INF 212
ANALYSIS OF PROG. LANGS.

INTERACTIVITY

Kaj Dreef

Interactivity

- Program continually receives input and updates its state
- Opposite of batch processing

Batch processing

```
dataIn = getInput()
dataOut = process(dataIn)
display(dataOut)
```

Event loop

```
state
while (True)
    event = eventSource.getNextEvent()
    process(event)
    render(state)
```

Event loop handled by framework

User code

state

```
callback(event)
    process(event)
    render(state)
```

Hollywood style

framework

```
while (True)
    event = eventSource.getNextEvent()
    callback(event)
```

Issues

- How to manage internal state and external views
- How to deal with application "memory"
  - Behavior that depends on history

- These are unique to interactive applications

Model-View-Controller

MVC

MVC Trinity

- Model
  - Represents the application's data and logic
- View
  - Represents a specific rendition of the model
- Controller
  - Provides input controls for populating/updating the model and for invoking the right view

- Objects/functions belong to only one of these

Term Frequency v1 – Model

```
class WordFrequenciesModel:
  """ Models the data. In this case, we're only interested
  in words and their frequencies as an end result """
  freqs = {}
  def __init__(self, path_to_file):
      self.update(path_to_file)

  def update(self, path_to_file):
    try:
      stopwords = set(open('../stop_words.txt').read().split(','))
      words = re.findall('[a-z]{2,}', open(path_to_file).read().lower())
      self.freqs = collections.Counter(w for w in words if w not in stopwords)
    except IOError:
      print "File not found"
      self.freqs = {}
```

Term Frequency v1 – View

```
class WordFrequenciesView:
    def __init__(self, model):
        self._model = model

    def render(self):
        sorted_freqs = sorted(self._model.freqs.iteritems(), \
                              key=operator.itemgetter(1),
reverse=True)
        for (w, c) in sorted_freqs[:25]:
            print w, '-', c
```

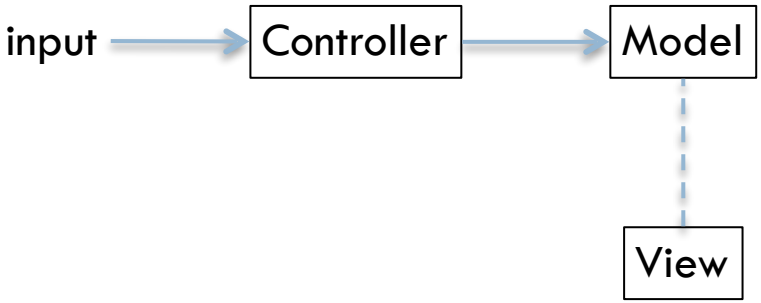
Term Frequency v1 – Controller

```
class WordFrequencyController:
    def __init__(self, model, view):
        self._model, self._view = model, view
        view.render()

    def run(self):
        while True:
            print "Next file: "
            sys.stdout.flush()
            filename = sys.stdin.readline().strip()
            self._model.update(filename)
            self._view.render()
```

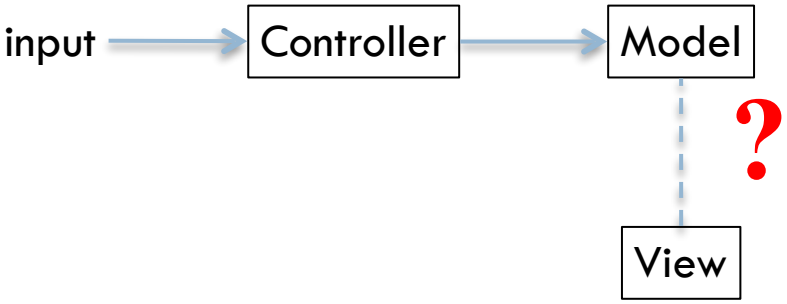
Passive vs. Active

- Passive MVC
  - Controller is driver of both model & view updates
  - (Previous example)

