FRONTEND FRIENDLY RAILS

Marcin Grzywaczewski

Frontend-friendly Rails

Arkency

© 2016 Arkency

Contents

Prepare JSON API endpoints for your API

Endpoints are the most important part of the backend for your frontend applications. No matter how sophisticated your backend can be, frontend is not concerned about it. All the frontend needs are well-designed endpoints where it can hit for data and commands.

The most common data standard of APIs nowadays is JSON. Ruby on Rails has a built-in support for crafting JSON responses called jbuilder[1]. It is good enough for not sophisticated endpoints - DSL is simple, yet powerful. It's easy to use. The problem starts when you want to consume existing best practices when it comes to crafting JSON responses.

Jbuilder power lies in its genericness - it allows you to craft *any* JSON response. If you want to use a community standard, you're left on your own. In this chapter you'll learn about alternatives which allow you to use JSON API[2] specification easily.

Why JSON API?

You can certainly specify and implement your own convention of structuring JSON responses. But there is the same problem with reinventing libraries and tools - you'll most likely will come with an implementation which is inferior than some industry standard. You can also have problem inside the team - each new team member will most likely be confused by the custom solution. You'll need to reintroduce these concepts every time you'll have a new colleague in your team.

JSON API solves many problems and comes with benefits:

- This is a standard. That means every developer who've worked with JSON API before will be familiar with it.
- There are client-side libraries which excel at parsing the resulting data. That means you need to write less code on the frontend side to process your endpoint responses.
- You don't need to reinvent the wheel. Somehow problematic cases like reporting of multiple errors or passing metadata not being the part of a resource are solved in an elegant way in JSON API.
- It is easy to implement more HATEOAS[3]-ish approach with JSON API, thanks to how a spec is designed. It comes with many benefits, especially you don't need to keep knowledge about endpoints on the frontend side. It "discovers" them by issuing HTTP calls to the backend as needed.

[1] https://github.com/rails/jbuilder

[2] http://jsonapi.org

[3] https://en.wikipedia.org/wiki/HATEOAS

I personally see no need of crafting my own JSON standard for applications. Before JSON API and similar specs I had a bad time with some decisions I've made during the process.

JSON API in Rails

There are many supported server-side Ruby libraries[4] that can help you with implementing endpoints conforming JSON API specification. Popular AMS (a.k.a. ActiveModel Serializers)[5] is coming with JSON API adapter in release candidate (RC) releases. Since you'll be likely looking for something stable, there is a library called jsonapi-serializers[6] that'll be used in this chapter.

The main idea behind creating proper responses (not only JSON API-compilant!) is all about *promoting* the concern of generating a resource representation to an object. Such objects (often called *serializers* or, more generic, *presenters*) are responsible for taking the resource object and create a representation out of it. Libraries like AMS or `jsonapi-serializers` are often just providing the DSL for such objects and an (usually private) implementation of serializing resource fields to the target format.

Knowing so, all you need to do is implementing such objects and use them as a JSON responses. There's more, though - like setting the media type. More on that later.

Example Relationships

To see how it can work in a real world example, let's create something more sophisticated than just one model. This is because explaining how to make JSON API without showing how to construct relationships is not complete at all.

Let's change the context a bit. Let's assume you have following models:

- `Dish`, which is a description of meal you can eat in the restaurant. It belongs to many orders.
- `Order` which has many dishes.
- `Waiter` which has many orders.
- `Tip` which belongs to an order.

This structure will be used to describe the whole process of creating serializers for your application.

Schema used for this examples

To make everything clear I provide a schema that was generated by my Rails app after applying migrations I made to model those resources. I'm using PostgreSQL here with built-in UUID type and UUID generations capabilities to make my work easier:

[4] http://jsonapi.org/implementations/#server-libraries-ruby
[5] https://github.com/rails-api/active_model_serializers
[6] https://github.com/fotinakis/jsonapi-serializers

```
ActiveRecord::Schema.define(version: 20160314192757) do

  # These are extensions that must be enabled in order to support this database
  enable_extension "plpgsql"
  enable_extension "pgcrypto"

  create_table "dishes", id: :uuid, default: "gen_random_uuid()", force: :cascad\
e do |t|
    t.string   "name",        null: false
    t.decimal  "price",       null: false
    t.text     "description", null: false
    t.datetime "created_at",  null: false
    t.datetime "updated_at",  null: false
  end

  add_index "dishes", ["name"], name: "index_dishes_on_name", unique: true, usin\
g: :btree

  create_table "order_dishes", id: :uuid, default: "gen_random_uuid()", force: :\
cascade do |t|
    t.uuid     "order_id",   null: false
    t.uuid     "dish_id",    null: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  add_index "order_dishes", ["order_id", "dish_id"], name: "index_order_dishes_o\
n_order_id_and_dish_id", unique: true, using: :btree

  create_table "orders", id: :uuid, default: "gen_random_uuid()", force: :cascad\
e do |t|
    t.uuid     "waiter_id",  null: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

  create_table "tips", id: :uuid, default: "gen_random_uuid()", force: :cascade \
do |t|
    t.uuid     "order_id",   null: false
    t.decimal  "amount",     null: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
```

```
43    end
44
45    create_table "waiters", id: :uuid, default: "gen_random_uuid()", force: :casca\
46  de do |t|
47      t.string   "name",       null: false
48      t.datetime "created_at", null: false
49      t.datetime "updated_at", null: false
50    end
51
52    add_index "waiters", ["name"], name: "index_waiters_on_name", unique: true, us\
53  ing: :btree
54
55    add_foreign_key "order_dishes", "dishes", on_delete: :cascade
56    add_foreign_key "order_dishes", "orders", on_delete: :cascade
57    add_foreign_key "orders", "waiters", on_delete: :cascade
58    add_foreign_key "tips", "orders", on_delete: :cascade
59  end
```

Installing jsonapi-serializers

To install `jsonapi-serializers` you need to follow a classic way of installing dependencies in Rails. Just add it to your `Gemfile`:

```
1  gem 'jsonapi-serializers', '~> 0.6.5'
```

Then, run `bundle install`. After that step you should be able to work with the library.

