Kay チュートリアル

開発環境の準備

下記のものをインストールします。

- Python-2.5
- App Engine SDK/Python
- Kay Framework
- ipython (推奨)

macports の python25 を使うばあいは、他に下記もインストールしましょう。 Kay のリポジトリバージョンを使うには mercurial も必要です。

- py25-hashlib
- py25-socket-ssl
- py25-pil
- py25-ipython (推奨)
- mercurial

mercurial を使用して、Kayのソースコードを下記のようにして clone できま す。

```
$ hg clone https://kay-framework.googlecode.com/hg/ kay
```

リリースバージョンを使う場合は http://code.google.com/p/kay-framework/downloads/list から最新版をダウ ンロードして下記のように解凍します。

```
$ tar zxvf kay-VERSION.tar.gz
```

> **Note**
>
> このチュートリアルでは Kay-0.10.0相当のバージョンを使用します。 Kay-0.10.0 が既にリリースされていればそちらのリリースバージョンを、 まだリリースされてなければリポジトリの最新版を使用してください。

もし zip 版の appengine SDK をインストールした場合は、下記のようにシン ボリックリンクを作ってください。appengine の SDK をインストーラーを使用 してインストールした場合には、この作業は必要ありません。

```
$ sudo ln -s /some/whare/google_appengine /usr/local/google_appengine
```

クイックスタート

プロジェクトの開始

新しいプロジェクトを始めるには、kay の manage.py スクリプトでプロジェ クトディレクトリの雛型を作成します。今後、プロジェクトの管理を行うには、 プロジェクトディレクトリ内で manage.py スクリプトを使用する事になり ます。

```
$ python kay/manage.py startproject myproject
$ tree myproject
myproject
|-- app.yaml
|-- kay -> /Users/tmatsuo/work/tmp/kay/kay
|-- manage.py -> /Users/tmatsuo/work/tmp/kay/manage.py
|-- settings.py
`-- urls.py

1 directory, 4 files
```

シンボリックリンクをサポートしているプラットフォームでは kay ディレクト リと manage.py へのシンボリックリンクが作成されます。後で kay の場 所を動かすと動かなくなりますが、そんな時はリンクを張り直してください。

アプリケーションを作る

Kay では、プロジェクト内にアプリケーションと呼ぶディレクトリを作成しそ こにプログラムを書くことになります。

出来たばかりの myproject ディレクトリに cd して、早速アプリケー ションを作りましょう。下記の例では myapp というアプリケーションを作 成しています。

```
$ cd myproject
$ python manage.py startapp myapp
$ tree myapp
myapp
|-- __init__.py
|-- models.py
|-- templates
|   `-- index.html
|-- urls.py
`-- views.py

1 directory, 5 files
```

アプリケーションが出来たら settings.py を編集して、プロジェクトに登 録する必要があります。

まずは settings.py の INSTALLED_APPS に myapp を追加します。 必要なら APP_MOUNT_POINTS を設定してどの url で動かすか設定する事も できます。下記の例では、アプリケーションをルート URL にマウントする例で す。

APP_MOUNT_POINTS を設定しない場合は /myapp というようにアプリケー ション名 URL にマウントされます。なお、ここでは認証用のアプリケーション である kay.auth も一緒に登録しています。

settings.py

```
#$/usr/bin/python
#..
#..

INSTALLED_APPS = (
  'kay.auth',
  'myapp',
)

APP_MOUNT_POINTS = {
  'myapp': '/',
}
```

ご覧になれば分かると思いますが INSTALLED_APPS はタプルで APP_MOUNT_POINTS は dict になっています。

アプリケーションを動かす

作ったアプリケーションを動かしてみましょう。下記のコマンドで開発サーバ が起動する筈です。

```
$ python manage.py runserver
INFO     2009-08-04 05:48:21,339 appengine_rpc.py:157] Server: appengine.google.com
...
...
INFO     ... Running application myproject on port 8080: http://localhost:8080
```

この状態で http://localhost:8080/ にアクセスすると、「Hello」と表示され る筈です。この文字列は、アプリケーションを作成した時に作られた view に より表示されています。

