
Fast JSON API — :warning: This project is no longer maintained!!!!

:warning:

build passing

A lightning fast JSON:API serializer for Ruby Objects.

Since this project is no longer maintained, please consider using alternatives or the forked project [jsonapi-serializer/jsonapi-serializer](#)!

Performance Comparison

We compare serialization times with Active Model Serializer as part of RSpec performance tests included on this library. We want to ensure that with every change on this library, serialization time is at least 25 `times` faster than Active Model Serializers on up to current benchmark of 1000 records. Please read the performance document for any questions related to methodology.

Benchmark times for 250 records

```
1 $ rspec
2 Active Model Serializer serialized 250 records in 138.71 ms
3 Fast JSON API serialized 250 records in 3.01 ms
```

Table of Contents

- Features
- Installation
- Usage
 - Rails Generator
 - Model Definition
 - Serializer Definition
 - Object Serialization
 - Compound Document
 - Key Transforms
 - Collection Serialization
 - Caching

-
- Params
 - Conditional Attributes
 - Conditional Relationships
 - Sparse Fieldsets
 - Using helper methods
- Contributing

Features

- Declaration syntax similar to Active Model Serializer
- Support for `belongs_to`, `has_many` and `has_one`
- Support for compound documents (included)
- Optimized serialization of compound documents
- Caching

Installation

Add this line to your application's Gemfile:

```
1 gem 'fast_jsonapi'
```

Execute:

```
1 $ bundle install
```

Usage

Rails Generator

You can use the bundled generator if you are using the library inside of a Rails project:

```
1 rails g serializer Movie name year
```

This will create a new serializer in `app/serializers/movie_serializer.rb`

Model Definition

```
1 class Movie
2   attr_accessor :id, :name, :year, :actor_ids, :owner_id, :
      movie_type_id
3 end
```

Serializer Definition

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3   set_type :movie # optional
4   set_id :owner_id # optional
5   attributes :name, :year
6   has_many :actors
7   belongs_to :owner, record_type: :user
8   belongs_to :movie_type
9 end
```

Sample Object

```
1 movie = Movie.new
2 movie.id = 232
3 movie.name = 'test movie'
4 movie.actor_ids = [1, 2, 3]
5 movie.owner_id = 3
6 movie.movie_type_id = 1
7 movie
```

Object Serialization

Return a hash

```
1 hash = MovieSerializer.new(movie).serializable_hash
```

Return Serialized JSON

```
1 json_string = MovieSerializer.new(movie).serialized_json
```

Serialized Output

```
1 {
2   "data": {
3     "id": "3",
4     "type": "movie",
5     "attributes": {
6       "name": "test movie",
7       "year": null
```

```
8     },
9     "relationships": {
10       "actors": {
11         "data": [
12           {
13             "id": "1",
14             "type": "actor"
15           },
16           {
17             "id": "2",
18             "type": "actor"
19           }
20         ]
21       },
22       "owner": {
23         "data": {
24           "id": "3",
25           "type": "user"
26         }
27       }
28     }
29   }
30 }
```

Key Transforms

By default fast_jsonapi underscores the key names. It supports the same key transforms that are supported by AMS. Here is the syntax of specifying a key transform

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3   # Available options :camel, :camel_lower, :dash, :underscore(default)
4   set_key_transform :camel
5 end
```

Here are examples of how these options transform the keys

```
1 set_key_transform :camel # "some_key" => "SomeKey"
2 set_key_transform :camel_lower # "some_key" => "someKey"
3 set_key_transform :dash # "some_key" => "some-key"
4 set_key_transform :underscore # "some_key" => "some_key"
```

Attributes

Attributes are defined in FastJsonapi using the `attributes` method. This method is also aliased as `attribute`, which is useful when defining a single attribute.

By default, attributes are read directly from the model property of the same name. In this example, `name` is expected to be a property of the object being serialized:

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attribute :name
5 end
```

Custom attributes that must be serialized but do not exist on the model can be declared using Ruby block syntax:

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attributes :name, :year
5
6   attribute :name_with_year do |object|
7     "#{object.name} (#{object.year})"
8   end
9 end
```

The block syntax can also be used to override the property on the object:

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attribute :name do |object|
5     "#{object.name} Part 2"
6   end
7 end
```

Attributes can also use a different name by passing the original method or accessor with a proc shortcut:

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attributes :name
5
6   attribute :released_in_year, &:year
7 end
```

Links Per Object

Links are defined in FastJsonapi using the `link` method. By default, links are read directly from the model property of the same name. In this example, `public_url` is expected to be a property of the

object being serialized.

You can configure the method to use on the object for example a link with key `self` will get set to the value returned by a method called `url` on the movie object.

