Rails API Authentication: A Guide to Devise and Devise-JWT Integration

Al Amin Khan Shakil Follow 11 min read · Mar 13, 2024

Cover image of API Authentication

In the world of web development, robust authentication mechanisms are crucial for securing access to sensitive data and resources. Rails, with its powerful framework, provides developers with a myriad of options for implementing authentication in their applications. Among these, Devise stands out as a popular choice for authentication in traditional Rails applications. However, as more developers turn to building APIs as part of their Rails projects, the need for seamless and secure API authentication becomes paramount. Enter Devise-JWT, an extension of Devise tailored specifically for token-based authentication in API-driven applications. In this guide, we'll delve into the world of Rails API authentication using Devise and Devise-JWT and explore their integration. Whether you're a seasoned Rails developer or just dipping your toes into the world of API authentication, this article aims to provide you with a comprehensive understanding of how to authenticate users securely in your Rails API applications. So, let's dive in!

. . .

In this article, we will explore the process of integrating API authentication into an application designed for users to create and reserve services. Through this journey, we'll focus on authenticating users via API endpoints, covering essential functionalities such as user registration, login, and logout. In our next article, we will discuss ***how to manage sessions and persist data on the frontend***.

I will create Rails-API-only app for our project and authenticate user using Devise and JWT, and serialization using jsonapi-serializer.

I am using the following versions:

- **Ruby:** 3.1.2
- **Rails:** 7.1.3.2
- **PostgreSQL:** 16.1

. . .

Now, let's delve into our step-by-step guide:

What is Authentication and why jwt?

HTTP operates as a stateless protocol, meaning each call made to the server is treated as independent, without any memory or 'state' of previous requests. However, in practical web applications, maintaining session information is essential. For instance, we often store login data to grant users access to specific parts and features of the application without requiring authentication with every request.

When it comes to managing authentication, there are two primary approaches:

- Cookies based approach
- JWT (JSON Web Token) approach

For our application, we've opted for the JWT approach:

Upon successful authentication (verifying username and password), the server generates an accessToken by encrypting essential information like the "userId" and "expiresIn." This token is then sent to the client (typically the browser), where it's stored. Subsequently, the client includes this token with every request.

With JWT, no session information is stored in the database. Instead, all relevant user data, or the "bearer," is encapsulated within the token itself. This means that with each request, the server can authenticate the user by decrypting the token, eliminating the need for database searches for session information. However, it's essential to note that only the server with the access token secret can decrypt the JWT token.

One consideration with JWT tokens is that they must have an expiration time. While this enhances security, it may impact user experience, especially in applications where users rarely sign out.

To implement authentication and session management in our Rails app, we'll leverage the **Devise** and **Devise-JWT** gems.

. . .

Create the Rails API app

```
rails new fluffy-transit --api -d postgresql
```

— api : Preconfigure a smaller stack for API-only apps

-d : Preconfigure for the selected database

. . .

Enabling CORS

CORS (cross-origin resource sharing) is an HTTP-header-based security mechanism that defines who's allowed to interact with your API. CORS is built into all modern web browsers.

In the most simple scenario, CORS will block all requests from a different origin than your API. "Origin" in this case is the combination of protocol, domain, and port.

Since our backend and front end will be hosted on different servers, we have to manually set up the application to accept requests from a different origin.

Luckily, there is a gem that will set this up for us with minimal configuration, just uncomment the following line from the Gemfile:

```
gem 'rack-cors'
```

run `bundle install` and uncomment the contents of the file `config/initializers/cors.rb` :

```
Rails.application.config.middleware.insert_before 0, Rack::Cors do
  allow do
    origins "*"  # later change to the domain of the frontend app

    resource "*",
      headers: :any,
      methods: [:get, :post, :put, :patch, :delete, :options, :head],
      expose: [:Authorization]
  end
end
```

You need to pay close attention to the **origins** parameter. Remember that CORS needs to match protocol, domain, and port. You'll need to specify all

three correctly. So if you put `http://example.com:80`, but the call will come from `http://example.com:8080` or `https://example.com:443`, then you'll still get blocked. Leaving it as `'*'` allows anyone to connect to your API. When first creating the app and for testing purposes, leaving it as `'*'` is a good idea, but remember to change it when you are ready to deploy.

