

Active Model Serializer

[New Attempt](#)

Due No Due Date **Points** 1 **Submitting** a website url

FORK

<https://github.com/learn-co-curriculum/phase-4-using-active-model-serializer/fork><https://github.com/learn-co-curriculum/phase-4-using-active-model-serializer><https://github.com/learn-co-curriculum/phase-4-using-active-model-serializer/issues/new>

Learning Goals

- Understand the purpose of a serializer in a JSON API application
- Configure `ActiveModel::Serializer` with a single model

Introduction

So far, we've learned how to create a Rails API and to set up our routes and controller actions to handle various requests and return the requested JSON. In this section, we'll learn how to customize the JSON that we return.

Let's start by taking a look at our movie app. To set up the app, run:


```
$ bundle install
$ rails db:migrate db:seed
$ rails s
```

We have two actions set up: index and show. If you navigate to `localhost:3000/movies/1`, you should see:

```
{
  "id": 1,
  "title": "The Color Purple",
  "year": 1985,
  "length": 154,
  "director": "Steven Spielberg",
  "description": "Whoopi Goldberg brings Alice Walker's Pulitzer Prize-winning fem",
  "poster_url": "https://pisces.bbystatic.com/image2/BestBuy_US/images/products/30",
  "category": "Drama",
  "discount": false,
  "female_director": false,
  "created_at": "2021-05-21T17:11:35.682Z",
  "updated_at": "2021-05-21T17:11:35.682Z"
}
```

Rails makes it very easy to provide this JSON: all we needed to do was set up a `show` route in `routes.rb`, and a `show` action in our controller. But so far, we have no control over specifically what information is returned. For example, we might decide that we don't need to include the `created_at` or `updated_at` attributes in our list. One way we could do this is by using Active Record's built-in `to_json` method in our controller. It might look something like this:

```
# app/controllers/movies_controller.rb
def show
  movie = Movie.find(params[:id])
  render json: movie.to_json(only: [:id, :title, :year, :length, :director, :descr
end
```




We can simplify matters with the following:

```
def show
  movie = Movie.find(params[:id])
  render json: movie.to_json(except: [:created_at, :updated_at])
end
```

This is fairly straightforward so far. But what if we also had a nested resource we wanted to include? For example, if we had a blogging app in which posts belong to authors, we might want to do something like this:

```
def show
  post = Post.find(params[:id])
  render json: post.to_json(only: [:title, :description, :id], include: [author: {
end
```



Even in this very simple case, you can see how building out JSON strings by hand would get to be very cumbersome — and very error-prone — very quickly.

But there's an additional problem with this approach: it does not exhibit good separation of concerns. Recall that, in a full-stack Rails app, the controller's job is to interact with the model to access whatever data is requested and then pass that data along to the View layer. The views are responsible for determining exactly how the information is presented to the user. The same should be true here: rather than depending on the controller to determine how the data is returned, that task should be handled elsewhere. Enter `ActiveModel::Serializer`.

ActiveModel::Serializer

`ActiveModel::Serializer` (or AMS) provides an easy way to customize how the JSON rendered by our controllers is structured. It is a very "Rails-y" tool, in that it uses a "convention over configuration" approach, and is consistent with separation of concerns. Let's take a look at how we can use it to render the JSON for our movie app.

Using AMS

First we need to add the gem:

```
# Gemfile
#...
gem 'active_model_serializers'
```

Run `bundle install` to activate the gem. Now we need to generate an `ActiveModel::Serializer` for our `Movie` model. Thankfully, the gem provides a generator for that. Drop into your console and run:

```
rails g serializer movie
```

Take a look at the generated `movie_serializer.rb` in the `app/serializers` directory. It should look something like this:

```
# app/serializers/movie_serializer.rb
class MovieSerializer < ActiveModel::Serializer
  attributes :id
end
```

To customize our JSON, we simply provide the list of `attributes` that we want to be included:

```
class MovieSerializer < ActiveModel::Serializer
  attributes :id, :title, :year, :length, :director, :description, :poster_url, :c
end
```

With this in place, we can return our `movies_controller` to its original state:

```
# app/controllers/movies_controller.rb
def show
  movie = Movie.find(params[:id])
  render json: movie
end
```

Much better!

AMS provides a convention-based approach to serializing our resources, which means that if we have a `Movie` model, we can also have a `MovieSerializer` serializer, and by default, Rails will use our serializer if we simply call `render json: movie` in our controller.

Now, if you return to the browser and navigate to `localhost:3000/movies` or `localhost:3000/movies/:id`, you'll see that we're rendering just the JSON we want.