You can also use a block to define a url as shown in `custom_url`. You can access params in these blocks as well as shown in `personalized_url`

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   link :public_url
5
6   link :self, :url
7
8   link :custom_url do |object|
9     "http://movies.com/#{object.name}-(#{object.year})"
10  end
11
12  link :personalized_url do |object, params|
13    "http://movies.com/#{object.name}-#{params[:user].reference_code}"
14  end
15 end
```

Links on a Relationship You can specify relationship links by using the `links:` option on the serializer. Relationship links in JSON API are useful if you want to load a parent document and then load associated documents later due to size constraints (see related resource links)

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   has_many :actors, links: {
5     self: :url,
6     related: -> (object) {
7       "https://movies.com/#{object.id}/actors"
8     }
9   }
10 end
```

This will create a `self` reference for the relationship, and a `related` link for loading the actors relationship later. NB: This will not automatically disable loading the data in the relationship, you'll need to do that using the `lazy_load_data` option:

```
1   has_many :actors, lazy_load_data: true, links: {
2     self: :url,
3     related: -> (object) {
4       "https://movies.com/#{object.id}/actors"
5     }
6   }
```

```
6 }
```

Meta Per Resource

For every resource in the collection, you can include a meta object containing non-standard meta-information about a resource that can not be represented as an attribute or relationship.

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   meta do |movie|
5     {
6       years_since_release: Date.current.year - movie.year
7     }
8   end
9 end
```

Compound Document

Support for top-level and nested included associations through `options[:include]`.

```
1 options = {}
2 options[:meta] = { total: 2 }
3 options[:links] = {
4   self: '...',
5   next: '...',
6   prev: '...'
7 }
8 options[:include] = [:actors, :'actors.agency', :'actors.agency.state']
9 MovieSerializer.new([movie, movie], options).serialized_json
```

Collection Serialization

```
1 options[:meta] = { total: 2 }
2 options[:links] = {
3   self: '...',
4   next: '...',
5   prev: '...'
6 }
7 hash = MovieSerializer.new([movie, movie], options).serializable_hash
8 json_string = MovieSerializer.new([movie, movie], options).
  serialized_json
```

Control Over Collection Serialization You can use `is_collection` option to have better control over collection serialization.

If this option is not provided or `nil` autedetect logic is used to try understand if provided resource is a single object or collection.

Autodetect logic is compatible with most DB toolkits (ActiveRecord, Sequel, etc.) but **cannot** guarantee that single vs collection will be always detected properly.

```
1 options[:is_collection]
```

was introduced to be able to have precise control this behavior

- `nil` or not provided: will try to autodetect single vs collection (please, see notes above)
- `true` will always treat input resource as *collection*
- `false` will always treat input resource as *single object*

Caching

Requires a `cache_key` method be defined on model:

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3   set_type :movie # optional
4   cache_options enabled: true, cache_length: 12.hours
5   attributes :name, :year
6 end
```

Params

In some cases, attribute values might require more information than what is available on the record, for example, access privileges or other information related to a current authenticated user. The `options[:params]` value covers these cases by allowing you to pass in a hash of additional parameters necessary for your use case.

Leveraging the new params is easy, when you define a custom attribute or relationship with a block you opt-in to using params by adding it as a block parameter.

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attributes :name, :year
5   attribute :can_view_early do |movie, params|
6     # in here, params is a hash containing the `:current_user` key
```

```

7     params[:current_user].is_employee? ? true : false
8   end
9
10  belongs_to :primary_agent do |movie, params|
11    # in here, params is a hash containing the `:current_user` key
12    params[:current_user].is_employee? ? true : false
13  end
14 end
15
16 # ...
17 current_user = User.find(cookies[:current_user_id])
18 serializer = MovieSerializer.new(movie, {params: {current_user:
19   current_user}})
19 serializer.serializable_hash

```

Custom attributes and relationships that only receive the resource are still possible by defining the block to only receive one argument.

Conditional Attributes

Conditional attributes can be defined by passing a Proc to the **if** key on the `attribute` method. Return **true** if the attribute should be serialized, and **false** if not. The record and any params passed to the serializer are available inside the Proc as the first and second parameters, respectively.

```

1  class MovieSerializer
2    include FastJsonapi::ObjectSerializer
3
4    attributes :name, :year
5    attribute :release_year, if: Proc.new { |record|
6      # Release year will only be serialized if it's greater than 1990
7      record.release_year > 1990
8    }
9
10   attribute :director, if: Proc.new { |record, params|
11     # The director will be serialized only if the :admin key of params
12     # is true
13     params && params[:admin] == true
14   }
15 end
16 # ...
17 current_user = User.find(cookies[:current_user_id])
18 serializer = MovieSerializer.new(movie, { params: { admin: current_user
19   .admin? }})
19 serializer.serializable_hash

```

Conditional Relationships

Conditional relationships can be defined by passing a Proc to the **if** key. Return **true** if the relationship should be serialized, and **false** if not. The record and any params passed to the serializer are available inside the Proc as the first and second parameters, respectively.

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   # Actors will only be serialized if the record has any associated
    actors
5   has_many :actors, if: Proc.new { |record| record.actors.any? }
6
7   # Owner will only be serialized if the :admin key of params is true
8   belongs_to :owner, if: Proc.new { |record, params| params && params[:
    admin] == true }
9 end
10
11 # ...
12 current_user = User.find(cookies[:current_user_id])
13 serializer = MovieSerializer.new(movie, { params: { admin: current_user
    .admin? } })
14 serializer.serializable_hash
```