By default, the rack-cors gem only exposes certain headers like `Content-Type`, `Last-Modified,` etc., so we have to manually expose the 'Authorization' header to cross-origin requests, which will be used to dispatch and receive JWT tokens.

. . .

devise, devise-JWT and jsonapi-serializer

- devise is a gem that handles users and authentication. It takes care of all the controllers necessary for user creation (users_controller) and user sessions (users_sessions_controller). I initially wondered whether devise, which is designed to use in full-stack Rails apps, is a good choice for API-only mode, but with a few small tweaks, it works and does the job.
- devise-jwt, a gem, is an extension to devise which handles the use of JWT tokens for user authentication.
- jsonapi-serializer is a gem that will serialize ruby objects in JSON format.

in gemfile add:

```
gem 'devise'
gem 'devise-jwt'
gem 'jsonapi-serializer'
```

then run **bundle install**

Setup Devise:

Run the following to create the installation files:

```
rails g devise:install
```

We need to tell devise we will not use navigational formats by making sure the array is empty in `config/initializers/devise.rb.` uncomment and edit the following line:

```
config.navigational_formats = []
```

This will prevent devise from using flash messages which are a default feature and are not present in Rails `api` mode.

We also need to add the following line at the end of `config/environments/development.rb`

```
config.action_mailer.default_url_options = { host: 'localhost', port: 3001 }
```

We will use port 3001 in our Rails app development and leave port 3000 for the React front-end app. To enable this, we also need to update the port PUMA (the default Rails server) will use in `config/puma.rb`

```
port ENV.fetch('PORT') { 3001 }
```

Next, we create the **devise model** for the user. It can be named anything. we will use User

```
rails g devise User
```

then run the first migration by executing:

```
rails db:create db:migrate
```

Next, we generate the **devise controllers** (sessions and registrations) to handle sign-ins and sign-ups

```
rails g devise:controllers users -c sessions registrations
```

we use the -c flag to specify a controller.

For devise to know it can respond to `JSON` format (and not try to render a view), we need to instruct the controllers:

```
class Users::SessionsController < Devise::SessionsController
  respond_to :json
end
```

All the commented code can be removed since we will be writing our own methods to handle the steps.

```
class Users::RegistrationsController < Devise::RegistrationsController
  respond_to :json
end
```

Then we must override the default routes provided by devise and add route aliases.

```
Rails.application.routes.draw do
  devise_for :users, path: '', path_names: {
    sign_in: 'login',
    sign_out: 'logout',
    registration: 'signup'
  },
  controllers: {
    sessions: 'users/sessions',
    registrations: 'users/registrations'
  }
end
```

Now, the endpoints will be set to login, logout, and signup.

Devise works by allowing certain default parameters during user registration, for example, email, password, password confirmation, remember me, etc. In the application we are building we also want to let the user add a name and a role. To achieve this, we need to "sanitize" (or permit) these new parameters. We should add the following lines to the Application Controller:

```
class ApplicationController < ActionController::API
  before_action :configure_permitted_parameters, if: :devise_controller?

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.permit(:sign_up, keys: %i[name])

    devise_parameter_sanitizer.permit(:account_update, keys: %i[name])
  end
end
```

Here we permit the use of name for signup and updates of the user.

We now need to create a migration to add the name to the Users table:

```
rails g migration AddNameToUsers name:string
```

run `rails db:migrate`

Set up devise-jwt

devise-jwt will handle token dispatch and authentication, which doesn't come with devise out of the box.

JWTs need to be created with a secret key that is private. It shouldn't be revealed to the public. When we receive a JWT from the client, we can verify it with that secret key stored on the server.

