Implementing an Authentication in a Ruby on Rails API-only Project using Devise and JWT

Hussien Elgammal · Dec 13, 2023 · 16 min read

Table of contents

In this comprehensive guide, we explore implementing authentication mechanisms within a Ruby on Rails API-only project. Harnessing the power of two formidable tools Devise and JSON Web Tokens (JWT), we delve into creating a robust authentication system that combines the versatility of Devise's session-based authentication with the stateless and secure nature of JWT-based token authentication and discuss some interesting stuff ^-^

I am gonna use the following versions:

- Ruby: 3.2.2
- Rails: 7.1.2
- PostgreSQL: 15.4

Why you should use JWT?

JSON Web Tokens (JWT) represent a modern and widely used method for securely transmitting information between parties as a JSON object. JWTs are commonly employed for authentication and authorization in web applications. They consist of three parts separated by dots: a header, a payload, and a signature. These tokens are digitally signed using a secret or a public/private key pair, enabling verification of the token's authenticity and integrity. JWT offers several advantages for authentication: **Statelessness, Flexibility and Payload, Security, and Interoperability,** which is suitable for APIs-based applications.

Create the Rails API app

```
rails new authentication-api-1 --api -d postgresql
```

- `rails new authentication-api`: This initializes a new Rails application named "authentication-api-1".
- `--api`: This flag specifies that you're creating an API-only application. It configures Rails to skip generating unnecessary middleware, such as those for handling assets and rendering views, focusing solely on building APIs.
- `-d postgresql`: This sets the default database for the Rails application to PostgreSQL.

Enabling CORS

Cross-Origin Resource Sharing (CORS) is a mechanism that allows servers to specify who can access their resources, thereby relaxing the Same-Origin Policy. It defines a set of HTTP headers a server can use to declare which origins are permitted to access its resources.

In the context of a Rails application, enabling CORS allows your API to be accessed by client-side applications running on different domains. This is especially relevant when your frontend (like a React or Angular app) is hosted on a different domain than your Rails backend API.

Rails provides ways to configure CORS settings. You can use gems like `rack-cors` to handle CORS configuration within your Rails application.

just uncomment the following line from the Gemfile:

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

run `bundle install` and uncomment the contents in 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]
  end
end
```

It allows requests from any origin (`"*"`) and permits various HTTP methods (GET, POST, PUT, PATCH, DELETE, OPTIONS, HEAD) for all resources. leaving it as `"*"` is a good idea when creating the app and for testing purposes but later, when you are ready to deploy. instead of allowing requests from any origin (`"*"`), specify the specific domain(s) that your frontend application will be hosted on.

For instance, if your frontend application is hosted at `https://example.com`, you would update `origins "*"` to `origins " https://example.com "` to restrict access only to that specific domain.

Configuration Devise and Devise-JWT

To manage authentication and sessions in our Rails app, we will use the gems: Devise and Devise-JWT.

Devise

- Devise is a flexible authentication solution for Rails. It provides a comprehensive set of functionalities for user authentication, including user registration, session management, password recovery, and more.
- Devise operates by creating a User model, generating routes and controllers for authentication, and providing a set of helper methods and modules to handle user authentication-related tasks.

in Gemfile add:

```
gem 'devise'
```

then run `bundle install`

Set up Devise

Run the following to create **Initial Configuration Files**:

```
rails g devise:install
```

This command generates a Devise initializer file (`config/initializers/devise.rb`). The initializer file allows you to configure various options and settings for Devise, such as default mailer settings, the default URL options, lockable settings, and more.

Then uncomment and edit the following line in `config/initializers/devise.rb:`

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

By default, Devise redirects to the sign-in page for non-HTML requests (e.g., JSON, XML). However, setting `config.navigational_formats = []` prevent Devise from using flash messages or any redirects which are a default feature and are not present in Rails API mode.

For instance, in an API-only Rails application using Devise for authentication, if you prefer to handle authentication errors or redirect yourself within the API controllers and responses, you might use this configuration to avoid Devise's default navigational behaviors for non-HTML formats.

Configuring networks

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

