Web [Application] Frameworks

- **conventional approach to building a web service**
  - write ad hoc client code in HTML, CSS, Javascript, ... by hand
  - write ad hoc server code in [whatever] by hand
  - write ad hoc access to [whatever] database system
- **so well understood that it's almost mechanical**
- **web frameworks mechanize (parts of) this process**
- **lots of tradeoffs and choices**
  - what client and server language(s)
  - how web pages are generated
  - how web events are linked to server actions
  - how database access is organized (if at all)
- **can be a big win, but not always**
  - some are heavyweight
  - easy to lose track of what's going on in multiple layers of generated software
  - work well if your application fits their model, less well if it doesn't
- **examples:**
  - Ruby on Rails
  - Django, Flask
  - Google Web Toolkit
  - Express / Node.js, Zend (PHP), ASP.NET (C#, VB.NET), and many others

Minimal Python server

```
import SocketServer
import SimpleHTTPServer

class Reply(SimpleHTTPServer.SimpleHTTPRequestHandler):
  def do_GET(self):
    # query arrives in self.path; return anything, e.g.,
    self.wfile.write("query was %s\n" % self.path)

def main():
  # do initialization or whatever
  SocketServer.ForkingTCPServer('', 8080),
                              Reply).serve_forever()

main()
```

Overview of frameworks

- **client-server relationship is stereotypical**
  - client sends requests using information from forms
  - server parses request, dispatches proper function, which retrieves from database, formats response, returns it

- **URL names encode requests**

  …/login/name

  …/add/data_to_be_added

  …/delete/id_to_delete

- **server uses URL pattern to call proper function with right args**

- **server usually provides structured & safer access to database**

- **server may provide templating language for generating HTML**
  - e.g., replace `{% foo %}` with value of variable foo, etc.

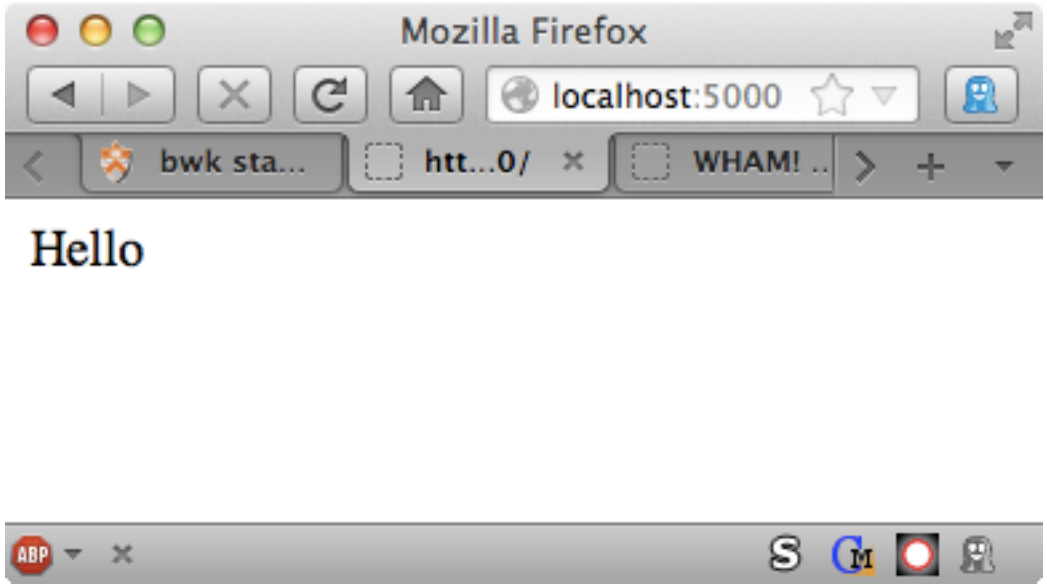
- **framework may automatically generate an admin interface**

Flask: Python-based microframework

- simplest example?

```
import flask
app = flask.Flask(__name__)
@app.route('/')
def hello():
    return 'Hello'
app.run()
```

```
$ python hello0.py
```



