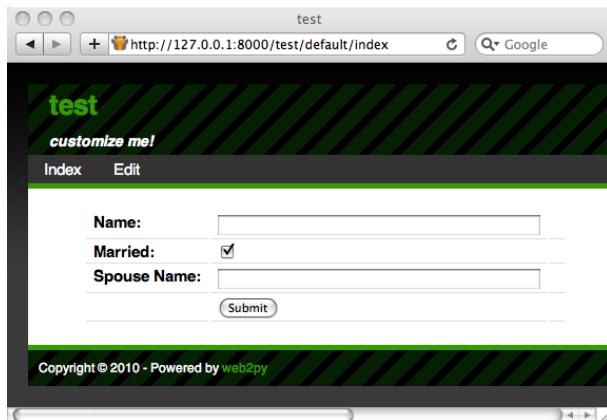
1 {{extend 'layout.html'}}
2 {{=form}}
3 <script>
4 jQuery(document).ready(function() {
5     jQuery('#taxpayer_spouse_name_row').hide();
6     jQuery('#taxpayer_married').change(function() {
7         if(jQuery('#taxpayer_married').attr('checked')) {
8             jQuery('#taxpayer_spouse_name_row').show();
9         } else jQuery('#taxpayer_spouse_name_row').hide();
10    });
11 </script>

```

ビューのスクリプトは扶養者名を含む列 (row) を非表示にするエフェクトを持っています :



納税者が”married” チェックボックスをチェックすると、扶養者名フィールドが再表示されます :



”taxpayer\_married” は ”taxpayer” テーブルの ”married” という ”boolean” フィールドに関連しているチェックボックスです。 ”taxpayer\_spouse\_name\_\_row” は ”taxpayer” テーブルの ”spouse\_name” という入力フィールドを含む列 (row) です。

### 11.2.2 削除の確認

もう一つの役立つアプリケーションは編集フォーム上の削除チェックボックスのように、 ”delete” チェックボックスをチェックしたときに確認を要求するものです。

先ほどの例に次のコントローラーアクションを追加してみましょう：

```

1 def edit():
2     row = db.taxpayer[request.args(0)]
3     form = SQLFORM(db.taxpayer, row, deletable=True)
4     if form.process().accepted:
5         response.flash = 'record updated'
6     return dict(form=form)

```

対応するビュー ”default/edit.html”

```

1 {{extend 'layout.html'}}
2 {{=form}}

```

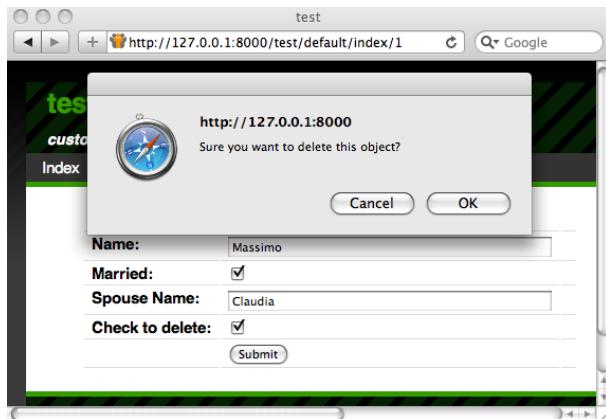
SQLFORM コンストラクタの `deletable=True` 引数は、 web2py が編集フォーム上に ”delete” チェックボックスを表示するようにします。デフォルトは `False` です。 web2py の ”web2py.js” は次のコードを含みます：

```

1  jQuery(document).ready(function(){
2      jQuery('input.delete').attr('onclick',
3          'if(this.checked) if(!confirm(
4              "\{\=T('Sure you want to delete this object?')\}\"))
5              this.checked=false;');
6  });

```

慣例として、このチェックボックスは”delete”クラスを持ちます。上記のjQueryコードはチェックボックスの onclick イベントと確認ダイアログ（JavaScript 標準）を関連付け、納税者が承認しない場合はチェックボックスのチェックを外します：



### 11.3 ajax 関数

web2py.js の中で、web2py は `ajax` という関数を定義しています。これは、jQuery 関数 `$.ajax` を基にしていますが、それと混同しないでください。後者（訳注：jQuery の `$.ajax`）は前者（訳注：web2py.js の `ajax`）より強力で、その使用方法については参照 [32] と参照 [79] を見てください。その一方で、前者は多くの複雑なタスクをこなすのに十分な機能を持ち、簡単に使用できます。

`ajax` 関数は JavaScript の関数で次のような構文になります：

```

1  ajax(url, [name1, name2, ...], target)

```

この関数は、url（第1引数）を非同期的に呼び出し、リスト（第2引数）の name と同じ name を持つフィールドの値を渡し、そしてターゲット（第3引数）と同

じ値の id を持つタグの内部要素に結果を保存します。

default コントローラーの例です：

```

1 def one():
2     return dict()
3
4 def echo():
5     return request.vars.name

```

対応する”default/one.html” ビュー：

```

1 {{extend 'layout.html'}}
2 <form>
3     <input name="name" onkeyup="ajax('echo', ['name'], 'target')"/>
4 </form>
5 <div id="target"></div>

```

INPUT フィールドに何か入力してキーを離す (onkeyup) と同時に、ajax 関数が呼ばれ、name="name" フィールドの値が”echo” アクションに渡されます。このアクションはそのテキストをビューに送り返します。ajax 関数はその結果を受け取り”target” の DIV に echo の結果を表示します。

### 11.3.1 Eval ターゲット

ajax 関数の第 3 引数には文字列”:eval” を指定できます。これは、サーバーから返された文字列がドキュメントに埋め込まれずに、代わりに評価されることを意味します。

default コントローラーの例：

```

1 def one():
2     return dict()
3
4 def echo():
5     return "jQuery('#target').html(%s);" % repr(request.vars.name)

```

対応する”default/one.html” ビュー：

```

1 {{extend 'layout.html'}}
2 <form>
3     <input name="name" onkeyup="ajax('echo', ['name'], ':eval')"/>
4 </form>
5 <div id="target"></div>

```

こうすることで複数のターゲット（訳注：対象）を更新でき、より複雑な結果を返せるようになります。

### 11.3.2 オートコンプリート

web2pyにはFormsの章で説明したように組み込みのオートコンプリート・ウィジェットがあります。ここでは簡単なものをイチから作成してみます。

上記の ajax 関数のもう一つの応用はオートコンプリートです。月名用の入力フィールドを作成し、訪問者が不完全な名前をタイプすると、Ajaxリクエスト経由でオートコンプリートを実施するようにします。応答すると、入力フィールドの下にオートコンプリートのドロップボックスが現れます。

次の default コントローラを経由して実現できます：

```

1 def month_input():
2     return dict()
3
4 def month_selector():
5     if not request.vars.month: return ''
6     months = ['January', 'February', 'March', 'April', 'May',
7                'June', 'July', 'August', 'September', 'October',
8                'November', 'December']
9     month_start = request.vars.month.capitalize()
10    selected = [m for m in months if m.startswith(month_start)]
11    return DIV(*[DIV(k,
12                    _onclick="jQuery('#month').val('%s')" % k,
13                    _onmouseover="this.style.backgroundColor='yellow'"'
14                    ,
15                    _onmouseout="this.style.backgroundColor='white'"'
16                    ) for k in selected])

```

対応する”default/month\_input.html”ビュー：

```

1 {{extend 'layout.html'}}
2 <style>
3 #suggestions { position: relative; }
4 .suggestions { background: white; border: solid 1px #55A6C8; }
5 .suggestions DIV { padding: 2px 4px 2px 4px; }
6 </style>
7
8 <form>
9   <input type="text" id="month" name="month" style="width: 250px" /><br
10  />
11  <div style="position: absolute;" id="suggestions"

```

```

11     class="suggestions">></div>
12 </form>
13 <script>
14 jQuery ("#month").keyup(function() {
15     ajax ('month_selector', [ 'month' ], 'suggestions')) ;
16 </script>

```

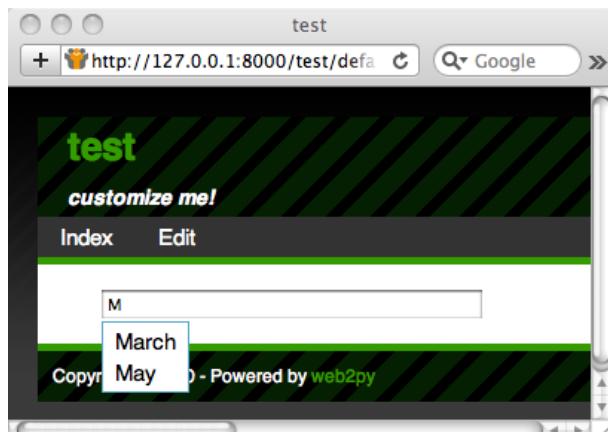
ビジターが”month” 入力フィールドに何かタイプするたびに、ビューの jQuery スクリプトが Ajax リクエストを実行します。入力フィールドの値は Ajax リクエストによって”month\_selector” アクションに送信されます。このアクションは送信された（選択された）テキストで始まる月名のリストを検索し、DIV のリスト（それぞれが候補の月名を含む）を作成し、シリアル化された DIV の文字列を返します。ビューは返された HTML を”suggestions”DIV に表示します。”month\_selector” アクションは候補と DIV に埋め込まれた JavaScript コードの両方を作成します。JavaScript コードはビジターがそれぞれの候補をクリックするたびに実行されます。例えば、ビジターが”M” とタイプすると、コールバックアクションは次の結果を返します：

```

1 <div>
2     <div onclick="jQuery('#month').val('March')"
3         onmouseout="this.style.backgroundColor='white'"
4         onmouseover="this.style.backgroundColor='yellow'">March</div>
5     <div onclick="jQuery('#month').val('May')"
6         onmouseout="this.style.backgroundColor='white'"
7         onmouseover="this.style.backgroundColor='yellow'">May</div>
8 </div>

```

最終的なエフェクトは：



もし月名が次のようにデータベースのテーブルに保存されている場合は：

```
1 db.define_table('month', Field('name'))
```

then simply replace the month\_selector action with:

簡単に month\_selector アクションを次のように置き換えるだけです：

```
1 def month_input():
2     return dict()
3
4 def month_selector():
5     if not request.vars.month:
6         return ''
7     pattern = request.vars.month.capitalize() + '%'
8     selected = [row.name for row in db(db.month.name.like(pattern)).\
9                 select()]
10    return ''.join([DIV(k,
11                         _onclick="jQuery('#month').val('%s')" % k,
12                         _onmouseover="this.style.backgroundColor='yellow'",
13                         _onmouseout="this.style.backgroundColor='white'"\
14                         ).xml() for k in selected])
```

jQuery は追加機能を持つオプションのオートコンプリート・プラグインをもちますが、ここでは説明しません。

### 11.3.3 Ajax フォーム送信

ページ全体をリロードしないで Ajax を利用してメッセージを送信できるページを考えてみましょう。12 章で後述しますが、LOAD ヘルパを利用することで、web2py はここで説明する内容よりもより良い仕組みを提供しています。ここでは jQuery を使ったシンプルなやり方をお見せします。

フォーム”myform”と”target”DIV があります。フォームが送信されると、サーバーはそれを許可（そしてデータベースへの挿入を実行する）するか、拒否（バリデーションをパスしなかったので）します。対応するメッセージが Ajax レスポンスで返され”target”DIV に表示されます。

次のモデルを持つ test アプリケーションを作成してください：

```
1 db = DAL('sqlite://db.db')
2 db.define_table('post', Field('your_message', 'text'))
3 db.post.your_message.requires = IS_NOT_EMPTY()
```

それぞれの投稿は”your\_message”という入力が必須の單一フィールドを持つ点に注意してください。

default.py コントローラを編集して次の二つのアクションを書いてください：

```

1 def index():
2     return dict()
3
4 def new_post():
5     form = SQLFORM(db.post)
6     if form.accepts(request, formname=None):
7         return DIV("Message posted")
8     elif form.errors:
9         return TABLE(*[TR(k, v) for k, v in form.errors.items()])

```

最初のアクションは単にビューを返すだけです。

二つ目のアクションは Ajax のコールバックです。これは request.vars 内のフォーム変数を待ち、それらを処理して、成功時には DIV("Message posted") を返し、失敗時にはエラーメッセージの TABLE を返します。

さあ、”default/index.html” ビューを編集してください：

```

1 {{extend 'layout.html'}}
2
3 <div id="target"></div>
4
5 <form id="myform">
6     <input name="your_message" id="your_message" />
7     <input type="submit" />
8 </form>
9
10 <script>
11 jQuery('#myform').submit(function() {
12     ajax('{{=URL('new_post')}}',
13           ['your_message'], 'target');
14     return false;
15 });
16 </script>

```

この例は HTML を利用して手書きでフォームが作成されていますが、フォームを表示しているのとは異なるアクションで SQLFORM によって処理されている点に注意してください。SQLFORM オブジェクトは HTML にシリアル化されることはありません。この場合は SQLFORM.accepts はセッションを受け取らず formname=None をセットします、これは手書きの HTML フォームにおいてフォーム名とフォームキーをセットしないようにしたからです。

ビューの下部にあるスクリプトは”myform” 送信ボタンとインライン関数を関連付けています。このインライン関数は、web2py の ajax 関数を利用して id="your\_message"を持つ入力フィールドを送信し、id="target"を持つ DIV の中にその答えを表示します。

### 11.3.4 投票と評価

もう一つの Ajax アプリケーションはページにおいてアイテムへ投票 (訳注 : Voting) または評価 (訳注 : Rating) するものです。投稿された画像にビジターが投票できるアプリケーションを考えてみます。そのアプリケーションは投票結果に応じてソートされた画像を表示する単一のページからなります。ここでは、ビジターが複数回投票できるようになっています。ただし、ビジターが認証されていればこのふるまいを変えるのは簡単です。データベースにある個別の投票を追跡して投票者の request.env.remote\_addr に関連付ければいいのです。

簡単なモデルを示します：

```
1 db = DAL('sqlite://images.db')
2 db.define_table('item',
3     Field('image', 'upload'),
4     Field('votes', 'integer', default=0))
```

default コントローラ :

```
1 def list_items():
2     items = db().select(db.item.ALL, orderby=db.item.votes)
3     return dict(items=items)
4
5 def download():
6     return response.download(request, db)
7
8 def vote():
9     item = db.item[request.vars.id]
10    new_votes = item.votes + 1
11    item.update_record(votes=new_votes)
12    return str(new_votes)
```

download アクションは、list\_items ビューが”uploads” フォルダに保存されている画像をダウンロードできるために必要です。vote アクションは Ajax のコールバックで使用されます。

”default/list\_items.html” ビュー :

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

```
2 <form><input type="hidden" id="id" name="id" value="" /></form>
```

```
3 {{for item in items:}}
```

```
4 <p>
```

```
5 
```

```
6 <br />
```

```
7 Votes=<span id="item{{=item.id}}">{{=item.votes}}</span>
```

```
8 [<span onclick="jQuery('#id').val('{{=item.id}}');
```

```
9     ajax('vote', ['id'], 'item{{=item.id}}');">vote up</span>]
```

```
10 </p>
```

```
11 {/pass}}
```

ビジターが”[vote up]”をクリックすると、JavaScript コードは item.id の値を隠している”id”INPUT フィールドに保存し、Ajax リクエストによってサーバーに値を送信します。サーバーは対応するレコードの投票カウンターを増やし、新しい投票カウンターを文字列で返します。そしてこの値が目標の item{{=item.id}} SPAN に挿入されます。

Ajax のコールバックはバックグラウンドでコンピュータ操作の実行に利用できますが、cron もしくは(4章で説明した)バックグラウンド処理の使用を推奨します。web サーバーがスレッドのタイムアウトを強要するからです。もし計算に時間がかかり過ぎると、web サーバーは処理を終了します。タイムアウト値のセット方法はあなたの web サーバーのパラメータを参照してください。

第3版 - 翻訳: Omi Chiba レビュー: 細田謙二

第4版 - 翻訳: Mitsuhiro Tsuda レビュー: Omi Chiba



# 12

## コンポーネントとプラグイン

コンポーネントとプラグインは比較的新しい web2py の機能です。それが何なのか、どうあるべきかに関して開発者の間では意見の相違があります。混乱のほとんどは、他のソフトウェアプロジェクトにおける、これらの用語のさまざまな用途に由来することと、開発者がまだ仕様を完成させるために働いているという事実に由来しています。

しかしながら、プラグインのサポートは重要な機能であり、我々はいくつかの定義を提供する必要があります。これらの定義は最終的なものではなく、単に、本章で説明するプログラミング・パターンに従ったものです。

ここでは次の 2 つの問題を扱いたいと思います。

- サーバー負荷を最小化しコード再利用を最大化するような、モジュール化されたアプリケーションをどのように構築できるのか？
- プラグイン・アンド・プレイ形式のようなコード断片を、どのように配布することができるか？

コンポーネントでは第 1 の問題を、プラグインでは第 2 の問題を扱います。

### 12.1 コンポーネント

コンポーネントは、ウェブページの自律的な機能部品です。

コンポーネントは、モジュール、コントローラ、ビューから構成されています。しかしウェブページに埋め込まれた時、HTML のタグ(例、DIV、SPAN、IFRAME)中に局所化しなければいけないこと、ページの残り部分とは独立してタスクを実行しなければならないこと以外には厳密な要求はありません。ここでは特にページ内でロードされ、Ajax を介してコンポーネントのコントローラ関数と交信を行う、コンポーネントに注目します。

コンポーネントの1つの例は、DIV に含まれている”コメント・コンポーネント”です。これはユーザーコメントと新規投稿コメントフォームを表示します。フォームをサブミットする時、フォームを Ajax を介してサーバーに送ります。さらにリストを更新し、コメントをサーバーサイドのデータベースに保存します。そして、ページの残り部分をリロードすることなく、DIV の中身を更新します。web2py の LOAD 関数は、明示的な JavaScript/Ajax の知識やプログラミンなしに、容易に行うことを可能にします。

私たちの目標は、ページレイアウトにコンポーネントを組み込むことで、Web アプリケーションを開発できるようにすることです。

デフォルトのひな形アプリを拡張子した、”test”という簡単な web2py のアプリを考えます。これは、”models/db\_comments.py” ファイルにおいて、次のようなカスタムモデルを持ちます。

```
1 db.define_table('comment',
2     Field('body', 'text', label='Your comment'),
3     Field('posted_on', 'datetime', default=request.now),
4     Field('posted_by', db.auth_user, default=auth.user_id))
5 db.comment.posted_on.writable=db.comment.posted_on.readable=False
6 db.comment.posted_by.writable=db.comment.posted_by.readable=False
```

”controllers/comments.py” コントローラのコードです。

```
1 @auth.requires_login()
2 def post():
3     return dict(form=crud.create(db.comment),
4                 comments=db(db.comment).select())
```

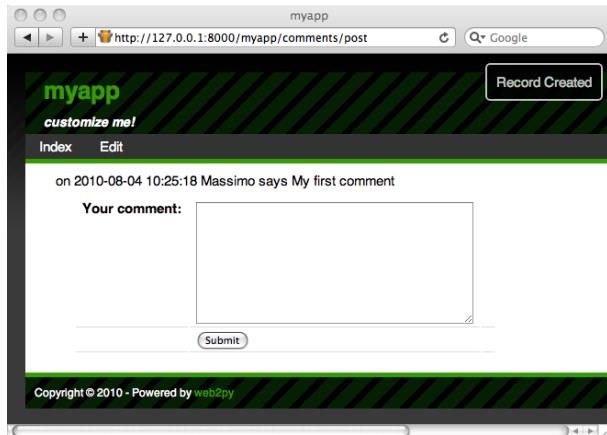
対応する、”views/comments/post.html” ビューです。

```
1 {{extend 'layout.html'}}
2 {{for comment in comments:}}
3 <div class="comment">
4     on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
5     says <span class="comment_body">{{=comment.body}}</span>
6 </div>
```