We can generate a secret by typing the following in the terminal:

```
bundle exec rails secret
```

We will then add it to the encrypted credentials file so it won't be exposed:

```
#VSCode
EDITOR='code --wait' rails credentials:edit
```

Then we add a new key: value in the encrypted .yml file.

```
# Other secrets...  # Used as the base secret for Devise-JWT
devise_jwt_secret_key: (copy and paste the generated secret here)
```

Inside the devise initializer, we will specify that on every login POST request it should append the JWT token to the 'Authorization' header as "Bearer + token" when there's a successful response sent back, and on a logout DELETE request, the token should be revoked.

in `config/initializers/devise.rb` at the end add the following:

```
  config.jwt do |jwt|
    jwt.secret = Rails.application.credentials.devise_jwt_secret_key!
    jwt.dispatch_requests = [
      ['POST', %r{^/login$}]
    ]
    jwt.revocation_requests = [
      ['DELETE', %r{^/logout$}]
    ]
    jwt.expiration_time = 1.day.to_i
  end
```

In the JWT requests, we should delimit the regex with `^` and `$` to avoid unintentional matches.

We are also setting the expiration time of the token to 1 day (you can use any time limitation you want) after which the user will need to authenticate again.

Revocation strategy:

As previously explained, tokens have an expiration date, they will become useless after that, but, because of their nature, they cannot realistically be revoked before that.

When would we want to revoke a token? When the user logs out.

Why would we want to revoke tokens? The bearer of the token has access granted to it, no further authentication is needed. So if a token is still active, and it somehow falls into the wrong hands, someone could access protected content and tamper with things.

The devise-jwt gem offers 3 different revocation strategies. We will use the JTIMatcher method.

from devise-jwt documentation:

> Here, the model class acts as the revocation strategy. It needs a new string column named `JTI` to be added to the user. `JTI` stands for JWT ID, and it is a standard claim meant to uniquely identify a token.
>
> It works like the following:
>
> When a token is dispatched for a user, the `JTI` claim is taken from the `JTI` column in the model (which has been initialized when the record has been created).

> At every authenticated action, the incoming token `JTI` claim is matched against the `JTI` column for that user. The authentication only succeeds if they are the same.
>
> When the user requests to sign out its `JTI` column changes, so that provided token won't be valid anymore.

To use it, we need to add the JTI column in the user model. so we must create a new migration:

```
rails g migration addJtiToUsers jti:string:index:unique
```

And then make sure to add `null: false` to the `add_column` line and `unique: true` to the `add_index` line.

the migration file should look like this:

```
class AddJtiToUsers < ActiveRecord::Migration[7.1]
  def change
    add_column :users, :jti, :string, null: false
    add_index :users, :jti, unique: true
  end
end
```

`rails db:migrate`

Then we have to add the strategy to the user model and configure it to use the correct revocation strategy:

```
class User < ApplicationRecord
  include Devise::JWT::RevocationStrategies::JTIMatcher

  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable, :trackable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable,
         :jwt_authenticatable, jwt_revocation_strategy: self

end
```

Working with jsonapi_serializer:

This serializer creates JSON responses following the JSON:API convention.

Since we already installed the gem using `bundle install`, we will create a serializer for the User model. We will call it every time we want to send user data from our API backend.

```
rails g serializer user id email name
```

It generates a serializer with the attributes mentioned that we can later call using the following command:

```
UserSerializer.new(#user).serializable_hash[:data][:attributes]
```

replace `#user` with the actual variable containing the user information.

. . .

After all the setups above, now we need to write the behavior of the app for user registration, login, and logout.

We will use some devise helper methods that tell the app what to do in different situations:

- `respond_with`: how to respond in the case of a POST (after registration or logging in)

- `respond_to_on_destroy`: how to respond to DELETE (after logging out)

in `app/controllers/users/registrations_controller.rb`:

```
class Users::RegistrationsController < Devise::RegistrationsController
  respond_to :json

  private

  def respond_with(resource, _opts = {})
    if resource.persisted?
      @token = request.env['warden-jwt_auth.token']
      headers['Authorization'] = @token

      render json: {
        status: { code: 200, message: 'Signed up successfully.',
                  token: @token,
                  data: UserSerializer.new(resource).serializable_hash[:data][:a
      }
    else
      render json: {
        status: { message: "User couldn't be created successfully. #{resource.er
      }, status: :unprocessable_entity
    end
  end
end
```

