Table of Contents

Introduction 1.1

Core Concepts

Application Design 2.1
Sinatra API Project 2.2
RESTful API 2.3
What is OOP 2.4
Ruby on Rails 2.5

Introduction to Ruby on Rails

Active Record 3.1
Action Controller 3.2
Action View 3.3

Authentication

Session Based Authentication 4.1
JWT Authentication 4.2
Authenticate with OAuth 4.3
Security 4.4

Authorisation

Design Roles From Scratch 5.1
Separation of roles with Nested Controllers 5.2

Rails API

API only Rails Application 6.1
Best Practices for API Design 6.2

Rails Mailers

Action Mailer 7.1
Action Mailbox 7.2

External API

Daraja API 8.1
Paystack API 8.2

Rails is Underrated

Pre-requisites

You should at-least have some experience with Ruby on Rails before reading this handbook!

1. Clone repository

Check out!

```
https://github.com/Rails-is-Underrated
```

2. Installation

1. Install Node.JS LTS Version
2. Install Honkit with NPM

```
npm install
```

1. Preview and serve your book using:

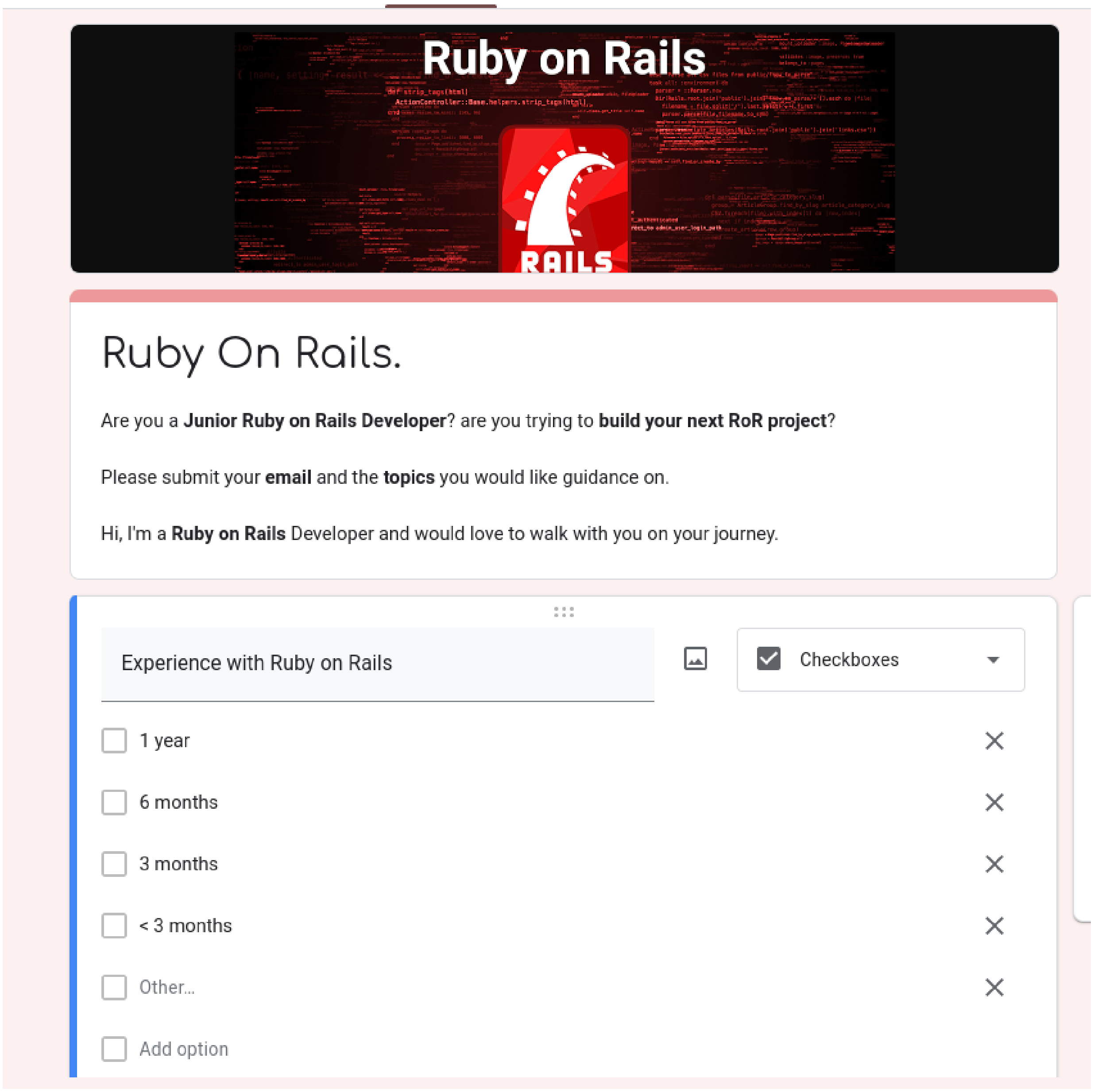
```
npx honkit serve
```

3. Modify existing document

This handbook uses Honkit and Markdown. To continue with editing, you may refer to Honkit documentation

4. Inspiration

Based on a google form results



Ruby On Rails.

Are you a **Junior Ruby on Rails Developer**? are you trying to **build your next RoR project**?

Please submit your **email** and the **topics** you would like guidance on.

Hi, I'm a **Ruby on Rails** Developer and would love to walk with you on your journey.

Experience with Ruby on Rails

Checkboxes

- ☐ 1 year
- ☐ 6 months
- ☐ 3 months
- ☐ < 3 months
- ☐ Other...
- ☐ Add option

Introduction.

Duka is an E-commerce Application, it provide an interface for buyers and sellers.

Objectives.

Make it easier for buyers to shop for products.

Make it easier for sellers to sell and monitor their products.

Requirements.

Functional

- Account creation for buyers and sellers
- Product creation.
- Managing orders.
- Complete purchases.