Custom Methods

So far, we've used AMS to return the values of the attributes for our `Movie` instances. But AMS also allows us to customize the information returned using an instance method on the `MovieSerializer` class. For example, say we wanted to create a movie summary that consisted of the movie's title and the first 50 characters of its description.

Let's start by adding `summary` to the list of attributes. Next, we'll define our method. For now, Let's put a `byebug` in the method's body:

```
class MovieSerializer < ActiveModel::Serializer
  attributes :id, :title, :year, :length, :director, :description, :poster_url, :c

  def summary
    byebug
  end
end
```

Refresh the page in the browser so you drop into `byebug` and enter `self` at the `byebug` prompt. The `MovieSerializer` instance that's returned includes an `object` attribute which, in turn, contains the first movie instance. This means you can enter `self.object` in `byebug` to access the movie instance, and `self.object.<attribute_name>` to access a specific attribute.

With this information, let's enter `q` to break out of the `byebug`, and create our `summary` method:

```
def summary
  "#{self.object.title} - #{self.object.description[0..49]}..."
end
```

Restart the server and navigate back to `localhost:3000/movies` and you should see our new summary added at the end of the JSON.

Explicitly Specifying a Serializer

So far, we have depended on Rails naming conventions for our serializers. When we ran `rails g serializer movie`, the AMS gem automatically created a `MovieSerializer` class for us. Whenever we use `render json:` with a `Movie` instance or a collection of `Movie` instances, Rails will follow naming conventions and **implicitly** look for a serializer that matches the name of the model.

Sometimes, however, we might want to create a custom serializer that doesn't follow Rails naming conventions; for example, we might have multiple different serializers for our `Movie` class depending on what information our frontend application needs. In that case, we'll need to **explicitly** specify the serializer to be used.

Let's say, for example, that we decided we wanted to create a custom serializer solely for displaying our movie summary. First, let's create a new file, `movie_summary_serializer.rb`, and move our custom method into it:

```
class MovieSummarySerializer < ActiveModel::Serializer
  attributes :summary

  def summary
    "#{self.object.title} - #{self.object.description[0..49]}..."
  end
end
```

To use our summary, we'll add a new route to `routes.rb` :

```
# config/routes.rb
...
get '/movies/:id/summary', to: 'movies#summary'
```

And, finally, add a `summary` action to our controller. In it, we specify that we want to use our new serializer to render the requested information:

```
# app/controllers/movies_controller.rb
def summary
  movie = Movie.find(params[:id])
  render json: movie, serializer: MovieSummarySerializer
end
```

Now if you navigate to `localhost:3000/movies/1/summary` in the browser, you should see:


```
{
  "summary": "The Color Purple - Whoopi Goldberg brings Alice Walker's Pulitzer Pr
}
```

The code above allows us to easily display just our movie summary for a single movie. If we wanted to use our new custom serializer to render the full collection of movies, we would need to create another route and action:

```
# config/routes.rb
...
get '/movie_summaries', to: 'movies#summaries'

# app/controllers/movies_controller.rb
def summaries
  movies = Movie.all
  render json: movies, each_serializer: MovieSummarySerializer
end
```

The use of `each_serializer: MovieSummarySerializer` in our action tells the app to use our custom movie summary serializer to render each of the movies in the collection.

A note on breaking convention: by creating these custom routes, we are breaking REST conventions. One alternate way to structure this kind of feature and keep our routes and controllers RESTful would be to create a new controller, such as `Movies::SummaryController`. The creator of Rails, DHH, advocates for [this approach for managing sub-resources](http://jeromedalbert.com/how-dhh-organizes-his-rails-controllers/)  (<http://jeromedalbert.com/how-dhh-organizes-his-rails-controllers/>). Ultimately, it is up to you as the developer to decide which approach works best for a particular circumstance.

Conclusion

In this lesson, we learned that the `ActiveModel::Serializer` gem enables us to customize how we want our JSON to be rendered without sacrificing the Rails principles of "convention over configuration" and separation of concerns. We also learned how to implement AMS with a single model. In the next lesson, we'll look at using AMS to serialize associations.

Check For Understanding

Before you move on, make sure you can answer the following questions:

1. What do we mean when we say Active Model Serializer uses a convention-based approach?
2. What are some ways to break convention when using `ActiveModel::Serializer` ?

Resources

- [ActiveModel::Serializer Documentation](https://github.com/rails-api/active_model_serializers/tree/v0.10.6/docs)  (https://github.com/rails-api/active_model_serializers/tree/v0.10.6/docs)