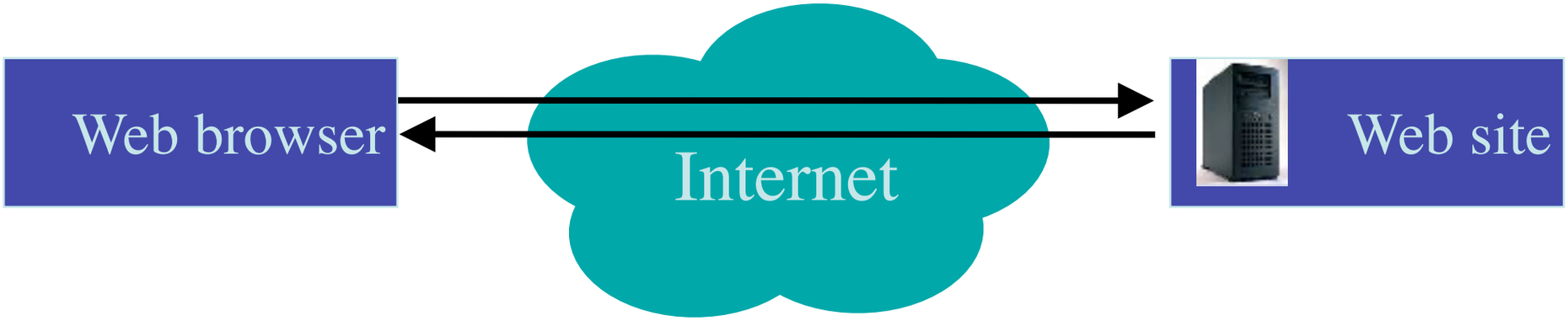
The Web as a Client-Server System; TCP/IP intro

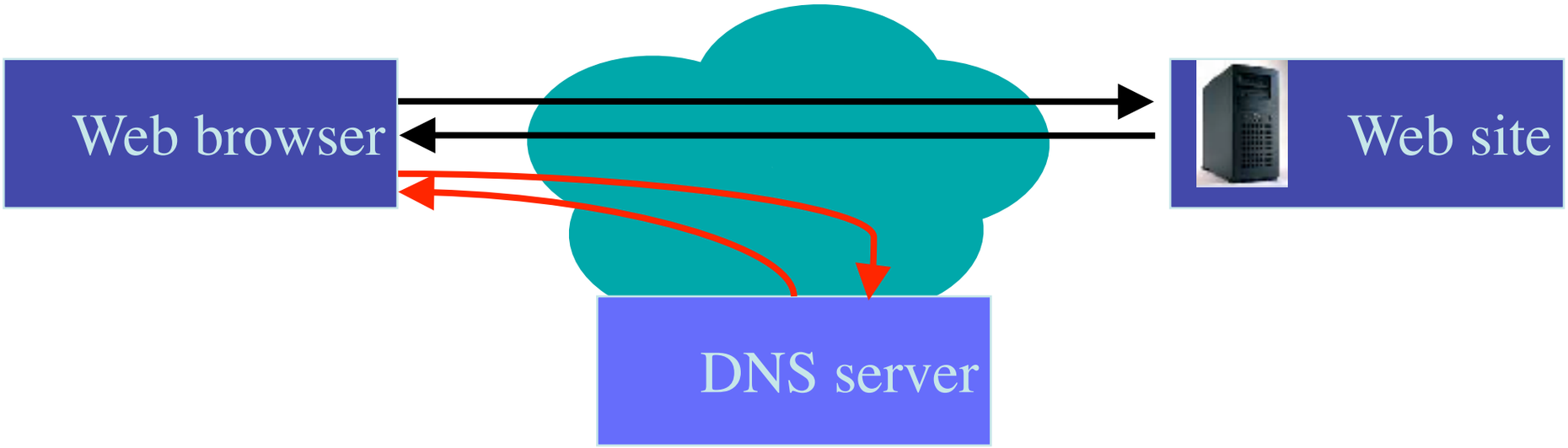
Web at 100,000 feet

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



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*

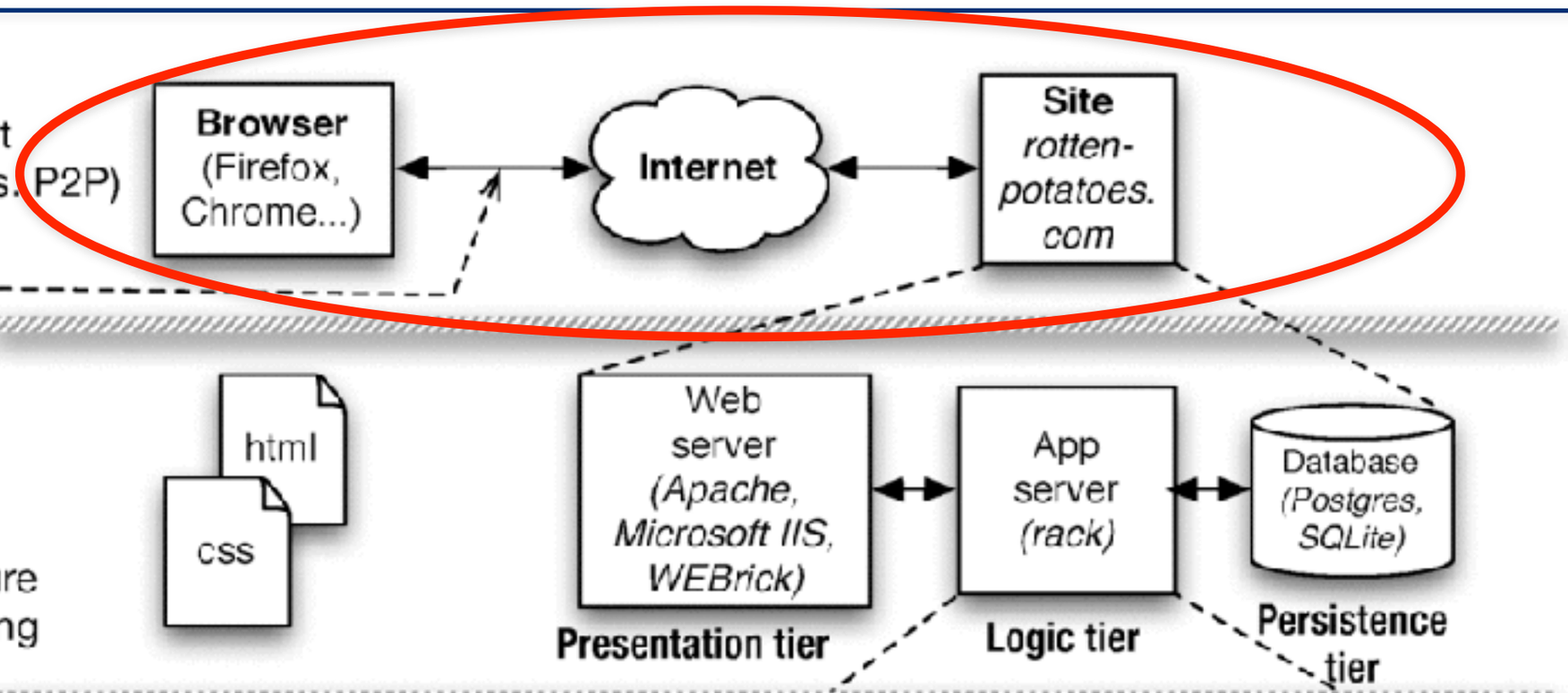


§2.1 100,000 feet
- Client-server (vs. P2P)

