Web Services Concept Details

Resources

One of the first fundamental issues to address when building a Web Service is "what are our resources?" Here we will keep it simple and stick to a simple model to work with

- Movies
- Actors
- MovieRoles

The first two are stand-alone resources. The third is a dependent resource.

- Standalone - can have an instance of that resource without another
  - Movies can exist without Actors or MovieRoles (but would be boring)
  - Actors can exist without Movies or MovieRoles (but cannot eat)
- Dependent - must have another resource to exist
  - MovieRole depends on Movie to exist
  - MovieRole is related to Actor, but still can exist before and if the relationship is severed.

As we know, Rails provides a lot of help in defining resources.

- `rails g` scaffold can build templated code to implement most CRUD behavior and interactions
- detailed ActiveModel or Mongoid frameworks help implement further details related to relating resources.

Resource Models

We can generate model classes to implement our data resources.

```
$ rails g model Movie title
$ rails g model Actor name
$ rails g model MovieRole character
```

Our global resources are ready to use out of the box.

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

  embeds_many :roles, class_name: "MovieRole"
end
class Actor
  include Mongoid::Document
  field :name, type: String
end
```

```
> movie=Movie.create(:id=>"12345", :title=>"rocky25")
 => #<Movie _id: 12345, title: "rocky25">
> Movie.find("12345")
 => #<Movie _id: 12345, title: "rocky25">

> actor=Actor.create(:id=>"100",:name=>"sylvester stallone")
 => #<Actor _id: 100, name: "sylvester stallone">
> Actor.find("100")
 => #<Actor _id: 100, name: "sylvester stallone">
```

We need to add a relationship to the generated model before we can use our dependent resource.

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

  embedded_in :movie
end
class Movie
  include Mongoid::Document
  field :title, type: String

  embeds_many :roles, class_name: "MovieRole"
end
```

With the embedded relationship in place, we can create a MovieRole within the Movie.

```
rocky=movie.roles.create(:id=>"0",:character=>"rocky")
 => #<MovieRole _id: 0, character: "rocky">
> Movie.find("12345").roles.where(:id=>"0").first
 => #<MovieRole _id: 0, character: "rocky">
```

We can also link our resource classes so that we can navigate from one to the other.

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

  embedded_in :movie
  belongs_to :actor
end
```

```
> rocky=Movie.find("12345").roles.where(:id=>"0").first
 => #<MovieRole _id: 0, character: "rocky", actor_id: nil>
> actor=Actor.find(100)
 => #<Actor _id: 100, name: "sylvester stallone">
> rocky.actor=actor
 => #<Actor _id: 100, name: "sylvester stallone">
> rocky.save
 => true
> Movie.find("12345").roles.where(:id=>"0").first.actor.name
 => "sylvester stallone"
```

We can also go the other way in a with a bit of application logic added

```
class Actor
  include Mongoid::Document
  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
```

```
> actor=Actor.find("100")
 => #<Actor _id: 100, name: "sylvester stallone">
> actor.roles.map {|r| "#{r.movie.title}, #{r.character}"}
 => ["rocky25, rocky"]
```

We have

- used `rails generate model` to generate our model classes
- used the ORM to add dependency and relationship details
- can perform basic CRUD on these resources
- have the ability to add more features as demonstrated with the custom application query logic to implement the inverse side of the embedded relationship.

URIs

We have our resources and now we need to expose them to the web using standard URIs. Rails will automatically create URIs for our global resources once we add a controller scaffolding and register the resource in 'config/routes.rb.

```
$ rails g scaffold_controller Movie title
$ rails g scaffold_controller Actor name
```

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

We can take a look at what URIs Rails has assigned to our resources using the command `rake routes` and work with only a few of the URIs for global resources now.

```
* collection resource URIs (/movies, /actors)
* indivial object URIs (/movies/:id, /actors/:id)
```

```
$ 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
```

Lets access on of these URIs using an HTTP request through the use of `HTTParty`. We add the following to our Gemfile, run `bundle`, and restart `rails console`. We also want to have our server running at this time.

```
gem 'httparty'
```

```
$ bundle
$ rails s
```

If we request a URI that does not exist (/roles does not exist), we get a `404/NOT_FOUND` error response. That says there is no resource at that URI.

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

If we use a valid URI for the collection (/movies and /actors) we get success responses. The payload of the response is made available thru `parsed_response()` as a hash.

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

> 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"}]
```

If we request a specific resource (/movies/:id and /actors/:id) we initially get a server error because the default mashaller wants to access timestamp information that does not yet exist in our models.

```
HTTParty.get("http://localhost:3000/movies/12345.json").response
 => #<Net::HTTPInternalServerError 500 Internal Server Error  readbody=true>
```

We can easily add the timestamp properties and provide at least an `updated_at` value using the rails console.

```
class Movie
  include Mongoid::Document
  include Mongoid::Timestamps
class Actor
  include Mongoid::Document
  include Mongoid::Timestamps

>  Movie.each {|m| m.touch}
>  Actor.each {|a| a.touch}

 > 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"}
```

A nested URI requires some work. We start off creating a controller and other scaffold items.

```
$ rails g scaffold_controller MovieRole character actor_id
```

We now need to register the resource as nested below movies.

```
Rails.application.routes.draw do
  resources :movies do
    resources :movie_roles
  end
  resources :actors
```

Except that alone will expose the URI with `movie_roles` as the name and we would like it to be `roles`.

```
$ 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
```

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

The `as:` property adjusted the helper methods available inside of the view when refering to the URI and method. The `path:` property changed the value from `movie_roles` to `roles` in between the `movie_id` and `role_id`.

```
$ 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
```

We now need to update the `MovieRoles` controller to obtain the `MovieRole` within the context of a `Movie` (`app/controllers/movie_roles_controller.rb`). The generated controller class wrote the accessor as if `MovieRole` was a global resource. We need to change the definition to a nested resource.

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

From the output of `rake routes`, we know that

- `:movie_id` - is the property supplying the ID of the Movie
- `:id` - is the property supplying the ID of the MovieRole within the Movie

These properties will be made available to the controller code within the `params` hash. From the rails console, we know that the following Mongoid query will produce the desired object. First the parent `Movie` document is retrieved from the database by `id` and then the embedded collection of `MovieRoles` is search for a `MovieRole` having a specific `id`. We use the `find_by()` on the second query just to highlight that more flexible queries could be used if we were not searching just by ID.

```
> Movie.find("12345").roles.find_by(:id=>"0")
 => #<MovieRole _id: 0, character: "rocky", actor_id: "100">
```

With that in-hand, we make the changes to the contoller helper method that is triggered as a `before_action` and supplies a `@movie_role` to the `movie_roles#show` action.

```
  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
```

We must make one more change. The default JSON marshaller definition for our nested resource wants to also see timestamp information within the model. (`app/views/movie_roles/show.json.jbuilder`)

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

To show a simple change to the marshaller can prevent us from making a unreasonable change to our model, we have chosen to remove those fields from the marshaller definition for this type.

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

We now can access the `MovieRole` as a nested resource below Movie.

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

That took care of the nested single resource URI, but not the nested resource collection (i.e., get all roles for a movie). We currently get nothing.

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

That is because the generated `index` method is still querying the `MovieRole` as if it were a global resource.

```
  # GET /movie_roles
  # GET /movie_roles.json
  def index
    @movie_roles = MovieRole.all
  end
```

With the knowledge that `Movie` must be found and the roles returned for that movie, lets fix it with a helper that can be reused elsewhere.

```
> 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>]
```

Lets define a `before_action` that is executed before all other methods and populates the `@movie` attribute with the parent `Movie` object scoping the MovieRoles. Also update the previous `set_movie_role` to use the product as well.

```
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
```

Finally the action method itself – the one who knows what we need to get from the `@movie` is called and returns the roles. The comment was also updated to reflect the new URI for this action.

