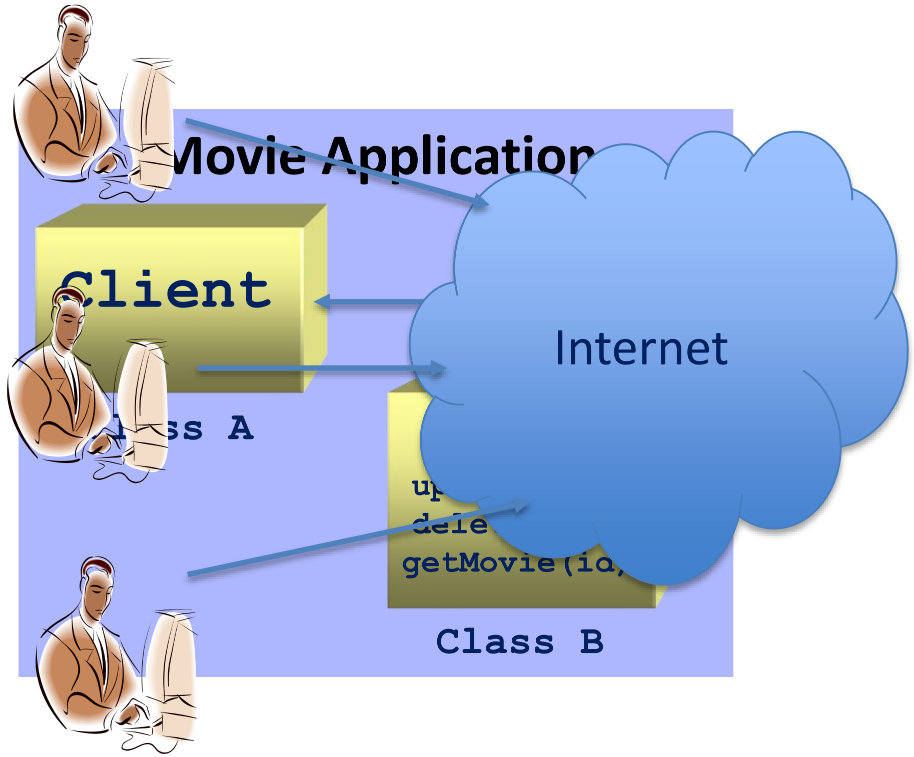
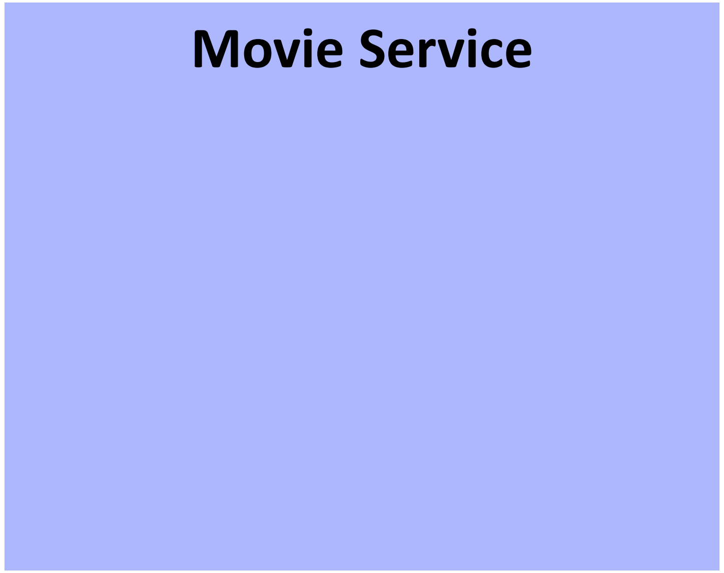
In this lecture, we will discuss…

- What a web service is
- Comparisons between web services and web applications
- Popular web applications

What Are Web Services?





Web Service: Introduction

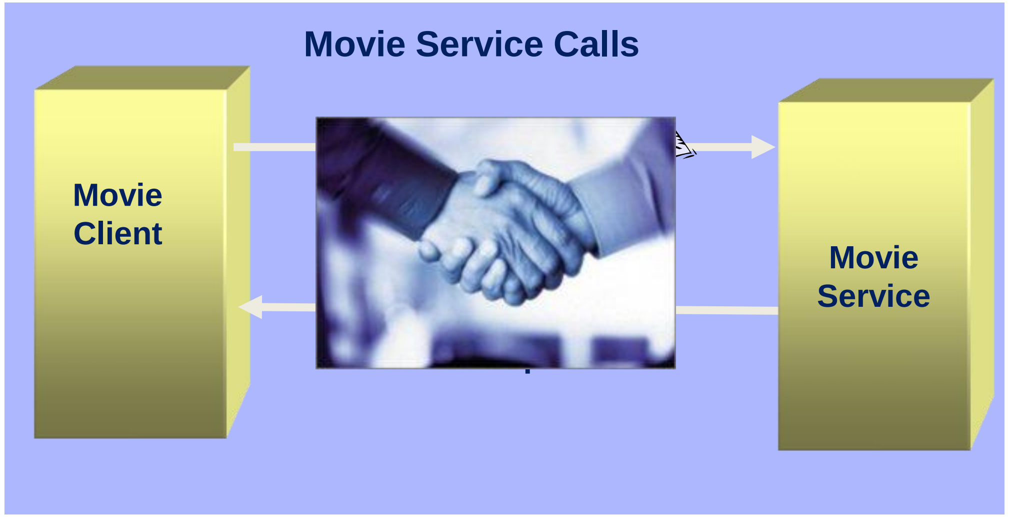
- Web Service - *“software that makes services available on a network using technologies such as XML/JSON and HTTP”*
- Goal is interoperability between enterprises
- Service Oriented - distributed collections of smaller, loosely coupled service providers

Web Services vs. Web Application

Web Services	Web Application
XML/JSON	HTML
Program to program interaction	User to program interaction
CRUD based API	User Interface
Possibility of service integration	Monolithic services

Web Service: Basic Example

- Get movie info from movie service



Web Service Users (REST)

Summary

- A web service is just an endpoint (interface) for a “consumer” to request data
- Consumer is typically an application unlike a web application where the consumer is a web browser (human)

What’s Next?