Sparse Fieldsets

Attributes and relationships can be selectively returned per record type by using the **fields** option.

```
1 class MovieSerializer
2   include FastJsonapi::ObjectSerializer
3
4   attributes :name, :year
5 end
6
7 serializer = MovieSerializer.new(movie, { fields: { movie: [:name] } })
8 serializer.serializable_hash
```

Using helper methods

You can mix-in code from another ruby module into your serializer class to reuse functions across your app.

Since a serializer is evaluated in the context of a **class** rather than an **instance** of a class, you need to make sure that your methods act as **class** methods when mixed in.

Using ActiveSupport::Concern

```
1
2 module AvatarHelper
3   extend ActiveSupport::Concern
4
5   class_methods do
6     def avatar_url(user)
7       user.image.url
8     end
9   end
10 end
11
12 class UserSerializer
13   include FastJsonapi::ObjectSerializer
14
15   include AvatarHelper # mixes in your helper method as class method
16
17   set_type :user
18
19   attributes :name, :email
20
21   attribute :avatar do |user|
22     avatar_url(user)
23   end
24 end
```

Using Plain Old Ruby

```
1 module AvatarHelper
2   def avatar_url(user)
3     user.image.url
4   end
5 end
6
7 class UserSerializer
8   include FastJsonapi::ObjectSerializer
9
10  extend AvatarHelper # mixes in your helper method as class method
11
12  set_type :user
13
14  attributes :name, :email
15
16  attribute :avatar do |user|
17    avatar_url(user)
18  end
19 end
```

Customizable Options

Option	Purpose	Example
set_type	Type name of Object	<code>set_type :movie</code>
key	Key of Object	<code>belongs_to :owner, key : :user</code>
set_id	ID of Object	<code>set_id :owner_id</code> or <code>set_id { record "#{record.name.downcase}-#{record.id}" }</code>
cache_options	Hash to enable caching and set cache length	<code>cache_options enabled: true, cache_length: 12.hours, race_condition_ttl: 10.seconds</code>
id_method_name	Set custom method name to get ID of an object (If block is provided for the relationship, <code>id_method_name</code> is invoked on the return value of the block instead of the resource object)	<code>has_many :locations, id_method_name: : place_ids</code>
object_method_name	Set custom method name to get related objects	<code>has_many :locations, object_method_name: : places</code>
record_type	Set custom Object Type for a relationship	<code>belongs_to :owner, record_type: :user</code>
serializer	Set custom Serializer for a relationship	<code>has_many :actors, serializer: : custom_actor</code> or <code>has_many :actors, serializer: MyApp::Api::V1::ActorSerializer</code>
polymorphic	Allows different record types for a polymorphic association	<code>has_many :targets, polymorphic: true</code>

Option	Purpose	Example
polymorphic	Sets custom record types for each object class in a polymorphic association	<code>has_many :targets, polymorphic: { Person => :person, Group => :group }</code>

Instrumentation

`fast_jsonapi` also has builtin Skylight integration. To enable, add the following to an initializer:

```
1 require 'fast_jsonapi/instrumentation/skylight'
```

Skylight relies on `ActiveSupport::Notifications` to track these two core methods. If you would like to use these notifications without using Skylight, simply require the instrumentation integration:

```
1 require 'fast_jsonapi/instrumentation'
```

The two instrumented notifications are supplied by these two constants: `* FastJsonapi::ObjectSerializer::SERIALIZABLE_HASH_NOTIFICATION` `* FastJsonapi::ObjectSerializer::SERIALIZED_JSON_NOTIFICATION`

It is also possible to instrument one method without the other by using one of the following require statements:

```
1 require 'fast_jsonapi/instrumentation/serializable_hash'
2 require 'fast_jsonapi/instrumentation/serialized_json'
```

Same goes for the Skylight integration:

```
1 require 'fast_jsonapi/instrumentation/skylight/normalizers/serializable_hash'
2 require 'fast_jsonapi/instrumentation/skylight/normalizers/serialized_json'
```

Contributing

Please see contribution check for more details on contributing

Running Tests

We use RSpec for testing. We have unit tests, functional tests and performance tests. To run tests use the following command:

```
1 rspec
```

To run tests without the performance tests (for quicker test runs):

```
1 rspec spec --tag ~performance:true
```

To run tests only performance tests:

```
1 rspec spec --tag performance:true
```