This repository Search Pull requests Issues Gist

rails-api / active_model_serializers Watch 112 Star 4,166 Fork 1,161

Code Issues 123 Pull requests 35 Projects 1 Wiki Pulse Graphs

Branch: master active_model_serializers / docs / general / getting_started.md Find file Copy path

tricknotes Fix example code in `doc/general/getting_started.md` cec6478 on Mar 23

7 contributors

134 lines (96 sloc) 3.55 KB Raw Blame History

Back to Guides

Getting Started

Creating a Serializer

The easiest way to create a new serializer is to generate a new resource, which will generate a serializer at the same time:

```
$ rails g resource post title:string body:string
```

This will generate a serializer in `app/serializers/post_serializer.rb` for your new model. You can also generate a serializer for an existing model with the serializer generator:

```
$ rails g serializer post
```

The generated serializer will contain basic `attributes` and `has_many` / `has_one` / `belongs_to` declarations, based on the model. For example:

```
class PostSerializer < ActiveModel::Serializer
  attributes :title, :body

  has_many :comments
  has_one :author
end
```

and

```
class CommentSerializer < ActiveModel::Serializer
  attributes :name, :body

  belongs_to :post
end
```

The attribute names are a whitelist of attributes to be serialized.

The `has_many`, `has_one`, and `belongs_to` declarations describe relationships between resources. By default, when you serialize a `Post`, you will get its `Comments` as well.

For more information, see Serializers.

Namespaced Models

When serializing a model inside a namespace, such as `Api::V1::Post`, ActiveModelSerializers will expect the corresponding serializer to be inside the same namespace (namely `Api::V1::PostSerializer`).

Model Associations and Nested Serializers

When declaring a serializer for a model with associations, such as:

```ruby
class PostSerializer < ActiveModel::Serializer
  has_many :comments
end
```

ActiveModelSerializers will look for `PostSerializer::CommentSerializer` in priority, and fall back to `::CommentSerializer` in case the former does not exist. This allows for more control over the way a model gets serialized as an association of an other model.

For example, in the following situation:

```ruby
class CommentSerializer < ActiveModel::Serializer
  attributes :body, :date, :nb_likes
end

class PostSerializer < ActiveModel::Serializer
  has_many :comments
  class CommentSerializer < ActiveModel::Serializer
    attributes :body_short
  end
end
```

ActiveModelSerializers will use `PostSerializer::CommentSerializer` (thus including only the `:body_short` attribute) when serializing a `Comment` as part of a `Post`, but use `::CommentSerializer` when serializing a `Comment` directly (thus including `:body, :date, :nb_likes`).

Extending a Base `ApplicationSerializer`

By default, new serializers descend from `ActiveModel::Serializer`. However, if you wish to share behavior across your serializers, you can create an `ApplicationSerializer` at `app/serializers/application_serializer.rb`:

```ruby
class ApplicationSerializer < ActiveModel::Serializer
end
```

Then any newly-generated serializers will automatically descend from `ApplicationSerializer`.

```
$ rails g serializer post
```

Now generates:

```ruby
class PostSerializer < ApplicationSerializer
  attributes :id
end
```

Rails Integration

ActiveModelSerializers will automatically integrate with your Rails app, so you won't need to update your controller. This is a example of how the controller will look:

```ruby
class PostsController < ApplicationController

  def show
    @post = Post.find(params[:id])
    render json: @post
  end

end
```

If you wish to use Rails url helpers for link generation, e.g., `link(:resources) { resources_url }`, ensure your application sets `Rails.application.routes.default_url_options`.

```ruby
Rails.application.routes.default_url_options = {
    host: 'example.com'
}
```