```
7 {{pass}}
8 {{=form}}
```

通常と同じようにアクセス可能です。

```
1 http://127.0.0.1:8000/test/comments/post
```



ここまで機能では、特別なことは行っていません。しかしレイアウトを拡張しない、“.load”拡張子が付いた新しいビューを定義することにより、コンポーネントに切り替えることが可能です。

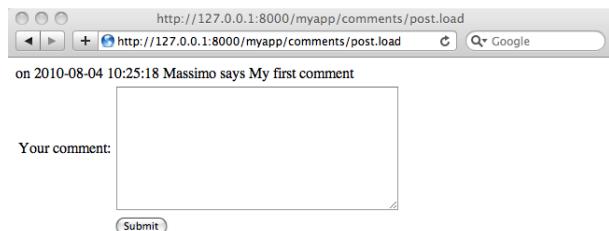
このため、“views/comments/post.load”を作成します。

```
1 {{#extend 'layout.html' <- notice this is commented out!}}
2 {{for comment in comments:}}
3 <div class="comment">
4   on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
5   says <span class="comment_body">{{=comment.body}}</span>
6 </div>
7 {{pass}}
8 {{=form}}
```

次の URL で、アクセス可能です。

```
1 http://127.0.0.1:8000/test/comments/post.load
```

これは次のように表示されます。



このコンポーネントは、次のように任意のページに埋め込むことができます。

```
1 {{=LOAD('comments', 'post.load', ajax=True)}}
```

例えば、”controllers/default.py”を次のように編集します。

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

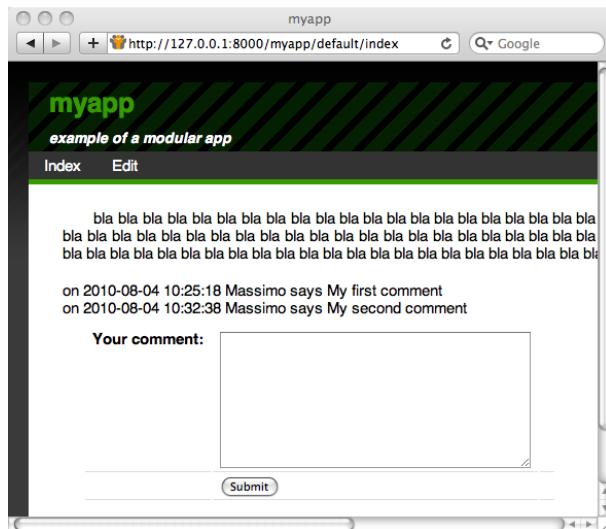
さらにビューについても、コンポーネントを加えます。

```
1 {{extend 'layout.html'}}
2 <p>{{='bla '*100}}</p>
3 {{=LOAD('comments', 'post.load', ajax=True)}}
```

ページにアクセスします。

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

通常のコンテンツとコメントのコンポーネントが表示されます。



{ {=LOAD (...) } } コンポーネントは次のようにレンダリングされます。

```
1 <script type="text/javascript"><!--  
2 web2py_component("/test/comment/post.load", "c282718984176")  
3 //--></script><div id="c282718984176">loading...</div>
```

(実際に生成されるコードは、LOAD 関数に渡すオプションに依存します)

`web2py_component(url, id)` 関数は、”`web2py_ajax.html`” に定義されています。これが、すべての魔法を叶えます。つまり、Ajax を介して `url` を呼び出し、そのレスポンスと対応する `id` を DIV に埋め込みます。これは DIV に対する全てのフォーム送信を捕捉し、Ajax 経由でこれらのフォームを送信します。Ajax のターゲットは、常に DIV そのものになります。

LOAD 関数のすべての引数は以下の通りです。

```
1 LOAD(c=None, f='index', args=[], vars={},
2       extension=None, target=None,
3       ajax=False, ajax_trap=False,
4       url=None, user_signature=False,
5       content='loading...'), **attr):
```

解説

- 最初の 2 つの引数 `c` と `f` は、呼び出すそれぞれのコントローラ及び関数です。
  - `args` と `vars` は関数に渡したい引数と変数です。前者はリスト型、後者は

辞書型です。

- extension は省略可能な拡張子です。なお、拡張子は、`f='index.load'` のように、関数の一部としても渡すことができます。
- target はターゲットとなる DIV の `id` です。指定していない場合、ランダムなターゲット `id` を生成します。
- ajax は、DIV が Ajax 経由で書き込まれる場合に `True` にします。現在のページが返される前に、書き込む必要がある場合は `False` にします (Ajax 呼び出しを回避します)。
- ajax\_trap=True とした場合、DIV 内のどのフォームの送信も捕捉され、Ajax 経由で送信します。そして、レスポンスは DIV 内でレンダリングされている必要があります。`ajax_trap=False` ではフォームは通常通り送信するため、ページ全体がリロードされます。`ajax=True` の場合は、`ajax_trap` の値が何であっても無視されて `True` として扱われます。
- url の指定がある場合、c、f、args、vars、extension の値を上書きし、url のコンポーネントをロードします。これは他のアプリケーション (自身もしくは web2py で作成されていないかもしれません) によって供給されるコンポーネントページとして、ロードするために使用されます。
- user\_signature はデフォルトは `False` です。しかしログインしている場合、`True` に設定すべきです。これは ajax のコールバックが、デジタル署名されていることを確認します。この件は 4 章で触っています。
- content は ajax 呼び出し実行中に表示されるコンテンツです。これは `content=IMG(..)` のようにヘルパーになります。
- オプション引数の `**attr` は、DIV へ属性値を渡すことが可能です。

`.load` ビューが指定されていない場合、処理から返される辞書型データをレイアウトなしにレンダリングする `generic.load` が使用されます。これは、単一のアイテムを保持する辞書型データの場合、最も上手く動作します。

`.load` 拡張子と、他の機能 (例えばログインフォーム) ヘリダイレクトするコントローラを持つコンポーネントを LOAD で使用する場合、`.load` 拡張子は伝搬し、新しい URL (リダイレクト先の一つ) にも `.load` 拡張子がロードされます。

\*以下、注意してください\* Ajax の post は、マルチパートフォーム (multipart

forms)、すなわちファイルアップロードをサポートしていません。このためアップロードフィールドは、LOAD コンポーネントでは機能しません。個別コンポーネントの.load ビューから POST した場合、アップロードフィールドは通常通り機能するので、これが機能するかのように錯覚するかもしれません。代わりに、ajax 互換サードパーティ製ウィジットと、web2py の手動アップロードを行う store コマンドによって、アップロードを行います。

### 12.1.1 クライアント・サーバー コンポーネント通信

コンポーネントの機能が Ajax を経由して呼ばれるとき、web2py はリクエストに 2 つの HTTP ヘッダを渡します。

```
1 web2py-component-location
2 web2py-component-element
```

これは次の変数を介して、アクセスできます。

```
1 request.env.http_web2py_component_location
2 request.env.http_web2py_component_element
```

後者はまた次のようにアクセスできます。

```
1 request.cid
```

前者は、コンポーネント機能を呼び出したページの URL が含まれています。後者は、レスポンスを含んだ DIV の id が含まれています。

コンポーネント機能はまた、2 つの特別な HTTP レスポンスのヘッダにデータを格納します。これらは、レスポンス時にページ全体で解釈されます。次の通りです。

```
1 web2py-component-flash
2 web2py-component-command
```

これらは、次のものを介して設定可能です。

```
1 response.headers['web2py-component-flash']='....'
2 response.headers['web2py-component-command']='....'
```

または（機能がコンポーネントから呼び出されたなら）、自動的に次のようにすることもできます。

```

1 response.flash='...'
2 response.js='...'

```

前者は、レスポンス時にフラッシュさせたいテキストを含みます。後者は、レスポンス時に実行させたい JavaScript を含みます。改行文字を含むことはできません。

例として、ユーザーが質問できるようなコンタクトフォームのコンポーネントを”controllers/contact/ask.py”に定義します。コンポーネントは、システム管理者に質問をメールし、“thank you” メッセージをフラッシュし、ページからそのコンポーネントを取り除きます。

```

1 def ask():
2     form=SQLFORM.factory(
3         Field('your_email', requires=IS_EMAIL()),
4         Field('question', requires=IS_NOT_EMPTY()))
5     if form.process().accepted:
6         if mail.send(to='admin@example.com',
7                     subject='from %s' % form.vars.your_email,
8                     message = form.vars.question):
9             response.flash = 'Thank you'
10            response.js = "jQuery('#%s').hide()" % request.cid
11        else:
12            form.errors.your_email = "Unable to send the email"
13    return dict(form=form)

```

最初の 4 行はフォームを定義し、それを処理します。送信用のメールオブジェクトは、デフォルトのひな形アプリケーションで定義されています。最後の 4 行は、HTTP リクエストヘッダからデータを取得し、HTTP レスポンスヘッダにそれを設定する、コンポーネント固有のロジックを実装しています。

これで次のように、このコンタクトフォームを任意のページに埋め込むことができます。

```

1 {{=LOAD('contact', 'ask.load', ajax=True)}}

```

ask コンポーネントに対して、.load ビューを定義しなかったことに、注目してください。これは単一のオブジェクト (form) を返すため、”generic.load” で十分なためです。汎用ビューは開発ツールであることに注意してください。開発においては、”views/generic.load” を ”views/contact/ask.load” にコピーする必要があります。

`user_signature` 引数を使用し、URL をデジタル署名することにより、Ajax を介して呼び出される関数へのアクセスをブロックすることができます。

```
1 {{=LOAD('contact', 'ask.load', ajax=True, user_signature=True)}}
```

URL に対するデジタル署名を有効にしています。デジタル署名ではコールバック関数に、デコレータを使った認可を必要とします。

```
1 @auth.requires_signature()
2 def ask(): ...
```

### 12.1.2 Ajax トランプリンク

通常、リンクはトランプリンクしません。このため、コンポーネント内のリンクをクリックすると、リンクページ全体がロードされます。しかし、リンクしたページがコンポーネント内にロードするようにしたい場合があります。これは次のように `A` ヘルパを用いて、実現することができます。

```
1 {{=A('linked page', _href='http://example.com', cid=request.cid)}}
```

`cid` が指定された場合、リンクページは Ajax を経由してロードされます。`cid` は、設置するロードページ・コンテンツの `html` 要素の `id` です。この例では `request.cid`、すなわちリンクを生成したコンポーネントの `id` を設定しています。リンクページは通常、URL コマンドを用いて生成した内部 URL にします。

## 12.2 プラグイン

プラグインは、アプリケーションファイルのサブセットです。

そして次のような、いくつかの現実的な特徴があります。

- プラグインはモジュールでなく、モデルでなく、コントローラやビューでもありません。むしろモジュール・モデル・コントローラを含んでおり、ビューは含まれたり、含まれなかったりします。
- プラグインは機能上、自律性がある必要はありません。他のプラグインや特定ユーザのコードに依存していても構いません。

- ・ プラグインは プラグインシステム ではありません。ある程度の独立性を実現するためのルールは決めようとしていますが、登録や独立といった概念は持っていません。
- ・ プラグインは、アプリケーションのためであり、web2py ( フレームワーク ) のためのプラグインではありません。

それでは、なぜプラグインと呼ぶのでしょうか？。それはアプリケーションのサブセットをパッキングし、他のアプリケーション上でアンパッキング(つまりプラグイン)するメカニズムを提供するからです。この定義に基づき、アプリケーション中のいくつかのファイルはプラグインとして扱われます。

アプリケーションが配布される時、プラグインはパックされ、アプリケーションと一緒に配布されます。

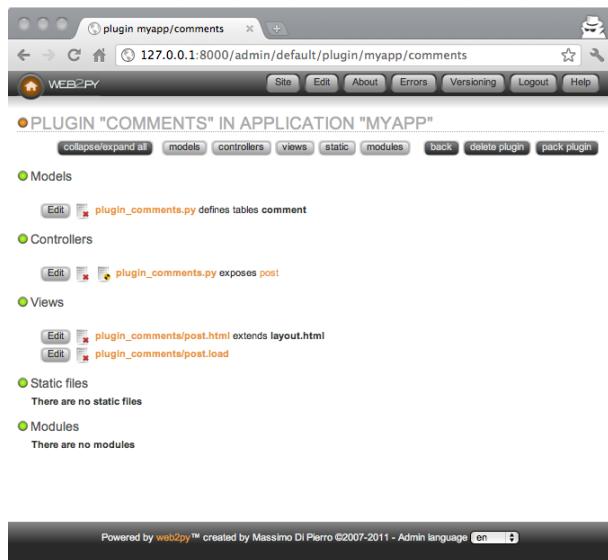
実際に admin はアプリケーションから独立して、パッキングやアンパッキングするためのインターフェイスを提供します。接頭語 `plugin_ name` の名前をもつアプリケーションのファイル及びフォルダは、次のファイルにまとめることができます。

`web2py.plugin.name.w2p`

そして一緒に配布されます。



admin が一緒に配布されることを名前から判断し、さらに別ページに表示することを除き、web2py はプラグインの構成するファイルを別に扱うことはしません。



まだ実際のところ、上記の定義に従い admin よって承認されたようなプラグインよりも、次のようなプラグインが一般的です。

現実的には、私たちは 2 つのタイプのプラグインだけ考えることにします。

- コンポーネントプラグイン。これは前セクションで定義したようなコンポーネントを含んだプラグインです。コンポーネントプラグインは、1 つ以上のコンポーネントを含めることができます。上記の *comments* コンポーネントを含んだ、*plugin\_comments* を例として考えることができます。他の例は、*tagging* コンポーネントと、プラグインによって定義したいいくつかのデータベーステーブルを共有する *tag-cloud* コンポーネントを含む、*plugin\_tagging* です。
- レイアウトプラグイン、これはレイアウトビューと必要な静的ファイルを含んだプラグインです。プラグインを適用するとアプリケーションに新しい、ルック & フィールを提供します。

上記の定義に従って、前セクションで製作したコンポーネントがあります。例えば、”controllers/contact.py” は既にプラグインです。あるアプリケーションで定義したコンポーネントを、他のアプリケーションに持って行き使用することができます。しかしながら、プラグインとしてのラベルが無いため、admin から承

認されていません。これには、解決すべき 2 つの問題があります。

- プラグインファイルを規定に従って命名することで、admin は同じプラグインの属するものとして承認できます。
- モデルファイルがプラグインにある場合、規定を設定することで名前空間を汚さず、互いの衝突も避けることができます。

*name* という名前のプラグインを仮定とします。以下のような従うべきルールがあります。

ルール 1: プラグインのモデルとコントローラはそれぞれ、次のように呼ばれる必要があります。

- `models/plugin_name.py`
- `controllers/plugin_name.py`

フォルダ内にあるプラグインのビュー、モデル、静的ファイル、プライベートファイルは次のように呼ばれる必要があります。

- `views/plugin_name/`
- `modules/plugin_name/`
- `static/plugin_name/`
- `private/plugin_name/`

ルール 2: プラグインのモデルは、次の名前で始まるオブジェクトを定義するだけです。

- `plugin_name`
- `PluginName`
- `_`

ルール 3: プラグインモデルは、次の名前で始まる `session` 変数を定義するだけです。

- `session.plugin_name`
- `session.PluginName`

ルール 4: プラグインはライセンスとドキュメントを含める必要があります。設置は次の場所です。

- static/plugin\_*name*/license.html
- static/plugin\_*name*/about.html

ルール 5: プラグインはひな形の”db.py”で定義された、次のグローバルオブジェクトだけに依存できます。

- db というデータベースのコネクション
- auth という Auth インスタンス
- crud という Crud インスタンス
- service という Service インスタンス

いくつかのプラグインはより高度です。複数の db インスタンスが存在する場合、これらのプラグインはパラメータ設定を持っています。

ルール 6: プラグインにパラメータが必要な場合、以下で説明する PluginManager を使用して設定してください。

上記のルールに従うことによって、次のことが確実になります。

- admin は全ての plugin\_*name* ファイルとフォルダーを、單一エンティティの一部として承認します。
- プラグインは互いに干渉しません。

上記のルールはプラグインのバージョンと依存関係の問題は解決しません。それは機能の範囲を超えていません。

### 12.2.1 コンポーネントプラグイン

コンポーネントプラグインはコンポーネントを定義するプラグインです。コンポーネントは通常、データベースにアクセスし独自モデルを定義します。

以前記述したコードと同じですが、前述のルールに従い、comments コンポーネントを comments\_plugin に変更します。

ステップ 1、“models/plugin\_comments.py” モデルを作成します。

```

1 db.define_table('plugin_comments_comment',
2     Field('body', 'text', label='Your comment'),
3     Field('posted_on', 'datetime', default=request.now),
4     Field('posted_by', db.auth_user, default=auth.user_id))
5 db.plugin_comments_comment.posted_on.writable=False
6 db.plugin_comments_comment.posted_on.readable=False
7 db.plugin_comments_comment.posted_by.writable=False
8 db.plugin_comments_comment.posted_by.readable=False
9
10 def plugin_comments():
11     return LOAD('plugin_comments', 'post', ajax=True)

```

(プラグイン組み込みを単純化する関数定義がある、最後の 2 行を注意のこと)

ステップ 2、“controllers/plugin\_comments.py” を定義します。

```

1 @auth.requires_login()
2 def post():
3     comment = db.plugin_comments_comment
4     return dict(form=crud.create(comment),
5                 comments=db(comment).select())

```

ステップ 3、“views/plugin\_comments/post.load” ビューを作成します。

```

1 {{for comment in comments:}}
2 <div class="comment">
3     on {{=comment.posted_on}} {{=comment.posted_by.first_name}}
4     says <span class="comment_body">{{=comment.body}}</span>
5 </div>
6 {{pass}}
7 {{=form}}

```

admin を使って配布用プラグインをパックできます。Admin はこのプラグインを次のように保存します。

```
1 web2py.plugin.comments.w2p
```

admin の edit ページでプラグインをインストールし、ビューに追加するだけで、どのビューでもプラグインを使用することが可能になります。

```
1 {{=plugin_comments()}}
```

もちろん、パラメータを取得しオプションを設定したコンポーネントを用いることで、より高度なプラグインの作成が可能です。コンポーネントがより複雑になれば、名前の衝突を避けるのが難しくなります。以下で説明するプラグインマネージャーは、この問題を回避するように設計されています。

### 12.2.2 プラグインマネージャー

PluginManager は、gluon.tools で定義されているクラスです。内部でどの様に動作するかを説明する前に、使用方法を説明します。

ここでは前述の comments\_plugin を改良します。また、プラグインのコード変更なしで、カスタマイズを行なってみます。

```
1 db.plugin_comments_comment.body.label
```

次のように改良します。

最初に、次のように”models/plugin\_comments.py”を書き換えます。

```
1 db.define_table('plugin_comments_comment',
2     Field('body', 'text', label=plugin_comments.comments.body_label),
3     Field('posted_on', 'datetime', default=request.now),
4     Field('posted_by', db.auth_user, default=auth.user_id))
5
6 def plugin_comments():
7     from gluon.tools import PluginManager
8     plugins = PluginManager('comments', body_label='Your comment')
9
10    comment = db.plugin_comments_comment
11    comment.label=plugins.comments.body_label
12    comment.posted_on.writable=False
13    comment.posted_on.readable=False
14    comment.posted_by.writable=False
15    comment.posted_by.readable=False
16    return LOAD('plugin_comments', 'post.load', ajax=True)
```

テーブル定義を除くすべてのコードが、单一関数内にカプセル化されていることに、注意してください。また関数が、PluginManager のインスタンスを作成することについても、注意してください。

アプリケーションの他のモデルファイルに、例えば”models/db.py”で、次のようにプラグインの設定をしてください。

```
1 from gluon.tools import PluginManager
2 plugins = PluginManager()
3 plugins.comments.body_label = T('Post a comment')
```

plugins オブジェクトのインスタンス化は、ひな形アプリケーションの”models/db.py”で既に設定されています。

PluginManager オブジェクトは、スレッドレベルのシングルトン ( singleton ) Storage オブジェクトです。これは、同じアプリケーション内で好きなだけインスタンスを作成できますが、しかし (同じ名前を持っていようがいまいが) これらは 1 つの PluginManager インスタンスであるかのように振る舞います。

特に各プラグインファイルは、独自に PluginManager オブジェクトを作成登録し、デフォルトパラメータを設定できます。

```
1 plugins = PluginManager('name', param1='value', param2='value')
```

他の場所( 例えば ”models/db.py” )で、これらのパラメータの上書きが可能です。

```
1 plugins = PluginManager()
2 plugins.name.param1 = 'other value'
```

一箇所で複数のプラグインの設定も可能です。

