Ruby on Rails Short Course
Part 1: Hello World

Armando Fox

UC Berkeley RAD Lab

"A language that doesn't affect the way you think about programming is not worth knowing" — *Alan Perlis*

Welcome...

- Tour guides: Armando Fox, Will Sobel

Today you will...

- learn basic Ruby on Rails (~3 hours + breaks)
- eat (~1 hour)
- learn more Ruby on Rails (~3 hours + breaks)
- optional: install RoR on your computer (15 min.)
  - (1-click installers available on course page)
  - 6 sessions approx. 1 hour each, w./examples
- optional: Post-course discussion
  - overview of other RoR-related activities at UCB
  - discuss pedagogical opportunities
- **Any organizational/logistical questions?**

Goals & Non-goals

- Goals: enable you to...
  - understand RoR, see it in action, understand virtues & limitations vs. other frameworks/languages
  - participate intelligently in discussions about RoR
  - know where to go for further study/info (a/k/a know what you don't know)
- Non-goals
  - completeness/formality at expense of breadth/rapid uptake
  - all things to all people
  - interactive lab exercises (not enough time)

Assumptions

We assume you're familiar with:

- language features such as OOP and inheritance (eg at the level of Java)
- **Basic** familiarity with HTTP, HTML, relational databases (quick review provided)

EULA

- If you benchmark this course against other courses, you may release your results as long as you agree to comply with the RAD Lab's conditions of publication.
- You acknowledge and agree that the RAD Lab may automatically check the version of the OS you're using, monitor your application use, and upgrade the OS or applications with ones we think you should use.
- By not walking out of the room right now, you agree to fill out a 1-minute survey about this class and you agree to take it seriously.

Why you should understand RoR...

- ...if you're a developer
- ...if you're a practitioner
- ...if you're a faculty member
- ...if you're a student

Outline of the day

1. Web apps, MVC, SQL, Hello World
2. Just enough Ruby
3. Basic Rails

Lunch break

4. Advanced model relations
5. AJAX & intro to testing
6. Configure & deploy

Informal discussion: RoR and pedagogy

Outline of the day

1. Web apps, MVC, SQL, Hello World
2. Just enough Ruby
3. Basic Rails

Lunch break

4. Advanced model relations
5. AJAX & intro to testing
6. Configure & deploy

Informal discussion: RoR and pedagogy

Outline of Session 1

- Review: Web Apps 101
- Review: Model-View-Controller design pattern (MVC)
- Deconstructing Hello World
  - MVC and Rails
  - What's Where in a Rails App
- A slightly less trivial example

The Web is basically RPC (remote procedure call)

- RPC protocol == HTTP
  - ASCII based request/reply protocol run over TCP/IP
  - protocol *headers* specify metadata about the request
  - Stateless: notion of "session" must be synthesized separately
- RPC arguments == URL's, HTML-form contents
  - URL names a server & resource
  - URL may embed "argument values", or these can be "uploaded" as encoded HTML form submission

HTTP in one slide

- Browser opens TCP connection to server on port 80 (default) and sends:

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/4.73 [en] (X11; U; Linux 2.0.35
   i686)
   ...other boring headers...
Cookie: B=2vsconq5p0h2n
```

- Server replies:

```
HTTP/1.0 200 OK
Content-Length: 16018
Content-Type: text/html

<html><head><title>Yahoo!</title><base
   href=http://www.yahoo.com/>
…etc.
```

HTML in one slide

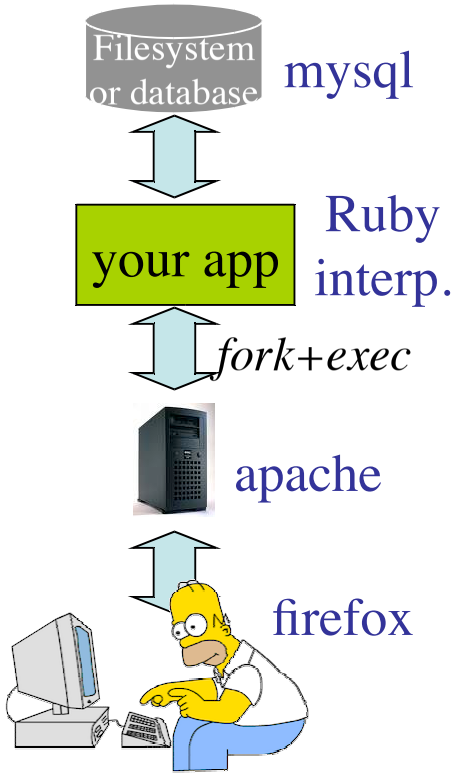
- Roughly hierarchical collection of elements that make up a viewable page
  - inline (headings, tables, lists...)
  - embedded (images, video, Java applets, JavaScript code...)
  - forms—allow user to submit simple input (text, radio/check buttons, dropdown menus...)
- Each element can have *attributes* (many optional)
  - of particular interest are *id* and *class* attributes
  - CSS (Cascading Style Sheets) allow specification of visual appearance of HTML pages based on the id's and/or classes of elements
- Current incarnation, XHTML, more device-portable by being strict about syntax that went to pot in HTML
  - RoR and many other frameworks generate XHTML-compliant code

Dynamic content generation in one slide

- Common gateway interface (cgi): run a program
  - Server (eg Apache) config info maps URLs to application names, hands URL off to program
  - Parameters and "function name" typically embedded in URL's or forms

```
http://www.foo.com/search?term=white
   %20rabbit&show=10&page=1
```

  - App generates HTML content (or instantiates HTML template with embedded code)
- *HTTP is stateless* (every request independent) so *cookies* quickly introduced
  - Client gets *cookie* from server on 1st visit
  - passes cookie to server on subsequent requests
  - Cookie typically used to look up *session info* in database or other store

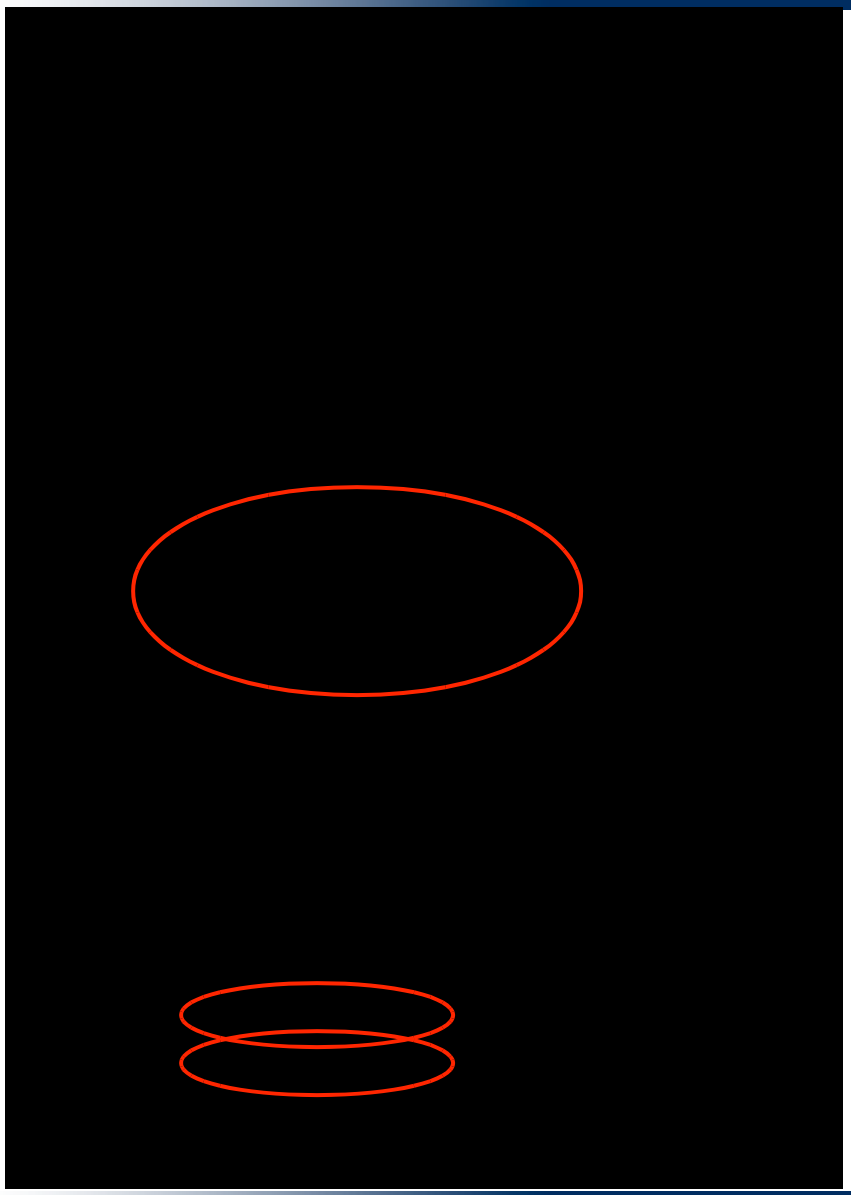


- Various *frameworks* have evolved to capture this common structure
  - IMHO, "framework" == locus of control/dispatching logic + class libraries, utility libraries, etc.

Summary: Web 1.0 apps

- Browser *requests* web resource (URL) using HTTP; server *responds* w/status code & content
  - HTML generally the most common content-type
  - Vast majority of HTML today is auto-generated from templates and/or dynamic content applications
- Another common request type: **POST**
- Another common (non-error) response status: **302 Found (redirect)**
  - original semantics: “This resource exists but has moved”
  - also used these days for handling “retry” type conditions in applications, as we’ll see

What `rails` *`appname`* does



- Once you install Rails...
  - *cd* somewhere
  - say `rails` *appname*
  - make sure your ISP has configured Apache to understand where Rails CGI dispatcher is
- *app/*, where the action is
  - especially *models, view, controllers*
- *script/*, useful scripts to help develop your app
- *test/* structure built right in! We'll meet it later

A truly trivial *hello world*

- in *app/controllers/hello_controller.rb*:

```
class HelloController < ApplicationController

  def say
  end

end
```

- in *app/view/hello/say.rhtml:*

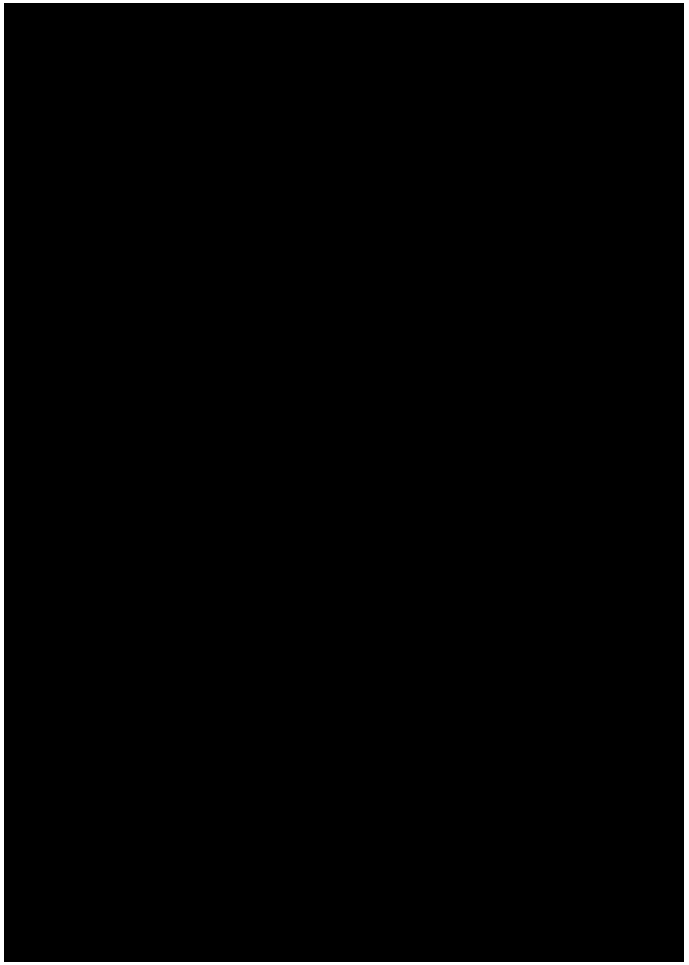
```
<h1> Hello World! </h1>
```

- And we invoke it by visiting:

http://mywebsite.com/cookbook/hello/say

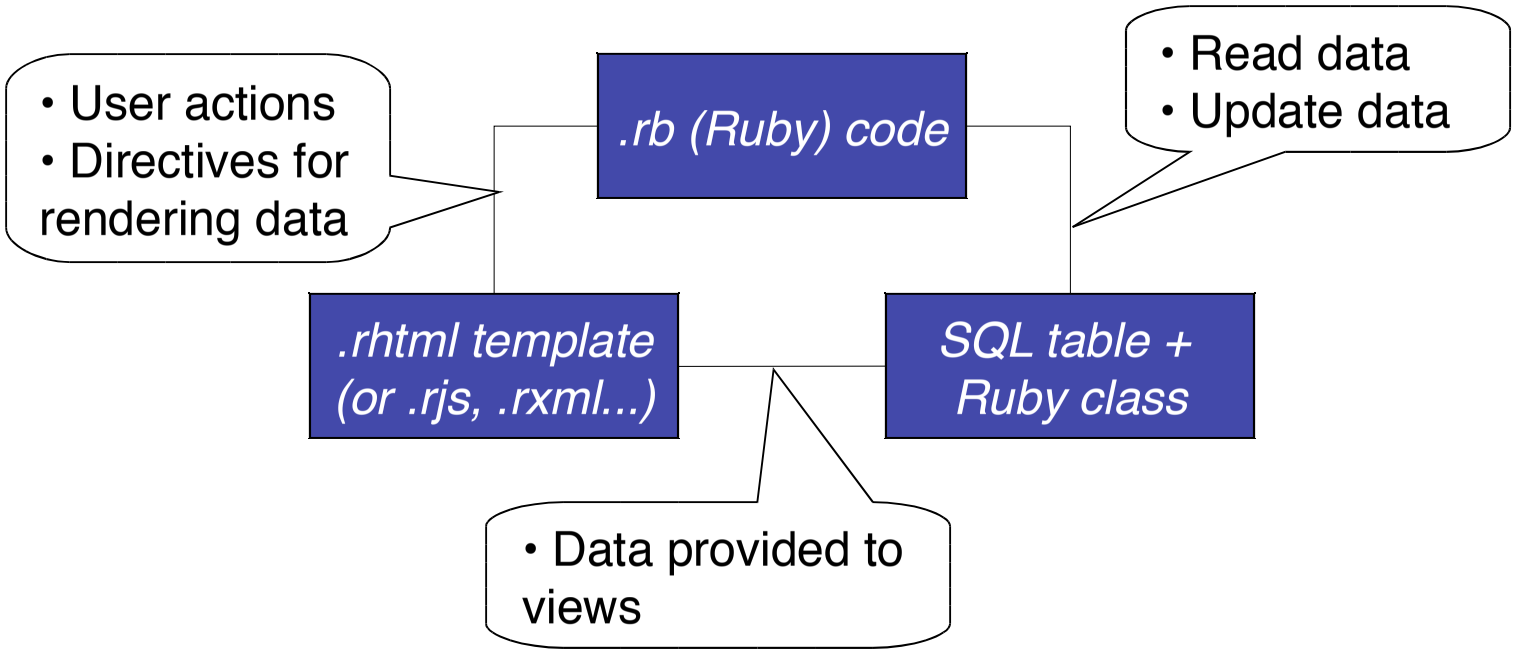
  - or maybe http://localhost:3002/hello/say
  - note similarities between URL and directory/file names...

- Let's make it only slightly less trivial...



The MVC Design Pattern

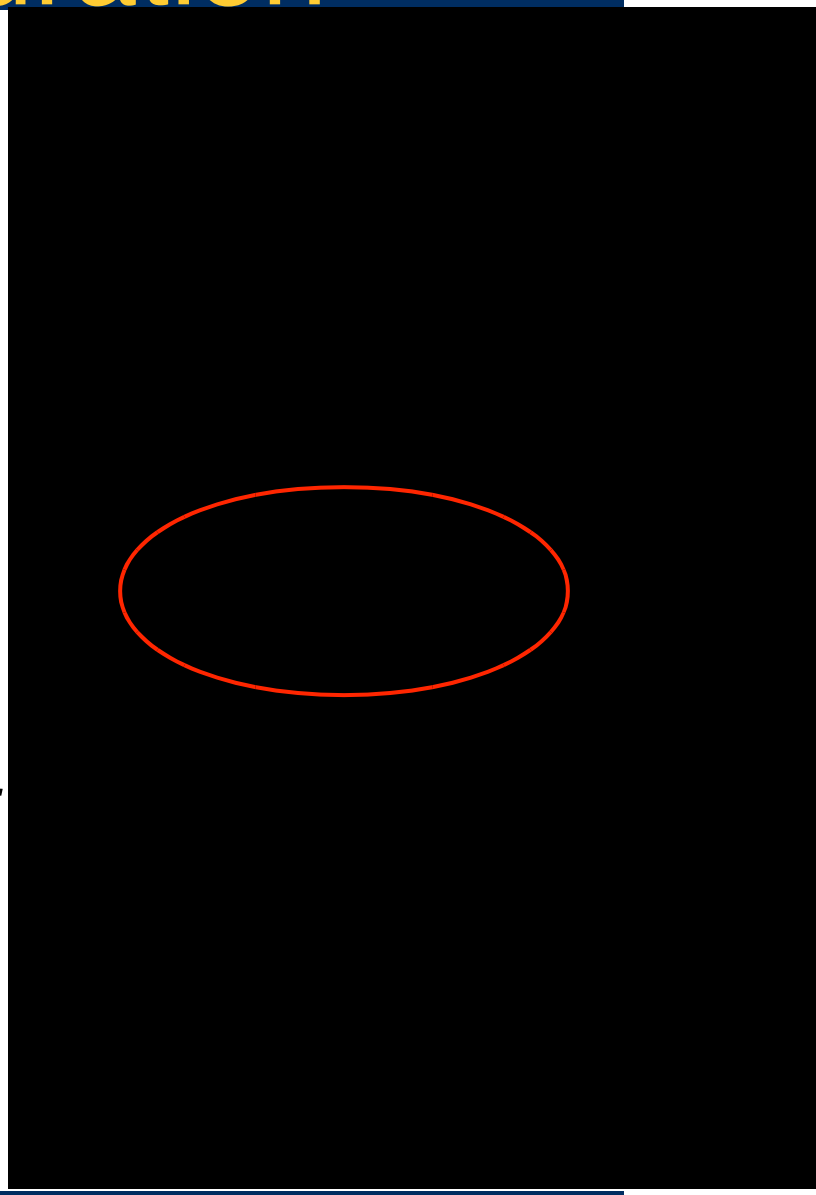
- Goal: separate organization of data (model) from UI & presentation (view) by introducing *controller*
  - mediates user actions requesting access to data
  - presents data for *rendering* by the view



MVC in RoR: Convention over Configuration

If data model is called *Student:*

- model (Ruby class) is *app/models/student.rb*
- SQL table is *students*
  - table row = object instance