Crafting a serializer

To create a serializer, you need to create a plain old Ruby class and include `JSONAPI::Serializer` module to it:

```
1  class WaiterSerializer
2    include JSONAPI::Serializer
3  end
```

This is enough to make the most basic serialization:

```
waiter = Waiter.create!(name: "Mark")
# #<Waiter id: "fb936012-dd61-4b06-aeba-13def948edb9", name: "Mark",
# created_at: "2016-03-14 19:58:50", updated_at: "2016-03-14 19:58:50">

pp JSONAPI::Serializer.serialize(waiter)
# {"data"=>
#  {"id"=>"fb936012-dd61-4b06-aeba-13def948edb9",
#   "type"=>"waiters",
#   "attributes"=>{"name"=>"Mark"},
#   "links"=>{"self"=>"/waiters/fb936012-dd61-4b06-aeba-13def948edb9"}}}
```

As you can see there is a default `self` link defined with all fields as `attributes` of your response. You can provide a list of attributes serialized by yourself:

```
class WaiterSerializer
  include JSONAPI::Serializer
  attributes :name, :created_at, :updated_at
end
```

To obtain the following result:

```
pp JSONAPI::Serializer.serialize(waiter)
# {"data"=>
#  {"id"=>"fb936012-dd61-4b06-aeba-13def948edb9",
#   "type"=>"waiters",
#   "attributes"=>
#    {"name"=>"Mark",
#     "created-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00,
#     "updated-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00},
#   "links"=>{"self"=>"/waiters/fb936012-dd61-4b06-aeba-13def948edb9"}}}
```

Of course you can use methods from `Waiter` and serialize it too:

```ruby
class Waiter < ActiveRecord::Base
  has_many :orders,
           dependent: :destroy

  def tips_total
    orders.map do |order|
      (order.tip || NoTip.new).amount
    end.sum(BigDecimal.new(0))
  end
end
```

```ruby
class WaiterSerializer
  include JSONAPI::Serializer
  attributes :name, :created_at, :updated_at, :tips_total
end
```

```ruby
pp JSONAPI::Serializer.serialize(waiter)
#{"data"=>
#  {"id"=>"fb936012-dd61-4b06-aeba-13def948edb9",
#   "type"=>"waiters",
#   "attributes"=>
#    {"name"=>"Mark",
#     "created-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00,
#     "updated-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00,
#     "tips-total"=>#<BigDecimal:7fea18ea27c0,'0.0',9(27)>},
#     "links"=>{"self"=>"/waiters/fb936012-dd61-4b06-aeba-13def948edb9"}}}}
```

So far so good. You can also create a *dynamic* attribute on the serializer side:

```ruby
class WaiterSerializer
  include JSONAPI::Serializer

  attributes :name, :created_at, :updated_at, :tips_total

  attribute :polite_name do
    "Mr/Ms. #{object.name}"
  end
end
```

```
pp JSONAPI::Serializer.serialize(waiter)
#{"data"=>
#  {"id"=>"fb936012-dd61-4b06-aeba-13def948edb9",
#   "type"=>"waiters",
#   "attributes"=>
#    {"name"=>"Mark",
#     "created-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00,
#     "updated-at"=>Mon, 14 Mar 2016 19:58:50 UTC +00:00,
#     "tips-total"=>#<BigDecimal:7fea18ea27c0,'0.0',9(27)>},
#   "links"=>{"self"=>"/waiters/fb936012-dd61-4b06-aeba-13def948edb9"}}}
```

That's all about attributes. But JSON API really shines when it comes to defining links and relationships. Let's take a look.

Defining relationships between resources

To define a relationship in the simplest case you don't even need to provide a serializer for the relationship. By default only links to resources are provided:

```
class OrderSerializer
  include JSONAPI::Serializer

  has_many :dishes
  has_one :tip
end
```

```
order = Order.first
# #<Order id: "d212a3cd-9bec-4147-aa99-16adbf14cf65",
# waiter_id: "12423d77-613d-47ea-945f-d275d9d5b960",
# created_at: "2016-03-14 20:18:34", updated_at: "2016-03-14 20:18:34">

pp JSONAPI::Serializer.serialize(order)

#{"data"=>
#  {"id"=>"d212a3cd-9bec-4147-aa99-16adbf14cf65",
#   "type"=>"orders",
#   "links"=>{"self"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65"},
#   "relationships"=>
#    {"tip"=>
#      {"links"=>
```

```
#         {"self"=>
#           "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/tip",
#          "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/tip"}},
#      "dishes"=>
#       {"links"=>
#         {"self"=>
#           "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/dishes",
#          "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/dishes"}}}}}
```

As you can see, there are two links generated for each relationship - `self` which is formatted in a way that is not quite useful for most Rails apps and `related` which makes a lot more sense. You can either get rid of the `self` link for associations:

```
class OrderSerializer
  include JSONAPI::Serializer

  has_many :dishes
  has_one :tip

  def relationship_self_link(_)
    nil
  end
end
```

Or provide something having more sense than the default in Rails:

```
class OrderSerializer
  include JSONAPI::Serializer

  has_many :dishes
  has_one :tip

  def relationship_self_link(relationship_name)
    # for "dishes" relation it'll be "/dishes"
    "/#{relationship_name}" # index path for resources?
  end
end
```

Or, even better - provide Rails' URL helpers and stop relying on conventions:

```ruby
class RailsBaseSerializer
  include JSONAPI::Serializer

  protected
  def url_adapter
    Rails.application.routes.url_helpers
  end
end

class OrderSerializer < RailsBaseSerializer
  has_many :dishes
  has_one :tip

  def self_link
    url_adapter.order_path(id)
  end

  def relationship_self_link(relationship_name)
    case relationship_name
    when "dishes" then url_adapter.dishes_path
    when "tip" then url_adapter.order_tip_path(id)
    else nil
  end
end
```

If that's your kind of thing you can also start to serve URLs, not paths. It is especially useful if you have multiple services (a.k.a. microservices app) so there are URLs which can span many domains / subdomains.

You can also specify how relationship will be loaded. For example for `Waiter` you can do:

```ruby
class OrderSerializer < RailsBaseSerializer
  has_many :dishes
  has_one :tip
  has_one :waiter do
    Waiter.find_by(object.waiter_id)
  end

  def self_link
    url_adapter.order_path(id)
  end

  def relationship_self_link(relationship_name)
```

```
    # ...
  end
end
```

```
pp JSONAPI::Serializer.serialize(order)

#{"data"=>
#  {"id"=>"d212a3cd-9bec-4147-aa99-16adbf14cf65",
#   "type"=>"orders",
#   "links"=>{"self"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65"},
#   "relationships"=>
#    {"tip"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/tip",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/tip"}},
#     "waiter"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/waiter",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/waiter"}},
#     "dishes"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/dishes",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/dishes"}}}}}
```

Just in Rails there are conventions about relationships - by default they'll be searched under the same method as relationship name.