Sending form data

```
<form name=top id=top METHOD=POST
      ACTION="http://localhost:5000">
<p> Name: <input type="text" name=Name id=Name >
<p> Netid: <input type="text" name=Netid id=Netid >
<p> Class:
<input type="radio" name=Class value="2015"> '15
<input type="radio" name=Class value="2016"> '16
<input type="radio" name=Class value="2017"> '17
<input type="radio" name=Class value="2018"> '18

<p> Courses:
<input type="checkbox" name=C126> 126
<input type="checkbox" name=C217> 217
<input type="checkbox" name=C226> 226
</ul>

<p> <input type="submit" value="Submit"> <input type=reset>
```

Processing form data

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/', methods=['POST','GET'])
def hello():
    s = ""
    for (k,v) in request.form.iteritems():
        s = "%s %s=%s<br>" % (s, k, v)
    return 'Hello<br>' + s

app.run()
```

Flaskr example: tiny blog site in Flask

- **part of the Flask documentation**
  - thanks to Armin Ronacher

- **URL routing for login, logout, add record, clear all**
- **CSS for styling**
- **templates for merging variable content into layout**
- **uses SQLite3 to store data**
  - my version uses MongoDB

Python @ decorators

- a way to insert or modify code in functions and classes

```
@decorate
function foo(): …
```

- compilation compiles foo, passes the object to `decorate`, which does something and replaces foo by the result
- used in Flask to manage URL routing

```
@app.route('/add', methods=['POST'])
def add_entry():
    blog.insert({"title": request.form['title'],
                 "text": request.form['text']})
    return redirect(url_for('show_entries'))

@app.route('/login', methods=['GET', 'POST']) ...
@app.route('/clear', methods=['GET', 'POST']) ...
@app.route('/logout') ...
```

Django: more heavyweight Python-based framework

- **by Adrian Holovaty and Jacob Kaplan-Moss (released July 2005)**

Django Reinhart, 1910-1953

- **a collection of Python scripts to**
- **create a new project / site**
  - generates Python scripts for settings, etc.
  - configuration info stored as Python lists
- **create a new application within a project**
  - generates scaffolding/framework for models, views
- **run a development web server for local testing**

- **generate a database or build interface to an existing database**
- **provide a command-line interface to application**
- **create an administrative interface for the database**
- **run automated tests**
- **...**

Conventional approach to building a web site

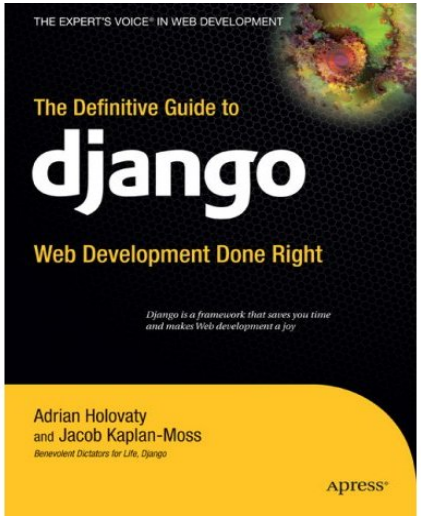
- **user interface, logic, database access are all mixed together**

```
import MySQLdb
print "Content-Type: text/html"
print
print "<html><head><title>Books</title></head>"
print "<body>"
print "<h1>Books</h1>"
print "<ul>"
connection = MySQLdb.connect(user='me', passwd='x', db='my_db')
cursor = connection.cursor()
cursor.execute("SELECT name FROM books ORDER BY pub_date DESC")
for row in cursor.fetchall():
    print "<li>%s</li>" % row[0]
print "</ul>"
print "</body></html>"
connection.close()
```

Model-View-Controller (MVC) pattern

- **an example of a design pattern**
- **model: the structure of the data**
  - how data is defined and accessed
- **view: the user interface**
  - what it looks like on the screen
  - can have multiple views for one model
- **controller: how information is moved around**
  - processing events, gathering and processing data, generating HTML, ...
- **separate model from view from processing so that when one changes, the others need not**
- **used with varying fidelity in**
  - Django, App Engine, Ruby on Rails, XCode Interface Builder, ...
- **not always clear where to draw the lines**
  - but trying to separate concerns is good

Django web framework



djangobook.com

- **write client code in HTML, CSS, Javascript, ...**
  - Django template language helps
    separate form from content
- **write server code in Python**
  - some of this is generated for you
- **write database access with Python library calls**
  - they are translated to SQL database commands

- **URLs on web page map mechanically to Python function calls**
  - regular expressions specify classes of URLs
  - URL received by server is matched against regular expressions
  - if a match is found, that identifies function to be called
    and arguments to be provided to the function

Django automatically-generated files

- **generate framework/skeleton of code by program**
- **three basic files:**

  models.py: database tables, etc.

  views.py: business logic, formatting of output

  urls.py: linkage between web requests and view functions
- **plus others for special purposes:**

  settings.py: db type, names of modules, ...

  tests.py: test files

  admin.py: admin info

  templates: for generating and filling HTML info

Example database linkage

in settings.py

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': '/Users/bwk/django/sql3.db', ...
```