- Active MVC
  - View(s) updated automatically when model changes

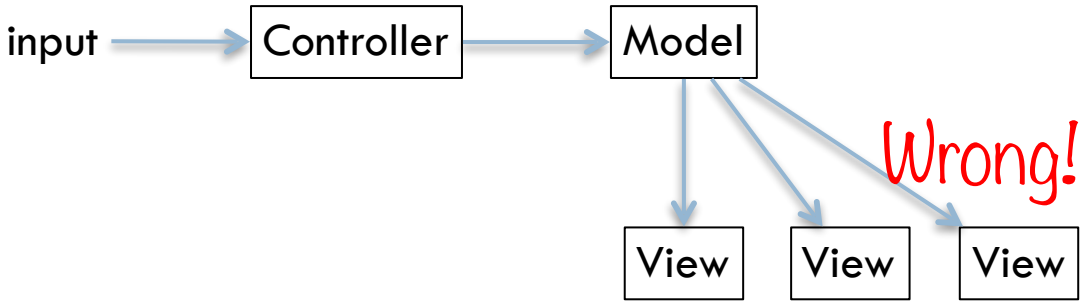


Active MVC



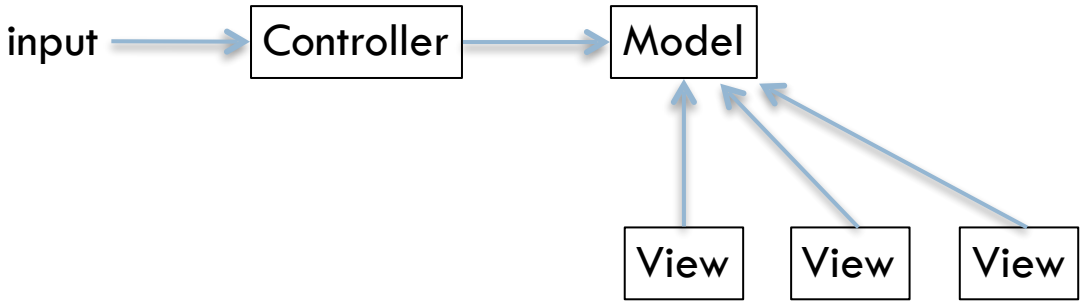
Active MVC – the wrong way

- Model holds references to views
  - Calls them when it changes



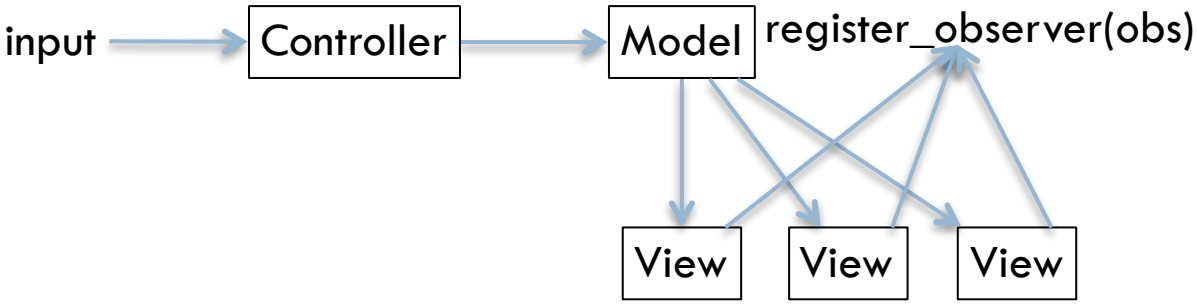
Active MVC – better

- Views hold references to model
  - Observe periodically
  - Free agents style



Active MVC – better

- Model is a "subject" that accepts "observers"
  - Calls them when it changes
  - Hollywood style ("I'll call you back")



MVC

- MVC can happen at several scales
- Separation sometimes is difficult

REST

Interesting ideas for how to deal with application "memory"