It's great to have your relationship defined in such *shallow* way. But there are moments when you need more deep tree as a response. You can achieve exactly that using `include` option while serializing.

First of all, you need to prepare a relationship serializer. By default `jsonapi-serializers` follows the convention that for a given resource named `Foo` the name of its serializer is `FooSerializer`. That can be changed of course - refer to docs for more details.

```ruby
class DishSerializer
  include JSONAPI::Serializer

  attributes :name, :price, :description
end
```

Then you can pass `include: ['dishes']` as a second argument of `serialize` while serializing `Order`:

```ruby
order = Order.preload(:dishes).first
pp JSONAPI::Serializer.serialize(order, include: ['dishes'])

#{"data"=>
#  {"id"=>"d212a3cd-9bec-4147-aa99-16adbf14cf65",
#   "type"=>"orders",
#   "links"=>{"self"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65"},
#   "relationships"=>
#    {"tip"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/tip",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/tip"}},
#     "waiter"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/waiter",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/waiter"}},
#     "dishes"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/dishes",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/dishes"},
#       "data"=>
#        [{"type"=>"dishes", "id"=>"b97c5cd3-e431-4f8d-a36e-77120230c9de"}]}}},
# "included"=>
#  [{"id"=>"b97c5cd3-e431-4f8d-a36e-77120230c9de",
#    "type"=>"dishes",
#    "attributes"=>
#     {"name"=>"Pasta",
#      "price"=>"#<BigDecimal:7fa037a39a38,'0.8E1',9(18)>",
#      "description"=>"Tasty pasta"},
#    "links"=>{"self"=>"/dishes/b97c5cd3-e431-4f8d-a36e-77120230c9de"}}]}
```

As you can see, `relationships` is populated with data *identifiers* (only `id` and `type` there). There is a new section called `included` where full resource descriptions are available.

This distinction between attributes, relationships, links and included resources is a great thing. Some clients may be unaware about any of those sections and will propably still do just fine.

I've used `.preload(:dishes)` while getting the first order. It is because I know I'll use `dishes` relationship really soon and I want to avoid 2 queries. That's a very good practice to avoid performance problems, especially if your resources count tend to be big.

Read more in bonus chapter: 3 ways to do eager loading (preloading) in Rails 3 & 4

Adding metadata to your response

There is also a possibility to add arbitrary metadata to your responses. It is especially useful if you'd like to provide some kind of pagination info or some off-the-wire data that can be still needed on the view.

For example in case of synchronization between frontend and backend; we return `timestamp` telling how far the data is synchronized. Next requests from frontend include those `timestamp`.

Our example contains more business related metadata.

```
1  class OrderSerializer < RailsBaseSerializer
2    include JSONAPI::Serializer
3
4    has_one :tip
5    has_many :dishes
6
7    has_one :waiter do
8      Waiter.find_by(object.waiter_id)
9    end
10
11   def meta
12     { "most_popular_dish_id" => Dish.most_popular.id }
13   end
14
15   def self_link
16     url_adapter.order_path(id)
17   end
18 end
```

And the result:

```
order = Order.first
pp JSONAPI::Serializer.serialize(order)

#{"data"=>
#  {"id"=>"d212a3cd-9bec-4147-aa99-16adbf14cf65",
#   "type"=>"orders",
#   "links"=>{"self"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65"},
#   "relationships"=>
#    {"tip"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/tip",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/tip"}},
#     "waiter"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/waiter",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/waiter"}},
#     "dishes"=>
#      {"links"=>
#        {"self"=>
#          "/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/relationships/dishes",
#         "related"=>"/orders/d212a3cd-9bec-4147-aa99-16adbf14cf65/dishes"}}},
#   "meta"=>{"most_popular_dish"=>"b97c5cd3-e431-4f8d-a36e-77120230c9de"}}}
```

The best part of JSON API is that you don't need to implement all features that are expected and covered by the spec. You can add features iteratively and you can be sure that most things will be achievable out of the box following the spec. No bikeshedding and an additional benefit that there are specialized clients to make your work even easier on the frontend side!

Integrating `jsonapi-serializers` with controllers

Before we've just seen how we can check the serializing results inside the Rails console. But really the biggest question is - how to integrate `jsonapi-serializers` with controllers? Apart from generating responses

```
render json: JSONAPI::Serializer.serialize(a_thing)
```

you need to take care of the valid content type.

JSON API specification specifies[7] that all requests to JSON API endpoints should have a content type `application/vnd.api+json`. The same with responses - your requests should have `application/vnd.api+json` in both `Accept` and `Content-Type` headers.

Your API must respond with the same `Content-Type` when sending back JSON API-compliant responses. This must be configured in your Rails application.

It is a little inconvenient, but powerful. Just think about it - if you have an API which is not conforming JSON API spec and your clients still use it, all you need to do is to provide yet another block to your `respond_to`. Your old API will still operate normally, but you'll be able to accept requests which are interested in JSON API format instead the old one. It is a natural way to introduce JSON API endpoints iteratively in your code!

So, let's do it. First of all, we need to tell Rails that there will be a new content type of requests and we'll call it `:jsonapi`. Put it inside your `config/initializers/mime_types.rb`:

```
1  Mime::Type.register "application/vnd.api+json", :jsonapi
```

Remember to restart the server after introducing this (and next) changes - otherwise they won't get loaded.

Right now you've *registered* a new content type. You can use it in your `respond_to` blocks like this:

```
1  class ExampleController < ApplicationController
2    def index
3      respond_to do |format|
4        format.jsonapi do
5          # ...
6        end
7      end
8    end
9  end
```

This is not the end, unfortunately. You'll be most likely using JSON as your parameters for `POST/PUT/DELETE` params in requests. Rails cannot figure out how to parse such parameters from requests coming with `application/vnd.api+json` content type. We need to introduce the *parser* for our parameters.