```
config.action_mailer.default_url_options = { host: 'localhost', port: 30
```

When using Devise or other authentication systems that involve sending emails for tasks like account confirmation, password reset, or email confirmation, setting the default URL options is crucial. This setting ensures that the links generated in these emails contain the correct host and port information.

The `config.action_mailer.default_url_options` setting in a Rails application specifies the default URL options used when generating links in emails sent via Action Mailer. In the provided example, `config.action_mailer.default_url_options` is configured with `{ host: ' localhost ', port: 3000 }`. This means that the default URL used in emails will contain `localhost:3000` as the host and port.

For instance, in production, you might have to Replace `'localhost'` with your actual production domain ex. `{ host: ' example.com ' }.`

Don't forget if you change the default Rails's port 3000 to 3001 for example, change it in the `config.action_mailer.default_url_options` and in the following line in `config/puma.rb:`

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

otherwise, leave everything as it is.

Create User model and controllers

let's generate a model named `User` and configures it to work with Devise by running the following command:

```
rails g devise User
```

then run the following command to create a database table and migrate `User` model:

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

the `db:migrate` in Ruby on Rails is used to execute any pending database migrations. Migrations in Rails are used to manage changes to the database schema, such as creating tables, modifying columns, or adding indexes.

then generate the **devise controllers** (sessions and registrations) to handle sign-ins and sign-ups by running the following command:

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

The `-c` flag specifies the controllers to generate. In this case, it generates custom controllers for sessions (authentication, sign-in, sign-out) and registrations (user sign-up, account creation).

Configure routes and response format

what if we want to edit the default routes configuration by Devise? let's make some interesting stuff in `config/routes.rb:`

```
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'
  },
  defaults: { format: :json }
end
```

`devise_for :users`: This line generates routes for the `User` model using Devise.
It sets up routes for user authentication (sign-in, sign-out), user registration, and other authentication-related actions.

`path_names` allows you to customize the path segments used in URLs for specific actions. For example, `sign_in` becomes `login`, `sign_out` becomes `logout`, and `registration` becomes `signup`.

The `path:''` part ensures that the root path is used for Devise routes. for example:
When `path: ''`:

- `sign_in` becomes `/login`
- `sign_out` becomes `/logout`
- `registration` becomes `/signup`

but if `path: 'auth':`

- `sign_in` becomes `auth/login`
- `sign_out` becomes `auth/logout`
- `registration` becomes `auth/signup`

Setting `controllers: { sessions: 'users/sessions', registrations: 'users/registrations' }`: This section specifies the custom controllers for handling sessions and registrations related to users. It tells Devise to use custom controller logic located in `app/controllers/users/sessions_controller.rb` and `app/controllers/users/registrations_controller.rb.`

Setting `defaults: { format: :json }` specifies that the default format for responses from these routes will be JSON. This is common in API-only applications, ensuring that responses are formatted as JSON by default.

Now, the endpoints will be set to log in, log out, and Sign up.

Adding new parameters to `User` model

In Devise, when you create a registration for a user (signing up), the default parameters are (email, password, password confirmation, and Remember Me), what if we want to add another parameter like the `name` attribute? To achieve that, we need to "sanitize" (or permit) these new parameters. We should add the following lines in `controllers/application_controller.rb:`

```
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
```

What is the line `before_action :configure_permitted_parameters, if: :devise_controller?`: do? This line specifies that the `configure_permitted_parameters` method should be executed as a `before_action` filter before any action in a controller that is a Devise controller (`if: :devise_controller?`).

the `configure_permitted_parameters` method: This method configures permitted parameters for Devise actions. Specifically:

- For the `sign_up` action (user registration), it permits the `name` parameter.
- For the `account_update` action (updating user account details), it also permits the `name` parameter.

know we need to create a new migration to add `name` attribute to the Users table to do that run the following command:

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

then run `rails db:migrate`

Set up Devise-JWT

- Devise-JWT is an extension or plugin for Devise that integrates JSON Web Tokens (JWT) for authentication.
- Devise-JWT hooks into Devise to issue JWT tokens upon successful authentication. Clients can then use these tokens to access protected API endpoints without the need for session management.

in Gemfile add:

