

22 实战

Day 1 - 搭建开发环境

搭建开发环境

首先，确认系统安装的Python版本是3.7.x:

```
$ python3 --version
Python 3.7.0
```

然后，用 `pip` 安装开发Web App需要的第三方库:

异步框架aiohttp:

```
$ pip3 install aiohttp
```

前端模板引擎jinja2:

```
$ pip3 install jinja2
```

MySQL 5.x数据库，从[官方网站](#)下载并安装，安装完毕后，请务必牢记root口令。为避免遗忘口令，建议直接把root口令设置为 `password`;

MySQL的Python异步驱动程序aiomysql:

```
$ pip3 install aiomysql
```

项目结构

选择一个工作目录，然后，我们建立如下的目录结构:

```
awesome-python3-webapp/ <-- 根目录
|
+- backup/                <-- 备份目录
|
+- conf/                  <-- 配置文件
|
+- dist/                  <-- 打包目录
|
+- www/                   <-- web目录，存放.py文件
| |
| +- static/              <-- 存放静态文件
| |
```

```
| +- templates/          <-- 存放模板文件
|
+- ios/                  <-- 存放iOS App工程
|
+- LICENSE               <-- 代码LICENSE
```

创建好项目的目录结构后，建议同时建立git仓库并同步至GitHub，保证代码修改的安全。

要了解git和GitHub的用法，请移步[Git教程](#)。

开发工具

自备，推荐用Sublime Text，请参考[使用文本编辑器](#)。

Day 2 - 编写Web App骨架

由于我们的Web App建立在asyncio的基础上，因此用aiohttp写一个基本的app.py：

```
import logging; logging.basicConfig(level=logging.INFO)

import asyncio, os, json, time
from datetime import datetime

from aiohttp import web

def index(request):
    return web.Response(body=b'<h1>Awesome</h1>')

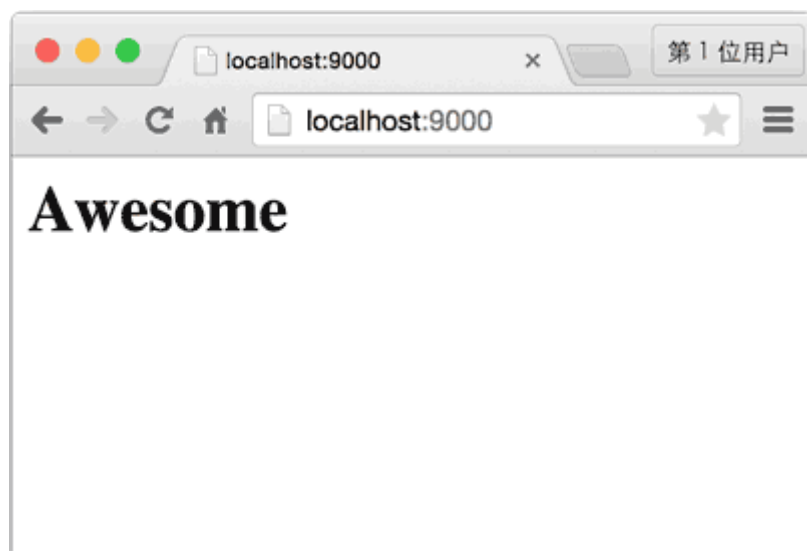
@asyncio.coroutine
def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', index)
    srv = yield from
loop.create_server(app.make_handler(), '127.0.0.1', 9000)
    logging.info('server started at
http://127.0.0.1:9000...')
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
loop.run_forever()
```

运行python app.py，Web App将在9000端口监听HTTP请求，并且对首页/进行响应：

```
$ python3 app.py
INFO:root:server started at http://127.0.0.1:9000...
```

这里我们简单地返回一个 `Awesome` 字符串，在浏览器中可以看到效果：



这说明我们的Web App骨架已经搭好了，可以进一步往里面添加更多的东西。

Day 3 - 编写ORM

在一个Web App中，所有数据，包括用户信息、发布的日志、评论等，都存储在数据库中。在`awesome-python3-webapp`中，我们选择MySQL作为数据库。

Web App里面有很多地方都要访问数据库。访问数据库需要创建数据库连接、游标对象，然后执行SQL语句，最后处理异常，清理资源。这些访问数据库的代码如果分散到各个函数中，势必无法维护，也不利于代码复用。

所以，我们要首先把常用的SELECT、INSERT、UPDATE和DELETE操作函数封装起来。

由于Web框架使用了基于`asyncio`的

这就是异步编程的一个原则：一旦决定使用异步，则系统每一层都必须是异步，“开弓没有回头箭”。

幸运的是 `aiomysql` 为MySQL数据库提供了异步IO的驱动。

创建连接池

我们需要创建一个全局的连接池，每个HTTP请求都可以从连接池中直接获取数据库连接。使用连接池的好处是不必频繁地打开和关闭数据库连接，而是能复用就尽量复用。

连接池由全局变量 `__pool` 存储，缺省情况下将编码设置为 `utf8`，自动提交事务：

```
@asyncio.coroutine
```

```
def create_pool(loop, **kw):
    logging.info('create database connection pool...')
    global __pool
    __pool = yield from aiomysql.create_pool(
        host=kw.get('host', 'localhost'),
        port=kw.get('port', 3306),
        user=kw['user'],
        password=kw['password'],
        db=kw['db'],
        charset=kw.get('charset', 'utf8'),
        autocommit=kw.get('autocommit', True),
        maxsize=kw.get('maxsize', 10),
        minsize=kw.get('minsize', 1),
        loop=loop
    )
```

Select

要执行SELECT语句，我们用 `select` 函数执行，需要传入SQL语句和SQL参数：

```
@asyncio.coroutine
def select(sql, args, size=None):
    log(sql, args)
    global __pool
    with (yield from __pool) as conn:
        cur = yield from conn.cursor(aiomysql.DictCursor)
        yield from cur.execute(sql.replace('?', '%s'),
                                args or ())
        if size:
            rs = yield from cur.fetchmany(size)
        else:
            rs = yield from cur.fetchall()
        yield from cur.close()
        logging.info('rows returned: %s' % len(rs))
    return rs
```

SQL语句的占位符是`?`，而MySQL的占位符是`%s`，`select()`函数在内部自动替换。注意要始终坚持使用带参数的SQL，而不是自己拼接SQL字符串，这样可以防止SQL注入攻击。

注意到`yield from`将调用一个子协程（也就是在一个协程中调用另一个协程）并直接获得子协程的返回结果。

如果传入`size`参数，就通过`fetchmany()`获取最多指定数量的记录，否则，通过`fetchall()`获取所有记录。

Insert, Update, Delete

要执行INSERT、UPDATE、DELETE语句，可以定义一个通用的`execute()`函数，因为这3种SQL的执行都需要相同的参数，以及返回一个整数表示影响的行数：

```
@asyncio.coroutine
def execute(sql, args):
    log(sql)
    with (yield from __pool) as conn:
        try:
            cur = yield from conn.cursor()
            yield from cur.execute(sql.replace('?', '%s'),
args)
            affected = cur.rowcount
            yield from cur.close()
        except BaseException as e:
            raise
    return affected
```

`execute()`函数和`select()`函数所不同的是，`cursor`对象不返回结果集，而是通过`rowcount`返回结果数。

ORM

有了基本的`select()`和`execute()`函数，我们就可以开始编写一个简单的ORM了。

设计ORM需要从上层调用者角度来设计。

我们先考虑如何定义一个`User`对象，然后把数据库表`users`和它关联起来。

```
from orm import Model, StringField, IntegerField

class User(Model):
    __table__ = 'users'

    id = IntegerField(primary_key=True)
    name = StringField()
```

注意到定义在`User`类中的`__table__`、`id`和`name`是类的属性，不是实例的属性。所以，在类级别上定义的属性用来描述`User`对象和表的映射关系，而实例属性必须通过`__init__()`方法去初始化，所以两者互不干扰：

```
# 创建实例：
user = User(id=123, name='Michael')
# 存入数据库：
user.insert()
# 查询所有User对象：
users = User.findAll()
```

定义Model

首先要定义的是所有ORM映射的基类 `Model`：

```
class Model(dict, metaclass=ModelMetaclass):

    def __init__(self, **kw):
        super(Model, self).__init__(**kw)

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(r"'Model' object has no attribute '%s'" % key)

    def __setattr__(self, key, value):
        self[key] = value

    def getValue(self, key):
        return getattr(self, key, None)

    def getValueOrDefault(self, key):
        value = getattr(self, key, None)
        if value is None:
            field = self.__mappings__[key]
            if field.default is not None:
                value = field.default() if callable(field.default) else field.default
                logging.debug('using default value for %s: %s' % (key, str(value)))
                setattr(self, key, value)
        return value
```

`Model` 从 `dict` 继承，所以具备所有 `dict` 的功能，同时又实现了特殊方法 `__getattr__()` 和 `__setattr__()`，因此又可以像引用普通字段那样写：

```
>>> user['id']
123
>>> user.id
123
```

以及 `Field` 和各种 `Field` 子类：

```
class Field(object):

    def __init__(self, name, column_type, primary_key,
        default):
        self.name = name
        self.column_type = column_type
        self.primary_key = primary_key
        self.default = default

    def __str__(self):
        return '<%s, %s:>' % (self.__class__.__name__,
            self.column_type, self.name)
```

映射 varchar 的 StringField:

```
class StringField(Field):

    def __init__(self, name=None, primary_key=False,
default=None, ddl='varchar(100)'):
        super().__init__(name, ddl, primary_key, default)
```

注意到 `Model` 只是一个基类，如何将具体的子类如 `User` 的映射信息读取出来呢？答案就是通过 `metaclass: ModelMetaClass`：

```
class ModelMetaclass(type):

    def __new__(cls, name, bases, attrs):
        # 排除Model类本身:
        if name=='Model':
            return type.__new__(cls, name, bases, attrs)
        # 获取table名称:
        tableName = attrs.get('__table__', None) or name
        logging.info('found model: %s (table: %s)' %
            (name, tableName))
        # 获取所有的Field和主键名:
        mappings = dict()
        fields = []
        primaryKey = None
        for k, v in attrs.items():
            if isinstance(v, Field):
                logging.info(' found mapping: %s ==> %s'
                    % (k, v))
                mappings[k] = v
                if v.primary_key:
                    # 找到主键:
                    if primaryKey:
                        raise RuntimeError('Duplicate
primary key for field: %s' % k)
                    primaryKey = k
            else:

```

```

        fields.append(k)
    if not primaryKey:
        raise RuntimeError('Primary key not found.')
    for k in mappings.keys():
        attrs.pop(k)
    escaped_fields = list(map(lambda f: '`%s`' % f,
fields))
    attrs['__mappings__'] = mappings # 保存属性和列的映射
关系
    attrs['__table__'] = tableName
    attrs['__primary_key__'] = primaryKey # 主键属性名
    attrs['__fields__'] = fields # 除主键外的属性名
    # 构造默认的SELECT, INSERT, UPDATE和DELETE语句:
    attrs['__select__'] = 'select `%s`, %s from `%s`'
% (primaryKey, ', '.join(escaped_fields), tableName)
    attrs['__insert__'] = 'insert into `%s` (%s, `%s`)
values (%s)' % (tableName, ', '.join(escaped_fields),
primaryKey, create_args_string(len(escaped_fields) + 1))
    attrs['__update__'] = 'update `%s` set %s where
`%s`=?' % (tableName, ', '.join(map(lambda f: '`%s`=?' %
(mappings.get(f).name or f), fields)), primaryKey)
    attrs['__delete__'] = 'delete from `%s` where
`%s`=?' % (tableName, primaryKey)
    return type.__new__(cls, name, bases, attrs)

```

这样，任何继承自Model的类（比如User），会自动通过ModelMetaclass扫描映射关系，并存储到自身的类属性如__table__、__mappings__中。

