Week 5
Modules: 5

SaaS Framework:
Advanced Programming Abstractions for SaaS

Topics:

- Partials, Validations, and Filters
- Single Sign-On (SSO) and Third-Party Authentication
- Associations, Through-Associations, and RESTful Routes for them

Partials

Partial

- A reusable chunk of a view
- DRY
  - Renders similar content without rewriting it each time in different views
    - Ex) A shared header and footer
- Convention over configuration
  - Names must begin with an underscore.
    - Ex) `_university_entry.html.erb`
  - Syntax to render a partial:
    - Ex) `render partial: 'layouts/footer'`
      - Render this partial file: app/views/layouts/_footer.html.erb
  - Has a local variable with the same name as the partial
    - Ex) The variable `university_entry` in _university_entry.html.erb

Example: Universities

Goal: for our index view, create a table of all the universities in our database.

Attempt 1

```
...
<div class="row">
    <div class="col-6 text-center">Name</div>
    <div class="col-2 text-center">Rank</div>
    <div class="col-4 text-center">Remarks</div>
</div>
<% for university in @universities do %>
<div class="row">
    <div class="col-6 text-center"><%= university.name %></div>
    <div class="col-2 text-center"><%= university.rank %></div>
    <div class="col-4 text-center"><%= university.remarks %></div>
</div>
<% end %>
...
```

Example: Universities

Goal: for our index view, create a table of all the universities in our database.

Attempt 1

- We can use erb and a loop to dynamically create the table with the information we need
- This gives us some degree of DRYness

But what if we need to render multiple subsets of universities in a table throughout our application?

Example: Universities

Standard index operation

```
...
<% for university in @universities do %>
<div class="row">
    <div class="col-6 text-center"><%= university.name %></div>
    <div class="col-2 text-center"><%= university.rank %></div>
    <div class="col-4 text-center"><%= university.remarks %></div>
</div>
<% end %>
...
```

Example: Universities

Page for top 10 universities in the U.S.

```
...
<% for university in @top_universities do %>
<div class=”row”>
    <div class=”col-6 text-center”><%= university.name %></div>
    <div class=”col-2 text-center”><%= university.rank %></div>
    <div class=”col-4 text-center”><%= university.remarks %></div>
</div>
<% end %>
...
```

Example: Universities

Section on show.html.erb with recommended universities.

```
...
<% for university in @similar_universities do %>
<div class="row">
    <div class="col-6 text-center"><%= university.name %></div>
    <div class="col-2 text-center"><%= university.rank %></div>
    <div class="col-4 text-center"><%= university.remarks %></div>
</div>
<% end %>
...
```

Example: Universities

This view functionality may appear multiple times for slightly different reasons.

And we are rewriting the for-loop again and again! This is not DRY at all.

How can we avoid rewriting it everywhere?

Example: Universities

Attempt 2

Extract the common view subsection into `_university_entry.html.erb`:

```
<div class="row">
    <div class="col-6 text-left"><%= university_entry.name %></div>
    <div class="col-2 text-center"><%= university_entry.rank %></div>
    <div class="col-4 text-left"><%= university_entry.remarks %></div>
</div>
```

Notice the filename begins with an underscore. Convention over configuration.

Every partial has a local variable with the same name as the partial (excluding the leading underscore) that we can set when we tell Rails to render the partial.

Example: Universities

Attempt 2

Render the partial `_university_entry.html.erb` for every entry in our collection. In `index.html.erb`

```
...
  <%= render partial: 'university_entry', collection: @universities %>
...
```

Now, for every university in our universities collection that was set in the controller, we will render that partial view.

The particular university we’re on will be assigned to the `university_entry` local variable by convention.

There are other ways to pass info to partials -- read the docs if you’re interested.

Example: Universities

Notice this scales really well.

In our top 10 universities page view:

```
...
  <%= render partial: ‘university_entry’, collection: @top_universities %>
...
```

In our recommended universities page view:

```
...
  <%= render partial: ‘university_entry’, collection: @similar_universities %>
...
```

Example: Universities

Conclusion

- While we *can* iterate in views and dynamically generate view content based on our collections, partial views may be a simpler and cleaner solution.