Non-functional

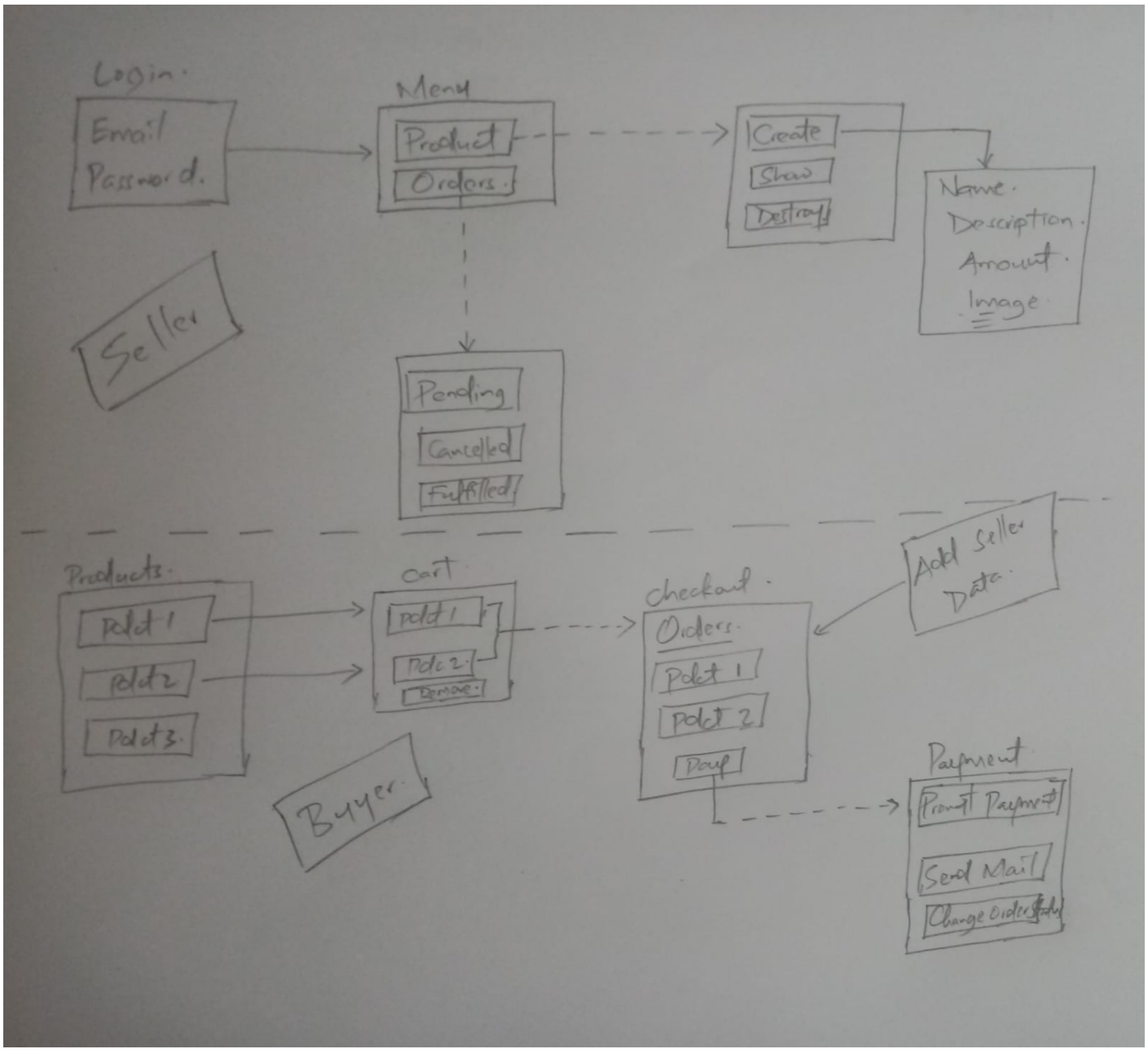
- Portability (Mobile first).
- Security.
- Intuitive (Easy to use).

User Stories

sellers	buyers
Create products to sell	Browse on product to buy
Determine status of orders	Select product to purchase
Complete orders	Provide metadata when completing an order

