Ruby on Roda

REST APIs with Roda & Sequel

Mateusz Urbański

Ruby on Roda

Mateusz Urbański

Version 1.0, May 2021

Table of Contents

Introduction

Thank you for reading Ruby on Roda. If you find any mistakes while reading this book, please email them to mateuszurbanski@yahoo.pl.

If you have a problem with your code, please put the code on GitHub and link me to it in the email to clone it and attempt to reproduce the problem myself.

If you don't understand something, then it's more likely that I made a mistake and rushed it when I wrote it. Let me know!

Who is this book for?

This book is best suited for experienced Ruby developers. If you're new to Ruby, you will probably understand some of the terms I'm using here, but to get the most out of this guide, you should have some Rails experience under your belt.

What are we going to build?

The purpose of this book is to teach you how to create JSON API using Roda and Sequel toolset.

Acknowledgements

I want to thank my amazing wife Marta for supporting this new adventure and encouraging me to pursue my passion for writing, teaching, and programming!

I also want to thank all my friends who read the book and gave me feedback about it. Most of them told me they learned new things, which made me extremely happy and motivated me to keep going.

Source code

You can find the source code for this book on my Github Profile.

Chapter 1. Getting started

1.1. Introduction to Roda

Roda is a web framework built on top of Rack, created by Jeremy Evans, that started as a fork of Cuba and was inspired by Sinatra. The following is the most straightforward app you can make in Roda, which returns "Hello world!" for every request:

config.ru

```ruby
require "roda"
Roda.route { "Hello world!" }
run Roda.app
```

Roda has a unique approach to routing compared to Rails, Sinatra, and other Ruby web frameworks. In Roda, you route incoming requests dynamically as they come.

```ruby
class App < Roda
  route do |r|
    r.on "posts" do
      r.is "recent" do
        r.get do
          @posts = Post.recent
        end
      end
    end
  end
end
```

First, we create an `App` class that inherits from the `Roda` class. The route block is called whenever a new request comes in. It is yielded to an instance of a subclass of `Rack::Request` with some additional methods for matching routes. By convention, this argument is named `r` (for "request").

If the request's path starts with `/posts,` the request will be matched by the `r.on` call, calling the given block. Next, it will be matched by the `r.is` call if the path continues and ends with `/recent` (`r.is` is a terminal matcher). Finally, `r.get` will match only `GET` requests. Altogether, this route block handles `GET /posts/recent` requests by assigning a list of recent posts.

The reason why this is called a "routing tree" is that routing is branched. If the request doesn't start with `/posts`, the whole `r.on` "posts" block ("branch") is immediately discarded, and routing continues to the following branches.

The route block is called whenever a request is coming, so this routing is happening in real-time. It means that you can handle the request while you're routing it.

```ruby
class App < Roda
  plugin :all_verbs

  route do |r|
    r.on "posts" do
      r.is ":id" do |id|
        @post = current_user.posts.find(id)

        r.get do
          @post
        end

        r.put do
          @post.update(r.params["post"])
        end

        r.delete do
          @post.destroy
        end
      end
    end
  end
end
```

Since all of these 3 `/posts/:id` routes have first to find the post, we can assign the post as soon as we know that the path will be `posts/:id` and then we reference it anywhere down that branch. In other web frameworks, you would solve this with before filters to avoid duplication, but that splits code making it harder to follow. With Roda, you can write DRY code in a very readable way.

This is a new concept, and it opens a whole new world of routing possibilities. From other web frameworks, we are used to routing only by the request path and method.

One downside of using Roda's routing tree is that, since routes are not stored in any data structure (because requests are routed dynamically as they come in), you cannot introspect the routes of the routing tree.

However, you can leave comments above your routes using a special syntax, and use the roda-route_list plugin to parse those comments and print the routes.

By design, Roda has a minimal core, providing only the essentials. All additional features are loaded via plugins that ship with Roda. This is why Roda is a "web framework toolkit" using a combination of Roda plugins you can build your flavor of the web framework that suits your needs and choose precisely the amount of complexity you need.

1.2. Introduction to Sequel

I've used ActiveRecord for most of my Ruby life. While I was in Rails, I couldn't imagine why I would want to use anything else. When I moved away from Rails, I was still using ActiveRecord at first, but some things started to bother me:

- limited query interface, you very quickly have to switch to SQL strings
- no good low-level query interface
- it isn't easy to set up ActiveRecord in a non-Rails project
- dependency on ActiveSupport and its core extensions

I wanted to try another ORM. I've thought about ROM, but he did not convince me. I wanted a gem that also implements the ActiveRecord pattern but better implementation than the ActiveRecord gem.

I've heard about Sequel before, and I decided to give him a chance. The Sequel has all the features I wanted from ActiveRecord and so much more. Jeremy Evans, the author of Sequel, keeps Sequel at 0 issues and maintains a mailing list to get help with anything.

While ActiveRecord is one monolithic gem, Sequel utilizes a plugin system. The Sequel consists of a relatively thin core, which gives you the most common behavior, and you can then choose to add additional functionality via plugins.

```ruby
require "sequel" # loads the core

DB = Sequel.connect("postgres:///my_database")

Sequel::Model.plugin :validation_helpers
Sequel::Model.plugin :auto_validations
Sequel::Model.plugin :prepared_statements
Sequel::Model.plugin :single_table_inheritance
```

Because of this design Sequel loads 5 times faster than ActiveRecord.

Creating the migration files is very similar to ActiveRecord in the layout. It is important to note that you should still include a number to the beginning of your file name, as Sequel requires it.

db/migrate/01_dogs.rb

```ruby
Sequel.migration do
  change do
    create_table(:dogs) do
      primary_key :id
      String :name, null: false
      String :breed, null: false
      foreign_key :owner_id, :people
    end
  end
end
```

db/migrate/02_owners.rb

```ruby
Sequel.migration do
  change do
    create_table(:owners) do
      primary_key :id
      String :name, null: false
    end
  end
end
```

Models in Sequel are set up very similarly to ActiveRecord.

```ruby
class Dog < Sequel::Model
  many_to_one :person

  def validate
    super
    errors.add(:name, 'must be present') if name.empty?
    errors.add(:breed, 'must be present') if breed.empty?
  end
end

class Owner < Sequel::Model
  one_to_many :dogs

  def validate
    super
    errors.add(:name, "must be present") if name.empty?
  end
end
```

Queries within Sequel are somewhat similar to those in ActiveRecord. Using `Dog.all` will return all dogs within the database. You can use `.first` and `.last` to get the first and last object in the database. `.order` allows you to order the instance within the database. You can use `.where()` to find all instances within the database that match the given object key.

On the other hand, Sequel allows you to write low-level queries using the same query interface you use for models! Instead of going through models, You can go through the `Sequel::Database` object directly, and the records will be returned as simple Ruby hashes.

```
Dogs.where(breed: "Labrador").first      #=> #<Dog breed="Labrador" ...>
DB[:dogs].where(breed: "Labrador").first #=> {bread: "Labrador", ...}
```

Jeremy Evans put a lot of effort into supporting as many Postgres features as possible in Sequel:

- Support for reading and writing to JSON columns with nice, readable API.
- Database views creation using the query interface.
- Support for Postgres cursors.
- Support for PostgresSQL's LISTEN/NOTIFY commands.
- Creating Partitioned/Unlogged tables.
- Support for PostgreSQL's COPY feature.
- And many more…

I have only scratched the surface of Sequel's features. ActiveRecord was long my ORM of choice only because it's part of Rails. After using Sequel for some time, I have found it to be much more stable (0 issues maintained), better designed, more performant, and more advanced than ActiveRecord.

Chapter 2. Starting project

2.1. Project Setup

Before we start make sure that you have Ruby 3.0.0 installed on your machine:

```
$ ruby -v
ruby 3.0.0p0 (2020-12-25 revision 95aff21468) [x86_64-darwin20]
```

So let’s start with creating a directory that will store our source code:

```
mkdir todo_api
cd todo_api
```

The next step will be creating our `Gemfile`:

Gemfile

```
# frozen_string_literal: true

source 'https://rubygems.org'

ruby '3.0.0'

# Routing Tree Web Toolkit.
gem 'roda', '>= 3.41'

# Use Puma as the app server.
gem 'puma', '~> 5.2'

# A make-like build utility for Ruby.
gem 'rake'

# Sequel: The Database Toolkit for Ruby.
gem 'sequel', '>= 5.41'

# Faster SELECTs when using Sequel with pg.
gem 'sequel_pg', '>= 1.14'

# A runtime developer console and IRB alternative with powerful introspection
capabilities.
gem 'pry'

# YARD is a Ruby Documentation tool. The Y stands for "Yay!"
gem 'yard'

# A fast JSON parser and Object marshaller as a Ruby gem.
```

```
gem 'oj'

# bcrypt-ruby is a Ruby binding for the OpenBSD bcrypt() password hashing algorithm,
allowing you to easily store a secure hash of your users' passwords.
gem 'bcrypt'

# Validation library with type-safe schemas and rules.
gem 'dry-validation'

# A powerful logger for Roda with a few tricks up it's sleeve.
gem 'roda-enhanced_logger'

# Organize your code into reusable components.
gem 'dry-system', '0.18.1'

# A toolkit of support libraries and Ruby core extensions extracted from the Rails
framework.
gem 'activesupport'

# Plugin that adds BCrypt authentication and password hashing to Sequel models.
gem 'sequel_secure_password'

# A gem providing "time travel" and "time freezing" capabilities, making it dead
simple to test time-dependent code. It provides a unified method to mock Time.now,
Date.today, and DateTime.now in a single call.
gem 'timecop'

# Ruby internationalization and localization (i18n) solution.
gem 'i18n'

group :development, :test do
  # A library for setting up Ruby objects as test data.
  gem 'factory_bot'

  # A Ruby gem to load environment variables from `.env`.
  gem 'dotenv'

  # A Ruby static code analyzer and formatter, based on the community Ruby style
guide.
  gem 'rubocop'

  # An extension of RuboCop focused on code performance checks.
  gem 'rubocop-performance'

  # Code style checking for RSpec files.
  gem 'rubocop-rspec'

  # Thread-safety analysis for your projects.
  gem 'rubocop-thread_safety'

  # Code style checking for Sequel.
  gem 'rubocop-sequel'

  # A RuboCop plugin for Rake
  gem 'rubocop-rake'

  # RSpec meta-gem that depends on the other components.
  gem 'rspec'

  # Rack::Test is a layer on top of Rack's MockRequest similar to Merb's
RequestHelper.
  gem 'rack-test'
end
```

and now run the `bundle install` command to install dependencies.

Now let’s add the `.ruby-version` file, which specifies a version number of Ruby we want to use in our project:

.ruby-version

```
ruby-3.0.0
```

We will also add a `.gitignore` file that will list files we don’t want to track with our version control:

.gitignore

```
.env.development
.env.test
/doc/*
.yardoc/*
```

We are going to use Rubocop to make sure our code fits nicely into the Ruby style guide, so let’s add `.rubocop.yml` file which store Rubocop configuration:

.rubocop.yml

```
require:
  - rubocop-rspec
  - rubocop-performance
  - rubocop-thread_safety
  - rubocop-sequel
  - rubocop-rake

AllCops:
  NewCops: enable
  EnabledByDefault: true
  TargetRubyVersion: 3.0.0
  Exclude:
    - vendor/bundle/**/*
```

```yaml
Lint/ConstantResolution:
  Enabled: false

Style/Copyright:
  Enabled: false

Style/MissingElse:
  Enabled: false

Style/MethodCallWithArgsParentheses:
  Exclude:
    - 'spec/**/*'

Style/StringHashKeys:
  Exclude:
    - app.rb
    - spec/**/*

RSpec/NestedGroups:
  Max: 5

Layout/LineLength:
  Max: 125
  IgnoredPatterns: ['(\A|\s)#']

Layout/SpaceBeforeBrackets:
  Exclude:
    - 'db/migrate/*'

Metrics/BlockLength:
  Max: 40
  Exclude:
    - 'spec/**/*'
    - 'app.rb'

Lint/ToJSON:
  Exclude:
    - app/serializers/**/*

RSpec/ExpectInHook:
  Enabled: false

RSpec/MessageExpectation:
  Enabled: false

RSpec/StubbedMock:
  Enabled: false

RSpec/MessageSpies:
  Enabled: false
```

```
RSpec/MultipleExpectations:
  Enabled: false

RSpec/MultipleMemoizedHelpers:
  Max: 10

Naming/VariableNumber:
  Enabled: false

Metrics/ClassLength:
  Exclude:
    - 'app.rb'

Layout/SingleLineBlockChain:
  Enabled: false

Layout/RedundantLineBreak:
  Enabled: false

Bundler/GemVersion:
  Enabled: false
```

This is just the setup that I use with apps built with Roda but feel free to modify it or add your favorite Rubocop configuration. Rubocop community is constantly adding new rules. There may be a scenario that my code that breaks newly added Rubocop rules. In that case, do not hesitate to update the code or disable Rubocop rules.

Now we will add a configuration file for YARD, which will be used to document our code. Let's create a `.yardopts` file at the root of our project directory:

.yardopts

```
--private
```

In the next section we will start organizing our application with dry-system.

2.2. Organize application structure with dry-system

If one day you decide to give up on Rails (as we did) and use Sinatra or Roda instead, even the first server launch might become painful. Rails were carefully autoloading all the gems, running initializers, managing dependencies. Now we have to manually add `require` keywords to dozens of files to launch your app. You have to do the same procedure for all the files with dependencies. It makes your app way faster but doesn't bring you any pleasure.

Dry-system is designed for solving such problems. It allows you to divide your app into independent encapsulated components, facilitates the initializing process, provides an Inversion of Control (IoC)- a container to register and instantiate dependencies.

The main API of dry-system is the abstract container that you inherit from. It allows you to configure basic settings and exposes APIs for requiring files quickly. The container is the entry point to your application, and it encapsulates the application state.

So let's create our `Application` container. We need to create a `application.rb` file in our `system` folder:

/system/application.rb

```ruby
# frozen_string_literal: true

require 'bundler/setup'
require 'dry/system/container'

# {Application} is a container that we use it to register dependencies we need to
call.
class Application < Dry::System::Container
  # Provide environment inferrerr.
  use :env, inferrer: -> { ENV.fetch('RACK_ENV', 'development') }

  # we set 'lib' relative to `root` as a path which contains class definitions that
can be auto-registered.
  config.auto_register = %w[lib app]

  # this alters $LOAD_PATH hence the `!`
  load_paths!('lib', 'app')
end
```

First, we `require 'bundler/setup'`, that ensures we're loading Gemfile defined gems.

Next, we `require 'dry/system/container'` to have access to `Dry::System::Container` class, which our `Application` will inherit from.

`use :env, inferrer: -> { ENV.fetch('RACK_ENV', 'development') }` includes dry-system plugin that determines the runtime environment.

`config.auto_register = %w[lib app]` Configure the folder in the root folder where the dependencies will be automatically registered. It means that the `require` method will be called for all the files in this folder.

`load_paths!('lib', 'app')` add the folders passed in arguments to `$LOAD_PATH`, making it easier to use `require`.

Before the launch, any app usually needs to initialize external libraries. For this purpose, create one more `system/boot` folder that will store our bootable dependencies. It's the same as a `config/initializers` folder in Ruby on Rails. After `.finalize!` is called on our `Application` class, the `require` for all the files in this folder will be executed.

Our first dry-system component will be responsible for loading environment variables. Let's create `environment_variables.rb` file in `system/boot` folder:

/system/boot/environment_variables.rb

```ruby
# frozen_string_literal: true

# This file contains setup for environment variables using Dotenv.

Application.boot(:environment_variables) do
  start do
    # Get Application current environment.
    env = Application.env

    # Load environment variables if current environment is development or test.
    if %w[development test].include?(env)
      require 'dotenv'

      Dotenv.load('.env', ".env.#{env}")
    end
  end
end
```

First, get the current environment of our application using `Application.env`. Then we check our current environment. If it's test or production, we require `dotenv` and load environment variables from files using `Dotenv.load`.

Next, we need a component that will handle database connection for us. Let’s create `database.rb` file in the `system/boot` folder:

/system/boot/database.rb

```
# frozen_string_literal: true

# This file contain code to setup the database connection.

Application.boot(:database) do |container|
  # Load environment variables before setting up database connection.
  use :environment_variables

  init do
    require 'sequel/core'
  end

  start do
    # Delete DATABASE_URL from the environment, so it isn't accidently passed to
subprocesses.
    database = Sequel.connect(ENV.delete('DATABASE_URL'))

    # Register database component.
    container.register(:database, database)
  end
end
```

Before setting up a database connection, we need to have access to environment variables which stores our `DATABASE_URL`. Because of that, we write `use :environment_variables` at the beginning of our component to auto-boot the required dependency and make it available in the booting context.

Then we set up our database connection using `Sequel.connect`, and we register our database component so we will be able to use it later using `Application['database']` notation.

Our next component will be responsible for configuring our `Application` logger. Let’s create `logger.rb` in `/system/boot` folder:

/system/boot/logger.rb

```ruby
# frozen_string_literal: true

# This file contains logger configuration.

Application.boot(:logger) do
  init do
    require 'logger'
  end

  start do
    # Define Logger instance.
    logger = Logger.new($stdout)

    # Because the Logger's level is set to WARN , only the warning, error, and fatal
messages are recorded.
    logger.level = Logger::WARN if Application.env == 'test'

    # Register logger component.
    register(:logger, logger)
  end
end
```

First, we create a new `Logger` instance that outputs to the standard output stream. We set the `Logger` level to `WARN` in the `test` environment to not see the logs during launch tests with the `rspec` command. The last step is to register our component.

Oj is our choice for serialization, so lets create a component for it:

/system/boot/oj.rb

```ruby
# frozen_string_literal: true

# This file contains configuration for oj gem.

Application.boot(:oj) do
  init do
    require 'oj'
  end

  start do
    # :compat attempts to extract variable values from an Object using
    # to_json() or to_hash() then it walks the Object's variables if neither is found.
    Oj.default_options = { mode: :compat }
  end
end
```

Component for the oj gem configuration is really simple, we require oj gem and set the mode to :compat. More about oj modes can be found here.

BCrypt, a hashing algorithm used to store passwords securely. It is implemented in Ruby via the bcrypt gem. Component that configures bcrypt looks like this:

/system/boot/bcrypt.rb

```ruby
# frozen_string_literal: true

# This file contains configuration for bcrypt.

Application.boot(:bcrypt) do
  init do
    require 'bcrypt'
  end

  start do
    # Set BCrypt::Engine.cost to 1 in test environment to speedup tests.
    BCrypt::Engine.cost = 1 if Application.env == 'test'
  end
end
```

Here we require bcrypt gem and set BCrypt::Engine.cost to 1 to speedup our tests.

We need to have the option to translate our application to multiple languages in the future, in that case, we need to have the configuration for i18n Ruby Gem:

/system/boot/i18n.rb

```ruby
# frozen_string_literal: true

# This file contains setup for Ruby internationalization and localization (i18n).

Application.boot(:i18n) do
  init do
    require 'i18n'
  end

  start do
    # Load all locale .yml files in /config/locales folder.
    I18n.load_path << Dir["#{File.expand_path('config/locales')}/*.yml"]

    # Add :en to to the list of available locales.
    I18n.config.available_locales = %i[en]
  end
end
```

Here we require the i18n gem, we load all locale .yml files in /config/locales folder, then we create a list of available locales using I18n.config.available_locales method.

Let’s also add our translations to the `/config/en.yml` file that I prepared earlier:

/config/locales/en.yml

```
en:
  invalid_params: Your query contains incorrectly formed parameters.
  something_went_wrong: Something went wrong.
  invalid_email_or_password: Invalid email or password.
  invalid_authorization_token: Invalid authorization token.
  not_found: Record not found.
```

To secure our `JSON API`, we will use token-based authentication, and the ActiveSupport::MessageVerifier class will generate the tokens. Let’s create a component that will configure the `ActiveSupport` module:

/system/boot/active_support.rb

```
# frozen_string_literal: true

# This file contains configuration for ActiveSupport module.

Application.boot(:active_support) do
  init do
    require 'active_support/message_verifier'
    require 'active_support/json'
  end

  start do
    # Sets the precision of encoded time values to 0.
    ActiveSupport::JSON::Encoding.time_precision = 0
  end
end
```

Here we the require `active_support/message_verifier`, `active_support/json` and we set `ActiveSupport::JSON::Encoding.time_precision` to 0 to simplify testing timestamps values in our tests.

And last but not least, we need to create the component that will configure our Sequel models:

/system/boot/models.rb

```ruby
# frozen_string_literal: true

# This file contains configuration for Sequel Models.

Application.boot(:models) do
  init do
    require 'sequel/model'
  end

  start do
    # Whether association metadata should be cached in the association reflection.
    # If not cached, it will be computed on demand.
    # In general you only want to set this to false when using code reloading.
    # When using code reloading, setting this will make sure that if an associated class is removed or modified,
    # this class will not have a reference to the previous class.
    Sequel::Model.cache_associations = false if Application.env == 'development'

    # The validation_helpers plugin contains validate_* methods designed to be called inside Model#validate to perform validations.
    Sequel::Model.plugin(:auto_validations)

    # The validation_helpers plugin contains validate_* methods designed to be called inside Model#validate to perform validations.
    Sequel::Model.plugin(:validation_helpers)

    # The prepared_statements plugin modifies the model to use prepared statements for instance level inserts and updates.
    Sequel::Model.plugin(:prepared_statements)

    # Allows easy access all model subclasses and descendent classes.
    Sequel::Model.plugin(:subclasses) unless Application.env == 'development'

    # Creates hooks for automatically setting create and update timestamps.
    Sequel::Model.plugin(:timestamps)

    # Allows you to use named timezones instead of just :local and :utc (requires TZInfo).
    Sequel.extension(:named_timezones)

    # Use UTC time zone for saving time inside database.
    Sequel.default_timezone = :utc

    # Freeze all descendent classes. This also finalizes the associations for those classes before freezing.
    Sequel::Model.freeze_descendents unless Application.env == 'development'
  end
end
```

This component is pretty big. We require `sequel/model`, and we enable multiple Sequel plugins that we will use in our models. Plugins are modules that include submodules for model class methods, model instance methods, and model dataset methods. More about Sequel plugins can be found here.

That’s it. We’ve described the container with its components and how it should work. Now you can run it by calling a `.finalize!` method. After that, the application classes will be automatically registered, and the external dependencies from the boot folder will be initialized. Here is how we can launch our container:

```
require_relative './system/application'
Application.finalize!
```

Always running the container this way is not the greatest solution. So let’s save this logic as a separate `system/boot.rb` file:

/system/boot.rb

```
# frozen_string_literal: true

# This file is responsible for loading all configuration files.

require_relative 'application'

require 'pry'
require 'securerandom'
require 'dry-validation'

# Register automatically application classess and the external dependencies from the /system/boot folder.
Application.finalize!

# Add exsiting Logger instance to DB.loggers collection.
Application['database'].loggers << Application['logger']

# Freeze internal data structures for the Database instance.
Application['database'].freeze unless Application.env == 'development'
```

Now launching our `Application` container class is much better:

```
require_relative './system/boot'
```

Launching this inside IRB will raise `Sequel::Error (Sequel::Database.connect takes either a Hash or a String, given: nil)`. That’s because we do not configure our database connection. We will fix that in a second.

In the end, let’s create an empty `app` and `lib` folder that will store our application code.

```
mkdir app lib
```

That’s it, we configured and organized our application and its components with dry-system.

2.3. Database configuration

This database that we're connecting to doesn't exist yet, but we can quickly create it by using the `createdb` command-line tool. We need two databases, for the development and test environment:

```
createdb todo-api-development
createdb todo-api-test
```

Now let's create `.env` files for the development and test environment that dotenv will use to load environment variables:

.env.development

```
DATABASE_URL=postgres:///todo-api-development
```

.env.test

```
DATABASE_URL=postgres:///todo-api-test
```

Also, let's add `.env.development.template` and `.env.test.template` files. This will help people who will launch our project. We will also use those environment templates during CI configurations that we will do later.

