DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Using Rails for API-only Applications

In this guide you will learn:

- What Rails provides for API-only applications
- How to configure Rails to start without any browser features
- How to decide which middleware you will want to include
- How to decide which modules to use in your controller

What is an API Application?

Traditionally, when people said that they used Rails as an "API", they meant providing a programmatically accessible API alongside their web application. For example, GitHub provides an API that you can use from your own custom clients.

With the advent of client-side frameworks, more developers are using Rails to build a back-end that is shared between their web application and other native applications.

For example, Twitter uses its public API in its web application, which is built as a static site that consumes JSON resources.

Instead of using Rails to generate HTML that communicates with the server through forms and links, many developers are treating their web application as just an API client delivered as HTML with JavaScript that consumes a JSON API.

This guide covers building a Rails application that serves JSON resources to an API client, including client-side frameworks.

Why Use Rails for JSON APIs?

The first question a lot of people have when thinking about building a JSON API using Rails is: "isn't using Rails to spit out some JSON overkill? Shouldn't I just use something like Sinatra?".

For very simple APIs, this may be true. However, even in very HTML-heavy applications, most of an application's logic lives outside of the view layer.

The reason most people use Rails is that it provides a set of defaults that allows developers to get up and running quickly, without having to make a lot of trivial decisions.

Let's take a look at some of the things that Rails provides out of the box that are still applicable to API applications.

Handled at the middleware layer:

- Reloading: Rails applications support transparent reloading. This works even if your application gets big and restarting the server for every request becomes non-viable.
- Development Mode: Rails applications come with smart defaults for development, making development pleasant without compromising production-time performance.
- Test Mode: Ditto development mode.
- Logging: Rails applications log every request, with a level of verbosity appropriate for the current mode. Rails logs in development include information about the request environment, database queries, and basic performance information.
- Security: Rails detects and thwarts IP spoofing attacks and handles cryptographic signatures in a timing attack aware way. Don't know what an IP spoofing attack or a timing attack is? Exactly.
- Parameter Parsing: Want to specify your parameters as JSON instead of as a URL-encoded String? No problem. Rails will decode the JSON for you and make it available in `params`. Want to use nested URL-encoded parameters? That works too.
- Conditional GETs: Rails handles conditional `GET` (`ETag` and `Last-Modified`) processing request headers and returning the correct response headers and status code. All you need to do is use the `stale?` check in your controller, and Rails will handle all of the HTTP details for you.
- HEAD requests: Rails will transparently convert `HEAD` requests into `GET` ones, and return just the headers on the way out. This makes `HEAD` work reliably in all Rails APIs.

While you could obviously build these up in terms of existing Rack middleware, this list demonstrates that the default Rails middleware stack provides a lot of value, even if you're "just generating JSON".

Handled at the Action Pack layer:

- Resourceful Routing: If you're building a RESTful JSON API, you want to be using the Rails router. Clean and conventional mapping from HTTP to controllers means not having to spend time thinking about how to model your API in terms of HTTP.
- URL Generation: The flip side of routing is URL generation. A good API based on HTTP includes URLs (see the GitHub Gist API for an example).
- Header and Redirection Responses: `head :no_content` and `redirect_to user_url(current_user)` come in handy. Sure, you could manually add the response headers, but why?
- Caching: Rails provides page, action, and fragment caching. Fragment

caching is especially helpful when building up a nested JSON object.
- Basic, Digest, and Token Authentication: Rails comes with out-of-the-box support for three kinds of HTTP authentication.
- Instrumentation: Rails has an instrumentation API that triggers registered handlers for a variety of events, such as action processing, sending a file or data, redirection, and database queries. The payload of each event comes with relevant information (for the action processing event, the payload includes the controller, action, parameters, request format, request method, and the request's full path).
- Generators: It is often handy to generate a resource and get your model, controller, test stubs, and routes created for you in a single command for further tweaking. Same for migrations and others.
- Plugins: Many third-party libraries come with support for Rails that reduce or eliminate the cost of setting up and gluing together the library and the web framework. This includes things like overriding default generators, adding Rake tasks, and honoring Rails choices (like the logger and cache back-end).

Of course, the Rails boot process also glues together all registered components. For example, the Rails boot process is what uses your `config/database.yml` file when configuring Active Record.

The short version is: you may not have thought about which parts of Rails are still applicable even if you remove the view layer, but the answer turns out to be most of it.

The Basic Configuration

If you're building a Rails application that will be an API server first and foremost, you can start with a more limited subset of Rails and add in features as needed.

