CS 169 Software Engineering SaaS Architecture

Armando Fox, David Patterson, and Koushik Sen

Spring 2012

Outline and § ELLS sections

From 100,000 foot view to 500 foot view of SaaS

- Client-server architecture, HTTP, URIs, cookies (§ 2.1-2.2)
- HTML & CSS, XML & XPath (§ 2.2-2.3)
- 3-tier shared-nothing architecture, horizontal scaling (§ 2.4)
- model-view-controller design pattern (§ 2.5)
  - Models: ActiveRecord & CRUD (§ 2.6)
  - Routes, controllers, and REST (§ 2.7)
  - Template views (§ 2.8)
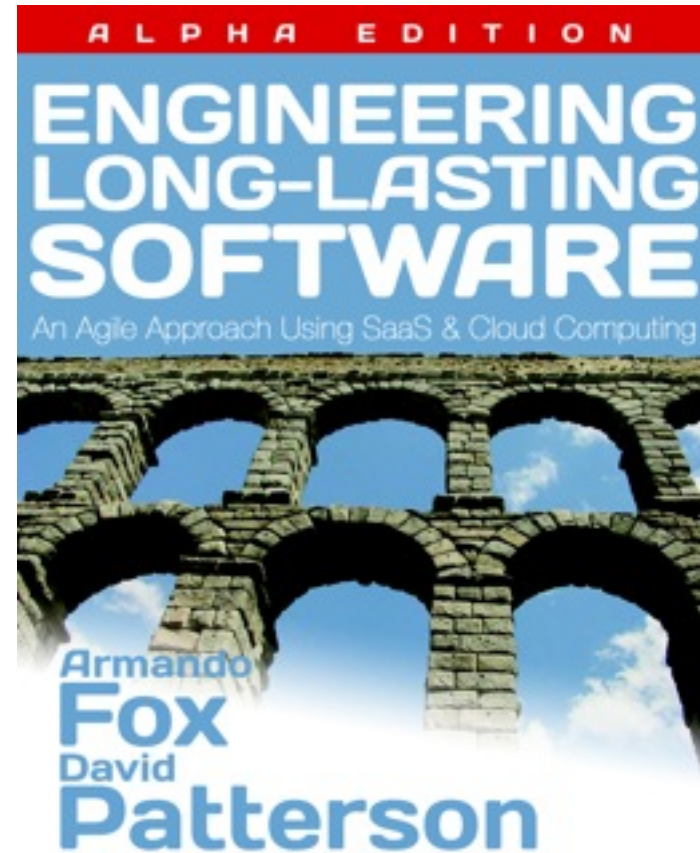- Fallacies & pitfalls, perspectives (§ 2.9-2.12)

The Web as a Client-Server System; TCP/IP intro
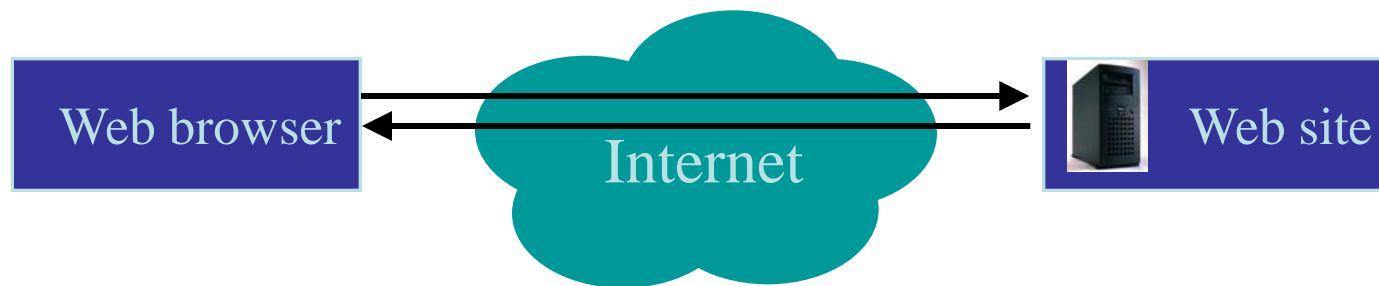
ELLS §2.1–2.2

Armando Fox

3

Reminder

- This material expands on and gives additional perspectives/demos about the material in *ELLS*

- We assume you will watch the screencasts too (screencast.saasbook.info)

- It *does not replace* reading that material

http://saasbook.info

Web at 100,000 feet

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

Web browser → Internet → Web site

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

Browser (Firefox, Chrome...)

Internet

Site *rotten-potatoes.com*

§2.2 50,000 feet
• HTTP & URIs

§2.3 10,000 feet
• XHTML & CSS

html

css

Web server *(Apache, Microsoft IIS, WEBrick)*

App server *(rack)*

Database *(Postgres, SQLite)*

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

Presentation tier

Logic tier

Persistence tier

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

Models

Control-lers

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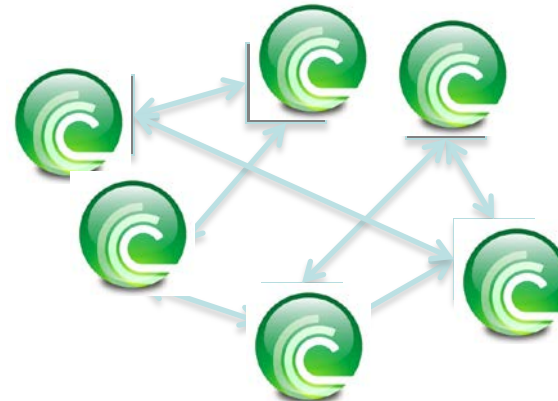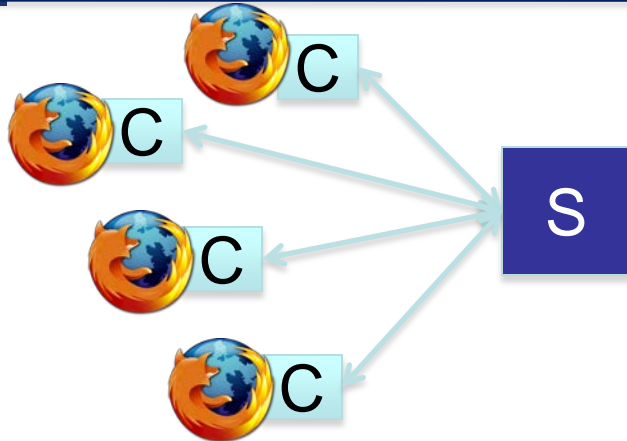
§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)

• **Active Record**
• Data Mapper

• **REST**

• **Template View**
• Transform View

6
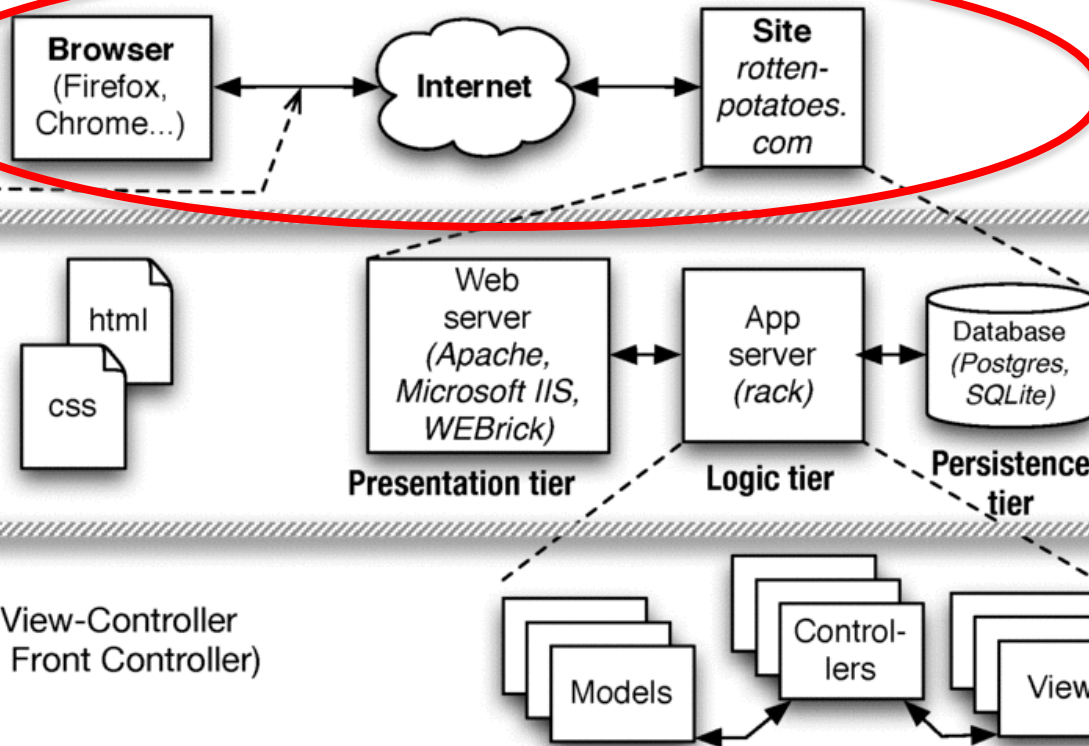
Client-Server vs. Peer-to-Peer

- High-level architecture of the overall system
  - Soon we'll talk about architecture "inside" boxes
- Client & server each *specialized* for their tasks
  - Client: ask questions on behalf of users
  - Server: wait for & respond to questions, serve many clients
- Design Patterns capture common structural solutions to recurring problems
  - Client-Server is an *architectural pattern*

Nuts and bolts: TCP/IP protocols

- IP (Internet Protocol) *address* identifies a physical network interface with four *octets,* e.g. `128.32.244.172`
  - Special address `127.0.0.1` is "this computer", named `localhost`, even if not connected to the Internet!

