Week 4
Modules: 4

Starting Berkeley Time

Topics:

- Model-View-Controller Architecture

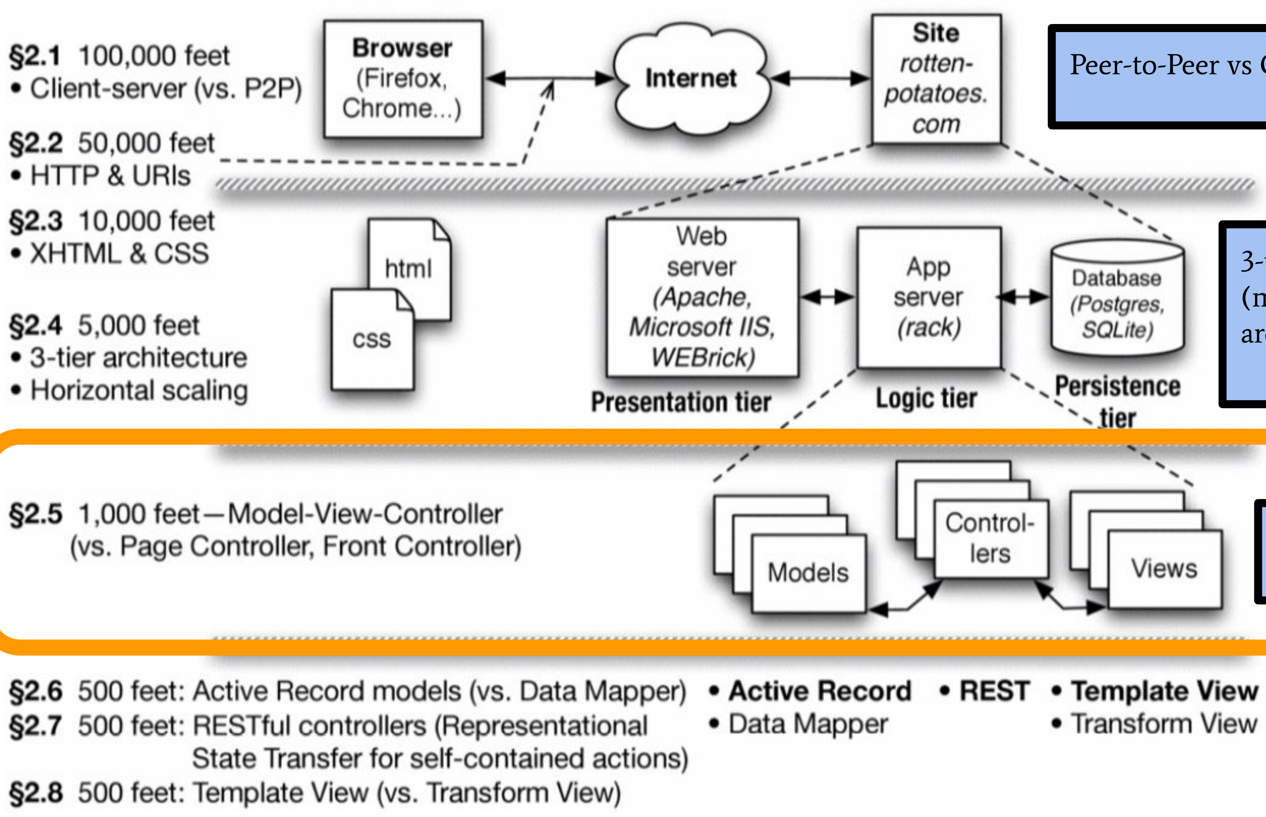
Model-View-Controller Architecture

General Idea

- **Models:** concerned with the data manipulated by the application: how to store it, how to operate on it, and how to change it.
- **Views:** serve as the interface between the system’s users and its data
- **Controllers:** mediate the interaction in both directions
  - When a user interacts with a view (passing data to the Model)
  - Delivering data from the model to render on the view

Model-View-Controller Architecture

Where are we?


§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
Browser (Firefox, Chrome...)
Internet
Site rotten-potatoes.com
html
css
Web server (Apache, Microsoft IIS, WEBrick)
App server (rack)
Database (Postgres, SQLite)
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

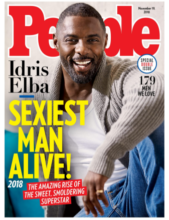
Peer-to-Peer vs Client-Server

3-tier architecture (multi-tier architecture)

MVC Design

Here!

