API Test - The CRM service

The required task consists in developing a REST API to manage customer data from a small shop. Users will have access to a full CRUD of customers and admin users will be allowed to manage the users themselves.

1.Entities

Four entities were developed to get the API running

BaseEntity

This is the base entity from which all others extend.

It's main purpose is to provide every persisted entity in our system with information about when was created, updated, and who was the user who performed these operations.

This is done using doctrine extensions *Gedmo\Timestampable* and *Gedmo\Blameable* which are basically listening to the doctrine events "create" and "update" to inject the required info when the entities are written to the database.

This class also handles two identifiers. A classic auto-generated integer which is used internally for foreign keys and a public uuid which is the one going to be exposed to the user.

The reason to use the public id is for obfuscation. Generally an API client should not know how many records we have in our tables.

User

This class is the one modelling the users who are going to interact with our API and the field that is going to be used as an identifier will be the email.

Customer

The resource that is going to be managed by our users. They basically have a name, surname, an avatar and also an email field was added.

Media

This entity represents a resource that is going to be in our system. It contains all the info regarding the files being uploaded by our users and provides more info than just an url to a file. Also gives us the power to authorize access depending on the media type.

2. Development stack and dependencies

A full stack is provided running on docker. Containers with php 8, nginx 1.19 and mysql are set with a few image customizations in php in order to make it work as we require.

The framework of choice is none other than symfony 5.2 and some dependencies were installed using composer:

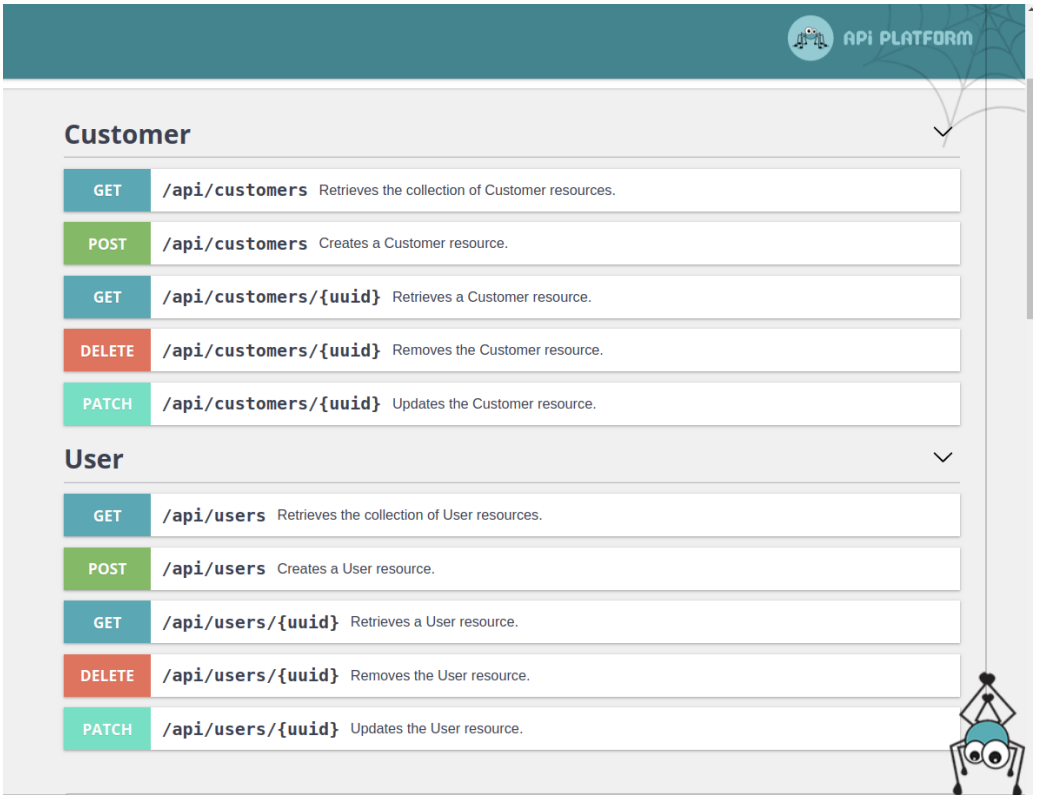
- **Api-Platform:** Extends symfony capabilities to let us develop API's in a more efficient and elegant way
- **symfony/security-bundle:** For the user authentication
- **symfony/orm-pack:** Let us use doctrine for all our database interaction
- **doctrine/annotations:** Let us describe our database schema by annotating our entities php code
- **stof/doctrine-extensions-bundle:** Used in the BaseEntity for letting us track all our entity activities
- **Orm-fixtures:** For populating our database with fixture data
- **fzaninotto/faker:** For generating realistic random data for our fixture
- **symfony/phpunit-bridge:** Let us develop our tests
- **symfony/browser-kit:** Simulates a web client in the testing section
- **dama/doctrine-test-bundle:** Let us modify our test database and revert all changes at the end of each test. This bundle generates a deprecation warning but it's just a bug. The developers are already working on it.

3.Api-Platform

Api-Platform is used for enhancing symfony capabilities and letting us design our API in a faster and more elegant way.

This library let us define the actions over the entities exposed in our API by using a set of simple, yet powerful, annotations directly written in our class code (In case of dealing with big projects consider moving these annotations to a separate yaml file).

It even generates documentation on it's own using swagger (every programmer's dream) and after you setup the project you can take a look at it in your browser with "localhost:81/api/docs"



One interesting feature is that out of the box resource pagination is provided and our serialized data is presented with a json-ld format.

Pay attention to the normalize context groups for every operation defining which attributes of our class are going to be serialized and also the attributes that are being read from the user request in the de-normalize context for de-serialization.