然后，我们往Model类添加class方法，就可以让所有子类调用class方法：

```

class Model(dict):

    ...

    @classmethod
    @asyncio.coroutine
    def find(cls, pk):
        ' find object by primary key. '
        rs = yield from select('%s where `%s`=?' %
(cls.__select__, cls.__primary_key__), [pk], 1)
        if len(rs) == 0:
            return None
        return cls(**rs[0])

```

User类现在就可以通过类方法实现主键查找：

```

user = yield from User.find('123')

```

往Model类添加实例方法，就可以让所有子类调用实例方法：


```
class Model(dict):

    ...

    @asyncio.coroutine
    def save(self):
        args = list(map(self.getValueOrDefault,
self.__fields__))

        args.append(self.getValueOrDefault(self.__primary_key__))
        rows = yield from execute(self.__insert__, args)
        if rows != 1:
            logging.warn('failed to insert record:
affected rows: %s' % rows)
```

这样，就可以把一个User实例存入数据库：

```
user = User(id=123, name='Michael')
yield from user.save()
```

最后一步是完善ORM，对于查找，我们可以实现以下方法：

- findAll() - 根据WHERE条件查找；
- findNumber() - 根据WHERE条件查找，但返回的是整数，适用于 `select count(*)` 类型的SQL。

以及 `update()` 和 `remove()` 方法。

所有这些方法都必须用 `@asyncio.coroutine` 装饰，变成一个协程。

调用时需要特别注意：

```
user.save()
```

没有任何效果，因为调用 `save()` 仅仅是创建了一个协程，并没有执行它。一定要用：

```
yield from user.save()
```

才真正执行了INSERT操作。

最后看看我们实现的ORM模块一共多少行代码？累计不到300多行。用Python写一个ORM是不是很容易呢？

Day 4 - 编写Model

有了ORM，我们就可以把Web App需要的3个表用 `Model` 表示出来：

```
import time, uuid
```

```

from orm import Model, StringField, BooleanField,
FloatField, TextField

def next_id():
    return '%015d%s000' % (int(time.time() * 1000),
uuid.uuid4().hex)

class User(Model):
    __table__ = 'users'

    id = StringField(primary_key=True, default=next_id,
ddl='varchar(50)')
    email = StringField(ddl='varchar(50)')
    passwd = StringField(ddl='varchar(50)')
    admin = BooleanField()
    name = StringField(ddl='varchar(50)')
    image = StringField(ddl='varchar(500)')
    created_at = FloatField(default=time.time)

class Blog(Model):
    __table__ = 'blogs'

    id = StringField(primary_key=True, default=next_id,
ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    name = StringField(ddl='varchar(50)')
    summary = StringField(ddl='varchar(200)')
    content = TextField()
    created_at = FloatField(default=time.time)

class Comment(Model):
    __table__ = 'comments'

    id = StringField(primary_key=True, default=next_id,
ddl='varchar(50)')
    blog_id = StringField(ddl='varchar(50)')
    user_id = StringField(ddl='varchar(50)')
    user_name = StringField(ddl='varchar(50)')
    user_image = StringField(ddl='varchar(500)')
    content = TextField()
    created_at = FloatField(default=time.time)

```

在编写ORM时，给一个Field增加一个 `default` 参数可以让ORM自己填入缺省值，非常方便。并且，缺省值可以作为函数对象传入，在调用 `save()` 时自动计算。

例如，主键 `id` 的缺省值是函数 `next_id`，创建时间 `created_at` 的缺省值是函数 `time.time`，可以自动设置当前日期和时间。

日期和时间用 `float` 类型存储在数据库中，而不是 `datetime` 类型，这么做的好处是不必关心数据库的时区以及时区转换问题，排序非常简单，显示的时候，只需要做一个 `float` 到 `str` 的转换，也非常容易。

初始化数据库表

如果表的数量很少，可以手写创建表的SQL脚本：

```
-- schema.sql

drop database if exists awesome;

create database awesome;

use awesome;

grant select, insert, update, delete on awesome.* to 'www-
data'@'localhost' identified by 'www-data';

create table users (
  `id` varchar(50) not null,
  `email` varchar(50) not null,
  `passwd` varchar(50) not null,
  `admin` bool not null,
  `name` varchar(50) not null,
  `image` varchar(500) not null,
  `created_at` real not null,
  unique key `idx_email` (`email`),
  key `idx_created_at` (`created_at`),
  primary key (`id`)
) engine=innodb default charset=utf8;

create table blogs (
  `id` varchar(50) not null,
  `user_id` varchar(50) not null,
  `user_name` varchar(50) not null,
  `user_image` varchar(500) not null,
  `name` varchar(50) not null,
  `summary` varchar(200) not null,
  `content` mediumtext not null,
  `created_at` real not null,
  key `idx_created_at` (`created_at`),
  primary key (`id`)
) engine=innodb default charset=utf8;

create table comments (
  `id` varchar(50) not null,
  `blog_id` varchar(50) not null,
  `user_id` varchar(50) not null,
  `user_name` varchar(50) not null,
  `user_image` varchar(500) not null,
```

```
`content` mediumtext not null,  
`created_at` real not null,  
key `idx_created_at` (`created_at`),  
primary key (`id`)  
) engine=innodb default charset=utf8;
```

如果表的数量很多，可以从 `Model` 对象直接通过脚本自动生成SQL脚本，使用更简单。

把SQL脚本放到MySQL命令行里执行：

```
$ mysql -u root -p < schema.sql
```

我们就完成了数据库表的初始化。

编写数据访问代码

接下来，就可以真正开始编写代码操作对象了。比如，对于 `User` 对象，我们就可以做如下操作：

```
import orm  
from models import User, Blog, Comment  
  
def test():  
    yield from orm.create_pool(user='www-data',  
password='www-data', database='awesome')  
  
    u = User(name='Test', email='test@example.com',  
passwd='1234567890', image='about:blank')  
  
    yield from u.save()  
  
for x in test():  
    pass
```

可以在MySQL客户端命令行查询，看看数据是不是正常存储到MySQL里面了。

Day 5 - 编写Web框架

在正式开始Web开发前，我们需要编写一个Web框架。

`aihttp` 已经是一个Web框架了，为什么我们还需要自己封装一个？

原因是从使用者的角度来说，`aihttp` 相对比较底层，编写一个URL的处理函数需要这么几步：

第一步，编写一个用 `@asyncio.coroutine` 装饰的函数：

```
@asyncio.coroutine
def handle_url_xxx(request):
    pass
```

第二步，传入的参数需要自己从 `request` 中获取：

```
url_param = request.match_info['key']
query_params = parse_qs(request.query_string)
```

最后，需要自己构造 `Response` 对象：

```
text = render('template', data)
return web.Response(text.encode('utf-8'))
```

这些重复的工作可以由框架完成。例如，处理带参数的URL `/blog/{id}` 可以这么写：

```
@get('/blog/{id}')
def get_blog(id):
    pass
```

处理 `query_string` 参数可以通过关键字参数 `**kw` 或者命名关键字参数接收：

```
@get('/api/comments')
def api_comments(*, page='1'):
    pass
```

对于函数的返回值，不一定是 `web.Response` 对象，可以是 `str`、`bytes` 或 `dict`。

如果希望渲染模板，我们可以这么返回一个 `dict`：

```
return {
    '__template__': 'index.html',
    'data': '...'
}
```

因此，Web框架的设计是完全从使用者出发，目的是让使用者编写尽可能少的代码。

编写简单的函数而非引入 `request` 和 `web.Response` 还有一个额外的好处，就是可以单独测试，否则，需要模拟一个 `request` 才能测试。

@get和@post

要把一个函数映射为一个URL处理函数，我们先定义 `@get()`：

```
def get(path):
    '''
    Define decorator @get('/path')
    '''
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            return func(*args, **kw)
        wrapper.__method__ = 'GET'
        wrapper.__route__ = path
        return wrapper
    return decorator
```

这样，一个函数通过 `@get()` 的装饰就附带了URL信息。

`@post` 与 `@get` 定义类似。

定义RequestHandler

URL处理函数不一定是一个 `coroutine`，因此我们用 `RequestHandler()` 来封装一个URL处理函数。

`RequestHandler` 是一个类，由于定义了 `__call__()` 方法，因此可以将其实例视为函数。

`RequestHandler` 目的就是URL函数中分析其需要接收的参数，从 `request` 中获取必要的参数，调用URL函数，然后把结果转换为 `web.Response` 对象，这样，就完全符合 `aiohttp` 框架的要求：

```
class RequestHandler(object):

    def __init__(self, app, fn):
        self._app = app
        self._func = fn
        ...

    @asyncio.coroutine
    def __call__(self, request):
        kw = ... 获取参数
        r = yield from self._func(**kw)
        return r
```

再编写一个 `add_route` 函数，用来注册一个URL处理函数：

```
def add_route(app, fn):
    method = getattr(fn, '__method__', None)
    path = getattr(fn, '__route__', None)
    if path is None or method is None:
        raise ValueError('@get or @post not defined in
%s.' % str(fn))
    if not asyncio.iscoroutinefunction(fn) and not
inspect.isgeneratorfunction(fn):
        fn = asyncio.coroutine(fn)
    logging.info('add route %s %s => %s(%s)' % (method,
path, fn.__name__, ',
'.join(inspect.signature(fn).parameters.keys()))
    app.router.add_route(method, path, RequestHandler(app,
fn))
```

最后一步，把很多次 `add_route()` 注册的调用：

```
add_route(app, handles.index)
add_route(app, handles.blog)
add_route(app, handles.create_comment)
...
```

变成自动扫描：

```
# 自动把handler模块的所有符合条件的函数注册了：
add_routes(app, 'handlers')
```

`add_routes()` 定义如下：

```
def add_routes(app, module_name):
    n = module_name.rfind('.')
    if n == (-1):
        mod = __import__(module_name, globals(), locals())
    else:
        name = module_name[n+1:]
        mod = getattr(__import__(module_name[:n],
globals(), locals(), [name]), name)
    for attr in dir(mod):
        if attr.startswith('_'):
            continue
        fn = getattr(mod, attr)
        if callable(fn):
            method = getattr(fn, '__method__', None)
            path = getattr(fn, '__route__', None)
            if method and path:
                add_route(app, fn)
```

最后，在 `app.py` 中加入 `middleware`、`jinja2` 模板和自注册的支持：

```

app = web.Application(loop=loop, middlewares=[
    logger_factory, response_factory
])
init_jinja2(app, filters=dict(datetime=datetime_filter))
add_routes(app, 'handlers')
add_static(app)

```

middleware

middleware 是一种拦截器，一个URL在被某个函数处理前，可以经过一系列的 **middleware** 的处理。

一个 **middleware** 可以改变URL的输入、输出，甚至可以决定不继续处理而直接返回。**middleware** 的用处就在于把通用的功能从每个URL处理函数中拿出来，集中放到一个地方。例如，一个记录URL日志的 **logger** 可以简单定义如下：

```

@asyncio.coroutine
def logger_factory(app, handler):
    @asyncio.coroutine
    def logger(request):
        # 记录日志:
        logging.info('Request: %s %s' % (request.method,
            request.path))
        # 继续处理请求:
        return (yield from handler(request))
    return logger

```

而 **response** 这个 **middleware** 把返回值转换为 **web.Response** 对象再返回，以保证满足 **aiohttp** 的要求：

```

@asyncio.coroutine
def response_factory(app, handler):
    @asyncio.coroutine
    def response(request):
        # 结果:
        r = yield from handler(request)
        if isinstance(r, web.StreamResponse):
            return r
        if isinstance(r, bytes):
            resp = web.Response(body=r)
            resp.content_type = 'application/octet-stream'
            return resp
        if isinstance(r, str):
            resp = web.Response(body=r.encode('utf-8'))
            resp.content_type = 'text/html; charset=utf-8'
            return resp
        if isinstance(r, dict):
            ...