The parameters parsing is done by one of Rails *middlewares*. You can get a list of them by running the `bundle exec rake middleware` command:

[7] http://jsonapi.org/format/#content-negotiation

```
bundle exec rake middleware

use Rack::Sendfile
use ActionDispatch::Static
use Rack::Lock
use #<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007f960bc4fd58>
use Rack::Runtime
use Rack::MethodOverride
use ActionDispatch::RequestId
use Rails::Rack::Logger
use ActionDispatch::ShowExceptions
use WebConsole::Middleware
use ActionDispatch::DebugExceptions
use ActionDispatch::RemoteIp
use ActionDispatch::Reloader
use ActionDispatch::Callbacks
use ActiveRecord::Migration::CheckPending
use ActiveRecord::ConnectionAdapters::ConnectionManagement
use ActiveRecord::QueryCache
use ActionDispatch::Cookies
use ActionDispatch::Session::CookieStore
use ActionDispatch::Flash
use ActionDispatch::ParamsParser
use Rack::Head
use Rack::ConditionalGet
use Rack::ETag
run FrontendFriendlyRails::Application.routes
```

The `ActionDispatch::ParamsParser` is responsible for parsing parameters. What you need to do is to swap this middleware which Rails provide by default with the new one containing parser you provide. To cut story short, put the following piece of code in `config/initializers/mime_types.rb`:

```
# substitute YourAppName with real name from config/application.rb
middlewares = YourAppName::Application.config.middleware
middlewares.swap(ActionDispatch::ParamsParser, ActionDispatch::ParamsParser, {
  Mime::Type.lookup('application/vnd.api+json') => lambda do |body|
    ActiveSupport::JSON.decode(body)
  end
})
```

Let's take a look at this code in a step by step manner:

1. First of all, the variable called `middlewares` is created. It is an object of `MiddlewareStackProxy`[8] type which represents a chain of your loaded middlewares.
2. `swap` is a function to replace the chosen middleware with another middleware. In this use case we're replacing the default `ActionDispatch::ParamsParser` middleware with the same type of middleware, but we're recreating it with custom arguments. `swap` also takes care of putting the middleware in the same place that the previous middleware sat before - that can avoid us subtle errors that could be possible with wrong order of middlewares.
3. The `parsers` object is keyed with identifiers of a content type which can be accessed using `Mime::Type.lookup` method. A value is a lambda (an in-place function) that will be called upon request's body every time the new request arrives - in this case it is just calling method for parsing the body as JSON. The result should be an object representing parameters.

You can read more about this in this blogpost.

After those steps you're ready to issue requests in an JSON API formats from your frontend. But first, create endpoints!

```
class ApplicationController
  private

  def jsonapi_collection(collection, options = {})
    JSONAPI::Serializer.serialize(collection, options.merge(is_collection: true))
  end

  def jsonapi_resource(resource, options = {})
    JSONAPI::Serializer.serialize(resource, options)
  end

  # Consult http://jsonapi.org/format/#error-objects for more sophisticated resp\
onses.
  def jsonapi_error(*error_objects)
    {
      errors: error_objects
    }
  end
end

class OrdersController < ApplicationController
  def index
    respond_to do |format|
      format.jsonapi { render json: jsonapi_collection(all_orders) }
```

[8] http://api.rubyonrails.org/classes/Rails/Configuration/MiddlewareStackProxy.html

```
25        # ...
26      end
27    end
28
29    def show
30      order = Order.find(params[:id])
31      respond_to do |format|
32        format.jsonapi { render json: jsonapi_resource(order) }
33        # ...
34      end
35    rescue ActiveRecord::RecordNotFound => err
36      respond_to do |format|
37        format.jsonapi do
38          render json: jsonapi_error({ title: "RecordNotFound",
39                          detail: err.message }),
40                 status: :not_found
41        end
42        # ...
43      end
44    end
45
46    private
47
48    def all_orders
49      Order.preload(:tip, :dishes)
50    end
51  end
```

As you can see, it is very similar to what we did in Rails console. The only thing is that you need to craft error messages by yourself. It is made by the `jsonapi_error` method in the code above. But in fact it is very easy so you should not have a problem with the format of errors[9] described in the JSON API spec. Just remember both `data` and `errors` cannot be together in one response!

Configuring clients to issue JSON API requests

Of course your client needs also to be aware that you'll be working with JSON API. All you need to do is providing appropriate headers. Examples here will be showing how to do it with two most popular options with Rails - jQuery & more modern `fetch API`[10].

[9] http://jsonapi.org/format/#error-objects

[10] https://developer.mozilla.org/en/docs/Web/API/Fetch_API

jQuery Client

```
// GET
$.ajax({
  url: "/orders",
  type: "GET",
  headers: {
    "Accept": "application/vnd.api+json",
    "Content-Type": "application/vnd.api+json"
  }
})

// POST
$.ajax({
  url: "/orders",
  type: "POST",
  headers: {
    "Accept": "application/vnd.api+json",
    "Content-Type": "application/vnd.api+json"
  },
  processData: false,
  data: JSON.stringify({ // Your parameters
    order: {
      // ...
    }
  })
})
// The same goes with PUT/PATCH/DELETE - just change type field accordingly.
```

WHATWG fetch client

```
// GET
fetch("/orders",
      { method: "GET",
        headers: {
          'Accept': 'application/vnd.api+json',
          'Content-Type': 'application/vnd.api+json'
        }
    });

// POST/PUT/PATCH/DELETE
fetch("/orders",
```

```
    { method: "POST", // or PUT, or PATCH, or DELETE
      headers: {
        'Accept': 'application/vnd.api+json',
        'Content-Type': 'application/vnd.api+json'
    },
    body: JSON.stringify({ // Your parameters
      order: {
        // ...
      }
    })
  });
```

Tips and tricks

- You can provide `include` option from parameters passed by the client. This way your clients will be able to specify what they need from your endpoint. Just remember to whitelist those collections.
- There is a good practice of providing a response for your root (`/`) path specifying all possible endpoints you may hit in your API. There is also a JSON API object[11] that can be returned in your response at the top level - try to include it at least on the root path as a part of your response.
- Your API resources can be different from your real resources on the database. Take the advantage over this fact and try to provide resources which *speaks* more with your frontend, not backend.
- If you want to keep CSRF protection, `meta` top level is a great place to send your CSRF tokens. You can do it like this:

```
class ResourceController
  def show
    # ...
    render json: jsonapi_resource(resource, meta: csrf_meta_section)
  end

  private
  def csrf_meta_section
    { "csrf_token" => form_authenticity_token }
  end
end
```

- Try to preload your relationships before serialization to avoid too many queries problem. Read a bonus about it if you want to know more.