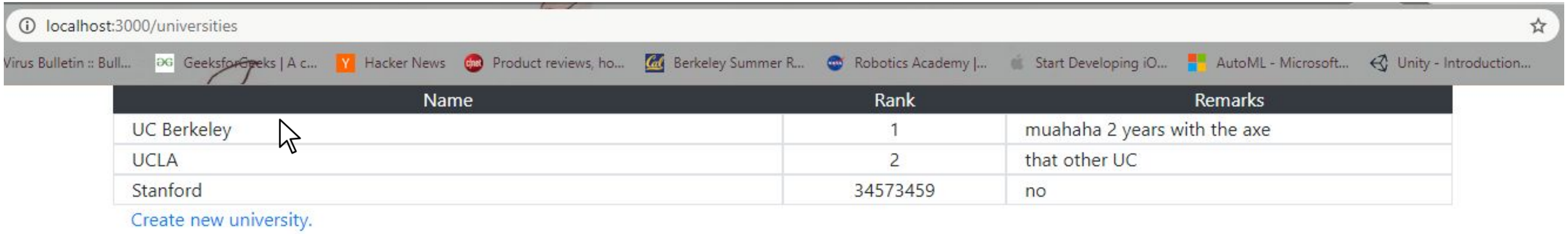
- The example only saved us a few lines per page, but with more complicated view content, such as navbars, tables with complicated conditional logic, etc., the savings can be huge.

Validations

Validation

- Enforce validity constraints on model
- DRY
  - Common to enforce certain validity constraints on a given type of model object
    - Ex) For `University`, its `name` cannot be `nil`
- Convention over configuration
  - Validation checks are triggered when you call the instance method `valid?` or when you try to save the model to the database
    - Ex) `create, save, update`
  - Validation errors are recorded in the `ActiveModel::Errors`
    - Ex) `university.errors[:name] # => [“cannot be nil”]`
  - Tons of built-in validation functions. We can also define custom ones.
    - Ex) `validates :name, :presence => true`

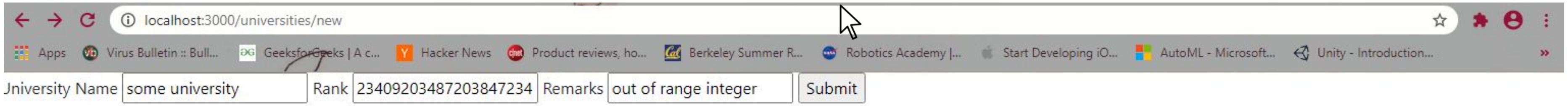
Model Validation



Name	Rank	Remarks
UC Berkeley	1	muahaha 2 years with the axe
UCLA	2	that other UC
Stanford	34573459	no

Create new university.

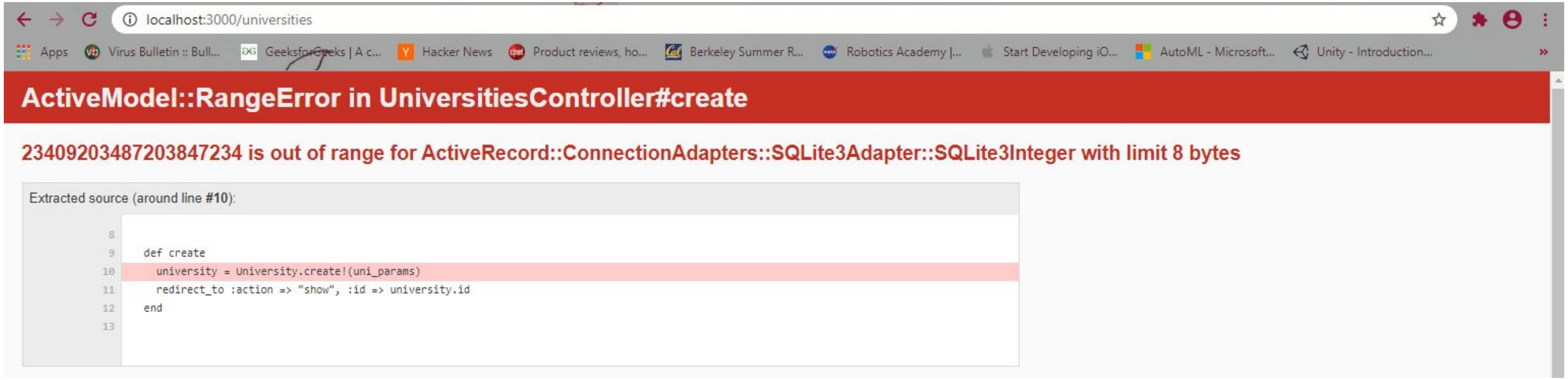
Model Validation


localhost:3000/universities/new

Apps | Virus Bulletin :: Bull... | GeeksforGeeks | A c... | Hacker News | Product reviews, ho... | Berkeley Summer R... | Robotics Academy |... | Start Developing iO... | AutoML - Microsoft... | Unity - Introduction...