- TCP/IP (Transmission Control Protocol/Internet Protocol)
  - IP: no-guarantee, best-effort service that delivers *packets* from one IP address to another
  - TCP: make IP reliable by detecting "dropped" packets, data arriving out of order, transmission errors, slow networks, etc., and respond appropriately
  - TCP *ports* allow multiple TCP apps on same computer

- Vint Cerf & Bob Kahn: 2004 Turing Award for Internet architecture & protocols, incl. TCP/IP

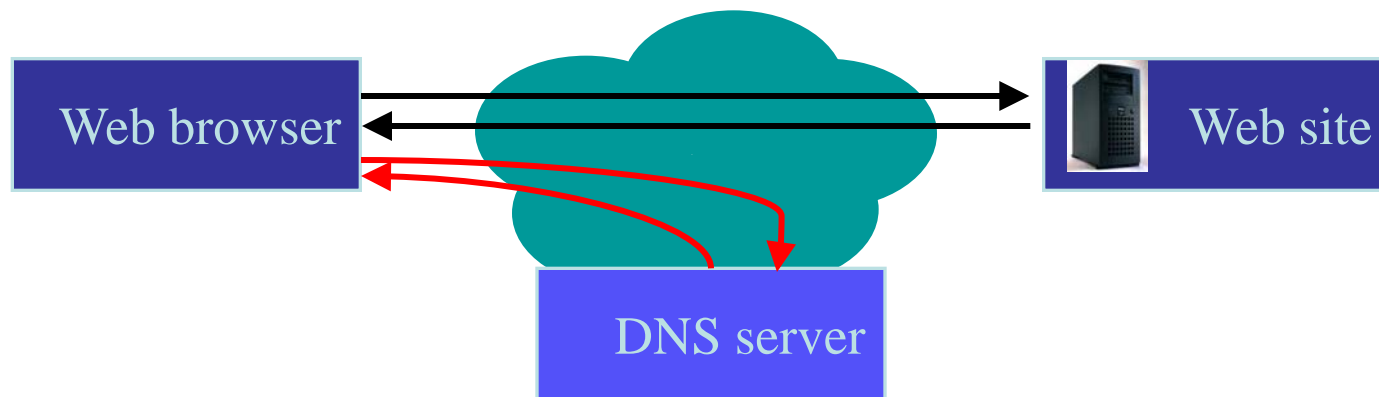`GET /bears/`   →   `GET /bears/`

`HTTP/0.9 200 OK`   ←   `HTTP/0.9 200 OK`

Web at 100,000 feet

- The web is a *client/server* architecture
- It is fundamentally *request/reply oriented*
- Domain Name System (DNS) is another kind of server that maps *names* to *IP addresses*

Web browser

Web site

DNS server

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*

Assuming ">" means "relies on", which statement is NOT correct:

☐ DNS > IP

☐ HTTP > TCP > IP

☐ TCP > DNS

☐ All the above are correct

Cookies

- Observation: *HTTP is stateless*

- Early Web 1.0 problem: how to guide a user "through" a flow of pages?
  - use IP address to identify returning user?
    - ✖ public computers, users sharing single IP
  - embed per-user junk into URI query string?
    - ✖ breaks caching

- Quickly superseded by *cookies*
  - Watch: *screencast.saasbook.info*

Uses of cookies

- Most sites quickly realized that the per-user state could be used for lots of things:
  - customization ("My Yahoo")
  - click tracking/flow tracking
  - *authentication* (logged in or not)
  - *Which of these could be implemented on the client side? Which ones shouldn't be and why?*
- A golden rule: *don't trust the client*—cookies must be tamper-evident

A _____ can create and modify cookies; the _____ is responsible for including the correct cookie with each request

☐ Browser; SaaS app

☐ SaaS app; browser

☐ HTTP request; browser

☐ SaaS app; HTTP response

HTML+CSS

ELLS § 2.3
Armando Fox

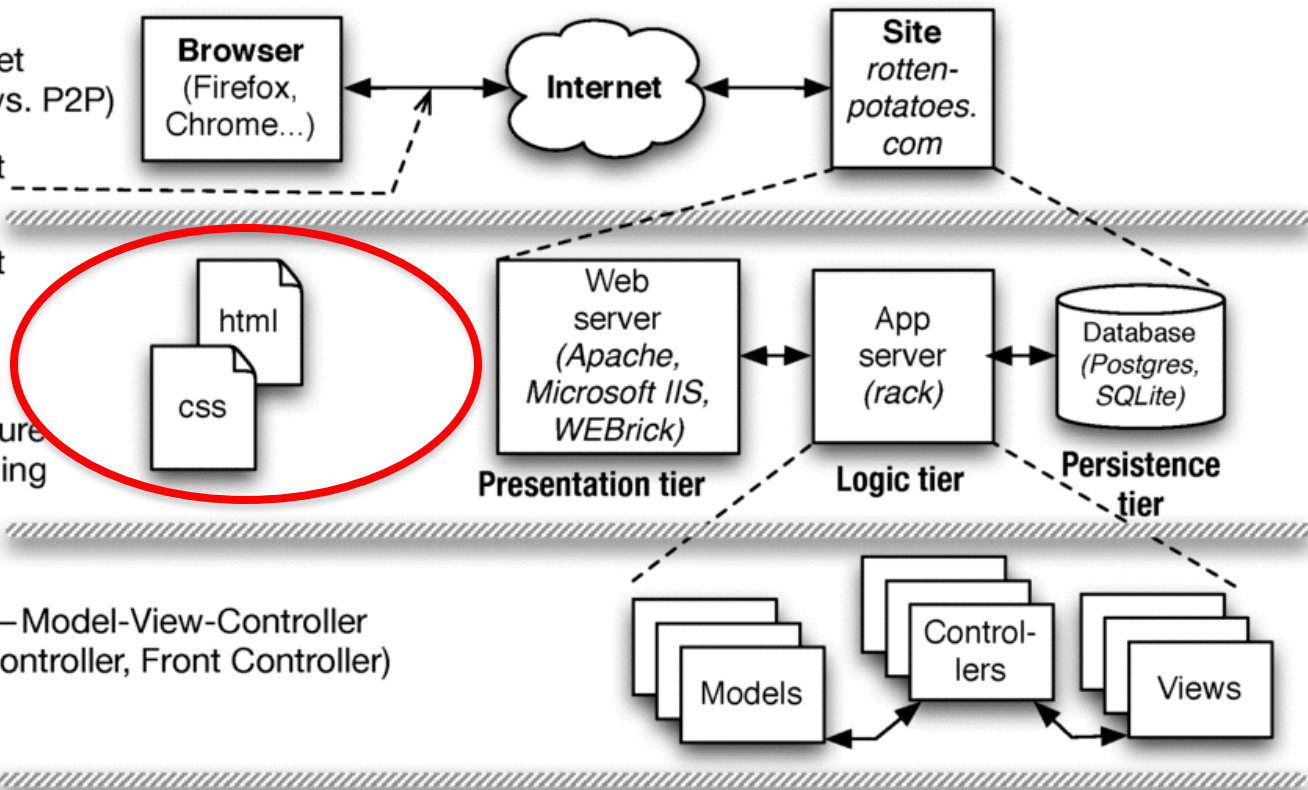§2.1 100,000 feet
• Client-server (vs. P2P)

§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)

Browser (Firefox, Chrome...)