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

A few of the templates also need to be updated. If you look back at the output of `rake routes` after we adjusted the `MovieRole` resource to be a nested resoure, all helper methods now require two parameters (`Movie` and `MovieRole`) and not just `Movie` because this is no longer a global resource.

```
$ 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
```

Update the offending default JSON marshaller for the index action (app/views/movie_roles/index.json.jbuilder). 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
```

With these changes we can get to the JSON-based URIs and views that we need to to make it work.

```
> 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"}]
```

And while you are at it, you can clean up some paths for the HTML side. Notice that the default Views created use the model object with no helper method and there is not a helper method available for every action. We have complicated things a bit by specify `:movie_roles` to be expressed as `:roles`. Otherwise some of the `movie_role` entries could be specified as `[@movie, movie_role]`. However, with using the path mapping we must explcitly supply the helper method thar generates the correct URI and add a `method:` override if that helper method does not reference the desired method for that link.

From:

```
app/views/movie_roles/index.html.erb
<td><%= link_to 'Show', movie_role %></td>
<td><%= link_to 'Edit', edit_movie_role_path(movie_role) %></td>
<td><%= link_to 'Destroy', movie_role, method: :delete, ...
```

To:

```
<td><%= link_to 'Show', movie_role_path(@movie, movie_role) %></td>
<td><%= link_to 'Edit', edit_movie_role_path(@movie, movie_role) %></td>
<td><%= link_to 'Destroy', movie_role_path(@movie, movie_role), method: :delete, ...
```

From:

```
app/views/movie_roles/show.html.erb
<%= link_to 'Edit', edit_movie_role_path(@movie_role) %> |
<%= link_to 'Back', movie_roles_path %>
```

To:

```
<%= link_to 'Edit', edit_movie_role_path(@movie, @movie_role) %> |
<%= link_to 'Back', movie_path(@movie)  %>
```

From:

```
app/views/movie_roles/_form.html.erb
<%= form_for(@movie_role) do |f| %>
```

To:

```
<%= form_for([@movie,@movie_role], url: movie_role_path(@movie, @movie_role)) do |f| %>
```

From:

```
app/views/movie_roles/edit.html.erb
<%= link_to 'Show', @movie_role %> |
<%= link_to 'Back', movie_roles_path %>
```

To:

```
<%= link_to 'Show', movie_role_path(@movie, @movie_role) %> |
<%= link_to 'Back', movie_path(@movie) %>
```

Custom URI and Methods

You can define a custom route to any controller action.

```
Rails.application.routes.draw do
  resources :movies do
    resources :movie_roles, as: :role, path: "roles"
  end
  resources :actors
  get "/api/movies/:id" => "movies#show"
```

```
$ rake routes | grep show
      movie_role GET    /movies/:movie_id/roles/:id(.:format)      movie_roles#show
           movie GET    /movies/:id(.:format)                      movies#show
           actor GET    /actors/:id(.:format)                      actors#show
                 GET    /api/movies/:id(.:format)                  movies#show
```

By adding the `as:` property, a helper method is generated with that name. This helper method is useful in deriving the URL and URI of that controller#action.

```
  get "/api/movies/:id" => "movies#show", as: "mystuff"
```

```
$ rake routes | grep show
         mystuff GET    /api/movies/:id(.:format)                  movies#show
```

Namespaces

You can define that certain resources are scoped below a namespace.

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

```
$ rake routes | grep show
      api_movie_role GET    /api/movies/:movie_id/roles/:id(.:format)      api/movie_roles#show
           api_movie GET    /api/movies/:id(.:format)                      api/movies#show
           api_actor GET    /api/actors/:id(.:format)                      api/actors#show
```

HTTParty Client Class

In the examples above, we used HTTParty as a single command and issued some of the same parameters each time. We can create a class to capture some of the repetitive information and things we might want to toss into a configuration file.

- app/services/movies_ws.rb

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

- config/application.rb - any directory below app/ should be automatically loaded by the server. However, I have ran into issues when Mongoid is configured and the following can be used to manually correct. The funny thing is that once corrected and rails console run a single time – it is no longer needed???

```
Mongoid.load!("./config/mongoid.yml")
config.eager_load_paths += %W( #{config.root}/app/services )
```

Now we just need to supply the URI for each call.

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

Query Parameters and Payloads

Parameter Types

- URI elements (e.g., :movie_id, :id) - introduced in the last section
- Query String - part of the URI, to the right of the "?", and contains individual query parameters
- POST Data - in the payload body. Only available with a POST

Rails makes all available through a parameter hash.

Query Param Example If we pass the following query string to the Rails server, the following hash gets passed to the controller in a variable called `params`.

```
> 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 If we post a document to the Rails server, the following hash gets passed to the controller in the variable called `params`.

```
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"}
```

Mixing POST Data and Query Parameters If we specify both a POST body and query parameters in the same POST, the two sources of information will be merged as if they came from one or the other. A common example is where the payload contains a complex query specification and the URL contains paging controls.

In the following example, the controller has access to all the properties thru the same hashes discussed earlier. Only the properties within the `movies` element will be provided to the white-list checker and only `id` and `title` will make it thru – same as usual.

```
MoviesWS.post("/movies.json",
              :body=>{:movie=>{:id=>"123458",:title=>"rocky28",:foo=>"bar"}}.to_json,
              :query=>{:bar=>1,:baz=>2},
              :headers=>{"Content-Type"=>"application/json"})
```

```
{"movie"=>{"id"=>"123458", "title"=>"rocky28", "foo"=>"bar"},
           "bar"=>"1", "baz"=>"2", "controller"=>"movies", "action"=>"create", "format"=>"json"}
```

Note that any duplicate query parameter (except the technique discussed below using arrays) will result in the second occurance replacing the previous occurence. That means you cannot supply some of the movie in the body and the rest of the movie in the query string because they would both be scoped below `movie` and the second would wipe out the first.

White Listing Parameters Rails has quite a bit of functionality based on parameters and mass assignment. This leads to security concerns when it is time to pass these values into queries or save/update logic. Each controller has a white-list of acceptable parameters and can built a sanitized version of the hash parameters upon request.

Here we are showing the white-list with two fields (`id`, and `title`). This list is initially populated when the controller is created and must be manually updated as you make changes.

```
    # Never trust parameters from the scary internet, only allow the white list through.
    def movie_params
      params.require(:movie).permit(:id,:title)
    end
```

The following is an example of the white-list being used.

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

The following `params` hash will produce the cleansed result. Elements must be scoped below `movie` and be either `id` or `title` to pass thru to the created movie.

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

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

Cross Site Script Attacks **KIRAN**: fill in background here

Out of the box – our POST above failed because the server would not allow the request to go through.

```
MoviesWS.post("/movies.json", :body=>movie.to_json, :headers=>{'Content-Type'=>'application/json'}).response
 => #<Net::HTTPUnprocessableEntity 422 Unprocessable Entity  readbody=true>
```

```
Started POST "/movies.json" for 127.0.0.1 at 2016-01-03 18:36:34 -0500
Processing by MoviesController#create as JSON
  Parameters: {"id"=>"123457", "title"=>"rocky27", "movie"=>{"title"=>"rockey27"}}
Can't verify CSRF token authenticity
Completed 422 Unprocessable Entity in 1ms (ActiveRecord: 0.0ms)
```

To get things to work, we either had to relax security on the server or go through the steps to provide authentication to our HTTParty client. Since we are early in WS topics, it was chosen to take one of the advertised paths within Rails to relax forgery checks when there is no session active.