  - columns = object attribute
- controller methods live in *app/controllers/student_controller.rb*
- views are *app/views/student/*.rhtml*



What about our trivial *hello world?*

- It manipulated no data models
  - though arguably this is where much of the strength of RoR lies
- One controller (*hello_controller.rb*)
- A handful of controller methods & views
  - Roughly, each controller method has a corresponding view

What is Ruby on Rails?

- Ruby is a *language* that is...
  - dynamically typed, interpreted, object-oriented, functionally-inspired
- Rails is a *web application framework* that...
  - embodies the MVC design pattern
  - emphasizes *convention over configuration*
  - *leverages* Ruby language features incl. dynamic typing, metaprogramming, & object-orientation to provide elegant support for both goals
- Recall: Framework == locus of control + class/utility libraries

A Less Trivial Example...

Let's walk through a full (single-table) MVC example...

1. Design the model
2. Instantiate the model (table & Ruby code)
3. Basic controller to do CRUD (Create, Read, Update, Destroy) operations on model

SQL 101 (Structured Query Language)

- *Relational model* of data organization (Codd, 1969) based on predicate logic & set theory
- Think of a table as an unordered collection of objects that share a *schema* of simply-typed attributes
  - *eg:* Student = <`lastname`:string, `ucb_id`:int, `degree_expected`:date>
- Think of SELECT as picking some records out
  - SELECT `lastname,ucb_id` FROM `students` WHERE `degree_expected < 12/31/07`
  - Generally: SELECT *attribs* FROM *tables* WHERE *constraints*
  - *Joins* are more interesting, we'll do them later

CRUD

- 4 basic operations on a table row: **C**reate, **R**ead, **U**pdate attributes, **D**estroy