[11] http://jsonapi.org/format/#document-jsonapi-object

Summary

JSON API can be integrated with Rails in an easy way thanks to the great `jsonapi-serializers` gem. There is also some configuration effort needed to be made about defining a new content type, but it is quite straightforward. You get a lot by sticking to the spec - you can grow your new API in an iterative way in legacy codebases too, and you got a lot of ways you can extend your API in an easy way, without inventing ways how to do it.

3 ways to do eager loading (preloading) in Rails 3 & 4

You are probably already familiar with the method `#includes` for eager loading data from database if you are using Rails and ActiveRecord. But do you know why you someties get few small and nice SQL queries and sometimes one giant query with every table and column renamed? And do you know about `#preload` and `#eager_load` which can help you achieve the same goal? Are you aware of what changed in Rails 4 in that matter? If not, sit down and listen. This lesson won't take long and will help you clarify some aspects of eager loading that you might not be yet familiar with.

Let's start with our Active Record class and associations definitions that we are going to use throughout the whole post:

```
1  class User < ActiveRecord::Base
2    has_many :addresses
3  end
4
5  class Address < ActiveRecord::Base
6    belongs_to :user
7  end
```

And here is the seed data that will help us check the results of our queries:

```
1  rob = User.create!(name: "Robert Pankowecki", email: "robert@example.org")
2  bob = User.create!(name: "Bob Doe", email: "bob@example.org")
3
4  rob.addresses.create!(country: "Poland", city: "Wrocław", postal_code: "55-555",\
5   street: "Rynek")
6  rob.addresses.create!(country: "France", city: "Paris", postal_code: "75008", st\
7  reet: "8 rue Chambiges")
8  bob.addresses.create!(country: "Germany", city: "Berlin", postal_code: "10551", \
9  street: "Tiergarten")
```

Rails 3

Typically, when you want to use the eager loading feature you would use the `#includes` method, which Rails encouraged you to use since Rails2 or maybe even Rails1 ;). And that works like a charm doing 2 queries:

```
User.includes(:addresses)
#  SELECT "users".* FROM "users"
#  SELECT "addresses".* FROM "addresses" WHERE "addresses"."user_id" IN (1, 2)
```

So what are those two other methods for? First let's see them in action.

```
User.preload(:addresses)
#  SELECT "users".* FROM "users"
#  SELECT "addresses".* FROM "addresses" WHERE "addresses"."user_id" IN (1, 2)
```

Apparently `#preload` behave just like `#includes`. Or is it the other way around? Keep reading to find out.

And as for the `#eager_load`:

```
User.eager_load(:addresses)
#  SELECT
#  "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_r2, "us\
ers"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
#  "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses"."count\
ry" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS t1_r4,\
 "addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "addresses"."up\
dated_at" AS t1_r7
#  FROM "users"
#  LEFT OUTER JOIN "addresses" ON "addresses"."user_id" = "users"."id"
```

It is a completely different story, isn't it? The whole mystery is that Rails has 2 ways of preloading data. One is using separate db queries to obtain the addtional data. And one is using one query (with `left join`) to get them all.

If you use `#preload`, it means you always want separate queries. If you use `#eager_load` you are doing one query. So what is `#includes` for? It decides for you which way it is going to be. You let Rails handle that decision. What is the decision based on, you might ask. It is based on query conditions. Let's see an example where `#includes` delegates to `#eager_load` so that there is one big query only.

```
User.includes(:addresses).where("addresses.country = ?", "Poland")
User.eager_load(:addresses).where("addresses.country = ?", "Poland")

# SELECT
# "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_r2, "use\
rs"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
# "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses"."countr\
y" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS t1_r4, \
"addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "addresses"."upd\
ated_at" AS t1_r7
# FROM "users"
# LEFT OUTER JOIN "addresses"
# ON "addresses"."user_id" = "users"."id"
# WHERE (addresses.country = 'Poland')
```

In the last example Rails detected that the condition in `where` clause is using columns from preloaded (included) table names. So `#includes` delegates the job to `#eager_load`. You can always achieve the same result by using the `#eager_load` method directly.

What happens if you instead try to use `#preload` explicitly?

```
User.preload(:addresses).where("addresses.country = ?", "Poland")
#  SELECT "users".* FROM "users" WHERE (addresses.country = 'Poland')
#
#  SQLite3::SQLException: no such column: addresses.country
```

We get an exception because we haven't joined `users` table with `addresses` table in any way.

Is this intention revealing?

If you look at our example again

```
User.includes(:addresses).where("addresses.country = ?", "Poland")
```

you might wonder, what is the original intention of this code. What did the author mean by that? What are we trying to achieve here with our simple Rails code:

- Give me users with polish addresses and preload only polish addresses
- Give me users with polish addresses and preload all of their addresses
- Give me all users and their polish addresses.

Do you know which goal we achieved? The first one. Let's see if we can achieve the second and the third ones.

Is `#preload` any good?

Our current goal: *Give me users with polish addresses but preload all of their addresses. I need to know all addreeses of people whose at least one address is in Poland.*

We know that we need only users with polish addresses. That itself is easy: `User.joins(:addresses).where("addres = ?", "Poland")` and we know that we want to eager load the addresses so we also need `includes(:addresses)` part right?

```
r = User.joins(:addresses).where("addresses.country = ?", "Poland").includes(:ad\
dresses)

r[0]
#=> #<User id: 1, name: "Robert Pankowecki", email: "robert@example.org", create\
d_at: "2013-12-08 11:26:24", updated_at: "2013-12-08 11:26:24">

r[0].addresses
# [
#   #<Address id: 1, user_id: 1, country: "Poland", street: "Rynek", postal_code\
: "55-555", city: "Wrocław", created_at: "2013-12-08 11:26:50", updated_at: "201\
3-12-08 11:26:50">
# ]
```

Well, that didn't work exactly like we wanted. We are missing the user's second address that expected to have this time. Rails still detected that we are using included table in where statement and used `#eager_load` implementation under the hood. The only difference compared to previous example is that is that Rails used `INNER JOIN` instead of `LEFT JOIN`, but for that query it doesn't even make any difference.

```
SELECT
"users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_r2, "users\
"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
"addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses"."country"\
 AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS t1_r4, "a\
ddresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "addresses"."updat\
ed_at" AS t1_r7
FROM "users"
INNER JOIN "addresses"
ON "addresses"."user_id" = "users"."id"
WHERE (addresses.country = 'Poland')
```