Model-View-Controller Architecture

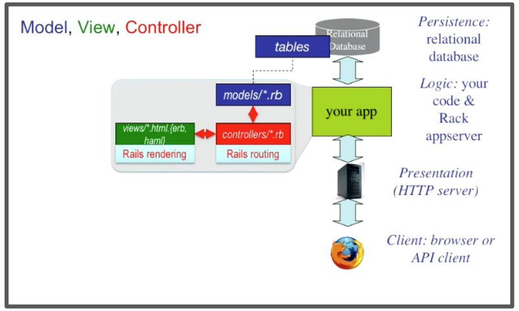
Models 

????? Rails **??? Explain**

- Rails supports apps that follow the MVC Pattern. The Framework provides powerful base classes that your apps models, view, and controllers inherit
- Each Rails model is a resource type whose instances are rows in a particular table of a relational db. These DB operations are exposed to the developer via **Active Record** (more on this later)
- Rails apps are, by default, REST. Each resource consists of its own set **M-V-C**. Resources can have relationships with each other. Using Foreign Keys we can map relationships between each of our resources.

Where does the responsibility of mapping relationships to other resources lie (M, V, or C)?

The Responsibility lies with the Model! Why?



Model-View-Controller Architecture

What is Ruby-on-Rails? (cont.)

- Rails is a server-side framework, it needs a way to map an HTTP route to code in the app that performs the correct action. The Rails routing subsystem provides a flexible way to map routes to Ruby methods located in Rails controllers. You can define routes any way you like, but if you choose to use some “standard” routes based on RESTful conventions, most of the routing is set up for you automatically.
- Rails was initially developed for serving HTML pages to browser clients, but it can just as easily serve a RESTful JSON API
- Rails has very strong opinions on the mechanical detail of your apps implementation. As a result it will abstract away a lot of implementation details and auto generate a lot of code. It uses **convention over configuration** to save us a lot of time.

Throwback: This is an example of what developer productivity tool? (Clarity via conciseness, Synthesis of Implementation, Reuse, Automation via Tools)

Pros? Cons?

Less flexibility, more saved time

Synthesis of Implementation

Models + ActiveRecord (Deep Dive)

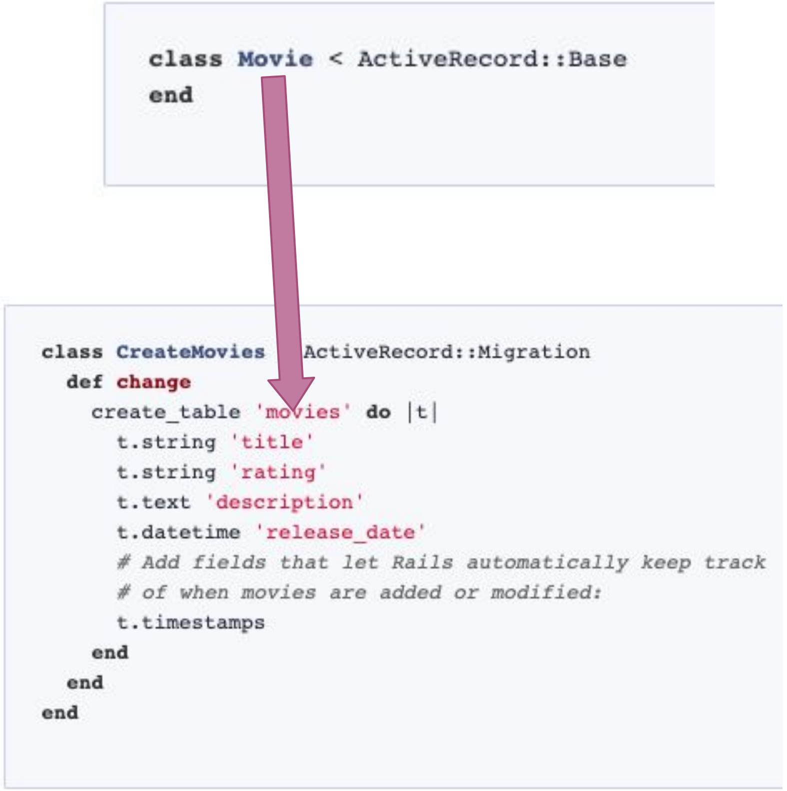
By the time this discussion is done, you need to be able to answer these questions… Might be helpful in your quiz? Who knows...

- What is the correspondence between how an instance of a resource is stored in the database and how it is represented in the programming language used by the framework (in this case, Ruby)?
- What software mechanisms mediate between those two representations, and what programming abstractions do those mechanisms expose?
- These sort of paradigms and ORM patterns are in a lot of languages. This course is trying to teach you how to learn how to learn (So meta).