Jniversity Name some university Rank 2340920348720384723 Remarks out of range integer Submit

Model Validation


localhost:3000/universities

Apps | Virus Bulletin :: Bull... | GeeksforGeeks | A c... | Hacker News | Product reviews, ho... | Berkeley Summer R... | Robotics Academy |... | Start Developing iO... | AutoML - Microsoft... | Unity - Introduction...

ActiveModel::RangeError in UniversitiesController#create

234092034872038447234 is out of range for ActiveRecord::ConnectionAdapters::SQLite3Adapter::SQLite3Integer with limit 8 bytes

Extracted source (around line #10):

```

  def create
    university = University.create!(uni_params)
    redirect_to :action => "show", :id => university.id
  end

```

Model Validation

Model Validation

- Users will be frustrated because they do not know what this means!

- Rails has built-in functionality for validating model parameters before creating and saving instances to the database
- We can also define custom validation functions that perform checks for us in ways specific to our application logic/structure

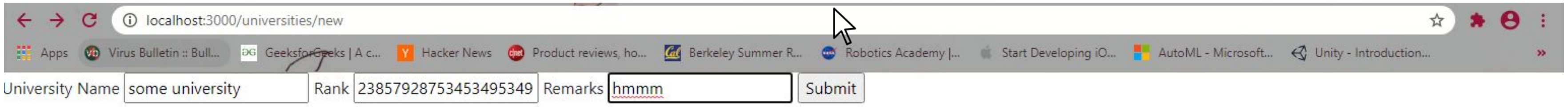
- Trigger validation check manually via call to `valid?` or `invalid?`
- Certain AR methods run validation checks under the hood:
  - `create / create!`
  - `save / save!`
  - `update / update!`

Example: Universities

```
class University < ApplicationRecord
  validates :name, :presence => true
  validate :correct_rank_provided

  def correct_rank_provided
    fixed_num_max = 2 ** (0.size * 8 - 2) - 1

    if rank > fixed_num_max || rank <= 0
      errors.add(:rank, ‘must be a valid positive integer’)
    end

    ...
  end

  ...
end
```

Example: Universities

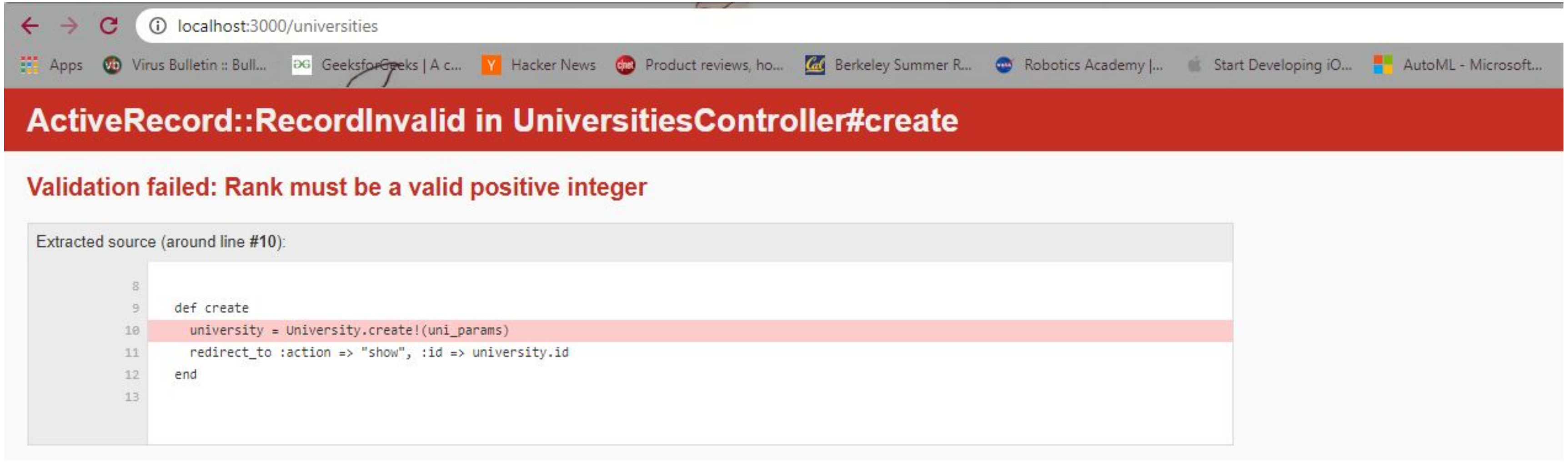
- Okay, so we've included some validations. What happens now?



