Slides adapted from CS 198 slides with the gracious permission of Armando Fox and Will Sobel

RUBY ON RAILS

CS 186 – Arsalan Tavakoli
3/18/2008

Today's Agenda / Goals

- Evolution of the Web
- What is Ruby on Rails?
- Brief overview of Ruby
- Rails
  - CRUD
  - Active Records/Controller/Views
  - Rendering
  - (Too Much to List Here)

- Probably won't finish it all, but serves as a good reference

Web 1.0

- Web 1.0: user interaction == server roundtrip
  - Other than filling out form fields
  - Every user interaction causes server roundtrip
  - Every roundtrip causes full page redraw
- Web 1.5: user interactions *without* contacting server
  - e.g. form validation before submit
  - e.g. selecting something from menu A causes contents of menu B to change
  - But every roundtrip *still* causes full page redraw

Web 2.0

- *Separation* of server roundtrip from page rendering
  - Initial load of page & draw page
  - User interaction causes *background* roundtrip to server
  - Response from server “captured” and passed to a programmer-defined JavaScript function
  - That function can redraw *part of* the page *in place* (using same mechanisms as Web 1.5)
- Result: “desktop-like” responsive UI’s *that can contact server*
  - Auto completion
  - “Lazy” fetch of complicated parts of page
  - etc

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
- Rails handles everything up to the point where *your code* is called
- And everything past the point where *your code* delivers stuff to the user.

A Couple of Notes

- We are using Rails 1.2.3, not Rails 2.0
  - Slight differences between the two, be careful when looking at tutorials on the web.
- Install RoR on your computer for easier access
  - InstantRails for Windows
  - Locomotive for Mac OSx
- LOTS of simple ROR tutorials out there
  - *Rolling with Ruby on Rails (Revisited)* is the most popular and a good place to start

Ruby

- Purely Object-Oriented Language
  - EVERYTHING is an object, and EVERYTHING has a type
- Borrows from:
  - Lisp, Perl, Smalltalk, and CLU
- Exists outside of Rails (but the reverse isn't true)

- irb: Ruby's built-in interpreter to test out commands and test code
- ri: Ruby's equivalent to 'man'

Variables

- A Variable in Ruby holds objects
  - Since everything is an object, a variable can hold anything!
- Variable names indicate scope, not type
  - **Local Variable**: foo, bar, _temp
  - **Instance Variable:** @foo, @bar, @_temp
  - **Symbol:** :foo, :bar, :_temp
  - **Array:** [1, “1”, :one]
  - **Hash:** { :one => 1, ‘two’ => 2}

Review: Naming Conventions & Syntax

- ClassNames

```
class NewRubyProgrammer ...  end
```

- method_names and variable_names

```
def learn_conventions  ...  end
```

- predicate_like_methods?

```
def is_faculty_member?  ...  end
```

- Return values
  - Each method returns a single object
  - If no explicit return statement, then return object is that which was last referenced.

```
Def return_10(v)
   10;
End;
```

Review: Syntax

- Syntax features
  - Whitespace is not significant (unlike Python)
  - Statements separated by semicolons or newlines
  - Statement can span a newline*
  - Parentheses can often be omitted*
    - when unambiguous to parser; use caution!!