Internet

Site *rotten-potatoes.com*

html

css

Web server *(Apache, Microsoft IIS, WEBrick)*

Presentation tier

App server *(rack)*

Logic tier

Database *(Postgres, SQLite)*

Persistence tier

Models

Control-lers

Views

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

16

Introduction

This article is a review of the book
Dietary Preferences of Penguins,
by Alice Jones and Bill Smith. Jones
and Smith's controversial work makes
three hard-to-swallow claims about
penguins:

First, that penguins actually prefer
tropical foods such as bananas and
pineapple to their traditional diet
of fish

Second, that tropical foods give
penguins an odor that makes them
unattractive to their traditional
predators

```html
<h1>Introduction</h1>
<p>
  This article is a review of the book
  <i>Dietary Preferences of Penguins</i>,
  by Alice Jones and Bill Smith. Jones and Smith's
  controversial work makes three hard-to-swallow claims
  about penguins:
</p>
<ul>
  <li>
   First, that penguins actually prefer tropical foods
   such as bananas and pineapple to their traditional diet
   of fish
  </li>
  <li>
   Second, that tropical foods give penguins an odor that
   makes them unattractive to their traditional predators
  </li>
</ul>

...
```

Introduction

This article is a review of the book *Dietary Preferences of Penguins*, by Alice Jones and Bill Smith. Jones and Smith's controversial work makes two hard-to-swallow claims about penguins:

- First, that penguins actually prefer tropical foods such as bananas and pineapple to their traditional diet of fish
- Second, that tropical foods give penguins an odor that makes them unattractive to their traditional predators

...

```
<h1>Introduction</h1>
<p>
This article is a review of the book
<i>Dietary Preferences of Penguins</i>,
by Alice Jones and Bill Smith. Jones
and Smith's controversial work makes
three hard-to-swallow claims about
penguins:
<ul>
<li>
First, ...
```

HTML ~1.0

- Descendant of IBM's Generalized Markup Language (1960's) via SGML (Standard Generalized Markup Language, 1986)
- Document = Hierarchical collection of *elements*
  - inline (headings, tables, lists...)
  - embedded (images, JavaScript code...)
  - forms—allow user to submit simple input (text, radio/check buttons, dropdown menus...)
- Each element can have *attributes* (many optional) and some elements also have *content*
  - of particular interest: *id* and *class* attributes, for *styling*

Cascading Style Sheets

- Idea: *visual appearance* of page described in a separate document (*stylesheet*)
  - accessibility
  - branding/targeting
  - separate designers' & developers' concerns
- *Current best practice: HTML markup should contain **no** visual styling information*

How does it work?

- `<link rel="stylesheet" href="http://..."/>` (inside `<head>` element) says what stylesheet goes with this HTML page
- HTML `id` & `class` attributes important in CSS
  - *id* must be ***unique within this page***
  - same *class* can be attached to many elements

```
<div id="right" class="content">
  <p>
   I'm Armando.  I teach CS169 and do
   research in the AMP Lab and Par Lab.
  </p>
</div>
```

Selectors identify specific tag(s)

```
<div class="pageFrame" id="pageHead">
   <h1>
     Welcome,
     <span id="custName">Armando</span>
     <img src="welcome.jpg" id="welcome"/>
   </h1>
</div>
```

- tag name: `h1`
- class name: `.pageFrame`
- element ID: `#pageHead`

  both of these match the outer *div* above. Don't do this!

- tag name & class: `div.pageFrame`
- tag name & id: `img#welcome` (usually redundant)
- descendant relationship: `div .custName`
- Attributes *inherit* browser defaults unless overridden

Which CSS selector will select *only* the word "bar" for styling:
`<p class="a">`foo,
 `<span class="a">`bar`<span></p>`

☐ span.a

☐ p .a

☐ .a span

☐ All of these

3-tier shared-nothing architecture & scaling

ELLS § 2.4

Armando Fox

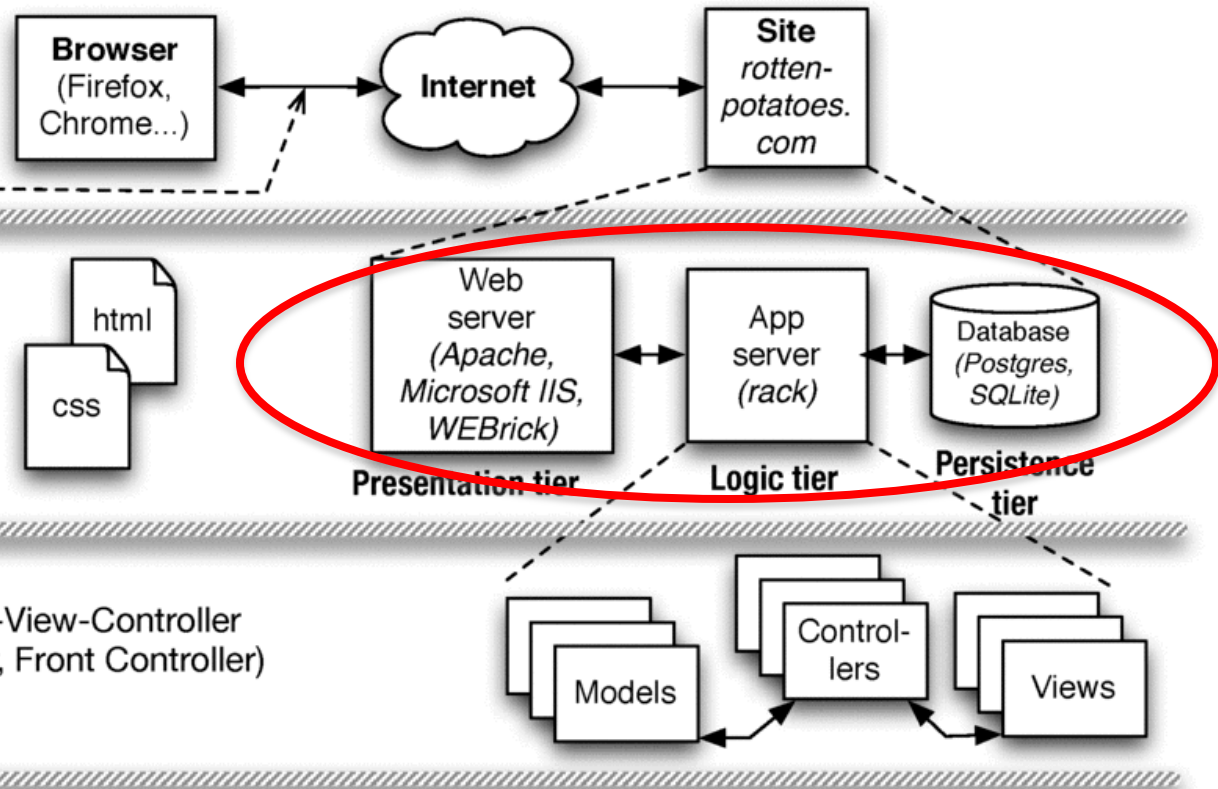§2.1 100,000 feet
• Client-server (vs. P2P)

§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)
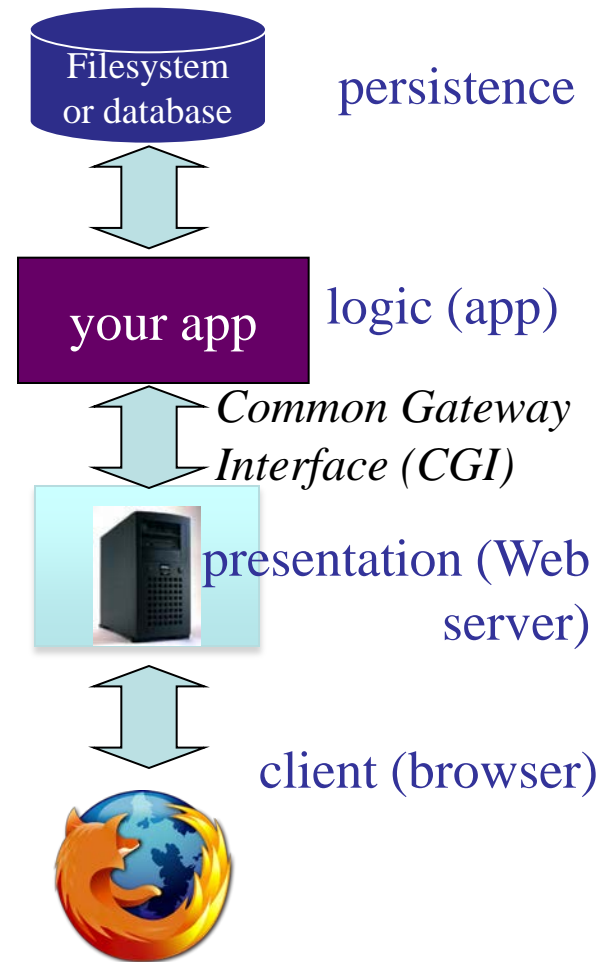§2.8 500 feet: Template View (vs. Transform View)