This is that kind of situation where you can outsmart Rails and be explicit about what you want to achieve by directly calling `#preload` instead of `#includes`.

```
r = User.joins(:addresses).where("addresses.country = ?", "Poland").preload(:add\
resses)
# SELECT "users".* FROM "users"
# INNER JOIN "addresses" ON "addresses"."user_id" = "users"."id"
# WHERE (addresses.country = 'Poland')

# SELECT "addresses".* FROM "addresses" WHERE "addresses"."user_id" IN (1)

r[0]
# [#<User id: 1, name: "Robert Pankowecki", email: "robert@example.org", created\
_at: "2013-12-08 11:26:24", updated_at: "2013-12-08 11:26:24">]

r[0].addresses
# [
#  <Address id: 1, user_id: 1, country: "Poland", street: "Rynek", postal_code: \
"55-555", city: "Wrocław", created_at: "2013-12-08 11:26:50", updated_at: "2013-\
12-08 11:26:50">,
#  <Address id: 3, user_id: 1, country: "France", street: "8 rue Chambiges", pos\
tal_code: "75008", city: "Paris", created_at: "2013-12-08 11:36:30", updated_at:\
 "2013-12-08 11:36:30">]
# ]
```

This is exactly what we wanted to achieve. Thanks to using `#preload` we are no longer mixing which users we want to fetch with what data we would like to preload for them. And the queries are plain and simple again.

Preloading subset of association

The goal of the next exercise is: *Give me all users and their polish addresses.*

To be honest, I never like preloading only a subset of association because some parts of your application probably assume that it is fully loaded. It might only make sense if you are getting the data to display it.

I prefer to add the condition to the association itself:

```
class User < ActiveRecord::Base
  has_many :addresses
  has_many :polish_addresses, conditions: {country: "Poland"}, class_name: "Addr\
ess"
end
```

And just preload it explicitely using one way:

```
r = User.preload(:polish_addresses)

# SELECT "users".* FROM "users"
# SELECT "addresses".* FROM "addresses" WHERE "addresses"."country" = 'Poland' A\
ND "addresses"."user_id" IN (1, 2)

r

# [
#   <User id: 1, name: "Robert Pankowecki", email: "robert@example.org", created\
_at: "2013-12-08 11:26:24", updated_at: "2013-12-08 11:26:24">
#   <User id: 2, name: "Bob Doe", email: "bob@example.org", created_at: "2013-12\
-08 11:26:25", updated_at: "2013-12-08 11:26:25">
# ]

r[0].polish_addresses

# [
#   #<Address id: 1, user_id: 1, country: "Poland", street: "Rynek", postal_code\
: "55-555", city: "Wrocław", created_at: "2013-12-08 11:26:50", updated_at: "201\
3-12-08 11:26:50">
# ]

r[1].polish_addresses
# []
```

or another:

```
r = User.eager_load(:polish_addresses)

# SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_r\
2, "users"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
#        "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses".\
"country" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS \
t1_r4, "addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "addresse\
s"."updated_at" AS t1_r7
# FROM "users"
# LEFT OUTER JOIN "addresses"
# ON "addresses"."user_id" = "users"."id" AND "addresses"."country" = 'Poland'

r
# [
```

```
#   #<User id: 1, name: "Robert Pankowecki", email: "robert@example.org", create\
d_at: "2013-12-08 11:26:24", updated_at: "2013-12-08 11:26:24">,
#   #<User id: 2, name: "Bob Doe", email: "bob@example.org", created_at: "2013-1\
2-08 11:26:25", updated_at: "2013-12-08 11:26:25">
# ]

r[0].polish_addresses
# [
#   #<Address id: 1, user_id: 1, country: "Poland", street: "Rynek", postal_code\
: "55-555", city: "Wrocław", created_at: "2013-12-08 11:26:50", updated_at: "201\
3-12-08 11:26:50">
# ]

r[1].polish_addresses
# []
```

What should we do when we only know at runtime about the association conditions that we would like to apply? I honestly don't know. Please tell me in the comments if you found it out.

The ultimate question

You might ask: *What is this stuff so hard?* I am not sure but I think most ORMs are build to help you construct single query and load data from one table. With eager loading the situation gest more complicated and we want load multiple data from multiple tables with multiple conditions. In Rails we are using chainable API to build 2 or more queries (in case of using `#preload`).

What kind of API would I love? I am thinking about something like:

```
User.joins(:addresses).where("addresses.country = ?", "Poland").preload do |user\
s|
  users.preload(:addresses).where("addresses.country = ?", "Germany")
  users.preload(:lists) do |lists|
    lists.preload(:tasks).where("tasks.state = ?", "unfinished")
  end
end
```

I hope you get the idea :) But this is just a dream. Let's get back to reality...

Rails 4 changes

... and talk about what changed in Rails 4.

```
class User < ActiveRecord::Base
  has_many :addresses
  has_many :polish_addresses, -> {where(country: "Poland")}, class_name: "Addres\
s"
end
```

Rails now encourages you to use the new lambda syntax for defining association conditions. This is very good because I have seen many times errors in that area where the condition were interpreted only once when the class was loaded.

It is the same way you are encouraged to use lambda syntax or method syntax to express scope conditions.

```
# Bad, Time.now would be always the time when the class was loaded
# You might not even spot the bug in development because classes are
# automatically reloaded for you after changes.
scope :from_the_past, where("happens_at <= ?", Time.now)

# OK
scope :from_the_past, -> { where("happens_at <= ?", Time.now) }

# OK
def self.from_the_past
  where("happens_at <= ?", Time.now)
end
```

In our case the condition `where(country: "Poland")` is always the same, no matter wheter interpreted dynamically or once at the beginning. But it is good that rails is trying to make the syntax coherent in both cases (association and scope conditions) and protect us from the such kind of bugs.