```
gem 'devise-jwt'
```

then run `bundle install` .

As we said before JWT tokens consist of three parts separated by dots: a header, a payload, and a signature. The signature is created by encoding the header and payload along with a secret (private) key using a specific cryptographic algorithm (like HMAC, RSA, etc.).

This process creates a unique signature that only the server, with access to the private key, can produce. Let's generate it by running the following command:

```
bundle exec rails secret
# output: 8fe0a68f8ea133c6a174b8c603e7ac8528cf14e0b0c9c5b0fb2b5b2c8a91f2
```

This generated secret key should be kept confidential and stored securely, preferably in an environment variable or a secure credentials file in `config/credentials.yml.enc`

Which contains sensitive information such as API keys, database passwords, and other secrets encrypted using the Rails encryption mechanism.

so let's add our JWT secret key in `config/credentials.yml.enc` by running this command:

```
EDITOR='nano' rails credentials:edit
#I'm here using nano as my editor you can change it to yours
```

When you execute this command, Rails retrieves the encryption key from a secure location

When you execute this command, Rails retrieves the encryption key from a secure location (usually specified in `config/master.key` or `config/credentials.yml.key`) to decrypt it, edit it, and add our sensitive information.

then add the secret key:

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

know we need to do some JWT configuration like setting our private key as a devise_jwt_secret_key, expiration time for JWT tokens, and more so, let's do that by adding the following lines at the end of `config/initializers/devise.rb:`

```
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 = 30.minutes.to_i
end
```

in `jwt.dispatch_requests`: Specifies which HTTP requests will dispatch token issuance. In this case, it indicates that a JWT token will be issued upon a `POST` request to the `/login` endpoint.

in `jwt.revocation_requests`: Defines the requests that will revoke or invalidate JWT tokens. Here, it specifies that a JWT token will be invalidated upon a `DELETE` request to the `/logout` endpoint.

We are also setting the expiration time of the token to 30 minutes after which the user will need to log in again. you can set it according to your business needs.

Revocation process

Tokens are revoked either when they expire or when a user signs out. Once expired, a token becomes invalid, and signing out immediately revokes the token to prevent further use and users must log in again to generate new tokens.

why do we need to do that? if you sign out and that active token gets into the wrong hands, it's like giving someone a key to your secure places. They could access restricted areas or manipulate things they shouldn't. Revoking or making that token invalid ensures that even if it's stolen or misused, it won't work anymore, keeping everything safe and secure

The Devise-JWT gem provides 3 different revocation strategies. we are going to 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.*

The `JTI` can take various forms depending on the system or application implementing JWTs and the requirements for uniqueness and compatibility within that system, commonly a UUID (Universally Unique Identifier), [GUID (Globally Unique Identifier)](techtarget.com/searchwindowsserver/definiti.., or any unique string that ensures uniqueness across tokens.

To use it, you need to add the `jti` column to the `user` model. So, run the following command:

```
```

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

then make sure to add `null: false` to the `add_column` line and `unique: true` to the `add_index` line in the migration file to look like this:

```
def change
  add_column :users, :jti, :string, null: false
  add_index :users, :jti, unique: true
end
```

then run `rails db:migrate`

Then, you have to add the strategy to the model class and configure it accordingly in `models/user.rb:`

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

  devise :database_authenticatable, :registerable, :recoverable,
         :validatable, :jwt_authenticatable, jwt_revocation_strategy: self
end
```

`devise` **Method:** This line configures Devise for this `User` model and specifies various authentication modules:

- `:database_authenticatable`: Allows authentication via a username/email and password stored in the database.
- `:registerable`: Provides registration-related functionalities.
- `:recoverable`: Enables password reset functionality.
- `:validatable`: Adds validations for email and password.
- `:jwt_authenticatable`: Enables JWT-based authentication for this model.
- `jwt_revocation_strategy: self`: Specifies the revocation strategy, using `self`, which indicates that the JWT revocation strategy is employed for this `User` model.

Handling JSON structure by jsonapi-serializer

`Jbuilder` and `jsonapi-serializer` are two Ruby gems used in Ruby on Rails applications for handling JSON serialization and formatting, particularly when building APIs.

firstly, let's start with `jsonapi-serializer`. we need to add it to the Gemfile:

```
gem 'jsonapi-serializer'
```

then run `bundle install`

then run the following command to create **UserSerializer:**

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

This serializer creates JSON responses following the JSON:API convention. It generates a serializer with the attributes mentioned that we can later call using the following line:

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

you can replace `<user>` with the actual variable containing the user information.

