Objective

- Add token-based authentication
- Move controller logic to the service layer

Constraints

- A user should be able to sign up
- A user should be able to log in and receive an access token
- A user should be able to use the received access token to access the different API requests in the application
- If a user has no access token, then they are not allowed to access information from the APIs

Proposed solution

- Use the devise token auth gem to add token-based authentication.
- Create three services for every controller request as below
    - Organization service
    - Project service
    - Task Service

Testing strategy

- Run the application RSpec suit ensuring that all tests pass as they are supposed to
- Update the RSpec suite to include token-based authentication and ensure that it passes
- Ensure that a new user can be created through the API
- Ensure that the user is able to log in and have access tokens
- Ensure that the user is able to access organizations, projects, and tasks through secure API requests
- Ensure that the user is able to perform CRUD operations on the projects and tasks they created.

Functional specifications

Git workflow

The git-flow has been updated to specify the ruby version 2.7.4 which the application depends on.

Gems

Devise Token Auth gem

The main repository for the gem is found here

The devise token auth gem is a great gem to add token-based authentication to rails as it uses devise in the background. So the devise calls like authenticate_user can be used within the API. It also adds email and password-based authentication to the user model or any model that it's assigned to.

Devise token auth also add routes for sign up, sign in, edit, passwords, and more devise relates routes through the API.

Models

Please see the diagram here for the database structure.

User model

The user model is added to store the information of the user, which includes the email, password, etc.
The user model has the relationships of having many user organizations and also has many organizations through user organizations

User Organization model

The user organization model is used to keep information of the user and the organization, using this model, we can have a one to many relationships between both the organization and the user model allowing a user to have many organizations and an organization to have many users through this model

Organization model

The organization model has been updated to have many user organizations and to have many users through user organizations.

Services

Organization create

This service is used to create an organization

Methods

- Initialize
  - Params: user_id, organization_name
  - Effect: Initialize the user object and set the organization name in a variable
  - Response: None
- Call
  - Params: none
  - Effect: Creates a new organization and a user organization mapping the user to the created organization.
  - Response: If the organization has been created, then the response is { status: true, organization: organization_object } and it’s { status: true, organization: organization errors } if there are errors in creating the organization.

Project Service

This method handles the logic concerned with creating, updating reading, and deleting a projects

Methods

- Initialize
  - Params: organization_id
  - Effects: Initializes an organization based on the organization id passed.
  - Response: None
- Project
  - Params: id (This is the project id)
  - Effect: Searches for the project in an organization based on the passed id
  - Response: Project object.
- Projects
  - Params: None
  - Effect: Searches for projects on the organization
  - Response: An array of projects in the organization

- Create_project
    - Params: Params (this is a hash containing the project attributes and values)
    - Effect: Creates a project and attaches it to the organization
    - Response: If the project has been created, then the response is { status: true, project: project_object } and it's { status: true, project: project errors } if there are errors in creating the project.

- Update
    - Params: id (project id), params (project parameters)
    - Effect: It finds the project in the organization and updates it based on the parameters passed.
    - Response: If the project has beenupdated, then the response is { status: true, project: project_object } and it's { status: true, project: project errors } if there are errors in updating the project
- Destroy
    - Params: id (project id)
    - Effect: It finds the project in the organization and deletes it
    - Response: True

Task Service

This method handles the logic concerned with creating, updating reading, and deleting a tasks
Methods

- Initialize
    - Params: project_id
    - Effects: Initializes an organization based on the organization id passed.
    - Response: None
- Task
    - Params: id (This is the task id)
    - Effect: Searches for the task in an organization based on the passed id
    - Response: task object.
- Tasks
    - Params: None
    - Effect: Searches for tasks on the organization
    - Response: An array of tasks in the organization
- Create_task
    - Params: Params (this is a hash containing the task attributes and values)
    - Effect: Creates a task and attaches it to the organization
    - Response: If the task has been created, then the response is { status: true, task: task_object } and it's { status: true, task: task errors } if there are errors in creating the task.

- Update
    - Params: id (task id), params (task parameters)

    - Effect: It finds the task in the organization and updates it based on the parameters passed.
    - Response: If the task has been updated, then the response is { status: true, task: task_object } and it’s { status: true, task: task errors } if there are errors in updating the task
- Destroy
    - Params: id (task id)
    - Effect: It finds the task in the organization and deletes it
    - Response: True

Routes

Sign up

This is the request made to sign up a new user

Request

POST /auth

Body:

```
{
 "email": "test3@email.com",
 "password": "password",
 "password_confirmation": "password"
}
```

Response

Body:

```
{
  "status": "success",
  "data": {
    "id": 5,
    "provider": "email",
    "uid": "test3@email.com",
    "allow_password_change": false,
    "name": null,
    "nickname": null,
    "image": null,
    "email": "test3@email.com",
    "created_at": "2022-01-23T21:22:52.249Z",
    "updated_at": "2022-01-23T21:22:52.441Z"
  }
}
```

Headers

access-token	This is the access token
token-type	bearer
uid	User email
expirary	Expiary date time
client	Clinet token

Sign in

This is the request made to sign in a user

Request
POST /auth/sign_in

Body:
```
{
  "email": "test@email.com",
  "password": "password"
}
```

Response
The response is the same as in the sing up request above.

User account

This request is made to get details of the current user signed in.
Request
GET /account

Request Headers
Add the headers to the request received from the signin in request above.

Response
Below is an example response.

```
{
  "data": {
    "id": "1",
    "type": "user",
    "attributes": {
      "id": 1,
      "email": "test@email.com",
      "organizations": [
        {
          "id": 4,
          "name": "Nienow-Douglas",
          "created_at": "2022-01-21T12:49:03.839Z",
          "updated_at": "2022-01-21T12:49:03.839Z"
        },
        {
          "id": 5,
          "name": "Lindgren-O'Connell",
          "created_at": "2022-01-21T12:50:28.784Z",
          "updated_at": "2022-01-21T12:50:28.784Z"
        }
      ]
    }
  }
}
```

Authentication

To authenticate routes add the headers below received from sign in to the request.

References

1. https://dbdiagram.io/d/61eab6b87cf3fc0e7c514015
2. https://github.com/lynndylanhurley/devise_token_auth