University Name some university Rank 23857928753453495349 Remarks hmmm

Example: Universities

- Okay, so we've included some validations. What happens now?



ActiveRecord::RecordInvalid in UniversitiesController#create

Validation failed: Rank must be a valid positive integer

Extracted source (around line #10):

```

def create
  university = University.create!(uni_params)
  redirect_to :action => "show", :id => university.id
end

```

Notice the error message is now coming from the validation we just wrote.

Example: Universities

- “Wait, we still got an error message. What did we gain?”
- Here is the `create` controller method we’ve been using:

```
...
def create
  university = University.create!(uni_params)
  redirect_to :action => “show”, :id => university.id
end
...
```

Example: Universities

- The `create!` method (notice the exclamation mark) throws an error on failure (such as before, where our input was invalid)
- Here we could implement some error handling or use the `create` method (no exclamation point), which returns false on failure.
- Notice that we'll need to think explicitly about what to do if validation fails. We can't just remove the ! below, because `university.id` would become `null`, throwing a routing error.

```
...
def create
  university = University.create!(uni_params)
  redirect_to :action => “show”, :id => university.id
end
...
```

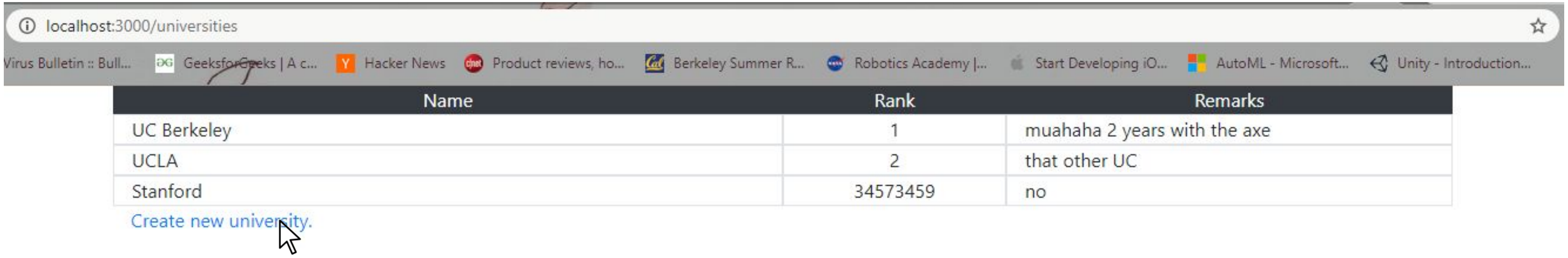
Example: Universities

- New attempt:

```
...
def create
  begin
    university = University.create!(uni_params)
    redirect_to :action => “show”, :id => university.id
  rescue ActiveRecord::RecordInvalid
    flash[:alert] = “Please input a valid university!”
    redirect_to :action => “new”
  end
end
...
```

Example: Universities

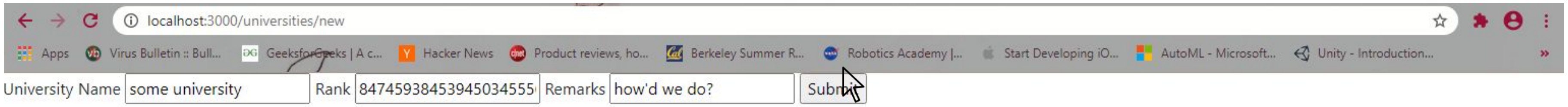
- Where does this get us?


localhost:3000/universities

Name	Rank	Remarks
UC Berkeley	1	muahaha 2 years with the axe
UCLA	2	that other UC
Stanford	34573459	no

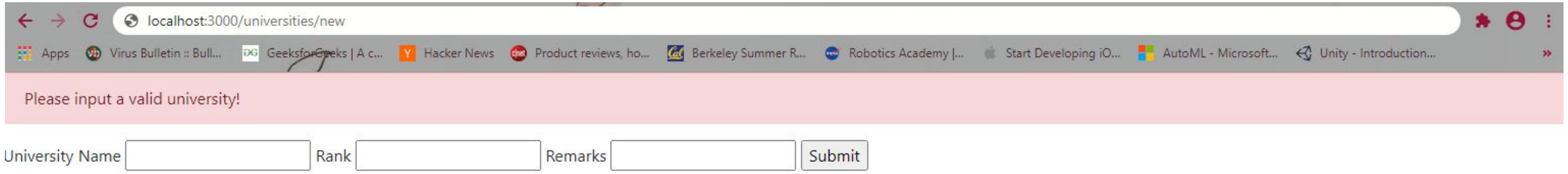
Create new university.