Devise controllers

finally, we reach the most important part which is the logical behavior of our app in different situations like user registration, login, and logout with some devise helper methods.

let's start with registration, you need to write the following lines

in `controllers/users/registrations_controller.rb:`

```
class Users::RegistrationsController < Devise::RegistrationsController

  private

  def respond_with(current_user, _opts = {})
    if resource.persisted?
      render json: {
        status: {code: 200, message: 'Signed up successfully.'},
        data: UserSerializer.new(current_user).serializable_hash[:data][
      }
    else
      render json: {
        errors: current_user.errors.full_messages
      }, status: :unprocessable_entity
    end
  end
end
```

the `respond_with` is a helper method that is overridden to customize the response format upon user sign-up.

The `resource.persisted?` is a method in Devise's context used to check whether the user registration was successful by confirming if the user record has been saved to the database If successful, it indicates that the user has been successfully created and persisted in the database.

then let's write login and logout, you need to write the following lines in `controllers/users/sessions_controller.rb:`

```
class Users::SessionsController < Devise::SessionsController

  private
  def respond_with(current_user, _opts = {})
    render json: {
      status: {
        code: 200, message: 'Logged in successfully.',
        data: {
            user: UserSerializer.new(current_user).serializable_hash[:da
          # return token in the body instead of response header but it's
          # token: request.env['warden-jwt_auth.token']
        }
      }
    }, status: :ok
  end
  def respond_to_on_destroy
    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
```

the `respond_with` is a helper method that is overridden to customize the response format upon user login and if it is unsuccessfully logged in it returns an error response message "email or password is invalid" by default.

the `respond_to_on_destroy` is a helper method that is overridden to customize the response format upon user logout.

Handling JSON structure by JBuilder

we need to uncomment `jbuilder` gem
in Gemfile:

```
gem "jbuilder"
```

then run `bundle install`

then create files that write out API response structures for every situation.

in `views/users/registrations/registration_success.json.jbuilder`

```
json.status do
  json.code 200
  json.message "Signed up successfully."
  json.data do
    json.user do
      json.id current_user.id
      json.email current_user.email
      json.name current_user.name
    end
  end
end

=begin
{
    "status":{
        "code":200,
        "message":"Signed up successfully.",
        "data":{
            "id":1,
            "email":"test@email.com",
            "name":"tom"
        }
    }
}
=end
```

in `views/users/sessions/session_success.json.jbuilder`

```
json.status do
  json.code 200
  json.message "Logged in successfully......"
  json.data do
    json.user do
      json.id current_user.id
      json.email current_user.email
      json.name current_user.name
      # return token in the body instead of response header but it's not
      # token: request.env['warden-jwt_auth.token']
    end
  end
end
```

in `views/users/sessions/session_destroy.json.jbuilder`

```
json.status 200
json.message "logged out successfully"
```

in `views/users/sessions/session_destroy_errors.json.jbuilder`

```
json.status 401
json.message "Couldn't find an active session."
```

in `controllers/users/registrations_controller.rb:`

```
class Users::RegistrationsController < Devise::RegistrationsController

  private

   def respond_with(current_user, _opts = {})
    if resource.persisted?
      render "users/registrations/registration_success", status: :ok
    else
      render json: {
        errors: current_user.errors.full_messages
      }, status: :unprocessable_entity
    end
  end
end
```

in `controllers/users/sessions_controller.rb:`

```ruby
class Users::SessionsController < Devise::SessionsController

  private
  def respond_with(current_user, _opts = {})
    render "users/sessions/session_success", status: :ok
  end

  def respond_to_on_destroy
    if current_user
      render "users/sessions/session_destroy", status: :ok
    else
      render "users/sessions/session_destroy_errors", status: :unauthori
    end
  end
end
```

Jbuilder offers more direct control and flexibility for ad-hoc JSON rendering, while serializers are dedicated tools for structured and consistent object serialization, especially in API contexts. Both serve different purposes and can be chosen based on the specific requirements of your Rails application.