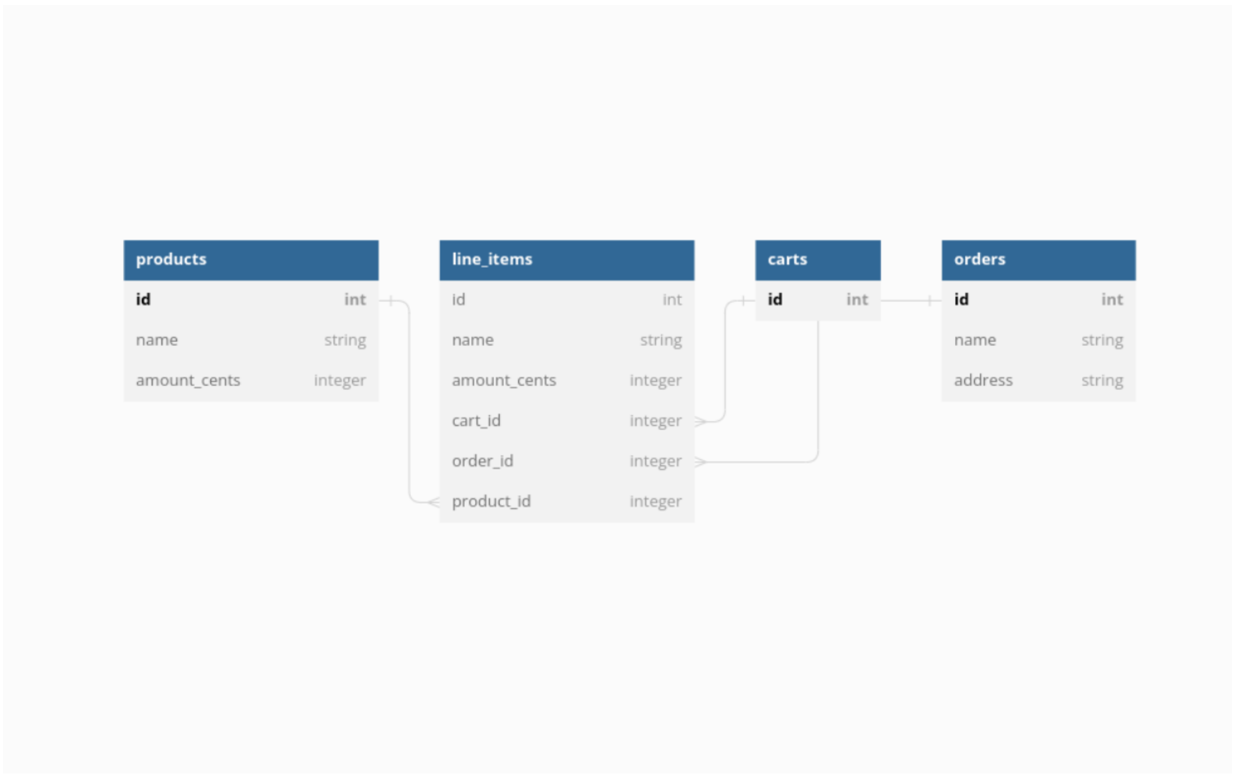
Sketch Workflow

- A simple sketch!



Data

Overview of our model structure, this might change in the future.



Now that we have everything in place, let's explore some fundamentals of building a web app,

See you in the next section 👉

Sinatra API project

Sinatra is a DSL for quickly creating web applications in Ruby with minimal effort.

Check out their docs

We are going to create a simple app that will perform `CRUD` operations on our product resource.

Product Application

Prerequisites

To follow along, it is helpful to have the following

- Basic Ruby knowledge.
- Understand how the web works.
- Postgresql and ruby Installed.
- How to use postman.

Above all, let's have fun!

Let the games begin.

I assume that you have your editor configured, create a directory and name it `product_api`

`mkdir product_api`

Set up Sinatra Application

In our directory, `bundle init` in your terminal, this will create a Gemfile.

Add this gems to your Gemfile.

```
gem "sinatra", "~> 3.0"
gem "http", "~> 5.1"
gem 'puma'
gem "pg"    # for Postgres
gem "rake"  # so we can run Rake tasks
gem "sinatra-activerecord"    # for Active Record models
gem 'rack-contrib', '~> 2.3'

group :development, :test do
  gem "rerun"
end
```

run `bundle install`

Create a config.ru file

`touch config.ru`

Copy and paste the above

```
require 'bundler'
require 'rack'
require 'rack/contrib'

Bundler.require

use Rack::JSONBodyParser
require_relative './product.rb'

run ProductAPI
```

We have alot going on here!, basically we are requiring packages from our `Gemfile` ☝.

Rack is a layer between the framework and the application server, good read.

Let's set up rake

Create a file in the `root` directory,

`touch Rakefile`

And copy the above

```
require 'sinatra/activerecord'
require 'sinatra/activerecord/rake'
# require './product'
```

Set up database

We are going to use postgres, in our `root` directory create a new directory `config` and add a file `database.yml` .

`mkdir config`

`cd config`

`touch database.yml`

Copy and paste the code above,

```
development:
  adapter: postgresql
  encoding: unicode
  database: database_name
  pool: 2
  username: add_psql_user # TO BE DONE
  password: add_psql_password # TO BE DONE
production:
  adapter: postgresql
  encoding: unicode
  pool: 5
  host: <%= ENV['DATABASE_HOST'] %>
  database: <%= ENV['DATABASE_NAME'] %>
  username: <%= ENV['DATABASE_USER'] %>
  password: <%= ENV['DATABASE_PASSWORD'] %>
```

Now that we have our database configured let's add our models.

Note: Change your postgres configurations.

Set up model

In our `root` directory create a new folder `mkdir model` .

Our `model` folder is going to have a simple ruby class, create a file `touch product.rb` .

Copy this ruby script:

```
class Product < ActiveRecord::Base

end
```

Dont worry about the `ActiveRecord::Base` class we'll revisit it.

Add our migrations

Create a folder in `root` directory `mkdir db`.

Run the above command,

`rake db:create_migration NAME=create_products`

In the newly generated file modify it to,

```
class CreateProducts < ActiveRecord::Migration[7.0]
  def change
    create_table :products do |t|
      t.string :name, null: false, default: ""
      t.integer :amount_cents, null: false, default: 0

      t.timestamps
    end
  end
end
```

Then run,

`rake db:migrate`

Set up a controller

Create a new file in `root` directory `touch product.rb`, and copy the above content.

Please read the comments above:

```
require 'sinatra/base'
require 'json'
require 'sinatra/activerecord'

set :database_file, 'config/database.yml'
$: << File.dirname(__FILE__) + "/model" # Assuming app.rb is at the same level as lib

require 'product'

class ProductAPI < Sinatra::Base

  get '/products' do
    # return product_array.to_json
    Product.select('id', 'name', 'amount_cents').all.to_json
  end

  get '/products/:id' do
    # return a particular product as json based on the id param from the url
    # params always come to a string so we convert to an integer
    # id = params["id"].to_i
    # return product_array[id].to_json
    product =  Product.find_by_id(params[:id])

    if product
      product.to_json
    else
      halt 404
    end
  end

  post '/products' do
    # create the new product
    # new_product = {name: body["name"], amount_cents: body["amount_cents"]}
    # push the new product into the array
    # product_array.push(new_product)
    # return the new product
    # return product_array.to_json
    # params = {name: body["name"], amount_cents: body["amount_cents"]}
    product = Product.create(params)

    if product
      product.to_json
    else
      halt 500
    end
  end

  put '/products/:id' do
    # get the id from params
    # id = params["id"].to_i
    # update the item
    # product_array[id][:name] = body["name"]
    # product_array[id][:amount_cents] = body["amount_cents"]
    # return the updated product
    # return product_array[id].to_json
    product = Product.find_by_id(params[:id])

    if product
      product.update(params)
    else
      halt 404
    end
  end

  delete '/products/:id' do
    # get the id from params
    # id = params["id"].to_i
    # delete the item
```

```
      # product = product_array.delete_at(id)
      # return the deleted item as json
      # return product.to_json
      product = Product.find_by_id(params[:id])

      if product
        product.destroy
      else
        halt 404
      end
    end

  end
```

Check out this Blog.