.env.development.template

```
DATABASE_URL=postgresql://localhost/todo-api-development?user=postgres&
password=postgres
```

.env.test.template

```
DATABASE_URL=postgresql://localhost/todo-api-test?user=postgres&password=postgres
```

A database without tables doesn't do very much. It's good practice to create tables within a database by using migrations; Ruby files allow us to gradually build up our database tables. Sequel has the concept of migrations too, but we need to do a bit of setup first before we can create our first migration. That setup will involve creating a `Rakefile` within our project will then provide us with some tasks to create and run these migrations. Let's create that `Rakefile` now:

Rakefile

```
# frozen_string_literal: true

# Rakefile contains all the application-related tasks.

require_relative './system/application'

# Enable database component.
```

```
Application.start(:database)

# Enable logger componnent.
Application.start(:logger)

# Add existing Logger instance to DB.loggers collection.
Application['database'].loggers << Application['logger']

migrate =
  lambda do |version|
    # Enable Sequel migration extension.
    Sequel.extension(:migration)

    # Perform migrations based on migration files in a specified directory.
    Sequel::Migrator.apply(Application['database'], 'db/migrate', version)

    # Dump database schema after migration.
    Rake::Task['db:dump'].invoke
  end

namespace :db do
  desc 'Migrate the database.'
  task :migrate do
    migrate.call(nil)
  end

  desc 'Rolling back latest migration.'
  task :rollback do |_, _args|
    current_version = Application['database'].fetch('SELECT * FROM
schema_info').first[:version]

    migrate.call(current_version - 1)
  end

  desc 'Dump database schema to file.'
  task :dump do
    # Dump database schema only in development environment.
    development = Application.env == 'development'

    sh %(pg_dump --schema-only --no-privileges --no-owner -s #{Application[
'database'].url} > db/structure.sql) if development
  end

  desc 'Seed database with test data.'
  task :seed do
    sh %(ruby db/seeds.rb)
  end
end

desc 'Generate project documentation using yard.'
task :docs do
```

```
  sh %(yard doc *.rb app/ lib/)
end
```

First we need to require_relative './system/container' to access our Application container class. We need to start database and logger components and add the existing Logger instance to the DB.loggers collection. We don't need whole Application components here, so we are not using our /system/boot.rb script.

Our Rakefile contains 5 tasks:

- rake db:dump - Dumps database schema to file.
- rake db:migrate - Migrate the database.
- rake db:rollback - Roll back latest migration.
- rake db:seed - Seed database with test data.
- rake docs - Generate project documentation using yard.

Now let's create empty db/seeds.rb, which will be used to prepopulate our database with test data, and db/migrate folder that will store our database migrations:

/db/seeds.rb

```
# frozen_string_literal: true

# seeds.rb file is launched during rake db:seed command.
```

```
mdkir db/migrate
```

We are ready to create our first migration that will enable uuid-ossp and citext PostgreSQL extensions:

/db/migrate/001_enable_postgresql_extensions.rb

```ruby
# frozen_string_literal: true

Sequel.migration do
  up do
    execute <<-SQL
      CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
      CREATE EXTENSION IF NOT EXISTS "citext";
    SQL
  end

  down do
    execute <<-SQL
      DROP EXTENSION IF EXISTS "uuid-ossp";
      DROP EXTENSION IF EXISTS "citext";
    SQL
  end
end
```

Let’s run our migration for development and test database:

```
$ rake db:migrate && RACK_ENV=test rake db:migrate
```

After every migration, the database schema will be dumped to `structure.sql.` file.

2.4. Interactive Console

In Ruby on Rails, `console` is a command-line program for interacting with the Rails applications. It has the full power of the Ruby language and Rails environment. To have the same functionality in our application, we need to create it by ourselves.

Let's create new file at `bin/console`:

/bin/console

```
# !/usr/bin/env ruby

# The ruby bin/console let's you interact with your application from the command line.

require_relative '../system/boot'

Pry.start
```

In this file, we start it with a shebang that tells the program to execute Ruby.

Then we require our boot script: `require_relative '../system/boot'` and start `Pry` with `Pry.start` command.

We can now run this console and use it to interact with our application. To start it, run:

```
$ ruby bin/console
[1] pry(main)> Application['database']
=> #<Sequel::Postgres::Database: "postgres:///todo-api-development">
```

2.5. Continuous integration

GitHub Actions is an automation platform that you run directly from inside a GitHub repository.

Using GitHub Actions, you build workflows triggered by any event. These workflows run arbitrary code as Jobs, and you can piece together multiple Steps to achieve pretty much whatever you want. The most obvious use case for this new platform is to build a testing CI/CD pipeline.

Let’s create our Github Actions configuration file at `/.github/workflows/ci.yml`:

/.github/workflows/ci.yml

```
name: Continuous integration
on: push

jobs:
  verify:
    name: Build
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:latest
        env:
          POSTGRES_USER: postgres
          POSTGRES_DB: todo-api-test
          POSTGRES_PASSWORD: postgres
        ports: ["5432:5432"]
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Ruby
        uses: ruby/setup-ruby@v1
        with:
          ruby-version: 3.0.0

      - name: Ruby gem cache
        uses: actions/cache@v2
        with:
          path: vendor/bundle
          key: ${{ runner.os }}-gems-${{ hashFiles('**/Gemfile.lock') }}
          restore-keys: |
            ${{ runner.os }}-gems-
      - name: Install gems
        run: |
          bundle install --jobs 4 --retry 3

      - name: Analyze with RuboCop
        run: bundle exec rubocop

      - name: Analyze with bundler audit
        run: |
          gem install bundler-audit
          bundle-audit check --update

      - name: Setup environment variables
        run: cp .env.test.template .env.test

      - name: Setup test database
        env:
          RACK_ENV: test
        run: rake db:migrate

      - name: Run tests
        env:
          RACK_ENV: test
        run: bundle exec rspec
```

Let's go over what's going on in our configuration file step by step:

- We will use the latest version of Ubuntu:

```
runs-on: ubuntu-latest
```

- Setup for a PostgreSQL database service:

```
services:
  postgres:
    image: postgres:latest
    env:
      POSTGRES_USER: postgres
      POSTGRES_DB: todo-api-test
      POSTGRES_PASSWORD: postgres
    ports: ["5432:5432"]
    options: >-
      --health-cmd pg_isready
      --health-interval 10s
      --health-timeout 5s
      --health-retries 5
```

- Checks out the latest version of the code:

```
  - name: Checkout code
    uses: actions/checkout@v2
```

- Sets up a base image with Ruby:

```
  - name: Set up Ruby
    uses: ruby/setup-ruby@v1
    with:
      ruby-version: 3.0.0
```

- Install dependencies with `bundler` and cache the results to speed up builds when they don't change:

```
- name: Ruby gem cache
  uses: actions/cache@v2
  with:
    path: vendor/bundle
    key: ${{ runner.os }}-gems-${{ hashFiles('**/Gemfile.lock') }}
    restore-keys: |
      ${{ runner.os }}-gems-
- name: Install gems
  run: |
    bundle install --jobs 4 --retry 3
```

- Run linters/checkers (rubocop, bundler-audit)

```
- name: Analyze with RuboCop
  run: bundle exec rubocop

- name: Analyze with bundler audit
  run: |
    gem install bundler-audit
    bundle-audit check --update
```

- Setup environment variables:

```
- name: Setup environment variables
  run: cp .env.test.template .env.test
```

- Setup test database and run the tests:

```
- name: Setup test database
  env:
    RACK_ENV: test
  run: rake db:migrate

- name: Run tests
  env:
    RACK_ENV: test
  run: bundle exec rspec
```

That’s it. Github Actions will launch our workflow every time we push a new commit to our repository.

2.6. README section

As a culmination of our work, let’s add a README section to our application.

A README is a text file that introduces and explains a project. It contains information that is commonly required to understand what the project is.

README.md

~~~
# todo-api

JSON API for todo project built with Roda + Sequel.

### Clone the repository

```shell
git clone git@github.com:maturbanski/todo-api.git
cd todo-api
```

### Check your Ruby version

```shell
ruby -v
```

The ouput should start with something like `ruby 3.0.0`

If not, install the right ruby version.

### Install dependencies

Using [Bundler](https://github.com/bundler/bundler):

```shell
bundle install
```

### Setup Database

This project uses PostgreSQL by default, to setup database for development and test environment use following instructions:

1. Create `.env.development` for development environment.
2. Add `DATABASE_URL=postgres:///todo-api-development` and `createdb todo-api-development` locally.
3. Create `.env.test` for test environment.
4. Add `DATABASE_URL=postgres:///todo-api-test` and `createdb todo-api-test` locally.

### Migrate database
~~~

```markdown
1. To migrate database in development environment use: `rake db:migrate`
2. To migrate database in test environment use: `RACK_ENV=test rake db:migrate`
3. To migrate database in production environment use: `RACK_ENV=production rake
db:migrate`

### Running the app

You can start your application using `rackup` command.

### Generating documentation

You can generate documentation using `rake docs` command.

### Launching tests

You can launch tests using `rspec` command.

### Seed database

To populate data with seeds use `rake db:seed` command.

### Check code style

To check code style using `rubocop` command.
```

This is how we completed the configuration of our project. In the next chapter we will create basic Roda application.

Chapter 3. Getting started with Roda

3.1. Basic Roda Application

We finished configuring our project so we can finally create a basic Roda application. We need to create `app.rb` file in our project folder:

app.rb

```ruby
# frozen_string_literal: true

require 'roda'

require_relative './system/boot'

# The main class for Roda Application.
class App < Roda
  # Adds support for handling different execution environments
(development/test/production).
  plugin :environments

  # Adds support for heartbeats.
  plugin :heartbeat

  configure :development, :production do
    # A powerful logger for Roda with a few tricks up it's sleeve.
    plugin :enhanced_logger
  end

  # The symbol_matchers plugin allows you do define custom regexps to use for specific
symbols.
  plugin :symbol_matchers

  # Adds ability to automatically handle errors raised by the application.
  plugin :error_handler

  # Allows modifying the default headers for responses.
  plugin :default_headers,
         'Content-Type' => 'application/json',
         'Strict-Transport-Security' => 'max-age=16070400;',
         'X-Frame-Options' => 'deny',
         'X-Content-Type-Options' => 'nosniff',
         'X-XSS-Protection' => '1; mode=block'

  # Adds request routing methods for all http verbs.
  plugin :all_verbs
end
```

First, we `require 'roda'` to access the `Roda` class that our `App` class will inherit from, then we `require_relative './system/boot'` to have access to all of our dependencies and classes.

Next, we create an `App` class that inherits from the `Roda` class. By inheriting from the `Roda` class, our `App` class is implicitly a Rack application.

First plugin in our `Roda` class is `plugin :environments`. With core Roda, the typical way to check or set which environment the application is operating in is to operate on at `ENV["RACK_ENV"]`. However, some people may want a more straightforward way to deal with environments, and for that, `Roda` offers an environments plugin. The `environments` plugin offers an `environment` method to return the environment as a symbol and a setter method to modify the environment. It also offers `development?`, `test?` and `production?` methods for checking for the most common environments. Finally, it provides a `configure` method, which can be called with any environment symbols, which will yield to the block if the application is running in one of those environments.

`plugin :heartbeat` handles `heartbeat` requests. If a request for the heartbeat path comes in, a 200 response with a text/plain Content-Type and a body of `OK` will be returned.

Another essential plugin is `plugin :enhanced_logger`, which is powerful logger dedicated for Roda that will make logs much more readable. Here we are using the `environments` configure method to enable that plugin in the `development` and `production` environment.

`plugin :symbol_matchers` allows different symbols to match different segments. Let’s say we have many routes that accept an id, and our application only allows ids that are compatible with the `UUID` format. We can use a custom symbol matcher that we will be sure will only match if the id format is valid.

`plugin :error_handler` adds an error handler to the routing so when routing the request raises an error, a nice error message page can be returned to the user.

`plugin :default_headers` change the response headers that are added by default. By default, the only header added is the `Content-Type` header, which is set to `text/html`. However, there are some security features we can enable in browsers by specifying headers. So if we are writing an application that browsers will use (as opposed to an API), we may want to specify these headers.

To keep Roda small, only the methods that browsers support are included by default. However, `Roda` ships with an `plugin :all_verbs` plugin that adds other request methods such as `r.head`, `r.put`, `r.patch`, and `r.delete` for handling the other HTTP methods.

Every Ruby web application framework is built on top of a universal compatibility layer called Rack. Roda is Rack-compatible, so we start by creating a `rackup file`, using the standard file name config.ru.

config.ru

```
# frozen_string_literal: true

# This file contains configuration to let the webserver which application to run.

require_relative 'app'

run App.freeze.app
```

Then from the command line we run the `rackup` command to start up the web server and start serving requests.

If we now load http://localhost:9292/heartbeat in our browser, we see that our application is working.

3.2. Setup Test Framework

Now when we have our first endpoint in the application, we can start thinking about writing automated tests. Our test framework of choice is RSpec. RSpec is a testing tool for Ruby, created for behavior-driven development (BDD). It is the most frequently used testing library for Ruby in production applications. Even though it has a very rich and powerful DSL (domain-specific language), at its core, it is a simple tool that you can start using rather quickly.

Let’s start our RSpec configuration by creating `spec_helper.rb` file in `/spec` folder:

/spec/spec_helper.rb

```ruby
# frozen_string_literal: true

# Set RACK_ENV to test.
ENV['RACK_ENV'] = 'test'

require_relative '../app'

# Require all files in spec/support folder.
root_path = Pathname.new(File.expand_path('..', __dir__))
Dir[root_path.join('spec/support/**/*.rb')].each { |f| require f }
```

First we set our `RACK_ENV` environment variable to `test`:

```ruby
ENV['RACK_ENV'] = 'test'
```

We need to have access to our `App` class so we need to need to require our `app.rb` file:

```ruby
require_relative '../app'.
```

`/spec/support` folder will contain helper methods and modules that we will use during our tests. We need to load all files from this folder before launching tests:

```ruby
# Require all files in spec/support folder.
root_path = Pathname.new(File.expand_path('..', __dir__))
Dir[root_path.join('spec/support/**/*.rb')].each { |f| require f }
```

Ok, now let’s configure `rack_test`, a tool for testing Rack apps:

/spec/support/rack_test.rb

```
# frozen_string_literal: true

# This file contains configuration for rack_test gem.

require 'rack/test'

RSpec.configure do |config|
  config.include Rack::Test::Methods, type: :request

  def app
    App.freeze.app
  end
end
```

We require the `rack/test` gem and include the `Rack::Test::Methods` module for `request` type tests, then we define the `app` method, which is required to launch our app during tests.

Next we need to configure FactoryBot, a library for setting up Ruby objects as test data:

/spec/support/factory_bot.rb

```
# frozen_string_literal: true

# This file contains configuration for FactoryBot gem.

require 'factory_bot'

# Load all factories defined in spec/factories folder.
FactoryBot.find_definitions

# By default, creating a record will call save! on the instance; since this may not
always be ideal,
# you can override that behavior by defining to_create on the factory:
FactoryBot.define do
  to_create(&:save)
end

# Allow factory associations to follow the parent's build strategy.
# Previously, all factory associations were created, regardless of whether the parent
was persisted to the database.
FactoryBot.use_parent_strategy = false

RSpec.configure do |config|
  config.include FactoryBot::Syntax::Methods
end
```

`FactoryBot` by default does not work with `Sequel`. First, we need to update `FactoryBot` to call `save` instead of `save!` during creation, because `Sequel` models do not implement the `save!` method. Next, we disable the `use_parent_strategy` option to created records with nested associations.

We also need to configure our database cleaning strategy:

/spec/support/database_cleaning.rb

```ruby
# frozen_string_literal: true

# This file contains configuration for database cleaning strategy.

RSpec.configure do |config|
  config.around do |example|
    Application['database'].transaction(rollback: :always, auto_savepoint: true) {
example.run }
  end
end
```

It's generally best to run each test in its transaction if possible. That keeps all tests isolated from each other, and it's simple as it handles all of the cleanups for you.

Our final step is to create `ApiHelpers` module that will contain helper methods used during request testing:

/spec/support/api_helpers.rb

```ruby
# frozen_string_literal: true

# {ApiHelpers} module contains helper methods that are used in the API request specs.
module ApiHelpers
  # It returns the response that our request has returned.
  def response
    last_response
  end

  # It parse the response JSON document into a Ruby data structure and return it.
  def json_response
    JSON.parse(response.body)
  end
end

RSpec.configure do |config|
  config.include ApiHelpers
end
```

Here we have two methods, `response` that returns the response body of our request and `json_response` that returns parsed JSON response.

That's it. We finished our RSpec configuration. We are ready to write our first test. Let's test that our `/hearbeart` endpoint works as expected:

/spec/requests/heartbeat_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe 'GET /heartbeat', type: :request do
  before { get('/heartbeat') }

  it 'returns 200 HTTP status' do
    expect(response.status).to eq 200
  end

  it 'returns OK in the response body' do
    expect(response.body).to eq 'OK'
  end
end
```

In this test, we are checking that calling `GET /heartbeat`:

- should return us `200 HTTP status`.
- should return `OK` in the response body.

Before running or tests we need to create `lib` and `app` folders:

```
mkdir lib app
```

Otherwise we will see `Dry::System::ComponentsDirMissing:` error.

Now we are ready to run our tests using `rspec` command:

```
$ rspec
..

Finished in 0.01698 seconds (files took 1.79 seconds to load)
2 examples, 0 failures
```

Excellent! Our tests are passing, so we configured RSpec correctly.

3.3. User model

Our API should support user accounts, with each user having the ability to manage their todos. Let’s start by creating a migration that will create `users` table in our database:

/db/migrate/002_create_users_table.rb

```
# frozen_string_literal: true

Sequel.migration do
  change do
    create_table(:users) do
      column :id,                   :uuid,    null: false, default: Sequel
.function(:uuid_generate_v4), primary_key: true
      column :email,                'citext', null: false, unique: true
      column :password_digest,      String,   null: false
      column :authentication_token, String,   null: false, unique: true
      column :created_at,           DateTime, null: false, default: Sequel
::CURRENT_TIMESTAMP
      column :updated_at,           DateTime, null: false, default: Sequel
::CURRENT_TIMESTAMP
    end
  end
end
```

A few things are going on here:

We want to use `UUID` primary keys, and because of that, we use the Sequel function for generating those: `Sequel.function(:uuid_generate_v4)`.

For the email column, we use `citext` column type. The `citext` module provides a case-insensitive character string type, `citext`. Essentially, it internally calls lower when comparing values. Otherwise, it behaves almost exactly like text. We also want email to be unique, `unique: true` will create a `unique` constraint in the database.

The `password_digest` column will be responsible for storing user hashed password.

The `authentication_token` is unique token among all users that is used during authorization token verification. This will be necessary to invalidate existing authorization token when user logout or wants to refresh their token.

`created_at` and `updated_at` are timestamps columns that value will be filled automatically by the PostgreSQL `CURRENT_TIMESTAMP()`, which returns the current date and time with time zone, which is the time when the transaction starts.

Let’s run our migration:

```
rake db:migrate && RACK_ENV=test rake db:migrate
```

Now we are ready to create our `User` model:

/app/models/user.rb

```
# frozen_string_literal: true

# {User} model stores authentication informations.
#
# @!attribute id
#   @return [UUID] ID of the {User} in UUID format.
#
# @!attribute email
#   @return [String] Email of the {User}, it's stored in the PostgreSQL citext column.
#
# @!attribute password_digest
#   @return [String] {User} hashed password with bcrypt.
#
# @!attribute authentication_token
#   @return [String] Unique token among all users({User}) used during authorization.
#
# @!attribute created_at
#   @return [DateTime] Time when {User} was created.
#
# @!attribute updated_at
#   @return [DateTime] Time when {User} was updated.
class User < Sequel::Model
  # Plugin that adds BCrypt authentication and password hashing to Sequel models.
  plugin :secure_password

  # It validates {User} object.
  #
  # @example Validate {User}:
  #   User.new.validate
  def validate
    super

    validates_format(Constants::EMAIL_REGEX, :email) if email
  end
end
```

First we enable `plugin :secure_password` which is provided by sequel_secure_password gem. This plugin adds BCrypt authentication and password hashing to Sequel models.

We are also adding additional format validation for user email. Our email `EMAIL_REGEX` constant is stored in the `Constants` module that is responsible for storing constants that are used across the application:

/lib/constants.rb

```ruby
# frozen_string_literal: true

# {Constants} module is responsible for storing constants that are used across the
application.
module Constants
  # Regex that is used during email validation process.
  EMAIL_REGEX = /^[^,;@ \r\n]+@[^,@; \r\n]+\.[^,@; \r\n]+$/

  public_constant :EMAIL_REGEX

  # List of valid values for sort directions.
  SORT_DIRECTIONS = %w[desc asc].freeze

  public_constant :SORT_DIRECTIONS

  # List of columns that can be used for sorting to-dos ({Todo}).
  TODO_SORT_COLUMNS = %w[name description created_at updated_at].freeze

  public_constant :TODO_SORT_COLUMNS

  # Regex that is used during UUID validation process.
  UUID_REGEX = /(\h{8}(?:-\h{4}){3}-\h{12})/

  public_constant :UUID_REGEX
end
```

Except `EMAIL_REGEX`, our `Constants` module also implements other constants that we will use later.

Let's write tests for our `Constants` module:

/spec/lib/constants_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Constants do
  it 'defines EMAIL_REGEX constant' do
    expect(described_class::EMAIL_REGEX).to eq(/^[^,;@ \r\n]+@[^,@; \r\n]+\.[^,@; \r\n]+$/)
  end

  it 'defines SORT_DIRECTIONS constant' do
    expect(described_class::SORT_DIRECTIONS).to eq %w[desc asc]
  end

  it 'defines TODO_SORT_COLUMNS constant' do
    expect(described_class::TODO_SORT_COLUMNS).to eq %w[name description created_at updated_at]
  end

  it 'defines UUID_REGEX constant' do
    expect(described_class::UUID_REGEX).to eq(/(\h{8}(?:-\h{4}){3}-\h{12})/)
  end
end
```

Before testing our `User` model we need to define first our factory:

/spec/factories/users.rb

```
# frozen_string_literal: true

FactoryBot.define do
  factory :user do
    sequence(:email)      { |n| "test-#{n}@user.com" }
    password              { 'password'             }
    password_confirmation { 'password'             }
    authentication_token  { SecureRandom.hex(40)    }
  end
end
```

Now we are ready to write tests for `User` model:

/spec/models/user_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe User, type: :model do
  describe 'email presence validation' do
    let(:user) { build(:user, email: email) }

    before { user.valid? }

    context 'when is blank' do
      let(:email) { nil }

      it 'adds error to the :email field' do
        expect(user.errors[:email]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:email) { 'test@user.com' }

      it 'does not add error to the :email field' do
        expect(user.errors[:email]).to eq nil
      end
    end
  end

  describe 'email format validation' do
    let(:user) { build(:user, email: email) }

    before { user.valid? }

    context 'when is invalid' do
      let(:email) { 'invalid-email' }

      it 'adds error to the :email field' do
        expect(user.errors[:email]).to eq ['is invalid']
      end
    end

    context 'when is valid' do
      let(:email) { 'test@user.com' }

      it 'does not add error to the :email field' do
        expect(user.errors[:email]).to eq nil
      end
    end
  end

  describe 'email uniqueness validation' do
    let(:user) { build(:user, email: email) }

    before do
      create(:user, email: 'test@user.com')

      user.valid?
    end

    context 'when user email is unique' do
      let(:email) { 'test_2@user.com' }

      it 'does not add error to the :email field' do
        expect(user.errors[:email]).to eq nil
      end
    end

    context 'when user email is not unique' do
      let(:email) { 'test@user.com' }

      it 'adds error to the :email field' do
        expect(user.errors[:email]).to eq ['is already taken']
      end
    end
  end

  describe 'password validation' do
    let(:user) { build(:user, password: password, password_confirmation:
password_confirmation) }

    before { user.valid? }

    context 'when is blank' do
      let(:password)              { nil }
      let(:password_confirmation) { nil }

      it 'adds errors to the :password field' do
        expect(user.errors[:password]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:password)              { 'password' }
      let(:password_confirmation) { 'password' }

      it 'does not add error to the :password field' do
        expect(user.errors[:password]).to eq nil
      end
    end

    context 'when password does not match confirmation' do
      let(:password)              { 'password' }
      let(:password_confirmation) { 'test' }

      it 'adds error to the :password field' do
        expect(user.errors[:password]).to eq ["doesn't match confirmation"]
      end
    end
  end

  describe 'authentication_token presence validation' do
    let(:user) { build(:user, authentication_token: authentication_token) }

    before { user.valid? }

    context 'when is blank' do
      let(:authentication_token) { nil }

      it 'adds error to the :authentication_token field' do
        expect(user.errors[:authentication_token]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:authentication_token) { 'test' }

      it 'does not add error to the :authentication_token field' do
        expect(user.errors[:authentication_token]).to eq nil
      end
    end
  end

  describe 'authentication_token uniqueness validation' do
    let(:user) { build(:user, authentication_token: authentication_token) }

    before do
      create(:user, authentication_token: 'test')

      user.valid?
    end

    context 'when user authentication_token is unique' do
      let(:authentication_token) { 'test_2' }

      it 'does not add error to the :authentication_token field' do
        expect(user.errors[:authentication_token]).to eq nil
      end
    end

    context 'when user authentication_token is not unique' do
      let(:authentication_token) { 'test' }

      it 'adds error to the :authentication_token field' do
        expect(user.errors[:authentication_token]).to eq ['is already taken']
      end
    end
  end

  describe '#password=' do
```

```
    let(:user) { build(:user, password: 'test') }

    it 'sets password_digest column with hashed user password' do
      expect(user.password_digest).not_to be_blank
    end
  end

  describe '#authenticate' do
    let(:user) { create(:user) }

    context 'when password is not valid' do
      it 'returns nil' do
        expect(user.authenticate('test')).to eq nil
      end
    end

    context 'when password is valid' do
      it 'returns User object' do
        expect(user.authenticate('password')).to eq user
      end
    end
  end
end
```

A lot is going on here, in our `User` model tests we are testing following things:

- `email` presence validation.
- `email` format validation.
- `email` uniqueness validation.
- `password` validation.
- `authentication_token` presence validation.
- `authentication_token` uniqueness validation.
- saving hashed user `password` with `password=` method.
- checking user password with `authenticate` method.

3.4. Todo model

Let’s create our Todo model that will store user todo informations. Like always, we will start by creating a migration:

/db/migrate/003_create_todos_table.rb