Fix some problems

DisabledSessionError

In our Rails API-only application, sessions are turned off by default. However, Devise, our authentication tool, depends on sessions to operate. In older versions of Rails before 7, sessions were transmitted as a hash. Even if sessions were disabled, Devise could still write into that hash. But starting from Rails 7, sessions are represented as an ActionDispatch::Session object, which isn't writable in API-only setups because this specific session mechanism is disabled. If we attempt to use Devise in this setup, we'll encounter an error due to this lack of session support:

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

so, there is a sample workaround solution to fix that by adding the following lines in `app/controllers/concerns/rack_sessions_fix.rb`

```ruby
module RackSessionsFix
  extend ActiveSupport::Concern
  class FakeRackSession < Hash
    def enabled?
      false
    end
    def destroy; end
  end
  included do
    before_action :set_fake_session
    private
    def set_fake_session
      request.env['rack.session'] ||= FakeRackSession.new
    end
  end
end
```

This workaround attempts to create a pseudo-session (`FakeRackSession`) and assigns it to `rack.session` when the middleware for sessions is not available.

we need to add it to our controllers, in `controllers/users/registrations_controller.rb:`

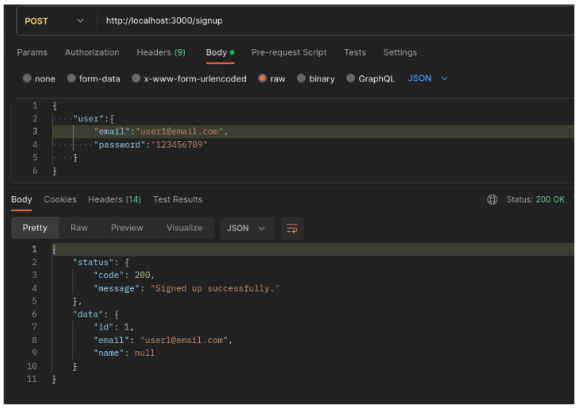
```ruby
class Users::RegistrationsController < Devise::RegistrationsController

  include RackSessionsFix
    ...
end
```

in `controllers/users/sessions_controller.rb:`

```ruby
class Users::SessionsController < Devise::SessionsController
  include RackSessionsFix
     ...
end
```

password confirmation



As we can see Devise allows the signup request body to be submitted without a password confirmation field. However, if a password confirmation field is present, Devise will perform the validation based on its value.

We aim to restrict the acceptance of requests without the password confirmation field so let's add this validation to the `models/user.rb:`

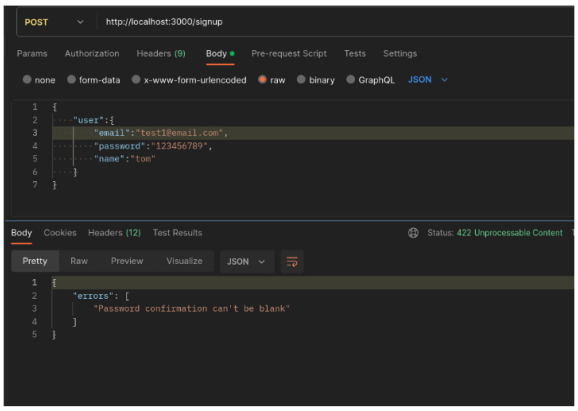
```
validates :password_confirmation, presence: true, on: %i[create update]
```

This validation ensures the presence of `password_confirmation` specifically during `create` and `update` actions.

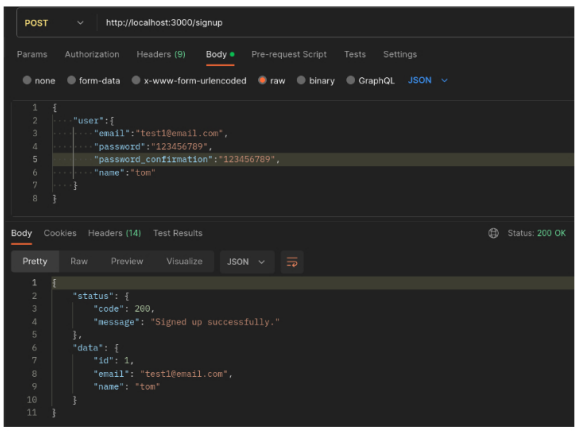
Testing with Postman

run `rails s` to start the server.

