MangaStore

One Piece Store

Bonnal Nathaël, Rafael Salort

2023

Homework N°

#

Introduction

In the conceptualisation of the logical part of our application, we have reflected on several ways of how to implement the various functionalities expected for an e-commerce site to sell manga and related products.

The first point, was about setting up a permissions system. And we set up a model (Role, Permission, User) where a user can have n roles and vice versa and it can also have n permissions and vice versa, and each role can also have n permissions and vice versa. In the idea, everything will happen in the permissions, it will be necessary to have a perm with the adequate key to be able to pass the Policies system of the server. For example, to create a product you need 'product:store'.

In this way, each model has its own policy, which is essential for establishing an authorisation for an action.

Another point is that certain functionalities, such as logging in or creating an account, can be done for visitors.

Architecture

For the creation of our application, we will focus on the creation of micro-services (which means that our backend app will be a REST-Api), using AdonisJS.
We will use Typescript, for its optional static typing, and code security.
For deployment we will use Docker and the Github CI/CD.

Solution: Choice of SGBD

For the storage of our data we took the solution to select PostgreSQL, for its relational side, its benchmarks and other features.

1.0.1 Routing

The routing system is essential for establishing functionality to a route.

- $/api/v1/manager/accounts/users$ to retrieve all users

Thanks to AdonisJS, we can model all our routes in a concise and clean way, in the following way :

```

  Route.group(() => {
    Route.group(() => {

      Route.group(() => {
        Route.group(() => {}).middleware('guest')
        Route.group(() => {}).middleware('auth')
      }).prefix('/authentication')

      Route.group(() => {
        Route.group(() => {
            Route.group(() => {
              Route.get('/', 'Manager/UsersController.index')
              Route.get('/:id', 'Manager/UsersController.show')

              Route.post('/', 'Manager/UsersController.store')
              Route.put('/:id', 'Manager/UsersController.update')
              Route.delete('/:id', 'Manager/UsersController.delete')
            }).prefix('/users')
        }).prefix('/accounts')
        // ... more routes

      }).middleware('auth').prefix('/manager')
    }).prefix('/v1')
  }).prefix('/api')


```

As we can see, we can determine a group and assign a prefix to it. We can also add a middleware that will allow us to protect all our routes.

1.0.2 Bouncer Policies

The bouncer policies part is essential for the data security of our application. Each action requested from the api will be submitted to a validation.

Here is an example of use :

```
export default class UserPolicy extends BasePolicy {
  public async before (user: User): Promise<true | undefined> {
    const permissions: string[] = await HelperPolicy.getPermissions(user)
    if (permissions.includes('admin')) return true
  }

  public async view (user: User): Promise<boolean> {
    const permissions: string[] = await HelperPolicy.getPermissions(user)
    return permissions.includes('user:view')
      || permissions.includes('user:store')
      || permissions.includes('user:update')
      || permissions.includes('user:delete')
  }

  public async store (user: User): Promise<boolean> {
    const permissions: string[] = await HelperPolicy.getPermissions(user)
    return permissions.includes('user:store')
  }

  public async update (currentUser: User, user: User): Promise<boolean> {
    const roleCurrentUser: Role = await HelperPolicy.getMaxRole(currentUser)
    const roleUser: Role = await HelperPolicy.getMaxRole(user)

    if (roleCurrentUser.power <= roleUser.power) return false

    const permissions: string[] = await HelperPolicy.getPermissions(currentUser)
    return permissions.includes('user:update')
  }

  public async delete (currentUser: User, user: User) {
    const roleCurrentUser: Role = await HelperPolicy.getMaxRole(currentUser)
    const roleUser: Role = await HelperPolicy.getMaxRole(user)

    if (roleCurrentUser.power <= roleUser.power) return false

    const permissions: string[] = await HelperPolicy.getPermissions(currentUser)
    return permissions.includes('user:delete')
  }
}

```

In the update function for example, I get the maximum role of the two users, and if that person does not have a higher role then I deny access and then check by permissions (like on Discord).

Functionalities

On the functionalities, we will find a page for the users to access their profiles. They will be able to view their orders, their baskets, their information...

An authentication page will be available for visitors as well as a registration page.

On the panel, sales managers will be able to organise, edit and create products and categories!

Administrators will also be able to manage roles and users.

Links

- Github: Github Repository