`app/controllers/application_controller.rb`

```
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
```

Implementations

Query Parameter Implementation As we saw with the query parameter example above, the controller was presented with a hash of values in a variable called `params`.

```
> 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"}
```

However, our default `movies#index` action is ignoring the provided parameters and will always return all values.

```
  def index
    @movies = Movie.all
  end
```

We know from our database queries that we cannot just pass the query directly into the where clause since the hash has extra values.

```
> Movie.where({"title"=>"rocky25", "foo"=>"1", "bar"=>"2", "baz"=>"3", \
    "controller"=>"movies", "action"=>"index", "format"=>"json"}).first
 => nil
```

If we trim the list down to the properties relevant to the Movie, we get the result we expect.

```
> Movie.where({"title"=>"rocky25", "foo"=>"1", "bar"=>"2", "baz"=>"3",
               "controller"=>"movies", "action"=>"index", "format"=>"json"}.slice("title")).first
 => #<Movie _id: 12345, created_at: nil, updated_at: 2016-01-03 17:05:36 UTC, title: "rocky25">
```

That is because slice removed all elements that were not the title value leaving just that value passed into the query.

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

If Movie has lots of attributes and we would like to query on them all if they are supplied – we could ask ActiveModel for that list of attributes.

```
> Movie.attribute_names
 => ["_id", "created_at", "updated_at", "title"]
{"title"=>"rocky25", "foo"=>"1", "bar"=>"2", "baz"=>"3",
 "controller"=>"movies", "action"=>"index", "format"=>"json"}.slice(*Movie.attribute_names)
 => {"title"=>"rocky25"}

> Movie.where({"title"=>"rocky25", "foo"=>"1", "bar"=>"2", "baz"=>"3",
               "controller"=>"movies", "action"=>"index", "format"=>"json"}.slice(*Movie.attribute_names)).t
 => [#<Movie _id: 12345, created_at: nil, updated_at: 2016-01-03 17:05:36 UTC, title: "rocky25">]
```

With the cleansed params hash in place . . .

```
  # GET /movies
  # GET /movies.json
  def index
    Rails.logger.debug("index.params=#{params}")
    @movies = Movie.all(params.slice(*Movie.attribute_names))
  end
```

We get the expected result – a match on title.

```
> 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"}]
```

POST Payload Implementation There are two issues to address on the server-side that was not part of the default scaffold

- address CSRF security concerns (addressed above)
- update the white-list to include everything we want as part of mass assignment (addressed above as well)

The biggest change is on the client-side. We have to now form a POST instead of a GET. This is done with HTTParty by

- changing `get()` to `post()`

- providing a JSON document as a :body element in the second parameter
- providing an `application/json` MIME type informing the server you are passing it JSON

The `:body` element must have a hash with the proper root key – `:movie` in this case. Everything in that movie hash will be subject to the white-list. Only `id` and `title` will pass thru to the create.

```
response=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"}
```

Among other things, as successful 201/CREATED is returned along with a payload that shows the object we created.

```
> response.response
 => #<Net::HTTPCreated 201 Created  readbody=true>
> response.parsed_response
 => {"id"=>"123457", "title"=>"rocky27",
     "created_at"=>"2016-01-04T00:09:50.271Z", "updated_at"=>"2016-01-04T00:09:50.271Z"}
```

Other Parameter Options

Arrays Query parameters names that are followed by an array "[]" symbol will get grouped by common name and passed in as an array.

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

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

Hash Query parameters with the format of `key[prop1]=val&key[prop2]=val` will get converted into a hash in the form of `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"}
```

HTTParty Client Changes

In the event that POSTs are used in the future and we are likely to be posting JSON as our default case, lets set the default header in our HTTParty service class to be `Content-Type"=>"application/json`.

```
class MoviesWS
  include HTTParty
  debug_output $stdout
  base_uri "http://localhost:3000"
  headers "Content-Type"=>"application/json"
end
```

Methods

Note that the `Content-Type` is being set in the class during this section. It needs to be included on each call.

Method Calls

POST: Create We create a new movie by

- issuing a POST to the `/movies` URI
- supplying a JSON payload
- supplying an `application/json` MIME-type for Content-Type

```
> 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\"}}"
```

The `resources` element in 'config/routes.rb has defined a route to implement all of our standard CRUD methods.

```
  resources :movies
```

`rake routes` shows that a POST to this URI will result in the request being routed to the `movies#create` controller action.

```
$ rake routes | grep movies | grep -i post
                POST   /movies/:movie_id/roles(.:format)          movie_roles#create
                POST   /movies(.:format)                          movies#create
```

The create action

- builds a white-list-checked version of the parameter hash built from the payload body and query string (there is only a payload body in this case). Elements selected have to be scoped below a movie element and have to be included in the white-list located within the controller.
- builds a new instance of the Movie class using the mass assignment interface for the model with a hash passed in for mass assignment.
- 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`.

```
  # POST /movies
  # POST /movies.json
  def create
    @movie = Movie.new(movie_params)

    respond_to do |format|
      if @movie.save
        format.json { render :show, status: :created, location: @movie }
      else
        format.json { render json: @movie.errors, status: :unprocessable_entity }
      end
    end
  end
```

Upon success, the `show` view of the created `Movie` is marshaled as JSON based on the definition in `app/views/movies/show.json.jbuilder`.

```
json.extract! @movie, :id, :title, :created_at, :updated_at
```

The response code is a 201/CREATED. The header returned contains

- content-type and length of the payload body
- URL location where the created resource can be located
- an Etag thumbprint of the state of the document
- cache controls
- etc.

```
-> "HTTP/1.1 201 Created \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Location: http://localhost:3000/movies/123457\r\n"
-> "Content-Type: application/json; charset=utf-8\r\n"
-> "Etag: W/\"51c29f87ec574767a34a2f3aa3bd5de6\"\r\n"
-> "Cache-Control: max-age=0, private, must-revalidate\r\n"
-> "X-Request-Id: 33003d5b-3acd-4f59-b2fd-bfda8e42ba86\r\n"
-> "X-Runtime: 0.008102\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 02:01:50 GMT\r\n"
-> "Content-Length: 113\r\n"
-> "Connection: close\r\n"
-> "\r\n"
reading 113 bytes...
-> ""
-> "{\"id\":\"123457\",\"title\":\"rocky27\",\"created_at\":\"2016-01-04T02:01:50.932Z\",
\"updated_at\":\"2016-01-04T02:01:50.932Z\"}"
```

The HTTParty client can access

- the response code
- any header information through a hash
- payload

Obviously the status code and payload response the the most critical parts to check and process.

```
> response.response
 => #<Net::HTTPCreated 201 Created  readbody=true>
> response.response.code
 => "201"

> response.headers["etag"]
 => "W/\"51c29f87ec574767a34a2f3aa3bd5de6\""

> response.parsed_response
 => {"id"=>"123457", "title"=>"rocky27",
    "created_at"=>"2016-01-04T02:01:50.932Z", "updated_at"=>"2016-01-04T02:01:50.932Z"}
```

PUT: Update In this next example, we are performing an update on an existing document. Technically this is consisted a wholesale replacement, but you will see that the timestamps are not impacted by the fact they are not included.

The client

- issues a PUT request
- addresses the request to be handled by the /movies/123457 URI
- supplies a JSON payload for the update
- specified the content-type as `application/json`

```
> 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\"}}"
```

The URI/Method to action routing directs processing at the `movies#update` method.