Example: Universities


localhost:3000/universities/new

University Name some university Rank 84745938453945034555 Remarks how'd we do? Submit

Example: Universities

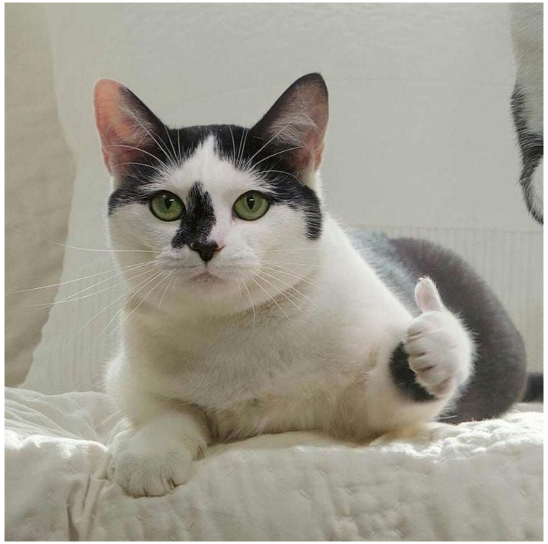


Please input a valid university!

University Name Rank Remarks Submit

Example: Universities

- We prevent the app from erroring out by rescuing the error and putting up an appropriate alert.



Example: Universities

- Extra note: we didn’t have to use the `create` / `create!` methods. An alternative, and perhaps cleaner, implementation might make use of the `new`, `valid?`, and `save` methods.

```
...
def create
  university = University.new(uni_params)
  if university.valid?
    university.save!
    redirect_to :action => “show”, :id => university.id
  else
    if !university.errors[:rank].empty?
      flash[:alert] = “Rank “ + university.errors[:rank].first
    else # missing required name
      flash[:alert] = “Name “ + university.errors[:name].first
    end
    redirect_to :action => “new”
  end
end
...
```

Example: Universities

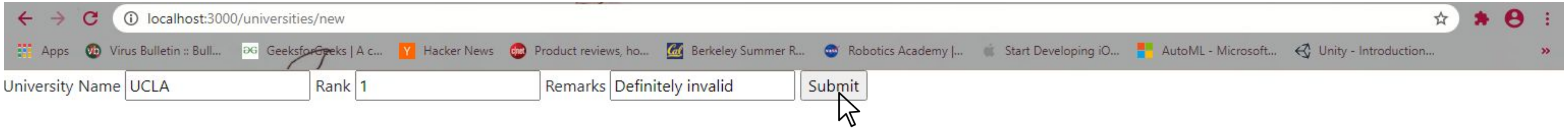
- But what is `university.errors`?
- Here’s our validation function:

```
def correct_rank_provided
  fixed_num_max = 2 ** (0.size * 8 - 2) - 1
  cal_names = [“UCB”, “Cal”, “UC Berkeley”]

  if !cal_names.include?(name) and rank == 1
      errors.add(:rank, ‘must not be 1 if university is not Cal’)
    elsif cal_names.include?(name) and rank != 1
      errors.add(:rank, ‘must be 1 if Cal is provided’)
    elsif rank > fixed_num_max || rank <= 0
      errors.add(:rank, ‘must be a valid positive integer’)
  end
end
```

- We can add errors as we see fit when performing model validation, and access them later in the controller like we were just doing. This also allows finer alerts.

Example: Universities

- Finer error logging after what we wrote in our controller

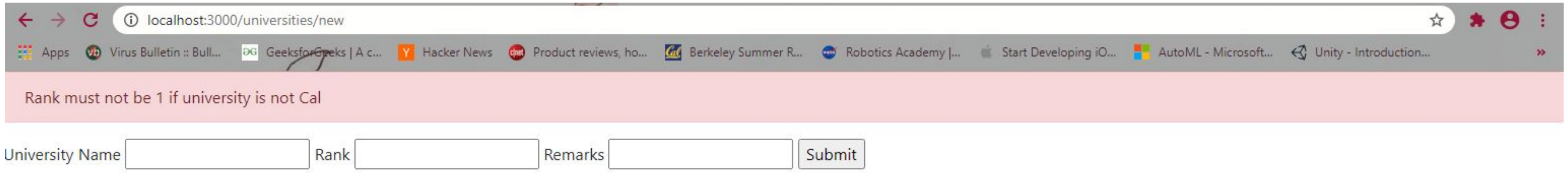


localhost:3000/universities/new

University Name	UCLA	Rank	1	Remarks	Definitely invalid

Example: Universities

- Finer error logging after what we wrote in our controller


localhost:3000/universities/new

Rank must not be 1 if university is not Cal

The error we set in the validation functions persist!