```


有了这些基础设施，我们就可以专注地往 `handlers` 模块不断添加URL处理函数了，可以极大地提高开发效率。

Day 6 - 编写配置文件

有了Web框架和ORM框架，我们就可以开始装配App了。

通常，一个Web App在运行时都需要读取配置文件，比如数据库的用户名、口令等，在不同的环境中运行时，Web App可以通过读取不同的配置文件来获得正确的配置。

由于Python本身语法简单，完全可以直接用Python源代码来实现配置，而不需要再解析一个单独的 `.properties` 或者 `.yaml` 等配置文件。

默认的配置文件应该完全符合本地开发环境，这样，无需任何设置，就可以立刻启动服务器。

我们把默认的配置文件命名为 `config_default.py`：

```
# config_default.py

configs = {
    'db': {
        'host': '127.0.0.1',
        'port': 3306,
        'user': 'www-data',
        'password': 'www-data',
        'database': 'awesome'
    },
    'session': {
        'secret': 'AwESOMe'
    }
}
```

上述配置文件简单明了。但是，如果要部署到服务器时，通常需要修改数据库的 `host` 等信息，直接修改 `config_default.py` 不是一个好办法，更好的方法是编写一个 `config_override.py`，用来覆盖某些默认设置：

```
# config_override.py

configs = {
    'db': {
        'host': '192.168.0.100'
    }
}
```

把 `config_default.py` 作为开发环境的标准配置，把 `config_override.py` 作为生产环境的标准配置，我们就可以既方便地在本地开发，又可以随时把应用部署到服务器上。

应用程序读取配置文件需要优先从 `config_override.py` 读取。为了简化读取配置文件，可以把所有配置读取到统一的 `config.py` 中：

```
# config.py
configs = config_default.configs

try:
    import config_override
    configs = merge(configs, config_override.configs)
except ImportError:
    pass
```

这样，我们就完成了App的配置。

Day 7 - 编写MVC

现在，ORM框架、Web框架和配置都已就绪，我们可以开始编写一个最简单的MVC，把它们全部启动起来。

通过Web框架的 `@get` 和ORM框架的Model支持，可以很容易地编写一个处理首页URL的函数：

```
@get('/')
def index(request):
    users = yield from User.findAll()
    return {
        '__template__': 'test.html',
        'users': users
    }
```

'__template__' 指定的模板文件是 `test.html`，其他参数是传递给模板的数据，所以我们在模板的根目录 `templates` 下创建 `test.html`：

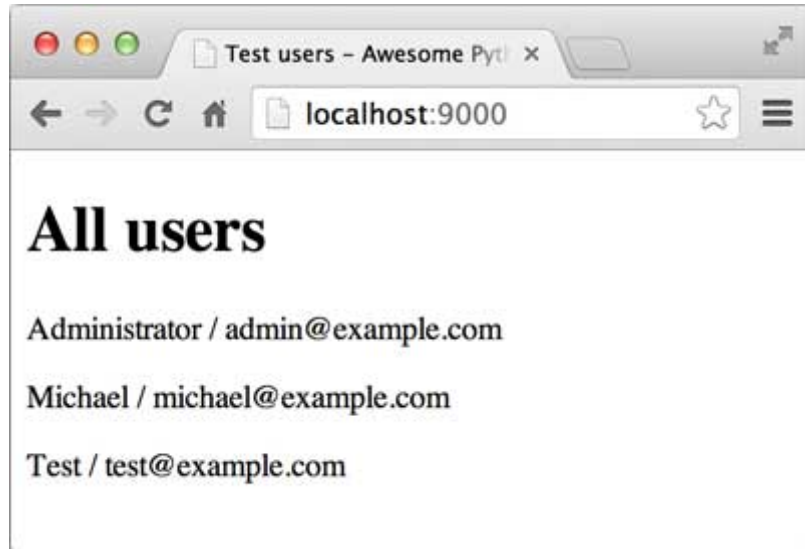
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Test users - Awesome Python webapp</title>
</head>
<body>
    <h1>All users</h1>
    {% for u in users %}
    <p>{{ u.name }} / {{ u.email }}</p>
    {% endfor %}
</body>
</html>
```

接下来，如果一切顺利，可以用命令行启动Web服务器：

```
$ python3 app.py
```

然后，在浏览器中访问 <http://localhost:9000/>。

如果数据库的 `users` 表什么内容也没有，你就无法在浏览器中看到循环输出的内容。可以自己在MySQL的命令行里给 `users` 表添加几条记录，然后再访问：



Day 8 - 构建前端

虽然我们跑通了一个最简单的MVC，但是页面效果肯定不会让人满意。

对于复杂的HTML前端页面来说，我们需要一套基础的CSS框架来完成页面布局和基本样式。另外，jQuery作为操作DOM的JavaScript库也必不可少。

从零开始写CSS不如直接从一个已有的功能完善的CSS框架开始。有很多CSS框架可供选择。我们这次选择uikit这个强大的CSS框架。它具备完善的响应式布局，漂亮的UI，以及丰富的HTML组件，让我们能轻松设计出美观而简洁的页面。

可以从uikit首页下载打包的资源文件。

所有的静态资源文件我们统一放到 `www/static` 目录下，并按照类别归类：

```
static/
+- css/
| +- addons/
| | +- uikit.addons.min.css
| | +- uikit.almost-flat.addons.min.css
| | +- uikit.gradient.addons.min.css
| +- awesome.css
| +- uikit.almost-flat.addons.min.css
| +- uikit.gradient.addons.min.css
| +- uikit.min.css
+- fonts/
| +- fontawesome-webfont.eot
| +- fontawesome-webfont.ttf
```

```
| +- fontawesome-webfont.woff
| +- FontAwesome.otf
+- js/
  +- awesome.js
  +- html5.js
  +- jquery.min.js
  +- uikit.min.js
```

由于前端页面肯定不止首页一个页面，每个页面都有相同的页眉和页脚。如果每个页面都是独立的HTML模板，那么我们在修改页眉和页脚的时候，就需要把每个模板都改一遍，这显然是没有效率的。

常见的模板引擎已经考虑到了页面上重复的HTML部分的复用问题。有的模板通过include把页面拆成三部分：

```
<html>
  <% include file="inc_header.html" %>
  <% include file="index_body.html" %>
  <% include file="inc_footer.html" %>
</html>
```

这样，相同的部分 `inc_header.html` 和 `inc_footer.html` 就可以共享。

但是include方法不利于页面整体结构的维护。jinja2的模板还有另一种“继承”方式，实现模板的复用更简单。

“继承”模板的方式是通过编写一个“父模板”，在父模板中定义一些可替换的block（块）。然后，编写多个“子模板”，每个子模板都可以只替换父模板定义的block。比如，定义一个最简单的父模板：

```
<!-- base.html -->
<html>
  <head>
    <title>{% block title%} 这里定义了一个名为title的
    block {% endblock %}</title>
  </head>
  <body>
    {% block content %} 这里定义了一个名为content的block
    {% endblock %}
  </body>
</html>
```

对于子模板 `a.html`，只需要把父模板的 `title` 和 `content` 替换掉：

```
{% extends 'base.html' %}

{% block title %} A {% endblock %}

{% block content %}
    <h1>Chapter A</h1>
    <p>blablabla...</p>
{% endblock %}
```

对于子模板 `b.html`，如法炮制：

```
{% extends 'base.html' %}

{% block title %} B {% endblock %}

{% block content %}
    <h1>Chapter B</h1>
    <ul>
        <li>list 1</li>
        <li>list 2</li>
    </ul>
{% endblock %}
```

这样，一旦定义好父模板的整体布局和CSS样式，编写子模板就会非常容易。

让我们通过uikit这个CSS框架来完成父模板 `__base__.html` 的编写：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    {% block meta %}<!-- block meta -->{% endblock %}
    <title>{% block title %} ? {% endblock %} - Awesome
    Python webapp</title>
    <link rel="stylesheet"
    href="/static/css/uikit.min.css">
    <link rel="stylesheet"
    href="/static/css/uikit.gradient.min.css">
    <link rel="stylesheet" href="/static/css/awesome.css"
    />
    <script src="/static/js/jquery.min.js"></script>
    <script src="/static/js/md5.js"></script>
    <script src="/static/js/uikit.min.js"></script>
    <script src="/static/js/awesome.js"></script>
    {% block beforehead %}<!-- before head -->{% endblock
    %}
</head>
<body>
    <nav class="uk-navbar uk-navbar-attached uk-margin-
    bottom">
```

```

        <div class="uk-container uk-container-center">
            <a href="/" class="uk-navbar-
brand">Awesome</a>
            <ul class="uk-navbar-nav">
                <li data-url="blogs"><a href="/"><i
class="uk-icon-home"></i> 日志</a></li>
                <li><a target="_blank" href="#"><i
class="uk-icon-book"></i> 教程</a></li>
                <li><a target="_blank" href="#"><i
class="uk-icon-code"></i> 源码</a></li>
            </ul>
            <div class="uk-navbar-flip">
                <ul class="uk-navbar-nav">
                    {% if user %}
                        <li class="uk-parent" data-uk-
dropdown>
                            <a href="#0"><i class="uk-icon-
user"></i> {{ user.name }}</a>
                            <div class="uk-dropdown uk-
dropdown-navbar">
                                <ul class="uk-nav uk-nav-
navbar">
                                    <li><a href="/signin"><i
class="uk-icon-sign-in"></i> 登录</a></li>
                                    <li><a href="/register"><i class="uk-
icon-edit"></i> 注册</a></li>
                                </ul>
                            </div>
                        </li>
                    {% else %}
                        <li><a href="/signin"><i class="uk-
icon-sign-in"></i> 登录</a></li>
                        <li><a href="/register"><i class="uk-
icon-edit"></i> 注册</a></li>
                    {% endif %}
                </ul>
            </div>
        </div>
    </nav>

    <div class="uk-container uk-container-center">
        <div class="uk-grid">
            <!-- content -->
            {% block content %}
            {% endblock %}
            <!-- // content -->
        </div>
    </div>

    <div class="uk-margin-large-top" style="background-
color:#eee; border-top:1px solid #ccc;">
        <div class="uk-container uk-container-center uk-
text-center">

```

```

        <div class="uk-panel uk-margin-top uk-margin-
bottom">
            <p>
                <a target="_blank" href="#" class="uk-
icon-button uk-icon-weibo"></a>
                <a target="_blank" href="#" class="uk-
icon-button uk-icon-github"></a>
                <a target="_blank" href="#" class="uk-
icon-button uk-icon-linkedin-square"></a>
                <a target="_blank" href="#" class="uk-
icon-button uk-icon-twitter"></a>
            </p>
            <p>Powered by <a href="#">Awesome Python
Webapp</a>. Copyright &copy; 2014. [<a href="/manage/"
target="_blank">Manage</a>]</p>
            <p><a href="http://www.liaoxuefeng.com/"
target="_blank">www.liaoxuefeng.com</a>. All rights
reserved.</p>
                <a target="_blank" href="#"><i class="uk-
icon-html5" style="font-size:64px; color: #444;"></i></a>
            </div>
        </div>
    </div>
</body>
</html>

```

`__base__.html` 定义的几个block作用如下:

用于子页面定义一些meta, 例如rss feed:

```
{% block meta %} ... {% endblock %}
```

覆盖页面的标题:

```
{% block title %} ... {% endblock %}
```

子页面可以在``标签关闭前插入JavaScript代码:

```
{% block beforehead %} ... {% endblock %}
```

子页面的content布局和内容:

```

{% block content %}
    ...
{% endblock %}

```

我们把首页改造一下, 从`__base__.html` 继承一个 `blogs.html`:

```

{% extends '__base__.html' %}

{% block title %}日志{% endblock %}

{% block content %}

    <div class="uk-width-medium-3-4">
        {% for blog in blogs %}
            <article class="uk-article">
                <h2><a href="/blog/{{ blog.id }}">{{
blog.name }}</a></h2>
                <p class="uk-article-meta">发表于{{
blog.created_at}}</p>
                <p>{{ blog.summary }}</p>
                <p><a href="/blog/{{ blog.id }}">继续阅读
<i class="uk-icon-angle-double-right"></i></a></p>
            </article>
            <hr class="uk-article-divider">
        {% endfor %}
    </div>

    <div class="uk-width-medium-1-4">
        <div class="uk-panel uk-panel-header">
            <h3 class="uk-panel-title">友情链接</h3>
            <ul class="uk-list uk-list-line">
                <li><i class="uk-icon-thumbs-o-up"></i> <a
target="_blank" href="#">编程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a
target="_blank" href="#">读书</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a
target="_blank" href="#">Python教程</a></li>
                <li><i class="uk-icon-thumbs-o-up"></i> <a
target="_blank" href="#">Git教程</a></li>
            </ul>
        </div>
    </div>

{% endblock %}

```

相应地，首页URL的处理函数更新如下：


```
@get('/')
def index(request):
    summary = 'Lorem ipsum dolor sit amet, consectetur
adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua.'
    blogs = [
        Blog(id='1', name='Test Blog', summary=summary,
created_at=time.time()-120),
        Blog(id='2', name='Something New',
summary=summary, created_at=time.time()-3600),
        Blog(id='3', name='Learn Swift', summary=summary,
created_at=time.time()-7200)
    ]
    return {
        '__template__': 'blogs.html',
        'blogs': blogs
    }
```

Blog的创建日期显示的是一个浮点数，因为它是由这段模板渲染出来的：

```
<p class="uk-article-meta">发表于{{ blog.created_at }}</p>
```

解决方法是通过jinja2的filter（过滤器），把一个浮点数转换成日期字符串。我们来编写一个 `datetime` 的filter，在模板里用法如下：

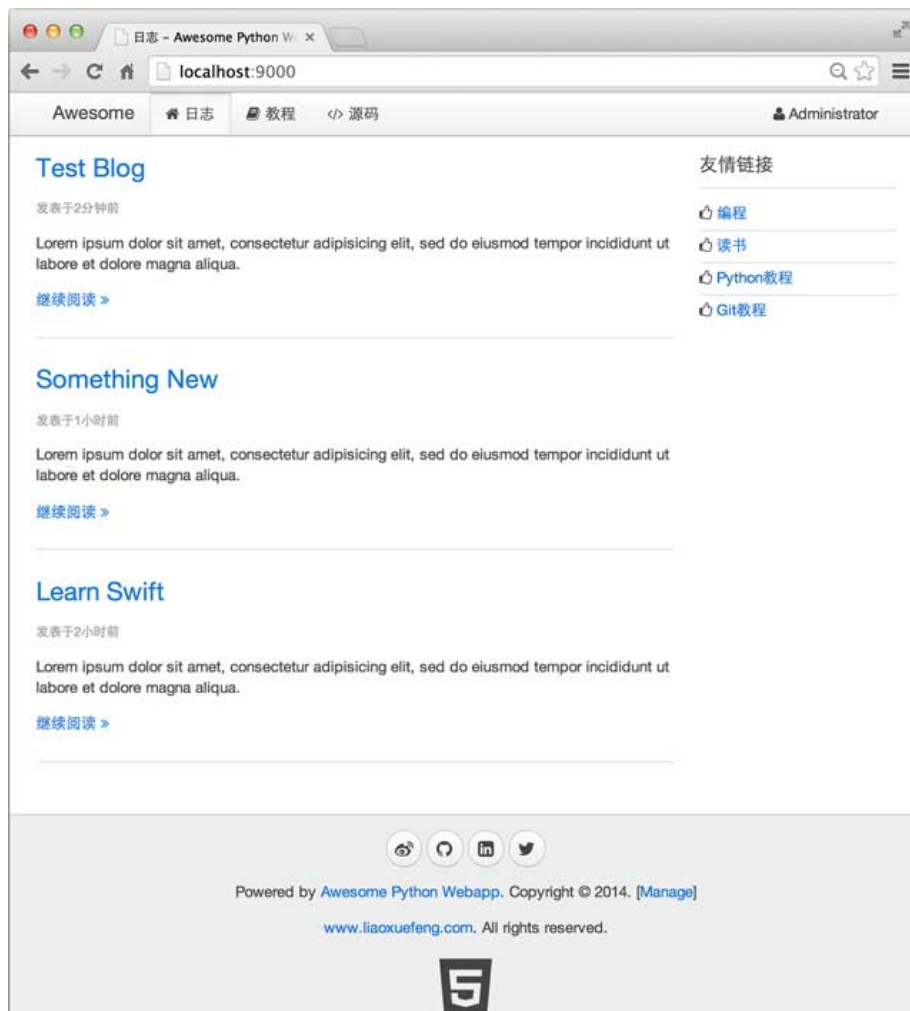
```
<p class="uk-article-meta">发表于{{
blog.created_at|datetime }}</p>
```

filter需要在初始化jinja2时设置。相关代码如下：

```
def datetime_filter(t):
    delta = int(time.time() - t)
    if delta < 60:
        return '1分钟前'
    if delta < 3600:
        return '%s分钟前' % (delta // 60)
    if delta < 86400:
        return '%s小时前' % (delta // 3600)
    if delta < 604800:
        return '%s天前' % (delta // 86400)
    dt = datetime.fromtimestamp(t)
    return '%s年%s月%s日' % (dt.year, dt.month, dt.day)

...
init_jinja2(app, filters=dict(datetime=datetime_filter))
...
```

现在，完善的首页显示如下：



Day 9 - 编写API

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取一篇Blog，输入<http://localhost:9000/blog/123>，就可以看到id为123的Blog页面，但这个结果是HTML页面，它同时混合包含了Blog的数据和Blog的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Blog的数据。

如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取<http://localhost:9000/api/blogs/123>，如果能直接返回Blog的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

一个API也是一个URL的处理函数，我们希望能直接通过一个@api来把函数变成JSON格式的REST API，这样，获取注册用户可以用一个API实现如下：

```
@get('/api/users')
def api_get_users(*, page='1'):
    page_index = get_page_index(page)
    num = yield from User.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, users=())
    users = yield from User.findAll(orderBy='created_at
desc', limit=(p.offset, p.limit))
    for u in users:
        u.passwd = '*****'
    return dict(page=p, users=users)
```

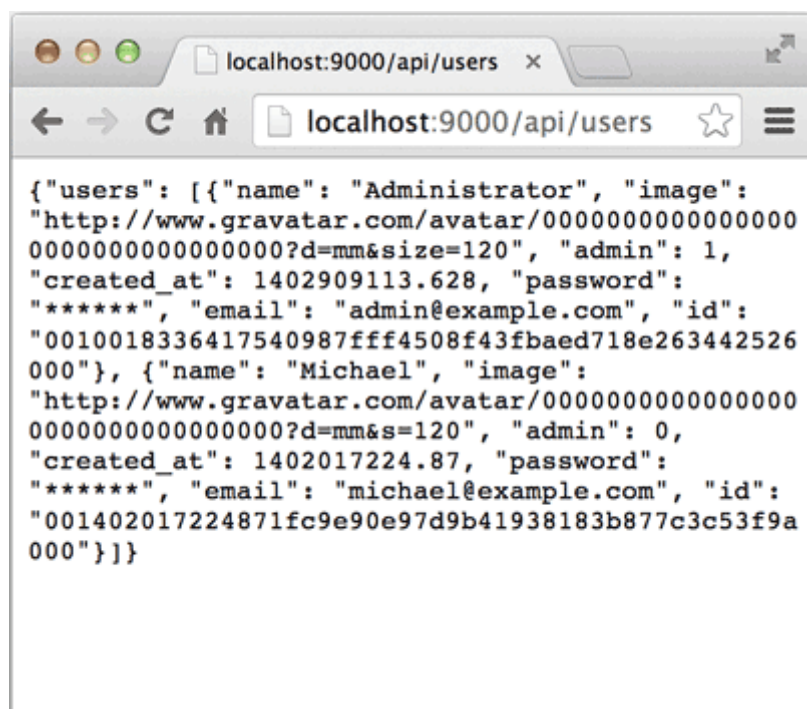
只要返回一个dict，后续的response这个middleware就可以把结果序列化为JSON并返回。

我们需要对Error进行处理，因此定义一个APIError，这种Error是指API调用时发生了逻辑错误（比如用户不存在），其他的Error视为Bug，返回的错误代码为internalerror。

客户端调用API时，必须通过错误代码来区分API调用是否成功。错误代码是用来告诉调用者出错的原因。很多API用一个整数表示错误码，这种方式很难维护错误码，客户端拿到错误码还需要查表得知错误信息。更好的方式是用字符串表示错误代码，不需要看文档也能猜到错误原因。

可以在浏览器直接测试API，例如，输入

http://localhost:9000/api/users，就可以看到返回的JSON：



用户管理是绝大部分Web网站都需要解决的问题。用户管理涉及到用户注册和登录。

用户注册相对简单，我们可以先通过API把用户注册这个功能实现了：

```
_RE_EMAIL = re.compile(r'^[a-z0-9\.\-\_]+\@[a-z0-9\.\-\_]+\n(\.[a-z0-9\.\-\_]+){1,4}$')
_RE_SHA1 = re.compile(r'^[0-9a-f]{40}$')

@post('/api/users')
def api_register_user(*, email, name, passwd):
    if not name or not name.strip():
        raise APIValueError('name')
    if not email or not _RE_EMAIL.match(email):
        raise APIValueError('email')
    if not passwd or not _RE_SHA1.match(passwd):
        raise APIValueError('passwd')
    users = yield from User.findAll('email=?', [email])
    if len(users) > 0:
        raise APIError('register:failed', 'email', 'Email\nis already in use.')
    uid = next_id()
    sha1_passwd = '%s:%s' % (uid, passwd)
    user = User(id=uid, name=name.strip(), email=email,\npasswd=hashlib.sha1(sha1_passwd.encode('utf-\n8')).hexdigest(),\nimage='http://www.gravatar.com/avatar/%s?d=mm&s=120' %\nhashlib.md5(email.encode('utf-8')).hexdigest())
    yield from user.save()
    # make session cookie:
    r = web.Response()
    r.set_cookie(COOKIE_NAME, user2cookie(user, 86400),\nmax_age=86400, httponly=True)
    user.passwd = '*****'
    r.content_type = 'application/json'
    r.body = json.dumps(user,\nensure_ascii=False).encode('utf-8')
    return r
```

注意用户口令是客户端传递的经过SHA1计算后的40位Hash字符串，所以服务器端并不知道用户的原始口令。

接下来可以创建一个注册页面，让用户填写注册表单，然后，提交数据到注册用户的API：

```
{% extends '__base__.html' %}

{% block title %}注册{% endblock %}

{% block beforehead %}
```

```

<script>
function validateEmail(email) {
    var re = /^[a-z0-9\.\-\_\_]+\@[a-z0-9\-\_\_]+\(\.[a-z0-9\-\_\_]+\){1,4}$/;
    return re.test(email.toLowerCase());
}
$(function () {
    var vm = new Vue({
        el: '#vm',
        data: {
            name: '',
            email: '',
            password1: '',
            password2: ''
        },
        methods: {
            submit: function (event) {
                event.preventDefault();
                var $form = $('#vm');
                if (! this.name.trim()) {
                    return $form.showFormError('请输入名字');
                }
                if (!
validateEmail(this.email.trim().toLowerCase())) {
                    return $form.showFormError('请输入正确的
Email地址');
                }
                if (this.password1.length < 6) {
                    return $form.showFormError('口令长度至少
为6个字符');
                }
                if (this.password1 !== this.password2) {
                    return $form.showFormError('两次输入的口令不一致');
                }
                var email =
this.email.trim().toLowerCase();
                $form.postJSON('/api/users', {
                    name: this.name.trim(),
                    email: email,
                    passwd: CryptoJS.SHA1(email + ':' +
this.password1).toString()
                }, function (err, r) {
                    if (err) {
                        return $form.showFormError(err);
                    }
                    return location.assign('/');
                });
            }
        }
    });
});


```



```
</form>
</div>

{% endblock %}
```

这样我们就把用户注册的功能完成了：



The screenshot shows a web browser window with the title '注册 - Awesome Python Webapp'. The address bar shows 'localhost:9000/register'. The page has a navigation bar with links: 'Awesome', '日志', '教程', '源码', '登陆', and '注册'. The main content area is titled '欢迎注册!' and contains a registration form with the following fields: '名字:' (Name), '电子邮件:' (Email), '输入口令:' (Password), and '重复口令:' (Repeat Password). Below the form is a blue button labeled '注册'. The footer contains social media icons for Weibo, GitHub, LinkedIn, and Twitter, followed by the text 'Powered by Awesome Python Webapp. Copyright © 2014. [Manage]' and 'www.liaoxuefeng.com. All rights reserved.' with a GitHub logo.