```
$ rake routes | grep movies | grep -i put
                PUT    /movies/:movie_id/roles/:id(.:format)      movie_roles#update
                PUT    /movies/:id(.:format)                      movies#update
```

This method:

- has the `set_movie` before action called to obtain the `Movie` to be updated. This method expects the primary key to be in the `:id` parameter, which is a parameter supplied in the URI (see `rake routes` output above)
- if the movie is found, processing continues with the movie stored in an instance variable @movie
- builds a white-list-checked set of parameters
- supplies the values to the update method
- returns the result document – same as with POST/create

```
class MoviesController < ApplicationController
  before_action :set_movie, only: [:show, :edit, :update, :destroy]

    def set_movie
      @movie = Movie.find(params[:id])
    end

  # PATCH/PUT /movies/1
  # PATCH/PUT /movies/1.json
  def update
    respond_to do |format|
      if @movie.update(movie_params)
        format.json { render :show, status: :ok, location: @movie }
      else
        format.json { render json: @movie.errors, status: :unprocessable_entity }
      end
    end
  end

-> "HTTP/1.1 200 OK \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Location: http://localhost:3000/movies/123457\r\n"
-> "Content-Type: application/json; charset=utf-8\r\n"
```

```
-> "Etag: W/\"42ac006fa7e27e4b0031eae4d08e8c74\"\r\n"
-> "Cache-Control: max-age=0, private, must-revalidate\r\n"
-> "X-Request-Id: 978f30ab-4b15-4d6f-8ab7-4c118c9b36d2\r\n"
-> "X-Runtime: 0.020640\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 03:23:18 GMT\r\n"
-> "Content-Length: 115\r\n"
-> "Connection: close\r\n"
-> "\r\n"
reading 115 bytes...
-> ""
-> "{\"id\":\"123457\",\"title\":\"rocky2700\",
\"created_at\":\"2016-01-04T02:01:50.932Z\",\"updated_at\":\"2016-01-04T03:23:18.865Z\"}"
```

The client receives a 200/OK (nothing was created) and is able to access the payload information and header information if necessary.

```
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.response.code
 => "200"
> response.parsed_response
 => {"id"=>"123457", "title"=>"rocky2700",
     "created_at"=>"2016-01-04T02:01:50.932Z", "updated_at"=>"2016-01-04T03:23:18.865Z"}
```

This method is idempotent. If we issue the same request multiple times, the Rails framework is smart enough to notice a non-change and maintain the same `updated_at` until we actually make a change (see third line).

```
 => {"id"=>"123457", "title"=>"rocky2700", "created_at"=>"2016-01-04T02:01:50.932Z", "updated_at"=>"2016-01-
 => {"id"=>"123457", "title"=>"rocky2700", "created_at"=>"2016-01-04T02:01:50.932Z", "updated_at"=>"2016-01-
 => {"id"=>"123457", "title"=>"rocky2701", "created_at"=>"2016-01-04T02:01:50.932Z", "updated_at"=>"2016-01-
```

PATCH Clients can use PATCH in the place of PUT when it is more appropriate for the type of update being performed.

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

HEAD Lets say that we already have a copy of the document and do not need another copy unless it has changed – or we simply want to know if what we have is current. The client can issue an initial GET, store off the document and the `Etag` of the document.

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

The client can, instead issue a HEAD to return only the header information, which in this case, includes the `Etag` stating the document has not been changed.

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

If you observe the debug in the server log, you will notice that

- the debug for HEAD states it is using the show action and rendering with the movies/show.json.jbuilder.
- the query to the database still occurs except the document is not marshalled back.

```
Started HEAD "/movies/123457.json" for 127.0.0.1 at 2016-01-03 23:11:31 -0500
Processing by MoviesController#show as JSON
  Parameters: {"id"=>"123457"}
MONGODB | localhost:27017 | wsmovies_development.find | STARTED | {"find"=>"movies", "filter"=>{"_id"=>"1234
MONGODB | localhost:27017 | wsmovies_development.find | SUCCEEDED | 0.00032894600000000005s
  Rendered movies/show.json.jbuilder (0.2ms)
Completed 200 OK in 4ms (Views: 1.6ms | ActiveRecord: 0.0ms)
```

```
<- "HEAD /movies/123457.json HTTP/1.1\r\n
Accept: */*\r\n
User-Agent: Ruby\r\n
Connection: close\r\n
Host: localhost:3000\r\n
\r\n"
```

```
-> "HTTP/1.1 200 OK \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Content-Type: application/json; charset=utf-8\r\n"
-> "Etag: W/\"4cff78bec23ff12c4af51a97719009f1\"\r\n"
-> "Cache-Control: max-age=0, private, must-revalidate\r\n"
-> "X-Request-Id: 954f3161-7c85-437e-9178-cfb90f0af1e6\r\n"
-> "X-Runtime: 0.019517\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 03:58:16 GMT\r\n"
-> "Content-Length: 0\r\n"
-> "Connection: close\r\n"
-> "Set-Cookie: request_method=HEAD; path=/\r\n"
-> "\r\n"
```

If we make another non-update with PATCH, both PATCH and HEAD return an `Etag` that is identical to previous values.

```
> response=MoviesWS.patch("/movies/123457.json",
    :body=>{:movie=>{:title=>"rocky2702",:foo=>"bar"}}.to_json,
    :headers=>{"Content-Type"=>"application/json"})
> response.header["etag"]
 => "W/\"4cff78bec23ff12c4af51a97719009f1\""

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

When we make an actual change, but PATCH and HEAD report the new value for Etag.

```
> response=MoviesWS.patch("/movies/123457.json",
    :body=>{:movie=>{:title=>"rocky2703",:foo=>"bar"}}.to_json,
    :headers=>{"Content-Type"=>"application/json"})
> response.header["etag"]
```

```
 => "W/\"90b5b99f98305dfc6097c97273b6d38a\""

> response=MoviesWS.head("/movies/123457.json")
> response.header["etag"]
 => "W/\"90b5b99f98305dfc6097c97273b6d38a\""
```

When we put the document back to its previous state – note the Etag has changed and not reverted to the original value. That could be due to the timestamp being part of the Etag, but there is no way to tell how the calculation is really made unless we look deeper than we have time for here.

“‘ > response=MoviesWS.patch(“/movies/123457.json”, :body=>{:movie=>{:title=>“rocky2702”,:foo=>“bar”}}.to_json, :headers=>{“Content-Type”=>“application/json”}) > response.header[“etag”] => “W/"ce87f8f306e1e20fbde8d4fbd5d043b7"”

2.2.2 :271 > response=MoviesWS.head(“/movies/123457.json”) > response.header[“etag”] => “W/"ce87f8f306e1e20fbde8d4fbd5d043

DELETE Delete works as advertised. It accepts an `:id` parameter from the URI and removes that document from the database. The client simply needs to supply the correct URI. The result in this case is still a 2xx-level result, but since no payload was returned – the status code is set to 204/NO CONTENT.

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

```
<- "DELETE /movies/123457.json HTTP/1.1\r\n
Connection: close\r\n
Host: localhost:3000\r\n
\r\n"
```

```
$ rake routes | grep movies | grep -i delete
                 DELETE /movies/:movie_id/roles/:id(.:format)      movie_roles#destroy
                 DELETE /movies/:id(.:format)                      movies#destroy
```

```
  # DELETE /movies/1
  # DELETE /movies/1.json
  def destroy
    @movie.destroy
    respond_to do |format|
      format.json { head :no_content }
    end
  end
```

```
-> "HTTP/1.1 204 No Content \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Cache-Control: no-cache\r\n"
-> "X-Request-Id: 1a513bd4-c62e-4bd5-a047-6214eab511e4\r\n"
-> "X-Runtime: 0.010063\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 04:19:46 GMT\r\n"
-> "Connection: close\r\n"
-> "\r\n"
```

Notice that if you attempt to remove it a second time, the status changes to a 404/NOT FOUND.

```
> response=MoviesWS.delete("/movies/123457.json")
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
> response.response.code
 => "404"
```

Error Cases

Handling Exceptions If we attempt to create the same Movie with the same ID – our current configuration throws back an HTML page even if we set that we only accept application/json. That is obviously a development environment setting or an issue in how quiet error checking and load exceptions work.

```
<- "POST /movies.json HTTP/1.1\r\n
Content-Type: application/json\r\n
Accept: 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\"}}"

<- "{\"movie\":{\"id\":\"123457\",\"title\":\"rocky27\"}}"
-> "HTTP/1.1 500 Internal Server Error \r\n"
-> "Content-Type: text/html; charset=utf-8\r\n"
-> "Content-Length: 127079\r\n"
-> "X-Request-Id: c752fef2-5f20-49d9-a469-175af0ce3bda\r\n"
-> "X-Runtime: 0.192869\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 02:45:09 GMT\r\n"
-> "Connection: close\r\n"
-> "\r\n"
-> ""
```