4.Authorization

The authorization is based on a symfony mechanism called **Voter**.

We can define our own classes that extend from a Voter and Symfony checks with them all every time an is_granted call is performed.

In every one of these classes we define which entity we are going to authorize actions on and a good practice is defining a single voter for every resource the user is going to interact with.

In our case UserVoter and CustomerVoter both live in the ‘src/Security/Voters” folder.

Each voter also defines a set of operations that the user should be authorized for. If you take a look at the api-platform annotations you will see the is_granted call in the security section for every api method that is defined.

CustomerVoter: since every user has all permissions over the Customer entity the CustomerVoter is empty and just returns authorized. Skeleton is provided nevertheless to showcase the proper way of doing it.

UserVoter: The operations over the users rely on user Roles to know if they can be performed. In the UserVoter we can see how we forbid the user to delete itself and how we provide users with Admin role full access and normal users with a very restricted one (they can only read themselves basically)

5.Validation

For data input validation we rely on assert annotations over our entities code and constraint Validators.

Assert annotations are just rules defined per each class property that are evaluated every time a write operation is going to be performed.

Constraint validators live in the validators folder and are performed after the annotations are evaluated. This is actual code we write to check for more complex things that would be more difficult to check with simple assert annotations.

Every time an assert annotation or a Constraint Validator check fails a violation is attached to a violations array that will be returned to the user to inform what went wrong.

6.Avatars

It is possible for Api-Platform operations to live together with more classic symfony controllers. This is visible in the MediaController and we are going to use it for letting our users upload and delete avatars.

The workflow in this case is to let the controller handle all the http request and response logic. It's responsibility is to grab the request parameters, read the needed entities from the repositories and then let the EntityManagers do the requested operation over the entities.

In this way we isolate our business logic from the input methods and we can call these methods from everywhere they are needed without duplicating any code.

These entity managers live in the "services" folder and since we are using an object oriented design we have a manager per entity. In this project some of them are empty and they just extend from the BaseEntityManager to perform some logic which is common to all of them.

Even if it's just extending the BaseEntityManager and has no code of its own, creating a manager for every entity is a good practice and prevents future errors in case we decide to add some code later.

After the EntityManager finishes with the operation it returns the control to the controller who generates a http response with the operation result and returns to the user.

7.Data fixtures

How do we test our system if we have no data in our DB?

Using the fzaninotto/faker bundle we can fill our database with fake random data that looks and behaves like if it was introduced by the real users themselves. This is really useful for getting the app ready for a test battery.

You can launch this generation with the command

```
php bin/console doctrine:fixtures:load --env=test
```

Along with 100 random customers, two users are generated. "adminuser@email.com" and "regularuser@email.com" with admin and not admin roles so you can test how the system behaves in a comfortable way.

8.Testing

- **symfony/phpunit-bridge** makes phpunit work nicely with symfony
- **symfony/browser-kit** let us hit the api like if we were a browser
- **dama/doctrine-test-bundle** tricks doctrine to revert every change made with every test so the database is unmodified.

With these 3 bundles installed we are ready to perform our test battery.

They live in the “tests/Controller” folder and they are going to let us hit every api endpoint in as many ways as we can imagine.

UserControllerTest

In the first test we generate a bunch of valid user jsons and we launch them to the POST User endpoint with an admin user and check that every single one of them is created.

Then we try to create another one with an existing mail to check that it fails and lastly we generate a lot of invalid jsons and launch them all to check that our asserts and ConstraintValidators are working correctly.

The second test tries to POST some valid users but this time logged as a non-admin user. The test goal is to check that our voters are working as expected and denies authorization for all the calls.

Third one checks that the user list can be retrieved by the admin but not for the regular user.

Last in this controller we create a new User and try to modify it with ‘PATCH’ calls. Check the admin can do it and not the regular user and that the modifications are reflected in the database. Same with ‘DELETE’ operations.

CustomerControllerTest

With customers we follow a similar strategy.

In the first test, we try to create a bunch of customers we know have valid data and we should receive a HTTP_CREATED code with every single call. Once we are done, we launch them all again to see how we get errors back because they all exist in the database.

Second test, we do it all again but this time with invalid data that we know should be rejected by our asserts and ValidationConstraints. 4xx codes are returned.

Third test starts by counting how many customers we have in the database and then starts fetching them all page by page and the count must match. This also tests that our serializer works with every existing customer and that the paginator system works as expected.

Fourth test checks that we can grab a customer from the database, modify it with a ‘PATCH’ call and the changes are reflected if we refresh the object from the database again.

The fifth is the same but with ‘DELETE’ operations.

Both the admin and regular users have full authorization over customer entities but all tests are launched for each one of them. Just to check this is true and in case modifications are made in the future.

MediaControllerTest

Again we test with both admin and not-admin users even if they should have the same authorization in this case.

The test here tries to upload a customer avatar from the “tests/Resources” and deletes it just right after. 200 and 204 (No content) responses are expected

9. The twelve factor app

First of all, the project is all tracked with a version control system (git). All its dependencies are handled by composer and you can take a look in the files composer.json and composer.lock to see which version of what dependency is exactly installed.

Composer.json defines the ruleset of which dependencies are needed and composer.lock contains the current installed exact versions.

Each deployment setup is configured in the env.local file, which is not tracked in git. Every variable defined in this file overrides the ones defined in .env.

All the backing services are referenced with as an url/connection string and the system can scale well just by configuring nginx to work with higher thread count among other measures. Also, all the system starts up and stops in a matter of seconds. As long as it takes nginx to stop or to boot.

Dev and Prod environment differences are minimal. Just a couple of different configurations in env files and config folders. All code is shared 99.9%.

Logs are treated as streams with components as monolog and one-off admin processes can be run with the symfony console commands running in the production machine.