✓
```
raise "D'oh!" unless valid(arg)
```

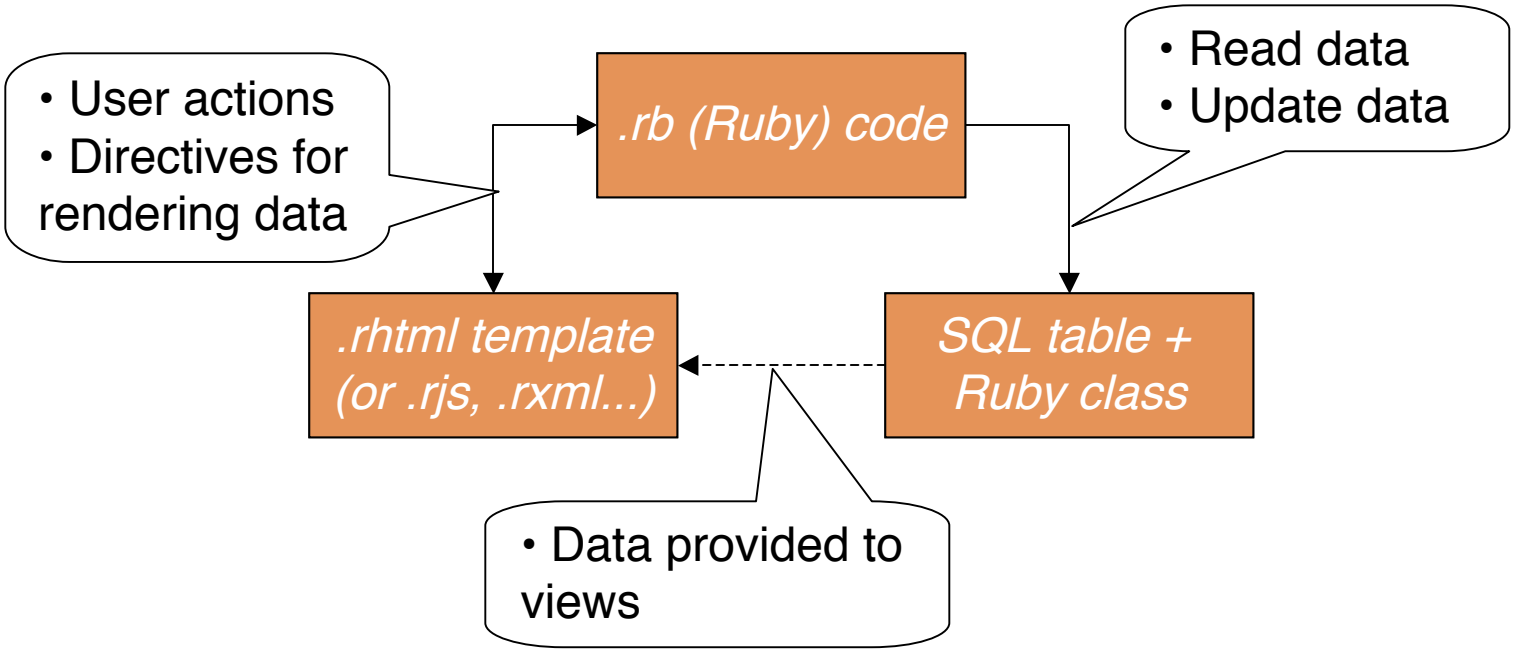
✓
```
raise "D'oh!" unless
  valid arg
```

✘
```
raise "D'oh!"
  unless valid(arg)
```

- Advice: *use a good text editor*

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 are “sort of” MVC by design



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 001

- A SQL *table* has a number of *rows* of identical structure
- Each row has several *columns (fields, attributes, etc.)*
- You can define relationships between tables (*associations*)—we'll get to that later
- A collection of tables & relationships is called a *schema*

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 methods (a/k/a object instance variables)
- controller methods live in *app/controllers/student_controller.rb*
- views are *app/views/student/*.erb.html*
  - and other types of views we'll meet later

Preview: CRUD in SQL

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

```
INSERT INTO students
  (last_name, ucb_sid, 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_sid=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
- Oversimplification: 1 instance of Ruby class Foo == 1 row in a SQL table called *Foos*
- Let Rails do the work of creating our model and related stuff:

```
script/generate scaffold student
last_name:string first_name:string
ucb_id:integer degree_expected:datetime
```

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

Capture common elements of *student*-related views

Creating the Students table

- We're not done yet! Students table doesn't exist...so let's define a student
  - edit the *migration* file `001_create_students.rb`
  - give each student a first & last name, UCB ID, degree date
- Let Rails do the work of interacting with the database:

```
rake db:migrate
```

- Question: *what database?*
  - *config/database.yml*

While we're on SQL... what's a *primary key* anyway?

- Column whose value must be unique for every table row
  - Why not just use (e.g.) last name or SID#?
- SQL AUTO_INCREMENT function makes it easy to specify an integer primary key
- If using *migrations* to create tables (recommended), Rails takes care of creating an autoincrement primary key field called ID

```
CREATE TABLE students (
  id INT NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(255),
  first_name VARCHAR(255),
  ucb_sid INT(11) DEFAULT 9999
);
```