NOTE Uncomment the `require './product'` in `Rakefile`.

We have created a simple API application 😎, let's fire up postman and our application.

Run `bundle exec rerun rackup` to start the server.

Yeey!! 💪.

End points

verb	Endpoint	Action	Functionality
GET	http://127.0.0.1:9292/products	index	All products
GET	http://127.0.0.1:9292/products/1	show	Display product with id 1
POST	http://127.0.0.1:9292/products	create	Create a new product
PUT	http://127.0.0.1:9292/products/1	update	Update product with id 1
DELETE	http://127.0.0.1:9292/products/1	delete	Delete product with id 1

Make sure you include the body fields in `POST` and `PUT` request.

Nice work!,

See you in the next section 👉

What is REST API?

REST is an acronym for **Representational State Transfer**, it is an architectural style for distributed hypermedia systems.

In order to understand REST, one needs to know what an API is and who is a client and what is a Resource.

- **API** - this is an interface that helps two softwares to communicate.
- **Client** - this is the person who uses the API, client can also be a web browser.
- **Resource** - any object the API can provide information about.

RESTful web application exposes information about itself in the form of information about its resources. It also enables the client to take actions on those resources.

REST enables transfer of a representation data from the server to the client.

The representation of the state can be in a **JSON** format and probably for most API this is indeed the case, it can also be **XML** or **HTML** format.

Server operations when a client sends an API request :

- an identifier for the resource you are interested in, URL known as the endpoint.
- the operation server is supposed to operate, common methods are **CRUD**.

In order for an API to be RESTful, it has to adhere to 6 constraints:

Uniform Interface

In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behaviour of components.

- the request to the server has to include a resource identifier.
- the response the server returns includes enough information so the client can modify the resource.
- each request to the API contains all the information the server needs to perform the request and each response the server returns contains all the information the client needs in order to understand the response.
- hypermedia as the engine of application state.

Client-server separation

Client and server act independently, each on its own, and the interaction between them is only in the form of request, initiated by the client only and responses which the server sends to the client only as a reaction to a request.

The server awaits a request from the client.

Stateless

The server does not remember anything about the user who uses the API.

Layered System

Allows the system to be composed of hierarchical layers by constraining component behaviour such that each component cannot “see” beyond the immediate layer with which they are interacting.

Cacheable

Data in the server sends contain information whether is cacheable, it might contain some sort of a version number, the version number is what makes caching possible.

The client knows which version of the data it already has, the client can avoid sending a new request, client should also know if the version number has expired in which the client should send another request to the server to get the post updated data about the state of the resource.

Code on demand

REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. The simplifies clients by reducing the number of features required to be pre-implemented.

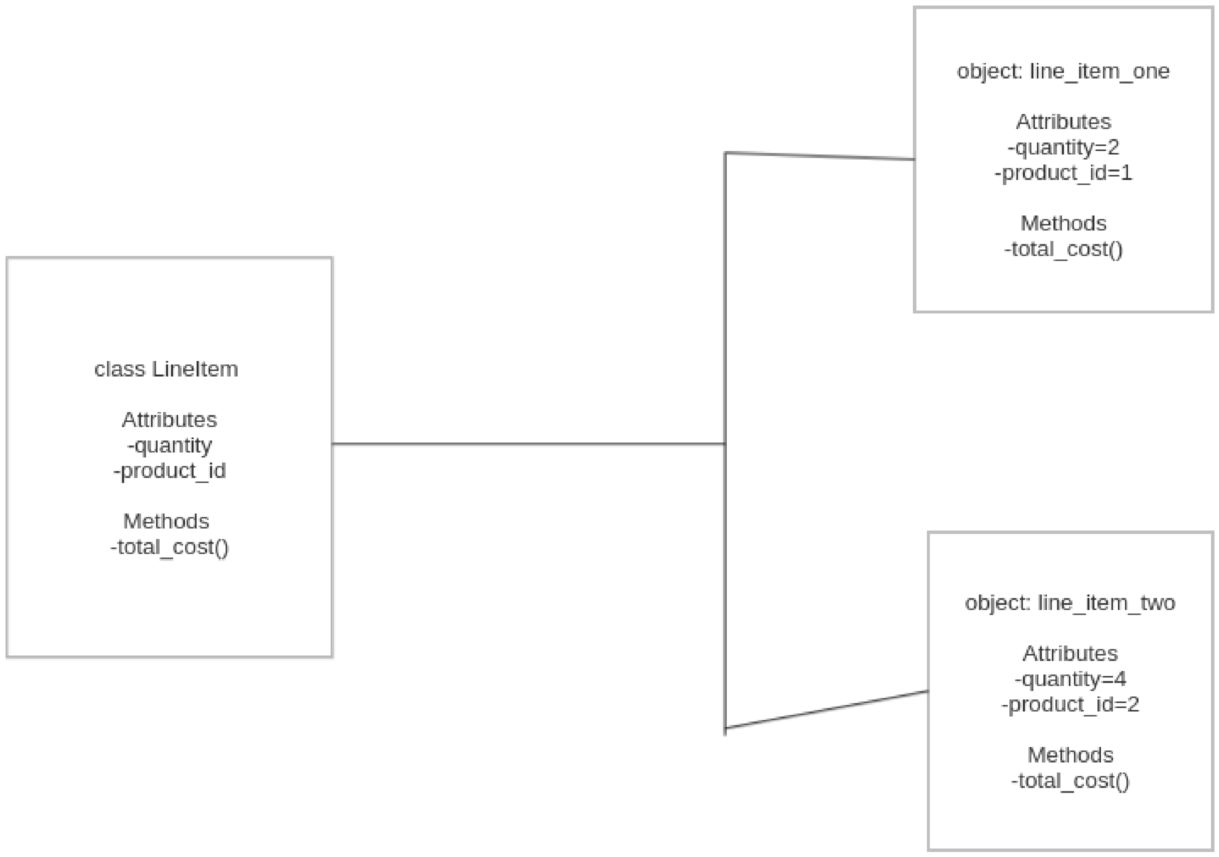
See you in the next section 👉

What is object Oriented Programming?