GAE にアップロードする

実際にコードを見る前に、GAE にアップロードしてみましょう。GAE にアップ ロードするには、あなたが持っている appid を app.yaml の application に設定してから、下記のコマンドを使用します。

```
$ python manage.py appcfg update
```

google アカウントのユーザー名とパスワードを聞かれる場合は、自分の情報を 入力します。成功すると、http://your-appid.appspot.com/ でアプリケーショ ンにアクセスできるようになります。

デフォルトアプリケーション

ここで、少しデフォルトのアプリケーションを見てみましょう。

myapp/urls.py

まずは urls.py です。このファイルでは、url と view の対応を定義します。

```
from kay.routing import (
  ViewGroup, Rule
)

view_groups = [
  ViewGroup(
    Rule('/', endpoint='index', view='myapp.views.index'),
```

```
  )
]
```

Rule の行で '/' -> 'myapp.views.index' という対応づけをしています。

myapp/views.py

次に views.py です。アプリケーション内の views.py には、所謂ビジネスロ ジックを書きます。

```
# -*- coding: utf-8 -*-
"""
myapp.views
"""

"""
import logging

from google.appengine.api import users
from google.appengine.api import memcache
from werkzeug import (
  unescape, redirect, Response,
)
from werkzeug.exceptions import (
  NotFound, MethodNotAllowed, BadRequest
)

from kay.utils import (
  render_to_response, reverse,
  get_by_key_name_or_404, get_by_id_or_404,
  to_utc, to_local_timezone, url_for, raise_on_dev
)
from kay.i18n import gettext as _
from kay.auth.decorators import login_required

"""

from kay.utils import render_to_response


# Create your views here.

def index(request):
  return render_to_response('myapp/index.html', {'message': 'Hello'})
```

ファイルの始めの方には、コメントとして良く使うであろう import 文が書い てありますので、必要に応じてコピーして使えます。モジュール本体には、関 数が一つ定義されています。

Kay では基本的に、関数を定義する事でビジネスロジックを書きます。実は関 数では無くても、callable であればなんでも構わないのですが、始めのうちは 関数を使っていきましょう。

index(request):

> view 関数は必ず request オブジェクトを第一引数として受け取ります。設 定によっては、追加でキーワード引数を受け取るようにもできますが、この index() は request のみです。
>
> view 関数は Response オブジェクトを返す必要があります。ここでは html テンプレートを使用して Response を生成するための関数 render_to_response を使っています。

render_to_response には、テンプレートの名前と、テンプレート内で 使用する値を辞書として渡す事ができます。

myapp/templates/index.html

最後に template を見てみます。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Top Page - myapp</title>
</head>
<body>
{{ message }}
</body>
</html>
```

Kay で使用している template engine は jinja2 です。当面二つの事を覚えて おきましょう。

- view から渡された値を表示するには {{}} で囲んで変数や関数呼び出し を記述します。
- 制御構造や jinja2 に対する命令は {% %} 形式のタグで記述します。こ の形式で記述するのは if..else や for 文などの制御構造および、テンプレー トの継承を意味する extends 文などです。

制御構造の使用例をひとつあげておきます。

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Top Page - myapp</title>
</head>
<body>
{% if message %}
  <div id="message">
    {{ message }}
  </div>
{% endif %}
</body>
</html>
```

この例では、message の表示部分を html の div で囲んでいます。さらに jinja2 の {% if %} を使用して、message に有意な値が入っている時のみ div を表示するようにしています。

当面はこれら二つの記法について覚えておいてください。

ユーザー認証

ユーザー認証を有効にするには、認証用ミドルウェアを有効にする必要があり ます。 このチュートリアルでは Google Account での認証を使う事とします。

ユーザー認証の設定

まずは settings.py に MIDDLEWARE_CLASSES というタプルを定義し kay.auth.middleware.AuthenticationMiddleware を設定します。

```
MIDDLEWARE_CLASSES = (
  'kay.auth.middleware.AuthenticationMiddleware',
```

```
)
```