Creating a new application

You can generate a new api Rails app:

```
$ rails new my_api --api
```

This will do three main things for you:

- Configure your application to start with a more limited set of middleware than normal. Specifically, it will not include any middleware primarily useful for browser applications (like cookies support) by default.
- Make `ApplicationController` inherit from `ActionController::API` instead of `ActionController::Base`. As with middleware, this will leave out any Action Controller modules that provide functionalities primarily used by browser applications.
- Configure the generators to skip generating views, helpers, and assets when you generate a new resource.

Changing an existing application

If you want to take an existing application and make it an API one, read the following steps.

In `config/application.rb`, add the following line at the top of the `Application` class definition:

```
config.api_only = true
```

In `config/environments/development.rb`, set `config.debug_exception_response_format` to configure the format used in responses when errors occur in development mode.

To render an HTML page with debugging information, use the value `:default`.

```
config.debug_exception_response_format = :default
```

To render debugging information preserving the response format, use the value `:api`.

```
config.debug_exception_response_format = :api
```

By default, `config.debug_exception_response_format` is set to `:api`, when `config.api_only` is set to true.

Finally, inside `app/controllers/application_controller.rb`, instead of:

```
class ApplicationController < ActionController::Base
end
```

do:

```
class ApplicationController < ActionController::API
end
```

Choosing Middleware

An API application comes with the following middleware by default:

- `ActionDispatch::HostAuthorization`
- `Rack::Sendfile`
- `ActionDispatch::Static`
- `ActionDispatch::Executor`
- `ActionDispatch::ServerTiming`
- `ActiveSupport::Cache::Strategy::LocalCache::Middleware`
- `Rack::Runtime`
- `ActionDispatch::RequestId`
- `ActionDispatch::RemoteIp`
- `Rails::Rack::Logger`
- `ActionDispatch::ShowExceptions`
- `ActionDispatch::DebugExceptions`
- `ActionDispatch::ActionableExceptions`

- `ActionDispatch::Reloader`
- `ActionDispatch::Callbacks`
- `ActiveRecord::Migration::CheckPending`
- `Rack::Head`
- `Rack::ConditionalGet`
- `Rack::ETag`

See the internal middleware section of the Rack guide for further information on them.

Other plugins, including Active Record, may add additional middleware. In general, these middleware are agnostic to the type of application you are building, and make sense in an API-only Rails application.

You can get a list of all middleware in your application via:

```
$ bin/rails middleware
```

Using the Cache Middleware

By default, Rails will add a middleware that provides a cache store based on the configuration of your application (memcache by default). This means that the built-in HTTP cache will rely on it.

For instance, using the `stale?` method:

```
def show
  @post = Post.find(params[:id])

  if stale?(last_modified: @post.updated_at)
    render json: @post
  end
end
```

The call to `stale?` will compare the `If-Modified-Since` header in the request with `@post.updated_at`. If the header is newer than the last modified, this action will return a "304 Not Modified" response. Otherwise, it will render the response and include a `Last-Modified` header in it.

Normally, this mechanism is used on a per-client basis. The cache middleware allows us to share this caching mechanism across clients. We can enable cross-client caching in the call to `stale?`:

```
def show
  @post = Post.find(params[:id])

  if stale?(last_modified: @post.updated_at, public: true)
    render json: @post
  end
end
```

This means that the cache middleware will store off the `Last-Modified` value for a URL in the Rails cache, and add an `If-Modified-Since` header to any subsequent inbound requests for the same URL.

Think of it as page caching using HTTP semantics.

Using Rack::Sendfile

When you use the `send_file` method inside a Rails controller, it sets the `X-Sendfile` header. `Rack::Sendfile` is responsible for actually sending the file.

If your front-end server supports accelerated file sending, `Rack::Sendfile` will offload the actual file sending work to the front-end server.

You can configure the name of the header that your front-end server uses for this purpose using `config.action_dispatch.x_sendfile_header` in the appropriate environment's configuration file.

You can learn more about how to use `Rack::Sendfile` with popular front-ends in the Rack::Sendfile documentation.

Here are some values for this header for some popular servers, once these servers are configured to support accelerated file sending:

```
# Apache and lighttpd
config.action_dispatch.x_sendfile_header = "X-Sendfile"

# Nginx
config.action_dispatch.x_sendfile_header = "X-Accel-Redirect"
```

Make sure to configure your server to support these options following the instructions in the `Rack::Sendfile` documentation.

Using ActionDispatch::Request

`ActionDispatch::Request#params` will take parameters from the client in the JSON format and make them available in your controller inside `params`.

To use this, your client will need to make a request with JSON-encoded parameters and specify the `Content-Type` as `application/json`.