Object Oriented Programming (OOP) is a programming paradigm that relies on the concept of classes and objects.

It structures a software program into a simple, reausable pieces of code blueprints(classes), which are used to create individual instances of objects.

For example a **line_item** is an object which has certain **properties** such as **quantity**, **product_id**, etc. It also has certain **methods** such as **calculate line item's total cost** etc.



We can create an instance of a **line_item** type object, **line_item_one** to represent a specific **line_item**.

```
line_item_one = LineItem.new
# properties
line_item_one.quantity()
line_item_one.product_id

# methods
line_item_one.total_cost()
```

We could then set the value of the properties defined in the class to describe **line_item**, without affecting other objects or the class template.

We can then reuse this class to represent any number of line_items.

Ruby is an Object-Oriented Language

Everything you manupulate in Ruby is an object and the results of those manupulations are themselves objects.

When you write object-oriented code, you're normally looking to model concepts from the real world.

You create objects by calling a constructor, a special method associated with a class.

The standard contructor is called new(), you invoke methods by sending a message to an object.

See you in the next section 👉

What is Rails?

Rails is a web application development framework written in the Ruby programming language, Rails docs.

MVC

Model View Controller is a design pattern that divides related programming logic, making is easier to reason about.

Rails follow this design pattern, MVC.

Creating our Rails App

Create a directory and name it `Duka` , `mkdir duka` .

Create a rails app with `rails new duka`

Navigate to the newly created `duka` folder. In the app folder, Rails maintains the files for the controllers, models, and views.

Create a route. In `config/routes.rb` .

```
  Rails.application.routes.draw do
    root 'products#index'
  end
```

In the snippet above, we are instructing Rails to set the root route (/) to index action in ProductsController. So, the index action will be triggered when the user visits the root route.

We will run the following command to create the controller:

```
rails generate controller Products index --skip-routes
```

We added the --skip-routes, meaning we have defined the routes and to skip.

The generator will create files but the most important file is the controller file located in, app/controllers/products_controller.rb

```
  class ProductsController < ApplicationController
    def index
    end
  end
```

This generator also creates a file in app/views/landing/ named index.html.erb. By default, Rails renders a view that corresponds to the name of a controller action.

Replace the view content with:

```
<h1>E-commerce App</h1>
```

Let’s test our app by running:

```
rails s
```

We'll see our newly created application,

See you in the next section 👉,

Active Record

ActiveRecord is an interface that rails provides between the database and application.

What is an ORM

In the past, web applications required the skills to code in your business logic language and your database language.

ORM is a technique that lets one manage a database in the business logic language that one is most comfortable with.

Rails uses ORM to map database tables to ruby objects.

Active Record as an ORM framework gives us several mechanisms, the most important are:

- represent models and their data,
- represent associations between these models,
- represent inheritance hierarchies through related models,
- validate models before they get persisted to the database and perform database operations in an object-oriented fashion.

Migrations

Migrations are scripts that tell rails how you want to set up or change a database, migrations are a convenient way to alter database schema over time.

Stand alone migrations.

Migrations are stored in the db/migrate directory, one for each migration class.

Migrations will do more than just append timestamps, based on naming conventions and additional arguments it can also start fleshing out migrations.

If the migration name is of the form *AddColumnToTable* or *RemoveColumnFromTable* and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statement will be created.

You can also add an *index* on the new column, that will generate an `add_index` statement.

Generators also accept column type as *references* (available as `belongs_to`), this generates `add_reference` call, this migration adds a *foreign_key* and an appropriate *index*.

There is also a generator that produces *JoinTables*.

The `change` Method, this is the primary way of writing migrations, it works for the majority of the cases where Active Record knows how to reverse the migrations automatically.

Migrations are usually reversible, this helps in case of errors . `rails db:rollback` , rolls back the last series of migrations.

In cases where one has a migration that can't be automatically undone, we have to specify the actions for doing and undoing the migrations.

Another thing to note is that migrations are not run when you create a new model, you have to run them manually.

Rails `db:migrate` will run the change or up methods for all migrations that have not yet been run.

The command above invokes `db:schema:dump` which updates `db/schema.rb` file to match the database structure.

Other useful commands;

`Rails db:reset`, drops the database and sets it up again, equivalent to `rails db:drop` `db:setup`.

Editing existing migrations because of legitimate cases is not recommended, one is advised to write new migrations that perform the changes he/she requires.

Editing migrations that have not yet been committed or editing migrations in development is relatively harmless.

Validations

Validations ensure that data persisted to the database is correct data.

Data can be validated in several ways before it is saved into the database, including **client side- validation**, one can write javascript code in the browser which detects form fills and provides warnings in case of errors.

Native database constraints validates user data before being persisted to the **database** and **controller level validations**.

Native database constraints validation is preferred for it truly enforces constraints on the **database level**.

`ActiveModel::Errors` contains all errors, each error represented by an `ActiveModel::Error object`

- `errors[:attribute_name]` is used to check error message of a specific attribute it returns an array of strings with all error message for a given attribute
- displays an empty array if there are no errors related to the attribute.