```
# frozen_string_literal: true

Sequel.migration do
  change do
    create_table(:todos) do
      column :id,          :uuid, null: false, default: Sequel
.function(:uuid_generate_v4), primary_key: true
      column :name,        String, null: false
      column :description, String, null: false
      column :created_at,  DateTime, null: false, default: Sequel::CURRENT_TIMESTAMP
      column :updated_at,  DateTime, null: false, default: Sequel::CURRENT_TIMESTAMP

      foreign_key :user_id, :users, type: 'uuid', null: false, on_delete: :cascade
    end
  end
end
```

foreign_key is used to create a foreign key column that references a column in another table. It takes the column name as the first argument, the table it references as the second argument, and an options hash as its third argument.

:on_delete Specify the behavior of this foreign key column when the row with the primary key it references is deleted. We are using cascade option here, which specifies that when a referenced row is deleted, rows referencing it should be automatically deleted as well.

Let’s run our migration:

```
rake db:migrate && RACK_ENV=test rake db:migrate
```

Now let’s add our model:

/app/models/todo.rb

```
# frozen_string_literal: true

# {Todo} model stores information about {User} todos.
#
# @!attribute id
#   @return [UUID] ID of the {Todo} in UUID format.
#
# @!attribute name
#   @return [String] Name of the {Todo}.
```

```ruby
#
# @!attribute description
#   @return [String] Description of the {Todo}.
#
# @!attribute user_id
#   @return [UUID] ID of the {User} which {Todo} belongs to in UUID format.
#
# @!attribute created_at
#   @return [DateTime] Time when {Todo} was created.
#
# @!attribute updated_at
#   @return [DateTime] Time when {Todo} was updated.
class Todo < Sequel::Model
  many_to_one :user

  dataset_module do
    # It filters todos by their name.
    #
    # @param [String] name of the {Todo} or its part.
    #
    # @return [Array<Todo>] Array of {Todo} objects.
    #
    # @example Search Todo by name:
    #   Todo.search_by_name('milk')
    def search_by_name(name)
      where(Sequel.ilike(:name, "%#{name}%"))
    end

    # It filters todos by their description.
    #
    # @param [String] description of the {Todo} or its part.
    #
    # @return [Array<Todo>] Array of {Todo} objects.
    #
    # @example Search Todo by description:
    #   Todo.search_by_description('buy milk')
    def search_by_description(description)
      where(Sequel.ilike(:description, "%#{description}%"))
    end
  end
end
```

`many_to_one` association is used when the table for the current class contains a foreign key that references the primary key in the table for the associated class. It is named because there can be many rows in the current table for each row in the associated table. We have `many_to_one :user` because `User` will have many `todo`.

We are also using here `dataset_module` to create named dataset methods for `Todo` dataset. We want to filter todos by name and description using `ILIKE` operator.

Before writing tests for our new `Todo` model, we need to setup factory:

/spec/factories/todos.rb

```
# frozen_string_literal: true

FactoryBot.define do
  factory :todo do
    name        { 'Buy milk'                }
    description { 'Remember to buy milk.' }

    user
  end
end
```

Now we are ready to write model tests:

/spec/factories/todos.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Todo, type: :model do
  describe 'name presence validation' do
    let(:user) { build(:todo, name: name) }

    before { user.valid? }

    context 'when is blank' do
      let(:name) { nil }

      it 'adds error to the :name field' do
        expect(user.errors[:name]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:name) { 'Buy milk.' }

      it 'does not add error to the :name field' do
        expect(user.errors[:name]).to eq nil
      end
    end
  end

  describe 'description presence validation' do
    let(:user) { build(:todo, description: description) }

    before { user.valid? }
    context 'when is blank' do
      let(:description) { nil }

      it 'adds error to the :description field' do
        expect(user.errors[:description]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:description) { 'Remember to but milk.' }

      it 'does not add error to the :description field' do
        expect(user.errors[:description]).to eq nil
      end
    end
  end

  describe 'user_id presence validation' do
    let(:todo) { build(:todo, user: user) }

    before { todo.valid? }

    context 'when is blank' do
      let(:user) { nil }

      it 'adds error to the :user_id field' do
        expect(todo.errors[:user_id]).to eq ['is not present']
      end
    end

    context 'when is not blank' do
      let(:user) { create(:user) }

      it 'does not add error to the :user_id field' do
        expect(todo.errors[:user_id]).to eq nil
      end
    end
  end

  describe '.search_by_name' do
    let!(:todo) { create(:todo, name: 'Buy milk.') }

    before { create(:todo, name: 'Buy cheese.') }

    it 'filters todos by their name' do
      expect(described_class.search_by_name('milk').all).to eq [todo]
    end
  end

  describe '.search_by_description' do
    let!(:todo) { create(:todo, description: 'Remember to buy milk.') }
    before { create(:todo, description: 'Remember to buy cheese.') }

    it 'filters todos by their description' do
      expect(described_class.search_by_description('milk').all).to eq [todo]
    end
  end
end
```

In the `Todo` model tests we are testing following things:

- `name` presence validation.
- `description` presence validation.
- `user_id` presence validation.
- filtering by `name`.
- filtering by `description`.

The last thing we need to do is to add to the connection to our new `Todo` model from the `User` model:

/app/models/user.rb

```
one_to_many :todos
```

The `one_to_many` association is used when the table for the associated class contains a foreign key that references the primary key in the table for the current class. It is named because for each row in the current table, there can be many rows in the associated table

That's it. We've finished our models setup.

Chapter 4. Securing API

4.1. Token-based Authentication

Token-based authentication is the way of handling the authentication of users in applications. It is an alternative to session-based authentication. The most notable difference between session-based and token-based authentication is that session-based authentication relies heavily on the server. A record is created for each logged-in user.

Token-based authentication is stateless - it does not store anything on the server but creates a unique encoded token that gets checked every time a request is made.

Unlike session-based authentication, a token approach would not associate a user with login information but with a unique token that is used to carry client-host transactions.

The way token-based authentication works is simple. The user enters credentials and sends a request to the server. If the credentials are correct, the server creates a unique encoded token. The client stores the newly created token and makes all subsequent requests to the server with the token attached in the `Authorization` header.

The server authenticates the user by decoding the token sent in the `Authorization` header with the request and finding the user that the token belongs.

4.2. Tokens generation

For the token generation we will gonna use ActiveSupport::MessageVerifier. It is helpful for cases like remember-me tokens and subscribes links or where the session store isn't available. Let's see an example:

```
secret         = '0eb411d07ac481e53b58a782e7d6c59fc51ad634'
verifier       = ActiveSupport::MessageVerifier.new(secret)
message        = {id: 42}
signed_message = verifier.generate(message) # => "BAh7BjoHaWRpLw==--
156da236d3c30c49a56373fa0b30552b581845f5"

verifier.verify(signed_message) # => {:id=>42}
```

Our data was converted to a long string and back again. If we try to verify invalid signature, `ActiveSupport::MessageVerifier::InvalidSignature` error will be raised:

```
verifier.verify('invalid-signature') # =>
ActiveSupport::MessageVerifier::InvalidSignature
```

By default, any message can be used throughout app, but we can also associate our message with a specific purpose:

```
token = verifier.generate('secret-token', purpose: :login)

verifier.verify(token, purpose: :login)  # => 'secret-token'
verifier.verify(token, purpose: :logout) # =>
ActiveSupport::MessageVerifier::InvalidSignature
```

By default, messages last forever, but messages can be set to expire at a given time `expires_at`:

```
verifier.generate(message, expires_at: Time.now + 360)
```

We know how `ActiveSupport::MessageVerifier` works, so we are ready to create a `MessageVerifier` class that will be responsible for generating access and refresh tokens:

/lib/message_verifier.rb

```
# frozen_string_literal: true

# {MessageVerifier} makes it easy to generate and verify messages which are signed to
prevent tampering.
# @see https://api.rubyonrails.org/v6.0.3.4/classes/ActiveSupport/MessageVerifier.html
ActiveSupport::MessageVerifier Documentation
class MessageVerifier
  class << self
```

```
    # It encode data using ActiveSupport::MessageVerifier.
    #
    # @param [Hash, Array, String] data that will be encoded.
    # @param [Integer] expires_at in seconds telling when the message expires.
    # @param [Symbol] purpose that describes how the message will be used.
    #
    # @return [String] signed message for the provided value.
    #
    # @example Encode a message:
    #   MessageVerifier.encode(data: 'secret', expires_at: Time.now + 360, purpose: :test)
    def encode(data:, expires_at:, purpose:)
      verifier.generate(data, expires_at: expires_at, purpose: purpose)
    end

    # It decodes data using ActiveSupport::MessageVerifier
    #
    # @param [String] message that will be used in decoding process.
    # @param [Symbol] purpose that describes how the message will be used.
    #
    # @return [Hash, Array, String] data that has been decoded.
    #
    # @raise [ActiveSupport::MessageVerifier::InvalidSignature] when message is invalid.
    #
    # @example Decode a valid message:
    #   message = "eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaEpJZ3R6WldOeVpYU"
    #   MessageVerifier.decode(message: message, purpose: :test)
    #
    # @example Decode invalid message:
    #   MessageVerifier.decode(message: 'invalid-message', purpose: :test)
    def decode(message:, purpose:)
      verifier.verify(message, purpose: purpose)
    end

    private

    # It returns instance of ActiveSupport::MessageVerifier.
    #
    # @return [ActiveSupport::MessageVerifier] instance of ActiveSupport::MessageVerifier.
    def verifier
      ActiveSupport::MessageVerifier.new(ENV['SECRET_KEY_BASE'], digest: 'SHA512')
    end
  end
end
```

For those unfamiliar, a singleton class restricts the instantiation of a class to a single object, which comes in handy when only one object is needed to complete the tasks at hand.

First of all, we need to have access to the instance of the `ActiveSupport::MessageVerifier` class. This will be done by the private `verifier` method. We initialize `ActiveSupport::MessageVerifier` with a unique secret that will be used to sign a message and hashing algorithm type `SHA512`:

```
  def verifier
    ActiveSupport::MessageVerifier.new(ENV['SECRET_KEY_BASE'], digest: 'SHA512')
  end
```

`encode` method accepts three parameters:

- `data` that we want to encode.
- `expires_at` telling when the message expires.
- `purpose` that describes how the message will be used.

The result of encoding is our signed message.

`decode` method accepts two parameters:

- `message` that we want to decode.
- `purpose` that describes how the message will be used.

The result of decoding is our data object.

Before we start using our new class, we need to add the `SECRET_KEY_BASE` environment variable. To generate those, we will use the `SecureRandom.hex` method that generate random hexadecimal strings:

```
SecureRandom.hex(40) =>
"8e4b385809c9d81cd28c4734ccf653b789f059224794691714c4bb0e3962d54a8932feac566e4ca2"
```

Let's add unique `SECRET_KEY_BASE` for our `.env` and `.env.template` files:

.env.development

```
SECRET_KEY_BASE=75b92df8aa89df8ae8a24fca30a057a11931c316d3854e725425089ff84cbf334fb607
4d662ea721
```

.env.development.template

```
SECRET_KEY_BASE=8e4b385809c9d81cd28c4734ccf653b789f059224794691714c4bb0e3962d54a8932fe
ac566e4ca2
```

.env.test

```
SECRET_KEY_BASE=47e024bd7d5f47c2a10b0404a00c1c89c4f4744c146381a81035a245945ee6bfc0be05
95a84e58fe
```

.env.test.template

```
SECRET_KEY_BASE=0a8fd14ddcf36cf33348c2514efad5d898470cec05fe8fd08be68b8a8005ab2e7e25d2e205502e59
```

It’s time to write tests for the `MessageVerifier` class:

spec/lib/message_verifier_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe MessageVerifier do
  let(:data)       { 'Test message' }
  let(:purpose)    { :test          }
  let(:expires_at) { Time.now + 60  }

  describe '.encode' do
    it 'encodes data' do
      expect(described_class.encode(data: data, purpose: purpose, expires_at: expires_at)).not_to eq data
    end
  end

  describe '.decode' do
    let(:message) { described_class.encode(data: data, expires_at: expires_at, purpose: purpose) }

    context 'when message is valid' do
      it 'decodes data from message' do
        expect(described_class.decode(message: message, purpose: purpose)).to eq data
      end
    end

    context 'when message is invalid' do
      it 'raise ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.decode(message: 'wrong message', purpose: purpose) }.to raise_error(
          ActiveSupport::MessageVerifier::InvalidSignature
        )
      end
    end

    context 'when message is expired' do
      let(:expires_at) { Time.now - 10 }

      let(:message) do
        described_class.encode(data: data, expires_at: expires_at, purpose: purpose)
      end
      it 'raise ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.decode(message: message, purpose: purpose) }.to
raise_error(
          ActiveSupport::MessageVerifier::InvalidSignature
        )
      end
    end

    context 'when purpose is invalid' do
      it 'raise ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.decode(message: message, purpose: :invalid) }.to
raise_error(
          ActiveSupport::MessageVerifier::InvalidSignature
        )
      end
    end
  end
end
```

Here we are testing that:

- `encode` method returns signed message.
- `decode` method returns decoded data.
- `decode` method raises an error when the message or purpose is invalid.

To simplify the generation of access and refresh tokens for the user, we will create dedicated classes for generating those.

`AccessTokenGenerator` class will generate an access token for the specified user:

/lib/access_token_generator.rb

```
# frozen_string_literal: true

# {AccessTokenGenerator} generates access token for {User}.
class AccessTokenGenerator
  # @param [User] user
  def initialize(user:)
    @user = user
  end

  # It generates access token for {User}.
  #
  # @return [String] access token, which is valid for 5 minutes.
  #
  # @example Generate access token for {User}:
  #   AccessTokenGenerator.new(user: User.last).call
  def call
    data = { user_id: @user.id, authentication_token: @user.authentication_token }

    MessageVerifier.encode(data: data, expires_at: Time.now + 300, purpose:
:access_token)
  end
end
```

Tests for that class are really simple:

spec/lib/access_token_generator_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'
require 'timecop'

describe AccessTokenGenerator do
  describe '#call' do
    let(:user)         { create(:user)  }
    let(:access_token) { 'access_token' }

    let(:data) do
      { user_id: user.id, authentication_token: user.authentication_token }
    end

    before do
      Timecop.freeze

      expect(MessageVerifier)
        .to receive(:encode)
        .with(data: data, expires_at: Time.now + 300, purpose: :access_token)
        .and_return(access_token)
    end

    after do
      Timecop.return
    end

    it 'returns access token for specified user' do
      expect(described_class.new(user: user).call).to eq access_token
    end
  end
end
```

We are mocking here `MessageVerifier` class to check that it receives correct parameters. We already have unit tests for `MessageVerifier`, so we don't need to do that here.

`RefreshTokenGenerator` is similar to `AccessTokenGenerator` and will generate a refresh token for the specified user:

/lib/refresh_token_generator.rb

```ruby
# frozen_string_literal: true

# {RefreshTokenGenerator} generates refresh token for {User}.
class RefreshTokenGenerator
  # @param [User] user
  def initialize(user:)
    @user = user
  end

  # It generates refresh token for {User}.
  #
  # @return [String] refresh token, which is valid for 15 minutes.
  #
  # @example Generate refresh token for {User}:
  #   Users::RefreshTokenGenerator.new(user: User.last).call
  def call
    data = { user_id: @user.id, authentication_token: @user.authentication_token }

    MessageVerifier.encode(data: data, expires_at: Time.now + 900, purpose:
:refresh_token)
  end
end
```

Here we also mock the `MessageVerifier` class to verify if the `encode` method was called with valid parameters:

spec/lib/refresh_token_generator_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'
require 'timecop'

describe RefreshTokenGenerator do
  describe '#call' do
    let(:user)          { create(:user)   }
    let(:refresh_token) { 'refresh_token' }

    let(:data) do
      { user_id: user.id, authentication_token: user.authentication_token }
    end

    before do
      Timecop.freeze

      expect(MessageVerifier)
        .to receive(:encode)
        .with(data: data, expires_at: Time.now + 900, purpose: :refresh_token)
        .and_return(refresh_token)
    end

    after do
      Timecop.return
    end

    it 'returns refresh token for specified user' do
      expect(described_class.new(user: user).call).to eq refresh_token
    end
  end
end
```

In our tests we will often need to create access and refresh token, because of that let's add helper methods to our `ApiHelpers` module:

spec/support/api_helpers.rb

```
# It generates access token for {User}.
#
# @see AccessTokenGenerator
def access_token(user)
  AccessTokenGenerator.new(user: user).call
end

# It generates refresh token for {User}.
#
# @see RefreshTokenGenerator
def refresh_token(user)
  RefreshTokenGenerator.new(user: user).call
end
```

The last thing we are gonna do here is to create a `AuthorizationTokensGenerator` class that will generate at once access and refresh token for the user:

lib/authorization_tokens_generator.rb

```ruby
# frozen_string_literal: true

# {AuthorizationTokensGenerator} generates access and refresh token for {User}.
class AuthorizationTokensGenerator
  # @param [User] user for whom access and refresh token will be generated.
  def initialize(user:)
    @user = user
  end

  # It generates access and refresh token for specified {User}.
  #
  # @return [Hash] that contains informations about access and refresh token for
{User}.
  #
  # @example Generate access and refresh token for {User}:
  #   AuthorizationTokensGenerator.new(user: User.last).call
  def call
    {
      access_token: { token: access_token, expires_in: 300 },
      refresh_token: { token: refresh_token, expires_in: 900 }
    }
  end

  private

  # It generates access token for specified {User}.
  #
  # @see AccessTokenGenerator
  #
  # @return [String] Access token for specified {User}.
  def access_token
    AccessTokenGenerator.new(user: @user).call
  end

  # It generates refresh token for specified {User}.
  #
  # @see RefreshTokenGenerator
  #
  # @return [String] Refresh token for specified {User}.
  def refresh_token
    RefreshTokenGenerator.new(user: @user).call
  end
end
```

Tests for the `AuthorizationTokensGenerator` looks like this:

spec/lib/authorization_tokens_generator_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe AuthorizationTokensGenerator do
  describe '#call' do
    let(:user)                    { create(:user)                          }
    let(:access_token_generator)  { instance_double(AccessTokenGenerator)  }
    let(:refresh_token_generator) { instance_double(RefreshTokenGenerator) }

    let(:data) do
      { user_id: user.id, authentication_token: user.authentication_token }
    end

    let(:tokens) do
      {
        access_token: { expires_in: 300, token: 'access_token' },
        refresh_token: { expires_in: 900, token: 'refresh_token' }
      }
    end

    before do
      expect(AccessTokenGenerator)
        .to receive(:new)
        .with(user: user)
        .and_return(access_token_generator)

      expect(access_token_generator)
        .to receive(:call)
        .and_return('access_token')

      expect(RefreshTokenGenerator)
        .to receive(:new)
        .with(user: user)
        .and_return(refresh_token_generator)

      expect(refresh_token_generator)
        .to receive(:call)
        .and_return('refresh_token')
    end

    it 'returns access and refresh token for specified user' do
      expect(described_class.new(user: user).call).to eq tokens
    end
  end
end
```

Here once again we use mocking to check if `AccessTokenGenerator` and `RefreshTokenGenerator` are called with valid parameters. In the next section, we will work on the validation of incoming tokens.

4.3. Tokens validation

Access and refresh token creation is done. Now let’s create a dedicated class that we will use for validating authorization token that will come in the Authorization header.

lib/authorization_token_validator.rb

```ruby
# frozen_string_literal: true

# {AuthorizationTokenValidator} validates authorization token that comes in
Authorization header.
class AuthorizationTokenValidator
  # @param [String] authorization_token that we need to validate.
  # @param [String] purpose of the message.
  def initialize(authorization_token:, purpose:)
    @authorization_token = authorization_token
    @purpose = purpose
  end

  # It validates the authorization token that from the Authorization header.
  #
  # @return [User] when authorization token is valid.
  #
  # @raise [ActiveSupport::MessageVerifier::InvalidSignature] when user
authentication_token is different.
  # @raise [ActiveSupport::MessageVerifier::InvalidSignature] when authorization token
is invalid.
  # @raise [ActiveSupport::MessageVerifier::InvalidSignature] when purpose is invalid.
  #
  # @example When authorization token is valid:
  #   AuthorizationTokenValidator.new(authorization_token: 'valid-authorization-
token', purpose: :authorization).call
  #
  # @example When authorization token is not valid:
  #   AuthorizationTokenValidator.new(authorization_token: 'invalid-authorization-
token', purpose: :authorization).call
  #
  # @example When user authentication_token is invalid:
  #   AuthorizationTokenValidator.new(authorization_token: 'valid-authorization-
token', purpose: :authorization).call
  #
  # @example When purpose is invalid:
  #   Authorizationalidator.new(refresh_token: 'valid-authorization-token', purpose:
:test).call
  def call
    unless current_user && current_user.authentication_token ==
data[:authentication_token]
      raise(ActiveSupport::MessageVerifier::InvalidSignature)
    end

    current_user
  end

  private

  # It returns {User} data decoded from the authorization token.
  #
  # @return [Hash] {User} data decoded from the authorization token.
  #
  # @raise [ActiveSupport::MessageVerifier::InvalidSignature] when authorization token
is invalid.
  def data
    @data ||= MessageVerifier.decode(message: @authorization_token, purpose: @purpose)
  end

  # It returns {User} found by id in decoded data.
  #
  # @return [User] when id in the decoded data is valid.
  # @return [NilClass] when id in the decoded data is not valid.
  def current_user
    @current_user ||= User.find(id: data[:user_id])
  end
end
```

The `AuthorizationTokenValidator` `initialize` method accepts two parameters:

- `authorization_token` that we need to validate.
- `purpose` of the message. In our case, it will be `access_token` or `refresh_token`.