用户登录比用户注册复杂。由于HTTP协议是一种无状态协议，而服务器要跟踪用户状态，就只能通过cookie实现。大多数Web框架提供了Session功能来封装保存用户状态的cookie。

Session的优点是简单易用，可以直接从Session中取出用户登录信息。

Session的缺点是服务器需要在内存中维护一个映射表来存储用户登录信息，如果有两台以上服务器，就需要对Session做集群，因此，使用Session的Web App很难扩展。

我们采用直接读取cookie的方式来验证用户登录，每次用户访问任意URL，都会对cookie进行验证，这种方式的好处是保证服务器处理任意的URL都是无状态的，可以扩展到多台服务器。

由于登录成功后是由服务器生成一个cookie发送给浏览器，所以，要保证这个cookie不会被客户端伪造出来。

实现防伪造cookie的关键是通过一个单向算法（例如SHA1），举例如下：

当用户输入了正确的口令登录成功后，服务器可以从数据库取到用户的id，并按照如下方式计算出一个字符串：

```
"用户id" + "过期时间" + SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

当浏览器发送cookie到服务器端后，服务器可以拿到的信息包括：

- 用户id
- 过期时间
- SHA1值

如果未到过期时间，服务器就根据用户id查找用户口令，并计算：

```
SHA1("用户id" + "用户口令" + "过期时间" + "SecretKey")
```

并与浏览器cookie中的哈希进行比较，如果相等，则说明用户已登录，否则，cookie就是伪造的。

这个算法的关键在于SHA1是一种单向算法，即可以通过原始字符串计算出SHA1结果，但无法通过SHA1结果反推出原始字符串。

所以登录API可以实现如下：

```
@post('/api/authenticate')
def authenticate(*, email, passwd):
    if not email:
        raise APIValueError('email', 'Invalid email.')
    if not passwd:
        raise APIValueError('passwd', 'Invalid password.')
    users = yield from User.findAll('email=?', [email])
    if len(users) == 0:
        raise APIValueError('email', 'Email not exist.')
    user = users[0]
    # check passwd:
    sha1 = hashlib.sha1()
    sha1.update(user.id.encode('utf-8'))
    sha1.update(b':')
    sha1.update(passwd.encode('utf-8'))
    if user.passwd != sha1.hexdigest():
        raise APIValueError('passwd', 'Invalid password.')
    # authenticate ok, set cookie:
    r = web.Response()
    r.set_cookie(COOKIE_NAME, user2cookie(user, 86400),
max_age=86400, httponly=True)
    user.passwd = '*****'
```



```

        r.content_type = 'application/json'
        r.body = json.dumps(user,
ensure_ascii=False).encode('utf-8')
        return r

# 计算加密cookie:
def user2cookie(user, max_age):
    # build cookie string by: id-expires-sha1
    expires = str(int(time.time() + max_age))
    s = '%s-%s-%s-%s' % (user.id, user.passwd, expires,
_COOKIE_KEY)
    L = [user.id, expires, hashlib.sha1(s.encode('utf-8')).hexdigest()]
    return '-'.join(L)

```

对于每个URL处理函数，如果我们都去写解析cookie的代码，那会导致代码重复很多次。

利用middle在处理URL之前，把cookie解析出来，并将登录用户绑定到 `request` 对象上，这样，后续的URL处理函数就可以直接拿到登录用户：

```

@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' %
(request.method, request.path))
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = yield from cookie2user(cookie_str)
            if user:
                logging.info('set current user: %s' %
user.email)
                request.__user__ = user
                return (yield from handler(request))
        return auth

# 解密cookie:
@asyncio.coroutine
def cookie2user(cookie_str):
    '''
    Parse cookie and load user if cookie is valid.
    '''
    if not cookie_str:
        return None
    try:
        L = cookie_str.split('-')
        if len(L) != 3:
            return None
        uid, expires, sha1 = L

```

```

        if int(expires) < time.time():
            return None
        user = yield from User.find(uid)
        if user is None:
            return None
        s = '%s-%s-%s-%s' % (uid, user.passwd, expires,
                               _COOKIE_KEY)
        if sha1 != hashlib.sha1(s.encode('utf-8')).hexdigest():
            logging.info('invalid sha1')
            return None
        user.passwd = '*****'
        return user
    except Exception as e:
        logging.exception(e)
        return None

```

这样，我们就完成了用户注册和登录的功能。

Day 11 - 编写日志创建页

在Web开发中，后端代码写起来其实是相当容易的。

例如，我们编写一个REST API，用于创建一个Blog:

```

@post('/api/blogs')
def api_create_blog(request, *, name, summary, content):
    check_admin(request)
    if not name or not name.strip():
        raise APIValueError('name', 'name cannot be empty.')
    if not summary or not summary.strip():
        raise APIValueError('summary', 'summary cannot be empty.')
    if not content or not content.strip():
        raise APIValueError('content', 'content cannot be empty.')
    blog = Blog(user_id=request.__user__.id,
                 user_name=request.__user__.name,
                 user_image=request.__user__.image, name=name.strip(),
                 summary=summary.strip(), content=content.strip())
    yield from blog.save()
    return blog

```

编写后端Python代码不但很简单，而且非常容易测试，上面的API:

`api_create_blog()` 本身只是一个普通函数。

Web开发真正困难的地方在于编写前端页面。前端页面需要混合HTML、CSS和JavaScript，如果对这三者没有深入地掌握，编写的前端页面将很快难以维护。

更大的问题在于，前端页面通常是动态页面，也就是说，前端页面往往是由后端代码生成的。

生成前端页面最早的方式是拼接字符串：

```
s = '<html><head><title>'
  + title
  + '</title></head><body>'
  + body
  + '</body></html>'
```

显然这种方式完全不具备可维护性。所以有第二种模板方式：

```
<html>
<head>
  <title>{{ title }}</title>
</head>
<body>
  {{ body }}
</body>
</html>
```

ASP、JSP、PHP等都是用这种模板方式生成前端页面。

如果在页面上大量使用JavaScript（事实上大部分页面都会），模板方式仍然会导致JavaScript代码与后端代码绑得非常紧密，以至于难以维护。其根本原因在于负责显示的HTML DOM模型与负责数据和交互的JavaScript代码没有分割清楚。

要编写可维护的前端代码绝非易事。和后端结合的MVC模式已经无法满足复杂页面逻辑的需要了，所以，新的MVVM：Model View ViewModel模式应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示：

```
<script>
  var blog = {
    name: 'hello',
    summary: 'this is summary',
    content: 'this is content...'
  };
</script>
```

View是纯HTML：

```

<form action="/api/blogs" method="post">
  <input name="name">
  <input name="summary">
  <textarea name="content"></textarea>
  <button type="submit">OK</button>
</form>

```

由于Model表示数据，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

好消息是已有许多成熟的MVVM框架，例如AngularJS，KnockoutJS等。我们选择Vue这个简单易用的MVVM框架来实现创建Blog的页面

`templates/manage_blog_edit.html`:

```

{% extends '__base__.html' %}

{% block title %}编辑日志{% endblock %}

{% block beforehead %}

<script>
var
  ID = '{{ id }}',
  action = '{{ action }}';
function initVM(blog) {
  var vm = new Vue({
    el: '#vm',
    data: blog,
    methods: {
      submit: function (event) {
        event.preventDefault();
        var $form = $('#vm').find('form');
        $form.postJSON(action, this.$data,
function (err, r) {
          if (err) {
            $form.showFormError(err);
          }
          else {
            return
location.assign('/api/blogs/' + r.id);
          }
        });
      }
    }
  });
  $('#vm').show();
}

```

```

}
$(function () {
    if (ID) {
        getJSON('/api/blogs/' + ID, function (err, blog) {
            if (err) {
                return fatal(err);
            }
            $('#loading').hide();
            initVM(blog);
        });
    }
    else {
        $('#loading').hide();
        initVM({
            name: '',
            summary: '',
            content: ''
        });
    }
});
</script>

{% endblock %}

{% block content %}

    <div class="uk-width-1-1 uk-margin-bottom">
        <div class="uk-panel uk-panel-box">
            <ul class="uk-breadcrumb">
                <li><a href="/manage/comments">评论</a>
            </li>
                <li><a href="/manage/blogs">日志</a></li>
                <li><a href="/manage/users">用户</a></li>
            </ul>
        </div>
    </div>

    <div id="error" class="uk-width-1-1">
    </div>

    <div id="loading" class="uk-width-1-1 uk-text-center">
        <span><i class="uk-icon-spinner uk-icon-medium uk-
icon-spin"></i> 正在加载...</span>
    </div>

    <div id="vm" class="uk-width-2-3">
        <form v-on="submit: submit" class="uk-form uk-
form-stacked">
            <div class="uk-alert uk-alert-danger uk-
hidden"></div>
            <div class="uk-form-row">
                <label class="uk-form-label">标题:</label>

```

```

        <div class="uk-form-controls">
            <input v-model="name" name="name"
type="text" placeholder="标题" class="uk-width-1-1">
        </div>
    </div>
    <div class="uk-form-row">
        <label class="uk-form-label">摘要:</label>
        <div class="uk-form-controls">
            <textarea v-model="summary" rows="4"
name="summary" placeholder="摘要" class="uk-width-1-1"
style="resize:none;"></textarea>
        </div>
    </div>
    <div class="uk-form-row">
        <label class="uk-form-label">内容:</label>
        <div class="uk-form-controls">
            <textarea v-model="content" rows="16"
name="content" placeholder="内容" class="uk-width-1-1"
style="resize:none;"></textarea>
        </div>
    </div>
    <div class="uk-form-row">
        <button type="submit" class="uk-button uk-
button-primary"><i class="uk-icon-save"></i> 保存</button>
        <a href="/manage/blogs" class="uk-button">
<i class="uk-icon-times"></i> 取消</a>
    </div>
</form>
</div>