Built-in Validators

- Remember: the top of our model, where we specified our validations, looked like this:

```
class University < ApplicationRecord
  validates :name, :presence => true
  validate :correct_rank_provided
  ...
```

- We defined the `correct_rank_provided` function ourselves, but there are built-in validators as well
- We use Rails' built-in functionality to make the name required for any valid university record (`:presence`)

Built-in Validators

There are various other options that Rails provides. Read the documentation if you’re interested in learning more.

- `exclusion`
- `inclusion`
- `length`
- `presence`
- `absence`
- `uniqueness`
- `...`

Filters

Filters

- Checks whether certain conditions are true before a controller action is run
- DRY
  - Similar to validations, it is common to ensure certain conditions are met before running an action.
    - Ex) A user needs to be logged in before paying for a product.
- Convention over Configuration
  - Syntax
    - Ex) `before_action :is_admin?, except: [:show, :index]`
    - `is_admin?` will be run for every action except `show` and `index`, meaning regular users can only run `show` and `index`.

Example: Universities

- Let’s pretend our university catalogue web service is publicly *readable*, but only writable by valid users (i.e. our admins). Here’s how we might use filters to do that:

```
class UniversitiesController < ApplicationController
  before_action :is_admin?, except: [:show, :index]

  def is_admin?
    unless logged_in? # depends on how we implement users/auth
      flash[:alert] = “Administrator access required.”
    redirect_to universities_path
    end
  end
  ...
end
```

Example: Universities

- With this, the `is_admin?` function will be run before any incoming controller actions except for show and index.

```
class UniversitiesController < ApplicationController
  before_action :is_admin?, except: [:show, :index]

  def is_admin?
    unless logged_in? # depends on how we implement users/auth
      flash[:alert] = “Administrator access required.”
    redirect_to universities_path
    end
  end
  ...
end
```

Filters

Other methods include:

- `skip_before_action`
- `around_action`
- `prepend_before_action`
- `append_after_action`
- `after_action`
- `before_action`
- … and more.

Note you might see some of these methods with “filter” in place of “action” -- “filter” was used in the method names in earlier versions of Rails.

Question #1

Partials are used in _____. Validations are used in _____. Filters are used in _____.

A) Controllers, Models, Views
B) Models, Controllers, Views
C) Models, Views, Controllers
D) Views, Models, Controllers
E) Views, Controllers, Models

Single Sign-On (SSO)
and Third-Party Authentication

SSO

- Users don’t want to have separate usernames and passwords for each site.
- SSO: The credentials established for one site (the provider) can be used to sign in to other sites that are administratively unrelated to it.
- DRY
  - It is pretty common for SaaS apps to need to authenticate users.

SSO

- Users don’t want to have separate usernames and passwords for each site.
- SSO: The credentials established for one site (the provider) can be used to sign in to other sites that are administratively unrelated to it.
- DRY
  - It is pretty common for SaaS apps to need to authenticate users.
- Problem: Users may not want to share their usernames and passwords to other sites!

Third-Party Authentication

- Third-party authentication: Users don’t reveal the credentials directly but uses a third-party to log in.