- REST Web Services

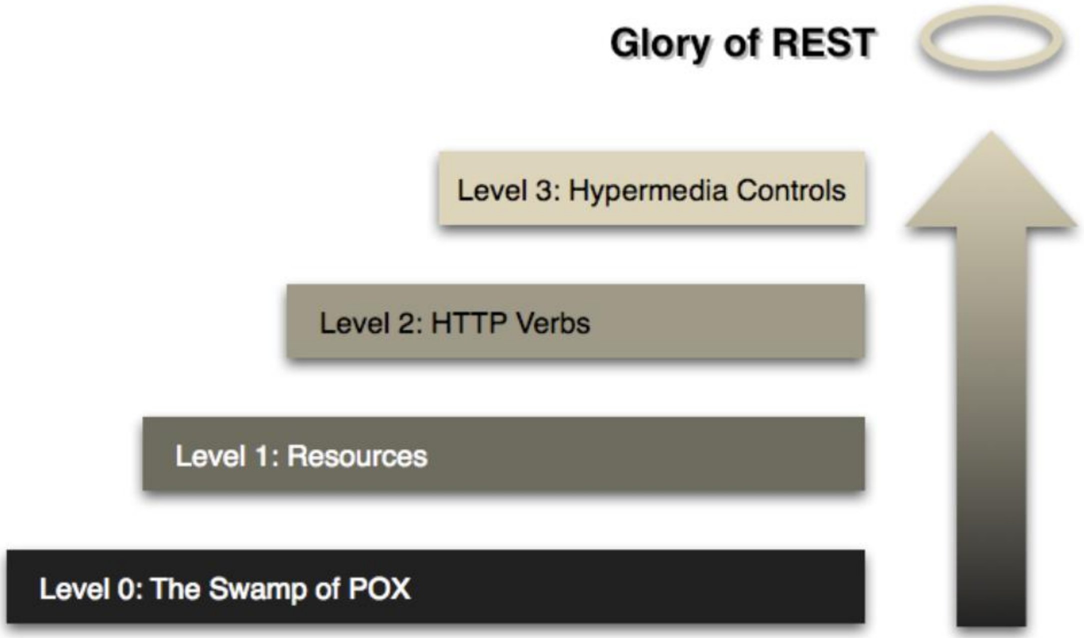
In this lecture, we will discuss…

- ✧ REST Introduction
- ✧ RESTful Services: Design Principles

REST: Introduction

- ✧ **RE**presentational **S**tate **T**ransfer
- ✧ Resource Instance(s) are identified by URI (Uniform Resource Indicator)
  - http://www.movieservice.com/movie/:id
  - http://www.movieservice.com/movie/12345
- ✧ Introduced by Roy Fielding in 2000

“Glory of REST”: Richardson Maturity Model



Source: http://martinfowler.com/articles/richardsonMaturityModel.html

REST: Web Services

- ✧ Stateless
- ✧ Expose directory structure-like URIs
- ✧ Supports multiple formats but JSON/XML – most popular formats.

Representations

- Represents a resource (Movie)
- A resource can contain other resources (Movie → Roles)
- Representation does not restrict representation format – XML/JSON
- JSON is ideal for web pages (RoR/Ajax)

HTTP Protocol

- ✧ GET - retrieve a resource
- ✧ POST - create a resource
- ✧ PATCH – update partial resource
- ✧ PUT - change the state of a source or to update it
- ✧ DELETE - remove a resource
- ✧ HEAD – similar to GET but no message body

Stateless

- ✧ Stateful
  - /movies/getNextPage
  - server needs to store previous page
- ✧ Stateless
  - /movies?offset=25&limit=4
  - /movies?page=3

Uniform Resource Indicator (URI)

http://www.movieservice.com/movies/12345

http://www.movieservice.com/movies/12345/roles

http://www.movieservice.com/movies/12345/roles/100

- Lower case
- Underlying technology can change

Resource Representations

Common MIME Types

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
HTML (XHTML)	application/xhtml

✧ Custom Type - application/vnd+company.category+xml

Summary

- ✧ HTTP-based web - predominant WS design model
- ✧ Simplicity - Most are in the level2 to 3 level
- ✧ "Truly RESTful" services only when you - add solid support for state, links to the use of URIs, methods, and exchangeable content

What’s Next?

- ✧ REST Web Services - Resources

In this lecture, we will discuss…

- Resources - Standalone and Dependent
- Using `rails` to build resources
- Example Resources
  - `Movie`
  - `Actor`
  - `MovieRole`

Resource Scope

- ✧ Resource - fundamental concept in any RESTful API
  - *is an object with a type, associated data, relationships to other resources, and a set of methods that operate on it.*

- ✧ Example Resources
  - Movies
  - Actors
  - MovieRoles

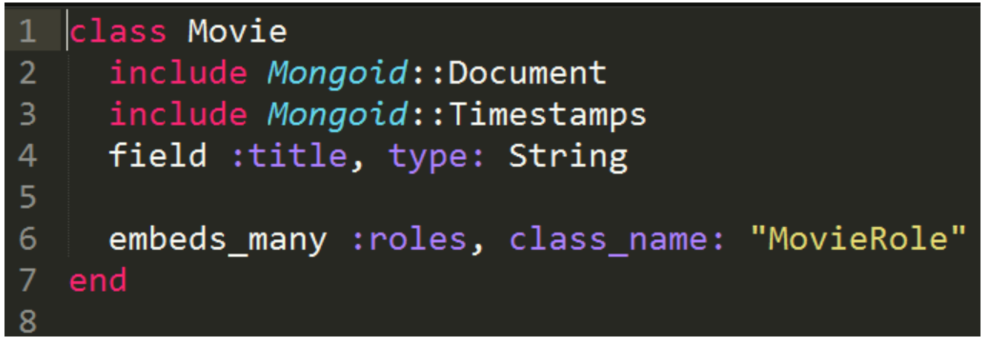
Resources

- Standalone Resources
  - Movies – can exist without Actors or MovieRoles
  - Actors – can exist without Movies or MovieRoles
- Dependent Resources
  - MovieRole
    - Depends on Movies to exist
    - Related to Actor, but can exist if relationship is severed