{% endblock %}

```

初始化Vue时，我们指定3个参数：

el: 根据选择器查找绑定的View，这里是`#vm`，就是id为`vm`的DOM，对应的是一个```标签；

data: JavaScript对象表示的Model，我们初始化为`{ name: '', summary: '', content: '' }`；

methods: View可以触发的JavaScript函数，`submit`就是提交表单时触发的函数。

接下来，我们在```标签中，用几个简单的`v-model`，就可以让Vue把Model和View关联起来：

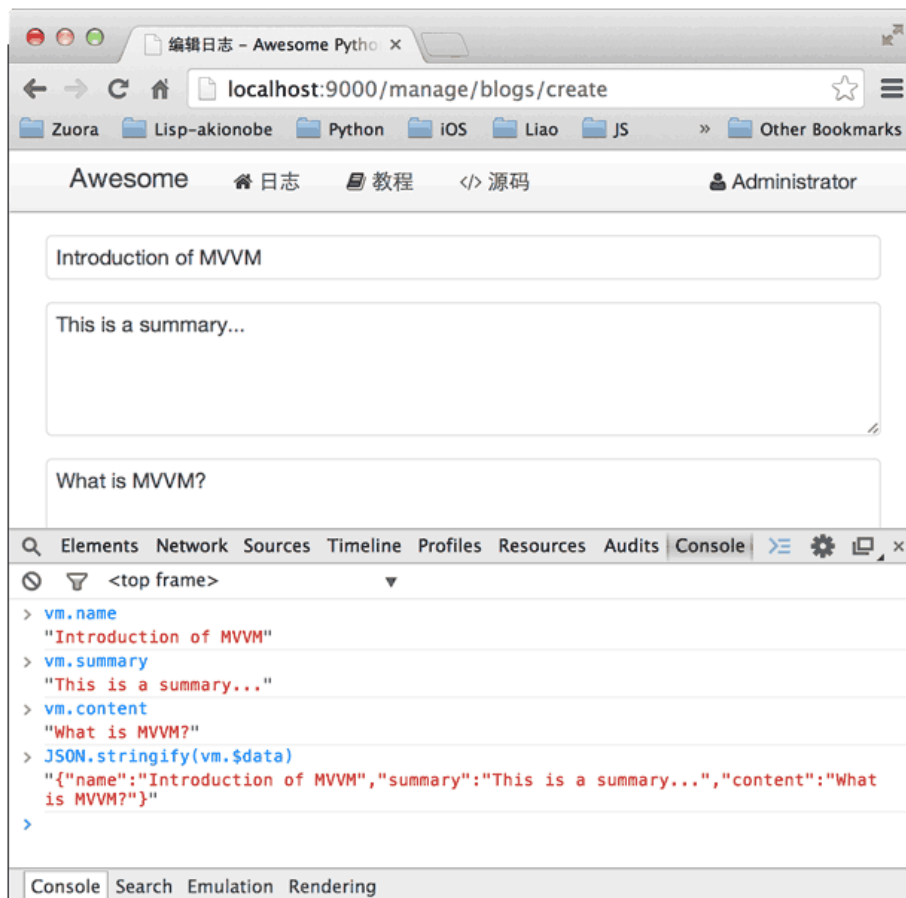
```

<!-- input的value和Model的name关联起来了 -->
<input v-model="name" class="uk-width-1-1">

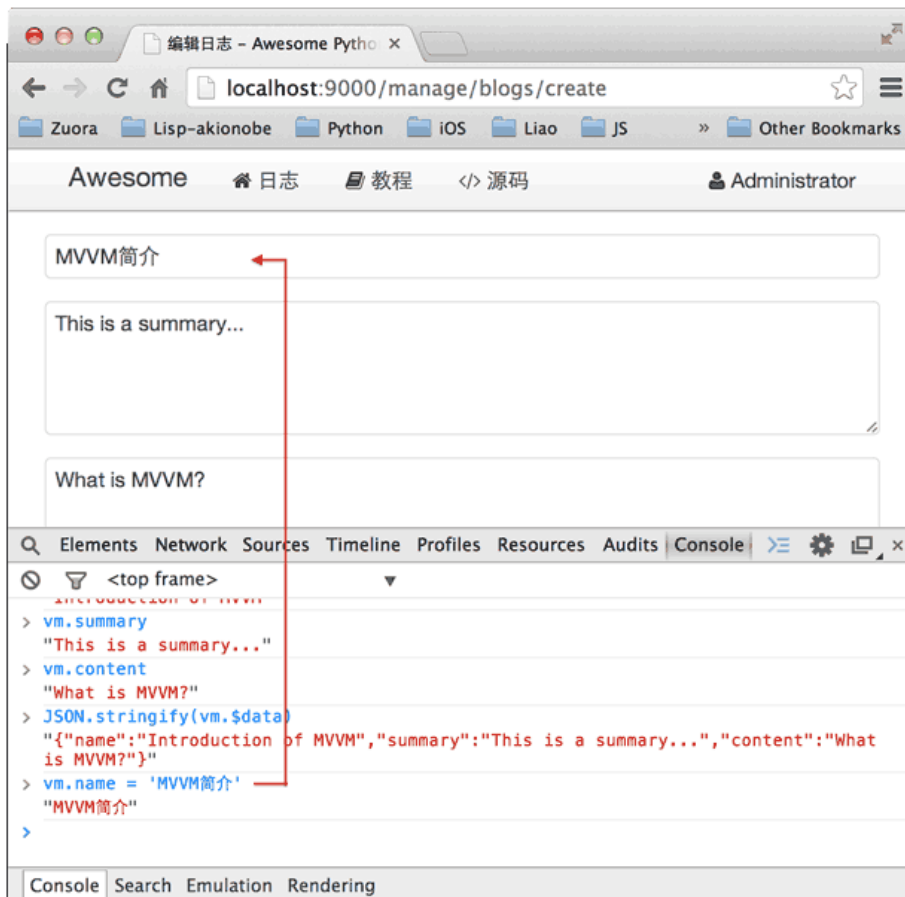
```

Form表单通过```把提交表单的事件关联到`submit`方法。

需要特别注意的是，在MVVM中，Model和View是双向绑定的。如果我们在Form中修改了文本框的值，可以在Model中立刻拿到新的值。试试在表单中输入文本，然后在Chrome浏览器中打开JavaScript控制台，可以通过`vm.name`访问单个属性，或者通过`vm.$data`访问整个Model：



如果我们在JavaScript逻辑中修改了Model，这个修改会立刻反映到View上。试试在JavaScript控制台输入`vm.name = 'MVVM简介'`，可以看到文本框的内容自动被同步了：



双向绑定是MVVM框架最大的作用。借助于MVVM，我们把复杂的显示逻辑交给框架完成。由于后端编写了独立的REST API，所以，前端用AJAX提交表单非常容易，前后端分离得非常彻底。

Day 12 - 编写日志列表页

MVVM模式不但可用于Form表单，在复杂的管理页面中也能大显身手。例如，分页显示Blog的功能，我们先把后端代码写出来：

在 `apis.py` 中定义一个 `Page` 类用于存储分页信息：

```
class Page(object):

    def __init__(self, item_count, page_index=1,
page_size=10):
        self.item_count = item_count
        self.page_size = page_size
        self.page_count = item_count // page_size + (1 if
item_count % page_size > 0 else 0)
        if (item_count == 0) or (page_index >
self.page_count):
            self.offset = 0
            self.limit = 0
            self.page_index = 1
        else:
            self.page_index = page_index
            self.offset = self.page_size * (page_index -
1)
```



```

        self.limit = self.page_size
        self.has_next = self.page_index < self.page_count
        self.has_previous = self.page_index > 1

    def __str__(self):
        return 'item_count: %s, page_count: %s,
page_index: %s, page_size: %s, offset: %s, limit: %s' %
(self.item_count, self.page_count, self.page_index,
self.page_size, self.offset, self.limit)

    __repr__ = __str__

```

在 `handlers.py` 中实现API:

```

@get('/api/blogs')
def api_blogs(*, page='1'):
    page_index = get_page_index(page)
    num = yield from Blog.findNumber('count(id)')
    p = Page(num, page_index)
    if num == 0:
        return dict(page=p, blogs=())
    blogs = yield from Blog.findAll(orderBy='created_at
desc', limit=(p.offset, p.limit))
    return dict(page=p, blogs=blogs)

```

管理页面:

```

@get('/manage/blogs')
def manage_blogs(*, page='1'):
    return {
        '__template__': 'manage_blogs.html',
        'page_index': get_page_index(page)
    }

```

模板页面首先通过API: `GET /api/blogs?page=?` 拿到Model:

```

{
  "page": {
    "has_next": true,
    "page_index": 1,
    "page_count": 2,
    "has_previous": false,
    "item_count": 12
  },
  "blogs": [...]
}

```

然后, 通过Vue初始化MVVM:

```

<script>
function initVM(data) {
    var vm = new Vue({
        el: '#vm',
        data: {
            blogs: data.blogs,
            page: data.page
        },
        methods: {
            edit_blog: function (blog) {
                location.assign('/manage/blogs/edit?id=' +
blog.id);
            },
            delete_blog: function (blog) {
                if (confirm('确认要删除"' + blog.name + '"?
删除后不可恢复! ')) {
                    postJSON('/api/blogs/' + blog.id +
'/delete', function (err, r) {
                        if (err) {
                            return alert(err.message ||
err.error || err);
                        }
                        refresh();
                    });
                }
            }
        }
    });
    $('#vm').show();
}

$(function() {
    getJSON('/api/blogs', {
        page: {{ page_index }}
    }, function (err, results) {
        if (err) {
            return fatal(err);
        }
        $('#loading').hide();
        initVM(results);
    });
});
</script>

```

View的容器是#vm，包含一个table，我们用v-repeat可以把Model的数组blogs直接变成多行的：

```

<div id="vm" class="uk-width-1-1">
    <a href="/manage/blogs/create" class="uk-button uk-
button-primary"><i class="uk-icon-plus"></i> 新日志</a>

    <table class="uk-table uk-table-hover">

```

```

<thead>
  <tr>
    <th class="uk-width-5-10">标题 / 摘要</th>
    <th class="uk-width-2-10">作者</th>
    <th class="uk-width-2-10">创建时间</th>
    <th class="uk-width-1-10">操作</th>
  </tr>
</thead>
<tbody>
  <tr v-repeat="blog: blogs" >
    <td>
      <a target="_blank" v-attr="href:
'/blog/'+blog.id" v-text="blog.name"></a>
    </td>
    <td>
      <a target="_blank" v-attr="href:
'/user/'+blog.user_id" v-text="blog.user_name"></a>
    </td>
    <td>
      <span v-
text="blog.created_at.toDateTime()"></span>
    </td>
    <td>
      <a href="#0" v-on="click:
edit_blog(blog)"><i class="uk-icon-edit"></i>
      <a href="#0" v-on="click:
delete_blog(blog)"><i class="uk-icon-trash-o"></i>
    </td>
  </tr>
</tbody>
</table>