Now that we have the syntax changes in place, we can check for any differences in the behavior.

```
User.includes(:addresses)
#  SELECT "users".* FROM "users"
#  SELECT "addresses".* FROM "addresses" WHERE "addresses"."user_id" IN (1, 2)

User.preload(:addresses)
#  SELECT "users".* FROM "users"
#  SELECT "addresses".* FROM "addresses" WHERE "addresses"."user_id" IN (1, 2)

User.eager_load(:addresses)
#  SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_\
```

```
r2, "users"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
#         "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses"\
."country" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS\
 t1_r4, "addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "address\
es"."updated_at" AS t1_r7
#  FROM "users"
#  LEFT OUTER JOIN "addresses"
#  ON "addresses"."user_id" = "users"."id"
```

Well, this looks pretty much the same. No surprise here. Let's add the condition that caused us so much trouble before:

```
User.includes(:addresses).where("addresses.country = ?", "Poland")

#DEPRECATION WARNING: It looks like you are eager loading table(s)
# (one of: users, addresses) that are referenced in a string SQL
# snippet. For example:
#
#    Post.includes(:comments).where("comments.title = 'foo'")
#
# Currently, Active Record recognizes the table in the string, and knows
# to JOIN the comments table to the query, rather than loading comments
# in a separate query. However, doing this without writing a full-blown
# SQL parser is inherently flawed. Since we don't want to write an SQL
# parser, we are removing this functionality. From now on, you must explicitly
# tell Active Record when you are referencing a table from a string:
#
#   Post.includes(:comments).where("comments.title = 'foo'").references(:comment\
s)
#
# If you don't rely on implicit join references you can disable the
# feature entirely by setting `config.active_record.disable_implicit_join_refere\
nces = true`. (

# SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_r\
2, "users"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
#        "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses".\
"country" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS \
t1_r4, "addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "addresse\
s"."updated_at" AS t1_r7
# FROM "users"
# LEFT OUTER JOIN "addresses" ON "addresses"."user_id" = "users"."id"
# WHERE (addresses.country = 'Poland')
```

Wow, now that is quite a verbose deprection :) I recommend that you read it all because it explains the situation quite accuratelly.

In other words, because Rails does not want to be super smart anymore and spy on our `where` conditions to detect which algorithm to use, it expects our help. We must tell it that there is condition for one of the tables. Like that:

```
User.includes(:addresses).where("addresses.country = ?", "Poland").references(:a\
ddresses)
```

I was wondering what would happen if we try to preload more tables but reference only one of them:

```
User.includes(:addresses, :places).where("addresses.country = ?", "Poland").refe\
rences(:addresses)

#  SELECT "users"."id" AS t0_r0, "users"."name" AS t0_r1, "users"."email" AS t0_\
r2, "users"."created_at" AS t0_r3, "users"."updated_at" AS t0_r4,
#         "addresses"."id" AS t1_r0, "addresses"."user_id" AS t1_r1, "addresses"\
."country" AS t1_r2, "addresses"."street" AS t1_r3, "addresses"."postal_code" AS\
 t1_r4, "addresses"."city" AS t1_r5, "addresses"."created_at" AS t1_r6, "address\
es"."updated_at" AS t1_r7,
#         "places"."id" AS t2_r0, "places"."user_id" AS t2_r1, "places"."name" A\
S t2_r2, "places"."created_at" AS t2_r3, "places"."updated_at" AS t2_r4
#  FROM "users"
#  LEFT OUTER JOIN "addresses" ON "addresses"."user_id" = "users"."id"
#  LEFT OUTER JOIN "places" ON "places"."user_id" = "users"."id"
#  WHERE (addresses.country = 'Poland')
```

I imagined that `addresses` would be loaded using the `#eager_load` algorithm (by doing `LEFT JOIN`) and `places` would be loaded using the `#preload` algorithm (by doing separate query to get them) but as you can see that's not the case. Maybe they will change the behavior in the future.

Rails 4 does not warn you to use the `#references` method if you explicitely use `#eager_load` to get the data and the executed query is identical:

```
User.eager_load(:addresses).where("addresses.country = ?", "Poland")
```

In other words, these two are the same:

```
User.includes(:addresses).where("addresses.country = ?", "Poland").references(:a\
ddresses)
User.eager_load(:addresses).where("addresses.country = ?", "Poland")
```

And if you try to use `#preload`, you still get the same exception:

```
User.preload(:addresses).where("addresses.country = ?", "Poland")
#  SELECT "users".* FROM "users" WHERE (addresses.country = 'Poland')
#
#  SQLite3::SQLException: no such column: addresses.country: SELECT "users".* FR\
OM "users"  WHERE (addresses.country = 'Poland')
```

If you try to use the other queries that I showed you, they still work the same way in Rails 4:

```
# Give me users with polish addresses and preload all of their addresses
User.joins(:addresses).where("addresses.country = ?", "Poland").preload(:address\
es)

#Give me all users and their polish addresses.
User.preload(:polish_addresses)
```

Finally in Rails 4 there is at least some documentation for the methods, which Rails 3 has been missing for years:

- `#includes`[12]
- `#preload`[13]
- `#eager_load`[14]

Summary

There are 3 ways to do eager loading in Rails:

- `#includes`
- `#preload`
- `#eager_load`

[12] http://api.rubyonrails.org/v4.0.1/classes/ActiveRecord/QueryMethods.html#method-i-includes
[13] http://api.rubyonrails.org/v4.0.1/classes/ActiveRecord/QueryMethods.html#method-i-preload
[14] http://api.rubyonrails.org/v4.0.1/classes/ActiveRecord/QueryMethods.html#method-i-eager_load

`#includes` delegates the job to `#preload` or `#eager_load` depending on the presence or absence of condition related to one of the preloaded table.

`#preload` is using separate DB queries to get the data.

`#eager_load` is using one big query with `LEFT JOIN` for each eager loaded table.

In Rails 4 you should use `#references` combined with `#includes` if you have the additional condition for one of the eager loaded table.

Creating new content types in Rails 4.2

This is a blogpost written by Marcin Grzywaczewski. It is available online on the Arkency Blog[15].

While working on the application for React.js+Redux workshop[16] I've decided to follow the JSON API[17] specification of responses for my API endpoints. Apart from a fact that following the spec allowed me to avoid bikeshedding, there was also an interesting issue I needed to solve with Rails.

In JSON API specification there is a requirement about the `Content-Type` being set to an appropriate value[18]. It's great, because it allows generic clients to distinguish JSONAPI-compliant endpoints. Not to mention you can serve your old API while hitting the endpoint with an `application/json` Content-Type and have your new API responses crafted in an iterative way for the same endpoints.

While being a very good thing, there was a small problem I've needed to solve. First of all - how to inform Rails that you'll be using the new `Content-Type` and make it possible to use `respond_to` in my controllers? And secondly - how to tell Rails that JSON API requests are very similar to JSON requests, thus request params must be a JSON parsed from the request's body?

I've managed to solve both problems and I'm happy with this solution. In this article I'd like to show you how it can be done with Rails.