If we add the following recue block to our create action we get a JSON error response back.

```
    rescue Mongo::Error::OperationFailure => e
      logger.info e
      respond_to do |format|
        format.html { render :new }
        format.json { render json: {msg:e.message}, status: :unprocessable_entity }
      end
```

```
-> "HTTP/1.1 422 Unprocessable Entity \r\n"
-> "X-Frame-Options: SAMEORIGIN\r\n"
-> "X-Xss-Protection: 1; mode=block\r\n"
-> "X-Content-Type-Options: nosniff\r\n"
-> "Content-Type: application/json; charset=utf-8\r\n"
-> "Cache-Control: no-cache\r\n"
-> "X-Request-Id: bbe0656e-266a-474c-bf1a-9c09cbc56b75\r\n"
-> "X-Runtime: 0.007916\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 03:13:58 GMT\r\n"
-> "Content-Length: 111\r\n"
```

```
-> "Connection: close\r\n"
-> "\r\n"
reading 111 bytes...
-> ""
-> "{\"msg\":\"E11000 duplicate key error index:
wsmovies_development.movies.$_id_ dup key: { : \\\"123457\\\" } (11000)\"}"
```

Method Not Supported Rails does not seem to have the notion of a URI existing and a method on that URI not existing. We get a 404/NOT FOUND event when we can show we can call other methods on the same URI.

Lets show a case where not all Methods are supported. In the following case, both /api/movies/:id and /movies/:id are handled by the same controller action.

```
Rails.application.routes.draw do
  resources :movies
  get "/api/movies/:id" => "movies#show", as: "mystuff"
```

We can get the same document from two different URIs

```
> response=MoviesWS.get("/movies/123457.json")
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.header["etag"]
 => "W/\"87fb946912c7c1056b53f0c9f6d8f842\""
```

```
> reponse=MoviesWS.get("/api/movies/123457.json")
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.header["etag"]
 => "W/\"87fb946912c7c1056b53f0c9f6d8f842\""
```

We can call PUT on the fully registered resource.

```
 > response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response=MoviesWS.put("/movies/123457.json",
    :body=>{:movie=>{:title=>"rocky2702"}}.to_json,
    :headers=>{"Content-Type"=>"application/json"})
```

We cannot call PUT on the custom path that only supports GET.

```
> response.response
 => #<Net::HTTPNotFound 404 Not Found  readbody=true>
> response=MoviesWS.put("/api/movies/123457.json",
   :body=>{:movie=>{:title=>"rocky2702"}}.to_json,
   :headers=>{"Content-Type"=>"application/json"})
```

OPTIONS Out of the box, Rails does not support the OPTIONS method will should return a list of methods supported by a URI. However, I have seen at least one gem advertised that looks to address this issue.

```
<- "OPTIONS /movies.json HTTP/1.1\r\nConnection: close\r\nHost: localhost:3000\r\n\r\n"
-> "HTTP/1.1 404 Not Found \r\n"
-> "Content-Type: text/html; charset=utf-8\r\n"
-> "Content-Length: 46390\r\n"
```

```
-> "X-Web-Console-Session-Id: 7217160b99f7e88fe282af2453c1f098\r\n"
-> "X-Request-Id: afb05ef3-20ee-40e7-b695-72d4d04b012d\r\n"
-> "X-Runtime: 0.067830\r\n"
-> "Server: WEBrick/1.3.1 (Ruby/2.2.2/2015-04-13)\r\n"
-> "Date: Mon, 04 Jan 2016 05:11:57 GMT\r\n"
-> "Connection: close\r\n"
-> "\r\n"
```

Method Idempotence

Representations

Rendering

- default rendering
  - controller derives data to render
  - view defaults to action requested
  - content form based on client specification
- controller-based action rendering
  - controller derives data to render
  - controller makes decision on action to render
  - controller may decision on format to render
    - supplying default format or overriding client specification

Example Core Create an example `Hello` controller with one `hello` action/view.

- Can use `rails g controller` command
- Default template generates several HTML-based files we won't be using

```
$ rails g controller Hello sayhello
$ find app | grep hello
app/views/hello
app/views/hello/sayhello.html.erb
app/controllers/hello_controller.rb
app/helpers/hello_helper.rb
app/assets/stylesheets/hello.scss
app/assets/javascripts/hello.coffee
```

- Contoller action method not required at this point
  - could be deleted without issue.

```
#app/controllers/hello_controller.rb
class HelloController < ApplicationController
  def sayhello
  end
end
```

- Route to action

```
#config/routes.rb
  get 'hello/sayhello'

$ rake routes | grep hello
  hello_sayhello GET    /hello/sayhello(.:format)                  hello#sayhello
```

Default Rendering and View Templates Place view template in `apps/views/(resource)` directory.

```
(action).(format).(handler suffix)
sayhello.json.jbuilder
sayhello.xml.builder
sayhello.text.raw
```

- JSON: `jbuilder`
- XML: 'builder'

Examples:

- `@msg` is not being set yet
- `text.raw` has limited functionality used in this way
  - used as a simple third example of the view handler formats

```
#app/views/hello/hello.json.jbuilder
json.msg @msg

#app/views/hello/hello.xml.builder
xml.msg @msg

#app/views/hello/hello.text.raw
raw text message from: app/views/hello/hello.text.raw
```

- GET request for URI
- handled by action (noop)
  - controller not deriving any data
  - view will have nothing in @msg to render (nil)
- rendered using caller-specified view format

```
> response=MoviesWS.get("/hello/sayhello.json")
> response.header.content_type
 => "application/json"
> response.body
 => "{\"msg\":null}"


> response=MoviesWS.get("/hello/sayhello.xml")
> response.header.content_type
 => "application/xml"
> response.body
 => "<msg/>"

> response=MoviesWS.get("/hello/sayhello.txt")
> response.header.content_type
 => "text/plain"
> response.body
 => "raw text message from: app/views/hello/sayhello.text.raw\n"
```

Adding Controller Drived Data

- derive data (from model)
- store in controller instance variable
- we now have use for that generated controller method

```
#app/controllers/hello_controller.rb
class HelloController < ApplicationController
  def sayhello
    @msg="hello world"
  end
end
```

```
> MoviesWS.get("/hello/sayhello.json").response.body
=> "{\"msg\":\"hello world\"}"

> MoviesWS.get("/hello/sayhello.xml").response.body
=> "<msg>hello world</msg>\n"
```

HTTParty Note

- `parsed_response` result for an XML response is a Ruby hash.

```
> response=MoviesWS.get("/hello/sayhello.xml")
> response.content_type
 => "application/xml"
> response.body
 => "<msg>hello world</msg>"
> response.parsed_response
 => {"msg"=>"hello world"}
```

Controller Action Rendering

- Controller can have more active say in action rendered

Example: Add a new URI and action.

- define a route and action that takes a parameter

```
#config/routes.rb
  get 'hello/say/:something' => "hello#say"
```

Add new controller action method with decision logic.