• **Active Record**   • **REST**   • **Template View**
• Data Mapper   • Transform View
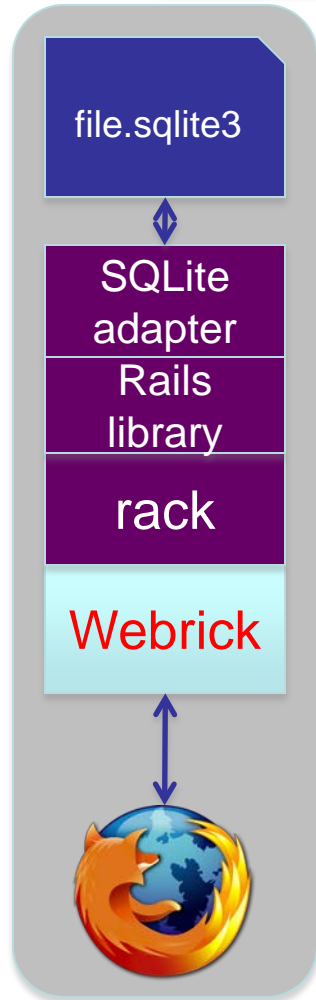
Dynamic content generation

- 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
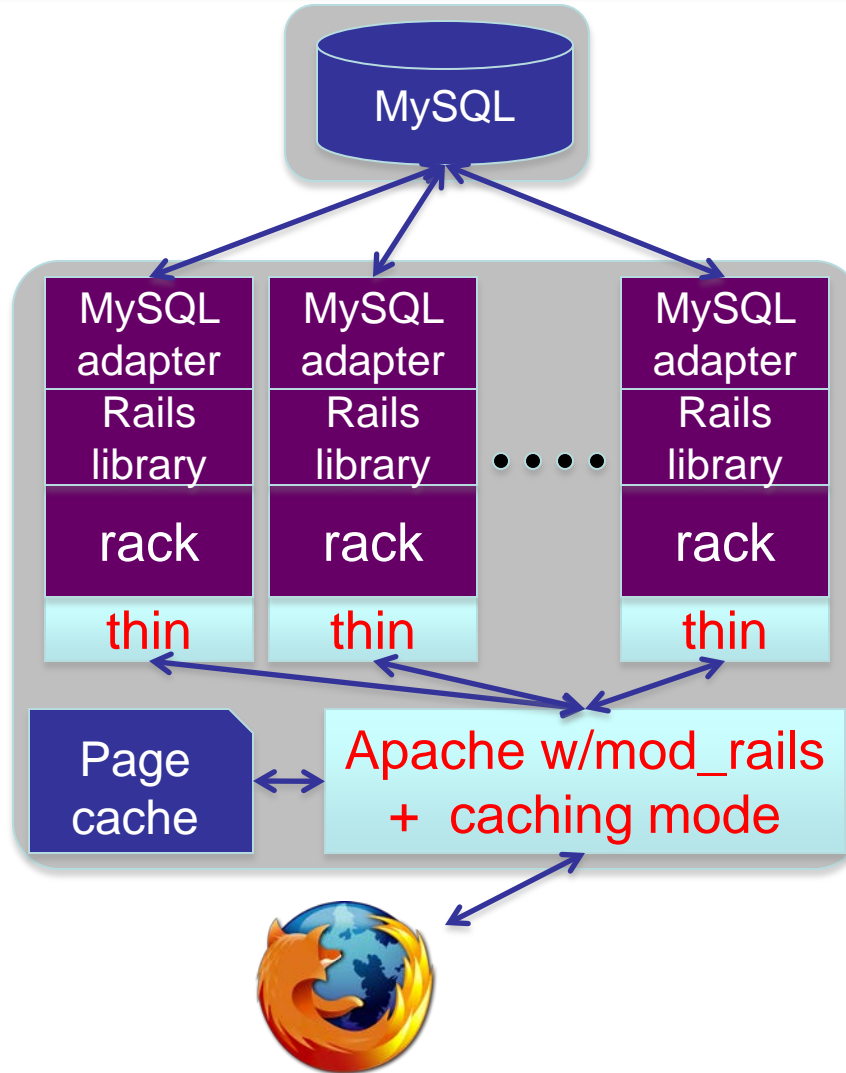
Sites that are really programs (SaaS)

- 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?
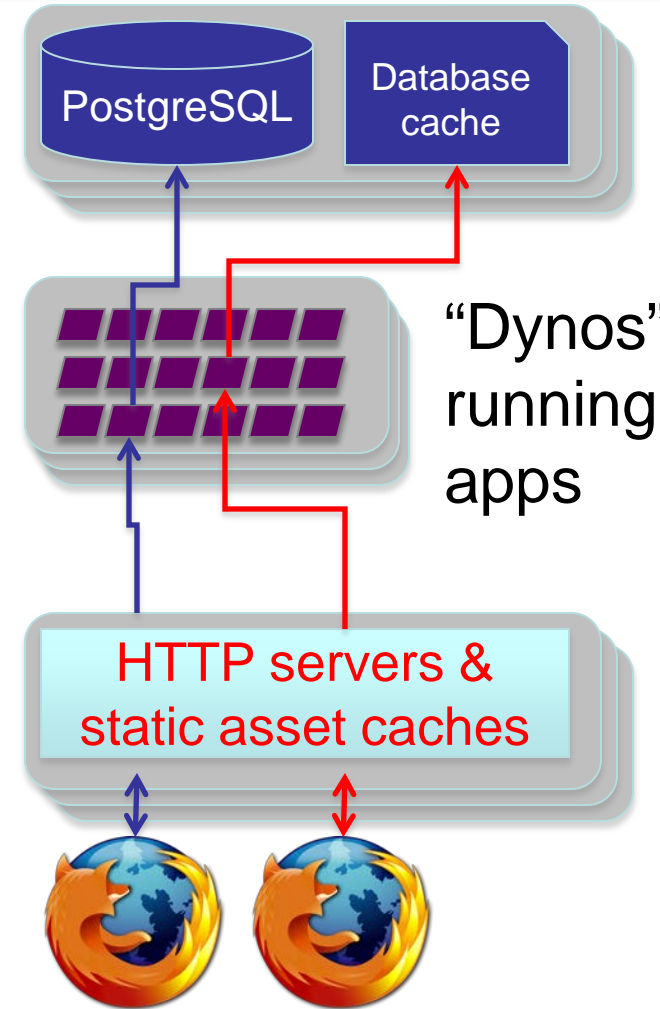- *Frameworks* support these common tasks

Filesystem or database — persistence

your app — logic (app)

Common Gateway Interface (CGI)

presentation (Web server)

client (browser)

Developer environment vs. medium-scale deployment

file.sqlite3

SQLite adapter

Rails library

rack

Webrick

Developer

MySQL

MySQL adapter	MySQL adapter	· · · ·	MySQL adapter
Rails library	Rails library		Rails library
rack	**rack**		**rack**
thin	thin		thin