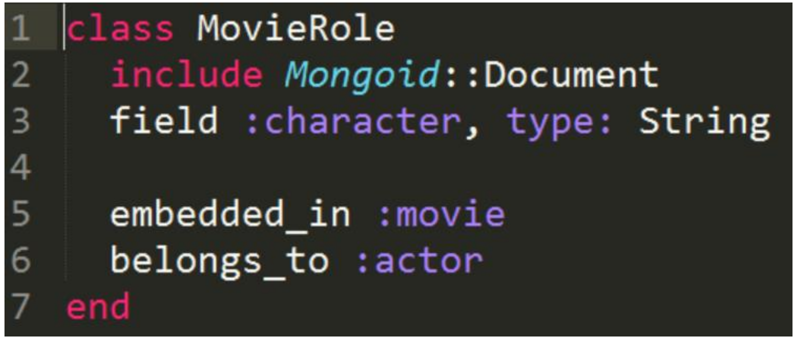
Rails - Resources

- ✧ `rails g scaffold` command
  - build templated code for CRUD operations
  - Mongoid or ActiveModel – additional implementation
- ✧ `rails g model Movie title`
- ✧ `rails g model Actor name`
- ✧ `rails g model MovieRole character`

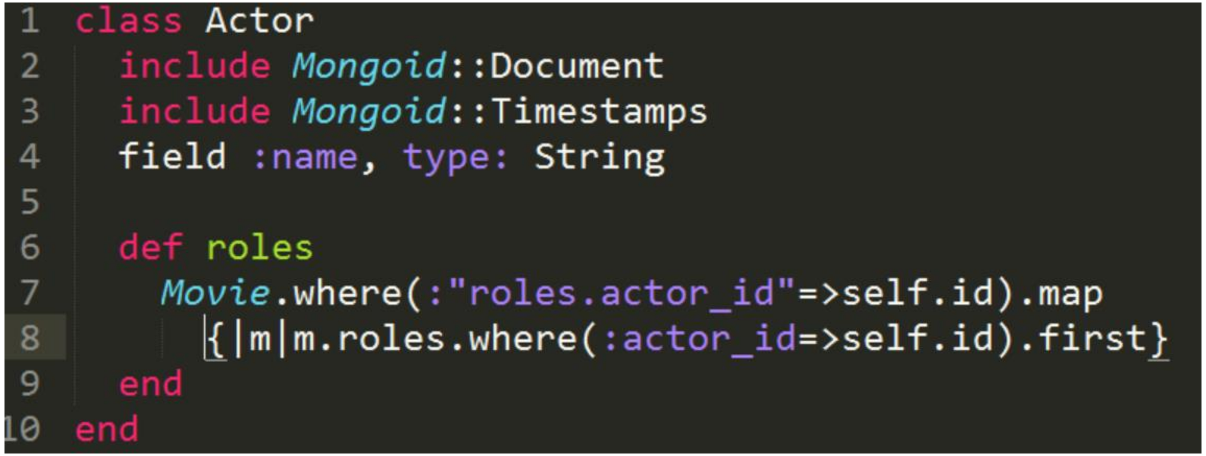
Model Classes



```ruby
class Movie
  include Mongoid::Document
  include Mongoid::Timestamps
  field :title, type: String

  embeds_many :roles, class_name: "MovieRole"
end

```



```ruby
class MovieRole
  include Mongoid::Document
  field :character, type: String

  embedded_in :movie
  belongs_to :actor
end
```



```ruby
class Actor
  include Mongoid::Document
  include Mongoid::Timestamps
  field :name, type: String

  def roles
    Movie.where(:"roles.actor_id"=>self.id).map
      {|m|m.roles.where(:actor_id=>self.id).first}
  end
end
```

Summary

- Generated Model Classes
- Used the ORM to add dependency and relationship details
- Perform basic CRUD on these resources
- Ability to add more features

What’s Next?

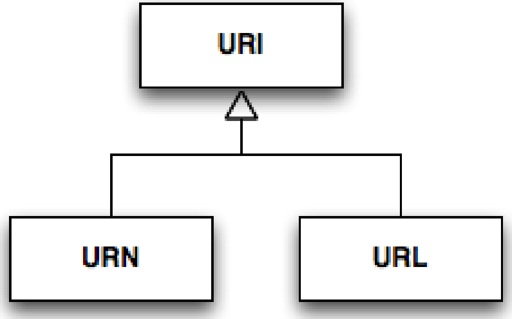
- URIs

In this lecture, we will discuss…

- URI – Uniform Resource Indicator
- Exposing resource as URI
- `rake routes`
- httparty

URI vs. URL vs. URN

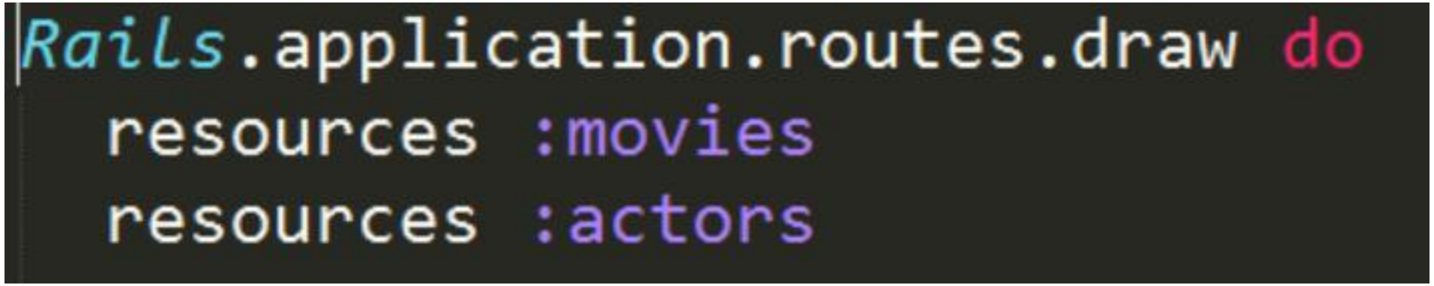
- A **Uniform Resource Identifier** (URI)
  - string of characters which identifies an Internet resource
  - www.coursera.org
- A **Uniform Resource Locator**(URL)
  - Most common URI out there
  - http://www.coursera.org
- A **Uniform Resource Name** (URN)
  - Another form of URI
  - urn:isbn:0-619-0125356-5