ミドルウェア設定の最後にコンマが必要な事に気を付けてください。python で は要素が一つだけのタプルを定義する時には明示的なコンマが必要です。

このままでも認証自体は動くのですが、さらにユーザー情報を入れるモデルを 自分で定義する事をお勧めします。後でユーザーに紐付く情報を殖やしたくなっ た時など、独自のモデルを定義しておいた方が何かと楽です。

Google Account での認証を行う場合は kay.auth.models.GoogleUser を継 承したモデルを定義し、そのモデル名を settings.py の AUTH_USER_MODEL に記載します(文字列で構いません)。

myapp.models:

```
from google.appengine.ext import db
from kay.auth.models import GoogleUser

class MyUser(GoogleUser):
  pass
```

settings.py

```
AUTH_USER_MODEL = 'myapp.models.MyUser'
```

ここでは、モデルにはまだ独自プロパティを定義していませんが、将来のため に始めから独自モデルにしておく事をお勧めします。

使用方法

request.user

認証用ミドルウェアを有効にすると request.user が設定されます。ユー ザーがログインしていればユーザーエンティティ、そうでなければ kay.auth.models.AnonymousUser というクラスのインスタンスが入ってい ます。

これらのクラスに共通して使用できるアトリビュートとメソッドを示します。

- is_admin

  このアトリビュートは、そのユーザーが管理者かどうかを表す真偽値です。

- is_anonymous()

  このメソッドはユーザーがログインしていれば False をログインしてなけれ ば True を返します。

- is_authenticated()

  ログインしていれば True, そうでなければ False を返します。

template 内での使用例

下記のような断片を myapp/templates/index.html に入れてみましょう。

```
<div id="greeting">
  {% if request.user.is_anonymous() %}
    <a href="{{ create_login_url() }}">login</a>
  {% else %}
    Hello {{ request.user }}! <a href="{{ create_logout_url() }}">logout</a>
  {% endif %}
```

```
</div>
```

このコードは、ユーザーがログインしていなければログイン画面へのリンクを 表示し、ログインしていればログアウトするためのリンクを表示します。

デコレーター

認証しないとアクセスできないページを簡単に作るには、デコレーターを使い ます。ログインしないとアクセスできないようにするには kay.auth.decorators.login_required で、管理者アカウントにてログイン が必要なページを作成するには、 kay.auth.decorators.admin_required で view 関数を修飾します。

例:

```
from kay.utils import render_to_response
from kay.auth.decorators import login_required

# Create your views here.

@login_required
def index(request):
  return render_to_response('myapp/index.html', {'message': 'Hello'})
```

index へのアクセス時にログインが必要になっている事を確認してみましょう。

ゲストブックの実装 - Step1

このチュートリアルでは簡単なゲストブックを作成します。その過程で、Kay の機能をできるだけ紹介していく予定です。

まずはモデルとフォームの基本的な使い方についてご紹介します。

モデル定義

Kay でのモデル定義には基本的に appengine の db モジュールをそのまま使い ます。 kay.db パッケージ内に少しだけ Kay 独自のプロパティがあります。

ここではゲストブック用のモデルを定義してみましょう。

myapp/models.py:

```
from google.appengine.ext import db
from kay.auth.models import GoogleUser
import kay.db

# ...

class Comment(db.Model):
  user = kay.db.OwnerProperty()
  body = db.TextProperty(required=True)
  created = db.DateTimeProperty(auto_now_add=True)
```

user に割り当てた kay.db.OwnerProperty は Kay 独自のプロパティ で、現在ログイン中であるユーザーの key を自動で格納するためのプロパティ です。

body にはコメント本体を保存します。また created には作成日時が 自動で入ります。

フォーム定義

次に投稿用のフォームを作ります。テンプレート内に直に html フォームを書 いても動かす事はできますが、値の検証などの事も考えると kay.utils.forms パッケージを使用してフォームを作成した方が良いでしょ う。

フォーム定義の場所に特にきまりはありませんが myapp/forms.py に定義 しましょう。

myapp/forms.py:

```
# -*- coding: utf-8 -*-

from kay.utils import forms

class CommentForm(forms.Form):
  body = forms.TextField("Your Comment", required=True)
```

kay.utils.forms.Form を継承したクラスを定義する事によりフォームを作 成できます。このクラスでは b ody という名前で forms.TextField の インスタンスを指定しています。初めの引数はフォームフィールドのラベルに なります。 required に True を指定すると、このフィールドは入力が必 須になります。