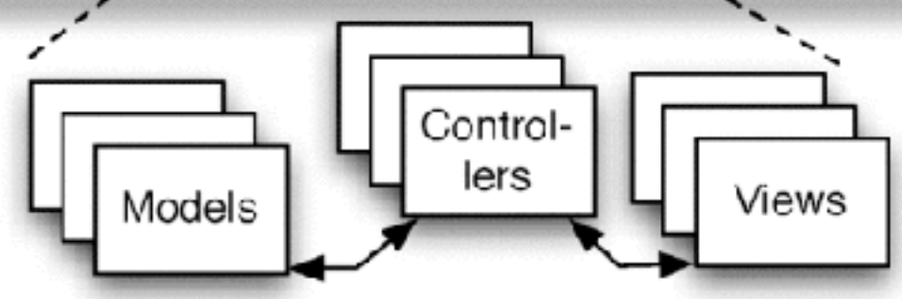
§2.2 50,000 feet
- HTTP & URIs

§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*

Try This Out

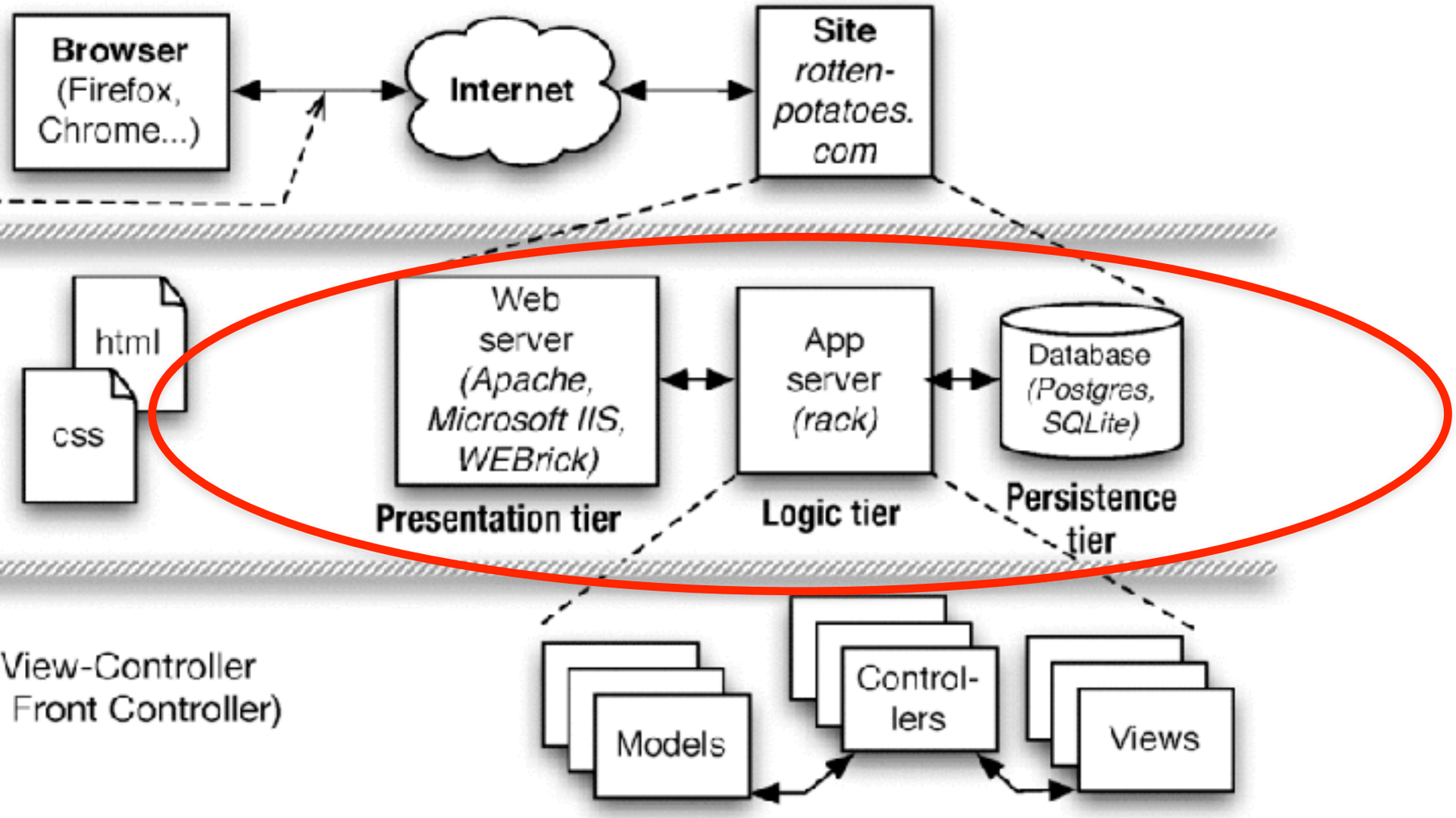
- In a terminal: nc –l 8000 (listen on port 8000)
- In web browser: localhost:8000/la/di/da
- Back to terminal: See what the browser got back- URI it wanted, protocol being used, some cookies (next week), now waiting- you are now playing the web server.
- In the terminal, type “Hello World”
- Back to browser: “Hello World” will appear

§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

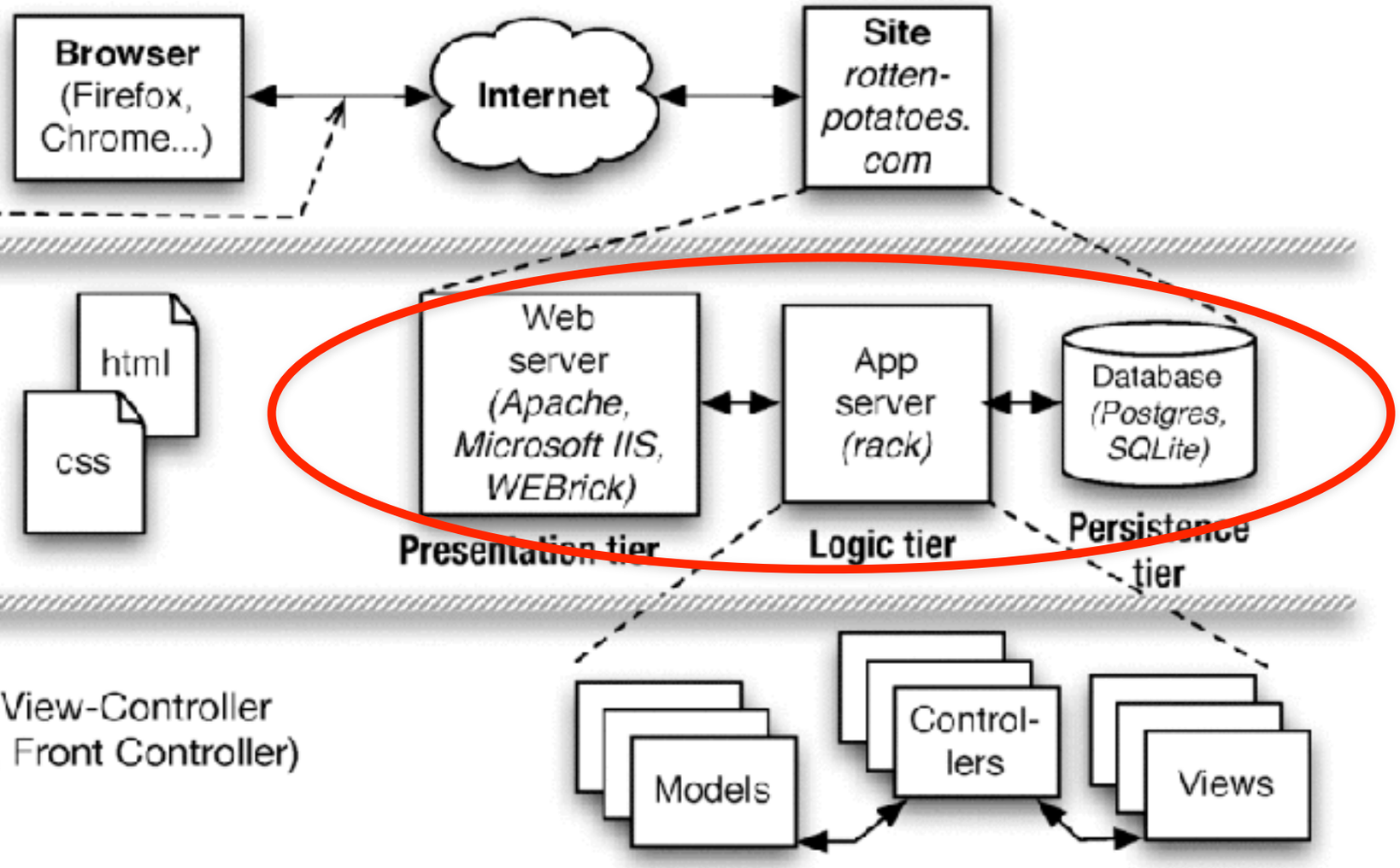
3-tier shared-nothing architecture & scaling

§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)
- **Active Record**
- Data Mapper

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
- **REST**

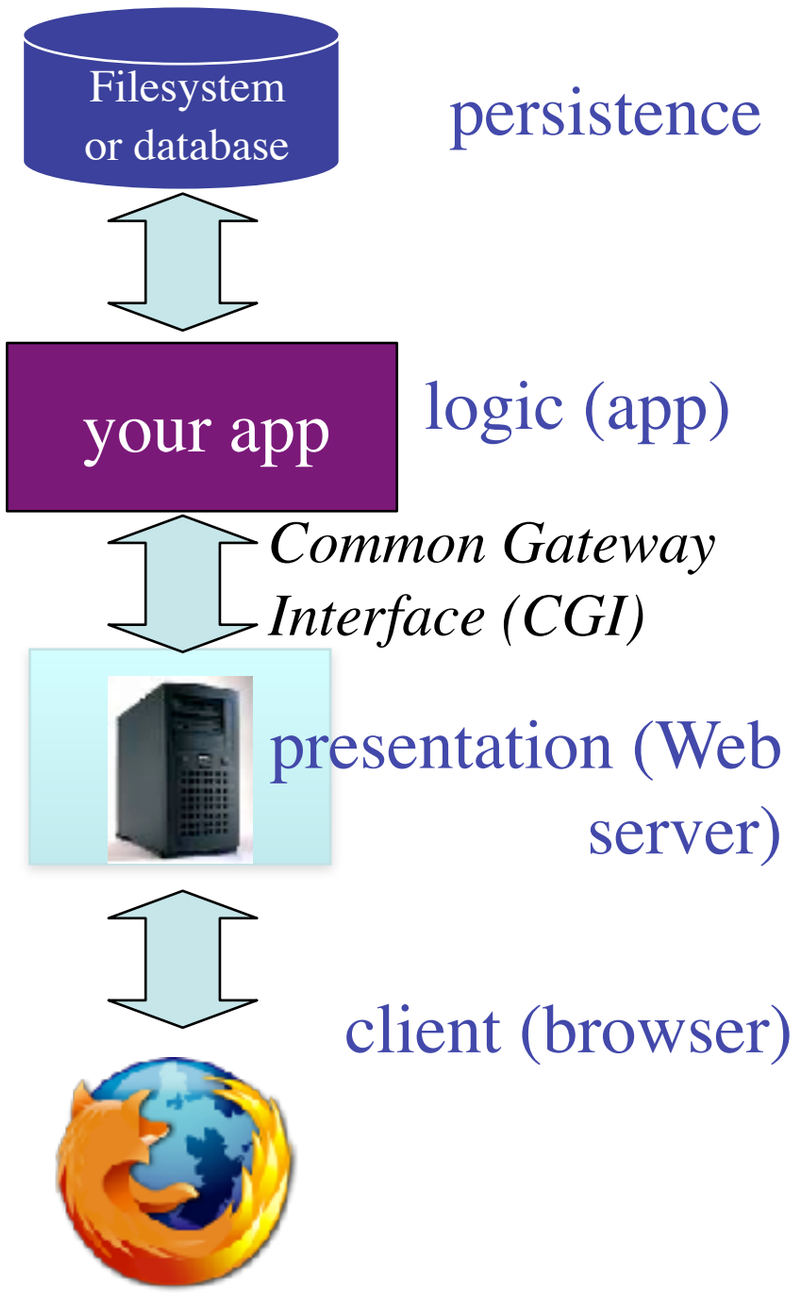
§2.8 500 feet: Template View (vs. Transform View)
- **Template View**
- Transform View