Recap

- HTTP
  - URLs
  - Methods
  - Headers
  - Status Codes
  - Caches
  - Cookies
- HTML and HTTP
  - hrefs/imgs
  - Forms
  - Scripts (XMLHttpRequest)

HTML and HTTP

- Links and images
  - **<link** href="mystyle.css" rel="stylesheet" type="text/css" **/>**
  - **<img** src="mypic.gif" alt="My Picture" **/>**
  - Semantics: Embedded Retrieval → GET
- Anchors
  - ****Anchor text****
  - Semantics: Potential Retrieval → GET
- Forms
  - **<form action=*URI* method=*OP*>**
    input fields
    </form>
  - Semantics: OP = Potential Retrieval → GET | Potential Creation → POST
- Scripts
  - **<script type="text/javascript">**
    script statements
    </script>
  - JavaScript has the capability of invoking HTTP operations on servers programmatically

First Web Programs

- GET http://example.com/file.html

- GET http://example.com/program.py?arg=3 (or POST)

- Web server needs to recognize files extensions and react appropriately
- Common Gateway Interface (CGI) model

First Web Programs – CGI

- A standard (see RFC3875: *CGI Version 1.1*) that defines how web server software can delegate the generation of webpages to a console application.
- Console app can be written in any PL
  - CGI programs generate HTML responses
  - First CGI programs used Perl

- 1993

First Web Programs – PHP

- Natural extension of CGI/Perl, 1994
- Embedded scripting language that helped Perl

```
#!/usr/local/bin/perl

print "Content-type: text/html\n\n";
print "<html>\n<head>";
print "<title>Test</title>\n";
print "</head>\n<body>\n";
print "Hello, world!\n";
print "</body>\n</html>";
```

helloworld.pl

```
<html>
 <head>
   <title>Test</title>
 </head>
 <body>
   <?php echo "Hello World";?>
 </body>
</html>
```

helloworld.php

Web Programming

- It all went down hill from here
  - 1995-2000: a lot of bad programming styles
- Generalized confusion about how to use HTTP
  - HTTP reduced to GET and POST
  - HTTP reduced to POST (!) in some models

REST

- REpresentational State Transfer
- Explanation of HTTP 1.1 (for the most part)
- Style of writing distributed applications
- “Story” that guides the evolution of Web standards

- Formulated by 2000, Roy Fielding (UCI/ICS)

The importance of REST

- Late-90's HTTP seen as
  - just convenient mechanism
  - just browser clients
  - not good enough for server-server interactions
- Ad-hoc use, generalized confusion
  - GET, POST, PUT … what's the difference?
- People started mapping other styles onto it
  - e.g. RPC, SOAP

- HTTP got no respect/understanding until REST was formulated

HTTP vs. REST

- REST is the conceptual story
- HTTP is an enabler of REST on the Web
- Not every use of HTTP is RESTful
- REST can be realized with other network protocols

- History lessons:
  - Realization (HTTP) came first, concepts (REST) became clear later
  - Good concepts are critically important

REST Design Principles

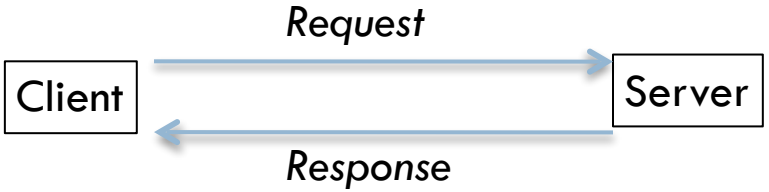
- Client-server / Request-Response
- Stateless
- Uniform interface
- Caching
- Layered
- Code-on-demand

REST in action

```
$ python tf-33.py
   What would you like to do?
   1 - Quit
   2 - Upload file
U> 2
   Name of file to upload?
U> ../pride-and-prejudice.txt

   #1: mr - 786

   What would you like to do next?
   1 - Quit
   2 - Upload file
   3 - See next most-frequently occurring word
U> 3

   #2: elizabeth - 635

   What would you like to do next?
   1 - Quit
   2 - Upload file
   3 - See next most-frequently occurring word
```