```
1 plugins = PluginManager()
2 plugins.name.param1 = '...'
3 plugins.name.param2 = '...'
4 plugins.name1.param3 = '...'
5 plugins.name2.param4 = '...'
6 plugins.name3.param5 = '...'
```

プラグインが定義される時、*PluginManager* は引数を取る必要があります。 プラグイン名と、デフォルトパラメータであるオプションの名前付き引数があります。しかし、プラグインが設定されている場合、*PluginManager* コンストラクタは引数を取りません。設定はプラグイン定義の前に行う必要があります (つまり、名前がアルファベット順で先頭のモデルファイルで行う必要があります)。

### 12.2.3 レイアウトプラグイン

レイアウトプラグインは通常コードは含まず、ビューと静的ファイルだけで構成されるため、コンポーネントプラグインより単純です。とはいえ、次のような良いプラクティスに従うべきです。

最初に、”static/plugin\_layout\_name/”(name はレイアウトの名前です) フォルダを作成し、そこに必要な静的ファイルを配置します。

2 番目に、レイアウトと画像リンクを含んだ”views/plugin\_layout\_name/layout.html” レイアウトファイルを作成

し、CSS と JavaScript ファイルを”static/plugin\_layout\_name/” に配置します。

3 番目に、”views/layout.html” を単純に読み込みするように修正します。

```
1 {{extend 'plugin_layout_name/layout.html'}}
2 {{include}}
```

この設計の利点は、このプラグインのユーザが複数レイアウトをインストールできることと、”views/layout.html” を編集するだけで、どのデザインを適用するか選択できることです。その上、”views/layout.html” はプラグインと一緒に admin によりパックされませんので、インストール済みレイアウトのユーザコードをプラグインが上書きする心配がありません。

### 12.3 plugin\_wiki

免責条項: *plugin\_wiki* は、まだ頻繁に開発を行っており、従って、*web2py* のコア関数と同じレベルの後方互換性を約束しません。

*plugin\_wiki* は強化されたプラグインです。この意味することは、複数の有用なコンポーネントを定義することであり、これはアプリケーションの開発方法を変えるかもしれないということです。

次の URL からダウンロード可能です。

```
1 http://web2py.com/examples/static/web2py.plugin.wiki.w2p
```

*plugin\_wiki* のアイデアの背景には、ほとんどのアプリケーションが半分静的なページを含んでいるということです。これらは複雑な、カスタムロジックを含んでいないページです。それらは構造化テキスト（ヘルプページを彷彿とさせるもの）、画像、音楽、ビデオ、crud フォーム、標準コンポーネントのセット（コメント、タグ、チャート、地図）など、を含んでいます。これらのページは公開されているかもしれませんし、ログインや他の認証制限を要求するかもしれません。これらのページはメニューでリンクされるかもしれませんし、ウィザードフォームを通じて辿りつけるだけかもしれません。*plugin\_wiki* は標準的な *web2py* アプリケーションに、この様なページを追加する簡単な方法を提供します。

特に *plugin\_wiki* が提供するのは以下の項目です。

- アプリケーションにページを追加したり、スラグ（固定 URL）での参照を許

可する wiki ライクなインターフェイス。これらのページ (wiki ページとして参照される) はバージョンを持っておりデータベースに保存されます。

- 公開と非公開ページ (ログインを要求)。ページがログインを要求する場合、ユーザが指定グループのメンバーシップを持つことも要求するかもしれません。
- 1, 2, 3 の 3 つのレベル。レベル 1 はテキスト、画像、音楽と動画のみを含むページです。レベル 2 はウィジット (前セクションで説明した、wiki ページに埋め込み可能な定義されたコンポーネント) を含むページです。レベル 3 は、web2py のテンプレートコードも含めることができるページです。
- markmin 構文か、WYSIWYG エディタを使用しての HTML か、ページの編集方法を選択できます。
- ウィジットコレクション。コンポーネントとして実装されています。セルフドキュメント化されおり、普通の web2py ビューに通常コンポーネントとして埋め込んだり、wiki ページに簡略化した構文を使用して埋め込むことができます。
- 特別なページ (`meta-code`、`meta-menu`、など) のセット。これはプラグイン (例えば、プラグインが実行するコードの定義、カスタマイズメニュー、など) をカスタマイズするために使用可能です。

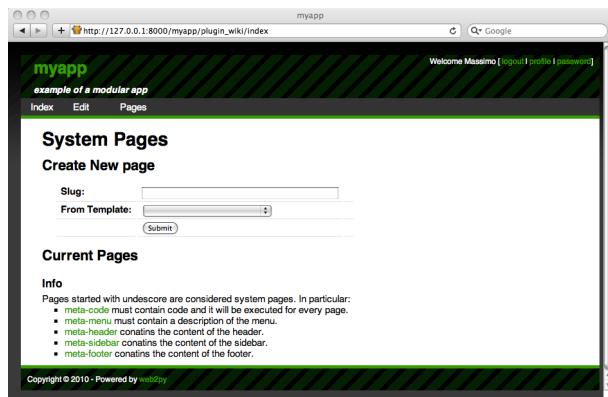
`welcome` に `plugin_wiki` を加えたアプリケーションは、ブログのような単純な Web アプリケーションを構築するための、適切な開発環境と考えることができます。

ここからは、`plugin_wiki` をひな形アプリケーションである `welcome` のコピーに適用させたとして話を進めます。

最初に、プラグインをインストールすると、`pages` と呼ばれる新しいメニューアイテムが追加されることに注意してください。

メニューアイテム `pages` をクリックすると、プラグインの機能にリダイレクトされます。

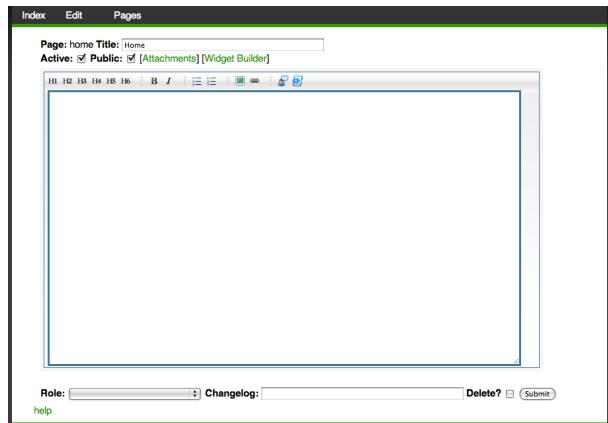
<sup>1</sup> [http://127.0.0.1:8000/myapp/plugin\\_wiki/index](http://127.0.0.1:8000/myapp/plugin_wiki/index)



プラグインのインデックスページは、プラグイン自身を使い作成したページの一覧を表示します。また `slug` を選択することで、新規ページを作成することができます。ここでは、`home` ページを作成してみます。次の URL にリダイレクトします。

1 [http://127.0.0.1:8000/myapp/plugin\\_wiki/page/home](http://127.0.0.1:8000/myapp/plugin_wiki/page/home)

`create page` をクリックし、内容を編集します。



デフォルトではプラグインは、レベル 3 になっています。これはページに、コードと同じようにウィジットの挿入も可能なことを示します。またデフォルトでは、ページ内容を記述するのに、*markmin* 構文を使用します。

### 12.3.1 MARKMIN 構文

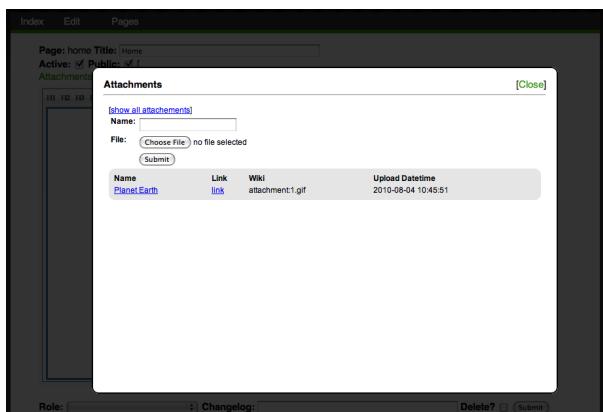
ここでは、markmin 構文の初步を説明します。

markmin	html
# title	<h1>title</h1>
## subtitle	<h2>subtitle</h2>
### subsubtitle	<h3>subsubtitle</h3>
**bold**	<strong>bold</strong>
'italic'	<i>italic</i>
http://...	<a href="http://...com">http:...</a>
http://...png	
http://...mp3	<audio src="http://...mp3"></audio>
http://...mp4	<video src="http://...mp4"></video>
qr:http://...	<a href="http://..."></a>
embed:http://...	<iframe src="http://..."></iframe>

リンク、画像、オーディオ、ビデオのファイルが自動的に埋め込まれていることに注意してください。 MARKMIN 構文の詳細については、第 5 章を参照してください。

ページが存在しない場合、ページを作成するか尋ねられます。

編集ページは、ページに添付ファイルを追加可能です（つまり、静的ファイル）。



これらへのリンクは、次のように記述することができます。

```
1 [[mylink name attachment:3.png]]
```

もしくは埋め込みの場合は、次のように記述します。

```
1 [[myimage attachment:3.png center 200px]]
```

画像サイズ(200px)はオプションです。`center`はオプションではありませんが、`left`や`right`で置き換えることもできます。`blockquoted`(引用ブロック)の埋め込みです。

```
1 -----
2 this is blockquoted
3 -----
```

同じようにテーブルです。

```
1 -----
2 0 | 0 | X
3 0 | X | 0
4 X | 0 | 0
5 -----
```

### 逐語的なテキスト

```
1 ``
2 verbatim text
3 ``
```

---- もしくは `` の最後に、`:class`をオプションで追加することも可能です。`blockquoted` テキストとテーブルは、タグのクラスに変換されます。例えば次のようになります。

```
1 -----
2 test
3 -----:abc
```

次のようにレンダリングされます。

```
1 <blockquote class="abc">test</blockquote>
```

逐語的なテキストのためクラスは、異なるタイプの埋め込みコンテンツに使用することができます。

例えば、`code_language`でプログラム言語を指定することで、シンタックスハイライトしたコードを埋め込むことができます。

```

1 ```
2 def index(): return 'hello world'
3 ```:code_python

```

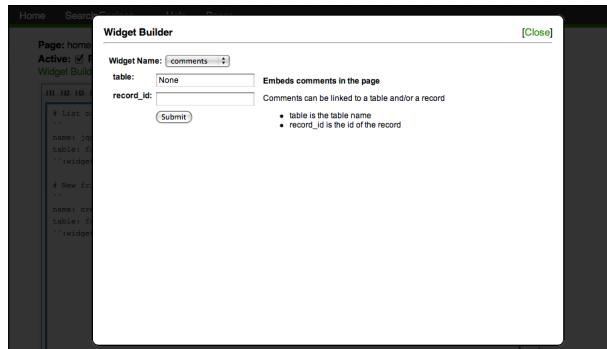
ウィジットの埋め込みも可能です。

```

1 ```
2 name: widget_name
3 attribute1: value1
4 attribute2: value2
5 ```:widget

```

編集ページから”widget builder”をクリックし、リストからウィジットフォームをインタラクティブで追加できます。



(ウィジットのリストは、次のサブセクションを参照のこと) web2py テンプレート言語のコードを埋め込むもできます。

```

1 ```
2 {{for i in range(10)}}<h1>{{=i}}</h1>{{pass}}
3 ```:template

```

### 12.3.2 ページの権限

ページの編集時には、次のフィールドを使用します。

- **active** (デフォルトは `True`)。ページが有効でない場合、訪問者はアクセスが許されません (例えパブリックでも)。
- **public** (デフォルトは `True`)。ページがパブリックの場合、訪問者はログイン無しでアクセスが許されます。

- Role (デフォルトは None)。ページにロールが設定されており、該当するロールのグループメンバーである訪問者がログインしている場合に、ページへのアクセスが許されます。

### 12.3.3 スペシャルページ

`meta-menu` はメニューを含みます。ページが存在しない場合、web2py は “models/menu.py” の `response.menu` 定義を使用します。meta-menu ページの内容は、メニューを上書きします。構文は次のようにになります。

```

1 Item 1 Name http://link1.com
2   Submenu Item 11 Name http://link11.com
3     Submenu Item 12 Name http://link12.com
4     Submenu Item 13 Name http://link13.com
5 Item 2 Name http://link1.com
6   Submenu Item 21 Name http://link21.com
7     Submenu Item 211 Name http://link211.com
8     Submenu Item 212 Name http://link212.com
9   Submenu Item 22 Name http://link22.com
10  Submenu Item 23 Name http://link23.com

```

インデントはサブメニューの構造を決定します。各項目は、メニュー・テキストと一緒に続くリンクで構成されます。リンクは、`page:slug` にすることが可能です。`None` リンクは、どのページにもリンクしません。余計な空白は無視されます。

他の例です。

```

1 Home           page:home
2 Search Engines None
3   Yahoo        http://yahoo.com
4   Google       http://google.com
5   Bing         http://bing.com
6 Help          page:help

```

このレンダリングは次のようになります。



`meta-code` はもう一つのスペシャルページであり、web2py のコードを含む必要があります。これはモデルを拡張したもので、実際ここにモデルコードを書くことが可能です。“models/plugin\_wiki.py” コードが実行される時に、実行が行

われます。

meta-code では、テーブル定義が可能です。

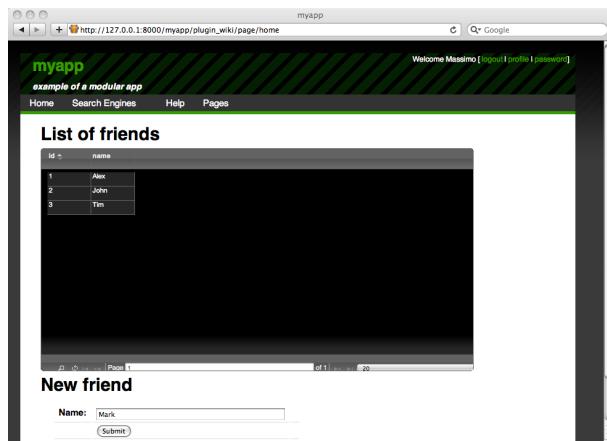
例えば meta-code に次のコードを置くことで、単純な”friends” テーブルを作成できます。

```
1 db.define_table('friend', Field('name', requires=IS_NOT_EMPTY()))
```

そして、選択した下のコードをページに埋め込むことによって、フレンド管理インターフェイスを作成できます。

```
1 ## List of friends
2 ``
3 name: jqgrid
4 table: friend
5 ``:widget
6
7 ## New friend
8 ``
9 name: create
10 table: friend
11 ``:widget
```

ページには、(先頭が#で始まる) ”List of friends” と ”New friend” の2つのヘッダーがあります。さらにページは、フレンドの一覧を表示する jqgrid と、新しいフレンドを追加する crud の、2つのウィジットが(対応するヘッダー下で)含まれています。



`meta-header`, `meta-footer`, `meta-sidebar` は、デフォルトレイアウトの”welcome/views/layout.html” では使用されていません。これらを使用する場合、`admin` を使用して”layout.html” を編集し、適切な場所に次のタグを置きます。

```
1 {{=plugin_wiki.embed_page('meta-header') or ''}}
2 {{=plugin_wiki.embed_page('meta-sidebar') or ''}}
3 {{=plugin_wiki.embed_page('meta-footer') or ''}}
```

これによって、ページコンテンツのレイアウトに、ヘッダー、サイドバー、フッターが表示されます。

#### 12.3.4 `plugin_wiki` の設定

”models/db.py” の他のプラグインと同様に、下記のように記述できます。

```
1 from gluon.tools import PluginManager
2 plugins = PluginManager()
3 plugins.wiki.editor = auth.user.email == mail.settings.sender
4 plugins.wiki.level = 3
5 plugins.wiki.mode = 'markmin' or 'html'
6 plugins.wiki.theme = 'ui-darkness'
```

#### それぞれ

- 現在のログインユーザが `plugin_wiki` ページの編集を許可される場合、`editor` は `true` です。
- `level` は権限です。1 の場合、通常のページの編集です。2 はウィジットの埋め込みです。3 はコードの埋め込みです。
- `mode` は”markmin” のエディタか、WYSIWYG の”html” エディタのどちらを使用するかを決定します。
- `theme` は jQuery UI テーマの名前を指定します。デフォルトではニュートラル色の、”ui-darkness” だけがインストールされています。

次の場所にテーマを追加することができます。

```
1 static/plugin_wiki/ui/%(theme)s/jquery-ui-1.8.1.custom.css
```

### 12.3.5 現在の *widgets*

各ウィジットは、plugin\_wiki ページや通常の web2py テンプレートに埋め込むことができます。

例えば、plugin\_wiki のページに YouTube ビデオを埋め込む場合、次のようにします。

```
1 ````  
2 name: youtube  
3 code: 17AWnfFRc7g  
4 ```:widget
```

同じウィジットを、 web2py ビューに埋め込む場合は、次のようにします。

```
1 {{=plugin_wiki.widget('youtube', code='17AWnfFRc7g') }}
```

どちらの場合でも、次のアウトプットになります。



デフォルト値を持たないウィジット引数が必要です。

現在のすべてのウィジットのリストは、次の通りです。

### 読み込み read

```
1 read(table, record_id=None)
```

レコードを読み込み、表示します。

- `table` テーブル名
- `record_id` レコード番号

### 作成 create

```
1 create(table, message='', next='', readonly_fields='',
2           hidden_fields='', default_fields '')
```

レコード作成フォームの表示です。

- `table` テーブル名
- `message` レコード作成後に表示するメッセージ
- `next` リダイレクト先、例 ”page/index/[id]”
- `readonly_fields` 読み取り専用フィールド、コンマで区切ったフィールドのリスト
- `hidden_fields` 非表示フィールド、コンマで区切ったフィールドのリスト
- `default_fields` フィールドのデフォルト値、コンマで区切った `fieldname=value` のリスト

### 更新 update

```
1 update(table, record_id='', message='', next='',
2         readonly_fields='', hidden_fields='', default_fields '')
```

レコード更新フォームの表示です。

- `table` テーブル名
- `record_id` 更新するレコード、もしくは `{=request.args(-1)}`
- `message` レコード作成後に表示するメッセージ
- `next` リダイレクト先、例 ”page/index/[id]”
- `readonly_fields` 読み取り専用フィールド、コンマで区切ったフィールドのリスト

- `hidden_fields` 非表示フィールド、コンマで区切ったフィールドのリスト
- `default_fields` フィールドのデフォルト値、コンマで区切った `fieldname=value` のリスト

### 選択 select

```
1 select(table,query_field='',query_value='',fields='')
```

### テーブルの全レコードリスト

- `table` レコード名
- `query_field` 及び `query_value` レコードフィルターがある場合のクエリ  
`query_field == query_value` で使用
- `fields` コンマで区切った表示するフィールドのリスト

### 検索 search

```
1 search(table,fields='')
```

レコード検索フォームの表示です。

- `table` レコード名
- `fields` 表示用フィールド、コンマで区切ったフィールドのリスト

### jqgrid

```
1 jqgrid(table,fieldname=None,fieldvalue=None,col_widths='',
2         colnames=None,_id=None,fields='',col_width=80,width=700,height
            =300)
```

### jqGrid プラグインの埋め込み

- `table` テーブル名
- `fieldname, fieldvalue` フィルターオプションで使用 `fieldname==fieldvalue`
- `col_widths` 各項目の幅
- `colnames` 表示する項目名のリスト
- `_id` jqGrid が含まれている TABLE タグの”id”
- `fields` 表示する項目のリスト
- `col_width` デフォルトの項目幅

- height jqGrid の高さ
- width jqGrid の幅

一度 plugin\_wiki をインストールすると、簡単に他のビューにも jqGrid を使用できます。次は使用例です(表示は、yourtable を fk\_id==47 でフィルタしたものです)。

```
1 {{=plugin_wiki.widget('jqgrid', 'yourtable', 'fk_id', 47, '70,150',
2   'Id,Comments',None,'id,notes',80,300,200)}}
```

### latex

```
1 latex(expression)
```

### latex

```
1 latex(expression)
```

LaTeX を埋め込むために、Google charting API を使用します

### 円グラフ

```
1 pie_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

円グラフを埋め込みます。

- data コンマで区切った値のリスト
- names コンマで区切ったラベルのリスト(データアイテム毎に一つ)
- width 画像の幅
- height 画像の高さ
- align 画像の位置を決定