Callbacks

Callbacks let one run custom methods at certain moments of an object life cycle.

Rails ships with a number of callbacks, these methods can be used as a means to perform more logic **during**, **after**, **before** or **around** manipulations of active record objects.

Callbacks should be registered before being used. It is advised to declare callback methods as private to conform with the principle of object encapsulation.

Relational callbacks -> this is performed when related objects are changed.

Conditional callbacks -> used to render conditional logic, this can be achieved in many ways. First is using `:if` and `:unless` option which can take a proc or an array.

Associations

An association is a connection between two Active Record objects, making common operations simpler and easier in the code.

Choosing between belongs_to and has_one: foreign key goes on the table of the class declaring the belongs_to association.

- belongs_to sets up a 1:1 connection with another model and has_one only contains one instance of another model.
- choosing between has_many :through and has_and_belongs_to_many: these associations declare many-to-many associations.

- has_and_belongs_to_many allows one to make associations directly and has_many :through makes an indirect association through a join model.
- polymorphic association -> assumes that a model belongs to more than one model on a single association.

Product Model

Let's add our `product` model,

We a going to use a generator command, `rails g model product name amount_cents:integer` ,

This will generate several files, we will only focus on `app/models/product.rb` and `db/migrate/<timestamp>_create_products.rb`

Remember to run your migrations. Migrations are feature of Active Record that allow you to evolve your database schema over time.

`rails db:migrate`

Let's try out our newly created model

`rails console`

```
Product.create!(name: "E-book", amount_cents: 2000)
```

See you in the next section 👉

Action Controller

Routers determine which controller to use for the request. Controllers will then receive the request and save or fetch data from our models.

A **controller** can thus be thought of as a middleman between **models** and **views**.

It makes the model data available to the view, so it can display that data to the user, and it saves or updates user data to the model.

Let’s add an `index` action in our `products_controller`, to fetch the newly created `product`.

```
def index
  @products = Product.all
end
```

In the code above, if a **seller** goes to `/` in your application, Rails will create an instance of `ProductsController` and call its `index` method.

Rails will render the `index.html.erb` view .

By **fetching** all products, the `index` method can make a `@products` instance variable accessible in the **view**.

See you in the next section 👉

Action View

In Rails, web requests are handled by **Action Controller** and **Action View**.

Typically, Action Controller is concerned with **communicating with the database and performing CRUD actions where necessary**. Action View is then responsible for **compiling the response**.

Controllers makes model data available to the view and this data can be **displayed** to the user.

In the, `index.html.erb`, paste the content below,

```
<h1>E commerce App</h1>


<% @products.each do |product| %>
  <p>Product Name: <%= product.name %></p>
  </p>Product Price: <%= product.amount_cents %></p>
<% end %>
```

You'll see your product listed below.

See you in the next section 👉

What is a session?

Session is just a place to **store data** during **one request** that you can **read** during **later requests**.

An **encrypted hash** in ruby, that stores data which can be sent forth and back in a request and can be **serialized**.

For a better understanding of what a session is, let's take a look and study **cookies**.

Cookies are set when a browser responds back to a request.

The browser will **store** those cookies until the cookies **expire**, every time one can make a request the browser will send the cookies back to the server.

What does this have to do with a session?

Cookies like session stores data from the client but they aren't always the right answer for session data.

Reasons why cookies are not equivalent to sessions:

- one can only store about 4kb of data in a cookie
- cookies are sent along with every request you make -> big cookies means bigger requests and responses which means slower browser.
- if you accidentally disclose secret_key_base, users can change the data you’ve put inside cookies.
- storing wrong data inside a cookie can be insecure.

Alternative session stores

Sessions and cookies are similar but **sessions are more secure** because cookies only contain a **session ID** and in a Rails app looks for **data** in the **session store** using the same **ID**.

Cookie store, cache store, or database store ?

Storing sessions in cookies by far is the easiest way to go because it does not need any extra infrastructure or setup.

But if one wants to move beyond the cookie session store. One has two options, **store sessions** in the **database** or store them in the **cache**.

Storing sessions in the cache

Cache store is the second-easiest place to store session data, since it's already set up.

One does not need to worry about session stores growing out of control because older sessions will automatically get kicked out of the cache if it is too big.

This is **fast** because cache is kept in memory.

But this is not perfect:

- if you actually care about keeping old sessions around one would probably dont want them kicked off out of the cache.
- sessions and cache data will have minimal space causing cache misses and early expired sessions.
- if you ever need to reset cache, there is no way to do that without expiring everyone sessions.

Storing sessions in the database

If you want to keep your session data around until it legitimately expires, you probably want to keep it in some kind of database.

Database session storage also has drawbacks:

- with some database stores, sessions won't get cleaned up automatically.
- you have to know how the database will behave when its full of session data
- you have to be more careful about when you create session data, or you'll find your database with useless sessions.

Session Based Authentication

We store our session when we want our users to be loged in.

We are going to set up a session controller,

Let's add the following routes:

```
# config/routes.rb
get '/login', to 'sessions#new'
post '/login', to: 'sessions#create'
delete '/logout', to: 'sessions#destroy'
```

```
class SessionsController < ApplicationController

  def new
  end

  # create session.

  def create
    user = User.find_by(email: params[:session][:email].downcase)

    if user && user.authenticate(params[:session][:password])
      log_in user
      redirect_back_or user
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  # remove session

  def destroy
    log_out
    redirect_to root_url
  end

end
```

Let's add a `helper_method` , `session_helper.rb`

```
module SessionsHelper

 # accepts user and creates a session for the user.
  def log_in(user)
    session[:user_id] = user.id
  end

 # returns a current user if present in the session.
  def current_user
    if session[:user_id]
      @current_user ||= User.find_by(id: session[:user_id])
    end
  end

  def logged_in?
    !current_user.nil?
  end

 # logs out user by deleting session and setting current_user to nil
  def log_out
    session.delete(:user_id)
    @current_user = nil
  end

  def current_user?(user)
  user == current_user
  end

 # redirect non logged in user to the first visited url after loggin in.
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
  end

  def store_location
    session[:forwarding_url] = request.original_url if request.get?
  end
end
```

This `session helper` will be imported in the application controller. Making the methods available to all controllers.

```
class ApplicationController <  ActionController::Base

  include SessionsHelper

  private

  def logged_in_user
    unless logged_in?
      store_location
      flash[:danger] = "Please log in."
      redirect_to login_url
    end
  end
end
```

We need to add a `before_action` filter in our controller to restrict our users,

```
# products_controller.rb

class ProductsController < ApplicationController

  before_action :logged_in_user
```

Check out this branch.