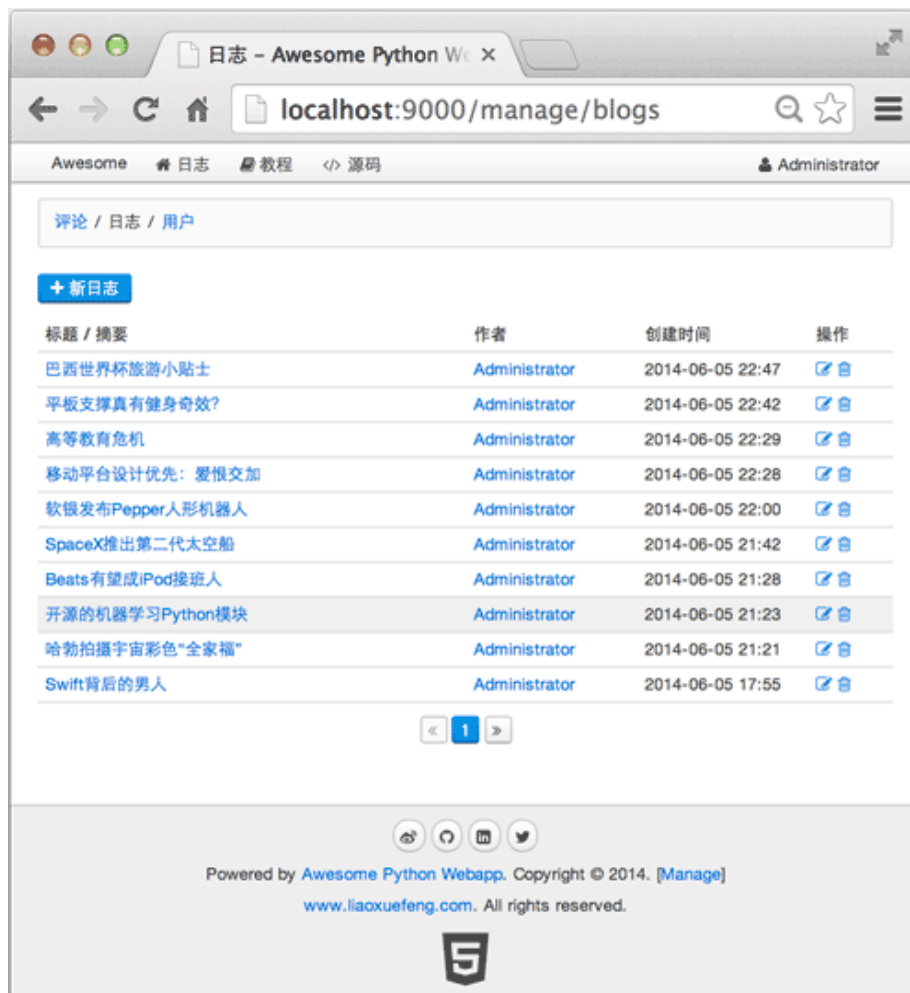
<div v-component="pagination" v-with="page"></div>
</div>

```

往Model的**blogs** 数组中增加一个Blog元素，table就神奇地增加了一行；把**blogs** 数组的某个元素删除，table就神奇地减少了一行。所有复杂的Model-View的映射逻辑全部由MVVM框架完成，我们只需要在HTML中写上**v-repeat**指令，就什么都不用管了。

可以把**v-repeat="blog: blogs"**看成循环代码，所以，可以在一个内部引用循环变量**blog**。**v-text**和**v-attr**指令分别用于生成文本和DOM节点属性。

完整的Blog列表页如下：



Day 13 - 提升开发效率

现在，我们已经把一个Web App的框架完全搭建好了，从后端的API到前端的MVVM，流程已经跑通了。

在继续工作前，注意到每次修改Python代码，都必须在命令行先Ctrl-C停止服务器，再重启，改动才能生效。

在开发阶段，每天都要修改、保存几十次代码，每次保存都手动来这么一下非常麻烦，严重地降低了我们的开发效率。有没有办法让服务器检测到代码修改后自动重新加载呢？

Django的开发环境在Debug模式下就可以做到自动重新加载，如果我们编写的服务器也能实现这个功能，就能大大提升开发效率。

可惜的是，Django没把这个功能独立出来，不用Django就享受不到，怎么办？

其实Python本身提供了重新载入模块的功能，但不是所有模块都能被重新载入。另一种思路是检测www目录下的代码改动，一旦有改动，就自动重启服务器。

按照这个思路，我们可以编写一个辅助程序pymonitor.py，让它启动wsgiapp.py，并时刻监控www目录下的代码改动，有改动时，先把当前wsgiapp.py进程杀掉，再重启，就完成了服务器进程的自动重启。

要监控目录文件的变化，我们也无需自己手动定时扫描，Python的第三方库 `watchdog` 可以利用操作系统的API来监控目录文件的变化，并发送通知。我们先用 `pip` 安装：

```
$ pip3 install watchdog
```

利用 `watchdog` 接收文件变化的通知，如果是 `.py` 文件，就自动重启 `wsgiapp.py` 进程。

利用Python自带的 `subprocess` 实现进程的启动和终止，并把输入输出重定向到当前进程的输入输出中：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

__author__ = 'Michael Liao'

import os, sys, time, subprocess

from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

def log(s):
    print('[Monitor] %s' % s)

class MyFileSystemEventHandler(FileSystemEventHandler):

    def __init__(self, fn):
        super(MyFileSystemEventHandler, self).__init__()
        self.restart = fn

    def on_any_event(self, event):
        if event.src_path.endswith('.py'):
            log('Python source file changed: %s' %
event.src_path)
            self.restart()

command = ['echo', 'ok']
process = None

def kill_process():
    global process
    if process:
        log('kill process [%s]...' % process.pid)
        process.kill()
        process.wait()
        log('Process ended with code %s.' %
process.returncode)
        process = None

def start_process():
```

```

global process, command
log('Start process %s...' % ' '.join(command))
process = subprocess.Popen(command, stdin=sys.stdin,
stdout=sys.stdout, stderr=sys.stderr)

def restart_process():
    kill_process()
    start_process()

def start_watch(path, callback):
    observer = Observer()

    observer.schedule(MyFileSystemEventHandler(restart_process
), path, recursive=True)
    observer.start()
    log('Watching directory %s...' % path)
    start_process()
    try:
        while True:
            time.sleep(0.5)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()

if __name__ == '__main__':
    argv = sys.argv[1:]
    if not argv:
        print('Usage: ./pymonitor your-script.py')
        exit(0)
    if argv[0] != 'python3':
        argv.insert(0, 'python3')
    command = argv
    path = os.path.abspath('.')
    start_watch(path, None)

```

一共70行左右的代码，就实现了Debug模式的自动重新加载。用下面的命令启动服务器：

```
$ python3 pymonitor.py wsgiapp.py
```

或者给 `pymonitor.py` 加上可执行权限，启动服务器：

```
$ ./pymonitor.py app.py
```

在编辑器中打开一个 `.py` 文件，修改后保存，看看命令行输出，是不是自动重启了服务器：

```

$ ./pymonitor.py app.py
[Monitor] watching directory
/Users/michael/Github/awesome-python3-webapp/www...
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-
python3-webapp/www) will start at 0.0.0.0:9000...
[Monitor] Python source file changed:
/Users/michael/Github/awesome-python-
webapp/www/handlers.py
[Monitor] Kill process [2747]...
[Monitor] Process ended with code -9.
[Monitor] Start process python app.py...
...
INFO:root:application (/Users/michael/Github/awesome-
python3-webapp/www) will start at 0.0.0.0:9000...

```

现在，只要一保存代码，就可以刷新浏览器看到效果，大大提升了开发效率。

Day 14 - 完成Web App

在Web App框架和基本流程跑通后，剩下的工作全部是体力活了：在Debug开发模式下完成后端所有API、前端所有页面。我们需要做的事情包括：

把当前用户绑定到 `request` 上，并对URL `/manage/` 进行拦截，检查当前用户是否是管理员身份：

```

@asyncio.coroutine
def auth_factory(app, handler):
    @asyncio.coroutine
    def auth(request):
        logging.info('check user: %s %s' %
(request.method, request.path))
        request.__user__ = None
        cookie_str = request.cookies.get(COOKIE_NAME)
        if cookie_str:
            user = yield from cookie2user(cookie_str)
            if user:
                logging.info('set current user: %s' %
user.email)
                request.__user__ = user
            if request.path.startswith('/manage/') and
(request.__user__ is None or not request.__user__.admin):
                return web.HTTPFound('/signin')
            return (yield from handler(request))
        return auth

```

后端API包括：

- 获取日志：GET /api/blogs

- 创建日志: POST /api/blogs
- 修改日志: POST /api/blogs/:blog_id
- 删除日志: POST /api/blogs/:blog_id/delete
- 获取评论: GET /api/comments
- 创建评论: POST /api/blogs/:blog_id/comments
- 删除评论: POST /api/comments/:comment_id/delete
- 创建新用户: POST /api/users
- 获取用户: GET /api/users

管理页面包括:

- 评论列表页: GET /manage/comments
- 日志列表页: GET /manage/blogs
- 创建日志页: GET /manage/blogs/create
- 修改日志页: GET /manage/blogs/
- 用户列表页: GET /manage/users

用户浏览页面包括:

- 注册页: GET /register
- 登录页: GET /signin
- 注销页: GET /signout
- 首页: GET /
- 日志详情页: GET /blog/:blog_id

把所有的功能实现, 我们第一个Web App就宣告完成!

Day 15 - 部署Web App

作为一个合格的开发者, 在本地环境下完成开发还远远不够, 我们需要把Web App部署到远程服务器上, 这样, 广大用户才能访问到网站。

很多做开发的同学把部署这件事情看成是运维同学的工作, 这种看法是完全错误的。首先, 最近流行DevOps理念, 就是说, 开发和运维要变成一个整体。其次, 运维的难度, 其实跟开发质量有很大的关系。代码写得垃圾, 运维再好也架不住天天挂掉。最后, DevOps理念需要把运维、监控等功能融入到开发中。你想服务器升级时不中断用户服务? 那就得在开发时考虑到这一点。

下面, 我们就来把awesome-python3-webapp部署到Linux服务器。

搭建Linux服务器

要部署到Linux, 首先得有一台Linux服务器。要在公网上体验的同学, 可以在Amazon的AWS申请一台EC2虚拟机(免费使用1年), 或者使用国内的一些云服务器, 一般都提供Ubuntu Server的镜像。想在本地部署的同学, 请安装虚拟机, 推荐使用VirtualBox。

我们选择的Linux服务器版本是Ubuntu Server 14.04 LTS, 原因是apt太简单了。如果你准备使用其他Linux版本, 也没有问题。

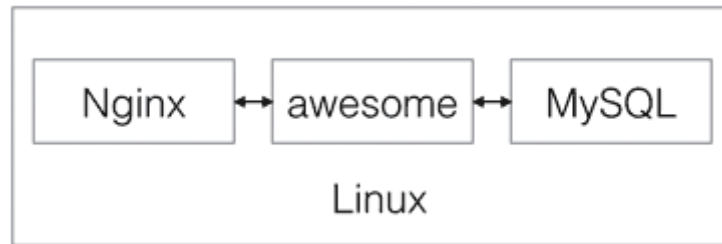
Linux安装完成后, 请确保ssh服务正在运行, 否则, 需要通过apt安装:


```
$ sudo apt-get install openssh-server
```

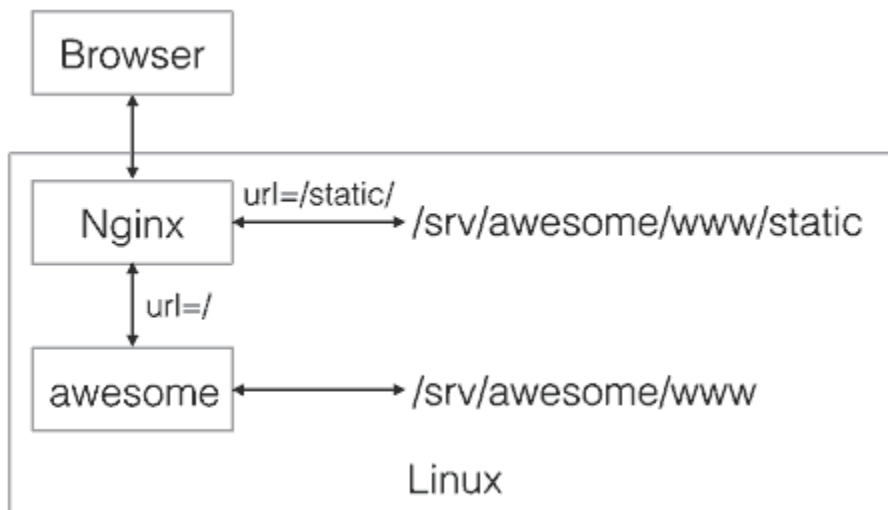
有了ssh服务，就可以从本地连接到服务器上。建议把公钥复制到服务器端用户的 `.ssh/authorized_keys` 中，这样，就可以通过证书实现无密码连接。

部署方式

利用Python自带的`asyncio`，我们已经编写了一个异步高性能服务器。但是，我们还需要一个高性能的Web服务器，这里选择Nginx，它可以处理静态资源，同时作为反向代理把动态请求交给Python代码处理。这个模型如下：