Models + ActiveRecord (Deep Dive)

ActiveRecord, Models, Databases (basic)

- The app must be able to store different types of data items, or entities, in which all instances of a particular type of entity share a common set of attributes.
- Deals with data: storing it, operating on it, or changing it. One model for each resource. Gives each model the knowledge to read, delete, create, update instances of itself in the database.
- Rails implements ActiveRecord Architectural Pattern:
  - Rails models are classes that are backed by a table in the RDBMSs such that an instance of the class corresponds to a single row in the table.
  - easy interaction between the logic and persistence layers (generate SQL statements at runtime)
  - built-in functionality to perform CRUDI operations on the model:
    - Create, Read, Update, Delete, Index



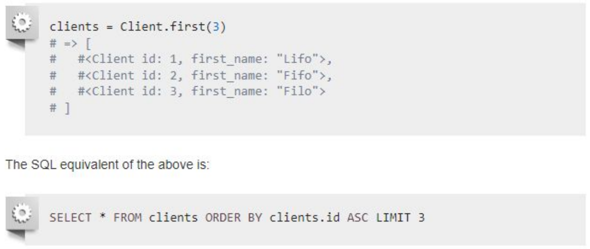
```
class Movie < ActiveRecord::Base
end
```

```
class CreateMovies   ActiveRecord::Migration
  def change
    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
end
```

Models + ActiveRecord (Deep Dive)

Why (and what) the Abstraction? ActiveRecord::Base Inheritance...

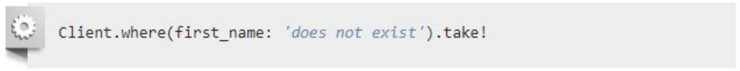
- The directory app/models is expected to contain one Ruby code file per model. The file name is determined by converting the model's name to lower_snake_case, so a file app/models/movie.rb is expected to define the class **`Movie`**.
- The database table name is determined by converting the model's class name to lower_snake_case *and* pluralizing it.
  Ex: **`Class AccountCode => account_codes`**
- The attributes of the model, and their types (string, integer, date, and so on), are inferred from the names and types of the table's columns (Wow!).
- The model class inherits a whole bunch of (static) methods. Examples include :find, :find_by, :create, :new. Full list here. Might be useful for a CHIP later IDK ¯_(ツ)_/¯. They take in a hash that typically matches a attribute in the table.



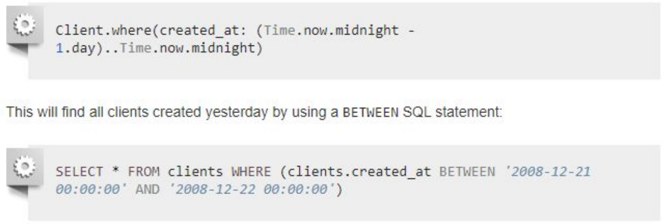
```
clients = Client.first(3)
# => [
#   #<Client id: 1, first_name: "Lifo">,
#   #<Client id: 2, first_name: "Fifo">,
#   #<Client id: 3, first_name: "Filo">
# ]
```

The SQL equivalent of the above is:

```
SELECT * FROM clients ORDER BY clients.id ASC LIMIT 3
```



```
Client.where(first_name: 'does not exist').take!
```



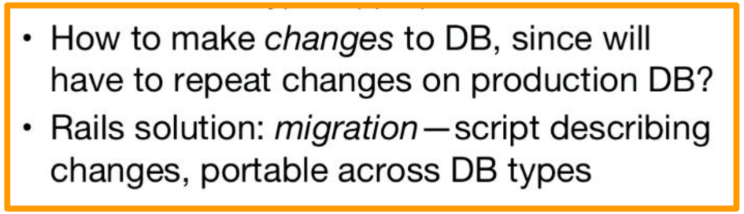
```
Client.where(created_at: (Time.now.midnight -
1.day)..Time.now.midnight)
```

This will find all clients created yesterday by using a BETWEEN SQL statement:

```
SELECT * FROM clients WHERE (clients.created_at BETWEEN '2008-12-21
00:00:00' AND '2008-12-22 00:00:00')
```

Models + ActiveRecord (Deep Dive)

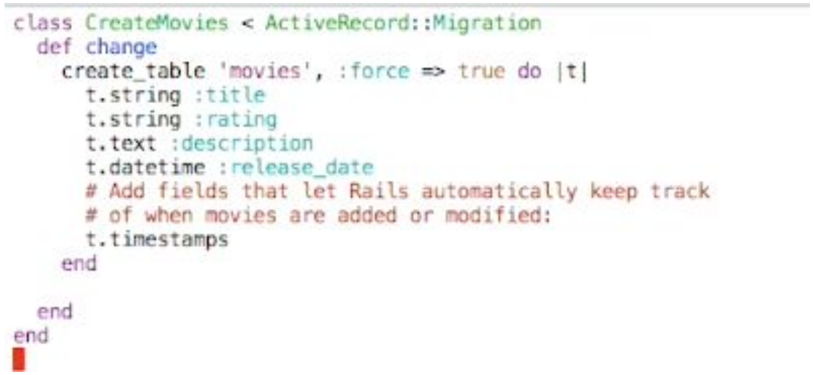
ActiveRecord::Migrations.