```
class CreateStudents<ActiveRecord::Migration
 def self.up
   create_table :students do |tbl|
     tbl.column :last_name, :string
     tbl.column :first_name, :string
     tbl.column :ucb_sid, :integer,
        :null=>false, :default=>9999
   end
 end

 def self.down
   drop_table :students
 end
end
```

Recap

- **CRUD,** the four basic operations on database rows
- **ActiveRecord**, a library that arranges to "map" your models into database rows
- **scaffolding** gets your app off the ground early, then you can selectively replace it
  - captures common model of a Web front-end to CRUD operations
- **convention over configuration** makes both of the above tractable to implement while saving you work

Active Record: what is it?

- A class library that provides an object-relational model over a plain old RDBMS
- Deal with objects & attributes rather than rows & columns
  - SELECT result rows ⇔ enumerable collection
  - (later) object graph ⇔ join query

More on Student Example

- object attributes are "just" instance methods (a la `attr_accessor`)
  - so can already say `stu.last_name`, `stu.ucb_sid`, etc.
  - what line in what file makes this happen?
- ActiveRecord accessors/mutators
  - default `attr_accessor` for each table column
  - perform type-casting as needed
  - can be overridden, virtualized, etc.

Example: a short tour

```
class Student

  def youngster?
    self.degree_expected > Date.parse("June 15, 2008")
  end

  def days_till_graduation_as_string
    graduation = self.degree_expected
    now = Date.today
    if graduation.nil?
      "This person will never graduate."
    elsif  graduation < now
      "Graduated #{now-graduation} days ago"
    else
      "Will graduate in #{graduation-now} days"
    end
  end

end
```

Predicate-like method names often end with question mark

`self` (like Java `this`) not strictly necessary here

Some useful class methods of `Date`

Interpolation of expressions into strings

Constructors

- Method named *initialize*, but invoked as *new*
- (at least) 3 ways to call it...

```
s = Student.new(:last_name => "Fox",
                # unspecified attributes get
                #   table column's DEFAULT values
                :ucb_id => 99988)

s = Student.new do |stu|
  stu.last_name = "Fox"
  stu.ucb_id = 99988
end

s = Student.new
s.last_name = "Fox"
s.ucb_id = 99988
```

New != Create

- Call `s.save` to write the object to the database

  `s.create(args)` ≈ `s.new(args); s.save`

  `s.update_attributes(hash)` can be used to update attributes in place

  `s.new_record?` is true iff no underlying database row corresponds to `s`

- `save` does right thing in SQL (INSERT or UPDATE)
- Convention over configuration:
  - if `id` column present, assumes primary key
  - if `updated_at/created_at` columns in table, automatically are set to update/creation timestamp

find() ≈ SQL SELECT

```
# To find an arbitrary single record:
s = Student.find(:first)  # returns a Student instance
# To find all records:
students = Student.find(:all)  # returns enumerable!

# find by 'id' primary key (Note! throws RecordNotFound)
book = Book.find(1235)
# Find a whole bunch of things
ids_array = get_list_of_ids_from_somewhere()
students = Student.find(ids_array)

# To find by column values:
armando = Student.find_by_last_name('Fox') # may return nil
a_local_grad =
   Student.find_by_city_and_degree_expected('Berkeley',
   Date.parse('June 15,2007'))

# To find only a few, and sort by an attribute
many_localgrads =
   Student.find_all_by_city_and_degree_expected('Berkeley',
   Date.parse('June 15,2007'),:limit=>30,:order=>:last_name)
```

Find by conditions

- Use ? for values from parameters. Rails will sanitize the SQL and prevent any SQL injection

```
Student.find(:all, :conditions => "last_name LIKE 'fox' AND
  degree_expected > #{Date.parse('June 15,2007').to_formatted_s}")
# better - sanitizes SQL to avoid injection attacks, and does type casting:
Student.find(:all, :conditions => ["last_name LIKE ? AND degree_expected > ?",
                                   tainted_lastname, Date.parse('Jun 15,07')])
```

You can also specify ordering and use arbitrary SQL operators:

```
# Using SQL conditions
books = Book.find(:all,
   :conditions => ['pub_date between ? and ?',
   params[:start_date], params[:end_date]],
   :order => 'pub_date DESC')
```

Find by conditions

- Use ? to substitute in condition values
  - not mandatory, but a good idea!