URI

- ✧ Expose the resources using standard URIs
- ✧ Rails will automatically create URIs
  - Will register the resource in config/routes.rb
- ✧ `$ rails g scaffold_controller Movie title`
- ✧ `$ rails g scaffold_controller Actor name`

config/routes.rb



```ruby
Rails.application.routes.draw do
  resources :movies
  resources :actors
```

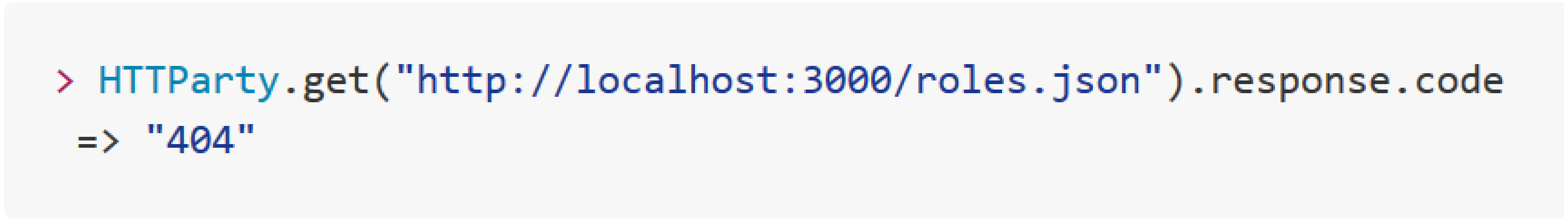
rake routes

```
$ rake routes
    Prefix Verb   URI Pattern                Controller#Action
    movies GET    /movies(.:format)          movies#index
     movie GET    /movies/:id(.:format)      movies#show

    actors GET    /actors(.:format)          actors#index
     actor GET    /actors/:id(.:format)      actors#show
```

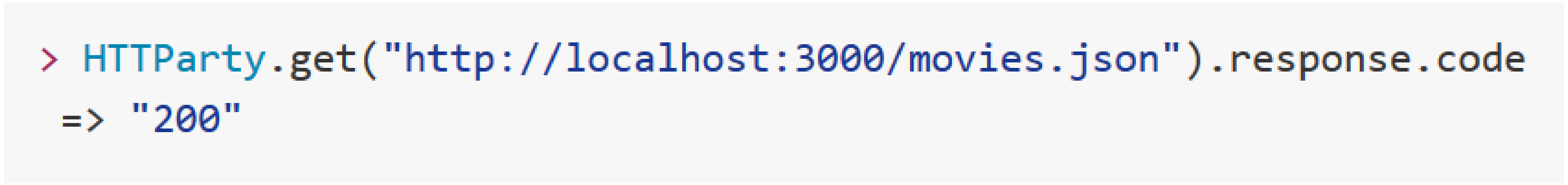
Scaled down to show unique URI – methods not shown

Access URI

- gem ‘httparty’



```
> HTTParty.get("http://localhost:3000/roles.json").response.code
 => "404"
```



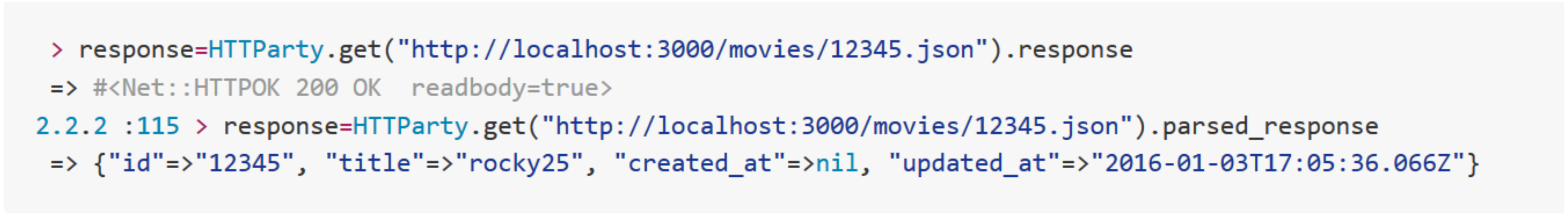
```
> HTTParty.get("http://localhost:3000/movies.json").response.code
 => "200"
```

Access URI – Actors and Movies

```
> pp HTTParty.get("http://localhost:3000/movies.json").parsed_response
[{"id"=>"12346",
  "title"=>"rocky26",
  "url"=>"http://localhost:3000/movies/12346.json"},
 {"id"=>"12345",
  "title"=>"rocky25",
  "url"=>"http://localhost:3000/movies/12345.json"}]

> pp HTTParty.get("http://localhost:3000/actors.json").parsed_response
[{"id"=>"100",
  "name"=>"sylvester stallone",
  "url"=>"http://localhost:3000/actors/100.json"}]
```

Access URI – Movie

- Specific resource
  - `/movies/:id` and `/actors/:id`



```
 > response=HTTParty.get("http://localhost:3000/movies/12345.json").response
 => #<Net::HTTPOK 200 OK  readbody=true>
2.2.2 :115 > response=HTTParty.get("http://localhost:3000/movies/12345.json").parsed_response
 => {"id"=>"12345", "title"=>"rocky25", "created_at"=>nil, "updated_at"=>"2016-01-03T17:05:36.066Z"}
```

Controller

- Update `MovieRoles` controller

```
def set_movie_role
  @movie_role = MovieRole.find(params[:id])
end
```

Summary

- ✧ HTTP provides an excellent interface to implement RESTful services with features like a URI and existing HTTP states

What's Next?