Here's an example in jQuery:

```
jQuery.ajax({
  type: 'POST',
  url: '/people',
  dataType: 'json',
  contentType: 'application/json',
  data: JSON.stringify({ person: { firstName: "Yehuda", lastName: "Katz" } }),
  success: function(json) { }
});
```

`ActionDispatch::Request` will see the `Content-Type` and your parameters will be:

```
{ :person => { :firstName => "Yehuda", :lastName => "Katz" } }
```

Using Session Middlewares

The following middlewares, used for session management, are excluded from API apps since they normally don't need sessions. If one of your API clients is a browser, you might want to add one of these back in:

- `ActionDispatch::Session::CacheStore`
- `ActionDispatch::Session::CookieStore`
- `ActionDispatch::Session::MemCacheStore`

The trick to adding these back in is that, by default, they are passed `session_options` when added (including the session key), so you can't just add a `session_store.rb` initializer, add `use ActionDispatch::Session::CookieStore` and have sessions functioning as usual. (To be clear: sessions may work, but your session options will be ignored - i.e. the session key will default to `_session_id`)

Instead of the initializer, you'll have to set the relevant options somewhere before your middleware is built (like `config/application.rb`) and pass them to your preferred middleware, like this:

```
# This also configures session_options for use below
config.session_store :cookie_store, key: '_interslice_session'

# Required for all session management (regardless of session_store)
config.middleware.use ActionDispatch::Cookies

config.middleware.use config.session_store, config.session_options
```

Other Middleware

Rails ships with a number of other middleware that you might want to use in an API application, especially if one of your API clients is the browser:

- `Rack::MethodOverride`
- `ActionDispatch::Cookies`
- `ActionDispatch::Flash`

Any of these middleware can be added via:

```
config.middleware.use Rack::MethodOverride
```

Removing Middleware

If you don't want to use a middleware that is included by default in the API-only middleware set, you can remove it with:

```
config.middleware.delete ::Rack::Sendfile
```

Keep in mind that removing these middlewares will remove support for certain features in Action Controller.

Choosing Controller Modules

An API application (using `ActionController::API`) comes with the following controller modules by default:

- `ActionController::UrlFor`: Makes `url_for` and similar helpers available.
- `ActionController::Redirecting`: Support for `redirect_to`.
- `AbstractController::Rendering` and `ActionController::ApiRendering`: Basic support for rendering.
- `ActionController::Renderers::All`: Support for `render :json` and friends.
- `ActionController::ConditionalGet`: Support for `stale?`.
- `ActionController::BasicImplicitRender`: Makes sure to return an empty response, if there isn't an explicit one.
- `ActionController::StrongParameters`: Support for parameters filtering in combination with Active Model mass assignment.
- `ActionController::DataStreaming`: Support for `send_file` and `send_data`.
- `AbstractController::Callbacks`: Support for `before_action` and similar helpers.
- `ActionController::Rescue`: Support for `rescue_from`.
- `ActionController::Instrumentation`: Support for the instrumentation hooks defined by Action Controller (see the instrumentation guide for more information regarding this).
- `ActionController::ParamsWrapper`: Wraps the parameters hash into a nested hash, so that you don't have to specify root elements sending POST requests for instance.
- `ActionController::Head`: Support for returning a response with no content, only headers.

Other plugins may add additional modules. You can get a list of all modules included into `ActionController::API` in the rails console:

```
irb> ActionController::API.ancestors - ActionController::Metal.ancestors
=> [ActionController::API,
    ActiveRecord::Railties::ControllerRuntime,
    ActionDispatch::Routing::RouteSet::MountedHelpers,
    ActionController::ParamsWrapper,
    ... ,
    AbstractController::Rendering,
    ActionView::ViewPaths]
```

Adding Other Modules

All Action Controller modules know about their dependent modules, so you can feel free to include any modules into your controllers, and all dependencies will be included and set up as well.

Some common modules you might want to add:

- `AbstractController::Translation`: Support for the `l` and `t` localization and translation methods.
- Support for basic, digest, or token HTTP authentication:
  - `ActionController::HttpAuthentication::Basic::ControllerMethods`
  - `ActionController::HttpAuthentication::Digest::ControllerMethods`
  - `ActionController::HttpAuthentication::Token::ControllerMethods`
- `ActionView::Layouts`: Support for layouts when rendering.
- `ActionController::MimeResponds`: Support for `respond_to`.
- `ActionController::Cookies`: Support for `cookies`, which includes support for signed and encrypted cookies. This requires the cookies middleware.
- `ActionController::Caching`: Support view caching for the API controller. Please note that you will need to manually specify the cache store inside the controller like this:

  ```
  class ApplicationController < ActionController::API
    include ::ActionController::Caching
    self.cache_store = :mem_cache_store
  end
  ```

  Rails does *not* pass this configuration automatically.

The best place to add a module is in your `ApplicationController`, but you can also add modules to individual controllers.