Dynamic content generation

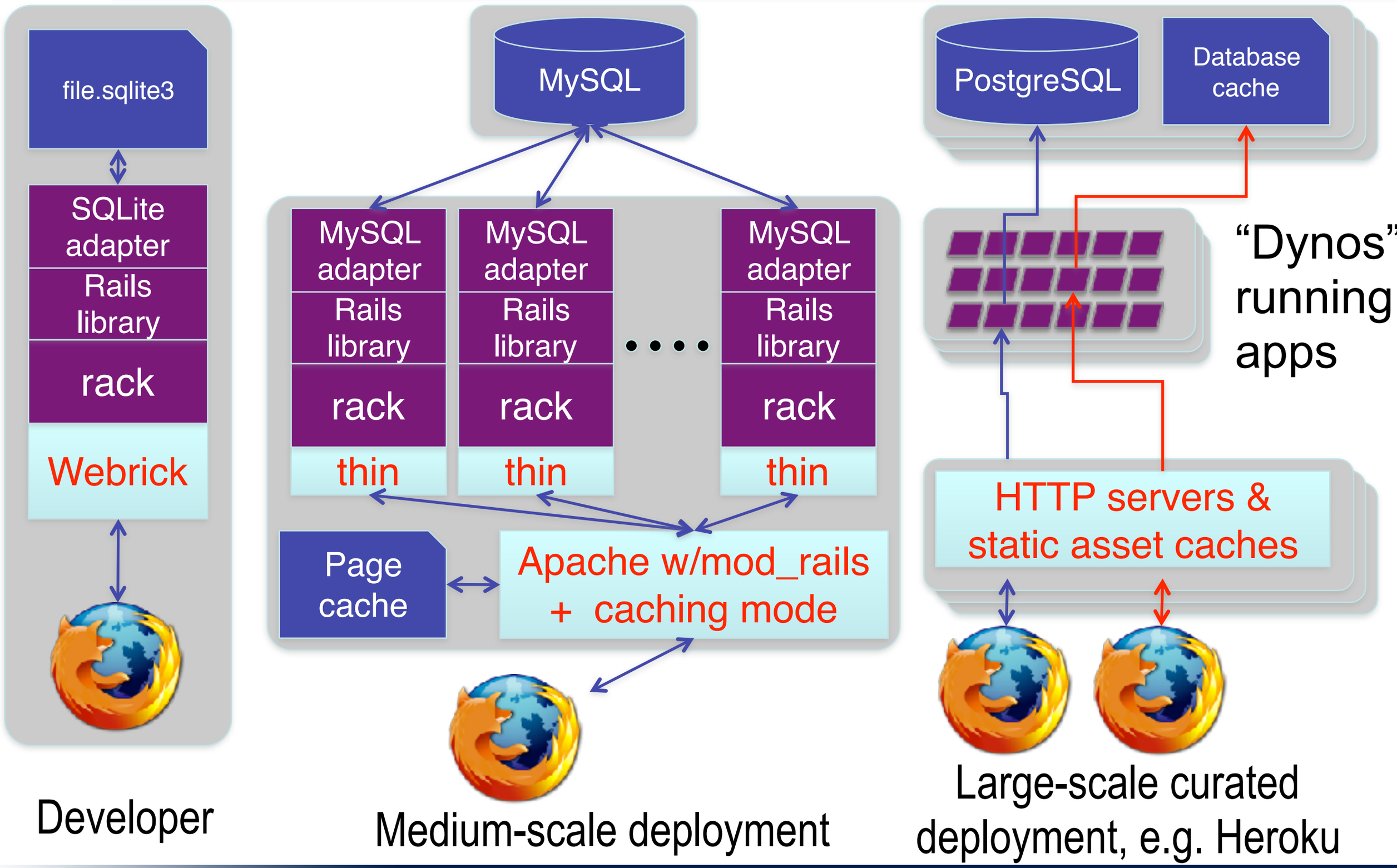
- In the Elder Days, most web pages were (collections of) plain old files
- But most interesting Web 1.0/e-commerce sites actually *run a program* to generate the "page"
- Originally: templates with embedded code "snippets"
- Eventually, code became "tail that wagged the dog" and moved out of the Web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

Sites that are really programs

- 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

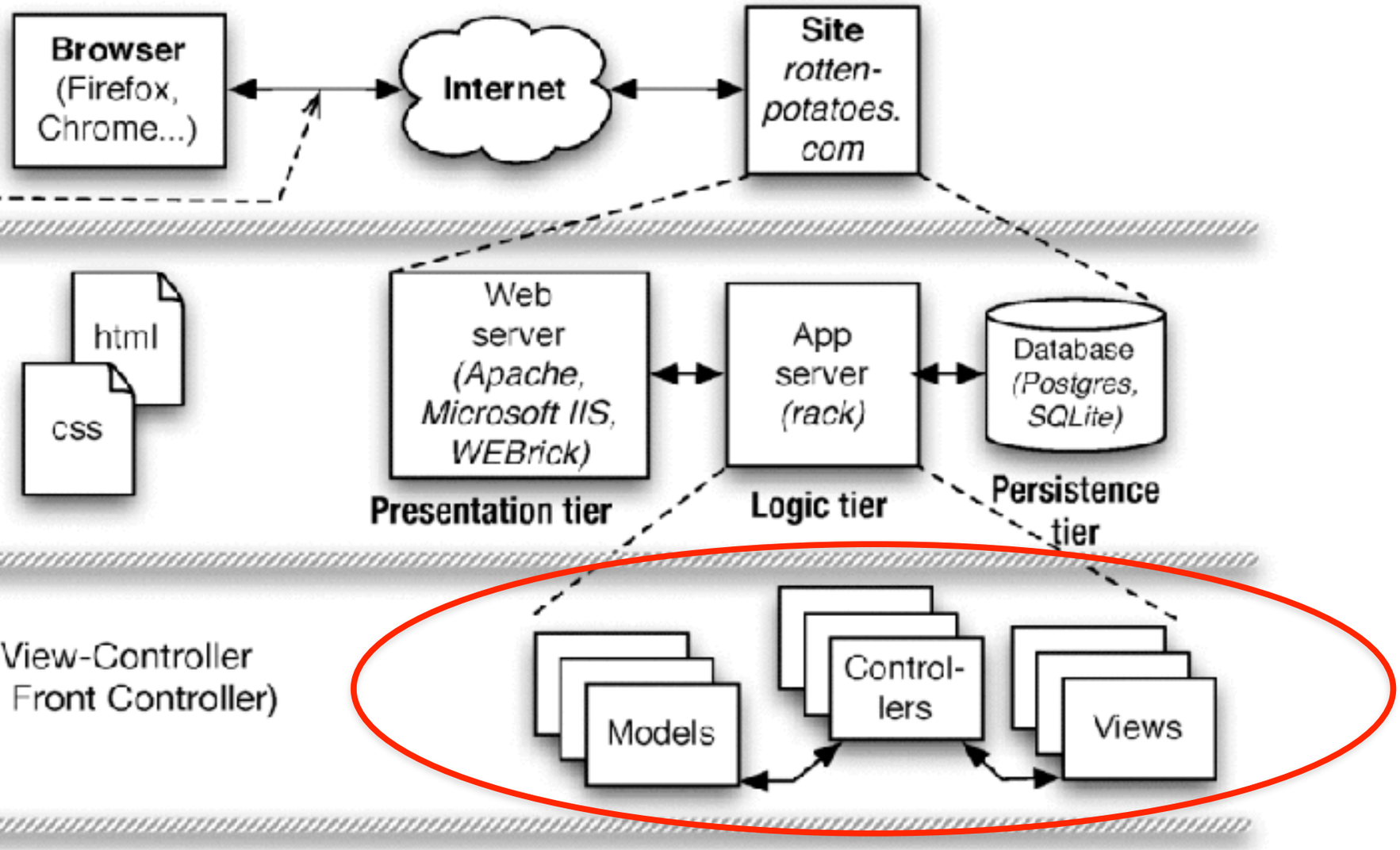


§2.1 100,000 feet
- Client-server (vs. P2P)

§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)

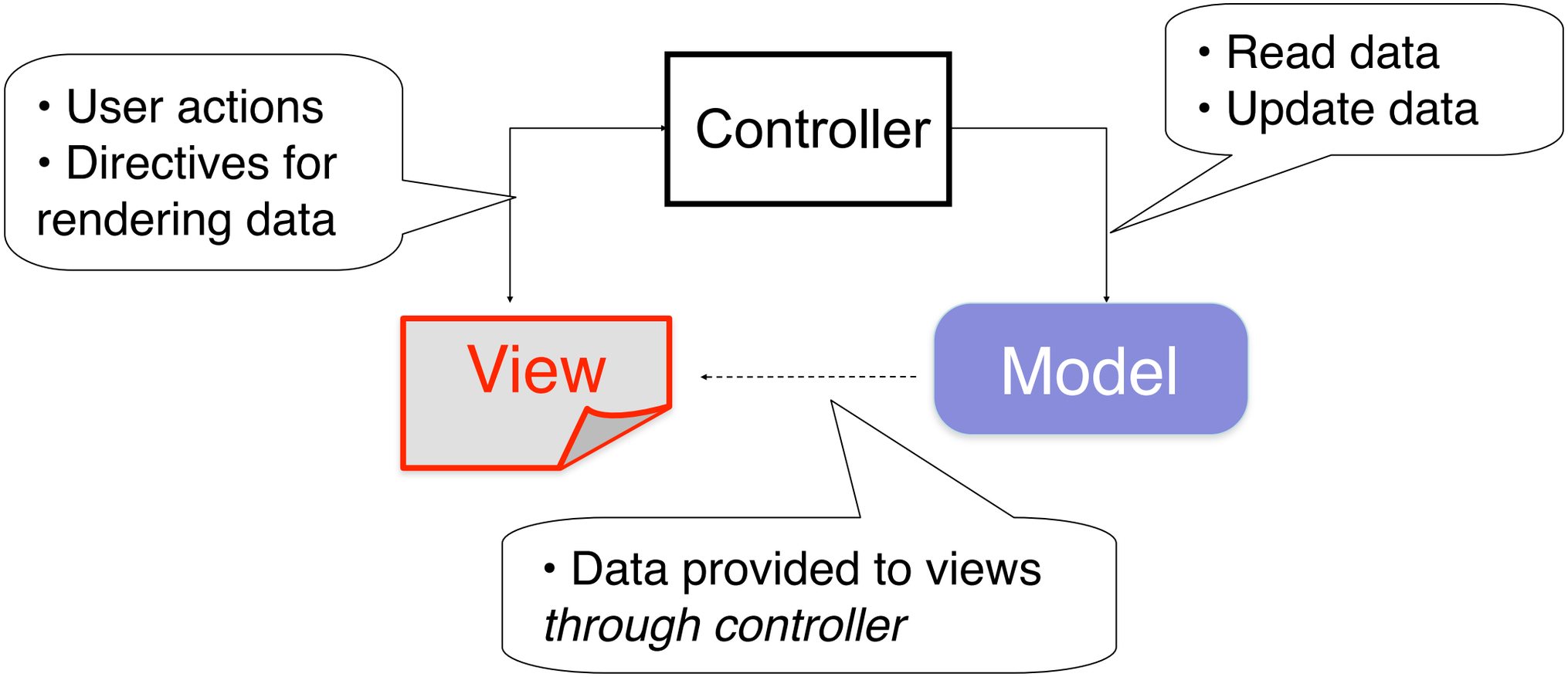
§2.6 500 feet: Active Record models (vs. Data Mapper)
- **Active Record**
- Data Mapper