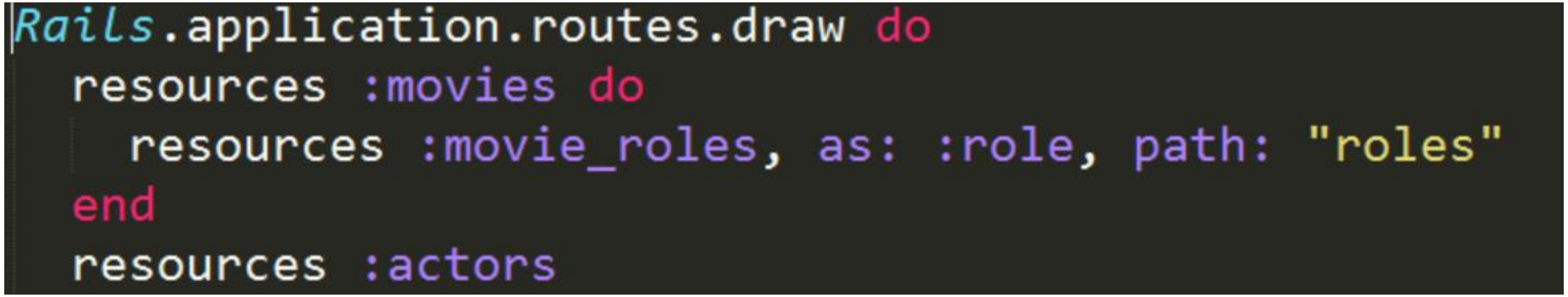
- ✧ Nested Resource URI

In this lecture, we will discuss…

- Nested URI
- Collection Resource
- `rake routes` – nested URI
- Movie example

Nested URI

- `$ rails g scaffold_controller MovieRole character actor_id`



```ruby
Rails.application.routes.draw do
  resources :movies do
    resources :movie_roles, as: :role, path: "roles"
  end
  resources :actors
```

rake routes

- Before `:as` property

```
$ rake routes
               Prefix Verb   URI Pattern                                      Controller#Action
    movie_movie_roles GET    /movies/:movie_id/movie_roles(.:format)          movie_roles#index
     movie_movie_role GET    /movies/:movie_id/movie_roles/:id(.:format)      movie_roles#show
```

- After `:as` property

```
$ rake routes
         Prefix Verb   URI Pattern                                  Controller#Action
    movie_roles GET    /movies/:movie_id/roles(.:format)            movie_roles#index
     movie_role GET    /movies/:movie_id/roles/:id(.:format)        movie_roles#show
```

Controller

- app/controllers/movie_roles_controller.rb

```
before_action :set_movie_role, only: [:show, :edit, :update, :destroy]

# GET /movie_roles/1
# GET /movie_roles/1.json
def show
end
...

  def set_movie_role
    @movie_role = Movie.find(params[:movie_id]).roles.find_by(:id=>params[:id])
  end
```

JSON Marshaller

- ✧ Default JSON marshaller definition expects timestamp
- ✧ `app/views/movie_roles/show.json.jbuilder`

```
json.extract! @movie_role, :id, :character, :actor_id, :created_at, :updated_at
```

- ✧ remove field for display

```
json.extract! @movie_role, :id, :character, :actor_id
```

Access URI – MovieRole

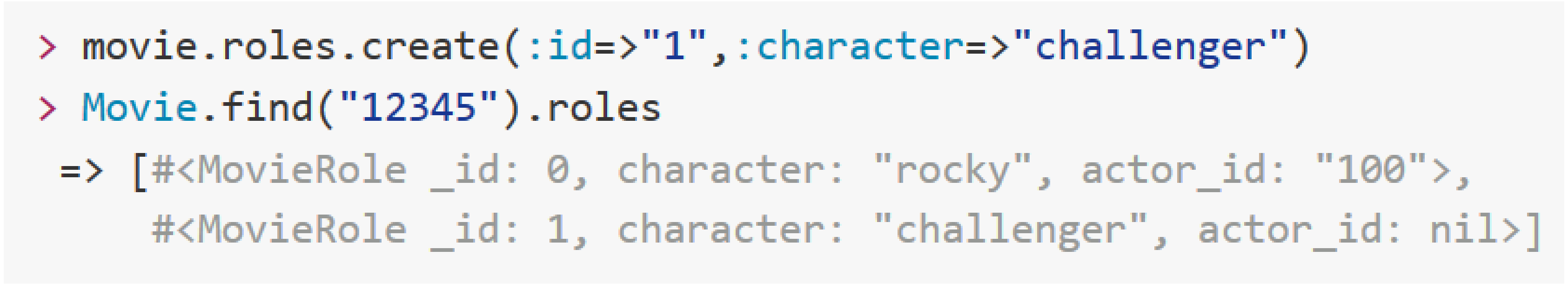
- ✧ MovieRole as a nested resource (Single) below Movie

```
> HTTParty.get("http://localhost:3000/movies/12345/roles/0.json").parsed_response
 => {"id"=>"0", "character"=>"rocky", "actor_id"=>"100"}
```

- ✧ Nested Resource (Collection)

```
> HTTParty.get("http://localhost:3000/movies/12345/roles.json").parsed_response
[]
```

Nested Resource - Collection



```
> movie.roles.create(:id=>"1",:character=>"challenger")
> Movie.find("12345").roles
 => [#<MovieRole _id: 0, character: "rocky", actor_id: "100">,
     #<MovieRole _id: 1, character: "challenger", actor_id: nil>]
```

- ✧ Define `before_action` and update `set_movie_role`
- ✧ Update JSON marshaller