We decode the authorization token using the `MessageVerifier.decode` method:

```
# It returns {User} data decoded from the authorization token.
#
# @return [Hash] {User} data decoded from the authorization token.
#
# @raise [ActiveSupport::MessageVerifier::InvalidSignature] when authorization token
is invalid.
def data
  @data ||= MessageVerifier.decode(message: @authorization_token, purpose: @purpose)
end
```

Next we are trying to find `User` in the database by id that was encoded in the authorization token:

```
# It returns {User} found by id in decoded data.
#
# @return [User] when id in the decoded data is valid.
# @return [NilClass] when id in the decoded data is not valid.
def current_user
  @current_user ||= User.find(id: data[:user_id])
end
```

In the `call` method, we check if `User` is present in the database and if the `authentication_token` that was encoded in the token is the same as user `authentication_token`:

```
# It validates the authorization token that from the Authorization header.
#
# @return [User] when authorization token is valid.
#
# @raise [ActiveSupport::MessageVerifier::InvalidSignature] when user
authentication_token is different.
# @raise [ActiveSupport::MessageVerifier::InvalidSignature] when authorization token
is invalid.
# @raise [ActiveSupport::MessageVerifier::InvalidSignature] when purpose is invalid.
#
# @example When authorization token is valid:
#   AuthorizationTokenValidator.new(authorization_token: 'valid-authorization-token',
purpose: :authorization).call
#
# @example When authorization token is not valid:
#   AuthorizationTokenValidator.new(authorization_token: 'invalid-authorization-
token', purpose: :authorization).call
#
# @example When user authentication_token is invalid:
#   AuthorizationTokenValidator.new(authorization_token: 'valid-authorization-token',
purpose: :authorization).call
#
# @example When purpose is invalid:
#   Authorizationalidator.new(refresh_token: 'valid-authorization-token', purpose:
:test).call
def call
  unless current_user && current_user.authentication_token ==
data[:authentication_token]
    raise(ActiveSupport::MessageVerifier::InvalidSignature)
  end

  current_user
end
```

Let’s write tests to check if the `AuthorizationTokenValidator` class works as expected:

spec/lib/authorization_token_validator_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe AuthorizationTokenValidator do
  describe '#call' do
    let(:user) { create(:user) }

    context 'when authorization_token is invalid' do
      let(:token) { 'invalid-token' }

      it 'raises ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.new(authorization_token: token, purpose:
:access_token).call }
          .to raise_error(an_instance_of(ActiveSupport::MessageVerifier
::InvalidSignature))
      end
    end

    context 'when authorization_token is valid' do
      let(:token) { access_token(user) }

      it 'returns user object' do
        expect(described_class.new(authorization_token: token, purpose:
:access_token).call).to eq user
      end
    end

    context 'when user authentication_token is invalid' do
      let!(:token) { access_token(user) }

      before { user.update(authentication_token: 'test') }

      it 'raises ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.new(authorization_token: token, purpose:
:access_token).call }
          .to raise_error(an_instance_of(ActiveSupport::MessageVerifier
::InvalidSignature))
      end
    end

    context 'when purpose is invalid' do
      let(:token) { access_token(user) }

      it 'raises ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.new(authorization_token: token, purpose:
:invalid).call }
```

```
          .to raise_error(an_instance_of(ActiveSupport::MessageVerifier
::InvalidSignature))
      end
    end

    context 'when user id is invalid' do
      let(:token) { access_token(User.new) }

      it 'raises ActiveSupport::MessageVerifier::InvalidSignature' do
        expect { described_class.new(authorization_token: token, purpose:
:access_token).call }
          .to raise_error(an_instance_of(ActiveSupport::MessageVerifier
::InvalidSignature))
      end
    end
  end
end
```

We test the following scenarios here:

- when `authorization_token` is invalid.
- when `authorization_token` is valid.
- when user `authentication_token` is invalid.
- when message `purpose` is invalid.
- when user id is not present in the database.

With that in place, we've finished the process of securing our API, in the next chapter we will talk about JSON serialization.

Chapter 5. JSON Serialization

Since JavaScript has become the primary language of the web and frontend frameworks are based on JavaScript, JSON serialization has become an essential part of many web apps.

5.1. What is JSON serialization?

JSON (JavaScript Object Notation) is a data format that encodes objects in a string. Such data representation can easily be translated between server and browser but also between server and server. Serialization is a process that transforms an object into that string.

Compared to other serialization alternatives such as XML, YAML, or Binary-serialization, JSON offers the following advantages:

- it’s a human-readable format.
- it’s largely adopted and supported by most of programming languages.
- it’s a language-independent format.
- can be compressed in one line to reduce stream size.
- can represent the most of standard objects.
- seamlessly integrates with JavaScript, which makes JSON the standard for streaming data over AJAX calls.

5.2. Our Own JSON Serializer

In the Ruby ecosystem, there are many libraries that we will help you with JSON serialization. Instead of adding another gem to our application, we will create our serializers from scratch using `oj` gem.

Let’s create our `ApplicationSerializer` that will be a base class for all of ours serializers:

/app/serializers/application_serializer.rb

```ruby
# frozen_string_literal: true

# {ApplicationSerializer} is base class that contains configuration that is
# used across all serializers for rendering JSON using Oj serializer.
# @see http://www.ohler.com/oj/doc/ Oj Documentation.
class ApplicationSerializer
  # It accepts Hash as an argument, sets instance variables based
  # on his keys and returns a new instance of {ApplicationSerializer}.
  #
  # @param [Hash] object that will be used for setting instance variables that will be
available inside serializers.
  def initialize(object)
    object.each_pair do |key, value|
      instance_variable_set(:"@#{key}", value)
    end
  end

  # It generates JSON using Oj serializer dump method.
  #
  # @return [String] which is compliant with the JSON standard.
  def render
    Oj.dump(to_json)
  end
end
```

In the `initialize` method, we use `each_pair` for every `key-value` pair of the `object` and we set the instance variables.

`render` method passes the result of the `to_json` method that will be implemented in `ApplicationSerializer` child classes to `Oj.dump` and return a string which is compliant with the JSON standard.

To test how it works, let’s create the first serializer. It will be responsible for serializing users and their access and refresh tokens information:

/app/serializers/user_serializer.rb

```ruby
# frozen_string_literal: true

# {UserSerializer} is responsible for representing {User}
# and their access and refresh token informations.
#
# @example Represent {User} in the JSON format:
#   user   = User.last
#   tokens = { access_token: 'access_token', refresh_token:'refresh_token' }
#   UserSerializer.new(user: user, tokens: tokens).render
class UserSerializer < ApplicationSerializer
  # It generates Hash object that contains {User} details and its access and refresh
token informations.
  #
  # @return [Hash] object that contains {User} details and its access and refresh
token informations.
  #
  # @example Prepare data before transformation to the JSON format:
  #   user   = User.last
  #   tokens = { access_token: 'access_token', refresh_token:'refresh_token' }
  #   UserSerializer.new(user: user, tokens: tokens).to_json
  def to_json
    {
      user: user,
      tokens: @tokens
    }
  end

  private

  # It returns Hash with {User} attributes
  #
  # @return [Hash] {User} attributes.
  def user
    {
      id: @user.id,
      email: @user.email,
      created_at: @user.created_at,
      updated_at: @user.updated_at
    }
  end
end
```

Serializer classes only need to implement the `to_json` method that prepares data before transformation to the JSON format. That's it. Our serializer is ready to use. Let's use `ruby bin/console` to test this out:

```
user   = User.last
tokens = { access_token: 'access_token', refresh_token:'refresh_token' }

UserSerializer.new(user: user, tokens: tokens).to_json # => {:user=>{:id=>"2f94a9d8-
facf-4738-8d46-0d7c0b0d81cc", :email=>"test@user.com.pl", :created_at=>Thu, 22 Apr
2021 09:58:45 +0000, :updated_at=>Thu, 22 Apr 2021 11:58:45 +0000},
:tokens=>{:access_token=>"access_token", :refresh_token=>"refresh_token"}}

UserSerializer.new(user: user, tokens: tokens).render # =>
"{\"user\":{\"id\":\"2f94a9d8-facf-4738-8d46-
0d7c0b0d81cc\",\"email\":\"asd@o2.pl\",\"created_at\":\"2021-04-
22T09:58:45+00:00\",\"updated_at\":\"2021-04-
22T11:58:45+00:00\"},\"tokens\":{\"access_token\":\"access_token\",\"refresh_token\":\
"refresh_token\"}}"
```

We will not write unit tests for serializers because those will be tested during `request` specs. In the next chapter, we will work on incoming params validation.

Chapter 6. Incoming params validation

Checking the validity of data sent to an API is an important responsibility of any service. Besides being an important security feature, it's also crucial for responding intelligently to consumers who provide invalid input.

6.1. dry-validation to the rescue

dry-validation allows validating data based on predicate logic. It is designed to work with any data input, whether it's a simple hash, an array, or a complex object with deeply nested data.

Here is quick example:

```ruby
class UserContract < Dry::Validation::Contract
  params do
    required(:name).filled(:string)
    required(:email).filled(:string)
    required(:age).value(:integer)
  end

  rule(:email) do
    unless /^[^,;@ \r\n]+@[^,@; \r\n]+\.[^,@; \r\n]+$/.match?(value)
      key.failure('has invalid format')
    end
  end

  rule(:age) do
    key.failure('must be greater than 18') if value <= 18
  end
end
```

```ruby
contract = UserContract.new(email: 'test@user.com', age: '17') # =>
#<Dry::Validation::Result{:email=>"test@user.com", :age=>17} errors={:name=>["is
missing"], :age=>["must be greater than 18"]}>
```

Let's start by creating the `AplicationParams` class that is a base class that will store configuration for our params validation classes:

app/params/application_params.rb

```ruby
# frozen_string_literal: true

# {ApplicationParams} is base class that contains configuration
# that is used across all params validator classes.
class ApplicationParams < Dry::Validation::Contract
  # It checks if passed params are valid base on params validation rules in child
class.
  # If params are invalid {Exceptions::InvalidParamsError} is raised.
  # When params are valid it returns provided params.
  #
  # @param [Hash] params that've beeen passed to the endpoint.
  #
  # @return [Hash] Hash when provided params are valid.
  #
  # @raise [InvalidParamsError] {Exceptions::InvalidParamsError} when provided params
are not valid.
  def permit!(params)
    params = self.class.new.call(params)

    raise(invalid_params_error(params)) if params.errors.any?

    params.to_h
  end

  private

  # It returns {Exceptions::InvalidParamsError} instance.
  #
  # @param [Hash] params with errors.
  #
  # @return [Exceptions::InvalidParamsError]
  def invalid_params_error(params)
    Exceptions::InvalidParamsError.new(params.errors.to_h, I18n.t('invalid_params'))
  end
end
```

In the `permit` method we check if incoming params are valid, if they are fine then we return those params, if not we raise `Exceptions::InvalidParamsError`. This error class is not defined so let's do this right now:

lib/exceptions.rb

```ruby
# frozen_string_literal: true

# {Exceptions} module defines errors classes that are used in application.
module Exceptions
  # {Exceptions::InvalidParamsError} is an error which is raised when invalid params are passed to the endpoint.
  class InvalidParamsError < StandardError
    attr_reader :object

    # @param [Hash] object that contains details about params errors.
    # @param [String] message of the error.
    def initialize(object, message)
      @object = object

      super(message)
    end
  end
end
```

Our newly created `Exceptions::InvalidParamsError` accepts two additional parameters inside `initialize` method:

- `object` that contains details about params errors.
- `message` of the error.

Ok, one of the first endpoints we will going to build will be the `sign up` endpoint. Let create a class that will validate incoming params for that:

app/params/sign_up_params.rb

```
# frozen_string_literal: true

# {SignUpParams} validates POST /api/v1/sign_up params.
#
# @example When params are valid:
#   SignUpParams.new.permit!(email: "test@user.com", password: "password",
password_confirmation: "password_confirmation")
#
# @example When params are invalid:
#   SignUpParams.new.permit!({})
class SignUpParams < ApplicationParams
  # @!method params
  #   It stores rules for validating POST /api/v1/sign_up endpoint params using dry-
validation DSL.
  params do
    required(:email).filled(:string).value(format?: Constants::EMAIL_REGEX)
    required(:password).filled(:string)
    required(:password_confirmation).filled(:string)
  end
end
```

Here we are defining params coercion schema using `dry-validation` DSL that will check the following things:

- `email` needs to be present.
- `email` must have a valid format.
- `password` needs to be present.
- `password_confirmation` needs to be present.

Let’s test that in `ruby bin/console`:

```
SignUpParams.new.permit!(email: "test@user.com", password: "password",
password_confirmation: "password_confirmation") # => {:email=>"test@user.com",
:password=>"password", :password_confirmation=>"password_confirmation"}

SignUpParams.new.permit!({}) # => Exceptions::InvalidParamsError: Your query contains
incorrectly formed parameters.
```

The last part is to test `SignUpParams` class with unit tests:

spec/params/sign_up_params_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe SignUpParams do
  describe '#call' do
    context 'when params are invalid' do
      before do
        expect(Exceptions::InvalidParamsError)
          .to receive(:new)
          .with(object, I18n.t('invalid_params'))
          .and_return(Exceptions::InvalidParamsError.new(object, I18n
.t('invalid_params')))
      end

      context 'when params are blank' do
        let(:params) { {} }

        let(:object) do
          {
            email: ['is missing'],
            password: ['is missing'],
            password_confirmation: ['is missing']
          }
        end

        it 'raises InvalidParamsError' do
          expect { described_class.new.permit!(params) }
            .to raise_error(an_instance_of(Exceptions::InvalidParamsError))
        end
      end

      context 'when email has invalid format' do
        let(:params) do
          {
            password: 'password',
            password_confirmation: 'password',
            email: 'invalid-email'
          }
        end

        let(:object) do
          {
            email: ['is in invalid format']
          }
        end

        it 'raises InvalidParamsError' do
          expect { described_class.new.permit!(params) }
```

```
            .to raise_error(an_instance_of(Exceptions::InvalidParamsError))
        end
      end
    end

    context 'when params are valid' do
      let(:params) do
        {
          email: 'test@user.com',
          password: 'password',
          password_confirmation: 'password'
        }
      end

      it 'returns validated params' do
        expect(described_class.new.permit!(params)).to eq params
      end
    end
  end
end
```

In the unit tests, we are testing the following scenarios:

- when `params` are invalid.
- when `params` are blank.
- when `email` has an invalid format.
- when `params` are valid.

It was a long run, but finally, we are ready to start working on our first endpoint!

Chapter 7. Authentication

In this chapter, we will work on authentication endpoints. We will create endpoints for registration, logging in, logging out, and refreshing the tokens.

7.1. Users signup

To have users authenticate in the first place, we need to have them sign up first. First, we will create a dedicated service object that will create users for us, but we need to do another thing before that.

Our `User` has an `authentication_token` attribute that needs to be unique. We need to set up this during User creation so let's create a module that will generate a unique authentication token for the `User`:

/lib/authentication_token_generator.rb

```ruby
# frozen_string_literal: true

# {AuthenticationTokenGenerator} generates unique authentication token for {User}.
module AuthenticationTokenGenerator
  # It generates unique authentication token.
  #
  # @return [String] unique authentication token among all users ({User}).
  #
  # @example Generate unique authentication token:
  #   AuthenticationTokenGenerator.call
  def self.call
    loop do
      random_token = SecureRandom.hex(40)
      break random_token unless User.where(authentication_token: random_token).any?
    end
  end
end
```

In the `AuthenticationTokenGenerator.call`, we generate a random token using `SecureRandom.hex(40)` and check if there is user with that token already. If the user is present, we repeat that process.

Let's quickly test our new module:

spec/lib/authentication_token_generator_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe AuthenticationTokenGenerator do
  describe '.call' do
    let(:token) { described_class.call }
    let(:user)  { create(:user)        }

    it 'returns token' do
      expect(token).not_to be_blank
      expect(token).not_to eq user.authentication_token
    end
  end
end
```

In this test, we are checking if the `authentication_token` generated by the `AuthenticationTokenGenerator` differs from the existing user `authentication_token`.

Now we are ready to create a service object that will set up user account during sign up:

app/services/users/creator.rb

```ruby
# frozen_string_literal: true

module Users
  # {Users::Creator} creates {User} account.
  class Creator
    # @param [Hash] attributes of the {User}
    def initialize(attributes:)
      @attributes = attributes
    end

    # It creates {User} account based on the passed attributes.
    #
    # @return [User] object when attributes are valid.
    #
    # @raise [Sequel::ValidationFailed] when attributes are not valid
    #
    # @example When attributes are valid:
    #   attributes = {email: 'test@user.com', password: 'test',
password_confirmations: 'test'}
    #   Users::Creator.new(user: User.last, attributes: attributes).call
    #
    # @example When attributes are not valid:
    #   Users::Creator.new(attributes: {}).call
    def call
      User.create(
        email: @attributes[:email],
        password: @attributes[:password],
        password_confirmation: @attributes[:password_confirmation],
        authentication_token: authentication_token
      )
    end

    private

    # It generates unique authentication token.
    #
    # @see AuthenticationTokenGenerator
    #
    # @return [String] unique authentication token among all users ({User}).
    def authentication_token
      AuthenticationTokenGenerator.call
    end
  end
end
```

- The `initialize` method accepts user `attributes` as an argument and set that to instance variable.
- Private `authentiacation_token` method generates a unique authentication token for the user using the `AuthenticationTokenGenerator` class.
- In the `call` method, we create a user using the `User.create` notation.

Let's write tests to check if our class works as expected:

spec/services/users/creator_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe Users::Creator do
  describe '#call' do
    let(:result)               { described_class.new(attributes: attributes).call }
    let(:authentication_token) { 'test_authentication_token'                      }

    before do
      expect(AuthenticationTokenGenerator)
        .to receive(:call)
        .and_return(authentication_token)
    end

    context 'when attributes are valid' do
      let(:attributes) do
        {
          email: 'test@user.com',
          password: 'test',
          password_confirmation: 'test'
        }
      end

      let(:created_user) do
        User.find(
          email: attributes[:email],
          authentication_token: authentication_token
        )
      end

      it 'creates and returns User' do
        expect(result).to eq created_user
      end

      it 'sets User password' do
        expect(result.authenticate(attributes[:password])).to eq created_user
      end

      it 'sets User authentication_token' do
        expect(result.authentication_token).to eq authentication_token
      end
    end

    context 'when attributes are invalid' do
      let(:attributes) { {} }

      it 'raise Sequel::ValidationFailed' do
        expect { result }.to raise_error(
          Sequel::ValidationFailed
        )
      end
    end
  end
end
```

Here we are checking two things:

- When attributes are valid `Users::Creator` class creates and returns user.
- When attributes are invalid, the `Sequel::ValidationFailed` error is raised.

Let’s recap what we we have:

- We have `SignUpParams` class that will validate incoming params for the `/api/v1/sign_up` endpoint.
- `Users::Creator` will create a new user account.
- `AuthorizationTokensGenerator` will generate access and refresh token for the `User` object.
- `UserSerializer` will represent the newly created user in the JSON format.

It seems that we have everything to finish our `/api/v1/sign_up` endpoint.

During sign up process two errors can be raised by our application:

- `Exceptions::InvalidParamsError` when incoming params are have invalid format.
- `Sequel::ValidationFailed` when user attributes are invalid.

We want to return a friendly JSON message to the user when one of that scenario happens, so we need to update the `:error_handler` plugin rules in the `app.rb` file:

app.rb

```
  # Adds ability to automatically handle errors raised by the application.
  plugin :error_handler do |e|
    if e.instance_of?(Exceptions::InvalidParamsError)
      error_object    = e.object
      response.status = 422
    elsif e.instance_of?(Sequel::ValidationFailed)
      error_object    = e.model.errors
      response.status = 422
    else
      error_object    = { error: I18n.t('something_went_wrong') }
      response.status = 500
    end

    response.write(error_object.to_json)
  end
```

- when `Exceptions::InvalidParamsError` is raised we return 422 HTTP status and we put `e.object` that contains error details into the JSON response.
- when `Sequel::ValidationFailed` is raised, we also return 422 HTTP status, and we put `e.model.errors` that contain validation error details in the JSON response.

Let's create our route block:

app.rb

```
route do |r|
  r.on('api') do
    r.on('v1') do
      r.post('sign_up') do
        sign_up_params = SignUpParams.new.permit!(r.params)
        user           = Users::Creator.new(attributes: sign_up_params).call
        tokens         = AuthorizationTokensGenerator.new(user: user).call

        UserSerializer.new(user: user, tokens: tokens).render
      end
    end
  end
end
```

Let's review what is going on here:

- First, we validate incoming params using the `SignUpParams` class:

```
sign_up_params = SignUpParams.new.permit!(r.params)
```

- Then we use `Users::Creator` to create a new user account:

```
user = Users::Creator.new(attributes: sign_up_params).call
```

- When user creation is successfull we generate access and refresh tokens:

```
tokens = AuthorizationTokensGenerator.new(user: user).call
```

- The last step is to represent our newly created user in the JSON format:

```
UserSerializer.new(user: user, tokens: tokens).render
```

Ok, let's write integration tests for our endpoint to check if it works as expected:

spec/requests/api/v1/sign_up_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe 'POST /api/v1/sign_up', type: :request do
  context 'when request contains incorrectly formatted params' do
    before { post '/api/v1/sign_up' }

    it 'returns 422 HTTP status' do
      expect(response.status).to eq 422
    end

    it 'returns error message in JSON response' do
      expect(json_response).to eq(
        { 'email' => ['is missing'], 'password' => ['is missing'],
'password_confirmation' => ['is missing'] }
      )
    end
  end

  context 'when request params are valid' do
    let(:params) do
      {
        email: 'test@user.com',
        password: 'password',
        password_confirmation: 'password'
      }
    end

    let(:created_user) do
      User.find(email: params[:email])
    end
    let(:authorization_tokens_generator) do
      instance_double(AuthorizationTokensGenerator)
    end

    let(:tokens) do
      {
        'access_token' => {
          'token' => 'authorization_token',
          'expires_in' => 1800
        },

        'refresh_token' => {
          'token' => 'refresh_token',
          'expires_in' => 3600
        }
      }
    end

    let(:sign_up_json_response) do
      {
        'user' => {
          'id' => created_user.id,
          'email' => created_user.email,
          'created_at' => created_user.created_at.iso8601,
          'updated_at' => created_user.updated_at.iso8601
        },

        'tokens' => tokens
      }
    end

    before do
      expect(AuthorizationTokensGenerator)
        .to receive(:new)
        .and_return(authorization_tokens_generator)

      expect(authorization_tokens_generator)
        .to receive(:call)
        .and_return(tokens)

      post '/api/v1/sign_up', params
    end

    it 'returns 200 status' do
      expect(response.status).to eq 200
    end

    it 'returns user data with its access and refresh token informations in the JSON
response' do
      expect(json_response).to eq sign_up_json_response
    end
```

```
  end

  context 'when password does not match password_confirmation' do
    let(:params) do
      {
        email: 'test@user.com',
        password: 'password',
        password_confirmation: 'test'
      }
    end

    before { post '/api/v1/sign_up', params }

    it 'returns 422 HTTP status' do
      expect(response.status).to eq 422
    end

    it 'returns error message in JSON response' do
      expect(json_response).to eq({ 'password' => ["doesn't match confirmation"] })
    end
  end

  context 'when email has already been taken' do
    let(:user) { create(:user) }

    let(:params) do
      {
        email: user.email,
        password: 'password',
        password_confirmation: 'password'
      }
    end

    before { post '/api/v1/sign_up', params }

    it 'returns 422 HTTP status' do
      expect(response.status).to eq 422
    end

    it 'returns error message in JSON response' do
      expect(json_response).to eq({ 'email' => ['is already taken'] })
    end
  end
end
```

Let’s summarize what we’re testing here:

- When the request contains incorrectly formatted params API should return 422 HTTP status code and error in the JSON response.
- When requests contain valid params API should return 200 HTTP status code and user in the

JSON format.

- When `password` doesn't match `password_confirmation` API should return 422 HTTP status code and error in the JSON response.
- When `email` has already been taken API should return 422 HTTP status code and error in the JSON response.

Now we can check if all tests pass:

```
$ rspec
...........................................................