Design Principle: Request-Response

- Components
  - Servers provide access to resources
  - Clients access the resources via servers




```
request = ["get", "default", None]
while True:
    # "server"-side computation
    state_representation, links = handle_request(*request)
    # "client"-side computation
    request = render_and_get_input(state_representation, links)
```

Design Principle: Uniform Interfaces

- Uniform identification of resources
- Manipulation of resources via representations
- Hypermedia as engine of app state

TF Resources

- Execution
- Default
- File
- Word

TF Uniform Interface

- [verb, resource, [data]]
  - Verb: get / post
- Representation of resources
  - Text (menu options) +
    Links (possible next operations on resources)

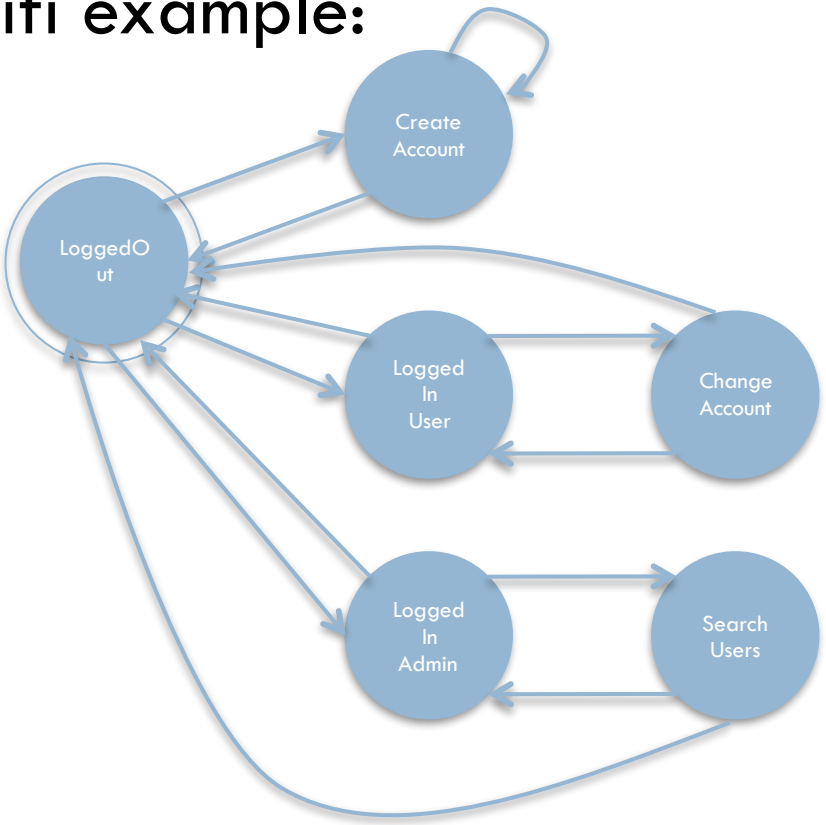
HATEOAS

Representations

- Server returns representations of resources, not the resources themselves.
  - E.g. HTML, XML
- Server response contains all metadata for client to interpret the representation

HATEOAS

- Hypermedia As The Engine Of Application State
- Insight: the application is a state machine
- Wifi example:

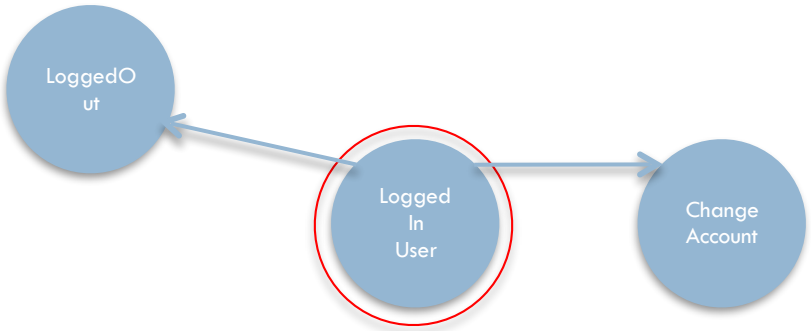


Question is:
Where is the clients' state stored?