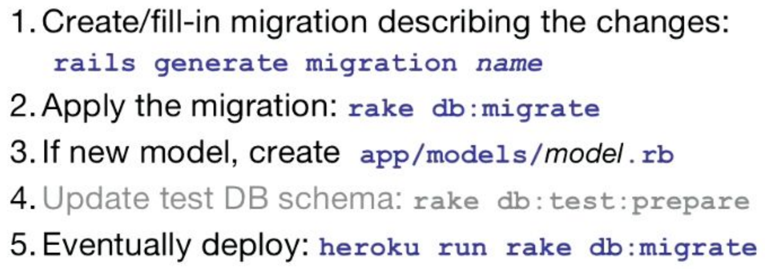
- How to make *changes* to DB, since will have to repeat changes on production DB?
- Rails solution: *migration*—script describing changes, portable across DB types

- Rail uses migrations for making changes to the schema.
- Programmatically apply changes to database
  - Version control
  - Less prone to error, removes human error
  - Automation
  - Reliably repeatable

Specify what to do to the code generator to generate a skeleton code with description of changes to the database:



```
class CreateMovies < ActiveRecord::Migration
  def change
    create_table 'movies', :force => true 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
end
```



1. Create/fill-in migration describing the changes: `rails generate migration` *`name`*
2. Apply the migration: `rake db:migrate`
3. If new model, create `app/models/`*`model`*`.rb`
4. Update test DB schema: `rake db:test:prepare`
5. Eventually deploy: `heroku run rake db:migrate`

Controllers + Routing

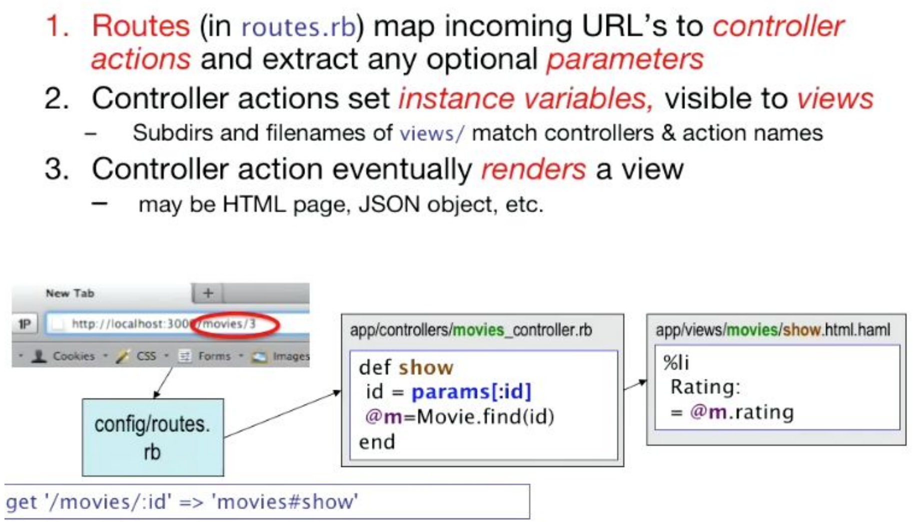
Controllers

- From here we are going to try to understand the life cycle of requests to your ROR server.
- When users interact with a SaaS app via a browser, they’re interacting with views and invoking controller actions, either by typing URIs into their browser (resulting in an HTTP GET) or interacting with page elements that generate GET requests (links) or POST requests (forms).
- Learn how a Single line : resources ’movies’. Defines a complete set of RESTful routes in Rails

Controllers + Routing

Routing (User HTTP Requests) to Controller (Actions)

- Each HTTP request incoming must be mapped to the appropriate controller and action (method). This mapping is called a route.
- Each controller action is handled by a particular Ruby method in a controller file
- In Rails, this lives in the routes.rb file
- Can leverage convention over configuration in Rails for routing or do it manually



1. Routes (in routes.rb) map incoming URL's to *controller actions* and extract any optional *parameters*
2. Controller actions set *instance variables,* visible to *views*
   – Subdirs and filenames of views/ match controllers & action names
3. Controller action eventually *renders* a view
   – may be HTML page, JSON object, etc.