```
INSERT INTO students
   (last_name, ucb_id, degree_expected)
   VALUES ("Fox", 99999, "1998-12-15"),
          ("Bodik", 88888, "2009-06-05")

SELECT * FROM students
   WHERE (degree_expected < "2000-01-01")

UPDATE students
   SET degree_expected="2008-06-05"
   WHERE last_name="Bodik")

DELETE FROM students WHERE ucb_id=99999
```

Rails ActiveRecord models

- ActiveRecord, a major component of Rails...
  - Uses SQL tables as underlying storage, and SQL commands as underlying manipulation, of collections of Ruby objects
  - (Later) Provides an object-relationship graph abstraction using SQL Joins as the underlying machinery
- For now, let's do a simple, single-table model
  1. Define the model attributes
  2. Create the database table
  3. Create a "degenerate" controller for manipulating Student objects

A simple, 1-table model

1. Define the model attributes: Student
   - last_name (string), UCB ID# (int), degree_expected (date)
1. Create the database table Students: 2 options
- Manually (bad, but simple for now...)
- Using migrations (good)...more on this later
  - Note, also creates *schema_info* table for schema versioning

Creating a simple controller: 2 ways to *scaffold*

1. "inline"
2. `script/generate scaffold` *modelname*
   - Individual controller methods & views overrideable either way

- What happened?
  - metaprogramming used to create the controller methods (which in turn call the methods that render "generic" views)
  - later method definitions override earlier ones
  - scaffold-rendering method respects existing *.rhtml* templates if they exist

More to notice about scaffolding

```
identical    app/models/student.rb
create    test/unit/student_test.rb
create    test/fixtures/students.yml
create  app/views/students/_form.rhtml
create  app/views/students/list.rhtml
create  app/views/students/show.rhtml
create  app/views/students/new.rhtml
create  app/views/students/edit.rhtml
create  app/controllers/students_controller.rb
create  test/functional/students_controller_test.rb
create  app/helpers/students_helper.rb
create  app/views/layouts/students.rhtml
create  public/stylesheets/scaffold.css
```

For creating test cases on *student* model & controller views

Capture common elements of *student*-related views

Convention over configuration

- Model *student*, table *students,* class *StudentsController* in *students_controller.rb,* views in *app/view/students/*, ...
  - metaprogramming makes it happen
- Table *students:* primary key *id*; object attribute names match table columns
  - does model *person* live in table *people*? does *goose* live in table *geese*?

Recap

- metaprogramming creates scaffolding, mapping between instance methods & table columns
- scaffolding gets your app off the ground early, then you can selectively replace it
- Rails scaffolding captures common model of a Web front-end to CRUD operations on database objects
  - *Much more powerful* when you consider multi-model relationships...after lunch
- Next: a closer look at MVC in Rails

Questions so far?
It just gets faster from here.

Questions