…

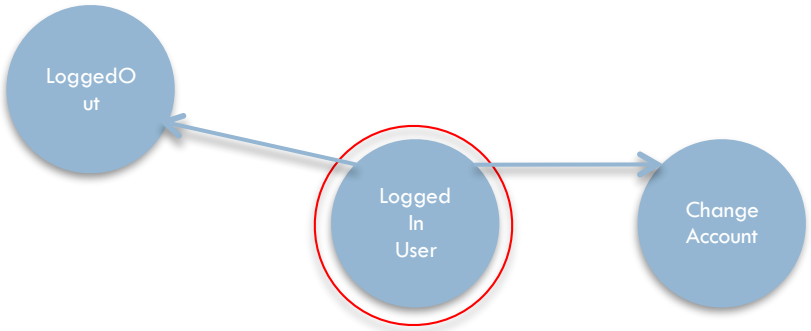
HATEOAS

- In many systems, clients' state is kept on the server
  - Traditional way of engineering apps
    - Server is both the state machine and the holder of state
- In REST, state machine is on the server, but clients' state is sent to the clients
  - At any step, client is sent a complete "picture" of where it can go next



HATEOAS

- Server sends representation of the client's state back to the client
  - Hence, REpresentional State Transfer
- Server does not "hold on" to client's state
- Possible next state transitions of the client are encoded in Hypermedia
  - Anchors, forms, scripted actions, eXternal reps



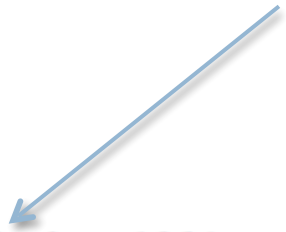
Design Principle: Stateless

- Stateless interaction, not stateless servers
- Stateless interaction:
  - Messages are self-contained, every message from client to server is independent of prior messages
- Server may create resources (e.g. session info) regarding clients
  - Critical for real applications
  - Preferably in DB
- After serving, server does not “hold on”

TF Statelessness

- Memory is sent back to client in hyperlinks

```
60      links = {"1" : ["post", "execution", None],
61               "2" : ["get", "file_form", None],
62               "3" : ["get", "word", [filename, word_index+1]]}
```



RESTful Design Guidelines

- Embrace hypermedia
  - Name your resources/features with URIs
  - Design your namespace carefully
- Hide mechanisms
  - Bad: http://example.com/cgi-bin/users.pl?name=John
  - Good: http://example.com/users/John
- Serve POST, GET, PUT, DELETE on those resources
  - Roughly, Create, Retrieve, Update, Delete (CRUD) life-cycle
- Don't hold on to state
  - Serve and forget (functional programming-y)
- Consider serving multiple representations
  - HTML, XML

RESTful Design Guidelines

- URIs are *nouns*
- The 8 HTTP operations are *verbs*

HTTP Operations (recap)

- GET
- PUT
- DELETE
- HEAD
- OPTIONS
- TRACE
- POST
- CONNECT

Idempotent methods

Means: the side effects of many invocations are exactly the same as the side effects of one invocation

See Wikipedia Idempotent

- Spec

REST, back to the beginning

- REpresentational State Transfer
  - Now you *really* know what this means!
- Explanation of HTTP 1.1 (for the most part)
  - Much needed conceptual model
- Style of writing distributed applications
  - Design guidelines
- "Story" that guides the evolution of Web standards
  - A lighthouse for new ideas