in models.py

```
from django.db import models
class Post(models.Model):
    title = models.TextField(5)
    text = models.TextField()
```

generated by Django

```
BEGIN;
CREATE TABLE "blog_post" (
    "id" integer NOT NULL PRIMARY KEY,
    "title" text NOT NULL,
    "text" text NOT NULL
)
;
```

URL patterns

- **regular expressions used to recognize parameters and pass them to Python functions**
- **provides linkage between web page and what functions are called for semantic actions**

```
urlpatterns = patterns('',
  (r'^time/$', current_datetime),
  (r'^time/plus/(\d{1,2})/$', hours_ahead),
)
```

- **a reference to web page …/`time/` calls the function**

```
current_datetime()
```

- **tagged regular expressions for parameters: url …/`time/plus/12` calls the function**

```
hours_ahead(12)
```

Templates for generating HTML

- **try to separate page design from code that generates it**
- **Django has a specialized language for including HTML within code**
  - loosely analogous to PHP mechanism

```
# latest_posts.html (the template)

<html><head><title>Latest Posts</title></head>
<body>
<h1>Posts</h1>
<ul>
{% for post in post_list %}
    <li>{{ post.title }} {{ post.text }}</li>
{% endfor %}
</ul>
</body></html>
```

Administrative interface

- **most systems need a way to modify the database even if initially created from bulk data**
  - add / remove users, set passwords, ...
  - add / remove records
  - fix contents of records
  - ...
- **often requires special code**

- **Django generates an administrative interface automatically**
  - loosely equivalent to MyPhpAdmin

Google App Engine (since 4/08)

- **web application development framework**
  - analogous to Django
  - template mechanism looks the same
  - YAML for configuration
- **supports Python, Java, Go, PHP on server side**
  - and other languages that use the Java Virtual Machine?
  - template language Jinja2
- **Google provides the server**
  - run locally for testing & debugging, upload to appspot.com to deploy
- **restrictions on what server-side code can do**
  - non-relational database based on BigTable
  - or a pseudo-relational database called GQL
  - only static files can be stored on the server, read only access
  - no sockets, threads, C-based modules, system calls, …

Node.js server

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
```

- Express framework

Google Web Toolkit (GWT) (May 2006)

- **write client (browser) code in Java**
  - widgets, events, layout loosely similar to Swing
- **test client code on server side**
  - test browser, or plugin for testing with real browser on local system
- **compile Java to Javascript and HTML/CSS**
  - [once it works]
- **use generated code as part of a web page**
  - generated code is browser independent (diff versions for diff browsers)
- **can use development environments like Eclipse**
  - can use JUnit for testing
- **strong type checking on source**
  - detect typos, etc., at compile time (unlike Javascript)
- **may not handle all Java runtime libraries**
- **no explicit support for database access on server**
  - use whatever package is available

Assessment of Web Frameworks

- **advantages**
  - takes care of repetitive parts
    more efficient in programmer time
  - automatically generated code is likely to be more reliable, have more uniformity of structure
  - "DRY" (don't repeat yourself) is encouraged
  - "single point of truth"
    information is in only one place so it's easier to change things
  - ...
- **potential negatives**
  - automatically generated code
    can be hard to figure out what's going on
    can be hard to change if you don't want to do it their way
  - systems are large and can be slow
  - ...
- **read Joel Spolsky's "Why I hate frameworks"**

  `http://discuss.joelonsoftware.com/default.asp?joel.3.219431.12`

Package managers

- pip Python (pypi.python.org/pypi/pip)