Finished in 0.23317 seconds (files took 1.65 seconds to load)
59 examples, 0 failures
```

Tests are passing, that's great. Let's also test manually our api using `curl`, let's launch our application server:

```
rackup
```

And let's test `/api/v1/sign_up` endpoint using curl:

```
curl --location --request POST 'http://localhost:9292/api/v1/sign_up' --header
'Content-Type: application/json' --data-raw '{
    "email": "test@user.com",
    "password": "password",
    "password_confirmation": "password"
}'

{"user":{"id":"9bf090b7-cedb-4bdc-bb46-3f9ceb8ae488","email":"test@user.com"
,"created_at":"2021-04-29T11:20:46+00:00","updated_at":"2021-04-29T13:20:46+00:00"
},"tokens":{"access_token":{"token":"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsO
XBaRWtpS1RsaVpqQTVNR0kzTFdObFpHSXROR0prWXkxaVlqUTJMVE5tT1dObFlqaGhaVFE0T0FZNkJrVlVPaGx
oZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYmtraVZXSTFZall4TkRWbFpEVXdPRGcxTTJaaE5ESTVPRGxtWW1ZM
016VTFObU0xT0RnNE1qaGtNVFJqT1RrelpUa3hNelEzT0dNMVpXRXlNak5sTURVek1EUTNaVE5pTXpnell6TXh
NR1kzWlRneUJqc0dWQT09IiwiZXhwIjoiMjAyMS0wNC0yOVQxMToyNTo0NloiLCJwdXIiOiJhY2Nlc3NfdG9rZ
W4ifX0=--
8d325cb844cab4d9ce73a245874a124a3e4cfff1559e5015966615ae31a06b8c76b98928fa30acdaff4a48
2169dd36125728e2ef783f71e3698895a9e8626efa","expires_in":300},"refresh_token":{"token"
:"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsOXBaRWtpS1RsaVpqQTVNR0kzTFdObFpHSXRO
R0prWXkxaVlqUTJMVE5tT1dObFlqaGhaVFE0T0FZNkJrVlVPaGxoZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYm
traVZXSTFZall4TkRWbFpEVXdPRGcxTTJaaE5ESTVPRGxtWW1ZM016VTFObU0xT0RnNE1qaGtNVFJqT1RrelpU
a3hNelEzT0dNMVpXRXlNak5sTURVek1EUTNaVE5pTXpnell6TXhNR1kzWlRneUJqc0dWQT09IiwiZXhwIjoiMj
AyMS0wNC0yOVQxMTozNTo0NloiLCJwdXIiOiJyZWZyZXNoX3Rva2VuIn19--
44f54a64213c6233b950c1fd4fa21261f7e1ddcbaafb5fda37296ba29cf056da3fd66ed800183fd0bad04e
4df67f0afcf3feef2fe320fe51d3b728ebcb84a30c","expires_in":900}}}
```

Everything works as expected. This is how we finished the work on user registration. In the next section, we will work on the `api/v1/login` endpoint that will be used by the existing users to authenticate.

7.2. Users login

Users of our API already have the opportunity to register. Now we have to give them the way to sign in. The parameters that will come to our login endpoint are straightforward. They will only contain the user's email and password. We will start by creating a class that will validate the incoming parameters for the `api/v1/login` endpoint:

/app/params/login_params.rb

```ruby
# frozen_string_literal: true

# {LoginParams} validaties POST /api/v1/login params.
#
# @example When params are valid:
#   LoginParams.new.permit!(email: "test@user.com", password: "password")
#
# @example When params are invalid:
#   LoginParams.new.permit!({})
class LoginParams < ApplicationParams
  # @!method params
  #   It stores rules for validating POST /api/v1/login endpoint params using dry-
validation DSL.
  params do
    required(:email).filled(:string).value(format?: Constants::EMAIL_REGEX)
    required(:password).filled(:string)
  end
end
```

Here we are checking two things:

- `email` needs to be present and have a valid format.
- `password` needs to be present.

Let's write tests for the `LoginParams` class:

/spec/params/login_params_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe LoginParams do
  describe '#call' do
    context 'when params are invalid' do
      before do
        expect(Exceptions::InvalidParamsError)
          .to receive(:new)
          .with(object, I18n.t('invalid_params'))
          .and_return(Exceptions::InvalidParamsError.new(object, I18n
.t('invalid_params')))
```

```
      end

      context 'when params are blank' do
        let(:params) { {} }

        let(:object) do
          {
            email: ['is missing'],
            password: ['is missing']
          }
        end

        it 'raises InvalidParamsError' do
          expect { described_class.new.permit!(params) }.to
raise_error(an_instance_of(Exceptions::InvalidParamsError))
        end
      end

      context 'when email has invalid format' do
        let(:params) { { password: 'password', email: 'invalid-email' } }

        let(:object) do
          {
            email: ['is in invalid format']
          }
        end

        it 'raises InvalidParamsError' do
          expect { described_class.new.permit!(params) }.to
raise_error(an_instance_of(Exceptions::InvalidParamsError))
        end
      end
    end

    context 'when params are valid' do
      let(:params) { { email: 'test@user.com', password: 'password' } }

      it 'returns validated params' do
        expect(described_class.new.permit!(params)).to eq params
      end
    end
  end
end
```

In the `LoginParams` unit tests we are checking the following things:

- when `params` are invalid or blank it should raise an `InvalidParamsError`.
- when `params` are valid it should return validated params
- when `email` has invalid format it should raise an `InvalidParamsError`.

When our incoming parameters are correct, we can think about the authentication of our user. First, we need to find users by email and then check their passwords. The `Users::Authenticator` class will be responsible for this:

/app/services/users/authenticator.rb

```
# frozen_string_literal: true

module Users
  # {Users::Authenticator} checks {User} email and password during authentication
process.
  class Authenticator
    # @param [String] email
    # @param [String] password
    def initialize(email:, password:)
      @email    = email
      @password = password
    end

    # It checks if user email and password are correct.
    #
    # @return [User] when email and password are valid.
    #
    # @raise [Exceptions::InvalidEmailOrPassword] when email or password is invalid.
    #
    # @example When email or password is invalid:
    #   Users::Authenticator.new(email: "invalid-email", password: "invalid-
password").call
    #
    # @example When email or password are valid:
    #   Users::Authenticator.new(email: "user@example.com", password: "password").call
    def call
      user = User.find(email: @email)

      return user if user&.authenticate(@password)

      raise(Exceptions::InvalidEmailOrPassword)
    end
  end
end
```

`initialize` method accepts `email` and `password` attributes that we will use to authenticate the user.

In the `call` method, we first find the user by `email`. If a user is present, we will check the password using the `authenticate` method, which is defined by sequel_secure_password gem. When the password is correct, our class returns the user. If not `Exceptions::InvalidEmailOrPassword` error will be raised. This error class is not defined yet, so let’s do this now:

/lib/exceptions.rb

```
# {Exceptions::InvalidEmailOrPassword} is an error which is raised during
authentication process when email or password is invalid.
class InvalidEmailOrPassword < StandardError
end
```

`Exceptions::InvalidEmailOrPassword` error will be raised when user provide invalid email or password to `api/v1/login` endpoint, because we want to return nicely formatted JSON error to users let's catch this error with `error_handler` plugin:

app.rb

```
# Adds ability to automatically handle errors raised by the application.
plugin :error_handler do |e|
  if e.instance_of?(Exceptions::InvalidParamsError)
    error_object    = e.object
    response.status = 422
  elsif e.instance_of?(Sequel::ValidationFailed)
    error_object    = e.model.errors
    response.status = 422
  elsif e.instance_of?(Exceptions::InvalidEmailOrPassword)
    error_object    = { error: I18n.t('invalid_email_or_password') }
    response.status = 401
  else
    error_object    = { error: I18n.t('something_went_wrong') }
    response.status = 500
  end

  response.write(error_object.to_json)
end
```

Now we are ready to write tests for `Users::Authenticator` class:

/app/services/users/authenticator.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Users::Authenticator do
  describe '#call' do
    let(:result) { described_class.new(email: email, password: password).call }

    context 'when email and password are valid' do
      let(:user) { create(:user) }

      context 'when user email is passed in downcase format' do
        let(:email)    { user.email }
        let(:password) { 'password' }
        it 'returns properly formatted hash' do
          expect(result).to eq user
        end
      end

      context 'when user email is passed in upcase format' do
        let(:email)    { user.email.upcase }
        let(:password) { 'password'        }

        it 'returns properly formatted hash' do
          expect(result).to eq user
        end
      end
    end

    context 'when email or password is invalid' do
      let(:email)    { 'wrong-email'    }
      let(:password) { 'wrong-password' }

      it 'raise Exceptions::InvalidEmailOrPassword' do
        expect { result }.to raise_error(
          Exceptions::InvalidEmailOrPassword
        )
      end
    end

    context 'when email or password are blank' do
      let(:email)    { nil }
      let(:password) { nil }

      it 'raise Exceptions::InvalidEmailOrPassword' do
        expect { result }.to raise_error(
          Exceptions::InvalidEmailOrPassword
        )
      end
    end
  end
end
```

We test the following things here:

- `Exceptions::InvalidEmailOrPassword` should be raised when the email or password is invalid.
- `User` object should be returned when email and password are correct.

Now we are ready to add our `api/v1/login` endpoint to our `api/v1` route block:

app.rb

```
r.post('login') do
  login_params = LoginParams.new.permit!(r.params)
  user         = Users::Authenticator.new(email: login_params[:email], password:
login_params[:password]).call
  tokens       = AuthorizationTokensGenerator.new(user: user).call

  UserSerializer.new(user: user, tokens: tokens).render
end
```

Here it is almost the same as with the registration endpoint:

- First, we validate the incoming parameters using the `LoginParams` class:

```
login_params = LoginParams.new.permit!(r.params)
```

- Then, we authenticate the user using the `Users::Authenticator` class:

```
user = Users::Authenticator.new(email: login_params[:email], password:
login_params[:password]).call
```

- `AuthorizationTokensGenerator` will generate access and refresh tokens for the user:

```
tokens = AuthorizationTokensGenerator.new(user: user).call
```

- `UserSerializer` will represent the authenticated user in the JSON format:

```
UserSerializer.new(user: user, tokens: tokens).render
```

The last step will be to write integration tests for our `api/v1/login` endpoint:

/spec/requests/api/v1/login_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe 'POST /api/v1/login', type: :request do
  context 'when request contains incorrectly formatted params' do
    before { post '/api/v1/login' }

    it 'returns 422 HTTP status' do
      expect(response.status).to eq 422
    end
    it 'returns error message in JSON response' do
      expect(json_response).to eq({ 'email' => ['is missing'], 'password' => ['is missing'] })
    end
  end

  context 'when email or password are invalid' do
    let(:params) { { email: 'wrong@email.com', password: 'wrong-password' } }

    before { post '/api/v1/login', params }

    it 'returns 401 status' do
      expect(response.status).to eq 401
    end

    it 'returns error message in JSON response' do
      expect(json_response).to eq 'error' => 'Invalid email or password.'
    end
  end

  context 'when email and password are valid' do
    let(:user)   { create(:user)                               }
    let(:params) { { email: user.email, password: 'password' } }

    let(:authorization_tokens_generator) do
      instance_double(AuthorizationTokensGenerator)
    end

    let(:tokens) do
      {
        'access_token' => {
          'token' => 'authorization_token',
          'expires_in' => 1800
        },

        'refresh_token' => {
          'token' => 'refresh_token',
          'expires_in' => 3600
        }
      }
    end

    let(:login_json_response) do
      {
        'user' => {
          'id' => user.id,
          'email' => user.email,
          'created_at' => user.created_at.iso8601,
          'updated_at' => user.updated_at.iso8601
        },
        'tokens' => tokens
      }
    end

    before do
      expect(AuthorizationTokensGenerator)
        .to receive(:new)
        .with(user: user)
        .and_return(authorization_tokens_generator)

      expect(authorization_tokens_generator)
        .to receive(:call)
        .and_return(tokens)

      post '/api/v1/login', params
    end

    it 'returns 200 status' do
      expect(response.status).to eq 200
    end

    it 'returns user data with its access and refresh token informations in the JSON
response' do
      expect(json_response).to eq login_json_response
    end
  end
end
```

In this case, our test includes the following scenarios:

- When the request contains incorrectly formatted params, API should return 422 HTTP status code and error in the JSON response.
- When `email` or `password` are invalid, API should return 422 HTTP status code and error in the JSON response.
- When `email` and `password` are valid API should return 200 HTTP status with user data in JSON response.

As a final step let's launch server with `rackup` command and test newly created endpoint using curl:

```
curl --location --request POST 'http://localhost:9292/api/v1/login' \
--header 'Content-Type: application/json' \
--data-raw '{
    "email": "test@user.com",
    "password": "password"
}'

{"user":{"id":"9bf090b7-cedb-4bdc-bb46-3f9ceb8ae488","email":"test@user.com"
,"created_at":"2021-04-29T11:20:46+00:00","updated_at":"2021-04-29T13:20:46+00:00"
},"tokens":{"access_token":{"token":"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsO
XBaRWtpS1RsaVpqQTVNR0kzTFdObFpHSXROR0prWXkxaVlqUTJMVE5tT1dObFlqaGhaVFE0T0FZNkJrVlVPaGx
oZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYmtraVZXSTFZall4TkRWbFpEVXdPRGcxTTJaaE5ESTVPRGxtWW1ZM
016VTFObU0xT0RnNE1qaGtNVFJqT1RrelpUa3hNelEzT0dNMVpXRXlNak5sTURVek1EUTNaVE5pTXpnell6TXh
NR1kzWlRneUJqc0dWQT09IiwiZXhwIjoiMjAyMS0wNS0wNFQxMzo0MzozMVoiLCJwdXIiOiJhY2Nlc3NfdG9rZ
W4ifX0=--
e1ec6de0f1ff1042c159467270b96daf829be97c0dbbf4447941cf3c76c86646ac6ee526be861ef3fce813
cba3ef24df6ff273fffbedeb29d8d7b3c6be894a14","expires_in":300},"refresh_token":{"token"
:"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsOXBaRWtpS1RsaVpqQTVNR0kzTFdObFpHSXRO
R0prWXkxaVlqUTJMVE5tT1dObFlqaGhaVFE0T0FZNkJrVlVPaGxoZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYm
traVZXSTFZall4TkRWbFpEVXdPRGcxTTJaaE5ESTVPRGxtWW1ZM016VTFObU0xT0RnNE1qaGtNVFJqT1RrelpU
a3hNelEzT0dNMVpXRXlNak5sTURVek1EUTNaVE5pTXpnell6TXhNR1kzWlRneUJqc0dWQT09IiwiZXhwIjoiMj
AyMS0wNS0wNFQxMzo1MzozMVoiLCJwdXIiOiJyZWZyZXNoX3Rva2VuIn19--
85281b591a39640543f9bc8423084e32a9a1a0b9dc6382326b2ee98738d25b314f458f23f8145017e58a72
3472422b9d770843f9b599945aa20a18994e433a1d","expires_in":900}}}
```

Everything works as expected. With that in place, we finished our work with the `api/v1/login` endpoint. In the next section, we will implement logout mechanism.

7.3. Users logout

At this moment in our application, there is no way to invalidate existing access and refresh tokens. You may ask why we need that? Our tokens are valid for a short period of time, but there is the possibility that the token can leak and the attacker can use it. The solution for this will be invalidating existing access and refresh tokens. For that, we will use `api/v1/logout` endpoint.

The process of invalidating existing user tokens it's really simple. The only thing we need to do is update user `authentication_token` to a new unique value that will be generated by `AuthenticationTokenGenerator`. After updating, all of the user existing tokens will no longer be valid because of check in the `AuthorizationTokenValidator` class:

lib/authorization_token_validator.rb

```
unless current_user && current_user.authentication_token == data[
:authentication_token]
  raise(ActiveSupport::MessageVerifier::InvalidSignature)
end
```

Let's start by creating a service object that will update user `authentication_token`:

app/services/users/update_authentication_token.rb

```ruby
# frozen_string_literal: true

module Users
  # {Users::UpdateAuthenticationToken} updates {User} authentication_token.
  class UpdateAuthenticationToken
    # @param [User] user
    def initialize(user:)
      @user = user
    end

    # It updates user authentication_token.
    #
    # @return [User] for which authentication_token was updated.
    #
    # @example Update {User} authentication_token:
    #   Users::UpdateAuthenticationToken.new(user: User.last).call
    def call
      @user.update(authentication_token: authentication_token)
    end

    private

    # It generates unique authentication token.
    #
    # @see AuthenticationTokenGenerator
    #
    # @return [String] unique authentication token among all users ({User}).
    def authentication_token
      AuthenticationTokenGenerator.call
    end
  end
end
```

In the `initialize` method our class accepts `user` object. In the `call` method we update user `authentication_token` that was generated by the `AuthenticationTokenGenerator` module.

Let's check if our newly created class works as expected:

spec/services/users/update_authentication_token_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Users::UpdateAuthenticationToken do
  describe '#call' do
    let(:authentication_token) { 'test_authentication_token'          }
    let(:user)                 { create(:user)                        }
    let(:result)               { described_class.new(user: user).call }

    before do
      expect(AuthenticationTokenGenerator)
        .to receive(:call)
        .and_return(authentication_token)

      result
    end

    it 'returns user object' do
      expect(result).to eq user
    end

    it 'updates user authentication_token' do
      expect(user.authentication_token).to eq authentication_token
    end
  end
end
```

Tests are straightforward. We mock `AuthenticationTokenGenerator` and check if the user `authentication_token` column was updated.

To access `api/v1/logout` endpoint, we will need to send the access token in the `Authorization` header and then validate it and get the user account from the database. Let's create a helper method for that:

app

```
# It validates authorization token that was passed in Authorization header.
#
# @see AuthorizationTokenValidator
def current_user
  return @current_user if @current_user

  purpose = request.url.include?('refresh_token') ? :refresh_token : :access_token

  @current_user = AuthorizationTokenValidator.new(
    authorization_token: env['HTTP_AUTHORIZATION'],
    purpose: purpose
  ).call
end

route do |r|
  ...
end
```

Here we first check if `@current_user` is already defined. If is, we will return it. If the `@current_user` instance variable is not present, we check what `purpose` we should use to decode our message. If the `endpoint` URL contains `refresh_token`, then the purpose is `:refresh_token`, otherwise the purpose is `:access_token`. Then we use `AuthorizationTokenValidator` that will return the user when the token is valid or raise `ActiveSupport::MessageVerifier::InvalidSignature` error when the incoming token is not valid.

Because `ActiveSupport::MessageVerifier::InvalidSignature` can be raised during logout process we should catch this error with `:error_handler` plugin and return friendly message to our users:

app

```
# Adds ability to automatically handle errors raised by the application.
plugin :error_handler do |e|
  if e.instance_of?(Exceptions::InvalidParamsError)
    error_object    = e.object
    response.status = 422
  elsif e.instance_of?(Sequel::ValidationFailed)
    error_object    = e.model.errors
    response.status = 422
  elsif e.instance_of?(Exceptions::InvalidEmailOrPassword)
    error_object    = { error: I18n.t('invalid_email_or_password') }
    response.status = 401
  elsif e.instance_of?(ActiveSupport::MessageVerifier::InvalidSignature)
    error_object    = { error: I18n.t('invalid_authorization_token') }
    response.status = 401
  else
    error_object    = { error: I18n.t('something_went_wrong') }
    response.status = 500
  end

  response.write(error_object.to_json)
end
```

Now we can create our endpoint in the `route` block:

app

```
  route do |r|
    r.on('api') do
      r.on('v1') do
        ...

        r.delete('logout') do
          Users::UpdateAuthenticationToken.new(user: current_user).call

          response.write(nil)
        end
      end
    end
  end
```

First, we use the `current_user` helper method to get the user based on the token passed in the `Authorization` header, then we call `Users::UpdateAuthenticationToken` to update user `authentication_token`, and we return an empty response. Simple as that.

Now we can start testing our first endpoint that requires an access token. The problem is that all of our endpoints that require access token will have repeated tests that check how our endpoint behaves when token is invalid, expired, etc.

```
context 'when Authorization header does not contain token' do
  before { delete '/api/v1/logout' }

  it 'returns 401 HTTP status' do
    expect(response.status).to eq 401
  end

  it 'returns error message in the JSON response' do
    expect(json_response).to eq({ 'error' => 'Invalid authorization token.' })
  end
end

context 'when Authorization header contains invalid token' do
  before do
    header 'Authorization', 'invalid-authorization-token'

    delete '/api/v1/logout'
  end

  it 'returns 401 HTTP status' do
    expect(response.status).to eq 401
  end

  it 'returns error message in the JSON response' do
    expect(json_response).to eq({ 'error' => 'Invalid authorization token.' })
  end
end

context 'when user authentication_token is invalid' do
  let(:user) { create(:user) }

  before do
    header 'Authorization', access_token(user)

    user.update(authentication_token: 'test')

    delete '/api/v1/logout'
  end

  it 'returns 401 HTTP status' do
    expect(response.status).to eq 401
  end

  it 'returns error message in the JSON response' do
    expect(json_response).to eq({ 'error' => 'Invalid authorization token.' })
  end
end
```

To not repeat those tests, we will use the great RSpec feature, shared examples. Let’s add our first shared examples to extract test for HTTP status and error response:

/spec/support/shared_examples/unauthorized.rb

```
# frozen_string_literal: true

RSpec.shared_examples 'unauthorized' do
  it 'returns 401 HTTP status' do
    expect(response.status).to eq 401
  end

  it 'returns error message in the JSON response' do
    expect(json_response).to eq({ 'error' => 'Invalid authorization token.' })
  end
end
```

Here we extract examples that test HTTP status and response body when the access token is invalid. Our next shared example will check the following examples:

- API behavior when Authorization header does not contain token.
- API behavior when Authorization header contains the invalid token.
- API behavior when user `authentication_token` is invalid.

/spec/support/shared_examples/authorization_check.rb

```
# frozen_string_literal: true

RSpec.shared_examples 'authorization check' do |method, url|
  context 'when Authorization header does not contain token' do
    before { public_send(method, url) }

    include_examples 'unauthorized'
  end

  context 'when Authorization header contains invalid token' do
    before do
      header 'Authorization', 'invalid-authorization-token'

      public_send(method, url)
    end

    include_examples 'unauthorized'
  end

  context 'when user authentication_token is invalid' do
    let(:user) { create(:user) }

    before do
      header 'Authorization', access_token(user)

      user.update(authentication_token: 'test')

      public_send(method, url)
    end

    include_examples 'unauthorized'
  end
end
```

`authorization check` shared example requires two arguments:

- Method of the request.
- URL where the request should be sent.

So for the `DELETE api/v1/logout` endpoint, we will use it like that:

```
include_examples 'authorization check', 'delete', '/api/v1/logout'
```

With that in place let's finish testing logout endpoint:

/spec/requests/api/v1/logout_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe 'DELETE /api/v1/logout', type: :request do
  include_examples 'authorization check', 'delete', '/api/v1/logout'

  context 'when Authorization headers contains valid token' do
    let(:user) { create(:user) }

    let(:update_authentication_token) do
      instance_double(Users::UpdateAuthenticationToken)
    end

    before do
      expect(Users::UpdateAuthenticationToken)
        .to receive(:new)
        .with(user: user)
        .and_return(update_authentication_token)

      expect(update_authentication_token)
        .to receive(:call)

      header 'Authorization', access_token(user)

      delete '/api/v1/logout'
    end

    it 'returns 200 HTTP status' do
      expect(response.status).to eq 200
    end

    it 'returns empty response body' do
      expect(response.body).to eq ''
    end
  end
end
```

In the `api/v1/logout` endpoint request specs, we first test how our endpoint behaves when the access token in the Authorization header is invalid. We do that using the `authorization check` RSpec shared example. Next, we check our API endpoint when authorization is successful. We mock `Users::UpdateAuthenticationToken` and check if API is response is correct.

Let’s run tests to see if everything works:

```
$ rspec
..................................................................................

Finished in 0.25945 seconds (files took 1.37 seconds to load)
82 examples, 0 failures
```

It works! In the next section will create a mechanism for refreshing tokens for users.

7.4. Refreshing an access token

After login user of our API gets two tokens in the JSON response:

- `access_token`, which is valid for 5 minutes.
- `refresh_token`, which is valid for 15 minutes.

It would not be best for our users to retype login and password every 5 minutes. This is where `refresh_token` can help us. We need to create `api/v1/refresh_token` endpoint that will accept refresh token in the Authorization header and generate pair of new of access and refresh token.

Let’s describe step by step what will happen during the refresh token process:

- We need to get current user from the refresh token sent in the Authorization header.
- Then, we need to update user `authentication_token` to invalidate existing access and refresh tokens.
- We need to generate new pair of access and refresh tokens and return them in the JSON response.

Let’s start by creating `TokensSerializer` that will represent in the JSON format our access and refresh token:

/app/serializers/tokens_serializer.rb

```ruby
# frozen_string_literal: true

# {TokensSerializer} is responsible for representing access and refresh token
informations.
#
# @example Represent {User} access and refresh token in the JSON format:
#   tokens = { access_token: 'access_token', refresh_token: 'refresh_token' }
#   TokensSerializer.new(tokens: tokens).render
class TokensSerializer < ApplicationSerializer
  # It generates Hash object that contains and authorization and refresh token
details.
  #
  # @return [Hash] object that contatins authorization and refresh token details.
  #
  # @example Prepare data before transformation to the JSON format:
  #   tokens = { access_token: 'access_token', refresh_token: 'refresh_token' }
  #   TokensSerializer.new(tokens: tokens).to_json
  def to_json
    {
      tokens: @tokens
    }
  end
end
```

`TokenSerializer` is responsible for representing access and refresh token information in the JSON format. With that in place, we've got everything to create an endpoint for refreshing token:

app.rb

```
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.post('refresh_token') do
        Users::UpdateAuthenticationToken.new(user: current_user).call

        tokens = AuthorizationTokensGenerator.new(user: current_user).call

        TokensSerializer.new(tokens: tokens).render
      end
    end
  end
end
```

Let's review what's going on here:

- First, we get the user for which we want to regenerate tokens using the `current_user` helper method.
- Next we use `Users::UpdateAuthenticationToken` class to update user `authentication_token`.
- With `AuthorizationTokensGenerator` we generate new pair of access and refresh tokens.
- Last step is to use `TokensSerializer` to represent tokens in JSON format.

