CS 169 Midterm 2 Review

Nov. 2, 2010

Don't forget...

- Mocks/stubs
- SOFA
- Ruby ideas like:
  - Blocks
  - Co-routines
  - Yield
  - Modules

Associations

- Read Appendix B from LR1E book -- that's how much DB knowledge we're expecting
- Chapter 9 in the LR1E book is also worth reading (and working through!)
- Types of associations:
  - One-to-many: has_many/belongs_to
  - One-to-one: has_one/belongs_to
  - Many-to-many:
    - has_many :through
    - has_and_belongs_to_many

One-to-many Associations

```
class Student < ActiveRecord::Base
  has_many :awards
end
class Award < ActiveRecord::Base
  belongs_to :student
end
```

awards

id	Award	Year	Student_id
1493	Best Handwriting	2007	1
1657	Nicest Smile	2007	3
1831	Cleanest Desk	2007	3
1892	Most likely to win the lottery	2008	4

students

id	given_name	middle_name	family_name	date_of_birth	grade_point_average	start_date
1	Giles	Prentiss	Boschwick	3/31/1989	3.92	9/12/2006
2	Milletta	Zorgos	Stim	2/2/1989	3.94	9/12/2006
3	Jules	Bloss	Miller	11/20/1988	2.76	9/12/2006
4	Greva	Sortingo	James	7/14/1989	3.24	9/12/2006
...	...	...	...	...	...	...

Foreign Key

Many-to-many Associations

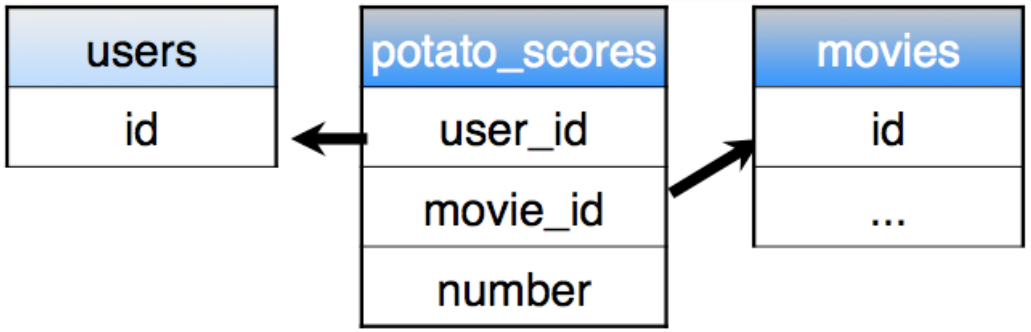
```
class User < ActiveRecord::Base
  has_many :potato_scores
  has_many :movies, :through => :potato_scores
end

class Movie < ActiveRecord::Base
  has_many :potato_scores
end

class PotatoScore < ActiveRecord::Base
  belongs_to :user
  belongs_to :movie
end
```

Many-to-many Associations

- Because of the way we've defined our models, we get the following helpers:
  - u.potato_scores
  - u.movies
  - m.potato_scores
  - ps.user
  - ps.movie
- Note: has_many :through => you can't associate a movie with a user unless you go through a potato score



Associations: Model vs. DB

- To correctly work with associations, you need to do both of the following:
  - Set up your migration so that any required foreign keys are generated
    eg, script/generate scaffold award name:string year:integer student_id:integer
  - Include association methods like has_many, belongs_to in your models. The associations module gives you helpers like s.awards
- Note: If you forget to include association methods but write a correct migration, you'll still have a foreign key in the awards table.