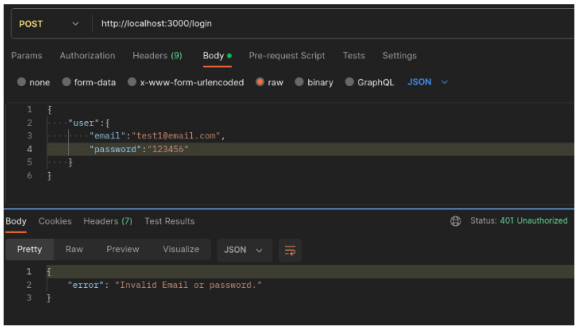
Creating a user: (invalid request):



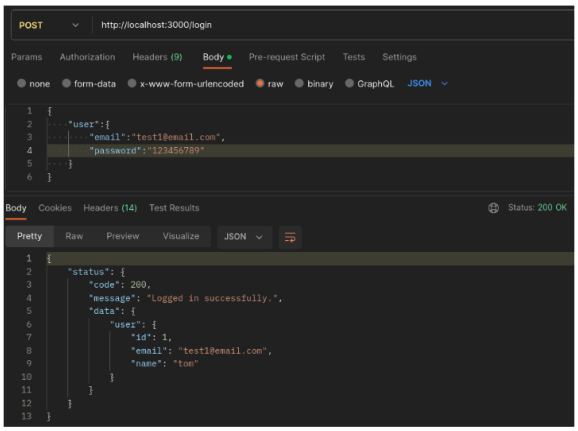
Creating a user: (valid request):



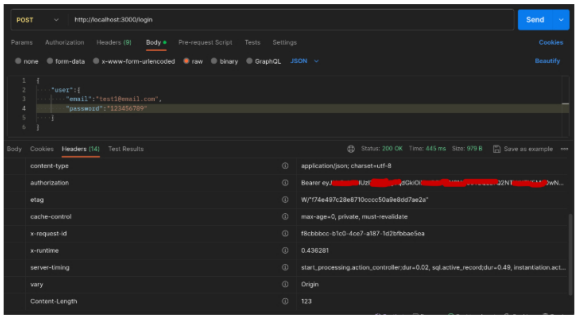
login a user: (invalid request):



login a user: (valid request):



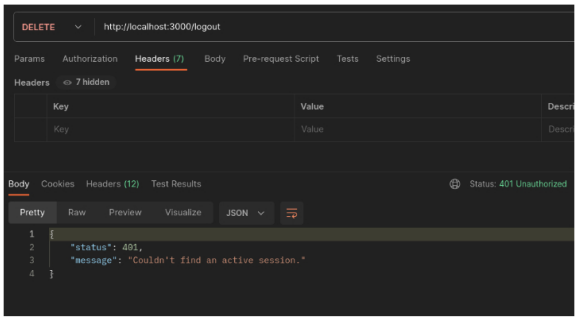
we can check that the authorization token was received:



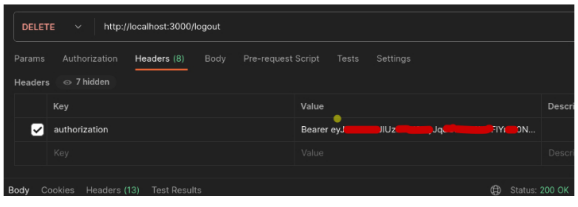
Logging out:

When we log out the authorization token needs to be passed, for testing, we have to manually add it to Postman:

logout a user: (invalid request):



logout a user: (valid request):



Conclusion

This mechanism carries a drawback: it doesn't support user login from multiple devices. If a user logs out from one device, it leads to the revocation of all tokens, affecting the ability to remain logged in on other devices. In the next blog, we will discuss how to resolve and write a more flexible and dynamic mechanism that deals with all of that.

Ruby on Rails Ruby backend JWT REST API software development