Page cache

Apache w/mod_rails + caching mode

Medium-scale deployment

PostgreSQL
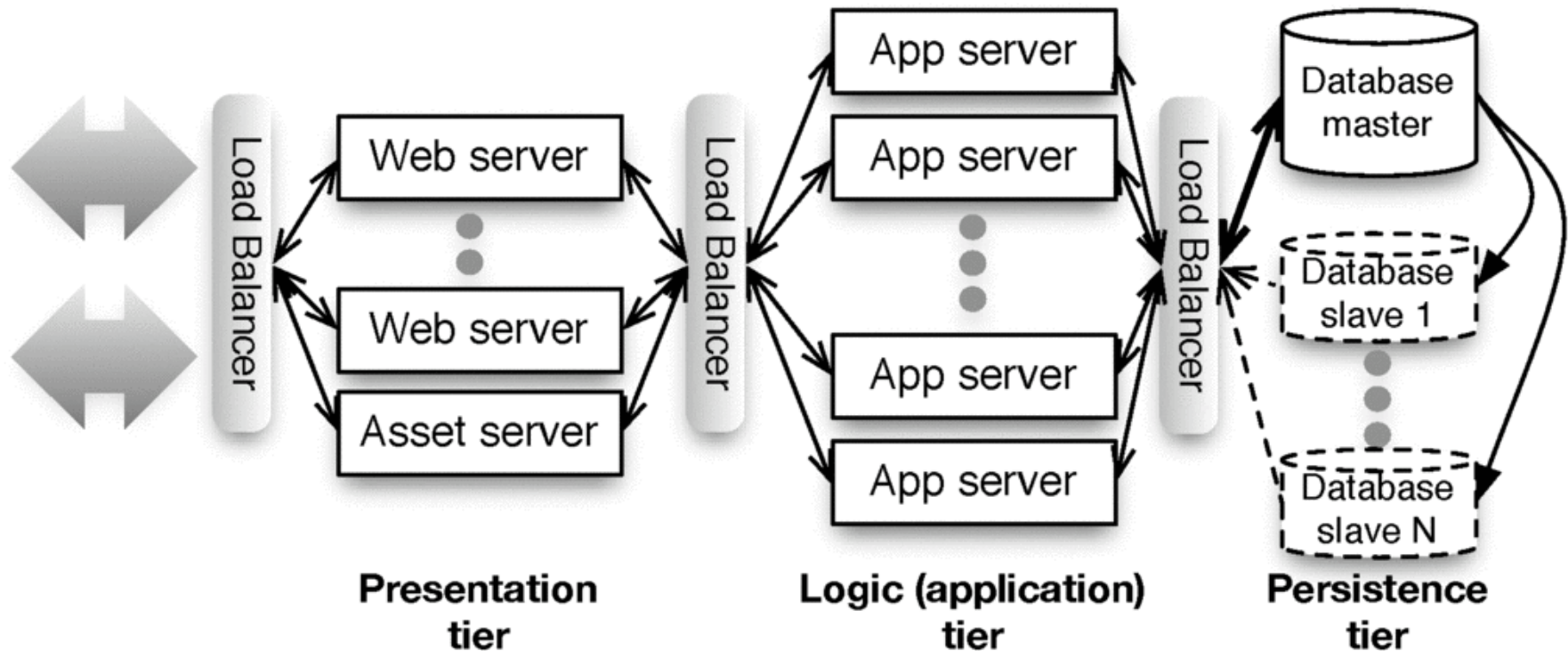
Database cache

"Dynos" running apps

HTTP servers & static asset caches

Large-scale curated deployment, e.g. Heroku

"Shared nothing"

Sharding vs. Replication

- Partition data across independent "shards"?
  - \+ Scales great
  - – Bad when operations touch >1 table
  - – Example use: user profile

- Replicate all data everywhere?
  - \+ Multi-table queries fast
  - – Hard to scale: writes must propagate to all copies => temporary *inconsistency* in data values
  - – Example: Facebook wall posts/"likes"
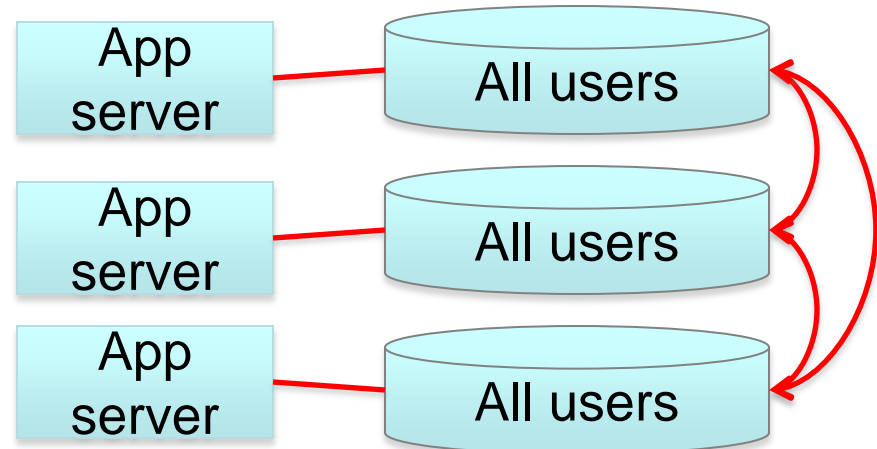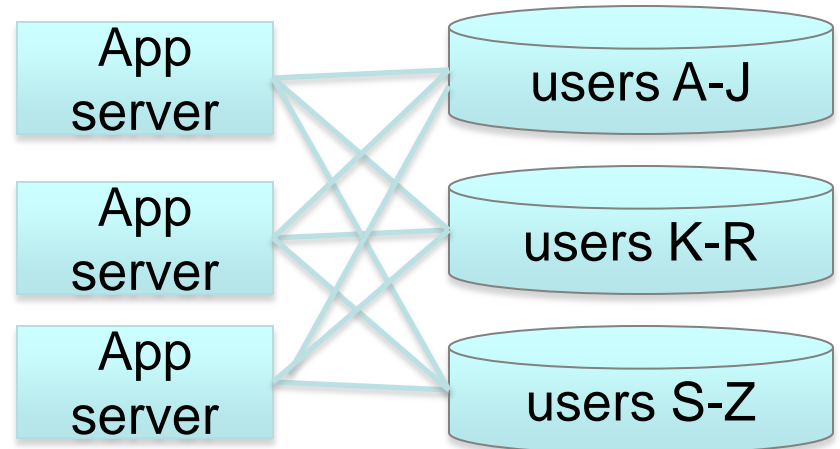
Summary: Web 1.0 SaaS

- Browser *requests* web resource (URI) using HTTP
  - HTTP is a simple request-reply protocol that relies on TCP/IP
  - In SaaS, most URI's cause a program to be run, rather than a static file to be fetched
- *HTML* is used to encode content, *CSS* to style it visually
- *Cookies* allow server to track client
  - Browser automatically passes cookie to server on each request
  - Server may change cookie on each response
  - Typical usage: cookie includes a *handle* to server-side information
  - That's why some sites don't work if cookies are completely disabled
- *Frameworks* make all these abstractions convenient for programmers to use, without sweating the details
- ...and help map SaaS to 3-tier, shared-nothing architecture

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

Model-View-Controller

ELLS § 2.5

Armando Fox

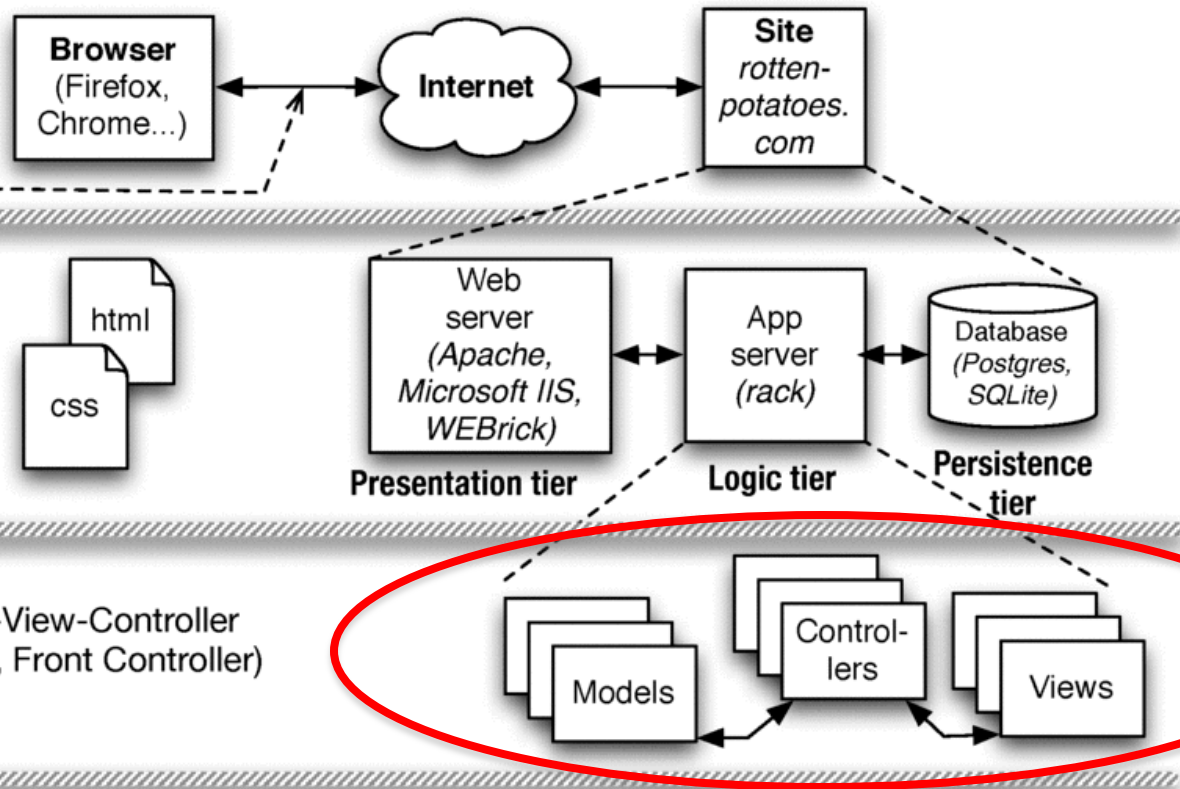§2.1 100,000 feet
• Client-server (vs. P2P)