in `app/controllers/users/sessions_controller.rb`:

```
class Users::SessionsController < Devise::SessionsController
  respond_to :json

  private

  def respond_with(resource, _opt = {})
    @token = request.env['warden-jwt_auth.token']
    headers['Authorization'] = @token

    render json: {
      status: {
        code: 200, message: 'Logged in successfully.',
        token: @token,
        data: {
          user: UserSerializer.new(resource).serializable_hash[:data][:attribute
        }
      }
    }, status: :ok
  end

  def respond_to_on_destroy
    if request.headers['Authorization'].present?
      jwt_payload = JWT.decode(request.headers['Authorization'].split.last,
                               Rails.application.credentials.devise_jwt_secret_k

      current_user = User.find(jwt_payload['sub'])
    end

    if current_user
      render json: {
        status: 200,
        message: 'Logged out successfully.'
      }, status: :ok
    else
      render json: {
        status: 401,
        message: "Couldn't find an active session."
      }, status: :unauthorized
    end
  end
end
```

Since we created a Rails API-only app, sessions are disabled by default, but devise relies on sessions to function. Before Rails 7, sessions were passed as a hash, so even if they were disabled, devise could still write into a session hash. Since Rails 7, a session is an `ActionDispatch::Session` object, which is not writable on Rails API-only apps, because the ActionDispatch::Session is disabled. If we try to use devise in this configuration we will get the error:

`ActionDispatch::Request::Session::DisabledSessionError (Your application has sessions disabled. To write to the session you must first configure a session store):`

To resolve this issue, we need to enable sessions in our application configuration. This can be achieved by adding the following code to the `config/application.rb` file:

```
class Application < Rails::Application
    # other codes

    # Enabled the session store for api_only application
    config.session_store :cookie_store, key: '_interslice_session'
    config.middleware.use ActionDispatch::Cookies
    config.middleware.use config.session_store, config.session_options
  end
```

And that's it. We can now create user login and log out.

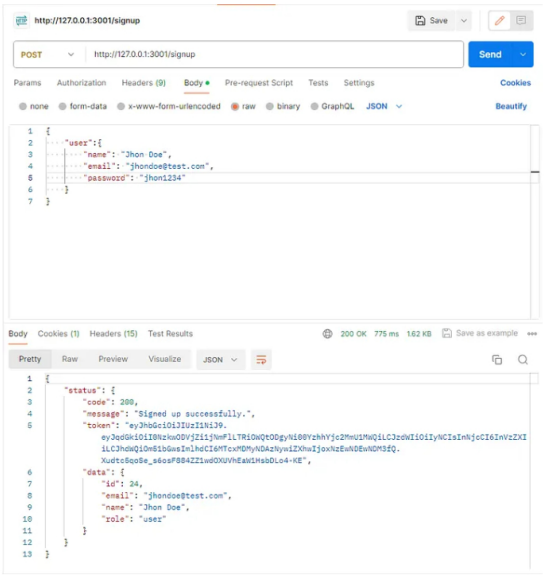
. . .

Testing with Postman:

In our application terminal, run `rails s` to start the server

In our application terminal, run `rails s` to start the server.

Creating a user:

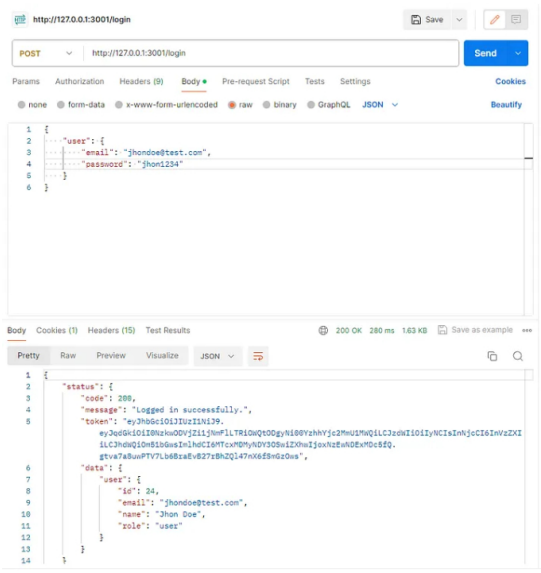


User registration test

It is important to note the format in which the information should be submitted.