New Tab
http://localhost:3000/movies/3
Cookies
CSS
Forms
Images
config/routes.rb
app/controllers/movies_controller.rb
app/views/movies/show.html.haml

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

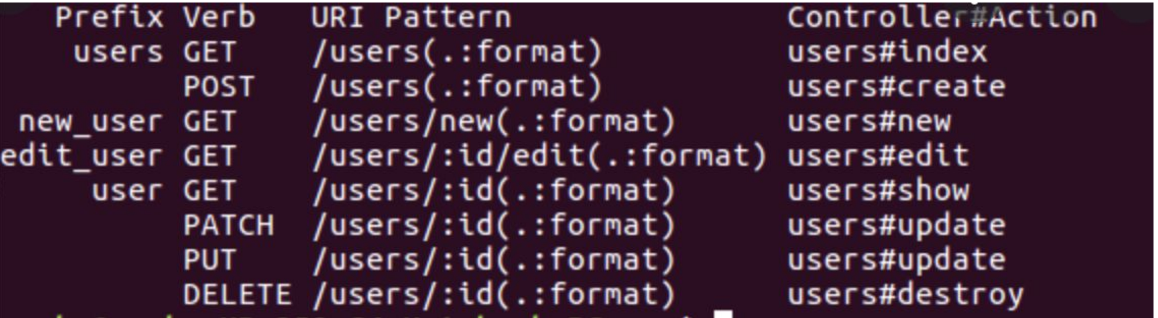
```
%li
 Rating:
 = @m.rating
```

```
get '/movies/:id' => 'movies#show'
```

Controllers + Routing

What do Controllers do?

- Users interact with Saas apps via a browser as they interact with the views, and invoking controller actions either by typing URIs into their browser (HTTP GET) or interacting with page elements that generate GET requests (links) or POST requests (forms)
  - The app receives a request in the form of an HTTP route
    - The app determines what piece of code should be invoked to handle that route



```
   Prefix Verb   URI Pattern               Controller#Action
    users GET    /users(.:format)          users#index
          POST   /users(.:format)          users#create
 new_user GET    /users/new(.:format)      users#new
edit_user GET    /users/:id/edit(.:format) users#edit
     user GET    /users/:id(.:format)      users#show
          PATCH  /users/:id(.:format)      users#update
          PUT    /users/:id(.:format)      users#update
          DELETE /users/:id(.:format)      users#destroy
```

How can we get our CRUD actions??
Where is PUT, PATCH, and DELETE???

Only GET and POST are possible when using HTML links (GET) and forms (GET or POST). The other methods can only be specified using JavaScript.

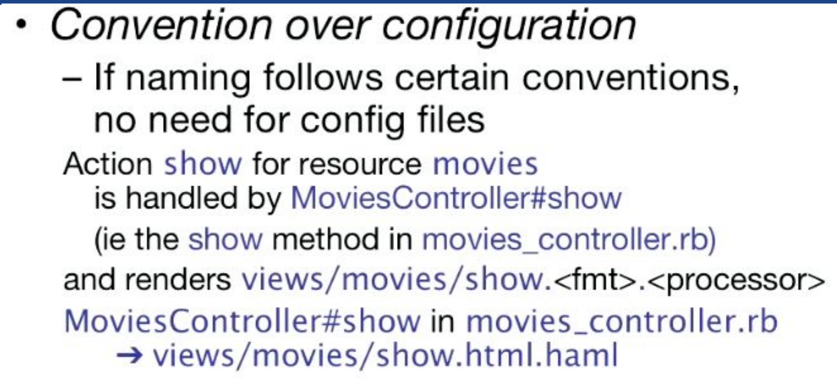
To compensate for the inability of forms/links to specify the other methods, Rails' routing mechanism lets browsers use POST for requests that normally would require PUT or DELETE. Rails annotates the Web forms associated with such requests so that when the request is submitted, Rails internally changes the HTTP method "seen" by the controller to PUT or DELETE as appropriate.

The result is that the Rails programmer can operate under the assumption that PUT and DELETE are actually supported, even though browsers don't implement them. As a result, the same set of routes can handle either requests coming from a browser (that is, from a human being) or requests coming from another service in a SOA.

Controllers + Routing

Controller Methods

- Seven standard controller actions
  - Index
  - new
  - create
  - show
  - edit
  - update
  - Destroy
- Can define more as needed
- Per the MVC architectural pattern, the controller methods may be interacting with models, rendering views, or both, depending on the action
- Each controller action is handled by a particular ruby method within that controller associated with the model resource.



