CS169.1x Lecture 5: SaaS Architecture and Introduction to Rails

Fall 2012

Web at 100,000 feet

- The web is a c*lient/server* architecture
- It is fundamentally *request/reply oriented*

§2.1 100,000 feet
- Client-server (vs. P2P)

§2.2 50,000 feet
- HTTP & URIs

§2.3 10,000 feet
- XHTML & CSS

§2.4 5,000 feet
- 3-tier architecture
- Horizontal scaling

§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)

§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)

- **Active Record**
- Data Mapper
- **REST**
- **Template View**
- Transform View

Now that we're talking, what do we say?
Hypertext Transfer Protocol

- an *ASCII-based request/reply protocol* for transferring information on the Web
- *HTTP request* includes:
  - *request method* (`GET`, `POST`, etc.)
  - Uniform Resource Identifier (URI)
  - HTTP protocol version understood by the client
  - *headers*—extra info regarding transfer request
- *HTTP response* from server
  - Protocol version & Status code =>
  - Response headers
  - Response body

HTTP status codes:
2xx — *all is well*
3xx — *resource moved*
4xx — *access problem*
5xx — *server error*

Dynamic content generation

- In the Elder Days, most web pages were (collections of) plain old files
- But most interesting Web 1.0/e-commerce sites actually *run a program* to generate the "page"
- Originally: templates with embedded code "snippets"
- Eventually, code became "tail that wagged the dog" and moved out of the Web server

Sites that are really programs (SaaS)

- How do you:
  - "map" URI to correct program & function?
  - pass arguments?
  - invoke program on server?
  - handle persistent storage?
  - handle cookies?
  - handle errors?
  - package output back to user?
- *Frameworks* support these common tasks

Developer environment vs. medium-scale deployment

Developer

Medium-scale deployment

Large-scale curated deployment, e.g. Heroku

Match the terms:
(a) presentation tier, (b) logic tier,
(c) persistence tier

- ☐ (a) Apache web server (b) Rack+Rails (c) Relational database
- ☐ (a) Firefox (b) Apache web server (c) PostgreSQL
- ☐ (a) Microsoft Internet Information Server (b) Rack+Rails (c) Apache web server
- ☐ (a) Firefox (b) Microsoft Internet Information Server (c) MySQL

§2.1 100,000 feet
- Client-server (vs. P2P)

§2.2 50,000 feet
- HTTP & URIs

§2.3 10,000 feet
- XHTML & CSS

§2.4 5,000 feet
- 3-tier architecture
- Horizontal scaling

§2.5 1,000 feet—Model-View-Controller (vs. Page Controller, Front Controller)

§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)

- **Active Record**
- Data Mapper
- **REST**
- **Template View**
- Transform View

The MVC Design Pattern

- Goal: separate organization of data (model) from UI & presentation (view) by introducing *controller*
  - mediates user actions requesting access to data
  - presents data for *rendering* by the view
- Web apps may seem "obviously" MVC by design, but other alternatives are possible...

Each entity has a model, controller, & set of views

Alternatives to MVC

Page Controller
(Ruby Sinatra)

Front Controller
(J2EE servlet)

Template View
(PHP)

Rails supports SaaS apps structured as MVC, but other architectures may be better fit for some apps.

Which statement is **NOT** true about the Model-View-Controller (MVC) architectural pattern:

- ☐ In SaaS apps on the Web, controller actions and view contents are transmitted using HTTP.
- ☐ All MVC apps have both a "client" part (e.g. Web browser) and a "cloud" part (e.g. Rails app on cloud).
- ☐ Model-View-Controller is just one of several possible ways to structure a SaaS app.
- ☐ Peer-to-peer apps can be structured as Model-View-Controller.

§2.1 100,000 feet
- Client-server (vs. P2P)

§2.2 50,000 feet
- HTTP & URIs

§2.3 10,000 feet
- XHTML & CSS

§2.4 5,000 feet
- 3-tier architecture
- Horizontal scaling

§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)

§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)

- **Active Record**
- Data Mapper
- **REST**
- **Template View**
- Transform View

In-Memory vs. In-Storage objects

```
#<Movie:0x1295580>
m.name, m.rating, ...
```

```
#<Movie:0x32ffe416>
m.name, m.rating, ...
```

- How to represent persisted object in storage
  - Example: Movie and Reviews
- Basic operations on object: CRUD (Create, Read, Update, Delete)
- ActiveRecord: every model knows how to CRUD itself, using common mechanisms

Rails Models Store Data in Relational Databases (RDBMS)

- Each type of model gets its own database *table*
  - All rows in table have identical structure
  - 1 row in table == one model instance
  - Each column stores value of an *attribute* of the model
  - Each row has unique value for *primary key* (by convention, in Rails this is an integer and is called *id*)

id	rating	title	release_date
2	G	Gone With the Wind	1939-12-15
11	PG	Casablanca	1942-11-26
...	...	...	...
35	PG	Star Wars	1977-05-25

- *Schema:* Collection of all tables and their structure

Alternative: DataMapper

- Data Mapper associates separate *mapper* with each model
  - Idea: keep mapping *independent* of particular data store used => works with more types of databases
  - Used by Google AppEngine
  - Con: can't exploit RDBMS features to simplify complex queries & relationships
- We'll revisit when talking about *associations*

Which statement is ***not*** true about the Model in Model-View-Controller:

- ☐ The CRUD actions only apply to models backed by a database that supports ActiveRecord.
- ☐ Part of the Model's job is to convert between in-memory and stored representations of objects.
- ☐ Although Model data is displayed by the View, a Models' direct interaction is with Controllers.
- ☐ Although DataMapper doesn't use relational databases, it's a valid way to implement a Model.