Aspect-Oriented Programming

- Cross-cutting concerns: logically centralized requirements that may appear in multiple places in implementation
- Advice: specific piece of code that implements a cross-cutting concern
- Pointcuts: places where we want to "run" advice at runtime
- Advice+Pointcuts make up aspect-oriented programming
- Downside to AOP: not necessarily clear what the execution order is because at pointcuts we alter it

AOP in Ruby on Rails

- In statically-compiled languages, compiler must "weave" advice into the code at the pointcuts
  - Ruby's dynamic nature means weaving is unnecessary!
  - But Ruby doesn't allow arbitrary pointcuts
- In Rails: AOP is used for validations, model lifecycle callbacks (e.g. before_save, after_update, etc), filters

Design Patterns

- We've seen:
  - Abstract Factory
  - Singleton
  - Null Object
  - Adapter
  - Composite
  - Proxy
  - Façade
  - Observer
  - Iterator
  - Strategy
  - Template

Dealing with APIs

- Adapter: translates an API to match a client's expectation
  - Goal: isolate client from differences in APIs that it might use
  - Example: database "adapters" for MySQL, Oracle, etc.
- Proxy: implements same methods as real API
  - Goal: hide remoteness, defer work
  - Example: intercept a bunch of bank deposit requests; send the batch of deposits all at once
- Façade: provides a wrapper around an elaborate API, exposing only what's needed
  - Goal: simplify client's interaction with API
  - Example: wrapper around AWS EC2 API that just lets you start and terminate instances

Singleton & Null Object

- Null Object: stand-in for a real object on which "important" methods of real object can be called
- Example: "the logged-in customer"
  - Can ask Customer.null_customer whether it's logged in, its name, etc.
  - Benefit: only one code path (don't always have to check whether user is logged in)
- Tie to Singleton: you may only want ONE copy of this null customer
  - Def: a class that provides only one instance, which anyone can access
  - It's still a member of the base class, but immutable and singular
- Though these often go together, know the point of each!

Composite

- Component whose operations make sense on both individuals and aggregates
- Example (from Design Patterns in Ruby): tasks involved in baking a cake
  - High-level task: MakeBatter
  - Decomposes into AddDryIngredients, AddLiquids, Mix
- Shared functionality:
  - Has an estimated duration
  - Can be added to process
- Can be recursive: MakeBatter is a sub task of MakeCake

Observer

- Problem: entity O ("observer") wants to know when certain things happen to entity S ("subject")
- Don't want to rewrite S every time a new observer arrives
- Example from HW4: ActiveRecord::CacheSweeper (for both models and controllers)
  - EventSweeper is the Observer
  - Event is the Subject

Security

- Know what the following attacks are:
  - SQL injection: when user input is unfiltered, allowing user to construct arbitrary SQL statement
  - Cross-site Request Forgery: unauthorized requests submitted on behalf of a user who is authenticated with a webpage
    - Exploits trust a site has for authenticated users

Security (2)

- More attacks:
  - Cross-site Scripting: vulnerability where malicious JS injected into a user's trusted webpage, bypassing browser security model
  - Buffer Overflow: bug where a program overwrites memory adjacent to a limited-length buffer
  - Denial of Service: attempt to make a site unavailable to users, usually by overwhelming it with bogus traffic

Security

- Know what good code practices can buy you:
  - Rails can "sanitize" text input by the user
    - h() function escapes HTML
    - Good coding practices in SQL queries:
      - Project.find(:all, :conditions => ["name = ?", params[:name]])
      - Project.find(:all, :conditions => "name = '#{params[:name]}'")

JavaScript

- Dynamic, OO language embedded in web browsers
  - First class functions/lambdas, "interesting" OO implementation
- Enables access to browser's rendering engine and DOM
  - Handlers associate JS functions with Events on DOM elements: onClick, onMouseOver, etc.
- XMLHTTPRequest: allow browser to make asynchronous requests in background, while page is loaded
- DOM: Document Object Model, accessed through window.document
  - Usual idiom: m = document.getElementByID("stuff"); then do something to/with m
  - Handlers on elements "trigger" JS by calling functions when event occurs