This are the core components in a `session based authentication`, you can check out How to set up authentication form scratch or Session Based Authentication blogs.

See you in the next section 👉

JSON

When exchanging data between a browser and a server, data can only be text format.

JSON is text that we can convert into javascript objects and send it to the server.

We can also convert any JSON received from the server into javascript objects, this enables us to work with the data as Javascript objects with no complicated parsing and translations.

JSON makes it possible to store javascript objects as text.

JSON has two structures;

- a collection of name/value pairs => hash
- an ordered list of values => array

JSON is lightweight data-interchange format

JSON is **self-describing** and easy to understand

JSON is language independent.

`JSON.parse()` -> this is a javascript built in function to convert a string written in JSON format into native JavaScript objects.

Understanding JSON Web Tokens (JWT)

A JSON web Token(JWT) is a JSON object defined in RFC 7519 as a safe way to represent a set of information between two parties, more.

The token is composed of a **header**, a **payload**, and a **signature**.

JWt is used for **authorization** not **authentication**.

To illustrate how it works we use 3 simple entities, the **user**, **application server** and the **authentication server**.

- User signs into the authentication server using a login system
- The authentication server then creates the JWT and sends it to the user.
- When user makes an API call to the application, user passes a JWT along with the API call, in the setup the application server would be configured to verify that the incoming JWT are created by the authentication server.
- When user makes API calls with the attached JWT, the application can use the JWT to verify that the API call is coming from an authenticated user.

API should be stateless

An API does not handle sessions, an API should be stateless.

Setting up the authentication token

Install `jwt`,

```
bundle add jwt
```

The library has two methods, `JWT.encode` and `JWT.decode`.

Open the `rails console`

```
token = JWT.encode({message: 'Rails is fun!!'}, 'my_secret_key')
JWT.decode(token, 'my_secret_key')
```

In the first line, we encoded a payload with the secret key my_secret_key. So we get a token we can simply decode.

The second line decodes the token, and we see that we find our payload well.

Let's add a `json_web_token.rb` file in our `lib` directory.

```
class JsonWebToken
  # secret to encode and decode token
  HMAC_SECRET = Rails.application.secrets.secret_key_base

  def self.encode(payload, exp = 24.hours.from_now)
    # set expiry to 24 hours from creation time
    payload[:exp] = exp.to_i
    # sign token with application secret
    JWT.encode(payload, HMAC_SECRET)
  end

  def self.decode(token)
    # get payload; first index in decoded Array
    decoded = JWT.decode(token, HMAC_SECRET)[0]
    HashWithIndifferentAccess.new decoded
  end
end
```

The method `JsonWebToken.encode` takes care of `encoding` the **payload** by adding an *expiration date* of 24 hours by default.

We also use the same encryption key as the one configured with Rails

The method `JsonWebToken.decode` decodes the JWT token and gets the payload.

Then we use the `HashWithIndifferentAccess` class provided by Rails, which allows us to retrieve a value of a Hash with a Symbol or String.

Error Handling

This class will handle our exceptions.

Create a new file `lib/errors.rb`

```
class Errors < Hash
  def add(key, value, _opts = {})
    self[key] ||= []
    self[key] << value
    self[key].uniq!
  end

  def add_multiple_errors(errors_hash)
    errors_hash.each do |key, values|
      values.each { |value| add key, value }
    end
  end

  def each
    each_key do |field|
      self[field].each { |message| yield field, message }
    end
  end
end
```

Authenticate User

Let's create a class that will authenticate our users with valid email and password.

`mkdir app/services`

`touch app/services/authenticate_user.rb`

```
class AuthenticateUser < ApplicationService
  def initialize(email, password)
    @email = email
    @password = password
  end

  private

  attr_accessor :email, :password

  def call
    # create token with user_id
    JsonWebToken.encode(user_id: user.id) if user
  end

  def user
    user = User.find_by(email: email)
    return user if user&.authenticate(password)
    errors.add :user_authentication, 'invalid credentials'
  end
end
```

We are inheriting from `ApplicationService` class, let's set it up

`touch app/services/application_service.rb`

```
class ApplicationService

  # def self.call(*arg)
  class << self
    def call(*arg)
      new(*arg).constructor
    end
  end

  attr_reader :result
  def constructor
    # AuthenticateUser#call
    @result = call
    self
  end

  def success?
    !failure?
  end

  def failure?
    errors.any?
  end

  def errors
    @errors ||= Errors.new
  end

  def call
    fail NotImplementedError unless defined?(super)
  end
end
```

Now that we have our authenticate user logic in place let's authorize our request.

Authorize API request

We are going to authorize all request that comes into our app.

`touch app/services/authorize_api_request.rb`

```
class AuthorizeApiRequest < ApplicationService
  attr_reader :headers

  def initialize(headers = {})
    @headers = headers
  end

  def call
    user
  end

  private

  def user
    @user ||= User.find(decoded_auth_token[:user_id]) if decoded_auth_token
    @user || errors.add(:token, 'invalid token') && nil
  end

  def decoded_auth_token
    # decodes a valid token
    @decoded_auth_token ||= JsonWebToken.decode(http_auth_header)
  end

  def http_auth_header
    if headers['Authorization'].present?
      return headers['Authorization'].split(' ').last
    else
      errors.add(:token, 'missing token')
    end
    nil
  end
end
```

We are checking our headers, making sure it has a valid `token`, if true it returns a `user` object.