### 棒グラフ

```
1 bar_chart(data='1,2,3',names='a,b,c',width=300,height=150,align='center')
```

棒グラフを埋め込みます。

- data コンマで区切った値のリスト
- names コンマで区切ったラベルのリスト(データアイテム毎に一つ)

- `width` 画像の幅
- `height` 画像の高さ
- `align` 画像の位置を決定

### スライドショー

```
1 slideshow(table, field='image', transition='fade', width=200, height
           =200)
```

スライドショーを埋め込みます。テーブルから画像を取得します。

- `table` テーブル名
- `field` 画像が入ったテーブルのアップロードフィールド
- `transition` 画像切替時のエフェクトタイプ、例 フェードなど
- `width` 画像の幅
- `height` 画像の高さ

### youtube

```
1 youtube(code, width=400, height=250)
```

YouTube ビデオを埋め込みます（コードを使用）

- `code` ビデオのコード
- `width` 画像の幅
- `height` 画像の高さ

### vimeo

```
1 vimeo(code, width=400, height=250)
```

Vimeo ビデオを埋め込みます（コードを使用）

- `code` ビデオのコード
- `width` 画像の幅
- `height` 画像の高さ

### メディアプレイヤー

```
1 mediaplayer(src, width=400, height=250)
```

メディアファイルを埋め込みます(Flash ファイルや MP3 ファイルコードなど)

- `src` ビデオのソース
- `width` 画像の幅
- `height` 画像の高さ

## コメント

```
1 comments(table='None', record_id=None)
```

ページにコメントを埋め込みます。

コメントは、テーブルまたはレコードもしくは両方共に、リンクを張ることが可能です。

- `table` テーブル名
- `record_id` レコードの id

## タグ

```
1 tags(table='None', record_id=None)
```

ページにタグを埋め込みます。

タグは、テーブルまたはレコードもしくは両方共に、リンクを張ることが可能です。

- `table` テーブル名
- `record_id` レコードの id

## タグクラウド

```
1 tag_cloud()
```

タグクラウドを埋め込みます。

## 地図

```
1 map(key='....', table='auth_user', width=400, height=200)
```

Google マップを埋め込みます。

テーブルからマップ上の位置を取得します。

- key Google マップ API のキー (デフォルト動作は 127.0.0.1)
- table テーブル名
- width マップの幅
- height マップの高さ

テーブルには、`latitude`、`longitude`、`map_popup` カラムがある必要があります。目印をクリックした時に、`map_popup` のメッセージが表示されます。

#### インラインフレーム

```
1 iframe(src, width=400, height=300)
```

`<iframe></iframe>` にページを埋め込みます。

#### load\_url

```
1 load_url(src)
```

LOAD 関数を使用して、URL のコンテンツを読み込みます。

#### load\_url

#### load\_action

```
1 load_action(action, controller='', ajax=True)
```

LOAD 関数を使用して、URL (アプリケーション・コントローラ・機能を指定) のコンテンツを読み込みます。

### 12.3.6 ウィジットの拡張

#### 12.3.7 Extending widgets

`plugin_wiki` は ウィジットを追加できます。これは `"models/plugin_wiki_"name` のように、`name` に任意の名前を付けた、新しいモデルファイルを作成することで行います。またこのファイルは次のようなコードを含んでいます。

```
1 class PluginWikiWidgets(PluginWikiWidgets):
2     @staticmethod
3     def my_new_widget(arg1, arg2='value', arg3='value'):
```

```

4      """
5      document the widget
6      """
7      return "body of the widget"

```

1行目は、ウィジットのリストを拡張することを宣言しています。クラスの中に、必要なだけの関数を定義できます。アンダースコアで始まる関数は除き、それぞれの静的関数は新しいウィジットになります。関数はデフォルト値の有無に関係なく、任意の数の引数を取ることが可能です。関数のドキュメンテーション文字列 (docstring) には、markmin 構文自体を使用し関数のドキュメントを記述する必要があります。plugin\_wiki ページにウィジットを埋め込んでいる場合、引数は文字列でウィジットに渡されます。これはウィジット関数の全ての引数は文字列を受け取り、最終的には必要な形式に変換することが必要なことを示しています。必要な文字列表現は決めることができますが、docstring にドキュメント化するようにしてください。

ウィジットは、web2py ヘルパー文字列を返すことができます。後述するケースでは、.xml() でシリアル化されています。

新しいウィジットは、グローバル名前空間で宣言された任意の変数にアクセスできることに注意してください。

この章の最初に作成した”contact/ask” フォームを表示する、新しいウィジットを例として作成します。これは以下のコードを含む、”models/plugin\_wiki\_contact” ファイルを作成することで可能になります。

```

1 class PluginWikiWidgets(PluginWikiWidgets):
2     @staticmethod
3     def ask(email_label='Your email', question_label='question'):
4         """
5             This plugin will display a contact us form that allows
6             the visitor to ask a question.
7             The question will be emailed to you and the widget will
8             disappear from the page.
9             The arguments are
10
11             - email_label: the label of the visitor email field
12             - question_label: the label of the question field
13
14             """
15             form=SQLFORM.factory(
16                 Field('your_email', requires=IS_EMAIL(), label=email_label),

```

```
17     Field('question', requires=IS_NOT_EMPTY()), label=
18         question_label)
19     if form.process().accepted:
20         if mail.send(to='admin@example.com',
21                     subject='from %s' % form.vars.your_email,
22                     message = form.vars.question):
23             command="jQuery('#%s').hide()" % div_id
24             response.flash = 'Thank you'
25             response.js = "jQuery('#%s').hide()" % request.cid
26         else:
27             form.errors.your_email="Unable to send the email"
28     return form.xml()
```

response.render(...) 関数がウィジットにより明示的に呼び出されない限り、*plugin\_wiki* ウィジットはレンダリングしません。

第3版 - 翻訳: 細田謙二 レビュー: Omi Chiba

第4版 - 翻訳: Hitoshi Kato レビュー: Omi Chiba

# 13

## デプロイレシピ

本番環境に web2py をデプロイするには、いくつかの方法があります。詳細は、ホストによって提供されているサーバーの構成とサービスに依存します。

この章では次の点を考えてみます：

- 本番環境 ( Apache、Lighttpd、Cherokee )
- セキュリティ
- スケーラビリティ
- Google App Engine プラットフォーム ( GAE [13] ) へのデプロイ

web2py には SSL [21] が有効な Rocket wsgiserver [22] という web サーバーがあります。これは高速な web サーバーですが、キャッシュ設定に制限があります。これにより web2py を、Apache [82] 、 Lighttpd [89] 、 Cherokee [90] でデプロイするのが最適です。これらのサーバーは無償のオープンソースで、カスタマイズでき、トランザクションの多い本番環境での信頼性が証明されています。静的なファイルを直接表示したり、HTTPS を処理したり、動的なコンテンツで web2py にコントロールを渡すことができます。

数年前まで web サーバーと web アプリケーション間で通信に使う、標準的なインターフェースは Common Gateway Interface (CGI) [81] でした。CGI の一番の問題は、HTTP リクエストのたびに新しいプロセスを作成することです。もし web アプリケーションがインタプリタ言語で書かれている場合は、CGI スクリプトによって実行された HTTP リクエストが新しいインタプリタ・インスタ

ンスを作成します。これは処理が遅く、本番環境では避けるべきです。さらに、CGI は簡単な処理結果だけを扱えます。例えばファイルストリーミングを扱うことはできません。web2py は CGI へのインターフェースに、`cgihandler.py` ファイルを提供しています。

この問題に対する一つの解決策として、Apache の mod\_python モジュールを使う方法があります。mod\_python プロジェクトは Apache Software Foundation における、正式な開発は既に中止されていますが、今だに一般的な方法なのでここで説明することにします。mod\_python は Apache が起動すると Python のインタプリタのインスタンスを開始し、Python を毎回再起動せずに自身のスレッドでそれぞれの HTTP リクエストを処理します。mod\_python は web サーバーと web アプリケーション間の通信に、独自のインターフェイスを使うので最適ではないですが、CGI よりは優れた解決策です。mod\_python ではホストされている全てのアプリケーションが、同じ user\_id/group\_id で実行されるので、セキュリティ上の問題があります。web2py は mod\_python へのインターフェースに、`modpythonhandler.py` ファイルを提供しています。

近年 Python コミュニティは、web サーバーと web アプリケーションを通信する、Python で書かれた新しい標準インターフェイスを開発する方向に進みました。それは Web Server Gateway Interface (WSGI) [17, 18] と呼ばれています。web2py は WSGI 上で構築されており、WSGI が使用できない時には、別のインターフェイスを使用するためのハンドラを提供しています。

Apache は Graham Dumpleton が開発した、mod\_wsgi [88] モジュール経由で WSGI をサポートします。web2py は WSGI へのインターフェースに、`wsgihandler.py` ファイルを提供しています。

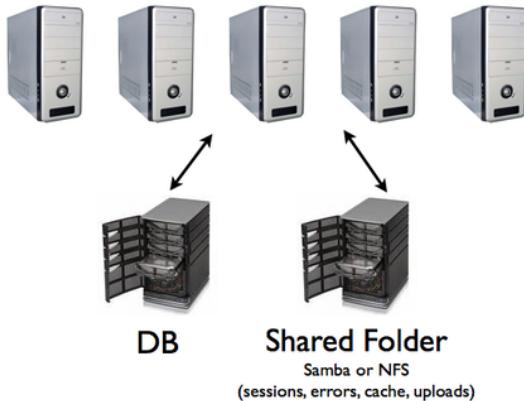
一部の web ホスティングサービスは、mod\_wsgi をサポートしていません。この場合は Apache を proxy として使用し、入ってくる全てのリクエストを web2py 組み込みの web サーバー（例えば localhost:8000 などで動作）に転送する必要があります。mod\_wsgi と mod\_proxy どちらの場合でも、Apache は静的なファイルと SSL 暗号化を直接処理するように設定でき、web2py の負荷を軽減します。

Lighttpd web サーバーは WSGI インターフェースをサポートしませんが、CGI を改良した FastCGI [91] インターフェイスをサポートします。FastCGI の主な目的は、web サーバーと CGI プログラム間のインターフェイスに関するオーバーヘッドを削減することで、サーバーが一度により多くの HTTP リクエストを処理できるようになることです。

Lighttpd web サイトによると、”Lighttpd は YouTube や Wikipedia といった、有名な Web2.0 サイトで使用されています。その高速な IO インフラストラクチャによって、同じハードウェアを使った他の web サーバーと比べても、数倍高いパフォーマンスを実現しています”。FastCGI を利用した Lighttpd は実際、mod\_wsgi を利用した Apache よりも高速です。web2py は FastCGI へのインターフェースに、fcgihandler.py ファイルを提供しています。web2py には Google App Engine (GAE) へのインターフェースである、gaehandler.py もあります。GAE では、web アプリケーションは”クラウド”で動作します。これはフレームワークが、ハードウェアの詳細から、完全に抽象化されていることを意味します。全てのリクエストを処理するのに必要な分だけ、web アプリケーションは自動で複製されます。ここで言う複製とは、単一のサーバー上で複数のスレッドというだけではなく、複数サーバー間でのマルチプロセスという意味です。GAE はファイルシステムへの書き込みを禁止し、全ての永続的なデータは Google BigTable か memcache に保存することで、このスケーラビリティのレベルを実現しています。

GAE 以外のプラットフォームでは拡張性は懸念事項であり、web2py アプリケーションでの調整が必要になる場合があります。拡張性を実現する最も一般的な方法は、ロードバランサ（簡単なラウンドロビン方式やそれぞれのサーバーからハートビートをフィードバックとして受け取るような高度な方式）の後方で、複数の web サーバーを動作させることです。

複数の web サーバーがあったとしても、データベースサーバーは常に一つだけです。デフォルトの web2py は、セッション、エラーチケット、アップロードファイル、キャッシュをファイルシステムに保存します。このためデフォルトの構成では、それらのフォルダは共有フォルダである必要があります：



この章の残りの部分で、この標準のアプローチを改善するいくつかのレシピを考えます。次のようなことを含みます：

- セッションをデータベースやキャッシュに保存します。もしくは全く保存しません。
- チケットをローカルのファイルシステムに保存した後に、バッチでデータベースに移動します。
- cache.ram と cache.disk の代わりに、memcached を利用します。
- アップロードしたファイルは、共有フォルダの代わりにデータベースに保存します。

最初の三つのレシピには従うことをお勧めしますが、一方で四つ目は、サイズの小さなファイルではメリットがありますが、大きなファイルで逆効果になる可能性があります。

### 13.0.8 anyserver.py

tornado:inx twisted:inx wsgiref web2py には、次のポピュラーなサーバーへの WSGI インターフェイスを実装した、anyserver.py ファイルがあります：bjoern, cgi, cherrypy, diesel, eventlet, fapws, flup, gevent, gunicorn, mongrel2, paste, rocket, tornado, twisted, wsgiref

これらのサーバーのいずれかを使えます。例えば Tornado の場合、次のようになります:

```
1 python anyserver.py -s tornado -i 127.0.0.1 -p 8000 -l -P
```

(`-l` はロギング用、`-P` はプロファイル用です。) 全てのコマンドラインオプションの情報を知るには”`-h`” を使ってください:

```
1 python anyserver.py -h
```

## 13.1 Linux と Unix

### 13.1.1 本番デプロイへの第一歩

ここでは apache+python+mod\_wsgi+web2py+postgresql を、ゼロからインストールする手順を説明します。

Ubuntu の場合 : On Ubuntu:

```
1 wget http://web2py.googlecode.com/hg/scripts/setup-web2py-ubuntu.sh
2 chmod +x setup-web2py-ubuntu.sh
3 sudo ./setup-web2py-ubuntu.sh
```

Fedora の場合 : On Fedora:

```
1 wget http://web2py.googlecode.com/hg/scripts/setup-web2py-fedora.sh
2 chmod +x setup-web2py-fedora.sh
3 sudo ./setup-web2py-fedora.sh
```

どちらのスクリプトも標準設定で動作しますが、どの Linux インストールも若干仕様が異なるので、実行する前にこれらのスクリプトのコードを確認してください。Ubuntu については、ほとんどの設定を以下で説明しています。後述のスケーラビリティの最適化については実施していません。

### 13.1.2 Apache セットアップ

このセクションでは、Ubuntu 8.04 Server Edition をプラットフォームとして使用します。設定コマンドは他の Debian ベースの Linux ディストリビューションによく似ていますが、Fedora ベースのシステム (`apt-get` の代わりに `yum` を使う) とは異なります。

初めに次のシェルコマンドを実行し、必要な全ての Python と Apache のパッケージをインストールします：

```

1 sudo apt-get update
2 sudo apt-get -y upgrade
3 sudo apt-get -y install openssh-server
4 sudo apt-get -y install python
5 sudo apt-get -y install python-dev
6 sudo apt-get -y install apache2
7 sudo apt-get -y install libapache2-mod-wsgi
8 sudo apt-get -y install libapache2-mod-proxy-html

```

そして、Apache の SSL モジュール、proxy モジュール、WSGI モジュールを有効にします：

```

1 sudo ln -s /etc/apache2/mods-available/proxy_http.load \
           /etc/apache2/mods-enabled/proxy_http.load
2 sudo a2enmod ssl
3 sudo a2enmod proxy
4 sudo a2enmod proxy_http
5 sudo a2enmod wsgi

```

SSL フォルダを作成し、SSL 証明書をその中に配置します：

```
1 sudo mkdir /etc/apache2/ssl
```

SSL 証明書は verisign.com のような認証局から取得する必要がありますが、テスト目的では、[87] に従うことで自己署名証明書の生成できます。

そして web サーバーを再起動します：

```
1 sudo /etc/init.d/apache2 restart
```

Apache 設定ファイル：

```
1 /etc/apache2/sites-available/default
```

Apache ログ：

```
1 /var/log/apache2/
```

### 13.1.3 mod\_wsgi

上記の web サーバーをインストールしたマシンに、web2py ソースをダウンロードして解凍します。

例えば、web2py を /users/www-data/ 配下にインストールし、www-data ユーザと www-data グループに権限を与えます。この手順は次のシェルコマンドで実行できます：

```
1 cd /users/www-data/
2 sudo wget http://web2py.com/examples/static/web2py_src.zip
3 sudo unzip web2py_src.zip
4 sudo chown -R www-data:www-data /user/www-data/web2py
```

web2py を mod\_wsgi でセットアップするには、新しい Apache 設定ファイルを作成して：

```
1 /etc/apache2/sites-available/web2py
```

次のコードを追加します：

```
1 <VirtualHost *:80>
2   ServerName web2py.example.com
3   WSGIDaemonProcess web2py user=www-data group=www-data \
4                           display-name=%{GROUP}
5   WSGIProcessGroup web2py
6   WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
7
8   <Directory /users/www-data/web2py>
9     AllowOverride None
10    Order Allow,Deny
11    Deny from all
12    <Files wsgihandler.py>
13      Allow from all
14    </Files>
15  </Directory>
16
17  AliasMatch ^/([^\/]+)/static/(.*) \
18        /users/www-data/web2py/applications/$1/static/$2
19  <Directory /users/www-data/web2py/applications/*/*static/>
20    Order Allow,Deny
21    Allow from all
22  </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>
```

Apache を再起動すると、全てのリクエストは Rocket wsgiserver を経由しないで、web2py に渡されます。

説明すると：

```
1 WSGIDaemonProcess web2py user=www-data group=www-data
2           display-name=%{GROUP}
```

上のコードは、”web2py.example.com” に関するデーモンプロセスグループを定義しています。バーチャルホスト内にこれを定義することで、バーチャルホストは WSGIProcessGroup を使用してアクセスできます。これは同じサーバー名のポートが違うバーチャルホストも含みます。”user” と”group” オプションには、web2py をセットアップしたディレクトリへの書き込み権限があるユーザをセットするべきです。Apache を実行するデフォルトユーザに、web2py インストレーション・ディレクトリへの書き込み権限を設定している場合、”user” と”group” を設定する必要はありません。”display-name” オプションは、PS コマンドによって表示する実行可能な Apache web サーバーのプロセス名の代わりに、”(wsgi:web2py)” のように出力する設定をします。”processes” や”threads” オプションを指定しないことで、デーモンプロセスグループはプロセス内で 15 個のスレッドを実行する単一のプロセスを持ちます。これは通常のほとんどのサイトで十分過ぎる設定なので、そのまま使用するべきです。設定を上書きする場合、”wsgi.multiprocess” フラグをチェックする全てのプラウザ内 WSGI デバッグツールが無効になるため、”processes=1” は使用しないでください。これはどの”processes” オプションの使用でも、例え單一プロセスでさえも、フラグを true にセットしてしまいます。それらのツールではフラグが false にセットされていることが前提になります。注意：もしアプリケーションコードやサードパーティ性の拡張モジュールがスレッドセーフでない場合は、”processes=5 threads=1” オプションを使用してください。これは個々のプロセスがシングルスレッドであるデーモンプロセスグループに、五つのプロセスを作成します。ガベージコレクトを適切に実施できずに、アプリケーションが Python オブジェクトをリークする場合は、”maximum-requests=1000” を使うことも検討してください。

```
1 WSGIProcessGroup web2py
```

全ての WSGI アプリケーションの実行を、WSGIDaemonProcess ディレクティブで設定されたデーモンプロセスグループに委譲します。

```
1 WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
```

web2py アプリケーションをマウントします。この場合は web サイトの root にマウントされます。

```
1 <Directory /users/www-data/web2py>
2 ...
3 </Directory>
```

Apache に WSGI スクリプトファイルにアクセスする権限を与えます。

```
1 <Directory /users/www-data/web2py/applications/*/*>
2   Order Allow,Deny
3   Allow from all
4 </Directory>
```

静的ファイルを検索する際に、web2py をバイパスするように Apache を設定します。

```
1 <Location /admin>
2   Deny from all
3 </Location>
```