JavaScript & Rails

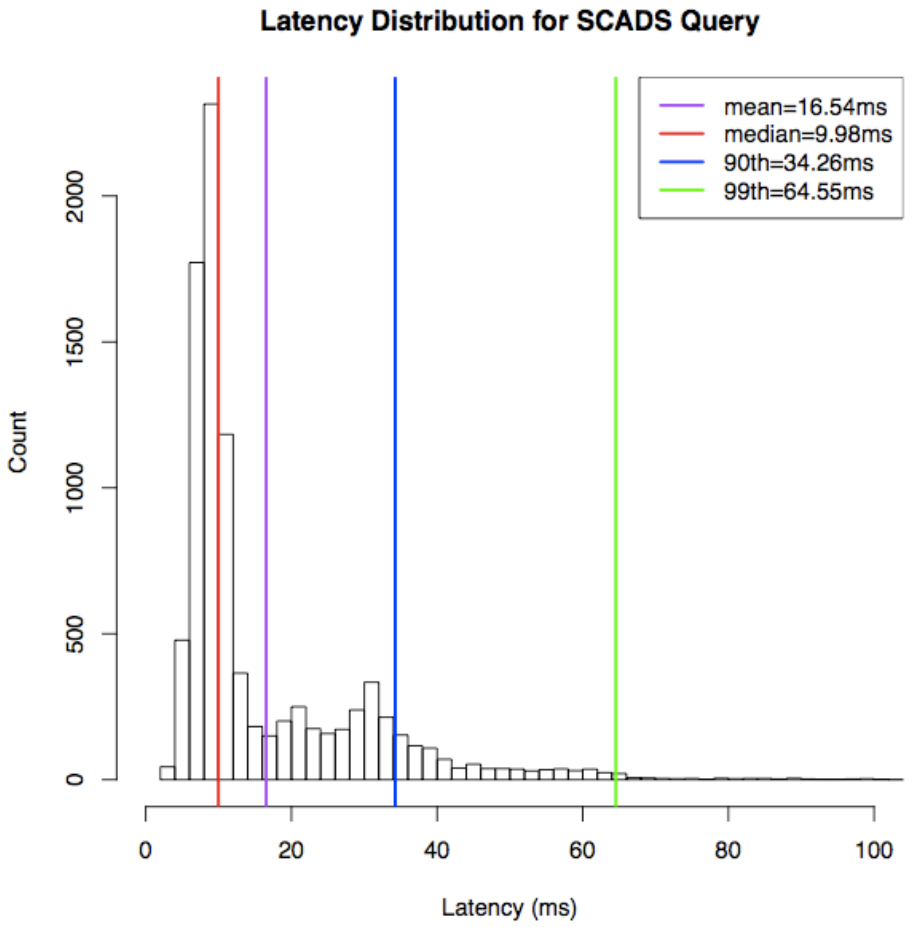
- Rails provides libraries to smooth out browser differences (e.g. Prototype library)
- Also provides helpers that generate JS
  - "Raw" JS requires many steps to do XHR
  - Rails tries to simplify using generators & controller methods
  - Usual thing: use methods in controller to respond to XHR requests & render partials, use link_to_remote or similar in view
- Concentrate on knowing when to use JS, not details on Rails helpers for using JS
- Knowing when to use will help your projects too :)

When to use JavaScript?

- Some things can be done using just CSS
  - e.g. changing link color when hovering above a link
  - e.g. having different styling for print vs. screen
- Interactive UIs
  - Many things can be done w/JS without XHR
    - e.g. disabling form fields depending on what other fields are set as ("Shipping Address same as Billing Address?")
  - Is the UI adding value or just annoying?
    - Sometimes minimalism provides a better user experience
- XHR
  - Provide rich experiences for users
    - Gmail, Google Docs
  - Beware: server load, "What's the URL" problem