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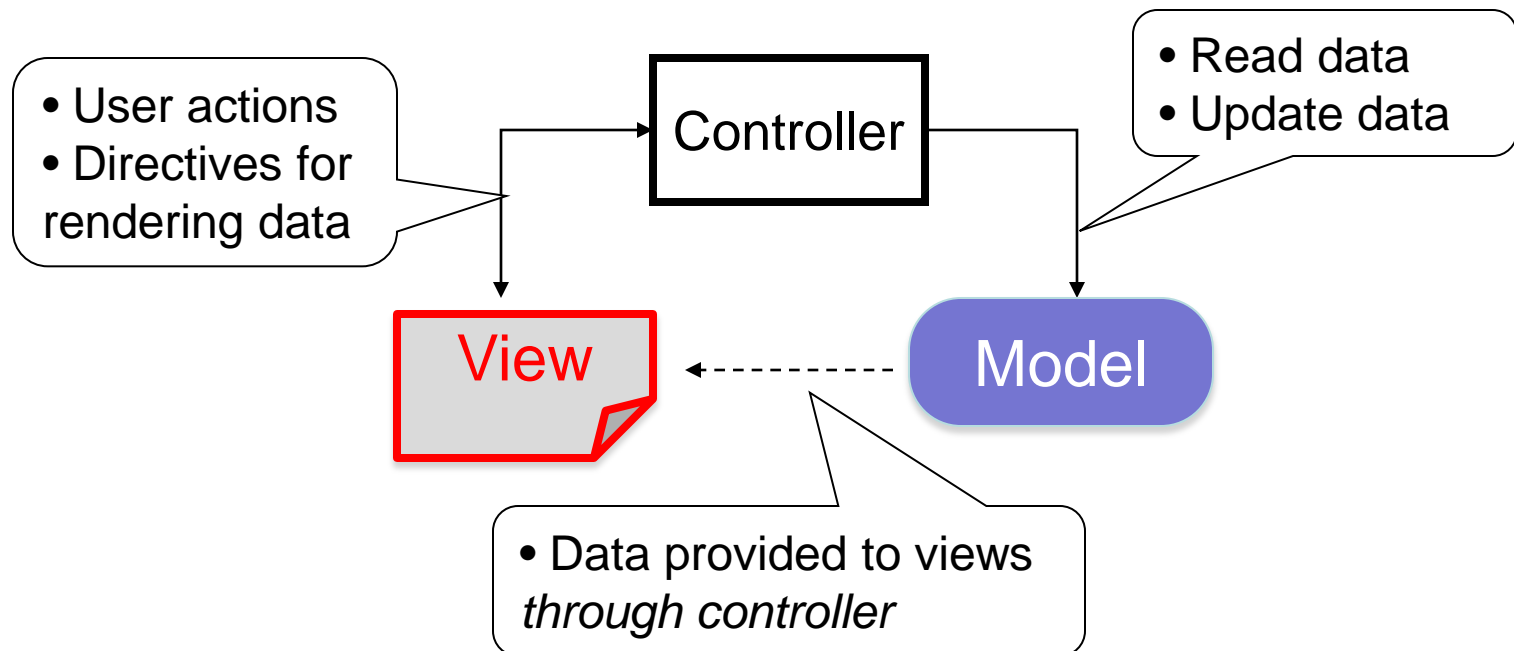
§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational
State Transfer for self-contained actions)
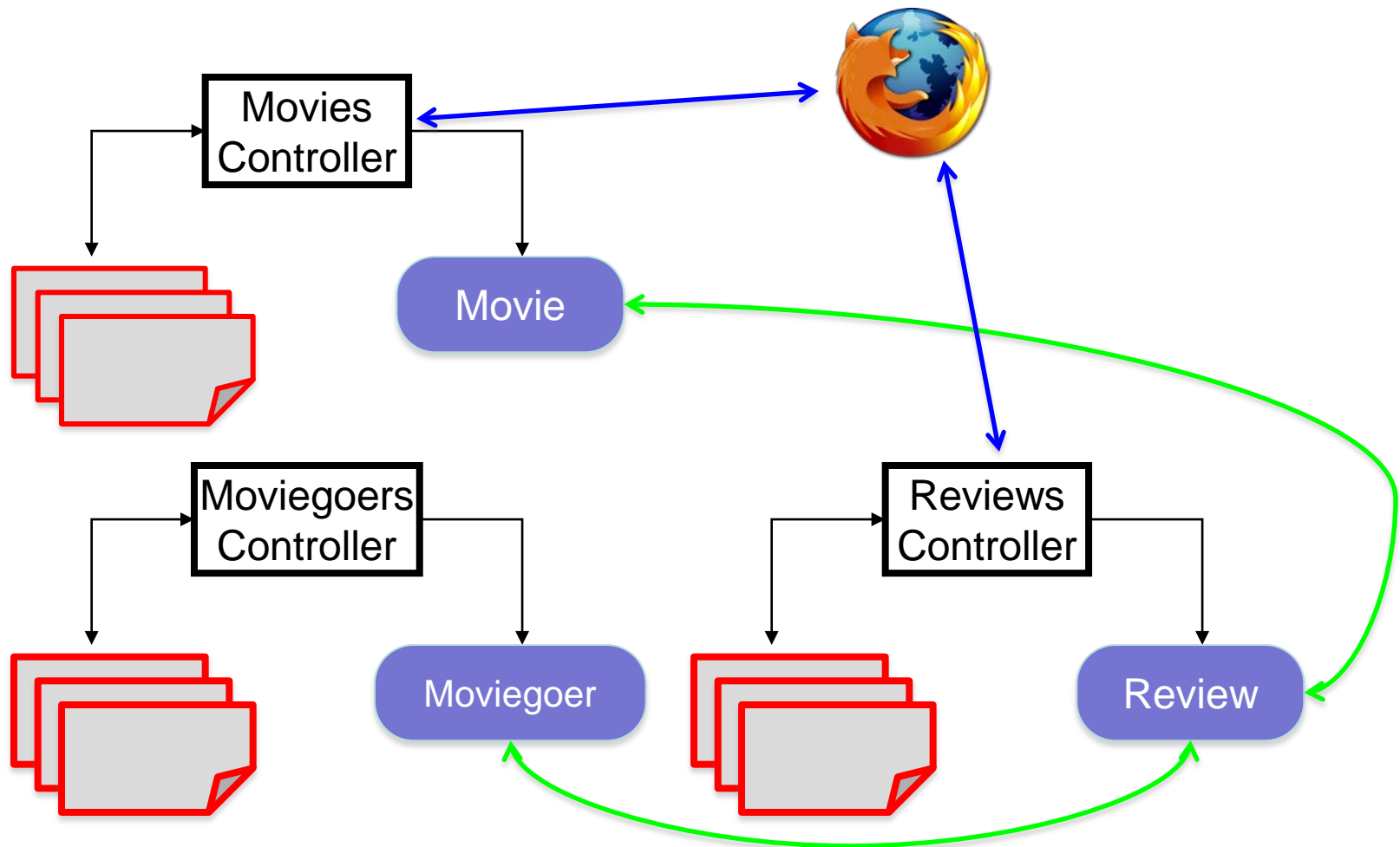§2.8 500 feet: Template View (vs. Transform View)

• **Active Record** • **REST** • **Template View**
• Data Mapper • 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...
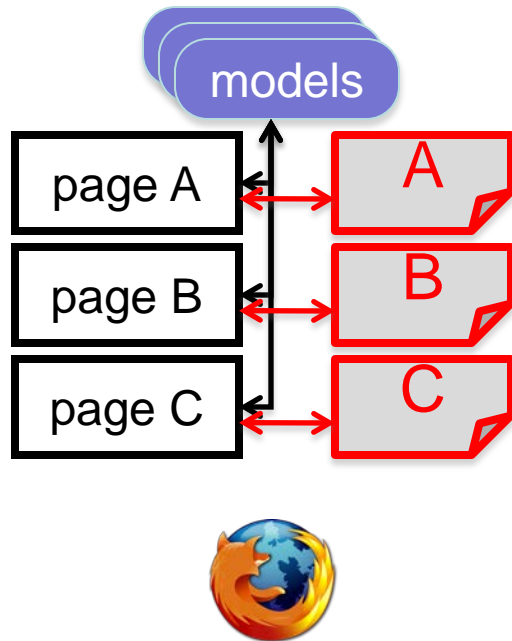

- User actions
- Directives for rendering data

Controller

- Read data
- Update data

View
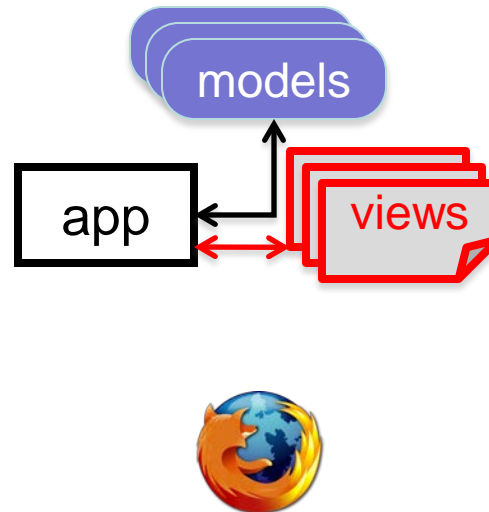
Model

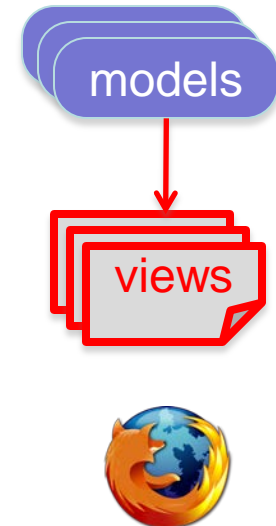- Data provided to views *through controller*

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.