及び、

```
1 <LocationMatch ^/([^\/]+)appadmin>
2   Deny from all
3 </LocationMatch>
```

admin と appadmin へのパブリックアクセスをブロックします。

通常は WSGI スクリプトがあるディレクトリ全体にアクセスを許可するだけでも良いですが、web2py は管理者パスワードを含む他のソースコードが入っているディレクトリに、WSGI スクリプトを配置します。技術的には Apache は、マッピングされた URL 経由してディレクトリを通過する全ユーザに対して、全てのファイルを提供する権限を与えます。このため、ディレクトリ全体を公開することは、セキュリティ上の問題を引き起こします。セキュリティ問題を回避するため、WSGI スクリプトファイルを除くディレクトリのファイルへのアクセスを明示的に拒否し、より安全にするためにユーザーによる .htaccess ファイルの上書きを禁止します。

以上をまとめた、コメントの入った Apache wsgi 設定ファイルは以下にあります：

```
1 scripts/web2py-wsgi.conf
```

このセクションは、mod\_wsgi 開発者の Graham Dumpleton の協力を得て作成されました。

### 13.1.4 mod\_wsgi と SSL

アプリケーション（例えば admin と appadmin）で強制的に HTTPS を利用させるには、SSL 証明書と秘密鍵を以下に保存し：

```
1 /etc/apache2/ssl/server.crt
2 /etc/apache2/ssl/server.key
```

Apache の設定ファイル web2py.conf を編集し、次のコードを追加します：

```
1 <VirtualHost *:443>
2   ServerName web2py.example.com
3   SSLEngine on
4   SSLCertificateFile /etc/apache2/ssl/server.crt
5   SSLCertificateKeyFile /etc/apache2/ssl/server.key
6
7   WSGIProcessGroup web2py
8
9   WSGIScriptAlias / /users/www-data/web2py/wsgihandler.py
10
11  <Directory /users/www-data/web2py>
12    AllowOverride None
13    Order Allow,Deny
14    Deny from all
15    <Files wsgihandler.py>
16      Allow from all
17    </Files>
18  </Directory>
19
20  AliasMatch ^/([^\/]+)/static/(.*) \
21    /users/www-data/web2py/applications/$1/static/$2
22
23  <Directory /users/www-data/web2py/applications/*/*>
24    Order Allow,Deny
25    Allow from all
26  </Directory>
27
28  CustomLog /private/var/log/apache2/access.log common
29  ErrorLog /private/var/log/apache2/error.log
30
31 </VirtualHost>
```

Apache を再起動すると、次にアクセスできますが：

```
1 https://www.example.com/admin
2 https://www.example.com/examples/appadmin
3 http://www.example.com/examples
```

次のものにはアクセスできません：

```
1 http://www.example.com/admin
2 http://www.example.com/examples/appadmin
```

### 13.1.5 mod\_proxy

幾つかの Unix/Linux ディストリビューションは Apache を実行できますが、mod\_wsgi をサポートしていないものがあります。この場合の一一番簡単な解決策は、Apache をプロキシとして実行し、静的ファイルのみを Apache に処理させます。

以下は最低限の Apache の設定です：

```
1 NameVirtualHost *:80
2 ##### deal with requests on port 80
3 <VirtualHost *:80>
4     Alias / /users/www-data/web2py/applications
5     ##### serve static files directly
6     <LocationMatch "^/welcome/static/.+">
7         Order Allow, Deny
8         Allow from all
9     </LocationMatch>
10    ##### proxy all the other requests
11    <Location "/welcome">
12        Order deny,allow
13        Allow from all
14        ProxyRequests off
15        ProxyPass http://localhost:8000/welcome
16        ProxyPassReverse http://localhost:8000/
17        ProxyHTMLURLMap http://127.0.0.1:8000/welcome/ /welcome
18    </Location>
19    LogFormat "%h %l %u %t \"%r\" %>s %b" common
20    CustomLog /var/log/apache2/access.log common
21 </VirtualHost>
```

上記のスクリプトは、”welcome” アプリケーションだけを公開します。他のアプリケーションを公開したい場合は、対応する<Location>...</Location>に”welcome” アプリケーションと同様の構文を追加する必要があります。

スクリプトは web2py サーバーが、ポート 8000 を使用することを前提にしています。Apache を再起動する前に、この点を再確認してください：

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

-a オプションでパスワードを指定するか、パスワードの代わりに”<recycle>”パラメータを指定することができます。後者の場合は、前回保存されたパスワードが再利用され、シェル履歴にパスワードが保存されることはありません。

”<ask>” パラメータを使って、パスワードの入力要求させることもできます。

nohup コマンドは、シェルを閉じた時にサーバーが落ちないようにします。nohup は nohup.out に全てのログを出力します。admin と appadmin に HTTPS を強制的に利用させるには、代わりに次の Apache の設定ファイルを使用します：

```
1 NameVirtualHost *:80
2 NameVirtualHost *:443
3 ##### deal with requests on port 80
4 <VirtualHost *:80>
5     Alias / /usres/www-data/web2py/applications
6     ##### admin requires SSL
7     <LocationMatch "^/admin">
8         SSLRequireSSL
9     </LocationMatch>
10    ##### appadmin requires SSL
11    <LocationMatch "^/welcome/appadmin/.+">
12        SSLRequireSSL
13    </LocationMatch>
14    ##### serve static files directly
15    <LocationMatch "^/welcome/static/.+">
16        Order Allow,Deny
17        Allow from all
18    </LocationMatch>
19    ##### proxy all the other requests
20    <Location "/welcome">
21        Order deny,allow
22        Allow from all
23        ProxyPass http://localhost:8000/welcome
24        ProxyPassReverse http://localhost:8000/
25    </Location>
26    LogFormat "%h %l %u %t \"%r\" %>s %b" common
27    CustomLog /var/log/apache2/access.log common
28 </VirtualHost>
29 <VirtualHost *:443>
30     SSLEngine On
31     SSLCertificateFile /etc/apache2/ssl/server.crt
32     SSLCertificateKeyFile /etc/apache2/ssl/server.key
```

```

33 <Location "/">
34   Order deny,allow
35   Allow from all
36   ProxyPass http://localhost:8000/
37   ProxyPassReverse http://localhost:8000/
38 </Location>
39   LogFormat "%h %l %u %t \"%r\" %>s %b" common
40   CustomLog /var/log/apache2/access.log common
41 </VirtualHost>
```

*web2py* を共有ホスト上で *mod\_proxy* を使用し動かす場合、管理画面は無効にしておく必要があります。そうしないと他のユーザに公開されてしまいます。

### 13.1.6 Linux デーモンとして起動

*mod\_wsgi* を使用する場合を除き、他の Linux デーモンとして起動/停止/再起動できるように *web2py* サーバーを設定することで、コンピュータのブート時に自動で起動することができます。

これを設定する方法は、個々の Linux/Unix ディストリビューションで固有です。*web2py* フォルダには、この設定のために二つのスクリプトが用意されています：

```

1 scripts/web2py.ubuntu.sh
2 scripts/web2py.fedora.sh
```

Ubuntu や他の Debian ベースの Linux ディストリビューションでは、”*web2py.ubuntu.sh*” を編集し”/usr/lib/web2py” パスを *web2py* インストールパスに置き換え、次のシェルコマンドを実行することでファイルを適切なフォルダへ移動し、スタートアップサービスに登録し起動します：

```

1 sudo cp scripts/web2py.ubuntu.sh /etc/init.d/web2py
2 sudo update-rc.d web2py defaults
3 sudo /etc/init.d/web2py start
```

Fedora や他の Fedora ベースのディストリビューションでは、”*web2py.fedora.sh*” を編集し”/usr/lib/web2py” パスを *web2py* インストールパスに置き換え、次のシェルコマンドを実行することでファイルを適切なフォルダに移動し、スタートアップサービスに登録し起動します：

```
1 sudo cp scripts/web2py.fedora.sh /etc/rc.d/init.d/web2pyd
```

```
2 sudo chkconfig --add web2pyd
3 sudo service web2py start
```

### 13.1.7 Lighttpd

次のシェルコマンドで、Ubuntu や他の Debian ベースの Linux ディストリビューションに、Lighttpd をインストールできます：

```
1 apt-get -y install lighttpd
```

インストールされたら、`/etc/rc.local` を編集し、fcgi web2py パックグランドプロセスを作成します

```
1 cd /var/www/web2py && sudo -u www-data nohup python fcgihandler.py &
```

そして、Lighttpd 設定ファイルを編集する必要があります。

```
1 /etc/lighttpd/lighttpd.conf
```

上記のプロセスで作成されたソケットを見つけることができます。設定ファイルに以下のように記述します：

```
1 server.modules          = (
2     "mod_access",
3     "mod_alias",
4     "mod_compress",
5     "mod_rewrite",
6     "mod_fastcgi",
7     "mod_redirect",
8     "mod_accesslog",
9     "mod_status",
10 )
11
12 server.port = 80
13 server.bind = "0.0.0.0"
14 server.event-handler = "freebsd-kqueue"
15 server.error-handler-404 = "/test.fcgi"
16 server.document-root = "/users/www-data/web2py/"
17 server.errorlog        = "/tmp/error.log"
18
19 fastcgi.server = (
20     "/handler_web2py.fcgi" => (
21         "handler_web2py" => ( #name for logs
22             "check-local" => "disable",
23             "socket" => "/tmp/fcgi.sock"
```

```

24     )
25   ),
26 )
27
28 $HTTP[ "host" ] = "(^|\\.)example\\.com$" {
29   server.document-root="/var/www/web2py"
30   url.rewrite-once =
31     "^^(/.+?/static/.+)$" => "/applications$1",
32     "^(^|/.*)$" => "/handler_web2py.fcgi$1",
33   )
34 }
```

次に構文エラーをチェックします：

```
1 lighttpd -t -f /etc/lighttpd/lighttpd.conf
```

そして次のように、web サーバーを（再）起動します：

```
1 /etc/init.d/lighttpd restart
```

FastCGI は、web2py を IP ソケットで無く、Unix ソケットにバインドする点に注意してください：

```
1 /tmp/fcgi.sock
```

これは Lighttpd が、HTTP 送受信を転送する部分です。Unix ソケットは IP ソケットより軽量で、それが Lighttpd+FastCGI+web2py が高速に動作する一つの理由です。Apache の場合は、静的ファイルディレクトリを Lighttpd に直接処理させ、アプリケーションを強制的に HTTPS 経由にする設定も可能です。詳細は Lighttpd のドキュメントを参照してください。

このセクションの例は、web2pyslices に投稿された John Heenan の記事から引用しました。

*web2py を FastCGI を使った共有ホストで動かす場合、管理画面は無効にしておく必要があります。そうしないと、他のユーザに公開されてしまします。*

### 13.1.8 mod\_python を使った共有ホスティング

しばしば、特に共有ホスト環境で、Apche 設定ファイルを直接編集する権限がない場合があります。この本を書いている現在、mod\_wsgi の登場によってメン

テナансがされていませんが、ほとんどのホストで未だ mod\_python を動かしています。

このような環境でも web2py を動かすことができます。ここでは、どのように設定するか例を見せます。web2py の中身を "htdocs" フォルダに入れます。web2py フォルダに、次の内容の "web2py\_modpython.py" を作成します：

```
1 from mod_python import apache
2 import modpythonhandler
3
4 def handler(req):
5     req.subprocess_env['PATH_INFO'] = req.subprocess_env['SCRIPT_URL']
6     return modpythonhandler.handler(req)
```

次の内容の ".htaccess" ファイルを作成/更新します：

```
1 SetHandler python-program
2 PythonHandler web2py_modpython
3 #PythonDebug On
```

この例は、Niktar によって提供されました。

### 13.1.9 Cherokee と FastCGI

Cherokee は高速な web サーバーで、web2py のように設定用に AJAX が有効な web ベースの管理画面を提供します。管理画面は Python で書かれており、ほとんどの変更に対してサーバーの再起動を必要としません。

こちらが、web2py を Cherokee でセットアップするのに必要な手順です：

Cherokee [90] をダウンロード tar の展開、ビルド、そしてインストール：

```
1 tar -xzf cherokee-0.9.4.tar.gz
2 cd cherokee-0.9.4
3 ./configure --enable-fcgi && make
4 make install
```

"applications" フォルダの作成を確認するために、少なくとも一度は web2py を正常に起動します。

次のコードで、"startweb2py.sh" という名前のシェルスクリプトを作成します：

```
1#!/bin/bash
2cd /var/web2py
3python /var/web2py/fcgihandler.py &
```

そして、スクリプトに実行権限を与えて実行してください。これは FastCGI ハンドラで、web2py が起動します。

Cherokee と cherokee-admin を起動：

```
1 sudo nohup cherokee &
2 sudo nohup cherokee-admin &
```

デフォルトで cherokee-admin は、ローカルのポート 9090 だけを使用します。マシンへ完全な物理的アクセス権があれば問題ではありません。もうしそうでない場合は、次のオプションを利用し、強制的に IP アドレスとポートをバインドできます：

```
1 -b, --bind[=IP]
2 -p, --port=NUM
```

もしくは、SSH ポートフォワード（より安全で、推奨される）を使用します：

```
1 ssh -L 9090:localhost:9090 remotehost
```

ブラウザで”http://localhost:9090”を開き、もし全ての設定がOKであれば、cherokee-admin が開きます。cherokee-admin web インターフェイスで、”info sources” をクリックします。”Local Interpreter” を選びます。次のコードを記述して”Add New” をクリックします。

```
1 Nick: web2py
2 Connection: /tmp/fcgi.sock
3 Interpreter: /var/web2py/startweb2py.sh
```

最後に次のように、残りステップを実行します：

- ”Virtual Servers” をクリックし、更に”Default” をクリックします。
- ”Behavior” をクリックし、更にその下の、”default” をクリックします。
- リストボックスから”List and Send” の代わりに、”FastCGI” を選択します。
- 一番下で、”web2py” を”Application Server” として選択します。
- 全てのチェックボックス（Allow-x-sendfile は外すことも可能）をチェックします。もし警告表示がでたら、どれか一つのチェックボックスを無効にしてから有効にします。（そうすることで、アプリケーションサーバーのパラメータを自動で再送信します。バグで上手く行かない時もあります。）

- ・ ブラウザで”`http://あなたのサイト`”を開くと、”Welcom to web2py” が表示されます。

### 13.1.10 Postgresql

PostgreSQL はフリーでオープンソースのデータベースで、本番環境で使用されています。例えば、.org ドメイン名を保存するデータベースに使用されて、データが何百テラバイトに拡張できることを証明されています。非常に高速で安定したトランザクションサポートを持ち、多くのデータベース保守タスクから管理者を解放する `auto-vacuum` という機能があります。

Ubuntu や他の Debian ベース Linux ディストリビューションでは、PostgreSQL とその Python API を簡単にインストールできます：

```
1 sudo apt-get -y install postgresql
2 sudo apt-get -y install python-psycopg2
```

web サーバーとデータベースは異なるマシンで運用したほうが賢明です。この場合、web サーバーを動かしているマシンは安全な内部（物理）ネットワーク、または SSL トンネルでデータベースサーバーと接続されるべきです。

PostgreSQL の設定ファイルを編集します

```
1 sudo nano /etc/postgresql/8.4/main/postgresql.conf
```

次の二行を記述します

```
1 ...
2 track_counts = on
3 ...
4 autovacuum = on    # Enable autovacuum subprocess? 'on'
5 ...
```

データベースサーバーを起動します：

```
1 sudo /etc/init.d/postgresql restart
```

PostgreSQL サーバーの再起動時に、どのポートで実行されているか表示されます。データベースサーバーが複数で構成される場合を除き、5432 のはずです。

PostgreSQL のログは、次の場所にあります：

```
1 /var/log/postgresql/
```

データベースサーバーが立ち上がって動き出したら、web2py アプリケーションに必要なユーザとデータベースを作成します：

```

1 sudo -u postgres createuser -PE -s myuser
2 postgresql> createdb -O myself -E UTF8 mydb
3 postgresql> echo 'The following databases have been created:'
4 postgresql> psql -l
5 postgresql> psql mydb

```

最初のコマンドは新規作成ユーザ `myuser` に、スーパーユーザ・アクセス権限を与えます。そして、パスワードを入力するようプロンプトを表示します。

どの web2py アプリケーションでも、以下のコマンドでこのデータベースに接続できます：

```
1 db = DAL("postgres://myuser:mypassword@localhost:5432/mydb")
```

`mypassword` はプロンプト時に入力したパスワードで、5432 はデータベースサーバーが実行されているポートです。

通常はアプリケーションごとに一つのデータベースを作成し、同じアプリケーションの複数のインスタンスは同じデータベースに接続します。異なるアプリケーション間で同じデータベースを共有することもできます。

データベースバックアップについては、PostgreSQL ドキュメントにある、`pg_dump` と `pg_restore` コマンドを読んでください。

## 13.2 Windows

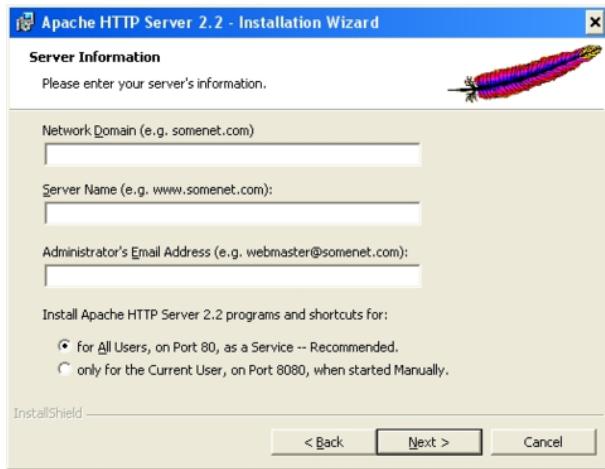
### 13.2.1 Apache と mod\_wsgi

Windows で Apache と mod\_wsgi をインストールするには、異なる手順が必要になります。Python 2.5 がインストールされていて、web2py がソースから実行され、さらに `c:/web2py` にあることを前提に説明します。

最初に、必要なパッケージをダウンロードします：

- [83] から Apache apache\_2.2.11-win32-x86-openssl-0.9.8i.msi
- [84] から mod\_wsgi

二つ目に、apache...msi を実行し、ウィザード画面に従います。サーバー・インフォメーション画面で



全ての要求項目を入力します：

- **Network Domain:** サーバーの現在のまたは登録予定の DNS ドメインを入力します。もしサーバーの正式な DNS 名が server.mydomain.net の場合は、mydomain.net と入力します。
- **ServerName:** サーバーの完全な DNS 名です。上記の例で言うと、server.mydomain.net と入力します。完全修飾ドメイン名、もしくはショートカットではない web2py インストールの IP アドレスを入力します。詳細は、[86]. を参照してください。
- **Administrator's Email Address:** サーバー管理者または web マスターのメールアドレスを入力します。このアドレスはデフォルトで、クライアントへのエラーメッセージと共に表示されます。

特に必要がなければ、標準インストールで最後まで続行します

ウィザードはデフォルトで、以下のフォルダに Apache をインストールします：

```
1 C:/Program Files/Apache Software Foundation/Apache2.2/
```

ここからは簡単にするためフォルダを、Apache2.2 と呼びます。

三つ目に、ダウンロードした mod\_wsgi.so を Apache2.2/modules にコピーし

ます Apache2.2/modules

Chris Travers によって書かれ、2007 年 12 月に Microsoft の Open Source Software Lab で公開されました。

四つ目に、server.crt と server.key 証明書（前のセクションで説明した）を作成し、Apache2.2/conf フォルダに設置します。cnf ファイルは、Apache2.2/conf/openssl.cnf にあることに注意してください。

五つ目に、Apache2.2/conf/httpd.conf を編集し、次の行のコメント (# 文字) を外します。

```
1 LoadModule ssl_module modules/mod_ssl.so
```

他の全ての LoadModule 行の後に、次のコードを追加します。

```
1 LoadModule wsgi_module modules/mod_wsgi.so
```