We are ready to write tests for the `api/v1/refresh_token` endpoint:

/spec/requests/api/v1/refresh_token_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe 'POST /api/v1/refresh_token', type: :request do
  let(:user) { create(:user) }

  include_examples 'authorization check', 'post', '/api/v1/refresh_token'

  context 'when Authorization headers contains valid refresh token' do
    let(:update_authentication_token) do
      instance_double(Users::UpdateAuthenticationToken)
    end

    let(:authorization_tokens_generator) do
      instance_double(AuthorizationTokensGenerator)
    end
    let(:tokens) do
      {
        'access_token' => {
          'token' => 'authorization_token',
          'expires_in' => 1800
        },

        'refresh_token' => {
          'token' => 'refresh_token',
          'expires_in' => 3600
        }
      }
    end

    before do
      expect(AuthorizationTokensGenerator)
        .to receive(:new)
        .with(user: user)
        .and_return(authorization_tokens_generator)

      expect(authorization_tokens_generator)
        .to receive(:call)
        .and_return(tokens)

      expect(Users::UpdateAuthenticationToken)
        .to receive(:new)
        .with(user: user)
        .and_return(update_authentication_token)

      expect(update_authentication_token)
        .to receive(:call)

      header 'Authorization', refresh_token(user)

      post '/api/v1/refresh_token'

      user.reload
    end

    it 'returns 200 HTTP status' do
      expect(response.status).to eq 200
    end

    it 'returns new authorization and refresh token in the JSON response' do
      expect(json_response).to eq('tokens' => tokens)
    end
  end

  context 'when Authorization headers contains valid authorization token with invalid purpose' do
```

```
    before do
      header 'Authorization', access_token(user)

      post '/api/v1/refresh_token'
    end

    include_examples 'unauthorized'
  end
end
```

In the `api/v1/refresh_token` endpoint tests, we are checking the following things:

- That endpoint needs to receive a valid token in the `Authorization` header. We are using our `authorization check` RSpec shared example.
- We test that it should regenerate tokens when a valid refresh token is sent in the Authorization header.
- We also check that API should return 401 HTTP status when access token instead of the refresh token is sent.

Finally, let’s check if all automatic tests pass:

```
 $ rspec
......................................................................................
......

Finished in 0.27692 seconds (files took 1.73 seconds to load)
92 examples, 0 failures
```

In this way, we were able to complete the authentication mechanisms in our API. Our users can sign up, log in, log out, and refresh their tokens.

In the next chapter we will create endpoints for managing user todos.

Chapter 8. Todo Management

In this chapter, we will give our users the possibility to manage their todos. We will create endpoints for creating, updating, destroying, and presenting todos. Let's get to work.

8.1. Listing Todos

The first endpoint in the Todo Management chapter will be the `/api/v1/todos` endpoint that will allow our users to get all of their todos, filter and order them.

Let's review what we need to do to implement `api/v1/todos` endpoint:

- We need to validate incoming parameters at first. We will support `search_by_name`, `search_by_description`, `sort` and `direction` parameters.
- The next thing we need is a query object responsible for filtering and ordering user todos based on incoming parameters.
- We will also need a serializer that will represent a list of todos in the JSON format.
- The last thing will be adding `api/v1/todos` endpoint to our routing tree.

So let's create `TodosParams` that will validate incoming parameters:

/app/params/todos_params.rb

```ruby
# frozen_string_literal: true

# {TodosParams} validates GET /api/v1/teachers params.
#
# @example When params are valid:
#   TodosParams.new.permit!(search_by_name: 'mat')
#
# @example When params are invalid:
#   TodosParams.new.permit!({direction: "invalid"})
class TodosParams < ApplicationParams
  # @!method params
  #   It stores rules for validating GET /api/v1/todos endpoint params using dry-
validation DSL.
  params do
    optional(:search_by_name).value(:string)
    optional(:search_by_description).value(:string)
    optional(:sort).value(included_in?: Constants::TODO_SORT_COLUMNS)
    optional(:direction).value(included_in?: Constants::SORT_DIRECTIONS)
  end
end
```

Our newly created class have the following rules:

- `search_by_name` and `search_by_description` should be `string` type.

- `sort` value should be one of defined in the `Constants::TODO_SORT_COLUMNS`.
- `direction` value should be one of defined in the `Constants::SORT_DIRECTIONS`.

All of those parameters are optional. As usual let’s write unit tests for that:

/spec/params/todos_params_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe TodosParams do
  describe '#call' do
    context 'when params are invalid' do
      before do
        expect(Exceptions::InvalidParamsError)
          .to receive(:new)
          .with(object, I18n.t('invalid_params'))
          .and_return(Exceptions::InvalidParamsError.new(object, I18n
.t('invalid_params')))
      end

      let(:params) do
        {
          direction: 'invalid',
          sort: 'invalid'
        }
      end

      let(:object) do
        {
          direction: ['must be one of: desc, asc'],
          sort: ['must be one of: name, description, created_at, updated_at']
        }
      end

      it 'raises InvalidParamsError' do
        expect { described_class.new.permit!(params) }.to
raise_error(an_instance_of(Exceptions::InvalidParamsError))
      end
    end

    context 'when params are valid' do
      let(:params) do
        {
          search_by_name: 'milk',
          search_by_description: 'buy milk',
          sort: 'name',
          direction: 'desc'
        }
      end
      it 'returns validated params' do
        expect(described_class.new.permit!(params)).to eq params
      end
    end
  end
end
```

We have two testing scenarios here:

- When parameters are invalid, `Exceptions::InvalidParamsError` should be raised.
- When parameters are valid, the class should return validated parameters.

Given the todo list we defined at the beginning of this chapter, the next step will be to create `TodosQuery` query object that will be responsible for filtering and ordering todos:

/app/queries/todos_query.rb

```
# frozen_string_literal: true

# {TodosQuery} class is responsible for filtering todos ({Todo}).
class TodosQuery
  # @param [Todo] dataset to process.
  # @param [Hash] params with attributes to filter.
  def initialize(dataset:, params:)
    @dataset = dataset
    @params  = params
  end

  # It filters todos ({Todo}) based on the provided params.
  #
  # @return [Array<Todo>] Array of {Todo} objects.
  #
  # @example Find {Todo} by name:
  #   TodosQuery.new(dataset: Todo, params: {search_by_name: 'milk'}).call
  #
  # @example Find {Todo} by description:
  #   TodosQuery.new(dataset: Todo, params: {search_by_description: 'buy milk'}).call
  #
  # @example Find {Todo} by multiple attributes:
  #   TodosQuery.new(dataset: Todo, params: {search_by_name: 'milk',
search_by_description: 'buy milk'}).call
  #
  # @example Order todos ({Todo}) by name in ascending order:
  #   TodosQuery.new(dataset: Todo, params: {sort: 'name', direction: 'asc'}).call
  def call
    scoped = @dataset
    scoped = scoped.search_by_name(@params[:search_by_name]) if @params
[:search_by_name]
    scoped = scoped.search_by_description(@params[:search_by_description]) if @params
```

```
[:search_by_description]
    scoped = scoped.order(Sequel.public_send(direction, sort))

    scoped.all
  end

  private

  # It returns sort order.
  #
  # @return [Symbol] Sort order.
  def sort
    (@params[:sort] || :created_at).to_sym
  end

  # It returns sort direction.
  #
  # @return [Symbol] Sort direction.
  def direction
    @params[:direction] || :desc
  end
end
```

It is the first query object class in our application, so let's go into more detail about what's going on here.

In the `initialize` method, we accept two parameters, `dataset`, which represents SQL query, and `params`, that will be used to filter and sort todos.

In the `call` method, we check if the `search_by_name` parameter is present. If it is, then we filter our dataset using the `search_by_name` method defined in the `app/todo.rb` file otherwise, we skip this line.

Next, we check the presence of the `search_by_description` parameter, when its present, our dataset will be filtered by the `search_by_description` method defined in the `app/todo.rb` file, same as in the previous case if parameters are missing, we skip the filtering by description.

We use Sequel order method together with `Sequel.desc` and `Sequel.asc` methods to sort our todos. If `sort` and `direction` parameters are empty, our collection will be by default sorted by `created_at` column in descending order: `scoped.order(Sequel.desc(:created_at))`.

The last step is to use the `all` method that returns an array with all records in the dataset.

Let's test if `TodosQuery` works as expected:

/spec/queries/todos_query_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe TodosQuery do
```

```ruby
describe '#call' do
  let!(:todo)   { create(:todo)                                            }
  let(:todos)   { described_class.new(dataset: Todo, params: params).call  }
  let(:dataset) { instance_double(Sequel::Postgres::Dataset)               }

  context 'when @params does not have any filters' do
    let(:params) { {} }

    before do
      expect(Todo)
        .to receive(:order)
        .with(Sequel.desc(:created_at))
        .and_return(dataset)

      expect(dataset)
        .to receive(:all)
        .and_return([todo])
    end

    it 'returns whole dataset records' do
      expect(todos).to eq [todo]
    end
  end

  context 'when @params[:search_by_name] is present' do
    let(:params) { { search_by_name: 'milk' } }

    before do
      expect(Todo)
        .to receive(:search_by_name)
        .with(params[:search_by_name])
        .and_return(dataset)

      expect(dataset)
        .to receive(:order)
        .with(Sequel.desc(:created_at))
        .and_return(dataset)

      expect(dataset)
        .to receive(:all)
        .and_return([todo])
    end

    it 'returns filtered todos' do
      expect(todos).to eq [todo]
    end
  end

  context 'when @params[:search_by_description] is present' do
    let(:params) { { search_by_description: 'buy milk' } }
```

```
      before do
        expect(Todo)
          .to receive(:search_by_description)
          .with(params[:search_by_description])
          .and_return(dataset)

        expect(dataset)
          .to receive(:order)
          .with(Sequel.desc(:created_at))
          .and_return(dataset)

        expect(dataset)
          .to receive(:all)
          .and_return([todo])
      end

      it 'returns filtered todos' do
        expect(todos).to eq [todo]
      end
    end

    context 'when @params[:sort] and @params[:direction] are present' do
      let(:params) { { sort: 'name', direction: 'asc' } }

      before do
        expect(Todo)
          .to receive(:order)
          .with(Sequel.asc(:name))
          .and_return(dataset)

        expect(dataset)
          .to receive(:all)
          .and_return([todo])
      end

      it 'returns ordered todos' do
        expect(todos).to eq [todo]
      end
    end
  end
end
```

In this test we do not check if searching by name or description works, we already wrote that tests in `spec/models/todo_spec.rb` where we test two dataset methods, `search_by_name` and `search_by_description`. Here we only need to check that `search_by_description` and `search_by_name` are called when relevant parameters are present.

The same is with the `order` method. We only need to check here that this method is called with valid attributes. Because of that we test the behavior of our class in four scenarios:

- When params object is empty.
- When search_by_name parameter is present.
- When search_by_description parameter is present.
- When sort and direction parameters are present.

We need to create a serializer that will represent our Todo object in JSON format. We will create two serializer classes:

- TodoSerializer, which will represent a single Todo object in the JSON format.
- TodosSerializer will represent multiple Todo objects in the JSON format.

Let's start by defining TodoSerializer:

/app/serializers/todo_serializer.rb

```ruby
# frozen_string_literal: true

# {TodoSerializer} is responsible for representing single todo ({Todo}) in JSON
format.
#
# @example Represent {Todo} in the JSON format:
#   TodoSerializer.new(todo: Todo.last).render
class TodoSerializer < ApplicationSerializer
  # It generates Hash object with single todo ({Todo}) details.
  #
  # @return [Hash] object with single todo ({Todo}) details.
  #
  # @example Prepare data before transformation to the JSON format:
  #   TodoSerializer.new(todo: Todo.last).to_json
  def to_json
    {
      id: @todo.id,
      name: @todo.name,
      description: @todo.description,
      created_at: @todo.created_at,
      updated_at: @todo.updated_at
    }
  end
end
```

In the to_json method, we prepare data before transformation to the JSON format, we are creating Hash that includes. all todo attributes: id, name, description, created_at and updated_at.

TodosQuery class will return a list of Todo objects so let's create a TodosSerializer that will use TodoSerializer under the hood to represent multiple todo objects in the JSON format:

/app/serializers/todos_serializer.rb

```ruby
# frozen_string_literal: true

# {TodosSerializer} is responsible for representing multiple todos ({Todo}) in JSON format.
#
# @example Represent multiple todos {Todo} in the JSON format:
#   TodosSerializer.new(todo: Todo.all).render
class TodosSerializer < ApplicationSerializer
  # It generates Hash object with multiple todos ({Todo}) details.
  #
  # @return [Hash] object with multiple todos ({Todo}) details.
  def to_json
    {
      todos: todos
    }
  end

  private

  # It returns array of todos ({Todo}).
  #
  # @return [Array<>Hash] todos ({Todo}) data.
  def todos
    @todos.map do |todo|
      TodoSerializer.new(todo: todo).to_json
    end
  end
end
```

Here we iterate over the `@todos` instance variable that contains a list of `todos`. For each of them, we use `TodoSerializer` to build an array of `todos` we later convert to the JSON format.

Now we've got everything to add `api/v1/todos` endpoint to our routing tree:

app.rb

```
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.on('todos') do
        # We are calling the current_user method to get the current user
        # from the authorization token that was passed in the Authorization header.
        current_user

        r.get do
          todos_params = TodosParams.new.permit!(r.params)
          todos        = TodosQuery.new(dataset: current_user.todos_dataset, params:
todos_params).call

          TodosSerializer.new(todos: todos).render
        end
      end
    end
  end
end
```

The `r.on` method creates branches in the routing tree. We called `r.on` with the string `todos`, which will match the current request path if the request path starts with `users`.

Before calling `r.get` we call `current_user` method to get the

Then we use `r.get` methods which are for routing based on the GET request method. If it is invoked without a matcher, it puts a simple match against the request method. If invoked with a matcher, a terminal match is performed against the request path.

As always, first we validate the incoming parameters using the `TodosParams` class:

```
todos_params = TodosParams.new.permit!(r.params)
```

The next step is to use the `TodosQuery` class to get the list of `todos` we want to return:

```
todos = TodosQuery.new(dataset: current_user.todos_dataset, params: todos_params).call
```

The last thing is to return the `todos` collection in the JSON format using the `TodosSerializer` class:

```
TodosSerializer.new(todos: todos).render
```

Now we are ready to write integration tests:

/spec/requests/api/v1/todos/index_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe 'GET /api/v1/todos', type: :request do
  include_examples 'authorization check', 'get', '/api/v1/todos'

  context 'when Authorization headers contains valid token' do
    let(:token) { access_token(user) }
    let(:user)  { create(:user)      }

    context 'when request contains incorrectly formatted params' do
      before do
        header 'Authorization', token

        get '/api/v1/todos', { sort: 'invalid' }
      end

      it 'returns 422 HTTP status' do
        expect(response.status).to eq 422
      end

      it 'returns error message in JSON response' do
        expect(json_response).to eq({ 'sort' => ['must be one of: name, description,
created_at, updated_at'] })
      end
    end

    context 'when request params are valid' do
      context 'when there is no todos in the database' do
        let(:todos_json_response) do
          {
            'todos' => []
          }
        end

        before do
          header 'Authorization', token

          get '/api/v1/todos'
        end

        it 'returns 200 HTTP status' do
          expect(response.status).to eq 200
        end

        it 'returns empty array in the reponse body' do
          expect(json_response).to eq todos_json_response
        end
      end

      context 'when there are todos in database' do
        let!(:todo) { create(:todo, user: user) }

        let(:todos_json_response) do
          {
            'todos' => [
              {
                'id' => todo.id,
                'name' => todo.name,
                'description' => todo.description,
                'created_at' => todo.created_at.iso8601,
                'updated_at' => todo.updated_at.iso8601
              }
            ]
          }
        end

        before do
          header 'Authorization', token

          get '/api/v1/todos'
        end

        it 'returns 200 HTTP status' do
          expect(response.status).to eq 200
        end

        it 'returns todos data in JSON reponse' do
          expect(json_response).to eq todos_json_response
        end
      end

      context 'when search params are present' do
        let!(:todo) { create(:todo, name: 'Buy milk.', description: 'Remember to buy milk.', user: user) }

        let(:params) do
          {
            search_by_name: 'milk',
            search_by_description: 'buy milk',
            sort: 'name',
            direction: 'asc'
          }
        end

        let(:todos_json_response) do
          {
            'todos' => [
              {
                'id' => todo.id,
                'name' => todo.name,
                'description' => todo.description,
                'created_at' => todo.created_at.iso8601,
                'updated_at' => todo.updated_at.iso8601
              }
            ]
          }
        end

        before do
          header 'Authorization', token

          get '/api/v1/todos', params
        end

        it 'returns 200 HTTP status' do
          expect(response.status).to eq 200
        end

        it 'returns filtered todos data in JSON reponse' do
          expect(json_response).to eq todos_json_response
        end
      end
    end
  end
end
```

Let’s discuss what our integration tests contain in this case:

- First, we use the `authorization check` shared example to test how our endpoint behaves when there is no access token in the `Authorization` header.
- We check that API should return 422 HTTP status code and error in the JSON response when the request contains incorrectly formatted parameters.
- When todos are present in the database API should return 200 HTTP status code and todos in the JSON response.
- When todos are not present in the database API should return 200 HTTP status code and return an empty array in the JSON response.
- When search parameters are present, API should return 200 HTTP status code and return a list of filtered todos in the JSON response.

Let’s see if tests are passing:

```
rspec
......................................................................................
..........................

Finished in 0.4416 seconds (files took 2.17 seconds to load)
112 examples, 0 failures
```

Tests are passing, so everything is working as expected. In the next section, we will work on todos creation.

8.2. Todos creation

In the previous section, we added our users the ability to download information about their todos. In this section, we'll look at adding the ability to create a todo for a user.

Every `Todo` has the `name` and `description`, so we need to validate that incoming paramters will have those values:

/app/params/todo_params.rb

```ruby
# frozen_string_literal: true

# {TodoParams} validates POST /api/v1/todos and PUT api/v1/todos/:id params.
#
# @example When params are valid:
#   Api::V1::TodoParams.new.permit!(name: 'Buy milk.', description: 'Please buy milk.')
#
# @example When params are invalid:
#   Api::V1::TodoParams.new.permit!({})
class TodoParams < ApplicationParams
  # @!method params
  #   It stores rules for validating POST /api/v1/todos and PUT api/v1/todos/:id endpoint params using dry-validation DSL.
  params do
    required(:name).filled(:string)
    required(:description).filled(:string)
  end
end
```

In the `TodoParams` class, we check if the `name` and `description` are `present`, we will also reuse that class when we work on the endpoint for updating todos later. Let's write some tests:

/spec/params/todo_params_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Api::V1::TodoParams do
  describe '#call' do
    context 'when params are invalid' do
      before do
        expect(Exceptions::InvalidParamsError)
          .to receive(:new)
          .with(object, I18n.t('invalid_params'))
          .and_return(Exceptions::InvalidParamsError.new(object, I18n
.t('invalid_params')))
      end

      context 'when params are blank' do
        let(:params) { {} }

        let(:object) do
          {
            name: ['is missing'],
            description: ['is missing']
          }
        end

        it 'raises InvalidParamsError' do
          expect { described_class.new.permit!(params) }.to
raise_error(an_instance_of(Exceptions::InvalidParamsError))
        end
      end
    end

    context 'when params are valid' do
      let(:params) { { name: 'Buy milk.', description: 'Please buy milk.' } }

      it 'returns validated params' do
        expect(described_class.new.permit!(params)).to eq params
      end
    end
  end
end
```

Our tests, in this case, do not differ from the previous ones:

- We ensure that `Exceptions::InvalidParamsError` is raised when provided parameters are invalid.
- We make sure that validated params will be returned when they are valid.

We have solved the problem of validating incoming parameters that we will be using when creating a todo. Our next step will be to create a dedicated class responsible for saving todo in the database.

/app/services/todos/creator.rb

```ruby
# frozen_string_literal: true

module Todos
  # {Todos::Creator} creates {Todo} for specified {User}.
  class Creator
    # @param [User] the user that newly created Todo will belong to.
    # @param [Hash] attributes of the {Todo}.
    def initialize(user:, attributes:)
      @user       = user
      @attributes = attributes
    end

    # It creates {Todo} object for specified {User} based on the passed attributes.
    #
    # @return [Todo] object when attributes are valid.
    #
    # @raise [Sequel::ValidationFailed] when attributes are not valid
    #
    # @example When attributes are valid:
    #   attributes = {name: 'Buy milk.', description: 'Please buy milk.'}
    #   Todos::Creator.new(user: User.last, attributes: attributes).call
    #
    # @example When attributes are not valid:
    #   Todos::Creator.new(user: User.last, attributes: {}).call
    def call
      Todo.create(
        user: @user,
        name: @attributes[:name],
        description: @attributes[:description]
      )
    end
  end
end
```

- In the `initialize` method we accept `User` object for which newly created `Todo` will belong to.
- The `call` method uses `Todo.create` method provided by `Sequel::Model` that will raise `Sequel::ValidationFailed` when validation fails or return newly created `Todo` object when creation will be successfull.

Let’s see if our class behaves the way we expect it to be:

/spec/services/todos/creator_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Todos::Creator do
  describe '#call' do
    let(:user)   { create(:user)                                                }
    let(:result) { described_class.new(user: user, attributes: attributes).call }

    context 'when attributes are valid' do
      let(:attributes) { { name: 'Buy milk.', description: 'Please buy milk.' } }

      let(:created_todo) do
        Todo.find(
          user: user,
          name: 'Buy milk.',
          description: 'Please buy milk.'
        )
      end

      it 'creates and returns Todo' do
        expect(result).to eq created_todo
      end
    end

    context 'when attributes are invalid' do
      let(:attributes) { {} }

      it 'raise Sequel::ValidationFailed' do
        expect { result }.to raise_error(
          Sequel::ValidationFailed
        )
      end
    end
  end
end
```

Here we are checking two things:

- When attributes are valid, the `Todos::Creator` class creates and returns the `Todo` object.
- When attributes are invalid, the `Sequel::ValidationFailed` error is raised.

Now we are ready to add todo creation endpoint to the routing tree:

app.rb

```
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.on('todos') do
        # We are calling the current_user method to get the current user
        # from the authorization token that was passed in the Authorization header.
        current_user

        ...

        r.post do
          todo_params = TodoParams.new.permit!(r.params)
          todo        = Todos::Creator.new(user: current_user, attributes:
todo_params).call

          TodoSerializer.new(todo: todo).render
        end
      end
    end
  end
end
```

As with the previous endpoints, our first step is to validate and make sure the incoming parameters are correct:

```
todo_params = TodoParams.new.permit!(r.params)
```

Next, we user the `Todos::Creator` class that is responsible for `Todo` creation:

```
todo = Todos::Creator.new(user: current_user, attributes: todo_params).call
```

The last step is to return the newly create `Todo` object in the JSON format:

```
TodoSerializer.new(todo: todo).render
```

That was quick. Let’s write tests to check if we didn’t make any mistakes:

/spec/requests/api/v1/todos/create_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'
```

```
describe 'POST /api/v1/todos', type: :request do
  include_examples 'authorization check', 'post', '/api/v1/todos'

  context 'when Authorization headers contains valid token' do
    let(:token) { access_token(user) }
    let(:user)  { create(:user)      }

    before do
      header 'Authorization', token

      post '/api/v1/todos', params
    end

    context 'when request contains incorrectly formatted params' do
      let(:params) { {} }

      it 'returns 422 HTTP status' do
        expect(response.status).to eq 422
      end

      it 'returns error message in JSON response' do
        expect(json_response).to eq({ 'description' => ['is missing'], 'name' => ['is
missing'] })
      end
    end

    context 'when request params are valid' do
      let(:params) { { name: 'Buy milk.', description: 'Please buy milk.' } }

      let(:created_todo) do
        Todo.find(
          user: user,
          name: params[:name],
          description: params[:description]
        )
      end

      let(:todo_json_response) do
        {
          'id' => created_todo.id,
          'name' => created_todo.name,
          'description' => created_todo.description,
          'created_at' => created_todo.created_at.iso8601,
          'updated_at' => created_todo.updated_at.iso8601
        }
      end

      it 'returns 200 HTTP status' do
        expect(response.status).to eq 200
      end
      it 'returns todo data in the JSON response' do
        expect(json_response).to eq todo_json_response
      end
    end
  end
end
```

In the todo creation tests, we are checking the following things:

- That endpoint needs to receive a valid token in the Authorization header. We are using our authorization check RSpec shared example.
- The API should return 422 HTTP status code and error in the JSON response when the request contains incorrectly formatted parameters
- The API should return 200 HTTP status code and the newly created `Todo` object when incoming parameters are valid.

Let's check if all tests in our application pass:

```
rspec
......................................................................................
........................................