- one path uses views/hello/sayhello.(format).(handler suffix)
- another path uses views/hello/saygoodbye.(format).(handler suffix)
- last two paths use self-contained text.raw output

```
  #/hello/say/:something
  def say
    case params[:something]
    when "hello" then @msg="saying hello"; render action: :sayhello
    when "goodbye" then @msg="saying goodbye"; render action: :saygoodbye
    when "badword" then render nothing: true
    else
      render plain: "what do you want me to say?"
    end
  end
```

- Add our two supported view formats

```
#app/views/hello/saygoodbye.json.jbuilder
json.msg1 @msg
json.msg2 "so long!"

#app/views/hello/saygoodbye.xml.builder
xml.msg do
  xml.msg1 @msg
  xml.msg2 "so long!"
end
```

- Request `something=hello`
  - supplied by caller
- Resulting View: sayhello
  - determined by controller
- Format: `json` format.
  - specified by caller

```
> response=MoviesWS.get("/hello/say/hello.json")
> response.body
 => "{\"msg\":\"saying hello\"}"
> response.content_type
 => "application/json"
```

- Request `something=goodbye`
  - supplied by caller
- Resulting View: goodbye
  - determined by controller
- Format: `xml` format.
  - specified by caller

```
> response=MoviesWS.get("/hello/say/goodbye.xml")
> response.content_type
 => "application/xml"
> response.body
 => "<msg>\n  <msg1>saying goodbye</msg1>\n  <msg2>so long!</msg2>\n</msg>"
```

- Request `something=badword`
  - supplied by caller
- Resulting View: internal
  - determined by controller
- Format: `text/plain` format.
  - determined by controller
  - client specification not used

```
> response=MoviesWS.get("/hello/say/badword")
> response.content_type
 => "text/plain"
> response.body
 => ""
```

- Request `something=mmm`
  - supplied by caller
- Resulting View: internal
  - determined by controller
- Format: `text/plain` format.
  - determined by controller
  - client specification not used
  - would work with no specification

```
> response=MoviesWS.get("/hello/say/mmm")
#-or-
> response=MoviesWS.get("/hello/say/mmm.json")
> response.content_type
 => "text/plain"
> response.body
 => "what do you want me to say?"
```

Partials (using this as example to add roles+actor to movie)

- Define a (partial) template for rendering content
- Action-independent
- Default path: `app/views/(resource)/_(partial).(format).(handler suffix)`
  - app/views/actors/_actor.json.jbuilder
  - app/views/actors/_actor.xml.builder
- example partials:

```
#app/views/actors/_actor.json.jbuilder
json.extract! actor, :id, :name
json.source "partial: app/views/actors/_actor.json.jbuilder"

#app/views/actors/_actor.xml.builder
  xml.id actor.id
  xml.name actor.name
  xml.source "app/views/actors/_actor.xml.builder"
```

- example render (movies/show - views):

```
#app/views/movies/show.json.jbuilder
json.extract! @movie, :id, :title, :created_at, :updated_at

json.roles @movie.roles do |role|
  json.id role.id
  json.character role.character
  if role.actor
    json.partial! "actors/actor", locals: {actor: role.actor}
  end
end
```

```
#app/views/movies/show.xml.builder
xml.movie do
  xml.id @movie.id
  xml.title @movie.title
  xml.created_at @movie.created_at
  xml.updated_at @movie.updated_at
  xml.roles do
    @movie.roles.each do |role|
      xml.id role.id
      xml.character role.character
      if role.actor_id
        xml << render(partial: "actors/actor", locals: { actor: role.actor })
      end
    end
  end
end
```

- example JSON partial within `show` action rending:

```
> pp MoviesWS.get("/movies/12345.json").parsed_response
{"id"=>"12345",
 "title"=>"rocky25",
 "created_at"=>nil,
 "updated_at"=>"2016-01-03T17:05:36.066Z",
 "roles"=>
  [{"id"=>"0",
    "character"=>"rocky",
    "name"=>"sylvester stallone",
    "source"=>"partial: app/views/actors/_actor.json.jbuilder"},
   {"id"=>"1", "character"=>"challenger"}]}
```

- example XML partial within `show` action rendering:

```
> y MoviesWS.get("/movies/12345.xml").body
--- |
  <movie>
    <id>12345</id>
    <title>rocky25</title>
    <created_at/>
    <updated_at>2016-01-03 17:05:36 UTC</updated_at>
    <roles>
      <id>0</id>
      <character>rocky</character>
  <name>sylvester stallone</name>
  <source>app/views/actors/_actor.xml.builder</source>
      <id>1</id>
      <character>challenger</character>
    </roles>
  </movie>
```

Format Versions

Format Version Problem Description

- sooner or later, there is going to be a change

```
> Actor.where(:name=>{:$exists=>1}).each do |actor|
    names=actor.name.split
    actor.first_name=names[0]
    actor.last_name=names[1] if names.count > 1
    actor.name=nil
    actor.save
  end
```

- we make our implementations backward-compatible to not break legacy code/clients.

```
class Actor
  field :first_name, type: String
  field :last_name, type: String

  #backwards-compatible reader
  def name
    "#{first_name} #{last_name}"
  end
```

- legacy users are still accessing their data

```
> pp MoviesWS.get("/actors/100.json").parsed_response
{"id"=>"100",
 "name"=>"sylvester stallone",
 "created_at"=>nil,
 "updated_at"=>"2016-01-05T22:19:42.642Z"}
```

- but we lack a way to get new feature to users using JSON – without breaking legacy users

```
>   pp Actor.find("100").attributes
{"_id"=>"100",
 "name"=>nil,
 "updated_at"=>2016-01-05 22:19:42 UTC,
 "first_name"=>"sylvester",
 "last_name"=>"stallone"}
```

Format Version Solution (vendor media type)

- Use/define a different media type from format
  - `type / vnd.  producer's name followed by product's name [+suffix]`
- Register the format within your application, with a format abbreviation (e.g., `v2json`)

```
#config/initializers/mime_types.rb‘
# Be sure to restart your server when you modify this file.
# Add new mime types for use in respond_to blocks:
Mime::Type.register "application/vnd.myorgmovies.v2+json", :v2json
```

- Now have access to `version 2` of our model
  - note replacement of `name` with `first/last_name`

```
> response=(MoviesWS.get("/actors/100.v2json")
> response.content_type
 => "application/vnd.myorgmovies.v2+json"
```

```
> pp JSON.parse(response.body)
{"id"=>"100",
 "first_name"=>"sylvester",
 "last_name"=>"stallone",
 "created_at"=>nil,
 "updated_at"=>"2016-01-05T22:19:42.642Z"}
```

- While providing access to `version 1`
- Note: plan for v2 while building v1
  - define `v1json` from the start
  - possibly default `json` to "current" version

```
> response=(MoviesWS.get("/actors/100.json")
> response.content_type
 => "application/json"

> pp JSON.parse(response.body)
{"id"=>"100",
 "name"=>"sylvester stallone",
 "created_at"=>nil,
 "updated_at"=>"2016-01-05T22:19:42.642Z"}
```

Content Negotiation

Decent Reference: REST implies Content Negotiation

URL-based Content Type Problem Descriptoion

- Approaching REST means using hypermedia links
  - what are you putting into the URIs of your links?
    - /movies.json
    - /movies.v2json
- Some clients are asking for compression
  - /movies/json.gz ?
  - /gz/movies.json ?
- After a version change
  - movies are still movies (maybe represented differently)
  - actors are still actors (maybe represented differently)
  - you don't get a fresh set of clients
  - how can you keep your URIs from changing?
  - RMM level 3 clients are not suppose to assemble URLs from templates
    - they may have internal meaning
    - they are suppose to be opaque externally

Content Negotiation Solution

- Leave content type out of the URI
- Use HTTP Headers ("Accept" and "Content-Type") to express formats and encodings
- With proper headers supplied, no format required in URI

  - Content-Type: what we are sending in
  - Accept: what we are willing to accept

- Note too that the URI returned in the header does not include a format

```
> response=MoviesWS.post("/directors",
    :body=>{:director=>{:id=>"300",:first_name=>"Tim",:last_name=>"Burton"}}.to_json,
    :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
> response.response
 => #<Net::HTTPCreated 201 Created  readbody=true>

> response.header["location"]
 => "http://localhost:3000/directors/300"
> response.header.content_type
 => "application/json"

> pp JSON.parse(response.body)
{"id"=>"300",
 "first_name"=>"Tim",
 "last_name"=>"Burton",
 "created_at"=>"2016-01-06T01:16:26.302Z",
 "updated_at"=>"2016-01-06T01:16:26.302Z"}
```

