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
  - somewhat 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
  - Google Web Toolkit
  - Zend (PHP), ASP.NET (C#, VB.NET), and lots of others [Wikipedia lists over 100]

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 normal HTML
  - 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)
- doesn't handle all Java runtime libraries
  - currently at Java version 1.5
- no explicit support for database access on server
  - use whatever package is available
Java startup...

```java
public class StockWatcher implements EntryPoint {
    private VerticalPanel mainPanel = new VerticalPanel();
    private FlexTable stocksFlexTable = new FlexTable();
    private HorizontalPanel addPanel = new HorizontalPanel();
    private TextBox newSymbolTextBox = new TextBox();
    private Button addStockButton = new Button("Add");
    private Label lastUpdatedLabel = new Label();
    private ArrayList<String> stocks = new ArrayList<String>();

    public void onModuleLoad() {
        // Create table for stock data.
        stocksFlexTable.setText(0, 0, "Symbol");
        stocksFlexTable.setText(0, 1, "Price");
        stocksFlexTable.setText(0, 2, "Change");
        stocksFlexTable.setText(0, 3, "Remove");

        // Assemble Main panel.
        mainPanel.add(stocksFlexTable);
        mainPanel.add(addPanel);
        mainPanel.add(lastUpdatedLabel);

        // Associate the Main panel with the HTML host page.
        RootPanel.get("stockList").add(mainPanel);
    }
}
```
Linkage between Java/Javascript and HTML

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html;">
    <link type="text/css" rel="stylesheet" href="StockWatcher.css">
    <title>Brian's Portfolio</title>
    <script type="text/javascript" language="javascript" src="stockwatcher/stockwatcher.nocache.js"></script>
  </head>
  <body>
    <h1>Brian's Portfolio</h1>
    <div id="stockList"></div>
  </body>
</html>
```

"Same Origin Policy"

- "The same origin policy prevents a document or script loaded from one origin from getting or setting properties of a document from another origin. This policy dates all the way back to Netscape Navigator 2.0." (Mozilla)

- "The SOP states that JavaScript code running on a web page may not interact with any resource not originating from the same web site." (Google)

- basically Javascript can only reference information from the site that provided the original code

- BUT: if a page loads Javascript from more than one site (e.g., as with cookies from third-party sites), then that JS code can interact with that third-party site
GWT assessment

- problem: Javascript is irregular, unsafe, not portable, easily abused

- solution: use Java, which is type-safe, standard, portable

- translate Java to Javascript to either be browser independent or tailored to specific browser as appropriate

- can take advantage of browser quirks, make compact code, discourage reverse engineering

- can provide standardized mechanisms for widgets, events, DOM access, server access, AJAX, RE's and other libraries, ...

- in effect, treat each browser as a somewhat irregular machine and compile optimized code for it specifically

Django

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

- 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

  - ...
Django web framework

- **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

Conventional approach to building a web site

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

```python
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 approach

- generate framework/skeleton of code by program

  # models.py (the database tables)
  from django.db import models
  class Book(models.Model):
      name = models.CharField(maxlength=50)
      pub_date = models.DateField()

  # views.py (the business logic)
  from django.shortcuts import render_to_response
  from models import Book
  def latest_books(request):
      book_list = Book.objects.order_by('-pub_date')[:10]
      return render_to_response('latest_books.html',
      {'book_list': book_list})

  # urls.py (the URL configuration)
  from django.conf.urls.defaults import *
  import views

  urlpatterns = patterns('',
      (r'latest/$', views.latest_books),
  )
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

```python
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

```html
# latest_books.html (the template)

<html><head><title>Books</title></head>
<body>
<h1>Books</h1>
<ul>
{% for book in book_list %}
    <li>{{ book.name }}</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

  urlpatterns = patterns('',
      ...
      # Uncomment this for admin:
      # (r'^admin/', include('django.contrib.admin.urls')),

GWT vs Django

- focusing on different parts of the overall problem

- **GWT provides**
  - reliable, efficient, browser-independent Javascript (from Java)
  - extensive widget set
  - no help with database access, generating HTML, ...

- **Django provides**
  - no Javascript help
  - no widgets
  - easy database access; template language for generating HTML, ...
  - easy linkage from URLs on web page to Python functions

- is **GWT + App Engine** a good combination?
Google App Engine  (since 4/08)

- web application development framework
  - analogous to Django
  - template mechanism looks the same
  - YAML for configuration
- supports Python and Java on server side
  - and other languages that use the Java Virtual Machine
- Google provides the server
- restrictions on what server-side code can do
  - non-relational database based on BigTable
  - only static files can be stored on the server, read only access
  - no sockets, threads, C-based modules, system calls, ...

```
application: helloworld
version: 1
runtime: python
api_version: 1

handlers:
- url: /.*
  script: helloworld.py
```

print 'Content-Type: text/plain'
print ''
print 'Hello, world!

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 usually large and could be slow
  - ...

- read Joel Spolsky's "Why I hate frameworks"
  http://discuss.joelonsoftware.com/default.asp?joel.3.219431.12
Mashups: duct tape programming

- the web version of components?
- the browser as operating system?

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?