Finished in 0.4194 seconds (files took 1.91 seconds to load)
126 examples, 0 failures
```

Once again, successful, all tests are green. The next step will be to give our users the ability to get the informations about the particular todo.

8.3. Showing particular Todo

In this section, we will work on creating an endpoint for showing particular todo in the JSON format. Later we will work on updating and deleting todos. Those endpoints have one thing in common. They have the same `URL` address, but they require a different HTTP method:

- `GET /api/v1/todos/:uuid` - Showing particular `Todo` object.
- `PUT /api/v1/todos/:uuid` - Updating `Todo` object.
- `DELETE /api/v1/todos/:uuid` - Deleting `Todo` object.

Part of our URL will be todo id in the UUID format eg.: `/api/v1/todos/a33253cf-7f03-4b38-b2e4-f4af966fc8a9`. We want to check that the todo id in the URL conforms to the UUID format. We will use symbol_matchers plugin for that. First, let’s define a custom symbol matcher that will validate the incoming id part of the `api/v1/todos/:id` URL:

app.rb

```
# The symbol_matchers plugin allows you do define custom regexps to use for specific
symbols.
plugin :symbol_matchers

# Validate UUID format.
symbol_matcher :uuid, Constants::UUID_REGEX
```

Now we can use our newly created symbol matcher in `todos` routing branch:

app.rb

```ruby
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.on('todos') do
        # We are calling the current_user method to get the current user
        # from the authorization token that was passed in the Authorization header.
        current_user

        r.on(:uuid) do |id|
          todo = current_user.todos_dataset.with_pk!(id)

          r.get do
            TodoSerializer.new(todo: todo).render
          end
        end

        ...
      end
    end
  end
end
```

First, we use a custom symbol matcher to validate that id in the URL is valid `UUID`:

```ruby
r.on(:uuid) do |id|
```

If the id passed in the URL is valid, we try to find `Todo` in the database:

```ruby
todo = current_user.todos_dataset.with_pk!(id)
```

The last step is to represent found `Todo` in the JSON format using the `TodoSerializer` class:

```ruby
TodoSerializer.new(todo: todo).render
```

Before we dive into writing integration tests, there is one more thing. the `with_pk!` method returns the first record in the dataset with the specified primary key value. If the record is not present, `Sequel::NoMatchingRow` will be raised. Because of that, we need to update our `error_handler` plugin rules to catch that error and present a friendly message to our users:

app.rb

```ruby
# Adds ability to automatically handle errors raised by the application.
plugin :error_handler do |e|
  if e.instance_of?(Exceptions::InvalidParamsError)
    error_object    = e.object
    response.status = 422
  elsif e.instance_of?(Sequel::ValidationFailed)
    error_object    = e.model.errors
    response.status = 422
  elsif e.instance_of?(Exceptions::InvalidEmailOrPassword)
    error_object    = { error: I18n.t('invalid_email_or_password') }
    response.status = 401
  elsif e.instance_of?(ActiveSupport::MessageVerifier::InvalidSignature)
    error_object    = { error: I18n.t('invalid_authorization_token') }
    response.status = 401
  elsif e.instance_of?(Sequel::NoMatchingRow)
    error_object    = { error: I18n.t('not_found') }
    response.status = 404
  else
    error_object    = { error: I18n.t('something_went_wrong') }
    response.status = 500
  end

  response.write(error_object.to_json)
end
```

Now we are ready to write integration tests:

/spec/requests/api/v1/todos/show_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe 'GET /api/v1/todos/:id', type: :request do
  include_examples 'authorization check', 'get', '/api/v1/todos/21c9177e-9497-4c86-945b-7d1097c8865f'

  context 'when Authorization headers contains valid token' do
    let(:token) { access_token(user)        }
    let(:user)  { create(:user)             }
    let(:todo)  { create(:todo, user: user) }

    context 'when id is valid' do
      let(:todo_json_response) do
        {
          'id' => todo.id,
          'name' => todo.name,
          'description' => todo.description,
          'created_at' => todo.created_at.iso8601,
          'updated_at' => todo.updated_at.iso8601
        }
      end

      before do
        header 'Authorization', token

        get "/api/v1/todos/#{todo.id}"
      end

      it 'returns 200 HTTP status' do
        expect(response.status).to eq 200
      end

      it 'returns todo data in the JSON response' do
        expect(json_response).to eq todo_json_response
      end
    end

    context 'when id is invalid' do
      before do
        header 'Authorization', token

        get '/api/v1/todos/21c9177e-9497-4c86-945b-7d1097c8865f'
      end

      it 'returns 404 HTTP status' do
        expect(response.status).to eq 404
      end

      it 'returns error message in the JSON response' do
        expect(json_response).to eq({ 'error' => 'Record not found.' })
      end
    end

    context 'when todo belongs to different user' do
      let(:todo) { create(:todo) }

      before do
        header 'Authorization', token

        get "/api/v1/todos/#{todo.id}"
      end

      it 'returns 404 HTTP status' do
        expect(response.status).to eq 404
      end

      it 'returns error message in the JSON response' do
        expect(json_response).to eq({ 'error' => 'Record not found.' })
      end
    end
  end
end
```

- First, we use the authorization check shared example to test how our endpoint behaves when there is no access token in the Authorization header.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo id is invalid.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo belongs to a different user
- When todo id is valid API should return 200 HTTP status code and return particular todo in the JSON response.

Let’s see if tests are passing:

```
rspec
......................................................................................
....................................................

Finished in 0.47376 seconds (files took 1.93 seconds to load)
138 examples, 0 failures
```

Everything is green. In the next section, we will create an endpoint for updating existing todos.

8.4. Updating Todo

What if our user makes a typo in the name or description of a todo? For this purpose, we will add a `PUT /api/v1/todos/:id` endpoint that will allow us to update the existing todo in our system. Let's follow our pattern and create dedicated class for updating todos:

/app/services/todo/updater.rb

```
# frozen_string_literal: true

module Todos
  # {Todos::Updater} updates existing {Todo}.
  class Updater
    # @param [Todo] todo
    # @param [Hash] attributes of the {Todo}
    def initialize(todo:, attributes:)
      @todo       = todo
      @attributes = attributes
    end

    # It updates the existing {Todo} with new attributes.
    #
    # @return [Todo] object when attributes are valid.
    #
    # @raise [Sequel::ValidationFailed] when attributes are not valid.
    #
    # @example When attributes are valid:
    #   attributes = {name: 'Buy milk.', description: 'Please buy milk.'}
    #   Todos::Updater.new(todo: Todo.last, attributes: attributes).call
    #
    # @example When attributes are not valid:
    #   Todos::Updater.new(todo: Todo.last, attributes: {}).call
    def call
      @todo.update(
        name: @attributes[:name],
        description: @attributes[:description]
      )
    end
  end
end
```

- In the initialize method, we accept the `Todo` object which we want to update and new `Todo` attributes that we want to use in the updating process.
- The call method uses the `Todo.update` method provided by `Sequel::Model` that will raise `Sequel::ValidationFailed` when validation fails or return the updated `Todo` object when updating is successful.

Let's test our class to be sure it works as it should:

/spec/services/todo/updater_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe Todos::Updater do
  describe '#call' do
    let(:todo)   { create(:todo)                                                }
    let(:result) { described_class.new(todo: todo, attributes: attributes).call }

    context 'when attributes are valid' do
      let(:attributes) { { name: 'Buy cheese.', description: 'Please buy cheese.' } }

      it 'updates and returns todo' do
        expect(result)
          .to have_attributes(name: attributes[:name], description:
attributes[:description])
      end
    end

    context 'when attributes are invalid' do
      let(:attributes) { {} }

      it 'raise Sequel::ValidationFailed' do
        expect { result }.to raise_error(
          Sequel::ValidationFailed
        )
      end
    end
  end
end
```

In the `Todo::Updater` test, we check the following things:

- When attributes are valid, the `Todos::Updator` class updates and returns the `Todo` object.
- When attributes are invalid, the `Sequel::ValidationFailed` error is raised, and the `Todo` object is not updated.

Now let’s add endpoint to the routing tree:

app.rb

```ruby
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.on('todos') do
        # We are calling the current_user method to get the current user
        # from the authorization token that was passed in the Authorization header.
        current_user

        r.on(:uuid) do |id|
          todo = current_user.todos_dataset.with_pk!(id)

          ...

          r.put do
            todo_params = TodoParams.new.permit!(r.params)

            Todos::Updater.new(todo: todo, attributes: todo_params).call

            TodoSerializer.new(todo: todo).render
          end
        end

        ...
      end
    end
  end
end
```

- We again use here TodoParams class to validate the incoming params.
- Then, we use Todo::Updater to update the existing Todo object.
- After a successful update, we return the updated Todo object in the JSON format.

The last step is to write the integration test to make sure our endpoint works:

/spec/requests/api/v1/todos/update_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe 'PUT /api/v1/todos/:id', type: :request do
  include_examples 'authorization check', 'put', '/api/v1/todos/21c9177e-9497-4c86-945b-7d1097c8865f'

  context 'when Authorization headers contains valid token' do
    let(:token) { access_token(user) }
    let(:user)  { create(:user)      }

    context 'when id is valid' do
      let(:todo) { create(:todo, user: user) }

      before do
        header 'Authorization', token

        put "/api/v1/todos/#{todo.id}", params

        todo.refresh
      end

      context 'when request contains incorrectly formatted params' do
        let(:params) { {} }

        it 'returns 422 HTTP status' do
          expect(response.status).to eq 422
        end

        it 'returns error message in JSON response' do
          expect(json_response).to eq({ 'description' => ['is missing'], 'name' =>
['is missing'] })
        end
      end

      context 'when request params are valid' do
        let(:params) { { name: 'Buy cheese.', description: 'Please buy cheese.' } }

        let(:todo_json_response) do
          {
            'id' => todo.id,
            'name' => todo.name,
            'description' => todo.description,
            'created_at' => todo.created_at.iso8601,
            'updated_at' => todo.updated_at.iso8601
          }
        end

        it 'returns 200 HTTP status' do
          expect(response.status).to eq 200
        end

        it 'returns updated todo data in the JSON response' do
          expect(json_response).to eq todo_json_response
        end
      end
    end

    context 'when id is invalid' do
      before do
```

```
        header 'Authorization', token

        put '/api/v1/todos/21c9177e-9497-4c86-945b-7d1097c8865f'
      end

      it 'returns 404 HTTP status' do
        expect(response.status).to eq 404
      end

      it 'returns error message in the JSON response' do
        expect(json_response).to eq({ 'error' => 'Record not found.' })
      end
    end

    context 'when todo belongs to different user' do
      let(:todo) { create(:todo) }

      before do
        header 'Authorization', token

        put "/api/v1/todos/#{todo.id}"
      end

      it 'returns 404 HTTP status' do
        expect(response.status).to eq 404
      end

      it 'returns error message in the JSON response' do
        expect(json_response).to eq({ 'error' => 'Record not found.' })
      end
    end
  end
end
```

- First, we use the authorization check shared example to test how our endpoint behaves when there is no access token in the Authorization header.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo id is invalid.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo belongs to a different user
- When incoming parameters are invalid, API should return 422 HTTP status code and error in the JSON response.
- When incoming parameters are valid, API should return 200 HTTP status code and the updated `Todo` object in the JSON response.

Let's see if all tests are green.

```
rspec
......................................................................................
....................................................................

Finished in 0.53029 seconds (files took 2.72 seconds to load)
154 examples, 0 failures
```

They are. In the next section will work on creating the last endpoint for removing todos.

8.5. Todo deletion

In this section, we will create an endpoint for removing todos from our system, but don't worry, this is not the end of our journey :)

This will be a quick job. We've got everything we need. We only need to add a route to the routing tree and write integration tests.

app

```
route do |r|
  r.on('api') do
    r.on('v1') do
      ...

      r.on('todos') do
        # We are calling the current_user method to get the current user
        # from the authorization token that was passed in the Authorization header.
        current_user

        r.on(:uuid) do |id|
          todo = current_user.todos_dataset.with_pk!(id)

          ...

          r.delete do
            todo.delete

            response.write(nil)
          end
        end

        ...
      end
    end
  end
end
```

This one is super simple, we delete `Todo` using the `delete` method provided by `Sequel::Model`, and we return an empty JSON response. Let's test if it works:

/spec/requests/api/v1/todos/delete_spec.rb

```
# frozen_string_literal: true

require 'spec_helper'

describe 'DELETE /api/v1/todos/:id', type: :request do
  include_examples 'authorization check', 'delete', '/api/v1/todos/21c9177e-9497-4c86-
945b-7d1097c8865f'
```

```
context 'when Authorization headers contains valid token' do
  let(:token) { access_token(user)        }
  let(:user)  { create(:user)             }
  let(:todo)  { create(:todo, user: user) }

  context 'when id is valid' do
    before do
      header 'Authorization', token

      delete "/api/v1/todos/#{todo.id}"
    end

    it 'returns 200 HTTP status' do
      expect(response.status).to eq 200
    end

    it 'returns empty response body' do
      expect(response.body).to eq ''
    end

    it 'deletes todo' do
      expect(Todo.count).to eq 0
    end
  end

  context 'when id is invalid' do
    before do
      header 'Authorization', token

      delete '/api/v1/todos/21c9177e-9497-4c86-945b-7d1097c8865f'
    end

    it 'returns 404 HTTP status' do
      expect(response.status).to eq 404
    end

    it 'returns error message in the JSON response' do
      expect(json_response).to eq({ 'error' => 'Record not found.' })
    end
  end

  context 'when todo belongs to different user' do
    let(:todo) { create(:todo) }

    before do
      header 'Authorization', token

      delete "/api/v1/todos/#{todo.id}"
    end
```

```
      it 'returns 404 HTTP status' do
        expect(response.status).to eq 404
      end

      it 'returns error message in the JSON response' do
        expect(json_response).to eq({ 'error' => 'Record not found.' })
      end
    end
  end
end
```

Those tests scenarios are similar to the tests from the previous chapter:

- We use the authorization check shared example to test how our endpoint behaves when there is no access token in the Authorization header.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo id is invalid.
- We check that API should return 404 HTTP status code and error in the JSON response when the todo belongs to a different user
- When `Todo` deletion is successful, API should return 200 HTTP status with empty JSON response.

Final check to see if tests are passing:

```
rspec
......................................................................................
.................................................................................

Finished in 0.57766 seconds (files took 1.75 seconds to load)
167 examples, 0 failures
```

Everything works as expected. With that in place, we finished our work on building REST API with Roda & Sequel. In the following chapters, we will focus on securing our application from abusive requests and deploy it to Heroku.

Chapter 9. Security

Sites used by people often become the target of malicious activity, whether that be account enumeration attacks, brute-force login attempts, DDoS attacks, or worse. Aside from the obvious requirement to protect the potentially sensitive data your application deals with, it's also crucial that it's available to your users when they want to use it and not unavailable due to being flooded with requests.

9.1. Blocking abusive requests with Rack::Attack

`Rack::Attack` is a `rack` middleware intended to protect applications through customized throttling and blocking.

It means it's a component that sits between users and your application and is responsible for processing requests from these users and returning responses from your application back to them. In the case of `Rack::Attack`, it acts as a filter by comparing each request made to your application against a set of rules you define, either globally or for specific endpoints.

You can prevent attempts at blunt-forcing passwords by throttling requests with the email or username being attacked or prevent troublesome scrapers or other offenders by throttling requests from IP addresses, making large volumes of requests. Rack::Attack makes protecting your applications easy but still provides quite a bit of freedom to choose what to throttle, block, whitelist, or blacklist.

The first step is to add the required gems to the Gemfile and run the `bundle install` command. I've opted to use Redis as the cache store for `Rack::Attack`, so I need the `redis` and `rack-attack` gem to the Gemfile:

Gemfile

```ruby
# Rack middleware for blocking & throttling.
gem 'rack-attack'

# Redis is an in-memory database that persists on disk.
gem 'redis'
```

Before we start configuring `rack-attack`, we need to check that Redis is up and running on our machine. We can use the `redis-cli` command for that:

```
$ redis-cli
127.0.0.1:6379>
```

If Redis is not installed on your machine, you can find instructions on how to do this here.

We will connect to our Redis instance using URL, so let's add the informations about the Redis URL to our environment files:

env.development.template

```
REDIS_URL=redis://127.0.0.1:6379
```

env.test.template

```
REDIS_URL=redis://127.0.0.1:6379
```

env.development

```
REDIS_URL=redis://127.0.0.1:6379
```

env.test

```
REDIS_URL=redis://127.0.0.1:6379
```

We also need a Redis instance in our Continous Integration process, so let's update: `.github/workflows/ci.yml` file:

github/workflows/ci.yml

```
    services:
      postgres:
        image: postgres:latest
        env:
          POSTGRES_USER: postgres
          POSTGRES_DB: todo-api-test
          POSTGRES_PASSWORD: postgres
        ports: ["5432:5432"]
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

      redis:
        image: redis
        ports: ['6379:6379']
        options: >-
          --health-cmd "redis-cli ping"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
```

The next step is to create a `dry-system` component that will be responsible for setting up Redis connection:

/system/boot/redis.rb

```ruby
# frozen_string_literal: true

# This file contain code to setup the redis connection.

Application.boot(:redis) do |container|
  # Load environment variables before setting up redis connection.
  use :environment_variables

  init do
    require 'redis'
  end

  start do
    # Define Redis instance.
    redis = Redis.new(url: ENV['REDIS_URL'])

    # Register redis component.
    container.register(:redis, redis)
  end
end
```

Before setting up a Redis connection, we need to have access to environment variables which store our `REDIS_URL`. Because of that, we write `use :environment_variables` at the beginning of our component to auto-boot the required dependency and make it available in the booting context. Then we require the `redis` gem and define a new Redis instance: `Redis.new(url: ENV['REDIS_URL'])`. The last thing is registering our component, so we will access it from the `Application` object: `Application['redis']`.

With the required gems installed and Redis up and running, the next step is to define the rules to be used by Rack::Attack in processing requests. Here is a list of rules that we want to implement with `rack-attack`:

- Limit POST requests to `/api/v1/login` to 10 requests every 60 seconds for one IP address.
- Limit POST requests to `/api/v1/login` to 10 requests every 60 seconds for one e-mail address.
- Limit POST requests to `/api/v1/sign_up` to 10 requests every 60 seconds for one IP address.
- Limit requests to all endpoints to 20 requests every 20 seconds for one IP address.

So let’s define a new component that will implement those rules:

/system/boot/rack_attack.rb

```ruby
# frozen_string_literal: true

# This file contains configuration for rack-attack.

Application.boot(:rack_attack) do
  init do
```

```
    require 'rack/attack'
  end

  start do |app|
    # Configure Redis cache.
    Rack::Attack.cache.store = Rack::Attack::StoreProxy::RedisStoreProxy
.new(app[:redis])

    # Throttle POST requests to /api/v1/login by IP address.
    #
    # Key: "rack::attack:#{Time.now.to_i/:period}:logins/ip:#{req.ip}".
    #
    # The most common brute-force login attack is a brute-force password
    # attack where an attacker simply tries a large number of emails and
    # passwords to see if any credentials match.
    #
    # Another common method of attack is to use a swarm of computers with
    # different IPs to try brute-forcing a password for a specific account.
    Rack::Attack.throttle('/logins/ip', limit: 10, period: 60) do |req|
      req.ip if req.path == '/api/v1/login' && req.post?
    end

    # Throttle POST requests to /api/v1/login by email param.
    #
    # Key: "rack::attack:#{Time.now.to_i/:period}:logins/email:#{normalized_email}".
    #
    # This creates a problem where a malicious user could intentionally
    # throttle logins for another user and force their login requests to be
    # denied, but that's not very common and shouldn't happen to you.
    Rack::Attack.throttle('/logins/email', limit: 10, period: 60) do |req|
      if req.path == '/api/v1/login' && req.post? && req.params['email']
        req.params['email'].to_s.downcase.gsub(/\s+/, '').presence
      end
    end

    # Throttle POST requests to /api/v1/sign_up by IP address.
    #
    # Key: "rack::attack:#{Time.now.to_i/:period}:sign_up/ip:#{req.ip}".
    #
    # During an attack, the hackers bots will typically sign up with a random email
then do something bad,
    # hundreds of times a minute, from a relatively small number of computers.
    # The judicious use of endpoint-based request restriction can prevent your site
from being an attractive
    # target for spammers and hackers. It can also reduce the size of any successful
bot attack by
    # limiting the amount of possible signups.
    # In this example, hackers can only add up to three users every quarter of an hour
    Rack::Attack.throttle('sign_up/ip', limit: 3, period: 900) do |req|
      req.ip if req.path == '/api/v1/sign_up' && req.post?
    end
    # Throttle all requests by IP (60rpm).
    #
    # Key: "rack::attack:#{Time.now.to_i/:period}:req/ip:#{req.ip}".
    #
    # If any single client IP is making tons of requests, then they're
    # probably malicious or a poorly-configured scraper. Either way, they
    # don't deserve to hog all of the app server's CPU. Cut them off!
    Rack::Attack.throttle('req/ip', limit: 20, period: 20, &:ip)

    # Allow all requests from localhost.
    Rack::Attack.safelist('allow from localhost') do |req|
      req.ip == '127.0.0.1' || req.ip == '::1'
    end
  end
end
```

- First we require `rack/attack` gem.
- In the `start` block we first configure Redis cache: `Rack::Attack.cache.store = Rack::Attack::StoreProxy::RedisStoreProxy.new(app[:redis])`
- We define rules that we described above using `Rack::Attack.throttle` method.
- The last step is to allow all requests from the localhost using `Rack::Attack.safelist` method.

To enable `Rack::Attack` we need add `use Rack::Attack` to our `config.ru` file:

config.ru

```
# frozen_string_literal: true

# This file contains configuration to let the webserver which application to run.

require_relative 'app'

use Rack::Attack

run App.freeze.app
```

With that in place, we are ready to write tests that will check if `Rack::Attack` secures our API as expected. The most important thing in the tests is clearing the `Rack::Attack` cache between spec examples. For that, we will create the `RedisHelpers` module in `/spec/support` file:

/spec/support/redis_helpers.rb

```
# frozen_string_literal: true

# {RedisHelpers} module contains helper methods that clean Redis between RSpec
examples.
module RedisHelpers
  # It returns Redis client instance.
  def redis
    @redis ||= Application[:redis]
  end

  # It calls with_clean_redis command around RSpec examples.
  def self.included(rspec)
    rspec.around do |example|
      with_clean_redis do
        example.run
      end
    end
  end

  # It cleans Redis using flushall command.
  def with_clean_redis
    redis.flushall
    begin
      yield
    ensure
      redis.flushall
    end
  end
end

RSpec.configure do |config|
  config.include RedisHelpers, type: :throttling
end
```

In `RedisHelpers` we have three simple methods:

- `redis` method gets the Redis client instance from the `Application` object:
- `with_clean_redis` methods cleans Redis using `flushall` command.
- `self.included` method calls `with_clean_redis` command around RSpec examples.

We want to include the `RedisHelpers` module only for specs with type `throttling`. This is because we don't want a clean Redis state in tests where we do not use Redis because this will slow down our specs.