他にどのようなフィールドがあるか、またそれらの使い方については kay.utils.forms パッケージについての ドキュメント も参照してくださ い。

ビュー定義

これらのモデルとフォームを使用して投稿用のビューを書きましょう。

myapp/views.py:

```
# -*- coding: utf-8 -*-
"""
myapp.views
"""

from werkzeug import redirect

from kay.utils import (
  render_to_response, url_for
)
from kay.auth.decorators import login_required

from myapp.models import Comment
from myapp.forms import CommentForm

# Create your views here.

@login_required
def index(request):
  form = CommentForm()
  if request.method == "POST" and form.validate(request.form):
    comment = Comment(body=form['body'])
    comment.put()
    return redirect(url_for('myapp/index'))
  return render_to_response('myapp/index.html',
                            {'form': form.as_widget()})
```

werkzeug.redirect, kay.utils.url_for と先程作成したモデル・フォー ムを import しています。 index ビューの内部ではフォームを作成し、 http メソッドが POST の時にはフォームのバリデーションを行っています。

フォームのバリデーションに成功した場合には Comment オブジェクトを作 成した後に、トップページへリダイレクトしています。

url_for というのは URL 逆引きのための関数で、引数で与えられた endpoint に対応する URL を返します。ここでデフォルトの urls.py を思い返 してみましょう。

```
view_groups = [
  ViewGroup(
    Rule('/', endpoint='index', view='myapp.views.index'),
  )
]
```

urls.py では endpoint として 'index' を指定していました。ですが逆引きの 時には 'myapp/index' を使用しています。実は Kay ではアプリケーション間 で endpoint が衝突する事を防ぐために、自動でアプリケーション名を前置し ます。

ですので、逆引きを行う時には urls.py で設定した endpoint そのままで は無く app_name/endpoint という形で endpoint を指定する必要がありま す。

テンプレート

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Top Page - myapp</title>
</head>
<body>
  <div id="greeting">
    {% if request.user.is_anonymous() %}
      <a href="{{ create_login_url() }}">login</a>
    {% else %}
      Hello {{ request.user }}! <a href="{{ create_logout_url() }}">logout</a>
    {% endif %}
  </div>

  <div id="main_form">
    {{ form()|safe }}
  </div>
</body>
</html>
```

ここまでで、フォームから投稿したコメントを datastore に保存できるように なりました。

開発用サーバーでデータが保存できるか試してみましょう。いくつかコメント を投稿した後に http://localhost:8080/_ah/admin へアクセスすると、データ ストアの中身を見る事ができます。

kind が myapp_comment というのが今回作成したコメントのエンティティ です。kind にもアプリケーション名が前置されている事がわかります。デフォ ルトでは Kay は クラス名 にアプリケーション名を前置して、さらに lowercase したものを kind として使用します。この挙動を抑制するには settings.py にて ADD_APP_PREFIX_TO_KIND を False に設定します。

ゲストブックの実装 - Step2

現在の実装だと投稿しても表示されないので実感がわきません。そこで最新20 件のコメントを表示するようにしてみましょう。

クエリーを使用する

myapp/views.py:

```
ITEMS_PER_PAGE = 20

# Create your views here.

@login_required
def index(request):
  form = CommentForm()
  if request.method == "POST" and form.validate(request.form):
    comment = Comment(body=form['body'])
    comment.put()
    return redirect(url_for('myapp/index'))
  query = Comment.all().order('-created')
  comments = query.fetch(ITEMS_PER_PAGE)
  return render_to_response('myapp/index.html',
                            {'form': form.as_widget(),
                             'comments': comments})
```

このコードでは、テンプレートに対して、最新20件のコメントを渡しています。

テンプレート内でのループ

テンプレートで受け取ったコメントを表示しましょう。

myapp/templates/index.html:

```
{% if comments %}
  <div id="comment_list">
    <ul>
    {% for comment in comments %}
      <li>{{ comment.body }}
        <span class="author"> by {{ comment.user }}</span>
    {% endfor %}
    </ul>
  </div>
{% endif %}
```

フォームを表示している部分の下に上記コードを追加しましょう。これで最新 20件のコメントが表示されるようになりました。

ゲストブックの実装 - Step 3

コメント投稿時に予め設定してあるカテゴリーを選べるようにしましょう。

モデルフォーム

まずはカテゴリーを保存するモデルを作り Comment クラスにもプロパティ を追加しましょう。

myapp/models.py:

```
class Category(db.Model):
  name = db.StringProperty(required=True)
```

```
  def __unicode__(self):
    return self.name

class Comment(db.Model):
  user = kay.db.OwnerProperty()
  category = db.ReferenceProperty(Category)
  body = db.StringProperty(required=True, verbose_name=u'Your Comment')
  created = db.DateTimeProperty(auto_now_add=True)
```

次にフォームですが、プロパティが増える度にフォームの実装も変更しなけれ ばならないのは面倒なので、モデルからフォームを自動生成できる仕組みを使 いましょう。

モデルからフォームを自動生成するには kay.utils.forms.modelform.ModelForm クラスを継承したフォームを作成 します。

```
# -*- coding: utf-8 -*-

from kay.utils import forms
from kay.utils.forms.modelform import ModelForm

from myapp.models import Comment

class CommentForm(ModelForm):
  class Meta:
    model = Comment
    exclude = ('user', 'created')
```

ModelForm の使いかたはまず ModelForm を継承したクラスを作成しま す。次にその中に内部クラス Meta を定義する事で設定を行います。 Meta 内で有効な attribute は下記の通りです。

- model

  フォーム生成の元にするモデルクラスを指定します。

- exclude

  モデルクラスに定義されているプロパティの中で、フォームに表示しないも のをタプルで指定します。次の fields とは排他的で、どちらか一方し か設定できません。

- fields

  モデルクラスに定義されているプロパティの中で、フォームに表示するもの をタプルで指定します。 fields に定義されていないプロパティは表示 されません。上記の exclude とは排他的で、どちらか一方しか設定でき ません。

- help_texts

  フォームフィールドにヘルプ文字列を与える時に使用します。フィールドの 名前をキーにした辞書として設定します。

管理用スクリプト

この段階で、カテゴリーを選ぶフォームはできているのですが、まだカテゴリー がありませんので、セレクトボックスには選択肢がありません。これは少し寂 しいので、カテゴリーを追加しましょう。ここでは、カスタムの管理用スクリ プトを追加してカテゴリーを追加できるようにしてみます。

myapp/management.py というファイルを下記の内容で作成しましょう。

```
# -*- coding: utf-8 -*-

from google.appengine.ext import db

from kay.management.utils import (
  print_status, create_db_manage_script
)
from myapp.models import Category

categories = {
  1: u'Programming',
  2: u'Testing',
  3: u'Management',
}

def create_categories():
  entities = []
  for idnum, name in categories.iteritems():
    entities.append(
      Category(name=name,
               key=db.Key.from_path(Category.kind(), idnum)))
  db.put(entities)
  print_status("Categories are created succesfully.")

def delete_categories():
  db.delete(Category.all().fetch(100))
  print_status("Categories are deleted succesfully.")

action_create_categories = create_db_manage_script(
  main_func=create_categories, clean_func=delete_categories,
  description="Create 'Category' entities")
```

うまくできると、 python manage.py の出力に下記のエントリが追加され ます:

```
create_categories:
  Create 'Category' entities

  -a, --appid                   string
  -h, --host                    string
  -p, --path                    string
  --no-secure
  -c, --clean
```

下記のようにして Category のエンティティを三つ追加できます。

- GAE にデプロイしたアプリに対して実行するには

  $ python manage.py create_categories

- 起動している開発用サーバーに対して実行するには

  $ python manage.py create_categories -h localhost:8080 --no-secure

Category を追加した後で、アプリケーションにアクセスしてみましょう。 三つの選択肢が選べるようになっていれば成功です。

Note

管理用スクリプトを追加する方法について詳しく知るには カスタムの管理 用スクリプトを書く方法 を参考にしてく ださい。