Controller and index changes

```ruby
class MovieRolesController < ApplicationController
  before_action :set_movie
  before_action :set_movie_role, only: [:show, :edit, :update, :destroy]
...

    def set_movie_role
      @movie_role = @movie.roles.find_by(:id=>params[:id])
    end
    def set_movie
      @movie = Movie.find(params[:movie_id])
    end
```

```ruby
# GET /movie/:movie_id/roles
# GET /movie/:movie_id/roles.json
def index
  @movie_roles=@movie.roles
end
```

JSON Marshaller

- Add the `@movie` as a parameter to the `movie_role_url` helper method.

```
json.array!(@movie_roles) do |movie_role|
  json.extract! movie_role, :id, :character, :actor_id
  json.url movie_role_url(movie_role, format: :json)
end
```

```
json.array!(@movie_roles) do |movie_role|
  json.extract! movie_role, :id, :character, :actor_id
  json.url movie_role_url(@movie, movie_role, format: :json)
end
```

Nested Resource - Collection

```
> pp HTTParty.get("http://localhost:3000/movies/12345/roles.json").parsed_response
[{"id"=>"0",
  "character"=>"rocky",
  "actor_id"=>"100",
  "url"=>"http://localhost:3000/movies/12345/roles/0.json"},
 {"id"=>"1",
  "character"=>"challenger",
  "actor_id"=>nil,
  "url"=>"http://localhost:3000/movies/12345/roles/1.json"}]
```

Summary

- Collection resource URI
- Nested data

What's Next?

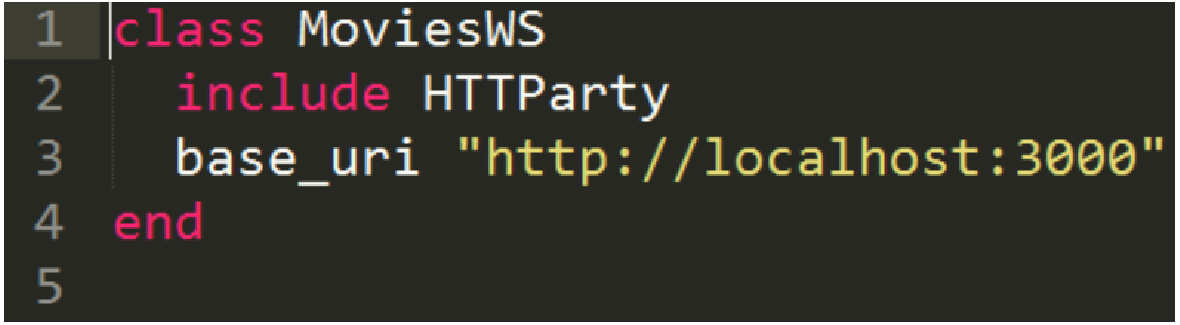
- Query Parameters and Payload

In this lecture, we will discuss…

- Query Parameters - GET
- POST Data
- Whitelisting parameters
- Cross Site Scripting (XSS)

HTTParty Client class

- ✧ Helper class - `app/services/movies_ws.rb`



```ruby
class MoviesWS
  include HTTParty
  base_uri "http://localhost:3000"
end

```

```
> MoviesWS.get("/movies/12345.json").parsed_response
 => {"id"=>"12345", "title"=>"rocky25", "created_at"=>nil, "updated_at"=>"2016-01-03T17:05:36.066Z"}
```

Parameter Types

- URI elements (e.g., :movie_id, :id)
- Query String - part of the URI, uses "?", and contains individual query parameters
- POST Data - in the payload body.

Parameter Types – Example

```
> MoviesWS.get("/movies.json?title=rocky25&foo=1&bar=2&baz=3").parsed_response
```

```
{"title"=>"rocky25", "foo"=>"1", "bar"=>"2", "baz"=>"3",
 "controller"=>"movies", "action"=>"index", "format"=>"json"}
```

Post Data – Example

```
MoviesWS.post("/movies.json",
     :body=>{:movie=>{:id=>"123457",:title=>"rocky27",:foo=>"bar"}}.to_json,
     :headers=>{"Content-Type"=>"application/json"})
```

```
{"movie"=>{"id"=>"123457", "title"=>"rocky27", "foo"=>"bar"},
  "controller"=>"movies", "action"=>"create", "format"=>"json"}
```

White Listing Parameters

- Rails has built in features based on parameters
- Controller has a “white list” of acceptable parameters
- White list with 2 fields

```
def movie_params
  params.require(:movie).permit(:id,:title)
end
```

- Usage

```
def create
  @movie = Movie.new(movie_params)
```

White Listing Parameters

```
{"movie"=>{"id"=>"123457", "title"=>"rocky27", "foo"=>"bar"},
  "controller"=>"movies", "action"=>"create", "format"=>"json"}
```

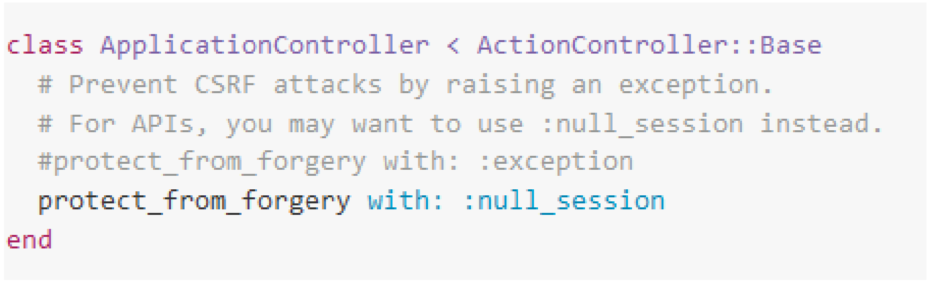
```
{"id"=>"123457", "title"=>"rocky27"}
```

Cross Site Scripting (XSS)

- ✧ Browsers can run scripts (JavaScript)
- ✧ If a user trusts a website, might allow the scripts to run
  - `<script type="text/javascript" > alert("Hard Disk Error. Click OK."); </script >`
- ✧ It is possible to inject malicious scripts into content from trusted sites
- ✧ Scripts can hijack user sessions, redirect user to other sites

Cross Site Scripting (XSS)