Another thing before writing tests is that we need to update the `app` method in the `spec/support/rack_test.rb` to parse whole `config.ru` instead using `Rack::Builder.parse_file` command:

/spec/support/rack_test.rb

```ruby
def app
  Rack::Builder.parse_file('config.ru').first
end
```

Now we are ready to write the tests:

/spec/requests/api/v1/throttling_spec.rb

```ruby
# frozen_string_literal: true

require 'spec_helper'

describe 'Throttling', type: :throttling do
  describe 'POST requests to /api/v1/login by IP address' do
    before do
      request_count.times do |i|
        # We increment the email address here so we can be sure that it's
        # the IP address and not email address that's being blocked.
        params = { email: "sample#{i}@example.com", password: 'password' }

        post '/api/v1/login', params, 'REMOTE_ADDR' => '1.2.3.4'
      end
    end

    context 'when number of requests is lower than the limit' do
      let(:request_count) { 10 }

      it 'does not change the request status' do
        expect(response.status).not_to eq(429)
      end
    end

    context 'when number of requests is higher than the limit' do
      let(:request_count) { 11 }

      it 'changes the request status to 429' do
        expect(response.status).to eq(429)
      end
    end
  end

  describe 'POST requests to /api/v1/login by email param' do
    before do
      request_count.times do |i|
        # This time we increment the IP address so we can be sure that
        # it's the email address and not the IP address that's being blocked.
        params = { email: 'sample@example.com', password: 'password' }
        post '/api/v1/login', params, 'REMOTE_ADDR' => "1.2.3.#{i}"
      end
    end

    context 'when number of requests is lower than the limit' do
      let(:request_count) { 10 }

      it 'does not change the request status' do
        expect(response.status).not_to eq(429)
      end
    end

    context 'when number of requests is higher than the limit' do
      let(:request_count) { 11 }

      it 'changes the request status to 429' do
        expect(response.status).to eq(429)
      end
    end
  end

  describe 'POST requests to /api/v1/sign_up by IP address' do
    before do
      request_count.times do |i|
        params = { email: "sample#{i}@example.com", password: 'password',
password_confirmation: 'password' }

        post '/api/v1/sign_up', params, 'REMOTE_ADDR' => '1.2.3.4'
      end
    end

    context 'when number of requests is lower than the limit' do
      let(:request_count) { 3 }

      it 'does not change the request status' do
        expect(response.status).not_to eq(429)
      end
    end

    context 'when number of requests is higher than the limit' do
      let(:request_count) { 4 }

      it 'changes the request status to 429' do
        expect(response.status).to eq(429)
      end
    end
  end

  describe 'Throttle all requests by IP (60rpm).' do
    before do
      request_count.times do
```

```
        get '/api/v1/todos', {}, 'REMOTE_ADDR' => '1.2.3.4'
      end
    end

    context 'when number of requests is lower than the limit' do
      let(:request_count) { 20 }

      it 'does not change the request status' do
        expect(response.status).not_to eq(429)
      end
    end

    context 'when number of requests is higher than the limit' do
      let(:request_count) { 21 }

      it 'changes the request status to 429' do
        expect(response.status).to eq(429)
      end
    end
  end
end
```

Let's review what is going on here:

- First, we test POST requests to `/api/v1/login` endpoint by IP address. When a number of requests are lower than the limit, API should not return 429 HTTP status code. Otherwise, 429 status should be returned.
- In the following scenario, we check POST requests to `/api/v1/login` endpoint by email param. If the limit of requests is exceeded, API should return 429 HTTP status code. Otherwise, 429 status should not be returned.
- We also check POST requests to `/api/v1/sign_up` endpoint by IP address. When a number of requests are lower than the limit, API should not return 429 HTTP status code. Otherwise, 429 status should be returned.
- The last check is checking all requests by IP address. If more than 20 requests hit our API in 20 seconds, 429 HTTP status code should be returned. Otherwise, requests should not be blocked.

Let's see if our tests are passing:

```
rspec
......................................................................................
......................................................................................
...
```

Everything is green. There is one last thing we need to fix before finishing our work. If we run the `rubocop` command, we will see the following response:

```
rubocop

Inspecting 86 files
......................................................C.............................C.

Offenses:

spec/requests/api/v1/throttling_spec.rb:13:56: C: Style/IpAddresses: Do not hardcode
IP addresses.
        post '/api/v1/login', params, 'REMOTE_ADDR' => '1.2.3.4'
                                                       ^^^^^^^^^
spec/requests/api/v1/throttling_spec.rb:67:58: C: Style/IpAddresses: Do not hardcode
IP addresses.
        post '/api/v1/sign_up', params, 'REMOTE_ADDR' => '1.2.3.4'
                                                         ^^^^^^^^^
spec/requests/api/v1/throttling_spec.rb:91:51: C: Style/IpAddresses: Do not hardcode
IP addresses.
        get '/api/v1/todos', {}, 'REMOTE_ADDR' => '1.2.3.4'
                                                  ^^^^^^^^^
system/boot/rack_attack.rb:66:17: C: Style/IpAddresses: Do not hardcode IP addresses.
      req.ip == '127.0.0.1' || req.ip == '::1'
                ^^^^^^^^^^^
system/boot/rack_attack.rb:66:42: C: Style/IpAddresses: Do not hardcode IP addresses.
      req.ip == '127.0.0.1' || req.ip == '::1'
                                         ^^^^^

86 files inspected, 5 offenses detected
```

We are breaking here Style/IpAddresses Rubocop rule. I decided to disable this rule in those two files:

.rubocop.yml

```
Style/IpAddresses:
  Exclude:
    - system/boot/rack_attack.rb
    - spec/requests/api/v1/throttling_spec.rb
```

And that's pretty much it. With just a little work, my app is now relatively well protected against misbehaving clients. Suppose I notice any apparent patterns of suspicious behavior in the future. We have the flexibility to lock the app down further by simply adding the appropriate rules.

Chapter 10. Deployment

Heroku is a platform that enables developers to build, run, and operate applications entirely on the cloud with multi-language support for Ruby, Go, Scala, PHP, etc. In this chapter, we will deploy our application to the Heroku server.

10.1. Deploying to Heroku

Our server of choice is Puma. Puma uses threads, in addition to worker processes, to make more use of available CPU. You can only utilize threads in Puma if your entire code-base is thread safe. Otherwise, you can still use Puma, but must only scale-out through worker processes. First thing we need to do is to create config file for Puma that will be used by Heroku:

/config/puma.rb

```ruby
# frozen_string_literal: true

# This file contains puma web server configuration.

# Puma forks multiple OS processes within each dyno to allow a Rails app to support multiple concurrent requests.
# In Puma terminology, these are referred to as worker processes
# (not to be confused with Heroku worker processes which run in their dynos).
# Worker processes are isolated from one another at the OS level, therefore not needing to be thread-safe
workers Integer(ENV['WEB_CONCURRENCY'] || 1)

# Puma can serve each request in a thread from an internal thread pool.
# This behavior allows Puma to provide additional concurrency for your web application.
# Loosely speaking, workers consume more RAM and threads consume more CPU, and both offer more concurrency.
threads_count = Integer(ENV['MAX_THREADS'] || 5)
threads threads_count, threads_count

# Preloading your application reduces the startup time of individual Puma worker processes and
# allows you to manage the external connections of each worker using the on_worker_boot calls.
# In the config above, these calls are used to establish Postgres connections for each worker process correctly.
preload_app!

# Heroku will set ENV['PORT'] when the web process boots up. Locally, default this to 3000 to match the Rails default.
port ENV['PORT'] || 3000

# Set the environment of Puma. On Heroku ENV['RACK_ENV'] will be set to 'production' by default.
environment ENV['RACK_ENV'] || 'development'

# *Cluster mode only* Code to run immediately before master process forks workers (once on boot).
# These hooks can block if necessary to wait for
# background operations unknown to puma to finish before the process terminates.
# This can be used to close any connections to remote servers (database, redis)
# that were opened when preloading the code.
# This can be called multiple times to add hooks.
before_fork do
  Sequel::Model.db.disconnect if defined?(Sequel::Model)
end
```

Let’s go through what’s going on here:

- First, we set a number of workers to 1. Each worker process used consumes additional memory. This behavior limits how many processes you can run in a single dyno. With a typical Rails memory footprint, you can expect to run 2-4 Puma worker processes on a `free`, `hobby` or `standard-1x` dyno.
- Next, we set the number of threads. Puma allows you to configure your thread pool with a min and max setting, controlling the number of threads each Puma instance uses.
- `preload_app!` reduces the startup time of individual Puma worker processes and allows you to manage the external connections of each worker using the `on_worker_boot` calls.
- Heroku will set `ENV['PORT']` when the web process boots up. Locally, default this to `3000` to match the Ruby on Rails default.
- We set the environment of Puma. On Heroku `ENV['RACK_ENV']` will be set to 'production' by default.
- In the `before_fork` we put code to run immediately before master process forks workers. In our case, we use this to close connection to our database using `Sequel::Model.db.disconnect`.

The next thing we need to do before the deployment to Heroku is adding `Procfile` file. The `Procfile` specifies the commands that are executed by the app on startup. You can use a `Procfile` to declare a variety of process types, including:

- Your app's web server.
- Multiple types of worker processes.
- A singleton process, such as a clock.
- Tasks to run before a new release is deployed.

Procfile

```
web: bundle exec puma -C config/puma.rb
release: rake db:migrate
```

In our Procfile, we have to process types:

- A Heroku app's `web` process type is special: it's the only process type that can receive external HTTP traffic from Heroku's routers. If your app includes a web server, you should declare it as your app's web process.
- The `release` process type is used to specify the command to run during your app's release phase. We want to run migrations of our database during release.

We also need to add `x86_64-linux` platform to the `Gemfile.lock` to be able to run `bundle install` on Heroku, we can do that with `bundle lock` command:

```
bundle lock  --add-platform x86_64-linux
```

Ok, we've got everything we need to start configuring our Heroku server. Before going further, make sure that you have heroku-cli installed and configured.

Make sure you are in the directory that contains our app, then create an app on Heroku:

```
heroku create sequel-roda-json-todo-api
https://sequel-roda-json-todo-api.herokuapp.com/ | https://git.heroku.com/sequel-roda-json-todo-api.git
```

You can verify that the remote was added to your project by running:

```
git config --list | grep heroku
remote.heroku.url=https://git.heroku.com/sequel-roda-json-todo-api.git
remote.heroku.fetch=+refs/heads/*:refs/remotes/heroku/*
```

Next we need to add PostgreSQL addon:

```
heroku addons:create heroku-postgresql:hobby-dev --app sequel-roda-json-todo-api
```

The last addon is Redis that is required for `Rack::Attack` cache:

```
heroku addons:create heroku-redis:hobby-dev --app sequel-roda-json-todo-api
```

Last step is to add `SECRET_KEY_BASE` environment variable that will be used to sign in messages generated by `MessageVerifier` class:

```
heroku config:set SECRET_KEY_BASE=274259656c6bf25bd9f48ecec253523348f7c44973198b86a42bcf0b47cdb3064a7d52b8a08559e2
```

Ok we are ready to deploy our application:

```
git push heroku
Enumerating objects: 313, done.
Counting objects: 100% (313/313), done.
Delta compression using up to 16 threads
Compressing objects: 100% (270/270), done.
Writing objects: 100% (313/313), 65.84 KiB | 732.00 KiB/s, done.
Total 313 (delta 112), reused 0 (delta 0), pack-reused 0
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Building on the Heroku-20 stack
remote: -----> Determining which buildpack to use for this app
remote: -----> Ruby app detected
remote: -----> Installing bundler 2.2.16
remote: -----> Removing BUNDLED WITH version in the Gemfile.lock
```

```
remote: -----> Compiling Ruby/Rack
remote: -----> Using Ruby version: ruby-3.0.0
remote: -----> Installing dependencies using bundler 2.2.16
remote:        Running: BUNDLE_WITHOUT='development:test' BUNDLE_PATH=vendor/bundle
BUNDLE_BIN=vendor/bundle/bin BUNDLE_DEPLOYMENT=1 bundle install -j4
remote:        Fetching gem metadata from https://rubygems.org/.......
remote:        Fetching rake 13.0.3
remote:        Installing rake 13.0.3
remote:        Fetching concurrent-ruby 1.1.8
remote:        Fetching minitest 5.14.4
remote:        Fetching bcrypt 3.1.16
remote:        Fetching zeitwerk 2.4.2
remote:        Installing minitest 5.14.4
remote:        Installing zeitwerk 2.4.2
remote:        Installing bcrypt 3.1.16 with native extensions
remote:        Installing concurrent-ruby 1.1.8
remote:        Using bundler 2.2.16
remote:        Fetching coderay 1.1.3
remote:        Fetching dry-equalizer 0.3.0
remote:        Installing dry-equalizer 0.3.0
remote:        Installing coderay 1.1.3
remote:        Fetching dry-inflector 0.2.0
remote:        Installing dry-inflector 0.2.0
remote:        Fetching dry-initializer 3.0.4
remote:        Fetching ice_nine 0.11.2
remote:        Installing dry-initializer 3.0.4
remote:        Installing ice_nine 0.11.2
remote:        Fetching method_source 1.0.0
remote:        Installing method_source 1.0.0
remote:        Fetching nio4r 2.5.5
remote:        Fetching oj 3.11.2
remote:        Fetching tty-color 0.6.0
remote:        Installing tty-color 0.6.0
remote:        Installing nio4r 2.5.5 with native extensions
remote:        Installing oj 3.11.2 with native extensions
remote:        Fetching pg 1.2.3
remote:        Installing pg 1.2.3 with native extensions
remote:        Fetching rack 2.2.3
remote:        Installing rack 2.2.3
remote:        Fetching redis 4.2.5
remote:        Installing redis 4.2.5
remote:        Fetching sequel 5.42.0
remote:        Installing sequel 5.42.0
remote:        Fetching timecop 0.9.4
remote:        Installing timecop 0.9.4
remote:        Fetching yard 0.9.26
remote:        Installing yard 0.9.26
remote:        Fetching pry 0.14.0
remote:        Installing pry 0.14.0
remote:        Fetching i18n 1.8.9
remote:        Installing i18n 1.8.9
```

```
remote:        Fetching tzinfo 2.0.4
remote:        Installing tzinfo 2.0.4
remote:        Fetching dry-core 0.5.0
remote:        Installing dry-core 0.5.0
remote:        Fetching pastel 0.8.0
remote:        Installing pastel 0.8.0
remote:        Fetching rack-attack 6.5.0
remote:        Installing rack-attack 6.5.0
remote:        Fetching roda 3.42.0
remote:        Installing roda 3.42.0
remote:        Fetching sequel_secure_password 0.2.15
remote:        Installing sequel_secure_password 0.2.15
remote:        Fetching activesupport 6.1.3
remote:        Installing activesupport 6.1.3
remote:        Fetching dry-configurable 0.12.1
remote:        Installing dry-configurable 0.12.1
remote:        Fetching dry-logic 1.1.0
remote:        Installing dry-logic 1.1.0
remote:        Fetching tty-logger 0.6.0
remote:        Installing tty-logger 0.6.0
remote:        Fetching dry-container 0.7.2
remote:        Installing dry-container 0.7.2
remote:        Fetching roda-enhanced_logger 0.4.0
remote:        Installing roda-enhanced_logger 0.4.0
remote:        Fetching dry-auto_inject 0.7.0
remote:        Installing dry-auto_inject 0.7.0
remote:        Fetching dry-types 1.5.1
remote:        Installing dry-types 1.5.1
remote:        Fetching dry-schema 1.6.1
remote:        Installing dry-schema 1.6.1
remote:        Fetching dry-struct 1.4.0
remote:        Installing dry-struct 1.4.0
remote:        Fetching dry-validation 1.6.0
remote:        Installing dry-validation 1.6.0
remote:        Fetching dry-system 0.18.1
remote:        Installing dry-system 0.18.1
remote:        Fetching puma 5.2.1
remote:        Installing puma 5.2.1 with native extensions
remote:        Fetching sequel_pg 1.14.0
remote:        Installing sequel_pg 1.14.0 with native extensions
remote:        Bundle complete! 28 Gemfile dependencies, 43 gems now installed.
remote:        Gems in the groups 'development' and 'test' were not installed.
remote:        Bundled gems are installed into `./vendor/bundle`
remote:        Bundle completed (36.13s)
remote:        Cleaning up the bundler cache.
remote: -----> Writing config/database.yml to read from DATABASE_URL
remote: -----> Detecting rake tasks
remote:
remote: ###### WARNING:
remote:
remote:        There is a more recent Ruby version available for you to use:
```

```
remote:
remote:        3.0.1
remote:
remote:        The latest version will include security and bug fixes. We always
recommend
remote:        running the latest version of your minor release.
remote:
remote:        Please upgrade your Ruby version.
remote:
remote:        For all available Ruby versions see:
remote:          https://devcenter.heroku.com/articles/ruby-support#supported-runtimes
remote:
remote:
remote: -----> Discovering process types
remote:        Procfile declares types     -> release, web
remote:        Default types for buildpack -> console, rake
remote:
remote: -----> Compressing...
remote:        Done: 30.8M
remote: -----> Launching...
remote:  !     Release command declared: this new release will not be available until
the command succeeds.
remote:        Released v8
remote:        https://sequel-roda-json-todo-api.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
remote: Running release command...
remote:
remote: I, [2021-05-17T12:38:58.938274 #15]  INFO -- : (0.001952s) SELECT
CAST(current_setting('server_version_num') AS integer) AS v
remote: I, [2021-05-17T12:38:58.947783 #15]  INFO -- : (0.008786s) CREATE TABLE IF NOT
EXISTS "schema_info" ("version" integer DEFAULT 0 NOT NULL)
remote: I, [2021-05-17T12:38:58.949452 #15]  INFO -- : (0.001319s) SELECT * FROM
"schema_info" LIMIT 1
remote: I, [2021-05-17T12:38:58.953065 #15]  INFO -- : (0.003305s) SELECT 1 AS "one"
FROM "schema_info" LIMIT 1
remote: I, [2021-05-17T12:38:58.956344 #15]  INFO -- : (0.002872s) SELECT
pg_attribute.attname AS pk FROM pg_class, pg_attribute, pg_index, pg_namespace WHERE
pg_class.oid = pg_attribute.attrelid AND pg_class.relnamespace = pg_namespace.oid AND
pg_class.oid = pg_index.indrelid AND pg_index.indkey[0] = pg_attribute.attnum AND
pg_index.indisprimary = 't' AND pg_class.oid = CAST(CAST('"schema_info"' AS regclass)
AS oid)
remote: I, [2021-05-17T12:38:58.959209 #15]  INFO -- : (0.002414s) INSERT INTO
"schema_info" ("version") VALUES (0) RETURNING NULL
remote: I, [2021-05-17T12:38:58.961262 #15]  INFO -- : (0.001373s) SELECT count(*) AS
"count" FROM "schema_info" LIMIT 1
remote: I, [2021-05-17T12:38:58.962621 #15]  INFO -- : (0.001121s) SELECT "version"
FROM "schema_info" LIMIT 1
remote: I, [2021-05-17T12:38:58.965307 #15]  INFO -- : Begin applying migration
version 1, direction: up
remote: I, [2021-05-17T12:38:58.966802 #15]  INFO -- : (0.001348s) BEGIN
```

```
remote: I, [2021-05-17T12:38:58.992704 #15]  INFO -- : (0.025666s)       CREATE
EXTENSION IF NOT EXISTS "uuid-ossp";
remote:       CREATE EXTENSION IF NOT EXISTS "citext";
remote:
remote: I, [2021-05-17T12:38:58.994776 #15]  INFO -- : (0.001632s) UPDATE
"schema_info" SET "version" = 1
remote: I, [2021-05-17T12:38:59.003593 #15]  INFO -- : (0.008481s) COMMIT
remote: I, [2021-05-17T12:38:59.003999 #15]  INFO -- : Finished applying migration
version 1, direction: up, took 0.038689 seconds
remote: I, [2021-05-17T12:38:59.004229 #15]  INFO -- : Begin applying migration
version 2, direction: up
remote: I, [2021-05-17T12:38:59.005901 #15]  INFO -- : (0.001287s) BEGIN
remote: I, [2021-05-17T12:38:59.036234 #15]  INFO -- : (0.023781s) CREATE TABLE
"users" ("id" uuid DEFAULT uuid_generate_v4() NOT NULL PRIMARY KEY, "email" citext NOT
NULL UNIQUE, "password_digest" text NOT NULL, "authentication_token" text NOT NULL
UNIQUE, "created_at" timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL, "updated_at"
timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL)
remote: I, [2021-05-17T12:38:59.039308 #15]  INFO -- : (0.002787s) UPDATE
"schema_info" SET "version" = 2
remote: I, [2021-05-17T12:38:59.046310 #15]  INFO -- : (0.006730s) COMMIT
remote: I, [2021-05-17T12:38:59.046470 #15]  INFO -- : Finished applying migration
version 2, direction: up, took 0.042235 seconds
remote: I, [2021-05-17T12:38:59.046521 #15]  INFO -- : Begin applying migration
version 3, direction: up
remote: I, [2021-05-17T12:38:59.059121 #15]  INFO -- : (0.012453s) BEGIN
remote: I, [2021-05-17T12:38:59.069829 #15]  INFO -- : (0.009719s) CREATE TABLE
"todos" ("id" uuid DEFAULT uuid_generate_v4() NOT NULL PRIMARY KEY, "name" text NOT
NULL, "description" text NOT NULL, "created_at" timestamp DEFAULT CURRENT_TIMESTAMP
NOT NULL, "updated_at" timestamp DEFAULT CURRENT_TIMESTAMP NOT NULL, "user_id" uuid
NOT NULL REFERENCES "users" ON DELETE CASCADE)
remote: I, [2021-05-17T12:38:59.075917 #15]  INFO -- : (0.005666s) UPDATE
"schema_info" SET "version" = 3
remote: I, [2021-05-17T12:38:59.079352 #15]  INFO -- : (0.003255s) COMMIT
remote: I, [2021-05-17T12:38:59.079505 #15]  INFO -- : Finished applying migration
version 3, direction: up, took 0.032976 seconds
remote: Waiting for release.... done.
To https://git.heroku.com/sequel-roda-json-todo-api.git
 * [new branch]      main -> main
```

That’s it. Our application was successfully deployed. Let’s test if we can create an account:

```
curl --location --request POST 'https://sequel-roda-json-todo-
api.herokuapp.com/api/v1/sign_up' \
> --form 'email="user@test.com"' \
> --form 'password="password"' \
> --form 'password_confirmation="password"'

{"user":{"id":"f193f999-2900-46c9-a625-7a4d6bf55e1b","email":"user@test.com"
,"created_at":"2021-05-17T12:40:21+00:00","updated_at":"2021-05-17T12:40:21+00:00"
},"tokens":{"access_token":{"token":"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsO
XBaRWtpS1dZeE9UTm1PVGs1TFRJNU1EQXRORFpqT1MxaE5qSTFMVGRoTkdRMlltWTFOV1V4WWdZNkJrVlVPaGx
oZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYmtraVZXTm1aV1ZsWlRNeE1HSXlOMk5pWkRoaVltUmhNbU16WkRVM
04ySmtOV1EwTXpka1ltTmhaamd4TWpKbU1qbG1ZV0kyT0dJeE5EUmlNalJrWldRMU56RTNPVFU0Tm1FMVltWmh
NbU13TnpobEJqc0dWQT09IiwiZXhwIjoiMjAyMS0wNS0xN1QxMjo0NToyMVoiLCJwdXIiOiJhY2Nlc3NfdG9rZ
W4ifX0=--
eef3e13ce7cbab0c89ea948306dcb187bcd683e57cdceed8e680fb1b15b9945de19c548d4793bd4f3b3426
c8dfbcee07b5e077d0bafd21fe2e0ce04f15fc02b3","expires_in":300},"refresh_token":{"token"
:"eyJfcmFpbHMiOnsibWVzc2FnZSI6IkJBaDdCem9NZFhObGNsOXBaRWtpS1dZeE9UTm1PVGs1TFRJNU1EQXRO
RFpqT1MxaE5qSTFMVGRoTkdRMlltWTFOV1V4WWdZNkJrVlVPaGxoZFhSb1pXNTBhV05oZEdsdmJsOTBiMnRsYm
traVZXTm1aV1ZsWlRNeE1HSXlOMk5pWkRoaVltUmhNbU16WkRVM04ySmtOV1EwTXpka1ltTmhaamd4TWpKbU1q
bG1ZV0kyT0dJeE5EUmlNalJrWldRMU56RTNPVFU0Tm1FMVltWmhNbU13TnpobEJqc0dWQT09IiwiZXhwIjoiMj
AyMS0wNS0xN1QxMjo1NToyMVoiLCJwdXIiOiJyZWZyZXNoX3Rva2VuIn19--
9b47e08779e7c568b9a061d94cb15caa9d70ec0f7ef6ba92781dfc223a4a1ab5c44add88dcc8a46900f15d
888e7bad4be4b87c93400d6e00e907186fecefd6c6","expires_in":900}}}
```

Congratulations! You have deployed your first Roda application to Heroku.

The icing on the cake will be generating code documentation using the `rake docs` command. After that, `/doc` folder will be created that will contain documentation generated by yard.

Chapter 11. Epilogue

So there you have it, a Roda application. Hopefully, this has been a good demonstration about organizing a Roda application in a better fashion than what’s given to us by default. The separation between classes makes this application a little more tedious to setup than a traditional Ruby on Rails application but lends itself to future maintenance. I hope you enjoyed reading this book. If you have some ideas or thoughts about the book, do not hesitate to contact me.

Happy Hacking!