§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
- **REST**

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

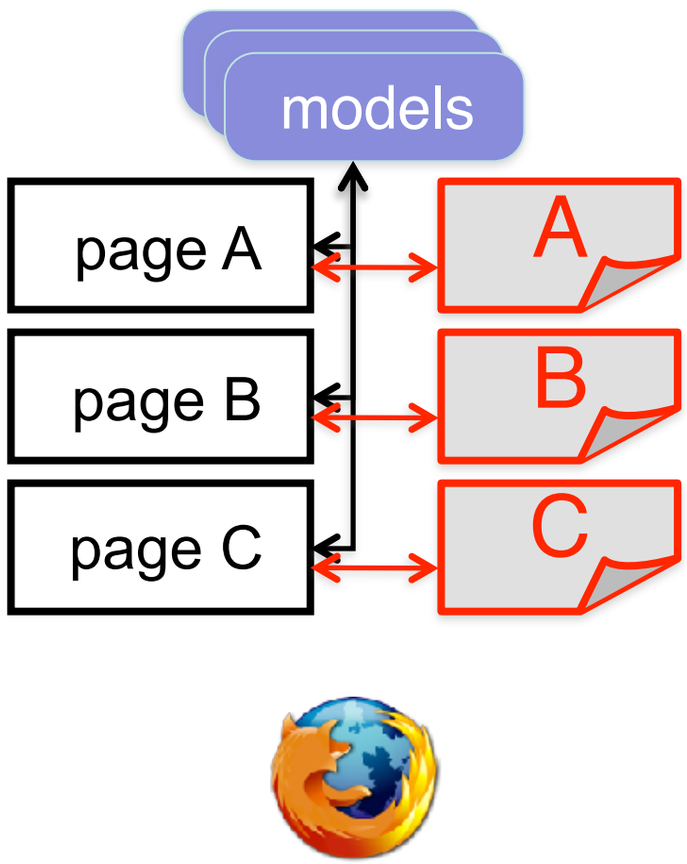
The MVC Design Pattern

- 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...

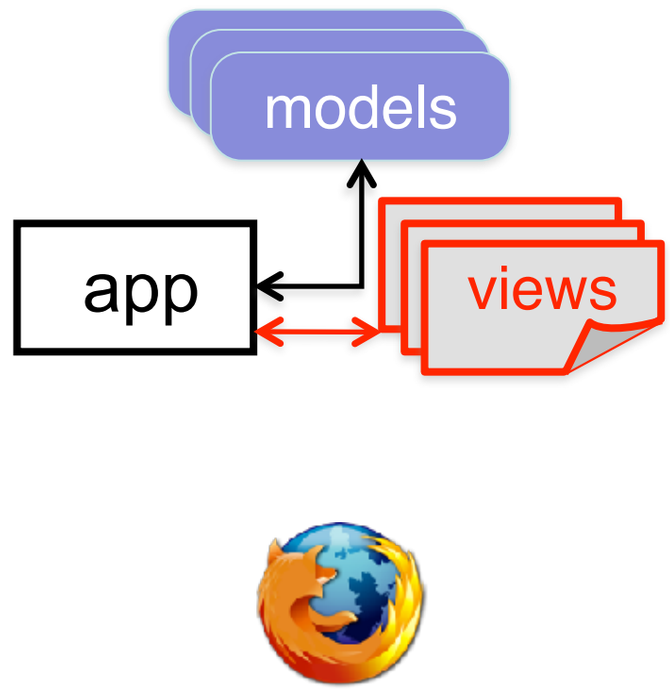


Alternatives to MVC

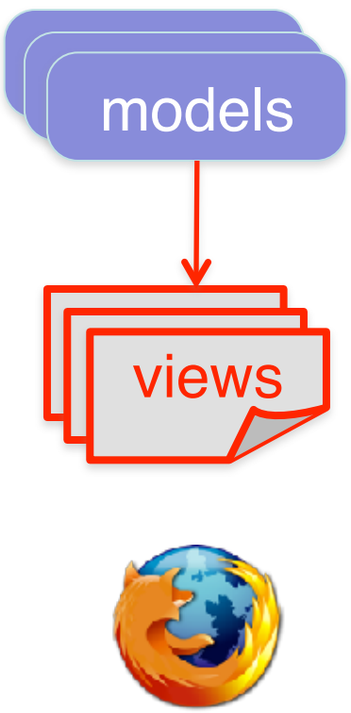
Page Controller
(Ruby Sinatra)



Front Controller
(J2EE servlet)



Template View
(PHP)



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

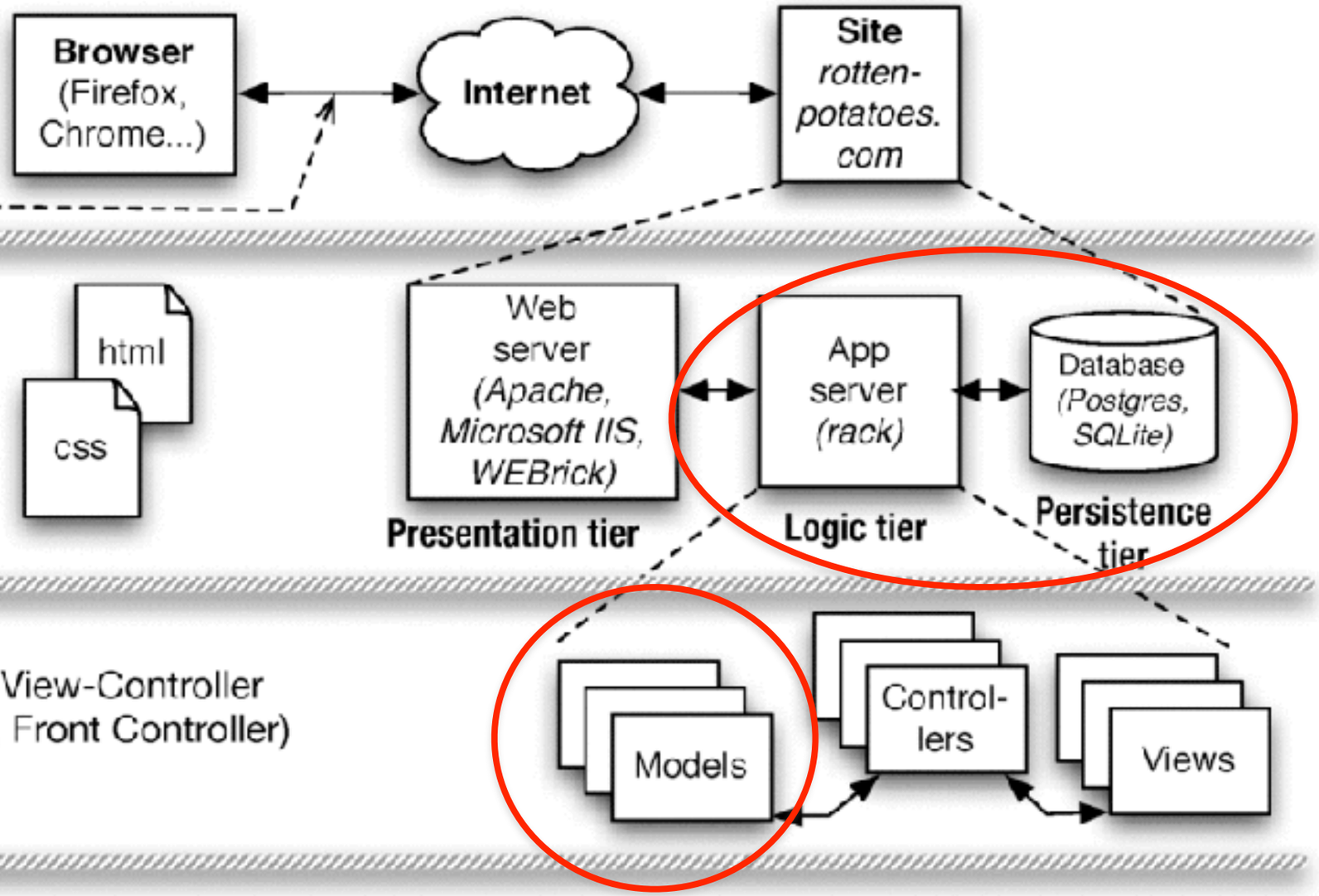
Models, Databases, and Active Record

§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)
- **Active Record**
- Data Mapper

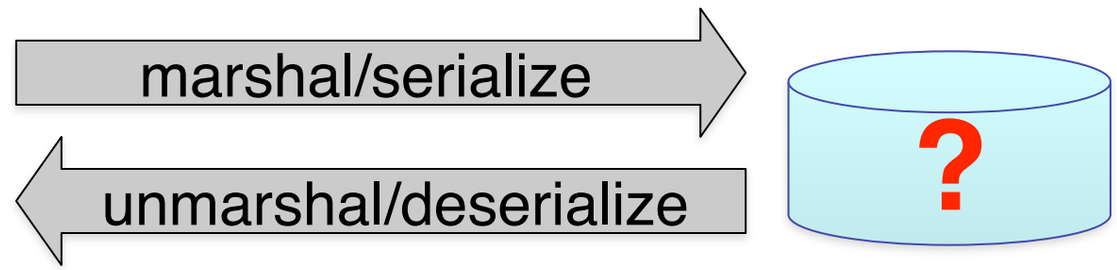
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
- **REST**