”Listen 80” の記述を検索し、次の行をその後に追加します

```
1 Listen 443
```

あなたの設定値に従って、ドライブレター、ポート番号、サーバー名を変更し、一番最後に次の行を追加します。

```
1 NameVirtualHost *:433
2 <VirtualHost *:443>
3   DocumentRoot "C:/web2py/applications"
4   ServerName server1
5
6   <Directory "C:/web2py">
7     Order allow,deny
8     Deny from all
9   </Directory>
10
11  <Location "/">
12    Order deny,allow
13    Allow from all
14  </Location>
15
16  <LocationMatch "^(/[\w_]*static/.*)">
17    Order Allow,Deny
18    Allow from all
19  </LocationMatch>
20
21  WSGIScriptAlias / "C:/web2py/wsgihandler.py"
22
```

```

23     SSLEngine On
24     SSLCertificateFile conf/server.crt
25     SSLCertificateKeyFile conf/server.key
26
27     LogFormat "%h %l %u %t \"%r\" %>s %b" common
28     CustomLog logs/access.log common
29 </VirtualHost>

```

設定を保存し、[Start > Program > Apache HTTP Server 2.2 > Configure Apache Server > Test Configuration] を使って設定を確認します。

もし問題が無ければコマンド画面が開いて閉じます。これで、次のように Apache を起動できます：

[Start > Program > Apache HTTP Server 2.2 > Control Apache Server > Start]

もっと良いのは、タスクバー モニタを起動します。

[Start > Program > Apache HTTP Server 2.2 > Control Apache Server]

これで赤い羽根のようなタスクバー アイコン上の右クリックで、”Open Apache Monitor” を選択し、起動、停止、再起動を必要に応じて実行できます。

このセクションは Jonathan Lundell によって作成されました。

### 13.2.2 Windows サービスとして起動

Linux のデーモンは、Windows ではサービスと呼ばれます。web2py サーバーは Windows サービスとして簡単に、インストール/起動/停止ができます。web2py を Windows サービスとして使用するには、起動パラメータを使用した”options.py”を作成する必要があります：

```

1 import socket, os
2 ip = socket.gethostname()
3 port = 80
4 password = '<recycle>'
5 pid_filename = 'httpserver.pid'
6 log_filename = 'httpserver.log'
7 ssl_certificate =
8 ssl_private_key =
9 numthreads = 10
10 server_name = socket.gethostname()

```

```

11 request_queue_size = 5
12 timeout = 10
13 shutdown_timeout = 5
14 folder = os.getcwd()

```

既にモデルとして使用できる、”options\_std.py” というファイルが web2py フォルダにあるため、”options.py” を一から作成する必要はありません。

”options.py” を web2py のインストール・フォルダに作成したら、次のコマンドで web2py をサービスとして追加できます：

```
1 python web2py.py -W install
```

そして、次のコマンドで起動/停止できます：

```

1 python web2py.py -W start
2 python web2py.py -W stop

```

### 13.3 セッション保護と admin

HTTPS 経由で実行されている場合を除き、admin アプリケーションと appadmin コントローラを公開することは、とても危険です。そしてパスワードと証明書は、暗号化無しで通信されるべきではありません。これは web2py と、他のどの web アプリケーションにも当てはまります。

アプリケーションで認証が必要な場合は、次のようにセッションクッキーを保護するべきです：

```
1 session.secure()
```

サーバー上で安全な本番環境をセットアップする簡単な方法は、まず web2py を停止し、web2py のインストール・フォルダから全ての parameters\_\*.py ファイルを削除することです。そしてパスワード無しで web2py を起動します。これで admin と appadmin は完全に無効になります。

```
1 nohup python web2py --nogui -p 8001 -i 127.0.0.1 -a '' &
```

次に、ローカルホストからだけアクセス可能な二つ目の web2py インスタンスを起動します：

```
1 nohup python web2py --nogui -p 8002 -i 127.0.0.1 -a '<ask>' &
```

そして、ローカルマシン( 管理画面にアクセスしたいマシン )からサーバー( web2py が実行されている、example.com )へ、SSH トンネルを作成します：

```
1 ssh -L 8002:127.0.0.1:8002 username@example.com
```

これでブラウザ経由の `localhost:8002` で、管理画面へローカルに接続することができます。

トンネルが閉じている（ユーザがログアウト）時は、admin へ接続できないので設定は安全になります。

この方法は他のユーザが、`web2py` を含むフォルダへの読み取りアクセスを持たない共有ホスト環境では安全です。そうでない時は、ユーザがサーバーから直接セッションクッキーを盗むことが可能です。

### 13.4 効率とスケーラビリティ

`web2py` は簡単に、デプロイとセットアップができるように設計されています。これは効率やスケーラビリティに妥協しているという意味ではなく、拡張するには調整が必要な場合があるということです。

このセクションではローカルのロードバランサを提供する NAT サーバーの後に、複数の `web2py` をインストールする場合を考えてみます。

この場合、幾つかの条件に合致するのであれば、`web2py` はそのまま動作します。具体的には、それぞれの `web2py` アプリケーションの全てのインスタンスが、同一のデータベースサーバーにアクセスし、さらに同一のファイルを参照している必要があります。後者の条件は、次のフォルダを共有させることで実現が可能です。

```
1 applications/myapp/sessions
2 applications/myapp/errors
3 applications/myapp/uploads
4 applications/myapp/cache
```

共有フォルダは、ファイルロックをサポートしなければいけません。方法としては ZFS(ZFS は Sun Microsystems に開発されており、推奨する選択肢です)、NFS ( ファイルロックを有効にするために nlockmgr デーモンを実行する必要があるかもしれません ) もしくは、Samba(SMB) です。`web2py` フォルダ全体や

アプリケーションフォルダ全体を共有することも可能ですが、ネットワーク帯域使用量が無駄に増加するだけなので良い方法ではありません。

共有する必要があるが、トランザクションの安全性を必要としないリソースを、共有ファイルシステムに移動します。これによってデータベースの負荷を減らすため、上記の設定の説明は、非常に拡張性が高いと考えます（一回一クライアントのみのセッションファイルのアクセス想定、常時必要なキャッシュのグローバルロック、一度の uploads と errors への書き込み/たくさんのファイルの読み取り）。

理想的には、データベースと共有ストレージは両方とも RAID 構成であるべきです。データベースを共有フォルダと同じストレージに保存する間違いをしないでください、新しいボトルネックを作ってしまいます。

ケースバイケースで、追加での最適化実行が必要であり、それについては後述します。具体的には、どのように共有フォルダを一つずつ排除するか、そして代わりに関連データをデータベースに保存する方法を説明します。これは可能ですが、良い解決策とは限りません。それでも行う理由があるかもしれません。そのような理由の一つは、共有フォルダを自由にセットアップできない場合です。

#### 13.4.1 効率のトリック

web2py アプリケーションコードはリクエストのたびに実行されるので、コードの量を最小限に抑えたい時があります。ここで以下の方法があります：

- 一度だけ `migrate=True` を実行してから、全てのテーブルに `migrate=False` をセットします。
- `admin` を使った、アプリケーションのバイトコードコンパイルします。
- `cache.ram` をできる限り使用するため、有限なキーセットを使っているか確認します。そうしないと、使用するキャッシュ量は任意に成長します。
- モデル内のコードを最小化します：ここに関数を定義せずそれを必要とするコントローラに定義します。もっといい方法は、モジュールに関数を定義して必要に応じてインポートして使用します。
- たくさんの関数を同じコントローラに設定しないで、たくさんのコントローラにいくつかの関数を定義します。

- セッションを変更しない全てのコントローラや関数で、`session.forget(response)` を実行します。
- web2py cron を使わず、代わりにバックグラウンドプロセスを利用するようになります。web2py cron は大量の Python インスタンスを起動し過剰にメモリを使用します。

### 13.4.2 データベースでのセッション

セッションフォルダの代わりに、データベースにセッションを保存するように web2py に指示することが可能です。セッションの保存に同じデータベースを使用することになるとしても、それぞれ個別のアプリケーションに対して設定が必要です。

#### データベース接続を作成

```
1 db = DAL(...)
```

接続を確立する同じモデルファイル内に次のコードを追加するだけで、データベースにセッションを保存することができます：

```
1 session.connect(request, response, db)
```

テーブルがまだ存在しない場合、we2py は `web2py_session_` アプリケーション名という、以下のフィールドを持ったデータベースのテーブルを作成します：

```
1 Field('locked', 'boolean', default=False),
2 Field('client_ip'),
3 Field('created_datetime', 'datetime', default=now),
4 Field('modified_datetime', 'datetime'),
5 Field('unique_key'),
6 Field('session_data', 'text')
```

”unique\_key” は、クッキー内のセッションを特定するために使用する uuid キーです。”session\_data” は cPickled セッションデータです。

データベースアクセスを最小限にするために、必要がない場合はセッションを保存しないようにするべきです：

```
1 session.forget()
```

この変更により、”sessions” フォルダはアクセスされなくなるので、共有フォルダにする必要は無くなります。

セッションが無効の場合は、`session` を `form.accepts` に渡してはいけません。また、`session.flash` や `CRUD` も使用できない点に注意してください。

### 13.4.3 HAProxy 高可用性ロードバランサ

複数の web2py プロセスを複数のマシンで実行する必要がある場合は、データベースにセッションを保存したり、キャッシュする代わりに、ステイッキー・セッションを利用したロードバランサを使用するオプションがあります。

Pound [92] と HQProxy [93] は、ステイッキー・セッションを提供する、二つの HTTP ロードバランサとリバースプロキシです。ここでは商用 VPS ホスティングで、より一般的な後者について説明します。

ステイッキー・セッションは一度セッションクッキーが発行されたら、ロードバランサはセッションに関連付けられたクライアントからのリクエストを、常に同じサーバーにルーティングします。こうすることでファイルシステムを共有せずに、ローカルのファイルシステムにセッションを保存することができます。

HAProxy を使うには：

初めに、Ubuntu テストマシンにインストールします：

```
1 sudo apt-get -y install haproxy
```

二つ目に、設定ファイル”/etc/haproxy.cfg”を以下のように編集します：

```
1 ## this config needs haproxy-1.1.28 or haproxy-1.2.1
2
3 global
4     log 127.0.0.1    local0
5     maxconn 1024
6     daemon
7
8 defaults
9     log      global
10    mode     http
11    option   httplog
12    option   httpchk
13    option   httpclose
14    retries 3
15    option   redispatch
16    contimeout      5000
17    clitimeout      50000
```

```

18     srvtimeout      50000
19
20 listen 0.0.0.0:80
21     balance url_param WEB2PYSTICKY
22     balance roundrobin
23     server L1_1 10.211.55.1:7003 check
24     server L1_2 10.211.55.2:7004 check
25     server L1_3 10.211.55.3:7004 check
26     appsession WEB2PYSTICKY len 52 timeout 1h

```

listen ディレクティブは、どのポートで接続を待つかを HAProxy に指示します。server ディレクティブは、プロキシサーバーがどこにあるかを指示します。appsession ディレクトリは、ステイッキー・セッションを作成し WEB2PYSTICKY というクッキーをこの目的で使用します。

三つ目に、この設定を有効にし、HAProxy を起動します：

```
1 /etc/init.d/haproxy restart
```

以下の URL に、Pound のセットアップの同様の手順があります

```
1 http://web2pyslices.com/main/slices/take_slice/33
```

#### 13.4.4 セッションのクリーンアップ

本番環境では、セッションが速く積み重なることに注意してください。web2py は次のスクリプトを提供しています：

```
1 scripts/sessions2trash.py
```

バックグラウンドで実行され、定期的に一定期間アクセスされていない全てのセッションを削除します。web2py は、これらのセッションをクリーンアップするスクリプトを提供します（ファイルベースのセッション及びデータベースのセッションのどちらでも有効です）。

次に典型的な使用例を挙げます：

- 5 分毎に期限切れのセッションを消去します：

```
1 nohup python web2py.py -S app -M -R scripts/sessions2trash.py &
```

- 詳細な出力と共に、有効期限に関わらず 60 分より古いセッションを消去し、終了します。

```
1 python web2py.py -S app -M -R scripts/sessions2trash.py -A -o -x 3600 -
f -v
```

- 有効期限に関わらず全てのセッションを消去し、終了します：

```
1 python web2py.py -S app -M -R scripts/sessions2trash.py -A -o -x 0
```

app に、対象のアプリケーション名を指定します。

#### 13.4.5 データベース上でのファイルアップロード

デフォルトでは、SQLFORM によって処理されたアップロードファイルは、安全に名前変更されファイルシステムの”uploads” フォルダに保存されます。web2py に対して指示し、フォルダの代わりにデータベースにアップロードファイルを保存させることも可能です。

次のテーブルを考えてみます：

```
1 db.define_table('dog',
2     Field('name')
3     Field('image', 'upload'))
```

dog.image は upload タイプです。犬の名前と同じレコードにアップロード画像を保存するには、blob フィールドを追加し upload フィールドにリンクするようにテーブル定義を修正する必要があります：

```
1 db.define_table('dog',
2     Field('name')
3     Field('image', 'upload', uploadfield='image_data'),
4     Field('image_data', 'blob'))
```

ここで”image\_data” は、任意につけた新しい blob フィールドの名前です。

3 行目は通常通りアップロードされた画像の名前を安全に変更し、変更された新しい名前を image フィールドに保存し、ファイルシステムに保存する代わりに”image\_data” という upload フィールドにデータを保存します。この全ての処理が SQLFORM によって自動で実行され、他のコードを変更する必要はありません。

この変更で、”uploads” フォルダは必要なくなります。

Google App Engine では、デフォルトで uploadfield が自動作成されるので、uploadfield を定義しなくてもデータベースに保存されます。

### 13.4.6 チケットの収集

デフォルトで、web2py はチケット（エラー）をローカルのファイルシステムに保存します。一番のエラー原因は、本番環境でのデータベース障害であるため、データベースにチケットを直接保存することは意味がありません。

通常は稀なイベントのため、チケットの保存はボトルネックになりません。このため複数サーバーで構成された本番環境では、共有フォルダに保存するのも適当です。とはいっても管理者だけがチケットを取り出す必要があるので、共有されていないローカルの”エラー” フォルダにチケットを保存し、定期的に収集して削除するのも大丈夫です。

定期的にローカルのチケットをデータベースに移動する、という方法もあります。このために、web2py は次のスクリプトを提供します：

```
1 scripts/tickets2db.py
```

デフォルトでこのスクリプトは ticket\_storage.txt という、プライベートフォルダに保存されたファイルから db の uri を取得します。このファイルには次のような、DAL インスタンスへ直接渡す文字列を含める必要があります：

```
1 mysql://username:password@localhost/test
2 postgres://username:password@localhost/test
3 ...
```

これは、スクリプトをそのまま残すことができます：もし複数のアプリケーションがある場合、全アプリケーションへの適切な接続を動的に選択します。もし uri をハードコーディングしたい場合は、*except* 行の直後に、db\_string への 2 つめの参照を記述してください。次のコマンドでスクリプトを実行できます：

```
1 nohup python web2py.py -S myapp -M -R scripts/tickets2db.py &
```

myapp は対象となるアプリケーション名です。

このスクリプトはバックグラウンドで動作し、5 分毎に全てチケットをデータベースに移動し、ローカルチケットを消去します。admin アプリの上部にある”switch to:db” ボタンをクリックすると、ファイルシステムに保存されている場合と全く同じような機能で、エラーを表示することができます。

この変更で、エラーはデータベースに保存されるので、”errors” フォルダは共有フォルダである必要はありません。

### 13.4.7 Memcache

web2py が提供する二つのタイプのキャッシュ: `cache.ram` と `cache.disk` を説明してきました。どちらも複数のサーバーによる分散環境で動作しますが、期待通りには動作しません。具体的には、`cache.ram` は、サーバーレベルでのみキャッシュします。このため、役には立ちません。`cache.disk` は、”cache” フォルダーがファイルロックをサポートする共有フォルダの場合を除き、同様にサーバーレベルでキャッシュします。このため、スピードアップでは無く、主なボトルネックになります。

解決策としてはこれらは使わず、代わりに `memcache` を使います。web2py には `memcache` API が付属しています。`memcache` を使うためには、例えば `0_memcache.py` という新規のモデルファイルを作成し、次のコードを記述（または追記）します：

```
1 from gluon.contrib.memcache import MemcacheClient
2 memcache_servers = ['127.0.0.1:11211']
3 cache.memcache = MemcacheClient(request, memcache_servers)
4 cache.ram = cache.disk = cache.memcache
```

最初の行は `memcache` をインポートします。2 行目は `memcache` socket(サーバー:ポート) のリストです。3 行目は `cache.memcache` を定義します。4 行目は `cache.ram` と `cache.disk` を、`memcache` で再定義します。

`Memcache` オブジェクトを指している全く新しいキャッシュオブジェクトを定義するために、それらの 1 つだけを再定義するために選択することができます。

この変更で、”cache” フォルダはアクセスが無くなるので、共有フォルダである必要はありません。

このコードは、`memcache` サーバーがローカルネットワークで動作していることが前提です。サーバーのセットアップ方法は、`memcache` ドキュメントを参照してください。

### 13.4.8 Memcache でのセッション

セッションが必要だが、ロードバランサでスティッキー・セッションを使用したくない場合は、`memcache` にセッションを保存するオプションがあります：

```
1 from gluon.contrib.memdb import MEMDB
```

```
2 session.connect(request, response, db=MEMDB(cache.memcache))
```

### 13.4.9 Redisによるキャッシング

[103] Memcacheの代替としてRedisが使用できます。

Redisがインストールされており、localhostの6379ポートで実行されていると仮定すると、次のコードで接続可能です（モデルに記述します）：

```
1 from gluon.contrib.redis import RedisCache
2 cache.redis = RedisCache('localhost:6379', db=None, debug=True)
```

'localhost:6379'は接続文字列です。dbはDALオブジェクトではなく、Redisのデータベース名です。

cache.ramとcache.diskの代わりに（または一緒に）cache.redisを使用できます。

次のコードでRedisの統計情報を取得できます：

```
1 cache.redis.stats()
```

### 13.4.10 アプリケーションの削除

本番環境では、デフォルトアプリケーション:admin、examples、welcomeをインストールしないほうが良いかもしれません。小さいですが必要なないアプリケーションだからです。

これらのアプリケーションを削除するのは簡単です。applicationsフォルダ下の対象のフォルダを削除するだけです。

### 13.4.11 レプリカデータベースの使用

高いパフォーマンスを必要とする本番環境では、たくさんのレプリカスレーブを持つマスタースレーブ・データベース構成や、恐らく一組のレプリカサーバーがあるかもしれません。DALは、このような状況を処理し、条件付きリクエストパラメータに応じて異なるサーバーに接続することができます。それを実現するAPIは、6章で説明済みです。以下はその例です：

```

1 from random import sample
2 db = DAL(sample(['mysql://...1', 'mysql://...2', 'mysql://...3'], 3))

```

この場合、異なる HTTP リクエストがランダムで異なるデータベースによって処理され、それぞれの DB が同じような頻度で使用されます。

シンプルなラウンドロビンを実装することもできます

```

1 def fail_safe_round_robin(*uris):
2     i = cache.ram('round-robin', lambda: 0, None)
3     uris = uris[i:] + uris[:i] # rotate the list of uris
4     cache.ram('round-robin', lambda: (i+1)%len(uris), 0)
5     return uris
6 db = DAL(fail_safe_round_robin('mysql://...1', 'mysql://...2', 'mysql://...3'))

```

これは、リクエストに割り当てられたデータベースサーバーの接続に失敗した場合は、DAL は順番に次のサーバーに接続を試みるという意味で、フェイルセーフです。

リクエストされたアクションやコントローラによって、異なるデータベースに接続することも可能です。マスタースレーブ・データベース構成で、あるアクションは読み取り専用で、いくらかの人物は読み取り/書き込みを両方します。前者はスレーブ DB サーバーに安全に接続でき、後者はマスターに接続されるべきです。これは以下のように記述できます：

```

1 if request.function in read_only_actions:
2     db = DAL(sample(['mysql://...1', 'mysql://...2', 'mysql://...3'], 3))
3 if request.action in read_only_actions:
4     db = DAL(shuffle(['mysql://...1', 'mysql://...2', 'mysql://...3']))
5 else:
6     db = DAL(sample(['mysql://...3', 'mysql://...4', 'mysql://...5'], 3))

```

1、2、3 はスレーブで、3、4、5 はマスターです。

### 13.5 Google App Engine でのデプロイ

DAL コードを含めて、Google App Engine(GAE) [13] で、web2py を動かすことは可能です。

GAE は 2 つのバージョンの Python をサポートします：2.5(デフォルト)と 2.7(ベータ)です。web2py は、デフォルトでは 2.5 を採用していますが両方サポー