Models, Databases, and Active Record

ELLS § 2.6

Armando Fox

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

§2.2 50,000 feet
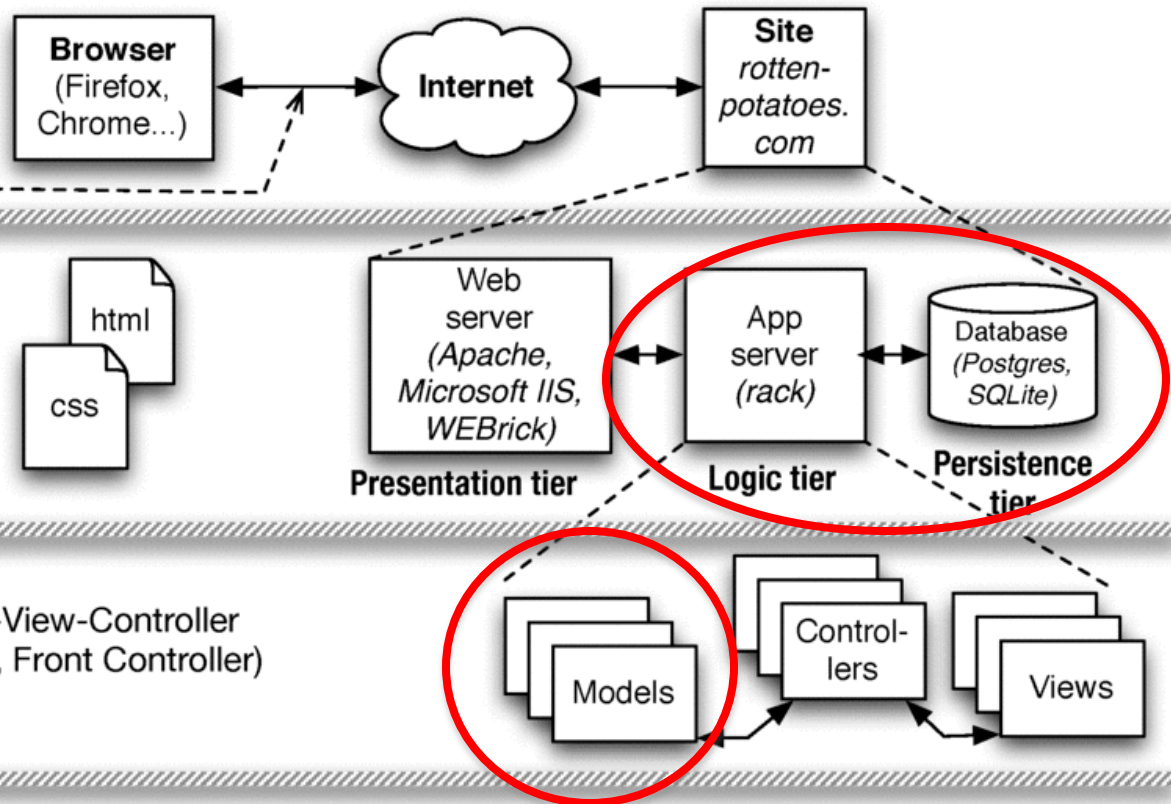• HTTP & URIs

§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)

In-Memory vs. In-Storage objects

```
#<Movie:0x1295580>
m.name, m.rating, ...
 #<Movie:0x32ffe416>
m.name, m.rating, ...
```

marshal/serialize →
← unmarshal/deserialize

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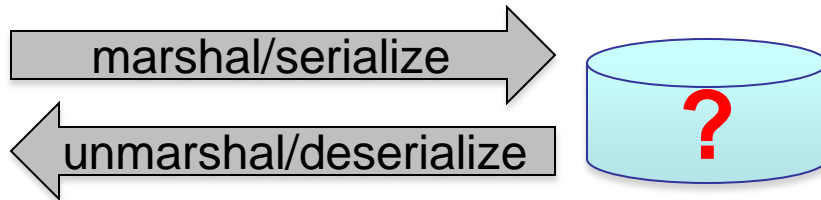
- How to represent persisted object in storage
  - Example: Movie and Reviews
- Basic operations on object: CRUD (Create, Read, Update, Delete)
- ActiveRecord: every model knows how to CRUD itself, using common mechanisms

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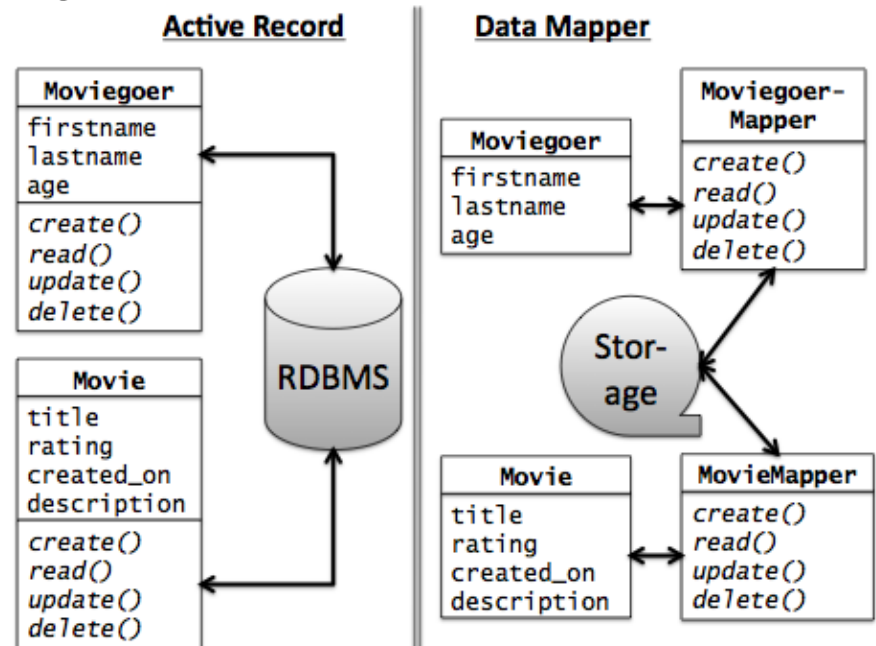
Which statement is *not* true about the Model in Model-View-Controller:

☐ 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.