Set up token Controller

Let's create a file `touch app/controllers/sessions_controller.rb`

```
class SessionsController < ApplicationController

  skip_before_action :authenticate_request

  def create
    auth = AuthenticateUser.call(params[:email], params[:password])
    if auth.success?
      render json: { auth_token: auth.result }
    else
      render json: { errors: auth.errors }, status: :unauthorized
    end
  end
end
```

Add a route, in `routes.rb`

```
post 'authenticate', to: 'sessions#create'
```

Set up the `application_controller`,

```
class ApplicationController < ActionController::API
  before_action :authenticate_request
  attr_reader :current_user

  private

  # authenticate all requests
  def authenticate_request
    auth = AuthorizeApiRequest.call(request.headers)
    @current_user = auth.result
    render json: { errors: auth.errors }, status: :unauthorized unless @current_user
  end
end
```

Checkout this branch.

More on this please visit Rails 6 API and Todos API.

See you in the next section 👉

Rails Authentication with OAuth 2.0 and OmniAuth

OAuth 2 is the industry-standard protocol for authorization, OAuth 2.0.

It enables a third-party application to obtain limited access to an HTTP service.

We will implement OmniAuth with **Google** and **Devise**

Overview of what happens when a user visits an app and tries to authenticate via a social google:

- User clicks *login* link
- User is redirected to the social network's website. The app's data are sent along for identification.
- User sees the app's details and which actions it would like to perform on his behalf.
- User can cancel the authorization incase of trust issues.
- If user trusts the app, the authorization is approved and the user is redirected back to the app(via callback URL).
- Information about the user and a generated token(secret key) is sent along

Demo

Add to your `Gemfile` :

```
gem 'omniauth-google-oauth2'
```

Then `bundle install` .

Add configuration in `config/initializers/devise.rb` :

```
config.omniauth :google_oauth2, 'GOOGLE_CLIENT_ID', 'GOOGLE_CLIENT_SECRET', {}
```

To register *Google*, you need to provide the `client id` and `client secret` .

Google API Setup

- Go to google Developer Console.
- Select your project
- Go to Credentials, then select the "OAuth consent screen" tab on top, and provide an 'EMAIL ADDRESS' and a 'PRODUCT NAME'
- Wait 10 minutes for changes to take effect.

Set-up a Callback URL

This is the URL where a user will be redirected to inside the app after successful authentication and approved authorization.

In our `config/routes.rb`

```
devise_for :users, controllers: { omniauth_callbacks: 'users/omniauth_callbacks' }
```

Authentication Hash

The `google` method inside the `Users::OmniauthCallbacksController` will parse the user data, save it into the database and perform sign in to the app.

In our `users/omniauth_callbacks_controller.rb` ,

```
class Users::OmniauthCallbacksController < Devise::OmniauthCallbacksController
  def google
    @user = User.from_omniauth(request.env['omniauth.auth'])
    if @user.persisted?
      flash[:notice] = I18n.t 'devise.omniauth_callbacks.success', kind: 'Google'
      sign_in_and_redirect @user, event: :authentication
    else
      session['devise.google_data'] = request.env['omniauth.auth'].except('extra')
      redirect_to new_user_registration_url, alert: @user.errors.full_messages.join("\n")
    end
  end
end
```

Saving User Data

Now that we have our `authentication_hash` up, we need to set up our `user` model , correctly.

Add `omniauthable` to devise and a method to create a user if not found.

```
class User < ActiveRecord::Base
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
         :recoverable, :rememberable, :validatable, :omniauthable, omniauth_providers: [:github]

  def self.from_omniauth(access_token)
    data = access_token.info
    user = User.where(email: data['email']).first
    unless user
      user = User.create(
        email: data['email'],
        password: Devise.friendly_token[0,20]
      )
    end
    user
  end
end
```

We have successfuly set up `OAuth` Authentication with Google.

See you in the next section 👉

Rails Security

In this guide we'll look into Rails vulnerabilities and how we ca go about it.

Please clone Rails Authentication from Scratch to follow along,

Session Fixation

This attack is focused on fixing user's session ID known to the attacker and forcing user's browser into using this ID

This is solved with the `reset_session` , this issues a new session identifier and declares the old one invalid after a successful login.

This is demonstrated in the `Authentication` module: login and logout methods both have `reset_session` .

```
# concerns/authentication.rb

  def login user
    reset_session
    active_session = user.active_sessions.create!(user_agent: request.user_agent, ip_address: request.ip)
    session[:current_active_session_id] = active_session.id

    active_session
  end

  def logout
    active_session = ActiveSession.find_by(id: session[:current_active_session_id])
    reset_session
    active_session.destroy! if active_session.present?
  end
```

User Management

User passwords must be stored only as cryptographic hash.

This is given in Rails with the `has_secure_password` , which supports secure password hashing, confirmation and recovery mechanism.

This is included in our `User` model along side `bcrypt gem` .

```
# models/user.rb

class user < ApplicationRecord

has_secure_password

end
```

Brute force Attack

Accounts on the trial and error attack on the login page

The best solution to this is to display a generic error message or require to enter a CAPTCHA

```
# controller/sessions_controller.rb

def create
  ##

  flash.now[:alert] = "Incorrect email or password."
  render :new, status: :unprocessable_entity
end
```

Account Hijacking

Passwords

In scenarios where hacker is about to reset a user password after session fixation, provide attacker with a field to type current password.

This makes it more difficult for the attackers.

Email

Require user to enter a password when changing an email

```
# controllers/users_controller.rb

def update
  ##

  if @user.authenticate(params[:user][:current_password])
  ##
end
```

This is just but a few security measures while authenticating a Rails app.

See you in the next section 👉