```
Student.find(:all, :conditions => "last_name LIKE 'fox' AND
  degree_expected > #{Date.parse('June 15,2007').to_formatted_s}")
# better - sanitizes SQL to avoid injection attacks, and does type casting:
Student.find(:all, :conditions => ["last_name LIKE ? AND degree_expected > ?",
                                   tainted_lastname, Date.parse('Jun 15,07')])
```

- You can include other SQL functionality

```
# Using SQL conditions
books = Book.find(:all,
   :conditions => ['pub_date between ? and ?',
   params[:start_date], params[:end_date]],
  :order => 'pub_date DESC')
```

- You can roll your own

```
s = Student.find_by_sql("SELECT * FROM students ...")
```

Advanced Find

You can also specify limits and offsets, and oh so much more

```
books = Book.find(:all,
   :conditions => ['pub_date between ? and ?',
       params[:start_date], params[:end_date]],
    :limit => 10, :offset => params[:page].to_i * 10)
```

- `:lock` - Holds lock on the records (default: share lock)
- `:select` - Specifies columns for SELECT (default *)
- `:group` - (used with select) to group
- `:readonly` - load as read-only (object can't be saved)
- `:include` - Prefetches joined tables
- Note: use SQL-specific features at your own risk....

Caveat!

- The result of a find-all operation *mixes in* `Enumerable`
- `Enumerable` defines methods `find` and `find_all`
- Not to be confused with `ActiveRecord::Base#find`!

```
students = Student.find(:all, :conditions => ["degree_expected > ?", Time.now])
palindromic = students.find_all { |s| s.last_name.reverse == s.last_name }
lucky = palindromic.find { |s| s.ucb_id.odd? }
```

Action View

- A template for rendering views of the model that allows some code embedding
  - commonly RHTML (.html.erb); also RXML, HAML, RJS
  - note...too much code breaks MVC separation
  - convention: views for model *foo* are in `app/views/`foo/
- "Helper methods" for interacting with models
  - model values→HTML elements (e.g. menus)
  - HTML form input→assignment to model objects
- DRY (Don't Repeat Yourself) support
  - *Layouts* capture common page content at application level, model level, etc. (`app/views/layouts/`)
  - *Partials* capture reusable/parameterizable view patterns

Helper Methods for Input & Output

- Views: Insert Ruby code snippets among HTML
  - Anatomy: `<% code %>` `<%= output %>`
- But these form tags are generic...what about model-specific form tags?
- In the RHTML template:

```
<%= form_for(@student) do |f| %>
```

  - ...etc....
- In HTML delivered to browser:

```
<input id="student_last_name"
  name="student[last_name]" size="30" type="text"
  value="Fox" />
```

- What happened?

Action Controller

- Each incoming request instantiates a new Controller object with its own instance variables
  - Routing determines which method to call
  - Parameter unmarshaling (from URL or form sub.) into `params[]` hash

 ...well, not really a hash...but responds to `[], []=`

- Controller methods set up instance variables
  - these will be visible to the view
  - controller has access to model's class methods; idiomatically, often begins with `Model.find(...)`

Then we render...

- Once logic is done, render the view

```
render :action => 'edit'
render :action => 'edit', :layout => 'false'
render :text => "a bare string"
# many other options as well...
```

  - exactly one *render* permitted from controller method (1 HTTP request ⇔ 1 response)
- Convention over configuration: implicit *render*
  - if no other *render* specified explicitly in action method
  - looks for template matching controller method name and renders with default layouts (model, app)

What about those model-specific form elements?

- Recall:
  ```
  <input type="text" id="student_last_name"
  name="student[last_name]"/>
  ```
- Related form elements for student attributes will be named `student[`*attr*`]`
  - marshalled into params as `params[:student][:last_name]`, `params[:student][:degree_expected]`, etc.
  - i.e, `params[:student]` is a hash :last_name=>string, :degree_expected=>date, etc.
  - and can be assigned directly to model object instance
  - helpers for dates and other “complex” types...magic

What else can happen?

- `redirect_to` allows falling through to different action *without* first rendering
  - fallthrough action will call render instead
  - works using HTTP 302 Found mechanism, i.e. separate browser roundtrip
- example: update method
  - fail: *render* the `edit` action again
  - success: redirect to "URL indicated by this `@student` object"
- alternate (older) syntax for redirects:

```
redirect_to :action => 'show', :id => @student.id
```

The Session Hash

- Problem: HTTP is stateless (every request totally independent). How to synthesize a *session* (sequence of related actions) by one user?
- Rails answer: `session[]` is a magic persistent hash available to controller
  - Actually, it's not really a hash, but it quacks like one
  - Managed at dispatch level using cookies
  - You can keep full-blown objects there, or just id's (primary keys) of database records
  - Deploy-time flag lets sessions be stored in filesystem, DB table, or distributed in-memory hash table

The Flash

- Problem: I'm about to `redirect_to` somewhere, but want to display a notice to the user
- yet that will be a different controller instance with all new instance variables
- Rails answer: `flash[]`
  - contents are passed to the *next* action, then cleared
  - to this action: `flash.now[:notice]`
  - visible to views as well as controller

```
def controller_method_1
  if (badness)
    flash[:notice] = "You lose!"
    redirect_to :action => 'try_it'
  end
end

def try_it
  #...some stuff...
end

# in try_it.rhtml:

<% if flash[:notice] %>
  <p class="errorMsg">
    <%= flash[:notice] %>
  </p>
<% end %>
```

- Strictly speaking, could use session & clear it out yourself

Controller predicates: `verify`

- A declarative way to assert various preconditions on calling controller methods
- You can check selectively (`:only, :except`) for:
  - HTTP request type (GET, POST, Ajax XHR)
  - Presence of a key in the flash or the session
  - Presence of a key in `params[]`
- And if the check fails, you can...
  - `redirect_to` somewhere else
  - `add_to_flash` a helpful message
- Example:

```
verify :method => :post, :only => 'dangerous_action',
  :redirect_to => {:action => 'index'},
  :add_to_flash => "Dangerous action requires Post"
```

More General Filters

- Code blocks that can go before, after or around controller actions; return Boolean

```
before_filter :filter_method_name
before_filter { |controller| ... }
before_filter ClassName
```

  - options include `:only`, `:except`, etc.
  - multiple filters allowed; calls provided to prepend or append to filter chain
  - subclasses inherit filters but can use `skip_filter` methods to selectively disable them
- If any before-filter returns false, chain halted & controller action method won't be invoked
  - so filter should redirect_to, render, or otherwise deal with the request
- Simple useful example: a before-filter for nested routes!

```
before_filter :load_professor
def load_professor
  @professor = Professor.find(params[:professor_id])
end
```

Intro. to Associations

- Let's define a new model to represent Courses.
  - keep it simple: name, CCN, start date (month & year)
- What's missing: a way to identify who is in the class!
- Rails solution: (similar to database *foreign keys*)
  - Add column `course_id` to Students table
  - Declare that a Course `has_many :students`
  - Both of these are Rails *conventions*
- Set a given student's course_id field for the course they are taking
  - An obvious problem with this approach...but we'll fix it later

Associations In General

- x *has_many* y
  - the *y* table has an *x_id* column
  - y *belongs_to* x
  - Note! Table structure unaffected by whether you also define the *belongs_to....*so why do it?
- x *has_one* y
  - actually like *has_many:* does same SQL query but returns *only the first* result row

Using Associations

- Going forward (course has_many students):

```
@c = Course.find(...)
@s = @c.students
```

  - What is the “type” of @s?

- Going the other way (student belongs_to course):

```
@s  = Student.find(...)
@c = @s.course
```

Modeling professors

- How should we change the schema to support *course belongs_to professor*?

```
class Course < ActiveRecord::Base
  has_many :students
  belongs_to :professor
end


class Professor < ActiveRecord::Base
  has_many :courses
end
```

What about all the students that a professor teaches?

```
@p = Professor.find(...)
@c = Professor.courses
@s = @c.students
```

- Or....

```
class Professor < ActiveRecord::Base
  has_many :courses
  has_many :students, :through => :courses
end
```

- Now we can just write:

```
@s = Professor.find(...).students
```

What is happening in terms of tables in this example?

- SQL is doing a **join**
- Which you'll learn about next time....
- The message is:
  Active Record tries to provide the abstraction of an *object graph* by using SQL table *joins*.
- The *xxx_id* fields are called *foreign keys.*

Virtual attributes example: simple authentication

- Assume we have a table *customers* with columns *salt* and *hashed_password*...

