Audit

Alexandre Gérault

December 2022

1 Quality

1.1 Initial quality of the application

The application is a web application that allows users to create and manage their own events. The application is developed in the context of a school project.

It uses a classic Symfony MVC architecture. It uses Symfony 3.1 and PHP until 5.5.9. As of 2022 these are outdated technologies.

This CodeClimate analysis reports that there are no technical debts used to code smells or duplication.

Though a PHPStan analysis with the maximum level compatible with the project version of PHP reports 5 errors:

```
$ ./vendor/bin/phpstan analyse src --level=7

 ------ ------------------------------------------------------
  Line   src/AppBundle/Controller/UserController.php
 ------ ------------------------------------------------------
  33     Call to an undefined method object::encodePassword().
  57     Call to an undefined method object::encodePassword().
 ------ ------------------------------------------------------

 ------ ---------------------------------------------------------------------
  Line   src/AppBundle/Controller/SecurityController.php
 ------ ---------------------------------------------------------------------
  18     Call to an undefined method object::getLastAuthenticationError().
  19     Call to an undefined method object::getLastUsername().
 ------ ---------------------------------------------------------------------

 ------ ------------------------------------------------------
  Line   src/AppBundle/Entity/Task.php
 ------ ------------------------------------------------------
  45     Class DateTime referenced with incorrect case: Datetime.
 ------ ------------------------------------------------------
[ERROR] Found 5 errors
```

Four of the five errors seem to be related to the use of the container instead of dependency injection, making the static analysis struggle to determine types.

Also, reading the source code reveals that services are accessed directly from the container instead of being injected. This is a bad practice that should be avoided nowadays.

1.2 Strategy to solve this issues

To solve these quality issues, we need to establish a strategy. The strategy is to:

1. Upgrade the application to the latest Symfony and the latest PHP versions
2. Fix the PHPStan errors
3. Increase the PHPStan level to the maximum (from 7 to 9)
4. Fix the new PHPStan errors
5. Fix the SRP violations by using repositories and voters for access control.
6. Check potential new errors from CodeClimate
7. Fix the new CodeClimate errors

1.3 Result

The last CodeClimate analysis can be read here (`https://codeclimate.com/github/AlexandreGerault/p8_todo_list`). We also applied stricter PHPStan rules and a PHP CS Fixer configuration to keep the code formatted in the same standard all over the project.

2 Performance

2.1 Database

2.1.1 N+1 issue

We can start to do some manual verfications across the application by reading the Database queries. One of the most common mistakes is too to meet a N+1 problem by lazy loading some entity relations. This is a common mistake when you are not used to Doctrine.

To solve this problem we can eager load the relationships instead, reducing the number of queries.

It does not seem to be any N+1 problem in the application.

2.1.2 Select

Also, when we query a lot of data, it can use a lot of memory and CPU. We can reduce the memory cost by selecting only the fields that we need.

2.1.3 Indexes

When we query a lot of data, it can use a lot of memories and CPU. We can reduce the CPU cost by adding indexes to the columns that are usually used to query data. We have to keep in mind that indexes increases speed for querying data but also decreases it when inserting data.

2.2 Profiling

We can do some profiling of the application using a professional tool like Blackfire. The profiler allows us to get information about what parts of the application use too much CPU or costs too much memory.

2.2.1 Comparing legacy results to rewritten application results

We can do some profiling of the application using a professional tool like Blackfire. The profiler allows us to get information about what parts of the application use too much CPU or costs too much memory.

2.2.2 Comparing legacy results to rewritten application results

When comparing results of the new application, using a newer version of symfony and PHP, we realise that the application is usually slower and uses more memory. At first, we could think we're doing something wrong, but we have to keep in mind that the new application is using a newer version of Symfony. When we look at the graph, we can see that the only parts that require more use of CPU and memory aren't application code, but code from the autoloader or Symfony core.

This can partially be explained by the use of the reflection API in the new IoC from Symfony 4. Indeed, even if it decreases the performance a bit, this is nothing critical. As compensation, the productivity of developers is greatly improved.