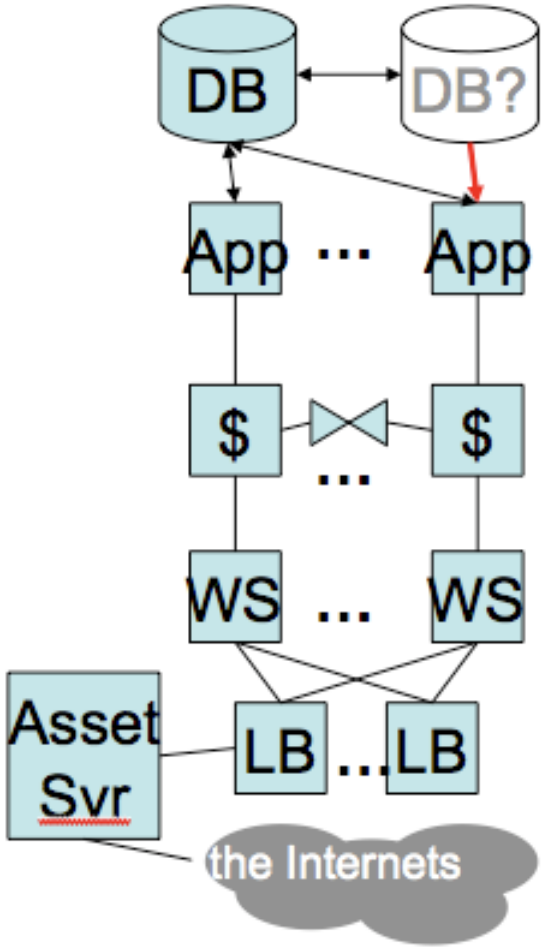
Service Level Objective (SLO)

- Time to satisfy user request ("latency" or "response time")
- Rather than average or worst case, specify target response time for X% of users
- Includes %ile, target response time, time window
- You should know what each of these colored lines means (eg, 99% of users have response time <= 64.55 ms)
- Don't forget that the *time window* over which the requests are observed is important!



Scaling

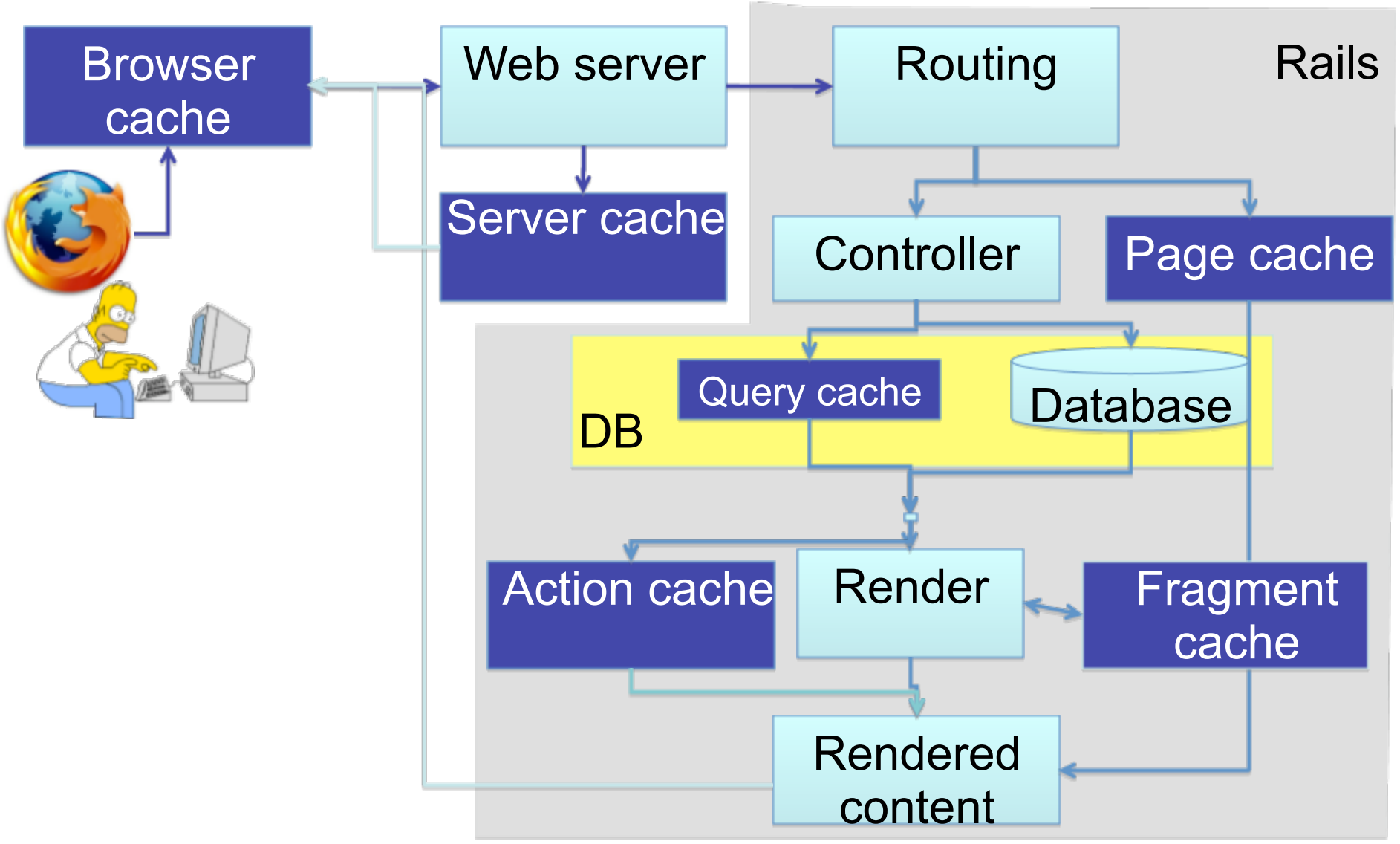
- How to respond to large user demand?
- Big idea: Replicate stateless components ("horizontal scalability")
- Goal: interchangeability (send any request to any server)
  - Requirement: app servers must be RESTful
- Very difficult to scale the database => it will become the bottleneck
- Can use EC2 servers => don't have to manage your own infrastructure; easy to scale out by requesting more instances