```
class Customer

  def password=(pass)
    pw=pass.to_s.strip
    self.salt = String.random_string(10)
    self.hashed_password = Digest::SHA1.hexdigest(pw + self.salt)
  end

  def self.authenticate(username,pass)
    (u=find(:first, :conditions=>["username LIKE ?", username]) &&
     Customer.encrypt(pass,u.salt) == u.hashed_password)
  end

end
```

Defines the *receiver method* for `password=`

Why do we want to use `self` here?

Where's the accessor for *password*?

Summary

- ActiveRecord provides (somewhat-)database-independent object model over RDBMS
- ActionView supports display & input of model objects
  - facilitates reuse of templates via layouts & partials
- ActionController dispatches user actions, manipulates models, sets up variables for views
  - declarative specifications capture common patterns for checking predicates before executing handlers

$APP_ROOT/config/routes.rb

- Ruby code (that makes use of high-level methods!) to declare “rules” for mapping incoming URLs to controllers/actions
- actually each rule has 2 purposes:
  1. map incoming URL to ctrler/action/params
  2. *generate* URL to match ctrler/action/params
     - e.g. when using link_to, redirect_to, etc.
- What’s in a rule?
  - A URL template
  - Keywords stating what to do

Simple example

- In routes.rb:

```
map.connect 'professors/:dept',
  :controller => 'professors', :action => 'list'
```

- In one of your views:

```
<%= link_to "List professors in EECS",
  :controller => 'professors', :action => 'list',
  :dept => 'eecs', :hired_since => 2005  %>
```

  - matching is determined by *keywords*
  - `link_to` uses underlying function `url_for`, which consults routing rules to build the URL:

```
http://www.yourapp.com/professors/eecs?hired_since=2
  005
```

Simple example cont.

- In routes.rb:

```
map.connect 'professors/:dept',
  :controller => 'professors', :action => 'list'
```

- Now if someone visits this URL:

```
http://www.yourapp.com/professors/eecs
```

  - Matching is determined by *position*

- How about:

```
http://www.yourapp.com/professors/eecs?glub=1&hired_
  since=2006
```

- How about:

```
http://www.yourapp.com/professors
```

Default routes

- URL is compared to routing rules, one at a time, until match found
  - then “wildcard” pieces of URL get put into `params[]`
- If no match, *default route* (last one in routes.rb) is used
  - typically something like:
    `map.connect ':controller/:action/:id'`
  - e.g., catches things like `professors/edit/35`
  - **Warning!** Can lead to dangerous behaviors
- Use the *root route* to map the “empty” URL (e.g. http://www.myapp.com):

```
map.root :controller=>'main', :action=>'index'
```

More on Routes

- Ordering of routes matters; more specific ones should come earlier so they'll match first

```
map.connect 'users/:action/:id'
map.connect ':controller/:action/:id'
```

- Many, many apps will never need to use more than the "conventional" predefined routes
- If you want to, you should definitely read more about routes offline

REST is CRUD

- REST Idea: each HTTP interaction should specify, *on its own*, a CRUD operation and which object to do it on.
  - GET used for read operations; POST for writes (create, update, delete)
  - Also guards against spidering/bots!
- Rails 2.0: routes, scaffolds and URL helpers are now all RESTful by default
  - result: syntax of link_to, etc. has changed
- **Get them by saying** `map.resources :model`

REST and URI's

Action & *named route* method to pass to *url_for*	HTTP method	Old style URL	New (RESTful) style URL
show: `student_url(@s)`	GET	/:ctrl/show/:id	/:ctrl/:id
index (list): `students_url`	GET	/:ctrl/list (or /:ctrl/index)	/:ctrl
new: `new_student_url`	GET	/:ctrl/new	/:ctrl/new
create: `students_url`	POST	/:ctrl/create (why no ID?)	/:ctrl
destroy: `student_url(@s)`	DELETE	/:ctrl/destroy/:id	/:ctrl (but see Hack!)
edit: `edit_student_url(@s)`	GET	/:ctrl/edit/:id	/:ctrl/:id/edit
update: `student_url(@s)`	PUT	/:ctrl/update/:id	/:ctrl/:id