So, using this API as an example, I want to show a modern PHP architecture for highly loaded projects. When the project is still at the very beginning, and not that the business logic (relationship with the database) is not spelled out, but the business model itself is not very clear, building an effective IT architecture can only go one way: it is necessary to strictly separate the frontend and backend.

What did you usually do in such situations two or three years ago? A monolithic framework like Laravel or Yii2 was taken, the whole business model was divided, at least, into blocks, and these blocks were already implemented as framework modules. As a result, after another 3 years, a huge non-rotating machine was obtained, which in itself is slow, but became almost unbearably slow, in which the frontend is rendered through the backend using the classic MVC architecture (the user sent a request, the controller picked it up, called the model, that into its the queue did something there with the database, returned everything to the controller, and the controller finally called the viewer, inserted the data from the model into it and gave it all to the user who had already opened another can of beer ...). And ... well, even especially advanced guys, they did not just view Tweeter Bootstrap, but they actually used very good libraries like JQuery into the viewer or used some frontend framework. As a result, it became more and more difficult to maintain such a Monster Truck, and it was very difficult to integrate a new programmer to the team, because not everyone is born Einstein. Let's add here the total lack of developer documentation (you read the comments in 9000 files - everything is there!) And in the end, looking at everything, it became really sad ...

But then a number of events took place that radically changed the situation. First, finally, PSR standards came out and Symfony suddenly ceased to be the only modular framework. secondly, ReactJS was released, which made it possible to fully separate the frontend from the backend and force them to communicate through the API. And finishing the last nail in the coffin of the old development system (MVC is our everything!) OpenAPI 3.0 comes out, in fact, which regulates the standards of this communication through the API between the frontend and backend.

And in the PHP world it became possible to do the following:

1. Divide the business model into services and micro services, and not raise the entire Monster Truck for this, but serve micro service requests literally in a couple of lines of code - the most common example: similar products (a separate GET request is a separate, tiny API that processed it and returned it, and the immediate output of this ReactJS information in the user's browser. The main Monster Truck did not even know what happened ...

2. Writing an API is standardized according to the OpenAPI 3.0 standard (https://swagger.io/) in the form of a YAML or JSON file, when every programmer does not climb into the core of the system in dirty boots, and for example, culturally adds his part in the general YAML file. Thereby eliminating the chance of error from person to person and reducing the amount of gray hair on testers. Just then, from the YAML file, it is possible generated a fully working server, and even with a middleware. In any language and framework you like.

3. Now there is no need to hire someone to do it, this someone wrote for your API libraries that your clients will use to access your API: https://github.com/OpenAPITools/openapi-generator - I counted the generation more than 40 servers for the API and did not even consider the libraries for accessing them, because the only programming language that I did not find there is Dlang)

So, I think we figured out the API. We write a YAML file in a swagger or Insomnia, through OpenAPITools we generate a server and user libraries. Please do not touch the abstract classes (/lib), but move all business logic into inherited classes (/scr), so that during the subsequent regeneration of the server we do not break anything, but simply copied the /lib to our framework root and added new business logic to the relocatable /scr. API ready - Fast, simple, functional. The client libraries are ready too. The director did not even have time to return from the Maldives...

Now there are questions, or rather two questions: what do we have in front of the API, and what we have "under the tail" after the API.

Answers:
1. "There, far beyond the river", far in front of the API we are blooming, spreading to new functionality and pictures - FRONTEND (ReactJS is preferable, but Vue will do too. Although there is so much out of the box that it will burden the process, but how much in real life it will be needed - not entirely clear and depends directly on the business model). AND YES! I won't even come close to this beast, because since childhood I have been allergic to it. A separate specialist is needed here. I am not a full stack and I do not write a frontend.

2. Right here in front of the API itself, we have .... DID NOT GUESS ... not NGINX, but RoadRunner https://roadrunner.dev/features. We go in, read, understand that it is faster and the walkers are signed by the number of processors, and therefore there will never be a sign "WE ARE IN PREVENTION", because you just need to switch the walkers.

And I want to dwell on this point in more detail. For in my understanding there are three ways "how to catch flies", requests, that is ...

1. If the entire API has already been written, and will be written in the future, in PHP - there is no need to break your head, install RoadRunner with https://prometheus.io
2. If the system is assembled from different pieces, different services are written in different languages and then it is also not clear what they will be written in:
2.1. Install NGINX UNIT - use supported languages.
2.2. Lift ANY system of containers, Docker, LXC, LXD. The choice again depends on the size of the project - to support the assembly of PROXMOX-LXC on hosting with 12 processors, with 32GB of memory, for 40 euros per month will be several times cheaper than Docker assemblies on the Google Cloud Platform. Put a server suitable for the language in each container, and link all this with HAProxy http://www.haproxy.org. HAProxy is a gorgeous balancer and proxy server that is no less popular in a corporate environment than NGINX. What it does and what it does not read here https://cbonte.github.io/haproxy-dconv/2.3/intro.html paragraph 3.1. With this architecture, services or microservices can be written on any language and no one depends on the restrictions imposed by RoadRunner or NGINX UNIT.

3. "Under the tail" - Cycle ORM. Not be lazy - watch the video (!Russian language) https://www.youtube.com/watch?v=o1wzzSoJJHg&ab_channel=fwdays, what exactly MySQL or Postgres will be behind it - again, I would leave on after the business scheme of the project is clear. MySQL scales easier, Postgres has more business logic inside the database itself.

4. An example that you can see and touch - https://bitbucket.org/rumatakira/api-example/src/master/. There, a test task is taken as a basis. All the most useful things are in the EXTRAS folder. There is already a generator jar file, a YAML swagger API file, generated by the API stub via OpenAPITools in SLIM4. Even with authentication and middleware. API documentation generated by swagger, not OpenAPITools. It is assumed that some users are logged in and have a token. There is already RoadRunner in front. The stack is PHP 7.4.10, PostgreSQL 12.4.

After Git clone, composer install, in the /bootstrap.php file, write the user and the password to the database, by default the server listen the local port 8888, if necessary, change it in the /.rr.yaml file, and run the command: composer run-script fill-database. Thats all - no migrations)).

Yours faithfully,
Kirill Lapchinsky
api-studio.com
mail@api-studio.com (working for clients)
rumatakira74@gmail.com (personal)
telegram @rumatakira