Controllers, Routes, and RESTfulness

ELLS § 2.7

Armando Fox

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

Template Views and Haml

ELLS § 2.8

Armando Fox

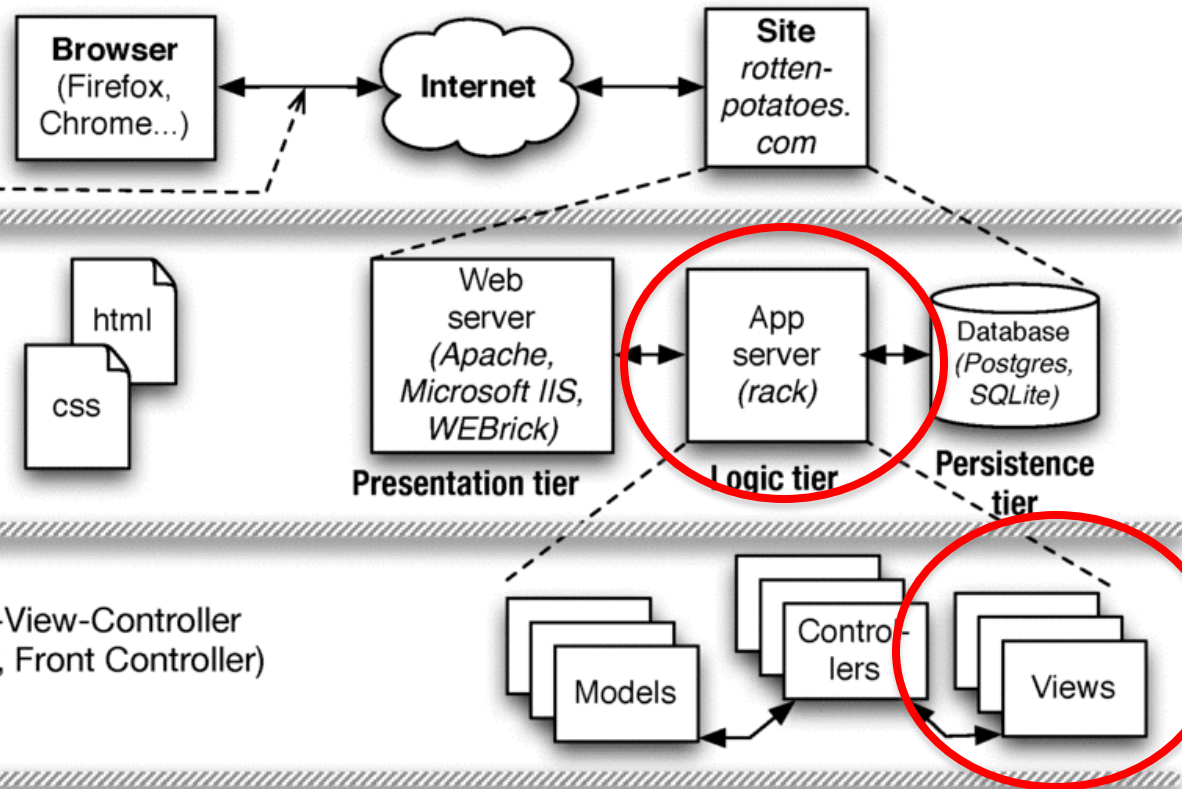§2.1 100,000 feet
- Client-server (vs. P2P)

§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**   - **REST**   - **Template View**
- Data Mapper   - Transform View

Template View pattern

- View consists of markup with selected *interpolation* to happen at runtime
  - Usually, values of variables or result of evaluating short bits of code
- In Elder Days, this *was* the app (e.g. PHP)
- *Alternative:* Transform View
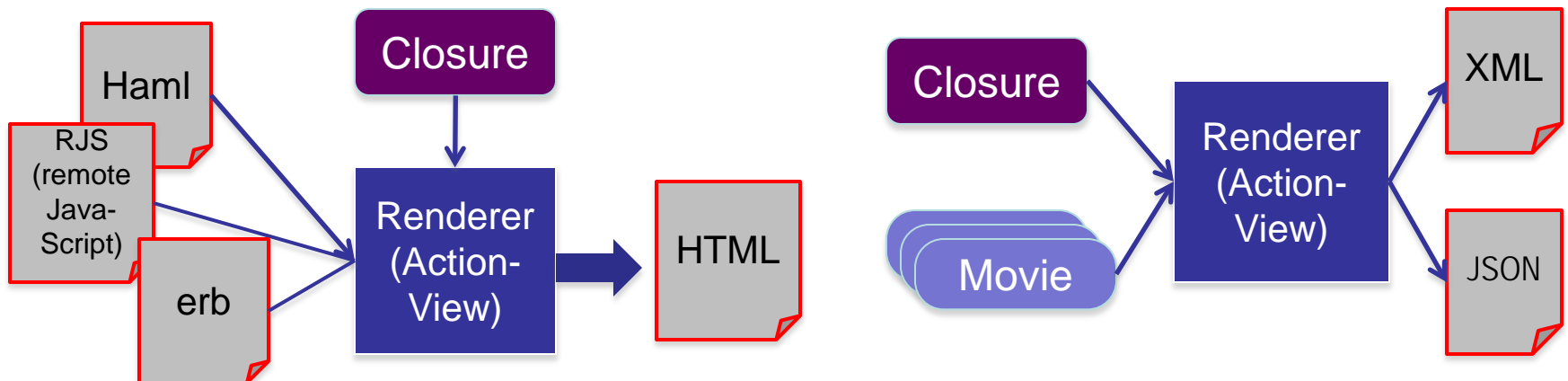
```
%h1.pagename All Movies
%table#movies
  %thead
    %tr
      %th Movie Title
      %th Release Date
      %th More Info
  %tbody
    - @movies.each do |movie|
      %tr
        %td= movie.title
        %td= movie.release_date
        %td= link_to "More on #{movie.title}",  |
          movie_path(movie) |
= link_to 'Add new movie', new_movie_path
```

Don't put code in your views

- Syntactically, you can put any code in view
- But MVC advocates thin views & controllers
  - Haml makes deliberately awkward to put in lots of code
- *Helpers* (methods that "prettify" objects for including in views) have their own place in Rails app
- Alternative to Haml: html.erb (Embedded Ruby) templates, look more like PHP

What happens if you embed code in your Rails views that directly accesses the model?

☐ It will work, but it's bad form and violates the MVC guidelines

☐ It will work when developing against a "toy" database, but not in production

☐ It won't work, because Views can't communicate directly with Models

☐ Behavior varies depending on the app

Summary & Reflections: SaaS Architecture

Armando Fox

The big picture (technologies)

- URI's, HTTP, TCP/IP stack
- REST & RESTful routes

Controller

View

Model
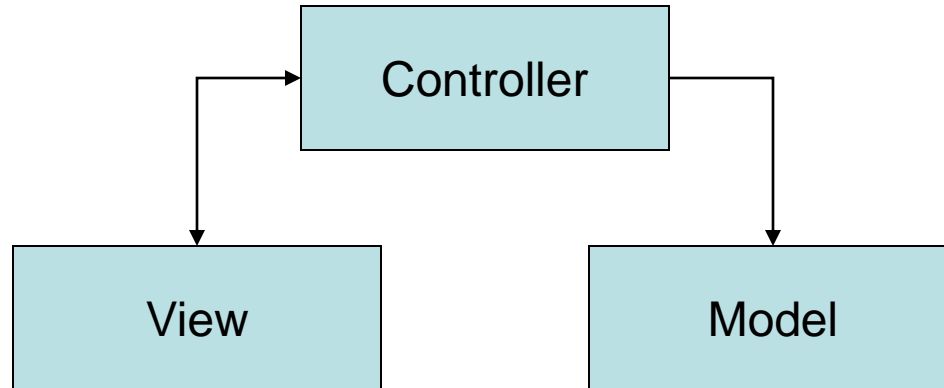
- HTML & CSS
- XML & XPath

- Databases & migrations
- CRUD

c. 2008: "Rails doesn't scale"

- Scalability is an *architectural* concern—not confined to language or framework
- The stateless tiers of 3-tier arch *do scale*
  - With cloud computing, just worry about constants
- Traditional <u>relational</u> databases *do not scale*
- Various solutions combining relational and non-relational storage ("NoSQL") *scale much better*
  - DataMapper works well with some of them
- Intelligent use of *caching* (later in course) can greatly improve the constant factors

Frameworks, Apps, Design patterns

- Many design patterns so far, more to come
- *In 1995, it was the wild west:* biggest Web sites were minicomputers, *not* 3-tier/cloud
- Best practices (patterns) "extracted" from experience and captured in frameworks
- But API's transcended it: 1969 protocols + 1960s markup language + 1990 browser + 1992 Web server works in 2011

Architecture is about Alternatives

Pattern we're using	Alternatives
Client-Server	Peer-to-Peer
Shared-nothing (cloud computing)	Symmetric multiprocessor, shared global address space
Model-View-Controller	Page controller, Front controller, Template view
Active Record	Data Mapper
RESTful URIs (all state affecting request is explicit)	Same URI does different things depending on internal state

As you work on other SaaS apps beyond this course, you should find yourself considering different architectural choices and questioning the choices being made.

Summary: Architecture & Rails

- Model-view-controller is a well known *architectural pattern* for structuring apps

- Rails codifies SaaS app structure as MVC

- *Views* are Haml w/embedded Ruby code, transformed to HTML when sent to browser

- *Models* are stored in tables of a relational database, accessed using ActiveRecord

- *Controllers* tie views and models together via *routes* and code in controller methods

Other factors being equal, which statement is **NOT** true regarding SaaS scalability?

☐ Shared-nothing clusters scale better than systems built from mainframes

☐ Relational databases scale better than "NoSQL" databases

☐ The programming language used (Ruby, Java, etc.) isn't a main factor in scalability

☐ Scalability can be impeded by *any* part of the app that becomes a bottleneck