Registering the new `Content-Type`

First problem I needed to solve is usage of a new content type with Rails and registering it so Rails would be aware that this new content type exists. This allows you to use this content type while working with `respond_to` or `respond_with` inside your controllers - a thing that is very useful if you happen to serve many responses dependent on the content type.

Fortunately this is very simple and Rails creators somehow expected this use case. If you create your new Rails project there will be an initializer created which is perfect for this goal - `config/initializers/mime_types.rb`.

All I needed to do here was to register a new content type and name it:

[15] http://blog.arkency.com/2016/03/creating-new-content-types-in-rails-4-dot-2/

[16] http://blog.arkency.com/2016/02/how-to-teach-react-dot-js-properly-a-quick-preview-of-wroc-love-dot-rb-workshop-agenda/

[17] http://blog.arkency.com/2016/02/how-and-why-should-you-use-json-api-in-your-rails-api/

[18] http://jsonapi.org/format/#content-negotiation

```
# Be sure to restart your server when you modify this file.

Mime::Type.register "application/vnd.api+json", :jsonapi

# Add new mime types for use in respond_to blocks:
# Mime::Type.register "text/richtext", :rtf
```

This way I managed to use it with my controllers - `jsonapi` is available as a method of `format` given by the `respond_to` block:

```
class EventsController < ApplicationController
  def show
    respond_to do |format|
      format.jsonapi do
        Event.find(params[:id]).tap do |event|
          serializer = EventSerializer.new(self, event.conference_id)
          render json: serializer.serialize(event)
      end

      format.all { head :bad_request }
    end
  end
end
```

That's great! - I thought and I forgot about the issue. Then during preparations I've created a simple JS client for my API to be used by workshop attendants:

```
const { fetch } = window;

function APIClient () {
  const JSONAPIFetch = (method, url, options) => {
    const headersOptions = {
      method,
      headers: {
        'Accept': 'application/vnd.api+json',
        'Content-Type': 'application/vnd.api+json'
      }
    };

    return fetch(url, Object.assign({}, options, headersOptions));
  };

```

```
  return {
    get (url) {
      const request = JSONAPIFetch("GET", url, {});
      return request;
    },
    post (url, params) {
      const request = JSONAPIFetch("POST", url,
                        { body: JSON.stringify(params) });
      return request;
    },
    delete (url) {
      const request = JSONAPIFetch("DELETE", url, {});
      return request;
    }
  };
}

window.APIClient = APIClient();
```

Then I've decided to test it...

Specifying how params should be parsed - ActionDispatch::ParamsParser middleware

Since I wanted to be sure that everything works correctly I gave a try to the `APIClient` I've just created. I opened the browser's console and issued the following call:

```
APIClient.post("/conferences", { conference:
                                 { id: UUID.create().toString(),
                                  name: "My new conference!" } });
```

Bam! I got the HTTP 400 status code. Confused, I've checked the Rails logs:

```
Processing by ConferencesController#create as JSONAPI
Completed 400 Bad Request in 7ms

ActionController::ParameterMissing (param is missing or the value is empty: conf\
erence):
  app/controllers/conferences_controller.rb:66:in `conference_params'
  app/controllers/conferences_controller.rb:16:in `block (2 levels) in create'
  app/controllers/conferences_controller.rb:13:in `create'
```

Oh well. I passed my params correctly, but somehow Rails cannot figure how to handle these parameters. And if you think about it - why it should do it? For Rails this is a *completely* new content type. Rails doesn't know that this is a little more structured JSON request.

Apparently there is a Rack middleware that is responsible for parsing params depending on the content type. It is called `ActionDispatch::ParamsParser` and its `initialize` method accepts a Rack app (which every middleware does, honestly) and an optional argument called `parsers`. In fact the constructor is very simple I can copy it here:

```
# File actionpack/lib/action_dispatch/middleware/params_parser.rb, line 18
def initialize(app, parsers = {})
  @app, @parsers = app, DEFAULT_PARSERS.merge(parsers)
end
```

As you can see there is a list of DEFAULT parsers and by populating this optional argument you can provide your own parsers.

Rails loads this middleware by default without optional parameter set. What you need to do is to unregister the "default" version Rails uses and register it again - this way with your custom code responsible for parsing request parameters. I did it in `config/initializers/mime_types.rb` again:

```
# check app name in config/application.rb
middlewares = YourAppName::Application.config.middleware
middlewares.swap(ActionDispatch::ParamsParser, ActionDispatch::ParamsParser, {
  Mime::Type.lookup('application/vnd.api+json') => lambda do |body|
    ActiveSupport::JSON.decode(body)
  end
})
```

Let's take a look at this code in a step by step manner:

1. First of all, the variable called `middlewares` is created. It is an object of `MiddlewareStack-Proxy`[19] type which represents a chain of your loaded middlewares.

[19] http://api.rubyonrails.org/classes/Rails/Configuration/MiddlewareStackProxy.html

2. `swap` is a function to replace the chosen middleware with another middleware. In this use case we're replacing the default `ActionDispatch::ParamsParser` middleware with the same type of middleware, but we're recreating it with custom arguments. `swap` also takes care of putting the middleware in the same place that the previous middleware sat before - that can avoid us subtle errors that could be possible with wrong order of middlewares.
3. The `parsers` object is keyed with identifiers of a content type which can be accessed using `Mime::Type.lookup` method. A value is a lambda that will be called upon request's body every time the new request arrives - in this case it is just calling method for parsing the body as JSON. The result should be an object representing parameters.

As you can see this is quite powerful. This is a very primitive use case. But this approach is flexible enough to extract parameters from any content type. This can be used to pass `*.Plist` files used by Apple technologies as requests (I saw such use cases) and, in fact, anything. Waiting for someone crazy enough to pass `*.docx` documents and extracting params out of it! :)

Summary

While new content types are often useful, there is a certain work needed to make it work correctly with Rails. Fortunately there is a very simple way to register new document types - and as long as you don't need to parse parameters out of it is easy.

As it turns out there is a nice way of defining your own parsers inside Rails. I was quite surprised that I had this issue (well, Rails is *magic* after all! :)), but thanks to `ActionDispatch::ParamsParser` being written in a way adhering to OCP[20] I managed to do it without monkey patching or other cumbersome solutions.

If you know a better way to achieve the same thing, or a gem that makes it easier - let us know. You can write a comment or catch us on Twitter[21] or write an e-mail[22] to us.

[20] https://en.wikipedia.org/wiki/Open/closed_principle
[21] http://twitter.com/arkency
[22] mailto:dev@arkency.com