- With proper headers specified, we can avoid the format in the GET URI
  - Accept: end-format we are willing to accept
  - Accept-Encoding: intermediate form encoded on wire

```
> response=MoviesWS.get("/directors",
    :headers=>{"Accept"=>"application/json","Accept-Encoding"=>"gzip"})

<- "GET /directors HTTP/1.1\r\n
Accept: application/json\r\n
Accept-Encoding: gzip\r\n
...

-> "HTTP/1.1 200 OK \r\n"
...
-> "Content-Type: application/json; charset=utf-8\r\n"
-> "Vary: Accept-Encoding\r\n"
-> "Content-Encoding: gzip\r\n"
...
reading 113 bytes...
-> ""
-> "\x1F\x8B\b\x00\xF2m\x8CV\x00\x03\x8A\xAEV\xCALQ\xB2R260P\x...
```

- Note result URI specifies JSON
  - generated show.json view hard-coded format to `:json`,

```
> pp JSON.parse(response.body)
[{"id"=>"300",
  "first_name"=>"Tim",
  "last_name"=>"Burton",
  "url"=>"http://localhost:3000/directors/300.json"}]

  json.url director_url(director, format: :json)
```

Adjustments

(too many to make with not enough bang for the buck to get to REST utopia. However, enough shown to indicate that it can be done and starting not far from the solution)

Negatives

- Unable to explore application through browser (because links do not have content types embedded in them).
- Counter-points
  - non-readlonly access to level 3 services require full suite of HTTP verbs
  - browsers do not support full HTTP suite
  - many application client tools available (e.g., curl, HTTParty).

Directory Assembly Notes Director is being created tp demonstrate eliminating of format information from the URI without changing previous examples. Think of it as a near 1:1 replacement with Actor except we are no showing any relationships with movie.

```
$ rails g scaffold Director id first_name last_name
```

```
Mongoid::Timestamps
```

JSON Marshaling

(not covering – although there is clearly a need)

XML Marshaling

(not covering – although there is clearly a need)

Headers and Status Codes

The following is an example use of headers to implement concurrency checks or idempotence. These same header fields can be used for cache management and most of the Rails helper methods are geared towards that purpose. In cache management, you only want to do things if they are out-of-date. In concurrency managemement you only wanted to do things when they are up-to-date. This flip of logic causes us to have to manually access the header fields and form our own logic.

State

- Etag
- Last-Modified Timestamp

Conditions

- If-Match: Etag
- If-None-Match: Etag
- If-UnModified-Since: Last-Modified Timestamp
- If-Modified-Since: Last-Modified Timestamp

Setting Headers: `fresh_when`

- Add `fresh_when` to augment return headers for returned model objects
  - Etag
  - Last-Modified

```
#app/controllers/movies_controller.rb
  def create
    @movie = Movie.new(movie_params)

    respond_to do |format|
      if @movie.save
        fresh_when(@movie)
        format.json { render :show, status: :created, location: @movie }
        format.v2json { render :show, status: :created, location: @movie }
      ...
```

- We issue a create
  - just happends to post as `application/json` and receive 'application/vnd.myogmovies.v2+json
  - get Etag value (this ends up being a different Etag source/algorithm than earlier default source/value)
  - get Last-Modified Timestamp

```
> response=MoviesWS.post("/movies",
    :body=>{:movie=>{:id=>"12347",:title=>"rocky27"}}.to_json,
    :headers=>{"Content-Type"=>"application/json",
               "Accept"=>"application/vnd.myorgmovies.v2+json"})
> response.response
 => #<Net::HTTPCreated 201 Created  readbody=true>
> response.content_type
 => "application/vnd.myorgmovies.v2+json"
 > pp JSON.parse(response.body)
{"id"=>"12347",
 "title"=>"rocky27",
 "created_at"=>"2016-01-06T02:45:37.958Z",
 "updated_at"=>"2016-01-06T02:45:37.958Z",
 "roles"=>[]}
```

- Etag and Last-Modified Header in Response

```
-> "HTTP/1.1 201 Created \r\n"
...
-> "Etag: \"2824e6c32d61a4d09c1632f54057b0a2\"\r\n"
-> "Last-Modified: Wed, 06 Jan 2016 02:45:37 GMT\r\n"
...

> response.header["etag"]
 => "\"2824e6c32d61a4d09c1632f54057b0a2\""
> response.header["last-modified"]
 => "Wed, 06 Jan 2016 02:45:37 GMT"
```

- Last-Modified base off Mongoid updated-at timestamp

```
> Movie.find("12347").updated_at
 => Wed, 06 Jan 2016 02:45:37 UTC +00:00
> Movie.find("12347").updated_at.usec
 => 958000
```

Getting Status

```
  # GET /movies/1
  def show
    fresh_when(@movie)
  end
```

- Etag and Last-Modified Headers are returned for GET

```
> response=MoviesWS.get("/movies/12347",:headers=>{"Accept"=>"application/vnd.myorgmovies.v2+json"})
> response.header["last-modified"]
 => "Wed, 06 Jan 2016 02:45:37 GMT"
 > response.header["etag"]
 => "\"1db42608e19f6e50209190f8ac7470d2\""
```

- Etag and Last-Modified Headers are returned for HEAD

```
 > response=MoviesWS.head("/movies/12347",:headers=>{"Accept"=>"application/json"})
> response.header["etag"]
 => "\"0258dce911d803a7ca3c394d83f52f9b\""
 > response.header["last-modified"]
 => "Wed, 06 Jan 2016 02:45:37 GMT"
```

Updates

- Can add headers any time responses for that resource are being returned
- However
  - Etags and Last-Modified are primarily designed for Internet cache management
  - Etags should be kept within a URI
  - Last-Modified is just a date, so we will rely on that later

```
  def update
    respond_to do |format|
      if @movie.update(movie_params)
        fresh_when(@movie)
        format.json { render :show, status: :ok, location: @movie }
        format.v2json { render :show, status: :ok, location: @movie }
```

- Make a change to the movie name

```
> response=MoviesWS.put("/movies/12347",
    :body=>{:movie=>{:title=>"rocky2700"}}.to_json,
    :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
```

- Etag and Last-Modified have changed

```
> response.header["etag"]
 => "\"fd9319ddeceb95b39af69b816705ee75\""
> response.header["last-modified"]
 => "Wed, 06 Jan 2016 03:22:33 GMT"
```

Concurrent or non-Idempotent Update Problem

- Nested resource updated to (not the only way to do it)
  - return header information of parent movie
  - return `:ok` instead of `:created`
  - signal update of movie and not creation of role
  - return location of movie and not the specific role
  - render the movie result instead of a `movie_role result`

```
#app/controllers/movie_roles_controller.rb
  # POST /movie/:movie_id/roles
  def create
    @movie_role = @movie.roles.build(movie_role_params)

    respond_to do |format|
      if @movie_role.save
        fresh_when(@movie)
        format.json { render :"movies/show", status: :ok, location: @movie }
...
```

- Current state obtained of movie

```
> response=MoviesWS.head("/movies/12347",:headers=>{"Accept"=>"application/json"})
> response.header["etag"]
 => "\"3ca361c249dfb78d10d21cdf6a5a69ed\""
 > response.header["last-modified"]
 => "Wed, 06 Jan 2016 03:44:55 GMT"
```

- New role added to movie

```
> response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"challenger"}}.to_json,
    :headers=>{"Content-Type"=>"application/json",
               "Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
> response.header["etag"]
 => "\"0b7edc03c813d03972d3802e04ceccfd\""
> response.header["last-modified"]
 => "Wed, 06 Jan 2016 03:44:55 GMT"
```

- New role added second time to movie

```
 > response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"challenger"}}.to_json,
    :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
 > response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
 > response.header["etag"]
 => "\"0b7edc03c813d03972d3802e04ceccfd\""
 > response.header["last-modified"]
 => "Wed, 06 Jan 2016 03:44:55 GMT"
```