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

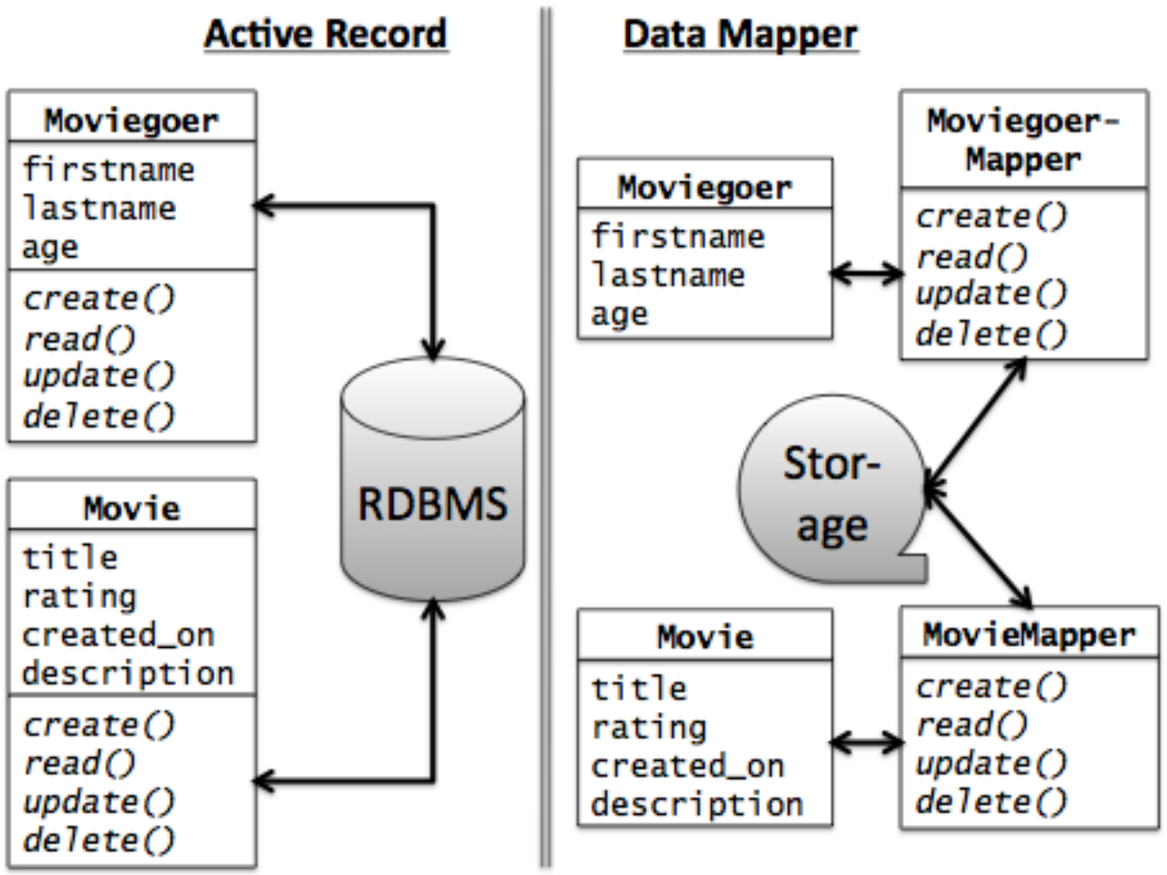
- Each type of model gets its own database *table*
  - All rows in table have identical structure
  - 1 row in table == one model instance
  - Each column stores value of an *attribute* of the model
  - Each row has unique value for *primary key* (by convention, in Rails this is an integer and is called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



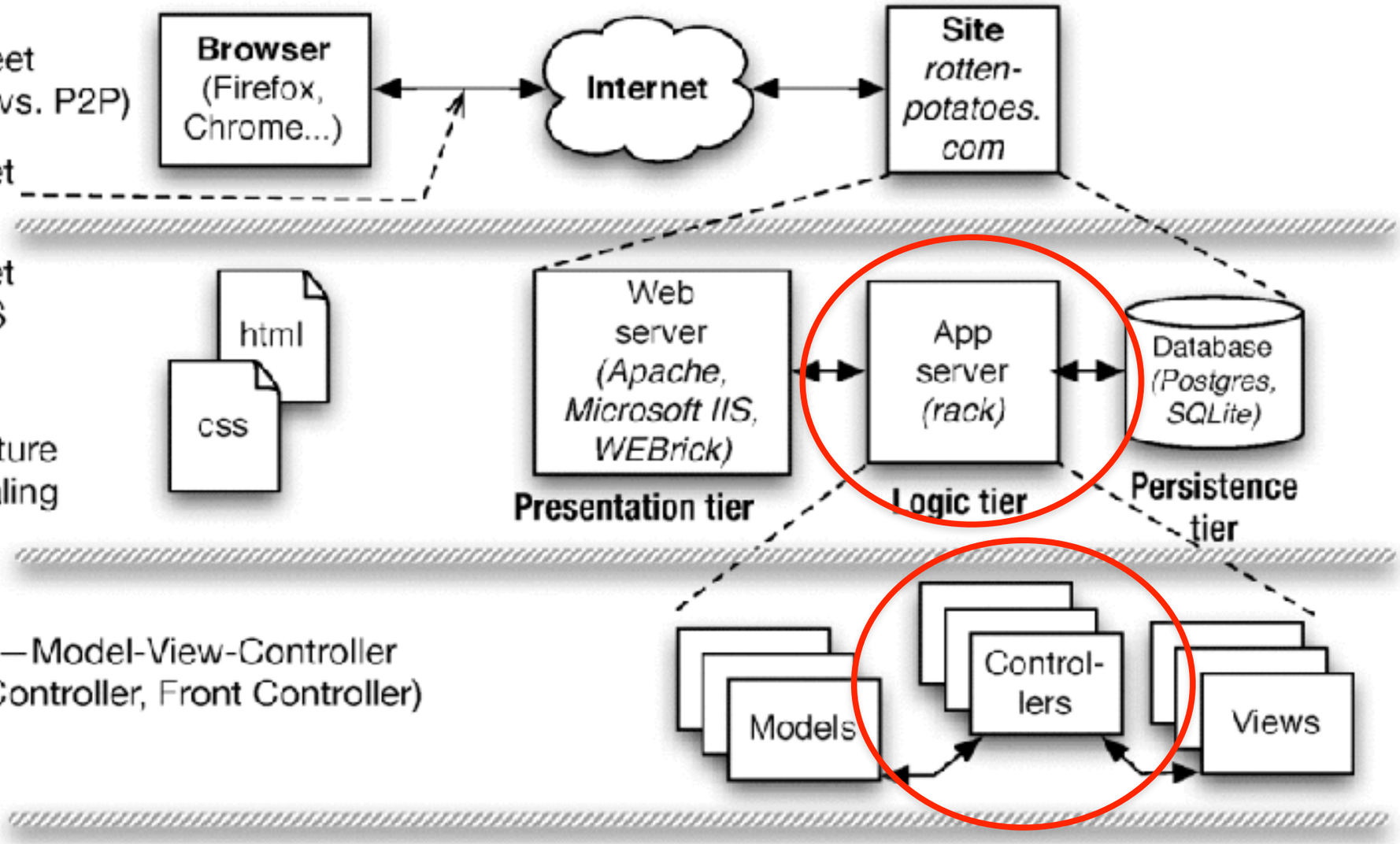
Controllers, Routes, and RESTfulness

§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

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"}

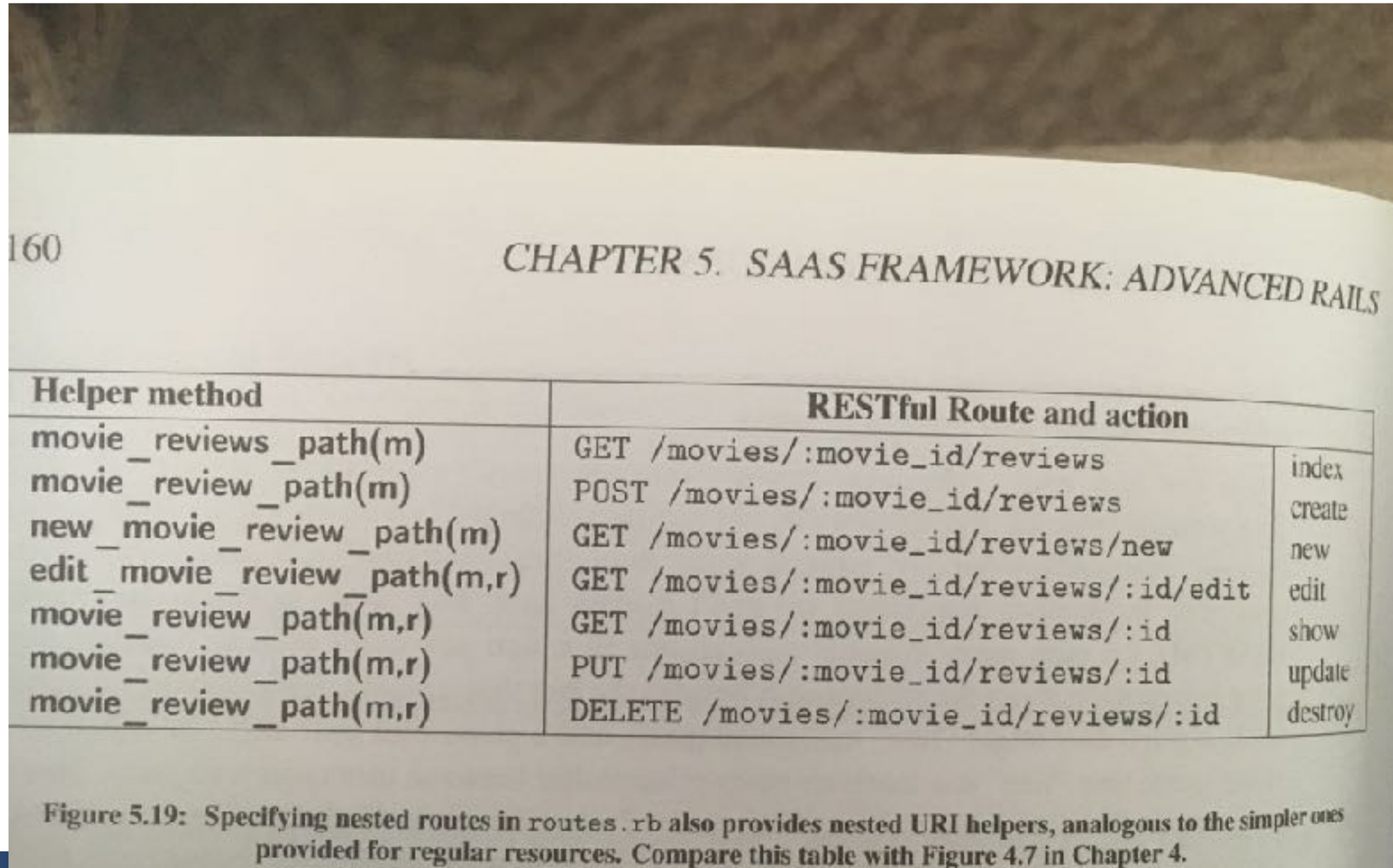
- Parse wildcard parameters: params[:id] = "3"
- Dispatch to edit method in movies_controller.rb
- To include a URI in generated view that will submit the form to the update controller action with params[:id]==3, call helper:
  update_movie_path(3) *# => PUT /movies/3*