Routes

- In MVC, each interaction the user can do is handled by a *controller action*
  - Ruby method that handles that interaction
- A *route* maps `<HTTP method, URI>` to controller action
-

Route	Action
`GET /movies/3`	Show info about movie whose ID=3
`POST /movies`	Create new movie from attached form data
`PUT /movies/5`	Update movie ID 5 from attached form data
`DELETE /movies/5`	Delete movie whose ID=5

Brief Intro to Rails' Routing Subsystem

- dispatch <method,URI> to correct controller action
- provides *helper methods* that generate a <method,URI> pair given a controller action
- parses query *parameters* from both URI and form submission into a convenient hash
- Built-in shortcuts to generate all CRUD routes (though most apps will also have other routes)

rake routes

```
I     GET /movies          {:action=>"index", :controller=>"movies"}
C    POST /movies          {:action=>"create", :controller=>"movies"}
      GET /movies/new      {:action=>"new", :controller=>"movies"}
      GET /movies/:id/edit {:action=>"edit", :controller=>"movies"}
R     GET /movies/:id      {:action=>"show", :controller=>"movies"}
U     PUT /movies/:id      {:action=>"update", :controller=>"movies"}
D  DELETE /movies/:id      {:action=>"destroy", :controller=>"movies"}
```

GET /movies/3/edit HTTP/1.0

- Matches route:

```
GET /movies/:id/edit {:action=>"edit", :controller=>"movies"}
```

- Parse wildcard parameters: `params[:id] = "3"`
- Dispatch to `edit` method in `movies_controller.rb`
- To include a URI in generated view that will submit the form to the update controller action with `params[:id]==3`, call helper:

```
update_movie_path(3) # => PUT /movies/3
```

rake routes

```
I     GET /movies           {:action=>"index", :controller=>"movies"}
C    POST /movies           {:action=>"create", :controller=>"movies"}
      GET /movies/new       {:action=>"new", :controller=>"movies"}
      GET /movies/:id/edit  {:action=>"edit", :controller=>"movies"}
R     GET /movies/:id       {:action=>"show", :controller=>"movies"}
U     PUT /movies/:id       {:action=>"update", :controller=>"movies"}
D  DELETE /movies/:id       {:action=>"destroy", :controller=>"movies"}
```

REST (Representational State Transfer)

- Idea: *Self-contained* requests specify what *resource* to operate on and what to do to it
  - Roy Fielding's PhD thesis, 2000
  - Wikipedia: "a *post hoc description of the features that made the Web successful*"
- A service (in the SOA sense) whose operations are like this is a RESTful service
- Ideally, RESTful URIs name the operations
- Let's see an *anti-example:*

http://pastebin.com/edF2NzCF

Which statement is **NOT** true regarding Rails RESTful routes and the resources to which they refer:

- ☐ A *resource* may be existing content or a request to modify something.
- ☐ Every route must eventually trigger a controller action.
- ☐ One common set of RESTful actions is the CRUD actions on models.
- ☐ The route always contains one or more parameters, such as `:id`, to identify the resource

Rails as an MVC Framework

Model, View, Controller

Subclasses of ActiveRecord::Base, an *object-relational mapping* layer

Subclasses of ActionView

Subclasses of ApplicationController

Persistence: mysql or sqlite3

Logic: your code & Rack appserver

Presentation: WEBrick, Apache, etc.

Client: Firefox

A trip through a Rails app

1. Routes (in routes.rb) map incoming URL's to *controller actions* and extract any optional *parameters*
   - Route's "wildcard" parameters (eg :id), plus any stuff after "?" in URL, are put into params[] hash accessible in controller actions
2. Controller actions set *instance variables,* visible to *views*
   1. Subdirs and filenames of views/ match controllers & action names
- Controller action eventually *renders* a view

config/
routes.rb

app/controllers/movies_controller.rb

```
def show id = params[:id]
 @mv=Movie.find(id)
end
```

app/views/movies/show.html.haml

```
%li
 Rating:
 = @mv.rating
```

```
GET /movies/:id
{:action=>'show',:controller=>'movies'}
```

Why must every interaction with a SaaS app eventually cause something to be rendered?

- ☐ Because of convention over configuration
- ☐ Because HTTP is a request-reply protocol
- ☐ Because Model-View-Controller implies that every action renders its own View
- ☐ All of the above