- We now have two duplicate `challenger` roles

```
> response=MoviesWS.get("/movies/12347",:headers=>{"Accept"=>"application/json"})
 > response.header["etag"]
 => "\"3ca361c249dfb78d10d21cdf6a5a69ed\""
 > response.header["last-modified"]
 => "Wed, 06 Jan 2016 03:44:55 GMT"

> pp JSON.parse(response.body)
{"id"=>"12347",
 "title"=>"rocky27",
 "created_at"=>"2016-01-06T03:44:55.126Z",
 "updated_at"=>"2016-01-06T03:44:55.126Z",
 "roles"=>
  [{"id"=>{"$oid"=>"568c8e9ae301d0b6c8000002"}, "character"=>"challenger"},
   {"id"=>{"$oid"=>"568c9195e301d0b6c8000003"}, "character"=>"challenger"}]}
```

- Notice that `Etag` value changed, but `Last-Modified` did not
- Caused by data model definition
- Can fix with touch cascade (makes updates more expensive)

```
#app/models/movie_role.rb
class MovieRole
  ...
  embedded_in :movie, touch: true
```

- `Last-Modified` has now been updated

```
> response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"challenger"}}.to_json,
    :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
 > response.header["etag"]
 => "\"ff4bf757f62da4dba7fc7d062644d237\""
 > response.header["last-modified"]
 => "Wed, 06 Jan 2016 04:11:42 GMT"

* Notice in the server log debug that ‘Mongoid‘ required two calls to the DB to satisy the update
  - one to insert the role
  - one to update the timestamp

D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>"12347"},
      "u"=>{"$push"=>{"roles"=>{"_id"=>BSON::ObjectId('568c93fee301d0b6c8000004'),
      "character"=>"challenger"}}},...
D, | {"update"=>"movies", "updates"=>[{"q"=>{"_id"=>"12347"},
      "u"=>{"$set"=>{"updated_at"=>2016-01-06 04:11:42 UTC,
      "roles.2._id"=>BSON::ObjectId('568c93fee301d0b6c8000004'), "roles.2.character"=>"challenger"}},
```

Concurrent or non-Idempotent Update Solution

- Add `IF_UNMODIFIED_SINCE` check to update methods
- If supplied,
  - use `fresh_when` to populate response with current Etag and Last-Modified
  - continue if request date later than or equal to current state
  - else report conflict and make no changes

- Continue if not supplied or passes check
- Last-Modified is a string in RFC 2822-compliant date format
- Algorithm shows header being optional
  - could be made mandatory as well as adding other checks

```
  def create
    last_modified=request.env["HTTP_IF_UNMODIFIED_SINCE"]
    fresh_when(@movie)   if last_modified  #get header values if condition supplied
    Rails.logger.debug("if_unmodified_since=#{last_modified}, current=#{response.last_modified}")

    if !last_modified ||
       (DateTime.strptime(last_modified,"%a, %d %b %Y %T %z") >= response.last_modified)

      @movie_role = @movie.roles.build(movie_role_params)
      respond_to do |format|
        if @movie_role.save
          fresh_when(@movie)
          format.json { render :"movies/show", status: :ok, location: @movie }
        ...
    else
      render nothing: true, status: :conflict
    end
  ...
```

- Create movie
- Store off Last-Modified
  - can be re-obtained thru a GET or HEAD

```
> response=MoviesWS.post("/movies",
    :body=>{:movie=>{:id=>"12347",:title=>"rocky27"}}.to_json,
    :headers=>{"Content-Type"=>"application/json","Accept"=>"application/json"})
> response.response
 => #<Net::HTTPCreated 201 Created  readbody=true>
> last_modified=response.header["Last-Modified"]
 => "Wed, 06 Jan 2016 06:13:09 GMT"
```

- Add new role to movie, allowing service to generate ID
- Supply retained Last-Modified in the If-Unmodified-Since header

```
 > response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"challenger"}}.to_json,
    :headers=>{"Content-Type"=>"application/json",
               "Accept"=>"application/json",
               "If-UnModified-Since"=>last_modified})
```

- We see the value being passed in the header from client to server

```
<- "POST /movies/12347/roles HTTP/1.1\r\n
Content-Type: application/json\r\n
Accept: application/json\r\n
If-Unmodified-Since: Wed, 06 Jan 2016 06:13:09 GMT\r\n
...
```

- We see the debug on the server showing the request date and current state date being the same

```
Started POST "/movies/12347/roles" for 127.0.0.1 at 2016-01-06 01:16:14 -0500
...
if_unmodified_since=Wed, 06 Jan 2016 06:13:09 GMT, current=2016-01-06 06:13:09 UTC
```

- The change is made and 200/OK is returned along with new state of movie

```
> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>
 > response.header["Last-Modified"]
 => "Wed, 06 Jan 2016 06:16:14 GMT"

 > pp JSON.parse(response.body)
{"id"=>"12347",
 "title"=>"rocky27",
 "created_at"=>"2016-01-06T06:13:09.095Z",
 "updated_at"=>"2016-01-06T06:16:14.898Z",
 "roles"=>
  [{"id"=>{"$oid"=>"568cb12ee301d0c4e0000005"}, "character"=>"challenger"}]}
```

- Repeat request or a concurrent request attempts to repeat or make alternate change
- Supplies a Last-Modified date from a previous state in the If-Unmodified-Since header

```
> response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"challenger"}}.to_json,
    :headers=>{"Content-Type"=>"application/json",
               "Accept"=>"application/json",
               "If-UnModified-Since"=>last_modified})

* We see the old timestamp passing in the request from caller to server

<- "POST /movies/12347/roles HTTP/1.1\r\n
Content-Type: application/json\r\n
Accept: application/json\r\n
If-Unmodified-Since: Wed, 06 Jan 2016 06:13:09 GMT\r\n
...

* We see debug on the server reporting the request timestamp is
older than the current state's timestamp

Started POST "/movies/12347/roles" for 127.0.0.1 at 2016-01-06 01:22:38 -0500
...
if_unmodified_since=Wed, 06 Jan 2016 06:13:09 GMT, current=2016-01-06 06:16:14 UTC

* The controller reports the conflict and does not change any state

> response.response
 => #<Net::HTTPConflict 409 Conflict  readbody=true>
> response.header["Last-Modified"]
 => "Wed, 06 Jan 2016 06:16:14 GMT"

* Caller issues a GET (or HEAD) to obtain the current Last-Modified timestamp
  - GET would be used to review the state before making next change
  - HEAD could be used to blindly overwrite the state without obtaining it
* Received value supplied in next request
```

```
> last_modified=MoviesWS.get("/movies/12347",
    :headers=>{"Accept"=>"application/json"}).header["Last-Modified"]

> response=MoviesWS.post("/movies/12347/roles",
    :body=>{:movie_role=>{:character=>"adrian"}}.to_json,
    :headers=>{"Content-Type"=>"application/json",
               "Accept"=>"application/json",
               "If-UnModified-Since"=>last_modified})

* We see the server debug reporting the request timestamp and current state
timestamp being the same

Started POST "/movies/12347/roles" for 127.0.0.1 at 2016-01-06 01:29:29 -0500
...
if_unmodified_since=Wed, 06 Jan 2016 06:16:14 GMT, current=2016-01-06 06:16:14 UTC

* The server makes the change and reports 200/OK

> response.response
 => #<Net::HTTPOK 200 OK  readbody=true>

* The movie has our new role inserted (no dups!)

> pp JSON.parse(response.body)
{"id"=>"12347",
 "title"=>"rocky27",
 "created_at"=>"2016-01-06T06:13:09.095Z",
 "updated_at"=>"2016-01-06T06:29:29.530Z",
 "roles"=>
  [{"id"=>{"$oid"=>"568cb12ee301d0c4e0000005"}, "character"=>"challenger"},
   {"id"=>{"$oid"=>"568cb449e301d0c4e0000006"}, "character"=>"adrian"}]}
```