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"}
```

Other available routes

- Convention over configuration
  - Automatically generated when you migrate (create) models



160 CHAPTER 5. SAAS FRAMEWORK: ADVANCED RAILS

Helper method	RESTful Route and action	
movie_reviews_path(m)	GET /movies/:movie_id/reviews	index
movie_review_path(m)	POST /movies/:movie_id/reviews	create
new_movie_review_path(m)	GET /movies/:movie_id/reviews/new	new
edit_movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id/edit	edit
movie_review_path(m,r)	GET /movies/:movie_id/reviews/:id	show
movie_review_path(m,r)	PUT /movies/:movie_id/reviews/:id	update
movie_review_path(m,r)	DELETE /movies/:movie_id/reviews/:id	destroy

Figure 5.19: Specifying nested routes in `routes.rb` also provides nested URI helpers, analogous to the simpler ones provided for regular resources. Compare this table with Figure 4.7 in Chapter 4.

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

Not RESTful

http://pastebin.com/edF2NzCF

```
def get_kindle_sales(cs_user,cs_pass)
  session = Mechanize.new
  session.user_agent_alias = 'Mac Safari'
  session.get 'https://www.amazon.com/ap/signin?
openid.assoc_handle=amzn_dtp&openid.identity=' #...etc.
  form = session.get('https://www.amazon.com/ap/signin?
openid.assoc_handle=amzn_dtp&openid.=' + # ...etc.
     '...').form_with(:name => 'signIn')
  params = {'email' => cs_user,  'password' => cs_pass}
  %w(appActionToken appAction openid.pape.max_auth_age openid.ns).each do |field| #
there's more, actually
    params[field] = form[field]
  end
  session.post('https://www.amazon.com/ap/signin', params)
  response = session.get('https://kdp.amazon.com/self-publishing/reports/transactionReport?
_=1326589411161&previousMonthReports=false&marketplaceID=ATVPDKIKX0DER')
  # note non-RESTful concept of "previousMonthReports" in URI
  hash = JSON.parse(response.body)
  kindle_units = hash['aaData'][0][5]
end
```

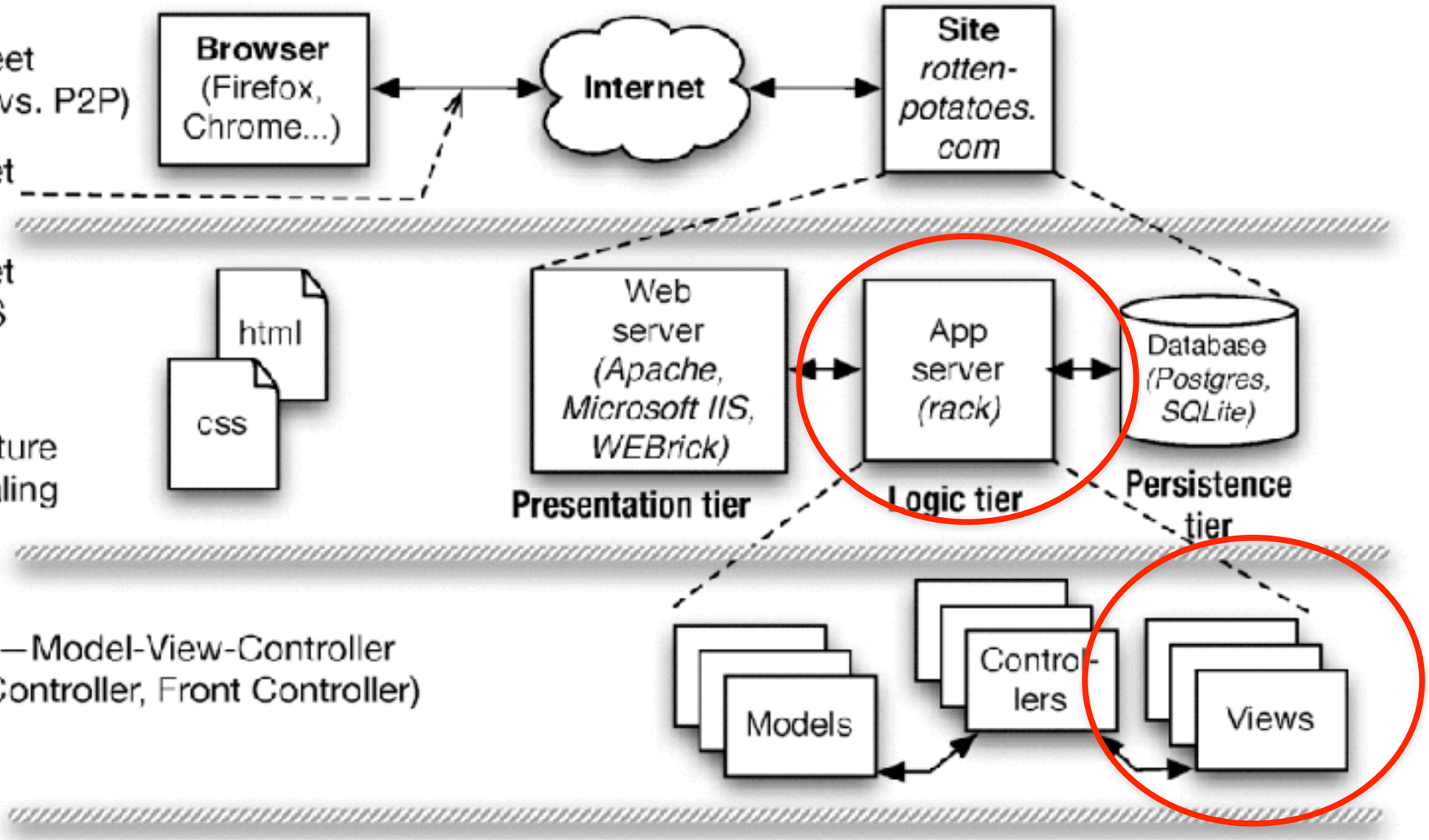
Template Views and Haml

§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



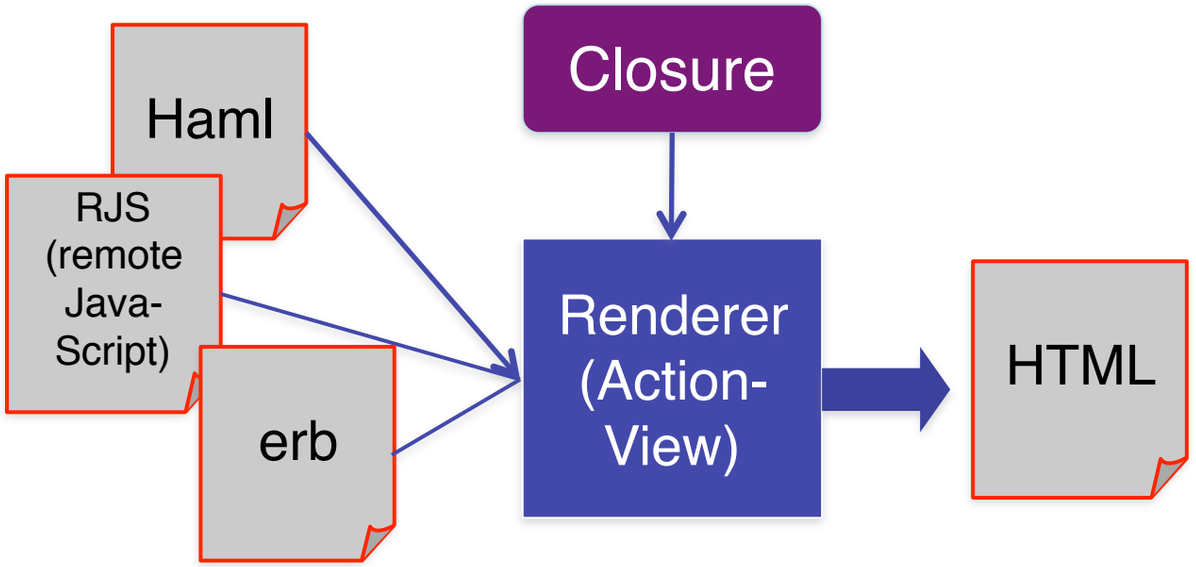
§2.5 1,000 feet—Model-View-Controller
(vs. Page Controller, Front Controller)

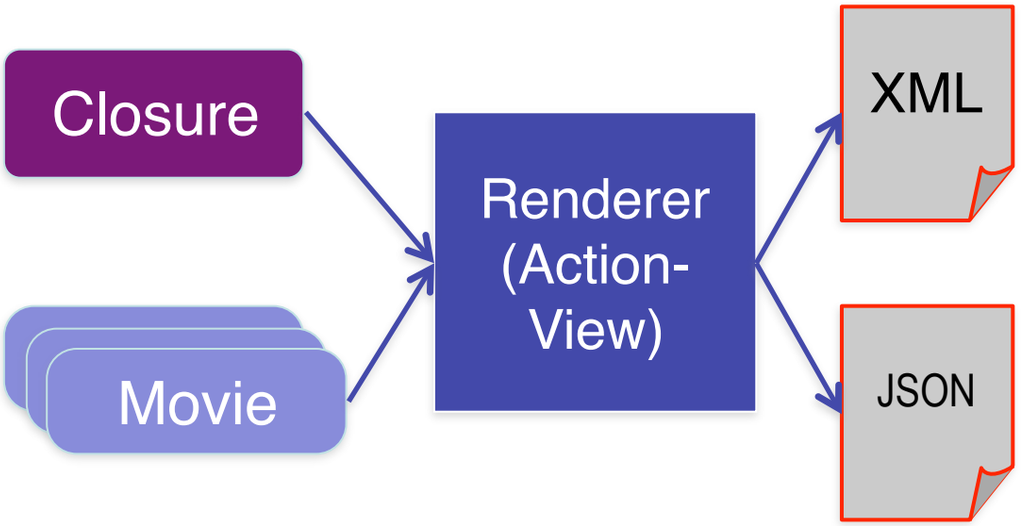
§2.6 500 feet: Active Record models (vs. Data Mapper)
§2.7 500 feet: RESTful controllers (Representational State Transfer for self-contained actions)
§2.8 500 feet: Template View (vs. Transform View)

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

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





Haml is HTML on a diet