- *Convention over configuration*
  - If naming follows certain conventions, no need for config files

  Action show for resource movies
  is handled by MoviesController#show
  (ie the show method in movies_controller.rb)
  and renders views/movies/show.<fmt>.<processor>

  MoviesController#show in movies_controller.rb
  ➔ views/movies/show.html.haml

Why 7 routes as opposed to just 5 (CRUDI)?

A RESTful request to create a movie would typically include information about the movie itself—title, rating, and so on. But in a user-facing app, we need a way to collect that information interactively from the user, usually by displaying a form the user can fill in. Submitting the form would clearly correspond to the create action, but what route describes displaying the form? The Rails approach is to define a default RESTful route new that displays whatever is necessary to allow collecting information from the user in preparation for a create request.

Controllers + Routing

Controller REST action lifecycle (Generally)

How do we choose what view to render?

Say it with me! **Convention Over ***gasps*** Configuration!** By default Rails will identify and render a view named app/views/model-name/action.html.erb. All instance variables set in the Controller become available in the View. This should be somewhat jarring. Doesn't this violate OOP? Isn't Ruby supposed to be the ultimate OOP language? The short answer; because it's convenient. What actually happens under the hood is that Rails creates an instance of ActionView::Base to handle rendering the view, and then uses Ruby's metaprogramming facilities to "copy" all of the controller's instance variables into that new object.

Collect Request information

Collect the information accompanying the RESTful request: parameters, resource IDs in the URI, and so on

Model Interactions

What model data do you need for validation? Verification? What do you need to save?

Set View Data

Set instance variables for any information that will need to be displayed in the view, such as information retrieved from the database.

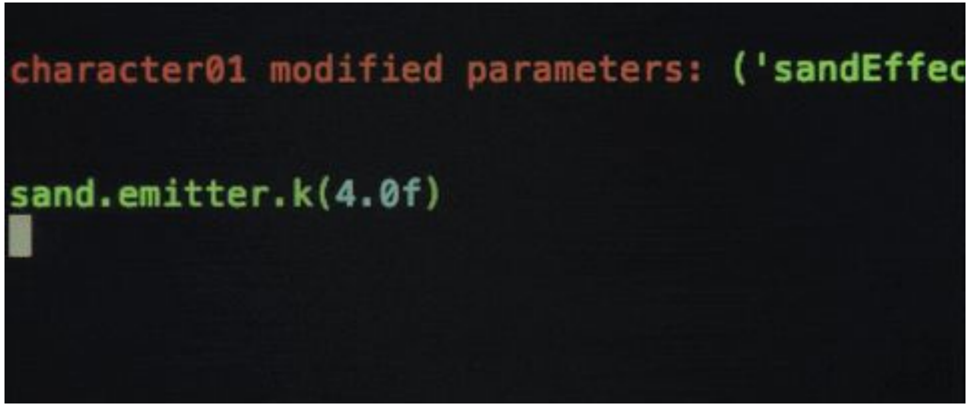
Render View

Render a view that will be returned as the result of the overall request.