Note: you will not get the **role** in response and it's ok. But if you want it you need to permit it in application_controller and add the role to the user table.

Logging in:



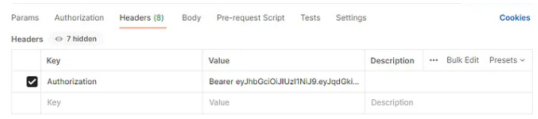
User login test

After we login or register, we can see that the authorization token was received:

Authorization token

Logging out:

When we log out the authorization token needs to be passed, for testing, we have to manually add it to Postman: (write 'Bearer + token')



User log out test

. . .

For now, this is the testing we will make.

In the next article, I will explain ***how to manage sessions and persist data on the frontend***.

Don't forget to follow for updates and notifications on the upcoming article!

Api Authentication | Rails | Devise | Devise Jwt | Rails Api Authentication

26 5

Written by Al Amin Khan Shakil

14 followers · 2 following

Full-stack Web Developer

Follow

Responses (5)

nahian rafi

What are your thoughts?

Mister Getman
Jun 9, 2024

Great work Al Amin. But it should be mentioned somewhere that you've rewritten all from this (https://sdrmike.medium.com/rails-7-api-only-app-with-devise-and-jwt-for-authentication-139721ffb97c) article.

And please, do not delete my comment again))).

4 1 reply Reply

Anyars Yussif
Mar 14, 2024

Great work Al Amin. Very helpful

2 Reply

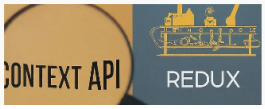
Ayokunnumi Omololu
Mar 13, 2024

This is so insightful. Thanks for sharing such an informative article on this topic.

1 Reply

See all responses

More from Al Amin Khan Shakil



Al Amin Khan Shakil

Understanding Function Declarations and Function...

JavaScript is a versatile and widely-used programming language, and one of its core...

Mar 11 4

Al Amin Khan Shakil

Context API vs Redux for React State Management

Ever built a React application where data needs to be shared across multiple...

Jul 14, 2024 1

Al Amin Khan Shakil

Getting Started with Front-End Web Development: A Beginner's...

Embarking on the journey of web development is akin to setting out on a...

Sep 30, 2023 2

See all from Al Amin Khan Shakil

Recommended from Medium

Ravi Prakash

🚀 Mastering Rails Inbuilt Serializers—A Beginner-Friendly...

If you're building a Rails API or returning structured JSON from your controllers, you...

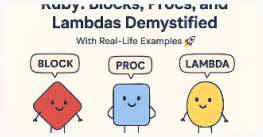
4d ago 4

Vaishnavi Ganeshkar

How I Structure a Production-Ready Ruby on Rails Application i...

What I've learned after building real Rails apps—and the patterns I'd recommend today

5d ago 55



Ben Forrest

Beyond 'strong' IDs: why ULID and UUID7 deserve a place in your Rai...

A quest for the ideal public identifier

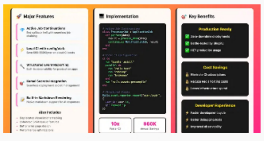
Sep 30 6

In Stackademic by Adarsh Mishra

Ruby: Blocks, Procs, and Lambdas Demystified (With Real-Life...

If you've been coding in Ruby for a while, you've probably seen blocks, procs, and...

Sep 8 3

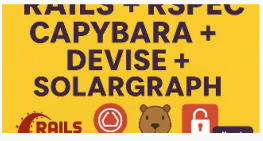


In Write A Catalyst by Raza Hussain

🚀 Rails 8.1 Just Dropped—and It's Packed with Developer-First...

Rails 8.1 isn't just another incremental update. It's a milestone for app-builders who care...

Oct 23 61 1



In Jungletronics by J3

Rails + RSpec + Capybara + Devise

+ Solargraph Setup (working alias br)—#Episode 1

Nov 25

See more recommendations