```
%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
```

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.

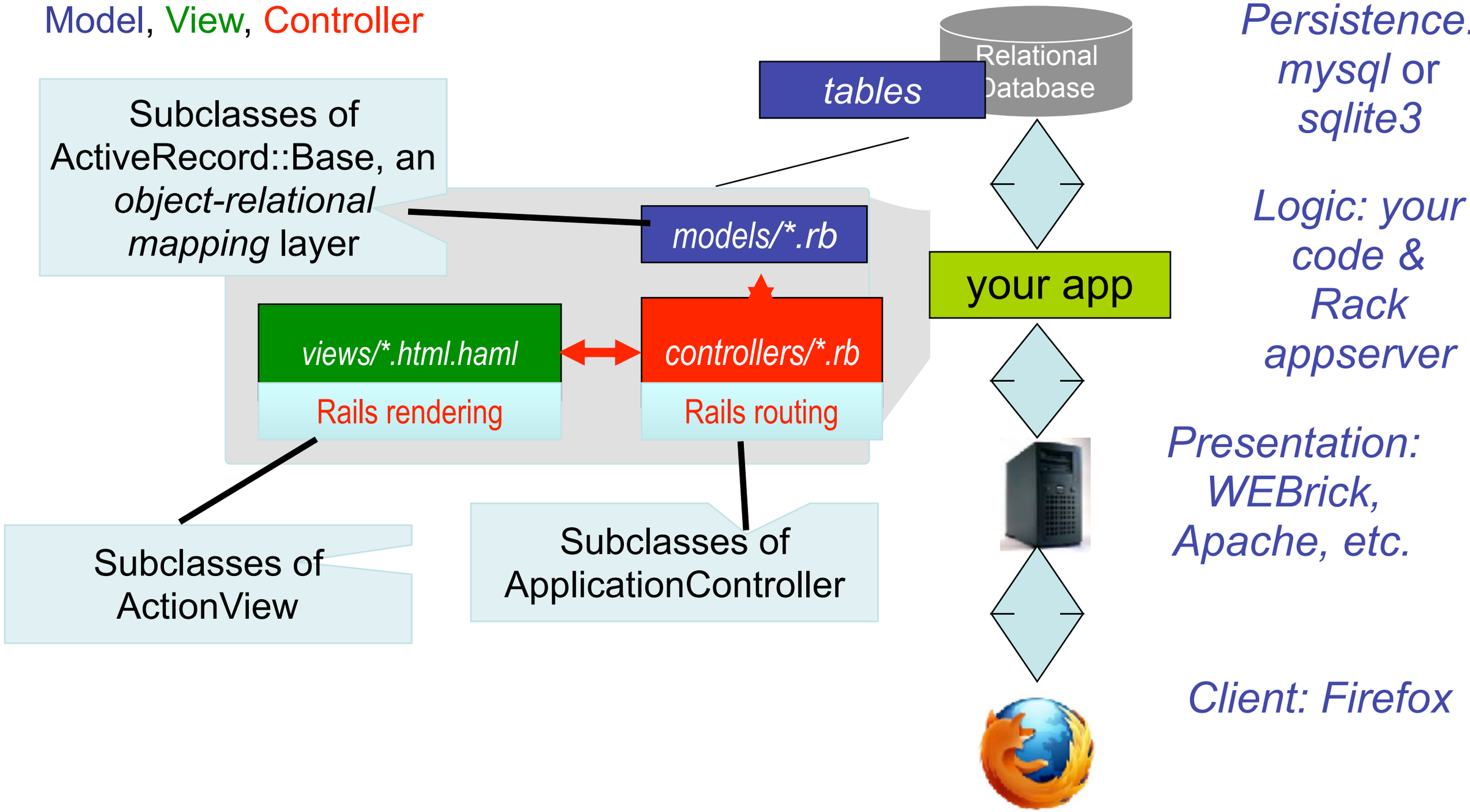
Don’t put code in your views

- 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

Rails from Zero to CRUD

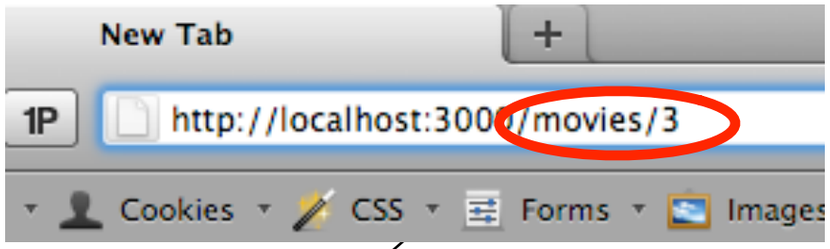
Rails as an MVC Framework

Model, View, Controller



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'}
```

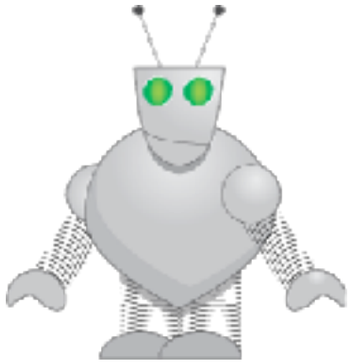
Databases & Migrations

The Database is Golden

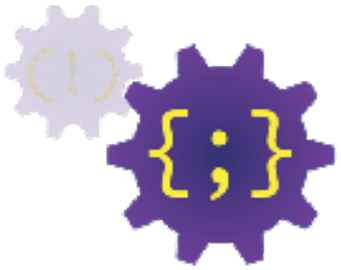
- Contains valuable customer data—don't want to test your app on that!
- Rails solution: development, production and test *environments* each have own DB
    - Different DB types appropriate for each
- How to make *changes* to DB, since will have to repeat changes on production DB?
- Rails solution: *migration*—script describing changes, portable across DB types

Migration Advantages

- Can identify each migration, and know which one(s) applied and when
  - Many migrations can be created to be *reversible*
- Can manage with version control
- *Automated == reliably repeatable*
  - Compare: use Bundler vs. manually install libraries/gems
- Theme: *don't do it—automate it*
  - *specify* what to do, create tools to automate



Meet a Code Generator



rails generate migration CreateMovies

- Note, this just *creates* the migration. We haven't *applied* it.

http://pastebin.com/VYwbc5fq

- Apply migration to development: **rake db:migrate**
- Apply migration to production: **heroku rake db:migrate**
- Applying migration also records in DB itself which migrations have been applied

Example

```
class CreateMovies < ActiveRecord::Migration
  def up
    create_table 'movies' do |t|
      t.string 'title'
      t.string 'rating'
      t.text 'description'
      t.datetime 'release_date'
      # Add fields that let Rails automatically keep track
      # of when movies are added or modified:
      t.timestamps
    end
  end
  def down
    drop_table 'movies' # deletes the whole table and all its data!
  end
end
```

Rails Cookery #1

- Augmenting app functionality == adding models, views, controller actions

To *add a new model* to a Rails app:

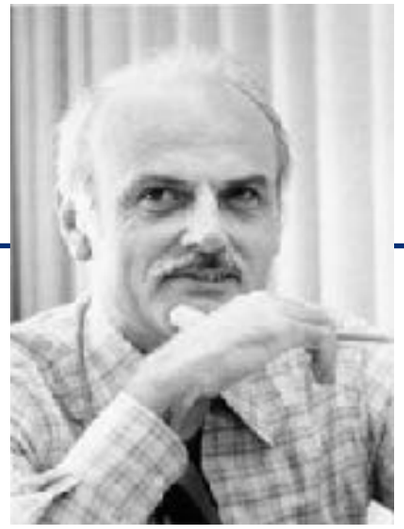
(or change/add attributes of an existing model)

1. Create a migration describing the changes: `rails generate migration` (gives you boilerplate)
2. Apply the migration: `rake db:migrate`
3. If new model, create model file `app/models/`*model*`.rb`
   - Update test DB schema: `rake db:test:prepare`

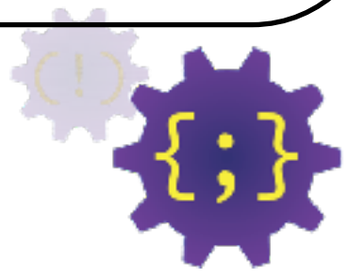
Models: ActiveRecord Basics

CRUD in SQL

"Ted" Codd



- Structured Query Language (SQL) is the query language used by RDBMS's



- Rails *generates* SQL statements at runtime, based on your Ruby code
- 4 basic operations on a table row:
  Create, **R**ead, **U**pdate attributes, **D**elete

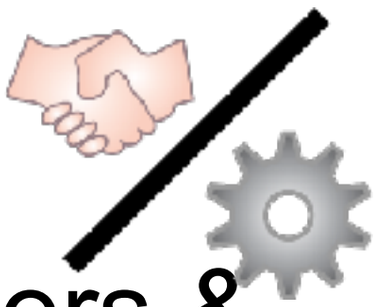
```
INSERT INTO users
(username, email, birthdate) VALUES ("kjustice",
"kjustice@uccs.edu", "????"),
     "randper", "randper@uccs", "1987-01-15")
SELECT * FROM users WHERE (birthdate BETWEEN "1987-01-01" AND
“2000-01-01”)
UPDATE users SET email = "kristen.justice@gmail.com"WHERE
username="kjustice"
DELETE FROM users WHERE id=1
```