Caching

- Types of caching:
  - Browser caching
  - Rails caching
    - Good resource: http://guides.rubyonrails.org/caching_with_rails.html
    - Types:
      - Page caching (HW4)
      - Action caching
      - Fragment caching
  - Query caching
- Main benefit: speed
- Main drawback: inconsistency => need to manage cache expiration
- You should know when to use each type of caching

Cache Flow


Browser cache
Web server
Routing
Rails
Server cache
Controller
Page cache
DB
Query cache
Database
Action cache
Render
Fragment cache
Rendered content

REST + Caching

- Cacheability is an aspect of RESTfulness:
- From REST Wikipedia article: "Clients are able to cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance."
- Main idea: server tells client (ie, browser) whether the item is cacheable and how long it is expected to be valid

Page Caching vs. Action Caching

- Both apply if you want to cache a whole page
- Both are specified in controller
- Page caching (caches_page): request can be fulfilled at web server without touching Rails stack (ie, doesn't even hit controller)
  - Benefit: Very fast!
  - Drawback: Can't be used when you need authentication
- Action caching (caches_action): incoming request does go to Rails stack => can run before filters
  - Benefit: Allows authentication; still serves request from cached copy

Fragment Caching

- Lets you mix real-time and cached content
- Important because dynamic sites build pages from a variety of content (eg, news articles, ads, logo banner, etc.)
  - Some changes infrequently => good candidate for caching
  - Some is very dynamic/requires authentication => bad candidate for caching
- Happens in view

```
<% Order.find_recent.each do |o| %>
  <%= o.buyer.name %> bought <% o.product.name %>
<% end %>

<% cache do %>
  All available products:
  <% Product.all.each do |p| %>
    <%= link_to p.name, product_url(p) %>
  <% end %>
<% end %>
```

Query Caching

- Resource: http://expressionengine.com/user_guide/general/caching.html#query_caching
- Main Idea:
  - Cache query results at database
  - When a new query comes in, see if it can be served from cache (ie, see if it's been executed before).
  - Reduces database load