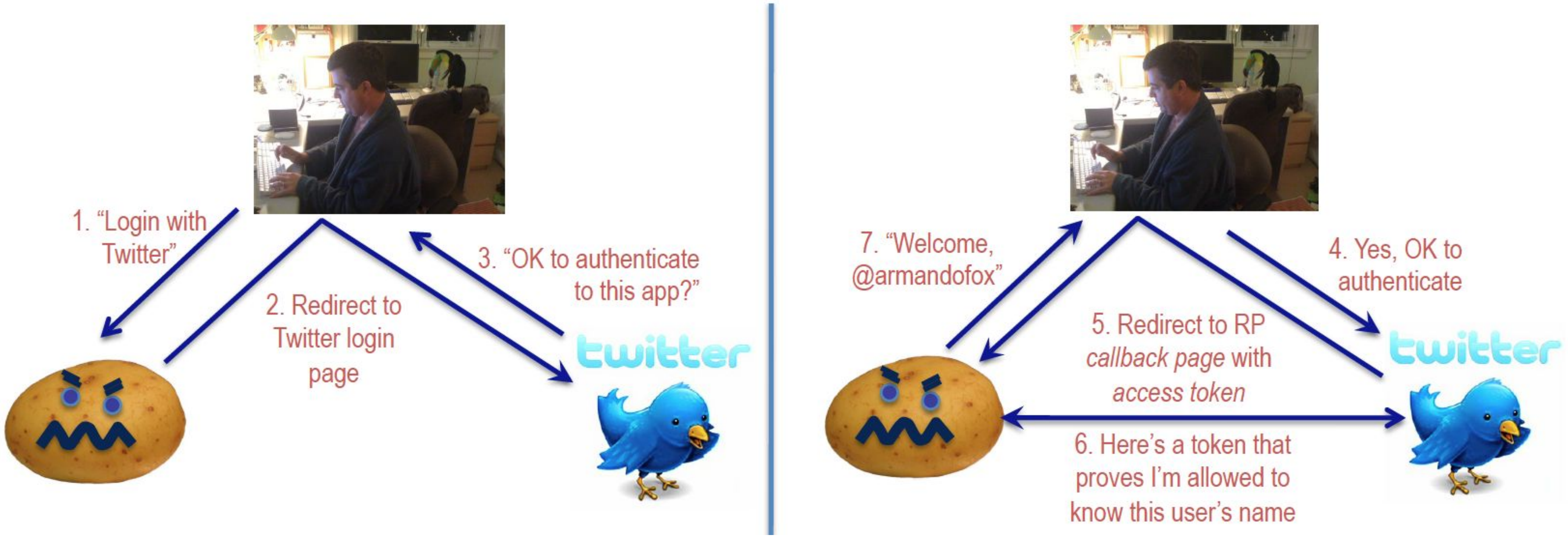
  That is, if site A wants to authenticate a user via site B, the user doesn’t provide its username and password of site B to site A directly. Instead, site A will send the user the login page of site B. When the user logs into site B, site B will provide a proof to site A, and site A will check this proof to authenticate the user.

  If users allow, the providers may share additional information.

- DRY
  - Service-oriented architecture: Developers can simply use the authentication service provided by third parties for their own users.
    - Ex) Google, Facebook, etc.

- In Rails, we can use the gem OmniAuth (see textbook for details)

Third-Party Authentication



1. "Login with Twitter"
2. Redirect to Twitter login page
3. "OK to authenticate to this app?"
twitter
7. "Welcome, @armandofox"
4. Yes, OK to authenticate
5. Redirect to RP *callback page* with *access token*
twitter
6. Here's a token that proves I'm allowed to know this user's name

Question #2

If you log in to a SaaS app using your Facebook account, that app must be able to send a post on your timeline.

A. True
B. False

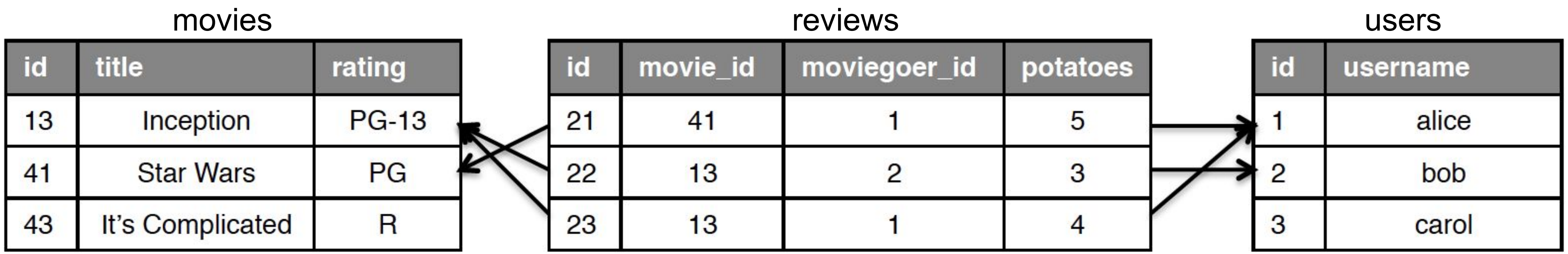
Associations, Through-Associations, and RESTful Routes for Them

Foreign Key

- In SQL, a foreign key is a column in one table whose job is to reference the primary key of another table to establish an association between two tables. We need to join two tables to find out the attributes of the associated table.
  Ex) Find all reviews for Star Wars

```
SELECT reviews.*
  FROM movies JOIN reviews ON movies.id=reviews.movie_id
  WHERE movies.title = “Star Wars”;
```