```
pip install Django
```

- apt-get Ubuntu Linux

```
apt-get install whatever
```

- npm Node.js

```
npm install node
```

- port Macports

```
port install ruby
```

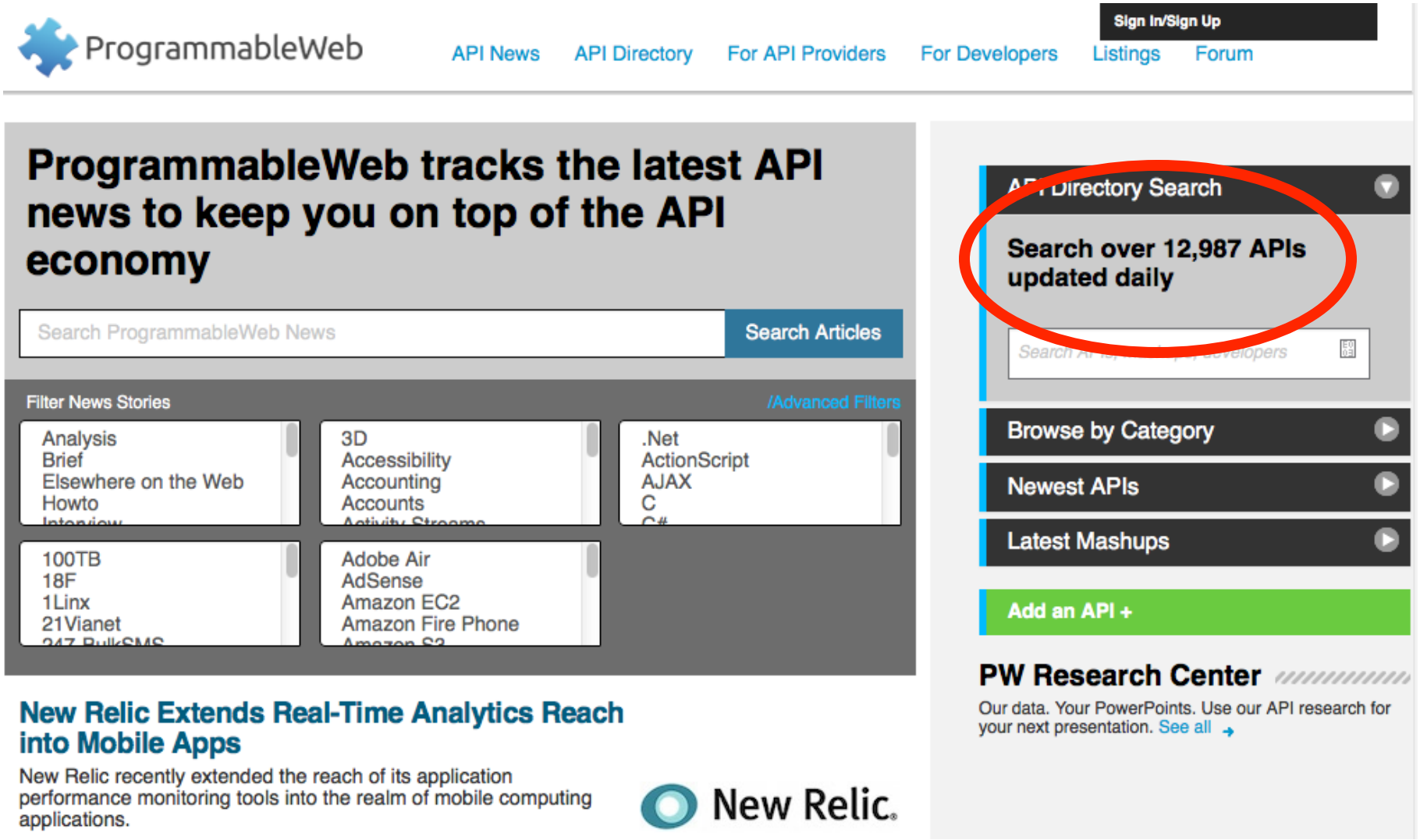
- brew Homebrew

```
brew install ruby
```

- gem Ruby

- ...

Mashups: duct tape programming



(programmableweb.com)

Assessment of Ajax-based systems

- **potential advantages**
  - can be much more responsive (cf Google maps)
  - can off-load work from server to client
  - code on server is not exposed
  - continuous update of services
- **potential negatives**
  - browsers are not standardized
  - Javascript code is exposed to client
  - Javascript code can be bulky and slow
  - asynchronous code can be tricky
  - DOM is very awkward
  - browser history not maintained without effort
- **what next? (changing fast)**
  - more and better libraries
  - better tools and languages for programming
  - better standardization?
  - will the browser ever replace the OS?