Nginx负责分发请求：



在服务器端，我们需要定义好部署的目录结构：

```
/
+- srv/
  +- awesome/      <-- web App根目录
    +- www/        <-- 存放Python源码
      +- static/    <-- 存放静态资源文件
    +- log/         <-- 存放log
```

在服务器上部署，要考虑到新版本如果运行不正常，需要回退到旧版本时怎么办。每次用新的代码覆盖掉旧的文件是不行的，需要一个类似版本控制的机制。由于Linux系统提供了软链接功能，所以，我们把`www`作为一个软链接，它指向哪个目录，哪个目录就是当前运行的版本：

```
michael@ubuntu:/srv/awesome$ ls -l
total 32
drwxr-xr-x 2 www-data www-data 4096 Jun  5 17:38 log
lrwxrwxrwx 1 root      root      21 Jun  5 17:50 www -> www-14-06-05_17.56.16
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:26 www-14-06-05_15.26.45
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:31 www-14-06-05_15.31.03
drwxr-xr-x 5 www-data www-data 4096 Jun  5 15:32 www-14-06-05_15.32.39
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:34 www-14-06-05_17.39.28
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:35 www-14-06-05_17.41.22
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:39 www-14-06-05_17.45.57
drwxr-xr-x 5 www-data www-data 4096 Jun  5 17:50 www-14-06-05_17.56.16
michael@ubuntu:/srv/awesome$ _
```

而Nginx和python代码的配置文件只需要指向www目录即可。

Nginx可以作为服务进程直接启动，但app.py还不行，所以，Supervisor登场！Supervisor是一个管理进程的工具，可以随系统启动而启动服务，它还时刻监控服务进程，如果服务进程意外退出，Supervisor可以自动重启服务。

总结一下我们需要用到的服务有：

- Nginx：高性能Web服务器+负责反向代理；
- Supervisor：监控服务进程的工具；
- MySQL：数据库服务。

在Linux服务器上，用apt可以直接安装上述服务：

```
$ sudo apt-get install nginx supervisor python3 mysql-server
```

然后，再把我们自己的Web App用到的Python库安装了：

```
$ sudo pip3 install jinja2 aiomysql aiohttp
```

在服务器上创建目录/srv/awesome/以及相应的子目录。

在服务器上初始化MySQL数据库，把数据库初始化脚本schema.sql复制到服务器上执行：

```
$ mysql -u root -p < schema.sql
```

服务器端准备就绪。

部署

用FTP还是SCP还是rsync复制文件？如果你需要手动复制，用一次两次还行，一天如果部署50次不但慢、效率低，而且容易出错。

正确的部署方式是使用工具配合脚本完成自动化部署。[Fabric](#)就是一个自动化部署工具。由于Fabric是用Python 2.x开发的，所以，部署脚本要用Python 2.7来编写，本机还必须安装Python 2.7版本。

要用Fabric部署，需要在本机（是开发机器，不是Linux服务器）安装Fabric：

```
$ easy_install fabric
```

Linux服务器上不需要安装Fabric，Fabric使用SSH直接登录服务器并执行部署命令。

下一步是编写部署脚本。Fabric的部署脚本叫 `fabfile.py`，我们把它放到 `awesome-python-webapp` 的目录下，与 `www` 目录平级：

```
awesome-python-webapp/  
+- fabfile.py  
+- www/  
+- ...
```

Fabric的脚本编写很简单，首先导入Fabric的API，设置部署时的变量：

```
# fabfile.py  
import os, re  
from datetime import datetime  
  
# 导入Fabric API:  
from fabric.api import *  
  
# 服务器登录用户名:  
env.user = 'michael'  
# sudo用户为root:  
env.sudo_user = 'root'  
# 服务器地址，可以有多个，依次部署:  
env.hosts = ['192.168.0.3']  
  
# 服务器MySQL用户名和口令:  
db_user = 'www-data'  
db_password = 'www-data'
```

然后，每个Python函数都是一个任务。我们先编写一个打包的任务：

```

_TAR_FILE = 'dist-awesome.tar.gz'

def build():
    includes = ['static', 'templates', 'transwarp',
                'favicon.ico', '*.py']
    excludes = ['test', '.*', '*.pyc', '*.pyo']
    local('rm -f dist/%s' % _TAR_FILE)
    with lcd(os.path.join(os.path.abspath('.'), 'www')):
        cmd = ['tar', '--dereference', '-czvf',
                '../dist/%s' % _TAR_FILE]
        cmd.extend(['--exclude=%s' % ex for ex in
                    excludes])
        cmd.extend(includes)
    local(' '.join(cmd))

```

Fabric提供`local('...')`来运行本地命令，`with lcd(path)`可以把当前命令的目录设定为`lcd()`指定的目录，注意Fabric只能运行命令行命令，Windows下可能需要Cgywin环境。

在`awesome-python-webapp`目录下运行：

```
$ fab build
```

看看是否在`dist`目录下创建了`dist-awesome.tar.gz`的文件。

打包后，我们就可以继续编写`deploy`任务，把打包文件上传至服务器，解压，重置`www`软链接，重启相关服务：

```

_REMOTE_TMP_TAR = '/tmp/%s' % _TAR_FILE
_REMOTE_BASE_DIR = '/srv/awesome'

def deploy():
    newdir = 'www-%s' % datetime.now().strftime('%y-%m-%d_%H.%M.%S')
    # 删除已有的tar文件：
    run('rm -f %s' % _REMOTE_TMP_TAR)
    # 上传新的tar文件：
    put('dist/%s' % _TAR_FILE, _REMOTE_TMP_TAR)
    # 创建新目录：
    with cd(_REMOTE_BASE_DIR):
        sudo('mkdir %s' % newdir)
    # 解压到新目录：
    with cd('%s/%s' % (_REMOTE_BASE_DIR, newdir)):
        sudo('tar -xzf %s' % _REMOTE_TMP_TAR)
    # 重置软链接：
    with cd(_REMOTE_BASE_DIR):
        sudo('rm -f www')
        sudo('ln -s %s www' % newdir)
        sudo('chown www-data:www-data www')
        sudo('chown -R www-data:www-data %s' % newdir)

```

```
# 重启Python服务和nginx服务器：
with settings(warn_only=True):
    sudo('supervisorctl stop awesome')
    sudo('supervisorctl start awesome')
    sudo('/etc/init.d/nginx reload')
```

注意 `run()` 函数执行的命令是在服务器上运行，`with cd(path)` 和 `with lcd(path)` 类似，把当前目录在服务器端设置为 `cd()` 指定的目录。如果一个命令需要 `sudo` 权限，就不能用 `run()`，而是用 `sudo()` 来执行。

配置Supervisor

上面让Supervisor重启awesome的命令会失败，因为我们还没有配置Supervisor呢。

编写一个Supervisor的配置文件 `awesome.conf`，存放
到 `/etc/supervisor/conf.d/` 目录下：

```
[program:awesome]

command      = /srv/awesome/www/app.py
directory    = /srv/awesome/www
user         = www-data
startsecs    = 3

redirect_stderr      = true
stdout_logfile_maxbytes = 50MB
stdout_logfile_backups  = 10
stdout_logfile        = /srv/awesome/log/app.log
```

配置文件通过 `[program:awesome]` 指定服务名为 `awesome`，`command` 指定启动 `app.py`。

然后重启Supervisor后，就可以随时启动和停止Supervisor管理的 services 了：

```
$ sudo supervisorctl reload
$ sudo supervisorctl start awesome
$ sudo supervisorctl status
awesome                                RUNNING    pid 1401, uptime 5:01:34
```

配置Nginx

Supervisor只负责运行 `app.py`，我们还需要配置Nginx。把配置文件 `awesome` 放到 `/etc/nginx/sites-available/` 目录下：

```
server {
    listen      80; # 监听80端口

    root        /srv/awesome/www;
```

```

access_log /srv/awesome/log/access_log;
error_log /srv/awesome/log/error_log;

# server_name awesome.liaoxuefeng.com; # 配置域名

# 处理静态文件/favicon.ico:
location /favicon.ico {
    root /srv/awesome/www;
}

# 处理静态资源:
location ~ ^/static/.*$ {
    root /srv/awesome/www;
}

# 动态请求转发到9000端口:
location / {
    proxy_pass http://127.0.0.1:9000;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For
$proxy_add_x_forwarded_for;
}
}

```

然后在 `/etc/nginx/sites-enabled/` 目录下创建软链接:

```

$ pwd
/etc/nginx/sites-enabled
$ sudo ln -s /etc/nginx/sites-available/awesome .

```

让Nginx重新加载配置文件，不出意外，我们的 `awesome-python3-webapp` 应该正常运行:

```

$ sudo /etc/init.d/nginx reload

```

如果有任何错误，都可以在 `/srv/awesome/log` 下查找Nginx和App本身的log。如果Supervisor启动时报错，可以在 `/var/log/supervisor` 下查看Supervisor的log。

如果一切顺利，你可以在浏览器中访问Linux服务器上的 `awesome-python3-webapp` 了:



如果在开发环境更新了代码，只需要在命令行执行：

```
$ fab build
$ fab deploy
```

自动部署完成！刷新浏览器就可以看到服务器代码更新后的效果。

友情链接

嫌国外网速慢的童鞋请移步网易和搜狐的镜像站点：

<http://mirrors.163.com/>

<http://mirrors.sohu.com/>

Day 16 - 编写移动App

网站部署上线后，还缺点啥呢？

在移动互联网浪潮席卷而来的今天，一个网站没有上线移动App，出门根本不好意思跟人打招呼。

所以，`awesome-python3-webapp`必须得有一个移动App版本！

开发iPhone版本

我们首先来看看如何开发iPhone App。前置条件：一台Mac电脑，安装XCode和最新的iOS SDK。

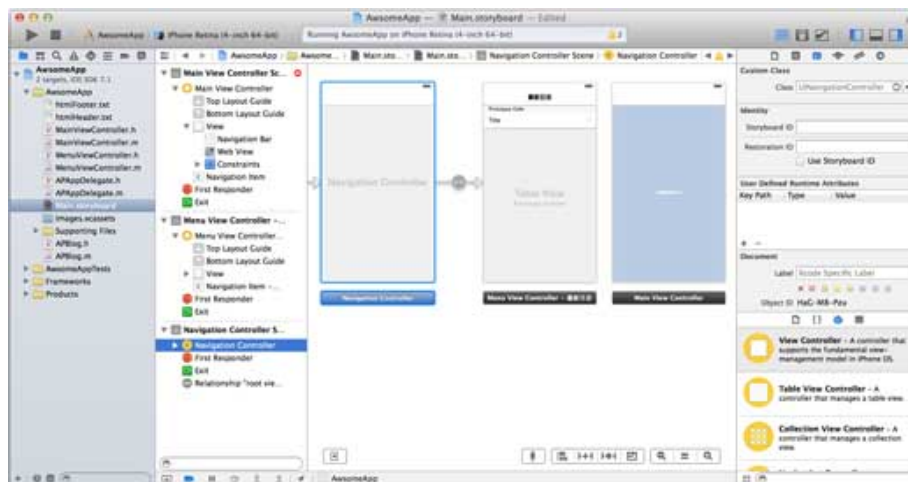
在使用MVVM编写前端页面时，我们就能感受到，用REST API封装网站后台的功能，不但能清晰地分离前端页面和后台逻辑，现在这个好处更加明显，移动App也可以通过REST API从后端拿到数据。

我们来设计一个简化版的iPhone App，包含两个屏幕：列出最新日志和阅读日志的详细内容：



只需要调用API：`/api/blogs`。

在XCode中完成App编写：



由于我们的教程是Python，关于如何开发iOS，请移步[Develop Apps for iOS](#)。

[点击下载iOS App源码](#)。

如何编写Android App？这个当成作业了。