トします（デフォルトは将来変更される可能があります）。“app.yaml” ファイルに、詳細な設定が書かれていますので参照ください。

GAE は、Google SQL データベース（MySQL と互換）と Google NoSQL（“Datastore” と呼ばれています）の両方をサポートします。web2py はどちらもサポートしています。もし Google SQL データベースを使用する場合は、6 章の指示に従ってください。このセクションでは、Google Datastore を使用すると仮定しています。

GAE プラットフォームは、通常のホスティング環境よりも幾つか良い点があります：

- 簡単にデプロイができます。Google は内部の構造を完全に抽象化します。
- スケーラビリティがあります。同時アクセスリクエストの数だけ、何度でもアプリケーションを複製します。
- SQL と NoSQL を選択できます（両方可）

しかし、不利な点もあります：

- ファイルシステムへの読み書きができません。
- Google の証明書で、appspot.com ドメインを使用しないと HTTPS が利用できません。

幾つか Datastore の仕様で不利な点があります：

- 典型的なトランザクションがありません。
- 複雑な Datastore クエリがありません。具体的には JOIN、LIKE、DATE/DATETIME 演算子がありません。
- あるフィールドと同じフィールド以外のサブクエリの、複数の OR がありません。

読み取り専用のファイルシステムのため、web2py はセッション、エラーチケット、キャッシュファイルやアップロードファイルをファイルシステムに保存できません。このようなものはファイルシステムではなく、Datastore に保存しなければいけません。

ここでは GAE の簡単な解説と web2py 特有の問題を説明しました。詳細については公式の GAE ドキュメントを参考にしてください。

注意：執筆時点では GAE は *Python 2.5* しかサポートしていません。それ以外のバージョンでは問題が発生します。また、*web2py* はバイナリではなく、ソースディストリビューションを実行する必要があります。

### 13.5.1 設定

注意すべき 3 つの設定ファイルがあります：

```
1 web2py/app.yaml  
2 web2py/queue.yaml  
3 web2py/index.yaml
```

*app.yaml* と *queue.yaml* の設定案を、テンプレートファイル *app.example.yaml* と *queue.example.yaml* から簡単に作成できます。*index.yaml* は Google デプロイメントソフトウェアで自動で作成されたものです。

*app.yaml* は次の構造を持っています（… の部分は省略しています）：

```
1 application: web2py  
2 version: 1  
3 api_version: 1  
4 runtime: python  
5 handlers:  
6 - url: /_ah/stats.*  
7   ...  
8 - url: /(?P<a>.+?)/static/(?P<b>.+)  
9   ...  
10 - url: /_ah/admin/*  
11   ...  
12 - url: /_ah/queue/default  
13   ...  
14 - url: .*  
15   ...  
16 skip_files:  
17   ...
```

*app.example.yaml* (*app.yaml* にコピーする時) は、*web2py* の *welcome* アプリケーションのデプロイ用に設定できますが、*admin* や *example* アプリケーション用ではありません。Google App Engine に登録したアプリケーション id で、*web2py* を置き換える必要があります。

url: /(.+?)/static/(.+) は処理速度を上げるため、web2py ロジックを呼び出すことなく、直接アプリの静的ファイルを提供するよう GAE に指示します。

url:.\* は他の全てのリクエストに、`gaehandler.py` を使うように web2py に指示します。

`skip_files:` セッションは、GAE でデプロイする必要がないファイルのための、正規表現のリストです。次の特定の行について説明します：

```
1 (applications/(admin|examples)/.*) |
2 ((admin|examples/welcome)\.(w2p|tar)) |
```

アンパックした `welcome` の雛形アプリケーションを除いて、デフォルトアプリケーションをデプロイしないように GAE に指示しています。ここに無視すべきアプリケーションをさらに追加することができます。

アプリケーション `id` とバージョン設定以外には、`app.yaml` を編集する必要はありませんがかもしれません。もっとも、`welcome` アプリケーションは除外するかもしれません。

ファイル `queue.yaml` は、GAE タスクキューを設定するのに使用されます。

ファイル `index.yaml` は、GAE の appserver ( Google SDK についてくる web サーバー ) で、アプリケーションをローカルで実行する際に自動で作成されます。中身には以下のコードを含みます：

```
1 indexes:
2 - kind: person
3   properties:
4     - name: name
5       direction: desc
```

この例では GAE に、”name” フィールドをアルファベットの降順ソートした、”person” というテーブルの `index` を作成するように指示します。対応する `index` がないと、アプリケーションで検索やソートを行うことができません。

デプロイメントの前には常に `appserver` を使って、アプリをローカルで動かし全ての機能を確認することが重要です。これはテスト目的だけではなく、”`index.yaml`” を自動で作成する目的もあります。時々このファイルを編集し、重複したエンティーなどを削除したりするなど掃除をしたほうがいいでしょう。

### 13.5.2 実行とデプロイメント

#### Linux

GAE SDK をインストール済みであるとします。執筆時には GAE は Python 2.5.2 で動作します。以下の appserver コマンドで、”web2py” フォルダ中のアプリケーションを実行できます：

```
1 python2.5 dev_appserver.py ../web2py
```

これで appserver が起動します。そして、次の URL でアプリケーションを実行できます：

```
1 http://127.0.0.1:8080/
```

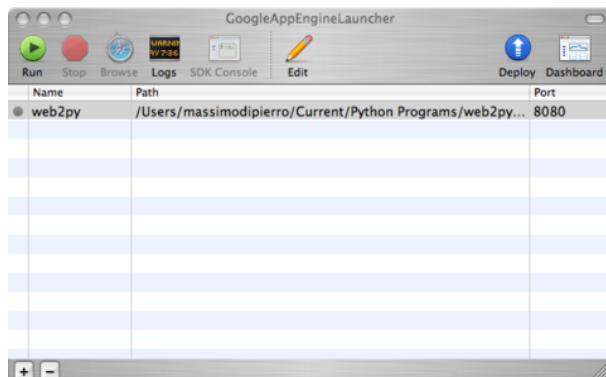
GAE にアプリケーションをアップロードするには、前述したように”app.yaml”を編集し、適切なアプリケーション id をセットしたことを確認した上で、次を実行します：

```
1 python2.5 appcfg.py update ../web2py
```

#### Mac, Windows

Mac や Windows では、Google App Engine ランチャを使用できます。次のリンクからダウンロードできます。 [13]

[File][Add Existing Application] を選択し、 web2py フォルダのトップレベルのパスを path にセットし、そしてツールバーにある [Run] ボタンを押します。ローカルで動作確認できたら、ツールバーにある [Deploy] ボタンをクリックするだけでデプロイできます（アカウントを持っているのが前提です）。



GAE 上での web2py のチケット/エラーは、ログにアクセスしオンライン検索できる GAE 管理コンソールに、同様のログが表示されます。

The screenshot shows the Google App Engine Logs interface. On the left, there's a sidebar with links to Dashboard, Logs (which is selected), Datastore, Indexes, Data Viewer, Administration, Application Settings, Developers, Versions, Resources, Documentation, and Developer forum. The main area displays a list of log entries. At the top of the log list, there are filters for 'Filter Logs' (with a question mark icon), 'Minimum Severity' set to 'Error', and an 'Options' link. A tip message says: 'Tip: Click a log line to show or hide its details.' Below this, several log entries are listed, each with a timestamp, severity level (prefixed with a blue square icon), URL, and a brief description. For example, one entry shows an 'E' (Error) at 07-05 12:13PM with the message 'unable to import wsgiserver'. The log list has a scrollbar on the right.

### 13.5.3 ハンドラの設定

gaehandler.py は GAE 用に提供されているファイルで、いくつかのオプションがあります。以下はデフォルトの値です：

```
1 LOG_STATS = False
2 APPSTATS = True
3 DEBUG = False
```

LOG\_STATS はページ表示にかかった時間を GAE のログに出力します。

APPSTATS はプロファイリング統計を提供する GAE appstats を有効にします。次の URL でアクセスできます：

```
1 http://localhost:8080/_ah/stats
```

DEBUG はデバッグモードをセットします。gluon.settings.web2py\_runtime を使って、コードで明示的にチェックしない限り、実際には違いはありません。

### 13.5.4 ファイルシステムの無効

GAE ではファイルシステムにアクセスできません。書き込みのためにファイルを開くこともできません。

このため GAE では、”upload” フィールドが uploadfiled 属性を持つ持たないにかかわらず、web2py は自動で全てのアップロードファイルをデータストアに保存します。

セッションとチケットもデータベースに保存する必要があり、以下のように明示しなければいけません：

```

1 if request.env.web2py_runtime_gae
2     db = DAL('gae')
3     session.connect(request, response, db)
4 else:
5     db = DAL('sqlite://storage.sqlite')
```

上記のコードは GAE で動作しているかをチェックし、BigTable に接続し、そして web2py にセッションとチケットをそこに保存するように指示します。それ以外の場合は sqlite データベースに接続します。このコードは雛形アプリケーションの”db.py” に、既に存在しています。

### 13.5.5 Memcache

必要であれば、memcache にセッションを保存することができます：

```

1 from gluon.contrib.gae_memcache import MemcacheClient
2 from gluon.contrib.memdb import MEMDB
3 cache.memcache = MemcacheClient(request)
4 cache.ram = cache.disk = cache.memcache
5
6 db = DAL('gae')
7 session.connect(request, response, MEMDB(cache.memcache))
```

GAE では cache.ram と cache.disk は使用されるべきではありません、そこで cache.memcache を使用している点に注意してください。

### 13.5.6 Datastore の問題

マルチエンティティ・トランザクションが無く、リレーションナルデータベースの標準的な関数を使用できない点で、GAE は他のホスティング環境と異なります。これは高い拡張性への代償です。これらの制限を受け入れることができるなら、GAE は非常に優れたプラットフォームです。そうでない場合は、代わりにリレーションナルデータベースを使った、通常のホスティング・プラットフォームを検討したほうが良いでしょう。

もし GAE 上で web2py アプリケーションが動作しなかったら、前述した制限の一つが原因です。多くの問題は、web2py クエリから JOIN を削除し、非正規化することで解決します。

Google App Engine は `ListProperty` や `StringListProperty` といった、特別なフィールドタイプをサポートします。web2py は以下の古い構文を使うことで、これらのタイプを使用できます：

```
1 from gluon.dal import gae
2 db.define_table('product',
3     Field('name'),
4     Field('tags', type=gae.StringListProperty())
```

もしくは、同等の新しい構文：

```
1 db.define_table('product',
2     Field('name'),
3     Field('tags', 'list:string')
```

どちらの場合も”tags” フィールドは `StringListProperty` タイプで、値はストリングのリストである必要があります。GAE のドキュメントに互換性に関する記載があります。we2py がフォームのコンテキストでフィールドをスマートに扱えること、リレーションナルデータベースでも動作すること、これらの理由により二つ目の記述の方が好ましいです。

同様に `ListProperty(int)` にマッピングする `list:integer` と `list:reference` を、web2py はサポートします。

`list` タイプの詳細は 6 章で説明されています。

### 13.5.7 GAE と HTTPS

アプリケーションの id が”myapp” の場合、GAE ドメインは

```
1 http://myapp.appspot.com/
```

です。これは、HTTPS でアクセスすることもできます

```
1 https://myapp.appspot.com/
```

この場合、Google によって提供される”appspot.com” の証明書を使用します。

DNS エントリーの登録で、他のドメイン名でアプリを使用できますが、HTTPS は使用できません。執筆時点で、これは GAE の制限です。

### 13.6 Jython

web2py は普段 CPython(C 言語で実装された Python インタプリタ) で稼動しますが、Jython(Java 言語で実装され Python インタプリタ) でも動作します。これは web2py を Java の基盤で稼働できることを意味します。web2py は Jython すぐに実行できますが、Jython や zxJDBC(Jython のデータベースアダプタ) のセットアップで、いくつかの複雑でトリッキーなことがあります。以下はその手順です：

- Jython.org から ”jython\_installer-2.5.0.jar” (または 2.5.x) をダウンロードします
- それをインストールします：

```
1 java -jar jython_installer-2.5.0.jar
```

- [101] から ”zxJDBC.jar” をダウンロードしインストールします
- [102] から ”sqlitejdbc-v056.jar” ファイルをダウンロードしインストールします
- zxJDBC と sqlitejdbc を java の CLASSPATH に追加します
- web2py を Jython で起動します

```
1 /path/to/jython web2py.py
```

執筆時点で、Jython では `sqlite` と `postgres` のみサポートされています。

第3版 - 翻訳: Omi Chiba レビュー: 中垣健志

第4版 - 翻訳: Yota Ichino レビュー: Hitoshi Kato

# 14

## その他のレシピ

### 14.1 アップグレード

管理者インターフェースの”site” ページには、”update now” があります。実行できないか、動作しない場合 (例えば、ファイルロック問題のため)、web2py の手動アップデートはとても簡単です。

単に、新しい *web2py* を古いインストレーションへ解凍するだけです。

`admin`, `examples`, `welcome` アプリケーションだけでなくライブラリも全てアップグレードします。それは、新しい空の”NEWINSTALL” ファイルを生成します。再起動時に、`web2py` は、空のファイルを消去し、`welcome` アプリケーションは、新しい雛形アプリケーションとして使われる”`welcome.w2p`” の中に格納されます。`web2py` は、他のファイルや既存のアプリケーションをアップグレードしません。

### 14.2 バイナリでアプリケーションを配布する方法

`web2py` のバイナリ形式の配布と一緒に独自のアプリケーションを同梱し配布できます。`web2py` をバンドルしている事をアプリケーションのライセンスに明確に記載している事、そして `web2py.com` へのリンクを追記していれば、`web2py` のライセンスはこれを許可します。

ここに Windows 用の配布方法を説明します:

- いつも通りあなたのアプリを製作
- admin を使い、アプリケーションをバイトコードにコンパイル (1 クリック)
- admin を使い、コンパイル済みアプリケーションをパック (もう 1 クリック)
- "myapp" フォルダを生成
- Windows 用の web2py バイナリ配布をダウンロード
- それを"myapp" フォルダに解凍し、起動 (2 クリック)
- admin を使いパックされたコンパイル済みアプリケーション名"init" をアップロード (1 クリック)
- "cd web2py; web2py.exe" と書かれた"myapp/start.bat" ファイルを作成
- あなたのアプリケーション用のライセンスと、"distributed with an unmodified copy of web2py from web2py.com" である明言に念を押せることを含んだ"myapp/license" ファイルを作成
- myapp フォルダを"myapp.zip" ファイルに圧縮
- "myapp.zip" を配布及び/または販売

ユーザーが"myapp.zip" を解凍し"run" をクリックしたときに、"welcome" アプリケーションの代わりにあなたのアプリケーションが起動します。ユーザーは事前に Python をインストールする必要さえありません。

Mac 用のバイナリを生成する過程も同じですが、"bat" ファイルは要りません。

### 14.3 web2py の最小構成

web2py をとても小さなメモリ量でデプロイする場合があります。その場合は余分なものをそぎ落とし最小構成にしたいです。

次に簡単な方法を示します:

- プロダクトマシン上に web2py をソースからインストール
- メイン web2py フォルダ内で次のコマンドを実行

```
1 python scripts/make_min_web2py.py /path/to/minweb2py
```

- 次にデプロイしたいアプリケーションを”/path/to/minweb2py/applications”内にコピー
- ”/path/to/minweb2py” を小さなメモリ量のサーバとしてデプロイ

この”make\_min\_web2py.py” スクリプトは、次の物を含まない最小構成の web2py ディストリビューションを構築します：

- admin
- examples
- welcome
- scripts
- 稼働に不必要的 contrib モジュール

デプロイテスト用の 1 つのファイルから成る”welcome” アプリを含んでいます。このスクリプトを見てください。一番上に、何を含み何が含まれていないかの詳細なリストが書かれています。

#### 14.4 外部 URL をフェッチ

Python には、URL フェッチするためのライブラリ `urllib` が同梱されています：

```
1 import urllib
2 page = urllib.urlopen('http://www.web2py.com').read()
```

通常は問題ないですが、`urllib` モジュールは、Google App Engine 上では動作しません。Google は、GAE 用に、URL をダウンロードするための異なる API を提供します。あなたのコードを移植しやすくするために、web2py は、他の Python 環境だけでなく GAE でも動く `fetch` 関数を同梱しています：

```
1 from google.tools import fetch
2 page = fetch('http://www.web2py.com')
```

#### 14.5 便利な日時表現 (Pretty dates)

日付を”2009-07-25 14:34:56” と表記するのではなく、”one years ago” と表記したほうが良い場合があります。このために、web2py は便利な関数を提供します：

```

1 import datetime
2 d = datetime.datetime(2009, 7, 25, 14, 34, 56)
3 from gluon.tools import prettydate
4 pretty_d = prettydate(d, T)

```

出力結果の国際化対応のために第2引数 (T) が渡される必要があります。

## 14.6 ジオコーディング

住所（例：“243 S Wabash Ave, Chicago, IL, USA”）を地理座標（緯度と経度）に変換する必要がある場合、web2py はそのように動作する関数を提供します。

```

1 from gluon.tools import geocode
2 address = '243 S Wabash Ave, Chicago, IL, USA'
3 (latitude, longitude) = geocode(address)

```

`geocode` 関数はネットワーク接続が必要で、ジオコーディング用の Google geocoding サービスに接続します。関数は失敗した場合に `(0, 0)` を返します。Google geocoding サービスはリクエスト回数に上限を定めることに注意し、それらのサービス同意書を確認すべきです。`geocode` 関数は `fetch` 関数上で実装されているため GAE でも動作します。

## 14.7 ページネーション

このレシピは、ページネーション時にデータベースアクセスを最小化することに役立つ技術です。例えば、データベースから行のリストを表示する必要があるが複数のページにまたがる場合です。

まず、データベースに 1000 個の素数を 1 番目から格納する `primes` アプリケーションを製作することから始めましょう。

`db.py` モデル: Here is the model `db.py`:

```

1 db = DAL('sqlite://primes.db')
2 db.define_table('prime', Field('value', 'integer'))
3 def isprime(p):
4     for i in range(2,p):
5         if p%i==0: return False
6     return True
7 if len(db().select(db.prime.id))==0:

```

```

8   p=2
9   for i in range(1000):
10      while not isprime(p): p+=1
11      db.prime.insert (value=p)
12      p+=1

```

そしてこのように、”default.py” コントローラにデータベースを読む `list_item` アクションを作ります:

```

1 def list_items():
2     if len(request.args): page=int(request.args[0])
3     else: page=0
4     items_per_page=20
5     limitby=(page*items_per_page, (page+1)*items_per_page+1)
6     rows=db().select(db.prime.ALL, limitby=limitby)
7     return dict(rows=rows, page=page, items_per_page=items_per_page)

```

このコードは必要な個数より 1 つ多いアイテムを選択することに注意してください (20+1)。範囲外の要素は次のページがあるかどうかを示します。

”default/list\_items.html” ビュー:

```

1 {{extend 'layout.html'}}
2
3 {{for i,row in enumerate(rows):}}
4 {{if i==items_per_page: break}}
5 {{=row.value}}<br />
6 {{pass}}
7
8 {{if page:}}
9 <a href="{{=URL(args=[page-1])}}">previous</a>
10 {{pass}}
11
12 {{if len(rows)>items_per_page:}}
13 <a href="{{=URL(args=[page+1])}}">next</a>
14 {{pass}}

```

この方法で処理ごとに 1 つの選択でページネーションができ、1 つの選択だけで必要な行だけを選んでくれます。

#### 14.8 httpserver.log とログファイルフォーマット

web2py の web サーバーはファイルへのリクエストを web2py のルートディレクトに置かれている:

```
1 httpserver.log
```

へ全て記録します。ファイル名やロケーションを web2py のコマンドラインオプションを使い指定できます。

新しいエントリーは、リクエストが処理されるたびにファイル末尾へ追加されます。それぞれの行はこのようになります:

```
1 127.0.0.1, 2008-01-12 10:41:20, GET, /admin/default/site, HTTP/1.1,
   200, 0.270000
```

フォーマットは:

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

各要素について

- ip はリクエストしてきたクライアントの IP アドレス
- timestamp はリクエストの日付と時間、ISO8601 フォーマット (YYYY-MM-DDT HH:MM:SS) で記述
- method は POST か GET のどちらか
- path はクライアントがリクエストしたパス
- protocol はクライアントへの送信に使用された HTTP プロトコル、たいていは HTTP/1.1
- [95] は HTTP のステータスコード 91
- time\_taken はサーバーがリクエストプロセスに掛けた時間の統計、単位は秒、アップロード/ダウンロードにかかる時間は含まれません

アプリケーションのリポジトリ [34] から、ログ解析のアプリケーションを利用できます。

このログは、mod\_wsgi 使用時は Apache ログと同じになるため、デフォルトでは有効になりません。

## 14.9 ダミーデータをデータベースに登録

テスト用にダミーデータをデータベーステーブルに登録できれば便利です。この目的に合う読解可能なダミーテキスを学習させたベイズ分類器を、web2py はすでに含んでいます。

簡単な使い方:

```
1 from gluon.contrib.populate import populate
2 populate(db.mytable, 100)
```

100 個のダミーレコードを db.mytable に挿入します。string フィールド用の短文、text フィールド用の長文、integer、double、date、datetime、time、boolean、その他のフィールドを知的に生成します。バリデータで指定された必要条件を尊重しようとします。フィールドに単語”name”が含まれているとダミーの名前を生成しようとします。リファレンスフィールドにより妥当なリファレンスを生成します。

もし B が A を参照する 2 つのテーブル (A と B) があるとき、A から先に登録し、次に B を登録します。

登録作業はトランザクションで実行されるので、とりわけリファレンスが含まれる場合は一度にたくさんのレコードを登録しようとしないでください。その代わりに、100 個ずつの登録を、コミット、ループすることを推奨します。

```
1 for i in range(10):
2     populate(db.mytable, 100)
3     db.commit()
```

ベイズ分類器にいくつかの文章を学習させ、学習データと似ているが無意味なダミーテキストを生成できます。

```
1 from gluon.contrib.populate import Learner, IUP
2 ell=Learner()
3 ell.learn('some very long input text ...')
4 print ell.generate(1000, prefix=None)
```

## 14.10 クレジットカード払いの承認

オンラインでのクレジットカードを認証する多様な方法があります。web2py は、いくつかの有名で実践的な API を提供します:

- Google Wallet [96]
- PayPal paypalcite
- Stripe.com [98]
- Authorize.net [99]
- DowCommerce [100]

最初 2 つの仕組みは外部のサービスへの受け取り人を証明するプロセスを上層に委任します。これがセキュリティにとって最良の方法であると同時に(あなたのアプリケーションは全てにおいてクレジットカード情報を取り扱わない)、それは面倒なプロセス(ユーザーは 2 回ログインする必要がある; 例えば、1 度目はあなたのアプリケーション、もう 1 回は Google に)を生成しあなたのアプリケーションが自動化された方法で繰り返し支払処理するのを許可しません。

さらなるコントロールが必要な時や自作のクレジットカード情報の入力フォームを生成したい時、すなわち、プログラムがクレジットカードからあなたのアカウントへ送金するための処理を依頼する時があります。

このため web2py は Stripe、Authorize.Net(モジュールは John Conde によって開発され若干修正されています)、DownCommerce との統合を既定で提供しています。Stripe は最も簡単に使え、低額の売買なら安くすみます(使用料は固定額ではなく取引価格の約 3%)。Authorize.net は高額の取引向けです(1 年間の定額の使用料が掛かり、加えて 1 回の取引に安い使用料が掛かります)。

Stripe と Authorize.net の場合、あなたのプログラムがクレジットカード情報を承認することを覚えておいてください。クレジットカード情報を記録する必要はありませんし、我々は法的必要条件が関係するため(Visa や Mastercard で取引する場合)あなたにそうすべきでないとアドバイスしますが、何回も会計をしたり Amazon one-click ボタンを再現するためにクレジットカード情報を記録したい場合もあります。

#### 14.10.1 Google Wallet

Google Wallet(Level 1) を簡単に使用する一番の方法は、あなたのページにボタンを埋め込み、そのボタンをクリック時に訪問者を Google から提供されている支払いページへリダイレクトする方法です。

まず始めに、あなたはこの URL で Google Merchant Account に登録する必要があります:

```
1 https://checkout.google.com/sell
```

Google にあなたの銀行口座情報を提供する必要があります。Google はあなたに merchant\_id と merchant\_key を割り当てます (これらを紛失せず、極秘で保管しておいてください)。

そして、ビューに次のコードを記述します:

```
1 {{from gluon.contrib.google_wallet import button}}
2 {{=button(merchant_id="123456789012345",
3         products=[dict(name="shoes",
4                         quantity=1,
5                         price=23.5,
6                         currency='USD',
7                         description="running shoes black")])}}
```

ボタンをクリック時に訪問者は、商品の支払いが可能な Google のページへリダイレクトされます。ここでの products は製品のリストであり、各製品はあなたが精算したいアイテムを記述したディクショナリ型のパラメータです (name, quantity, price, currency, description や他のオプションの説明は Google Wallet ドキュメントを見てください)。

この仕組みを使用する場合、在庫状況や訪問者のショッピングカートに基づき、button へ渡す値をプログラムで生成することもできます。

全ての税金と配送情報は Google 側で取り扱われます。支払情報も同様です。デフォルトだとあなたのアプリケーションに取引完了されたことは通知されないため、Google Merchant サイトにアクセスし、どの製品が購入され、支払いがされ、どの製品を購入者に届けるべきかを確認する必要があります。Google はこれらの情報を電子メールでもあなたに知らせてくれます。

しっかりと統合されたものが欲しい時には、Level 2 notification API を使用するべきです。これを使用した場合、Google へさらに多くの情報を渡すことができ、Google は製品購入を通知するためにあなたの API を呼びます。このことは、あなたに支払情報をアプリケーション内に保持することを許可しますが、Google Wallet と通信可能な Web サービスを公開することをあなたに要求します。

これは難しい処理ですが、そのような API はすでに実装されており、次の URL にプラグインとして公開されています。

```
1 http://web2py.com/plugins/static/web2py.plugin.google_checkout.w2p
```

ドキュメントはプラグイン自身に書かれています。

#### 14.10.2 Paypal

Paypal の統合について、ここでは説明しませんが、このリソースにたくさんの情報があります：

```
1 http://www.web2pyslices.com/main/slices/take_slice/9
```

#### 14.10.3 Stripe.com

これはおそらく最も簡単な方法の 1 つであり、クレジットカード払いを許可する柔軟な方法です。

Stripe.com に登録する必要がありますが、とても簡単です。実際、Strip はテスト用の API キーを認証情報を作成する以前に割り当ててくれます。

一度 API キーを取得しさえすれば、次のコードでクレジットカードを受け付けることができます：

```
1 from gluon.contrib.stripe import Stripe
2 stripe = Stripe(api_key)
3 d = stripe.charge(amount=100,
4                     currency='usd',
5                     card_number='4242424242424242',
6                     card_exp_month='5',
7                     card_exp_year='2012',
8                     card_cvc_check='123',
9                     description='the usual black shoes')
10 if d.get('paid',False):
11     # payment accepted
12 elif:
13     # error is in d.get('error', 'unknown')
```

レスポンス `d` はディクショナリ型ですので詳細は自分で確認してみてください。例で使われているカード番号はサンドボックスであり、いつも成功します。それぞれの取引は `d['id']` に格納されたトランザクション ID に関連付けられます。

Stripe は後から取引を照合することもできます：

```
1 d = Stripe(key).check(d['id'])
```

そして取引を返金できます:

```
1 r = Stripe(key).refund(d['id'])
2 if r.get('refunded', False):
3     # refund was successful
4 elif:
5     # error is in d.get('error', 'unkown')
```

Stripe でアプリケーション内に支払情報を簡単に保持できます。

アプリと Stripe 間の全ての通信は RESTfulWeb サービス上で行われます。Stripe は実際多くのサービスを公開しており、巨大な Python API セットを提供しています。Stripe のサイトには、よりたくさん的情報があります。

#### 14.10.4 Authorize.Net

もう 1 つの簡単にクレジットカード承認の方法として Authorize.net があります。いつもどおりあなたは登録する必要があり、`login` とトランザクションキー(`transaction`) を取得します。一度、それらを取得すると、Stripe の様に動作します: Another simple way to accept credit cards is to use Authorize.Net. As usual you need to register and you will obtain a `login` and a transaction key (`transkey`). Once you have them it works very much like Stripe does:

```
1 from gluon.contrib.AuthorizeNet import process
2 if process(creditcard='4427802641004797',
3             expiration="122012,
4             total=100.0, cvv='123', tax=None, invoice=None,
5             login='cnpdev4289', transkey='SR2P8g4jdEn7vFLQ', testmode=
6                 True):
7     # payment was processed
8 else:
    # payment was rejected
```

もし有効な Authorize.Net のアカウントを持っているなら、サンドボックス用の `login` と `transkey` をあなたのアカウントに置き換え、サンドボックスの代わりに本番のプラットフォーム上で動作させるため `testmode=False` に設定すべきです。そうすることで訪問者のクレジットカード情報を扱えます。

もし、`process` が `True` を返却したら、訪問者のクレジットカードからあなたの Authorize.Net アカウントへ送金されたことになります。あなたのプリケーション

ン内でデータと情報を一致させるために、`invoice` は、あなたが設定できる文字列であり、このトランザクションで Authorize.Net により保持されます。

次のコードは、今までより複雑な、より多くの変数を扱うワークフローの例です：

```

1 from gluon.contrib.AuthorizeNet import AIM
2 payment = AIM(login='cnpdev4289',
3                 transkey='SR2P8g4jdEn7vFLQ',
4                 testmod=True)
5 payment.setTransaction(creditcard, expiration, total, cvv, tax, invoice
6                         )
7 payment.setParameter('x_duplicate_window', 180) # three minutes
8     duplicate windows
9 payment.setParameter('x_cust_id', '1324')           # customer ID
10 payment.setParameter('x_first_name', 'Agent')
11 payment.setParameter('x_last_name', 'Smith')
12 payment.setParameter('x_company', 'Test Company')
13 payment.setParameter('x_address', '1234 Main Street')
14 payment.setParameter('x_city', 'Townsville')
15 payment.setParameter('x_state', 'NJ')
16 payment.setParameter('x_zip', '12345')
17 payment.setParameter('x_country', 'US')
18 payment.setParameter('x_phone', '800-555-1234')
19 payment.setParameter('x_description', 'Test Transaction')
20 payment.setParameter('x_customer_ip', socket.gethostname(socket.
21     gethostname())))
22 payment.setParameter('x_email', 'you@example.com')
23 payment.setParameter('x_email_customer', False)
24
25 payment.process()
26 if payment.isApproved():
27     print 'Response Code: ', payment.response.ResponseCode
28     print 'Response Text: ', payment.response.ResponseText
29     print 'Response: ', payment.getResultResponseFull()
30     print 'Transaction ID: ', payment.response.TransactionID
31     print 'CVV Result: ', payment.response.CVVResponse
32     print 'Approval Code: ', payment.response.AuthCode
33     print 'AVS Result: ', payment.response.AVSResponse
34 elif payment.isDeclined():
35     print 'Your credit card was declined by your bank'
36 elif payment.isError():
37     print 'It did not work'
38 print 'approved',payment.isApproved()
39 print 'declined',payment.isDeclined()
40 print 'error',payment.isError()

```

上記のコードはダミーのテストアカウントを使用していることに注意してください。Authorize.Net に登録（無料ではありません）する必要があり、AIM コンス

トラクタに login、transkey を渡し、testmode=True または False を設定します。

### 14.11 Dropbox API

Dropbox は、とても有名なストレージサービスです。ファイルをストレージするだけでなく、クラウド上に保存し全てのマシーンと同期してくれます。グループを作ったり、ユーザ個別やグループへ様々なフォルダに対し読み書きの権限を与えることが許可されています。全ファイルのバージョン履歴も保持してくれます。”Public” というフォルダを含んでおり、このフォルダ内のファイルには公開 URL が付けられます。共同作業に Dropbox は打って付けです。

Dropbox に簡単にアクセスできるようにするには

```
1 https://www.dropbox.com/developers
```

で登録を済まし、APP\_KEY と APP\_SECRET を取得します。一度それらを取得すると、ユーザの Dropbox の認証を使用できます。

”yourapp/private/dropbox.key” ファイルを作り、この様に書きます。

```
1 <APP_KEY>:<APP_SECERT>:app_folder
```

<APP\_KEY> と APP\_SECRET は、それぞれ key と secret です。

その次に、”models/db.py” へ次のコードを書きます:

```
1 from gluon.contrib.login_methods.dropbox_account import use_dropbox
2 use_janrain(auth, filename='private/dropbox.key')
3 mydropbox = auth.settings.login_form
```

このコードは、ユーザに Dropbox 認証を扱うアプリへのログインを許可し、あなたのプログラムはユーザの Dropbox アカウントにファイルをアップロードすることができます:

```
1 stream = open('localfile.txt', 'rb')
2 mydropbox.put('destfile.txt', stream)
```

ファイルのダウンロード:

```
1 stream = mydropbox.get('destfile.txt')
2 open('localfile.txt', 'wb').write(read)
```

ディレクトリのリストを取得:

```
1 contents = mydropbox.dir(path = '/')['contents']
```

## 14.12 Twitter API

ツイートを送信/取得する方法の手軽な例を示します。Twitter はシンプルな RESTful API を利用するため、サードパーティのライブラリを必要としません。

ツイートを送信する方法の例を示します:

```
1 def post_tweet(username,password,message):
2     import urllib, urllib2, base64
3     import gluon.contrib.simplejson as sj
4     args= urllib.urlencode([('status',message)])
5     headers={}
6     headers['Authorization'] = 'Basic ' +base64.b64encode(
7         username+':'+password)
8     req = urllib2.Request(
9         'http://twitter.com/statuses/update.json',
10        args, headers)
11    return sj.loads(urllib2.urlopen(req).read())
```

ツイートを受信する方法の例です:

```
1 def get_tweets():
2     user='web2py'
3     import urllib
4     import gluon.contrib.simplejson as sj
5     page = urllib.urlopen('http://twitter.com/%s?format=json' % user).
       read()
6     tweets=XML(sj.loads(page) ['#timeline'])
7     return dict(tweets=tweets)
```

より多くの複雑な操作には、Twitter API ドキュメントを参照してください。

## 14.13 仮想ファイルのストリーミング

悪意ある攻撃者にとって web サイトが脆弱かどうかをスキャンするのはよくあることです。彼らは Nessus といったセキュリティスキャナを使いスクリプトに脆弱性があるかを知ることで標的となるウェブサイトを探査します。スキャンされたマシンからの web サーバログや既知の脆弱性の多く示す Nessus データベースでの直接の解析は PHP スクリプトや ASP スクリプトにおけるものです。

我々は web2py を稼動させて以降も、それらの脆弱性が無いにもかかわらず、彼らから未だにスキャンされます。迷惑なので、これらの脆弱性に対応し、攻撃者に彼らの時間を空費することを理解させるのが望ましいです。

1つの方法は攻撃に対抗して.php、.asp、疑わしいダミーアクションへの全リクエストをリダイレクトすることで、攻撃者を長時間忙しくさせ続けることです。そのうち攻撃者はあきらめ、私たちを二度とスキャンしないでしょう。

このレシピは2つの部品を必要とします。

次のように”default.py”コントローラにjammerという専用のアプリケーションを作成します：

```
1 class Jammer():
2     def read(self,n): return 'x'*n
3 def jam(): return response.stream(Jammer(),40000)
```

このアクションが呼ばれたときに、一度に40000文字の”x”で満たされた無限のデータストリームを返します。

2つ目の要素は.php、.aps(大文字、小文字)などで終わるどのようなリクエストもこのコントローラにリダイレクトする”route.py”ファイルです。

```
1 route_in=(
2     ('.*\.(php|PHP|asp|ASP|jsp|JSP)', 'jammer/default/jam'),
3 )
```

一度目に攻撃を受けた際には小さなオーバヘッドしかないかもしれません、私たちの経験からして同じ攻撃者は2度と攻撃してきません。

第3版 - 翻訳: Yota Ichino レビュー: Omi Chiba

第4版 - 翻訳: Yota Ichino レビュー: Omi Chiba



## *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](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.pythonscurity.org>
- [21] [http://en.wikipedia.org/wiki/Secure\\_Sockets\\_Layer](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.gnu.org/licenses/lgpl.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](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](http://en.wikipedia.org/wiki/Cron\#crontab_syntax)
- [46] <http://www.xmlrpc.com/>
- [47] [http://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)
- [48] <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- [49] <http://www.modernizr.com/>
- [50] <http://getskeleton.com/>
- [51] <http://en.wikipedia.org/wiki/XML>
- [52] <http://www.w3.org/XML/>
- [53] <http://www.ez-css.org>
- [54] <http://en.wikipedia.org/wiki/XHTML>
- [55] <http://www.w3.org/TR/xhtml1/>
- [56] <http://www.w3schools.com/xhtml/>
- [57] <http://www.web2py.com/layouts>
- [58] <http://sourceforge.net/projects/zxjdbc/>
- [59] <http://pypi.python.org/pypi/psycopg2>
- [60] <https://github.com/petehunt/PyMySQL>
- [61] <http://sourceforge.net/projects/mysql-python>
- [62] [http://python.net/crew/atuining/cx\\_Oracle/](http://python.net/crew/atuining/cx_Oracle/)
- [63] <http://pyodbc.sourceforge.net/>
- [64] <http://kinterbasdb.sourceforge.net/>
- [65] <http://informixdb.sourceforge.net/>
- [66] <http://pypi.python.org/simple/ingresdbi/>
- [67] [http://docs.python.org/library/csv.html\#csv.QUOTE\\_ALL](http://docs.python.org/library/csv.html\#csv.QUOTE_ALL)

- [68] <http://www.faqs.org/rfcs/rfc2616.html>
- [69] <http://www.faqs.org/rfcs/rfc2396.html>
- [70] <http://tools.ietf.org/html/rfc3490>
- [71] <http://tools.ietf.org/html/rfc3492>
- [72] <http://mail.python.org/pipermail/python-list/2007-June/617126.html>
- [73] <http://mail.python.org/pipermail/python-list/2007-June/617126.html>
- [74] <http://www.recaptcha.net>
- [75] [http://en.wikipedia.org/wiki/Pluggable\\_Authentication\\_Modules](http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules)
- [76] <http://www.reportlab.org>
- [77] <http://gdwarner.blogspot.com/2008/10/brief-pyjamas-django-tutorial.html>
- [78] <http://en.wikipedia.org/wiki/AJAX>
- [79] <http://www.learningjquery.com/>
- [80] <http://ui.jquery.com/>
- [81] [http://en.wikipedia.org/wiki/Common\\_Gateway\\_Interface](http://en.wikipedia.org/wiki/Common_Gateway_Interface)
- [82] <http://www.apache.org/>
- [83] <http://httpd.apache.org/download.cgi>
- [84] [http://adal.chiriliuc.com/mod\\_wsgi/revision\\_1018\\_2.3/mod\\_wsgi\\_py25\\_apache](http://adal.chiriliuc.com/mod_wsgi/revision_1018_2.3/mod_wsgi_py25_apache)
- [85] [http://httpd.apache.org/docs/2.0/mod/mod\\_proxy.html](http://httpd.apache.org/docs/2.0/mod/mod_proxy.html)
- [86] <http://httpd.apache.org/docs/2.2/mod/core.html>
- [87] <http://sial.org/howto/openssl/self-signed>
- [88] <http://code.google.com/p/modwsgi/>
- [89] <http://www.lighttpd.net/>
- [90] <http://www.cherokee-project.com/download/>

- [91] <http://www.fastcgi.com/>
- [92] <http://www.apsis.ch/pound/>
- [93] <http://haproxy.1wt.eu/>
- [94] <http://pyamf.org/>
- [95] [http://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)
- [96] <https://wallet.google.com/manage>
- [97] <https://www.paypal.com/>
- [98] <https://stripe.com/>
- [99] <http://www.authorize.net/>
- [100] <http://www.dowcommerce.com/>
- [101] <http://sourceforge.net/projects/zxjdbc/>
- [102] <http://www.zentus.com/sqlitejdbc/>
- [103] <http://redis.io/>