- POST request by default will fail
  - Can't verify CSRF (Cross Site Request Forgery) token authenticity - message
- Relax Security
  - `app/controllers/application_controller.rb`



```ruby
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  #protect_from_forgery with: :exception
  protect_from_forgery with: :null_session
end
```

Query Parameters - Demo

Demo

Other Parameter - options

✧ Arrays

```
MoviesWS.get("/movies.json?id[]=12345&id[]=12346&foo[]=1&foo[]=2")
```

```
{"id"=>"12346", "foo"=>["1", "2"], "controller"=>"movies", "action"=>"index", "format"=>"json"}
```

Other Parameter - options

- Hash

```
key=>{"prop1"=>"val", "prop2"=>val}
```

```
MoviesWS.get("/movies.json?movie[id]=12345&movie[title]=rocky27&movie[year]=2016")
```

```
{"movie"=>{"id"=>"12345", "title"=>"rocky27", "year"=>"2016"},
   "controller"=>"movies", "action"=>"index", "format"=>"json"}
```

Summary

- ✧ Query Parameters – common way to request data
- ✧ CSRF security concerns and whitelisting parameters

What’s Next?

- ✧ Methods

In this lecture, we will discuss…

✧ HTTP Methods

- POST
- PUT
- PATCH
- HEAD

HTTP Methods - POST

- POST is for creating new resource instances
  - POST to a resource URI
  - Provide JSON payload (but optional)
  - Provide MIME type of the payload in the Content-Type header

HTTP Methods - POST

```
> MoviesWS.post("/movies.json",:body=>{:movie=>{:id=>"123457",:title=>"rocky27"}}.to_json)
```

```
<- "POST /movies.json HTTP/1.1\r\n
Content-Type: application/json\r\n
Connection: close\r\n
Host: localhost:3000\r\n
Content-Length: 43\r\n
\r\n"
<- "{\"movie\":{\"id\":\"123457\",\"title\":\"rocky27\"}}"
```

POST (Update) - Action

- Builds a white-list version of parameter hash
- Builds a new instance of the Movie class with the hash passed
- Saves the resultant Movie to the database
- Renders a result back to the caller based on the format requested in the response and the status of the save.

PUT

- ✧ PUT is for replacing the data (Update)
- ✧ The Client
  - issues a PUT request
  - issues the request to `/movies/123457` URI
  - provides a JSON payload for update
  - provides `application/json` MIME type

HTTP Methods - PUT

```
> response=MoviesWS.put("/movies/123457.json",:body=>{:movie=>{:title=>"rocky2700",:foo=>"bar"}}.to_json)
```

```
<- "PUT /movies/123457.json HTTP/1.1\r\n
Content-Type: application/json\r\n
Connection: close\r\n
Host: localhost:3000\r\n
Content-Length: 43\r\n
\r\n"
<- "{\"movie\":{\"title\":\"rocky2700\",\"foo\":\"bar\"}}"
```

PUT(Update) - Action

- PUT expects the primary key to be in the `:id` parameter
- If the movie is found, processing continues
- Builds a white-list-checked set of parameters
- Supplies the values to the update method
- Returns the result document

HTTP Methods - PATCH

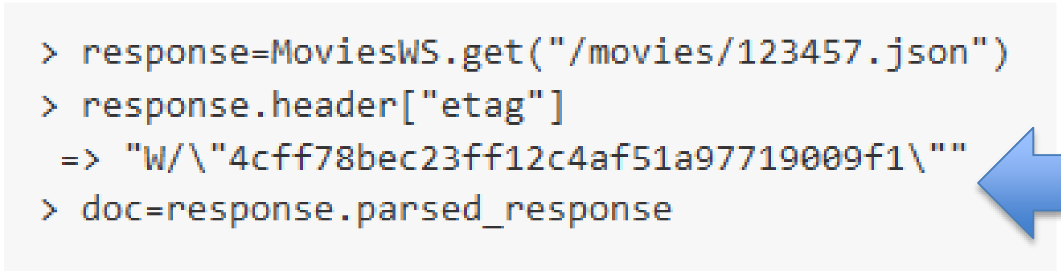
- PATCH is for partially updating a resource
- Update a field vs. entire resource

```
MoviesWS.patch("/movies/123457.json",:body=>{:movie=>{:title=>"rocky2702",:foo=>"bar"}}.to_json)
```

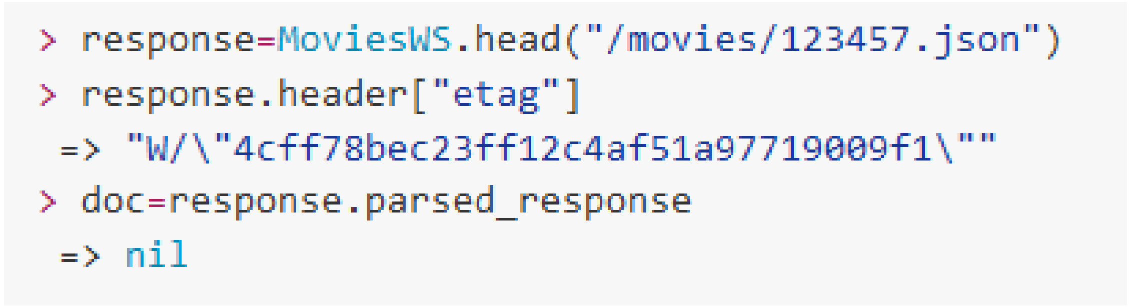
HTTP Methods - HEAD

- HEAD is basically GET without the response body
- Useful to retrieve meta-information written in response headers
- Issue GET and store `Etag` for comparison later

HEAD



```
> response=MoviesWS.get("/movies/123457.json")
> response.header["etag"]
 => "W/\"4cff78bec23ff12c4af51a97719009f1\""
> doc=response.parsed_response
```