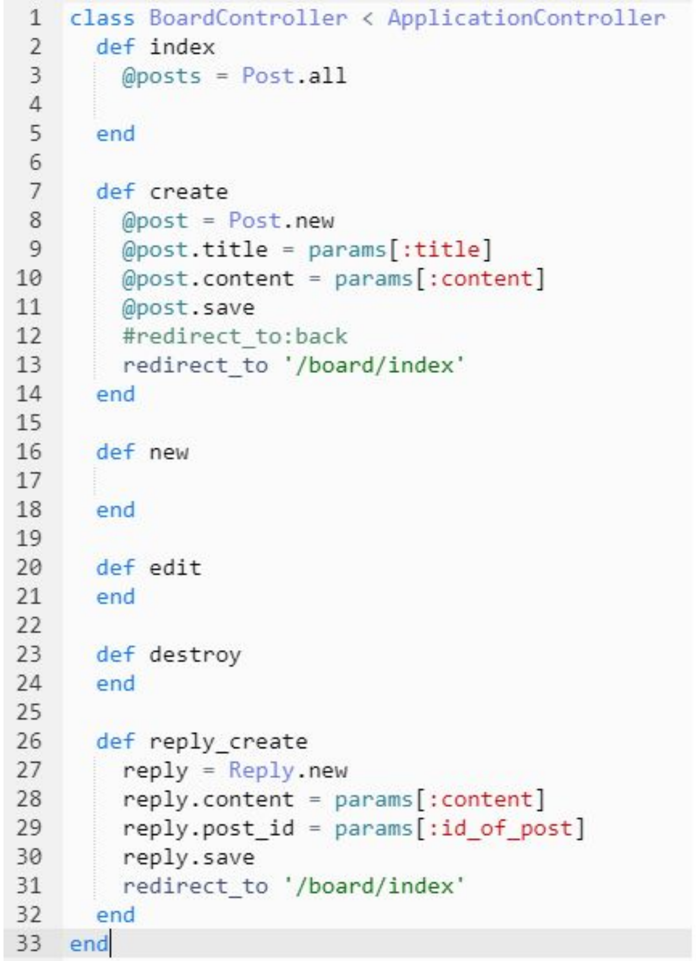
Controllers + Routing

Params[] Hash in Rails

- Parameters in some routes (like id to show a movie instance, or form of new instance attributes being sent as a post request) can be accessed by the Controller using params[] hash.
- Params[:movie] itself is a hash of key-value pairs corresponding to the Movie object’s attributes



```
character01 modified parameters: ('sandEffec

sand.emitter.k(4.0f)
```

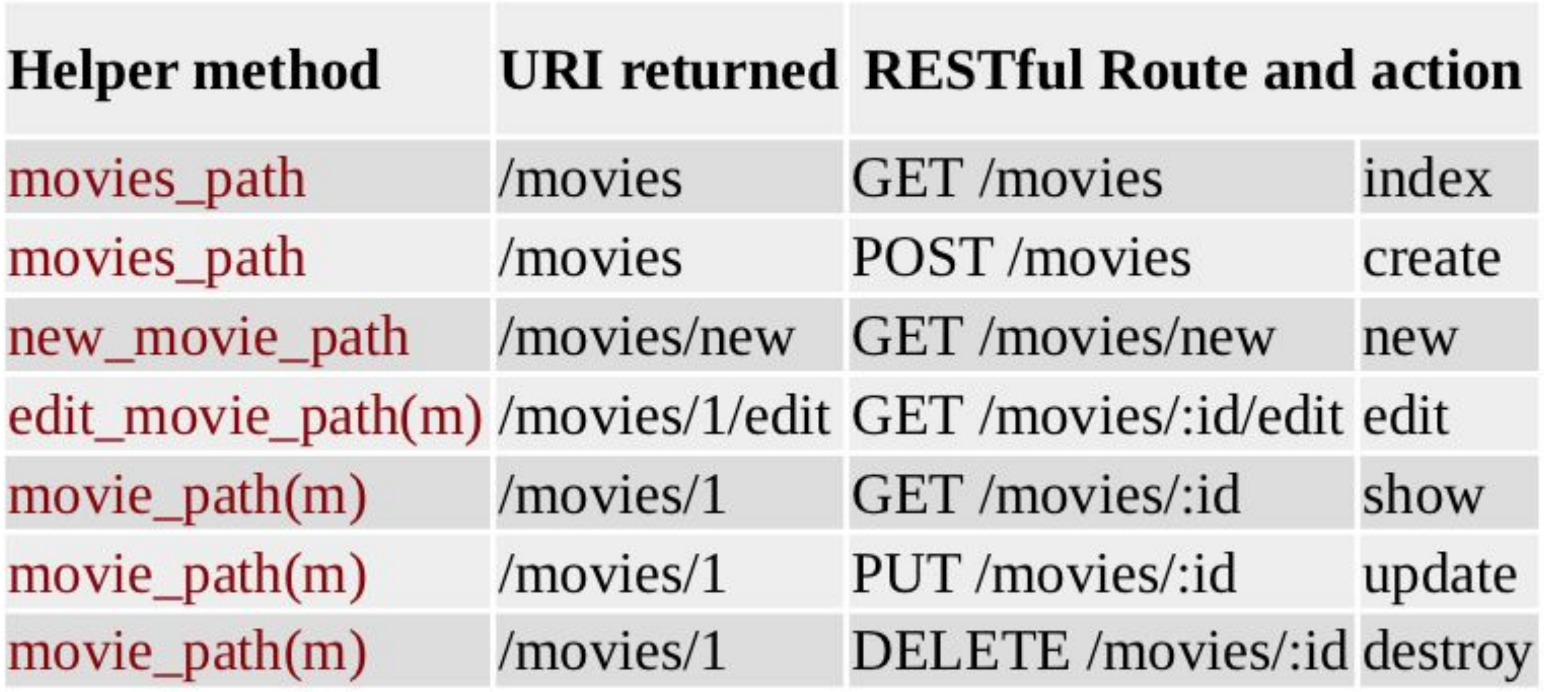


```ruby
class BoardController < ApplicationController
  def index
    @posts = Post.all

  end

  def create
    @post = Post.new
    @post.title = params[:title]
    @post.content = params[:content]
    @post.save
    #redirect_to:back
    redirect_to '/board/index'
  end

  def new

  end

  def edit
  end

  def destroy
  end

  def reply_create
    reply = Reply.new
    reply.content = params[:content]
    reply.post_id = params[:id_of_post]
    reply.save
    redirect_to '/board/index'
  end
end
```

Controllers + Routing

Routing 7 Controller Actions & Helper Methods

Why helper methods? Simple, Abstraction! It will always resolve to the correct URL regardless of how you map your URL’s or what you name them.