- In ActiveRecord, we do not need to use the foreign key explicitly or join to reference the associated object.
  - Ex) `star_wars = Movie.where(:title => “Star Wars”)`
    `star_wars.reviews`



movies

id	title	rating
13	Inception	PG-13
41	Star Wars	PG
43	It’s Complicated	R

reviews

id	movie_id	moviegoer_id	potatoes
21	41	1	5
22	13	2	3
23	13	1	4

users

id	username
1	alice
2	bob
3	carol

Associations

- Used to create a relationship between two resources / models
- Types of associations
  - `belongs_to`, `has_one`, `has_many`, `has_and_belongs_to_many`
  - `:through` option
- Through association
  - Used to represent an *indirect* relationship between two resources / models

Example: Universities

- A university `has_many` professors. (Assume a `Professor` is a model in our app.)
- Note this requires minor modifications to our schema as well.

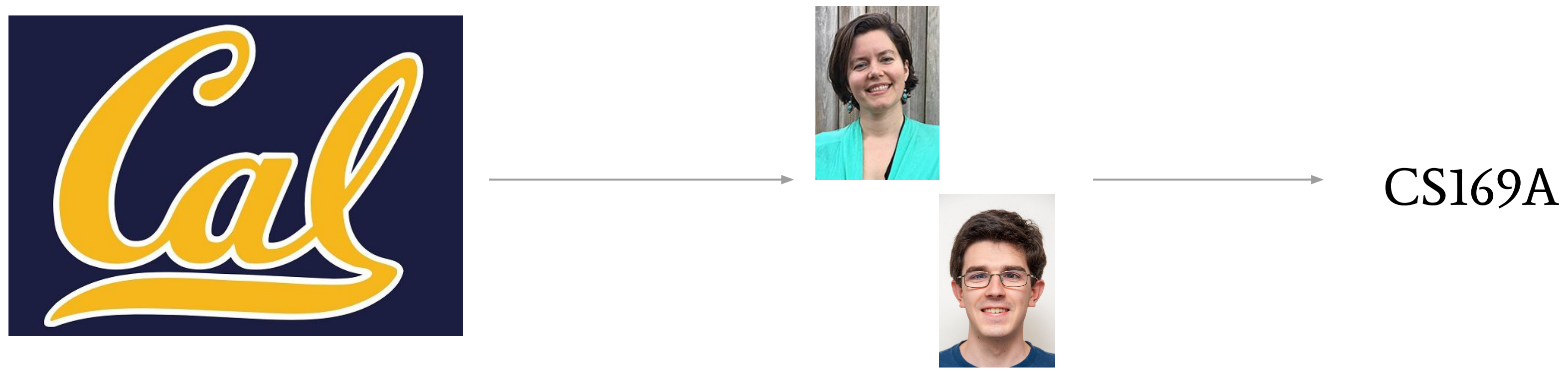
```
class University < ActiveRecord::Base do
  has_many :professors
  ...
  ... # validations, helper functions etc.
end
```

Allows us to do things like:

```
>>> prof_fox = Professor.create(:name => ”Pamela Fox”)
>>> prof_ball = Professor.create(:name => “Michael Ball”)
>>> Cal = University.find(1)
>>> Cal.professors << prof_fox
>>> Cal.professors << prof_ball
>>> Cal.save!
```

Through Associations

- Sometimes two resources have a relationship *through* a mutual association
- If a university has many professors, and a professor teaches many classes, then a university has many classes *through* its professors.



Example: Universities

- A university `has_many` professors and courses. (Assume `Professor` and `Course` are models in our app.) Note this requires minor modifications to our schema as well.

```
class University < ActiveRecord::Base do
  has_many :professors
  has_many :courses, through: :professors, :uniq => true
  ...
  ... # validations, helper functions etc.
end
```

Allows us to do things like:

```
>>> CS169A = Course.create(:name => “CS169A”)
>>> prof_ball.courses << CS169A
>>> prof_fox.courses << CS169A
>>> puts CS169A == Cal.courses.first #instead Cal.professors.all {|p| for
p.courses}
true
```

Associations and RESTful Routing

- Note that now, we may interact with several related resources in a single operation.
- To be as RESTful as possible in our API design, we can actually embed these resources and relationships into our routes

```
resources :universities do
  resources :professors
  ...
end
```

HTTP Method	Route	Description
GET	/universities/:id/professors	Display professors for a specific uni
GET	/universities/:id/professors/new	Form for a new prof for a given uni
POST	/universities/:id/professors	Add new prof to a given uni
GET	/universities/:id/professors/:id	Display a specific prof at a specific uni
⋮	⋮	⋮

Attendance Form
tinyurl.com/cs169a-dis-5

DDDRRRYYY