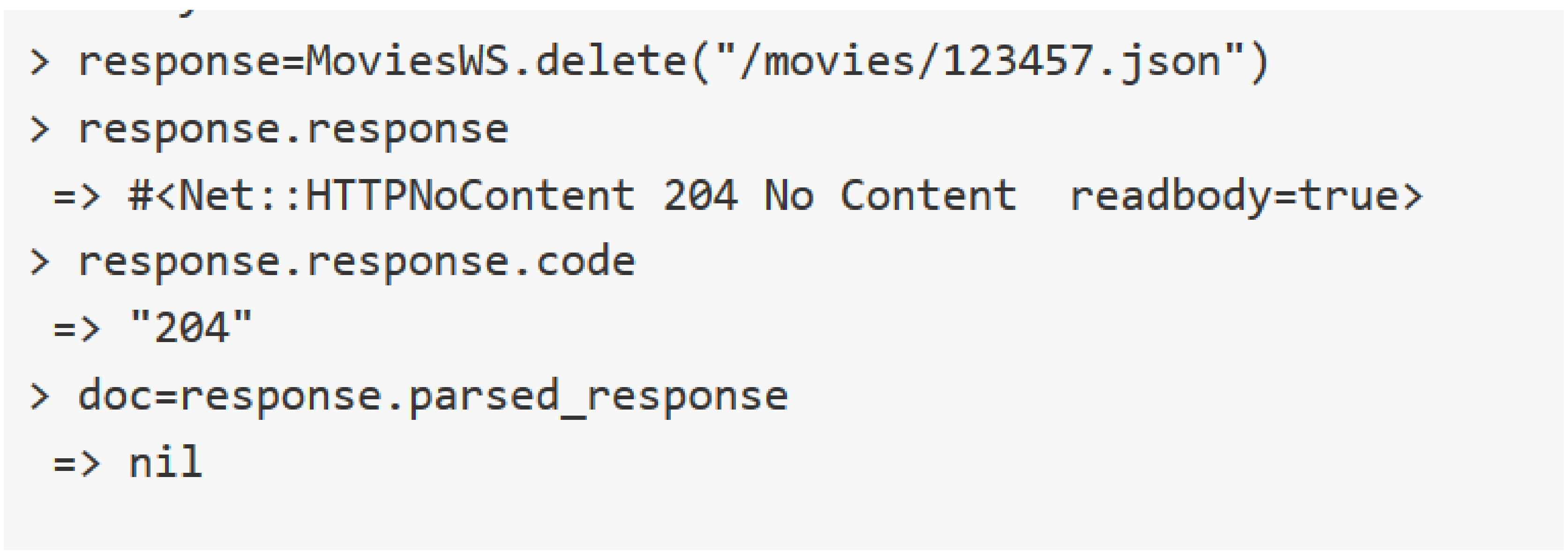


```
> response=MoviesWS.head("/movies/123457.json")
> response.header["etag"]
 => "W/\"4cff78bec23ff12c4af51a97719009f1\""
> doc=response.parsed_response
 => nil
```

HTTP Methods - DELETE

- DELETE is for deleting a resource
- It accepts an `:id` parameter from the URI and removes that document from the database.
- No request body

DELETE - Example



```
> response=MoviesWS.delete("/movies/123457.json")
> response.response
 => #<Net::HTTPNoContent 204 No Content  readbody=true>
> response.response.code
 => "204"
> doc=response.parsed_response
 => nil
```

HTTP Methods - GET

- GET is for data retrieval only
- Free of side effects, a property also known as *idempotence* (discussed later)

```
> MoviesWS.get("/movies.json?title=rocky25&foo=1&bar=2&baz=3").parsed_response
 => [{"id"=>"12345", "title"=>"rocky25", "url"=>"http://localhost:3000/movies/12345.json"}]
```

Summary

- ✧ HTTP Methods maps seamlessly to CRUD operations
- ✧ Elegant and easy for the clients

What’s Next?

- ✧ Idempotence

Next Topic…..

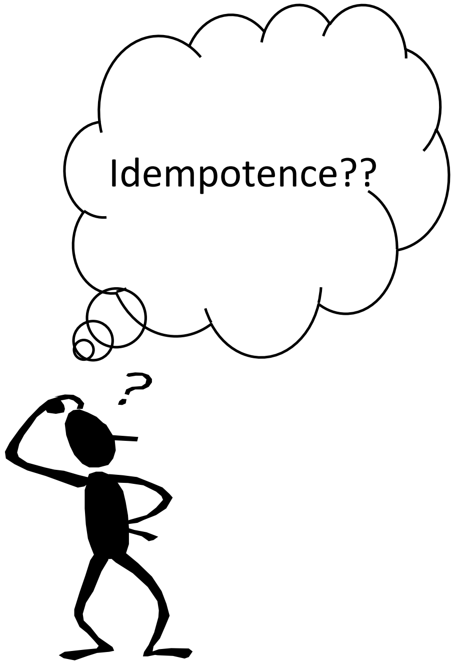
Idempotence

In this lecture, we will discuss…

- ✧ Method Idempotence
- ✧ GET
- ✧ PUT
- ✧ PATCH
- ✧ DELETE
- ✧ POST

Idempotence – Wikipedia

- *Idempotence is the property of certain operations in mathematics and computer science, that can be applied multiple times without changing the result beyond the initial application.*



HTTP Methods

- ✧ GET

read operation

- ✧ POST
- ✧ PATCH
- ✧ DELETE
- ✧ PUT

write operation

GET

- Definition – "…*that can be applied multiple times without changing the result*"
- GET – gets the data (no change in result)
- Idempotent ✓

DELETE

- ✧ Definition – "…*that can be applied multiple times without changing the result*"
- ✧ Multiple calls – (no change in result)
  - Server side exception
- ✧ Idempotent ✓

PUT

- Definition – "…*that can be applied multiple times without changing the result*"
- Multiple calls – (no change in result)
- Idempotent ✓

POST

- Definition – "…*that can be applied multiple times without changing the result*"
- New resource created
- Multiple calls – problem
- Idempotent ✗

Summary

- ✧ GET, PUT, PATCH and DELETE – idempotent
- ✧ POST – not idempotent

What’s Next?

- ✧ Representations