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

Forms and user input

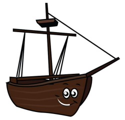
How do we take input from users

So far we’ve looked at views that display data to the user, but not views that collect data from the user. The simplest mechanism for doing so consist of HTML forms.These helpers are Ruby methods that generate HTML form tags whose names follow particular conventions that make them easy to parse by the controller action. In the below example, we are trying to register a new movie. In general manipulating DB entities like this is a common operation in Rails. So obviously, we use convention over configuration to simplify the view to entity update process. Below, the value of params[’movie’] is a hash of movie attribute names and values, which we can pass along directly using Movie.create!(params[’movie’]).

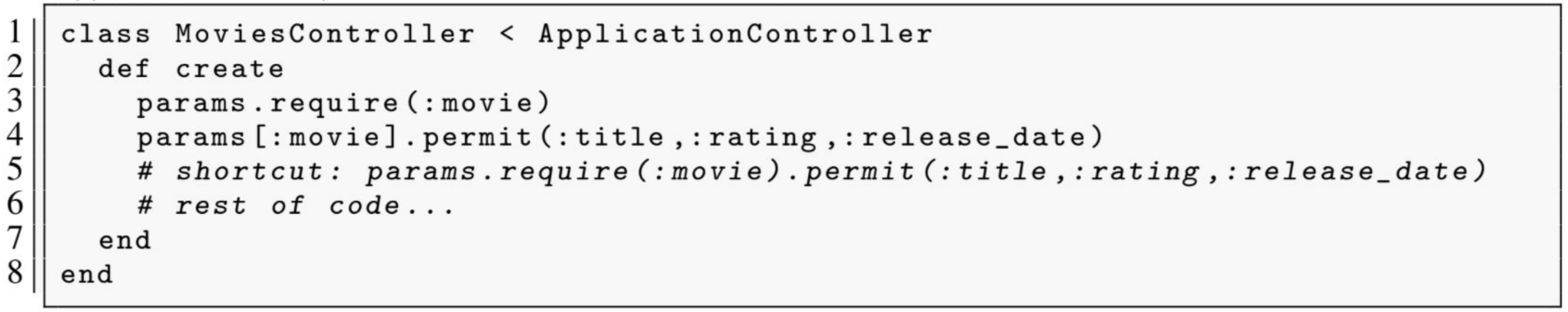
```
<h2>Create New Movie</h2>
<%= form_tag movies_path, :method => :post, :class => 'form' do %>
  <%= label :movie, :title, 'Title', :class => 'form-control' %>
  <%= text_field :movie, :title, :class => 'form-control' %>
  <%= label :movie, :rating, 'Rating', :class => 'form-control' %>
  <%= select :movie, :rating, ['G','PG','PG-13','R','NC-17'], :class => 'form-
      control' %>
  <%= label :movie, :release_date, 'Released On' %>
  <%= date_select :movie, :release_date, :class => 'form-control' %>
  <%= submit_tag 'Save Changes' %>
<% end %>
```

Forms and user input

Side boat, security



We run into a pretty big security problem if we just create on every parameter exposed in the params[‘movie’] hash. This is because an adversary could somewhat easily pollute the hash with other unwanted attributes. To prevent this, we need to follow the **principle of least privilege**. We only want the attribute associated with the movie, otherwise it can damage our DB schema.



```ruby
class MoviesController < ApplicationController
  def create
    params.require(:movie)
    params[:movie].permit(:title,:rating,:release_date)
    # shortcut: params.require(:movie).permit(:title,:rating,:release_date)
    # rest of code...
  end
end
```

Forms and user input

Flash hash (Keeping data between requests)

In the previous example, where we created our new movie object, we could just redirect them to our '/index' action with a friendly success message. Why rewrite the code? One problem. HTTP is stateless. So how can we pass our success message onto the index redirect. For this, Ruby has a special flash hash. The flash hash keeps data from the previous request for 1 more request. Thus we could pass in a flash[:notice] variable and tell the index view to render our success notice. We also have a unique session[] hash where we can store content *forever*. Or until the user clears their browser hash :)

Open question: We now have several tools in our toolbox for passing data around. When would you want to use flash vs params vs session to pass around data?

Kahoot

Attendance

Link: https://tinyurl.com/cs169-disc-4-2021
Password: I-MVC-U