The Ruby side of a model

- Subclassing from ActiveRecord::Base
- "connects" a model to the database
- provides CRUD operations on the model

http://pastebin.com/ruu5y0D8

- Database table name derived from model's name: **M**ovie→**m**ovie**s**



- Database table column names are getters & setters for model attributes
- *Observe: the getters and setters do not simply modify instance variables!*

Active Record Example

```
1. class Movie < ActiveRecord::Base
2. end
3. # 3 ways to create ActiveRecord objects
4. # (the constructor checks to see what arguments it got)
5. movie = Movie.new
6. movie.title = 'The Help'
7. movie.rating = 'PG-13'
8.
9. movie = Movie.new do |m|
10.   m.title = 'The Help'
11.   m.rating = 'PG-13'
12. end
13.
14. movie = Movie.new(:title => 'The Help', :rating => 'PG-13')
```

Creating: new ≠ save

- Must call save or save! on an AR model instance to actually save changes to DB
  - '!' version is “dangerous”: throws exception if operation fails
  - create just combines new and save
- Once created, object acquires a primary key (id column in every AR model table)
  - if x.id is nil or x.new_record? is true, x has never been saved
  - These behaviors inherited from ActiveRecord::Base—not true of Ruby objects in general

Read: finding things in DB

- class method where selects objects based on attributes

```
Movie.where("rating='PG'")
Movie.where('release_date < :cutoff and
   rating = :rating',
   :rating => 'PG', :cutoff => 1.year.ago)
Movie.where("rating=#{rating}") # BAD IDEA!
```

- Can be chained together efficiently

```
kiddie = Movie.where("rating='G'")


old_kids_films =
 kiddie.where "release_date < ?", 30.years.ago
```

Read: find_*

- find by id: Movie.find(3) #*exception if not found*
- Movie.find_by_id(3) # *nil if not found*

- dynamic attribute-based finders using

Movie.find_all_by_rating('PG')
Movie.find_by_rating('PG')
Movie.find_by_rating!('PG')

Update: 2 ways

- Modify attributes, then save object

```
m=Movie.find_by_title('The Help')
m.release_date='2011-Aug-10'
m.save!
```

- Update attributes on existing object

```
Movie.find_by_title('The Help').
  update_attributes!(
    :release_date => '2011-Aug-10'
  )
```

- Transactional: either all attributes are updated, or none are

Deleting is straightforward

- Note! destroy is an *instance* method

```
m = Movie.find_by_name('The Help')
m.destroy
```

- There's also delete, which doesn't trigger *lifecycle callbacks* we'll discuss later (so, avoid it)
- Once an AR object is destroyed, you can access *but not modify* in-memory object

```
m.title = 'Help'  # FAILS
```

Controllers & Views

Rails Cookery #2

- To *add a new action* to a Rails app

1. Create *route* in **config/routes.rb** if needed
2. Add the *action* (method) in the appropriate **app/controllers/*_controller.rb**
3. Ensure there is something for the action to *render* in **app/views/***model***/***action***.html.haml**

1. We'll do Show action & view (book walks through Index action & view)

MVC responsibilities

- *Model:* methods to get/manipulate data

Movie.where(...), Movie.find(...)

- *Controller:* get data from Model, make available to View

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

Instance variables set in Controller available in View

Absent other info, Rails will look for app/views/movies/show.html.haml

- *View:* display data, allow user interaction
- Show details of a movie (description, rating)
- But…
- What else can user do from this page?
- How does user get to this page?

http://pastebin.com/kZCB3uNj

How we got here: URI helpers

Helper method	URI returned	RESTful Route and action	
movies_path	/movies	GET /movies	index
movies_path	/movies	POST /movies	create
new_movie_path	/movies/new	GET /movies/new	new
edit_movie_path(m)	/movies/1/edit	GET /movies/:id/edit	edit
movie_path(m)	/movies/1	GET /movies/:id	show
movie_path(m)	/movies/1	PUT /movies/:id	update
movie_path(m)	/movies/1	DELETE /movies/:id	destroy

link_to movie_path(3)

index.
html.
haml

...

GET /movies/:id
params[:id]←3

{:action=>"show", :controller=>"movies"}

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

What else can we do?

- How about letting user return to movie list?
- RESTful URI helper to the rescue again:
- movies_path with no arguments links to Index action

=link_to 'Back to List', movies_path

Helper method	URI returned	RESTful Route and action	
movies_path	/movies	GET /movies	index
movies_path	/movies	POST /movies	create
new_movie_path	/movies/new	GET /movies/new	new
edit_movie_path(m)	/movies/1/edit	GET /movies/:id/edit	edit
movie_path(m)	/movies/1	GET /movies/:id	show
movie_path(m)	/movies/1	PUT /movies/:id	update
movie_path(m)	/movies/1	DELETE /movies/:id	destroy

When things go wrong: Debugging

Debugging SaaS can be tricky

- "Terminal" (STDERR) not always available
- Errors early in flow may manifest much later

URI→route→controller→model→view→render

- Error may be hard to localize/reproduce if affects only some users, routes, etc.

What	Dev?	Prd?
Printing to terminal ("printf debugging")	✔	
Logging	✔	✔
Interactive debugging	✔	

RASP

- ***Debugging is a fact of life.***
- **R**ead the error message. Really read it.
- **A**sk a colleague an *informed* question.
- **S**earch using StackOverflow, a search engine, etc.
- Especially for errors involving specific versions of gems, OS, etc.
- **P**ost on StackOverflow, class forums, etc.
- Others are as busy as you. Help them help you by providing *minimal but complete* information

Reading Ruby error messages

- The *backtrace* shows you the call stack (where you came from) at the stop point
- A very common message: undefined method 'foo' for nil:NilClass
- Often, it means an assignment silently failed and you didn't error check:

```
@m = Movie.find_by_id(id)  # could be nil
```

Instrumentation (a/k/a “Printing the values of things”)

- In views:

= debug(@movie)

= @movie.inspect

- In the log, usually from controller method:

logger.debug(@movie.inspect)

- Don’t just use puts or printf! It has nowhere to go when in production.

Search: Use the Internet to answer questions

- Google it
- "How do I format a date in Ruby?"
- "How do I add Rails routes beyond CRUD?"
- Check the documentation
- api.rubyonrails.org, complete searchable Rails docs
- ruby-doc.org, complete searchable Ruby docs (including standard libraries)
- Check StackOverflow

Use rails console

- Like *irb*, but loads Rails + your app code
- But context is still not quite right for “peeking into” controllers and views
- Controllers rely on environment prepared by presentation tier
- Views rely on context set up by controllers
- Big guns: ruby-debug (demo shortly)

Rails Cookery #3

- To create a new submittable form:

1. Identify the action that gets the form itself
2. Identify the action that receives *submission*
3. Create routes, actions, views for each

- In form view, form element `name` attributes control how values will appear in `params[]`
- Helpers provided for many common elements

Redirection, the Flash and the Session

Receiving the form

- A neat trick: use debugger to inspect what's going on
  - start with `rails server --debugger`
  - insert debugger where you want to stop
  - more details & summary table in book (§3.15)
- To notice: params[:movie] *is a hash,* because of the way we named form fields
  - Conveniently, just what Movie.create! wants

What view should be rendered

- Idiom: *redirect* user to a more useful page.
  - e.g., list of movies, if create successful
  - e.g., New Movie form, if unsuccessful
- Redirect triggers a *whole new HTTP* request
- How to inform user *why* they were redirected?
- Solution: flash[]—quacks like a hash that *persists until end of **next** request*
  - flash[:notice] conventionally for information
  - flash[:warning] conventionally for "*errors*"

Flash & Session

- session[]: like a hash that persists forever
- reset_session nukes the whole thing
- session.delete(:some_key), like a hash
- By default, cookies store *entire contents* of session & flash
- Alternative: store sessions in DB table (Google “rails session use database table”)
- Another alternative: store sessions in a “NoSQL” storage system, like *memcached*

Finishing CRUD

Edit/Update pair is analogous to New/Create pair

- What's the same?
  - 1st action retrieves form, 2nd action submits it
  - "submit" uses redirect (to show action for movie) rather than rendering its own view
- What's different?
  - Form should appear with *existing* values filled in: retrieve existing Movie first
  - Form action uses PUT rather than POST

http://pastebin.com/VV8ekFcn

http://pastebin.com/0drjjxGa

Helper method	URI returned	RESTful Route and action	
movie_path(m)	/movies/1	PUT /movies/:id	update
movie_path(m)	/movies/1	DELETE /movies/:id	destroy

Destroy is easy

- Remember, destroy is an *instance* method
- Find the movie first...then destroy it
- Send user back to Index

```
def destroy
  @movie = Movie.find(params[:id])
  @movie.destroy
  flash[:notice] =
     "Movie '#{@movie.title}' deleted."
  redirect_to movies_path
end
```

Fallacies, pitfalls, and perspectives on SaaS-on-Rails

Fat controllers & views

- Really easy to fall into “fat controllers” trap
- Controller is first place touched in your code
- Temptation: start coding in controller method
- Fat views
- “All I need is this for-loop.”
- “....and this extra code to sort the list of movies differently.”
- “...and this conditional, in case user is not logged in.”
- No! Let controller & model do the work.

Designing for Service-Oriented

- A benefit of *thin* controllers & views: easy to retarget your app to SOA
- Typically, SOA calls will expect XML or JSON (JavaScript Object Notation, looks like nested hashes) as result
- A trivial controller change accomplishes this

http://pastebin.com/bT16LhJ4

Summary

- Rails encourages you to put real code in models, keep controllers/views thin
  - Reward: easier SOA integration
- Rails encourages convention over configuration and DRY
  - Reward: less code ➔ fewer bugs
- Debugging can be tricky for SaaS
  - Use logging, interactive debugger, printf
  - Soon: Test-driven Development to help *reduce bugs* in the first place