

Итак, на примере этого API, я хочу показать современную PHP архитектуру для высоконагруженных проектов. Когда проект еще в самом начале, и не то, что бизнес-логика (взаимоотношения с базой данных) не прописана, но и сама бизнес-модель не очень ясна, построение эффективной IT архитектуры может идти только одним путем: **необходимо** жестко разделить frontend и backend.

Что обычно делали в таких ситуациях два-три года назад? Брался монолитный фреймворк типа Laravel или Yii2, вся бизнес-модель разбивалась, худо-бедно, на блоки, а эти блоки уже имплементировались как модули фреймворка. В итоге еще через 3 года получалась огромная не поворотная машина, которая сама по себе медленная, а становилась почти невыносимо медленной, в которой фронтенд рендился через бэкенд по средствам классической MVC архитектуры (пользователь отправил запрос, контроллер его подхватил, вызвал модель, та в свою очередь чего-то там натворила с базой данных, вернула все контроллеру, а тот наконец-то вызвал вьювер, вставил туда данные из модели и отдал это все пользователю, который уже успел открыть очередную банку пива...). А... ну еще особо продвинутые ребята они не просто вьюверели Tweeter Bootstrap, а во вьювер вкручивали самом деле очень хорошие библиотеки типа JQuery или вместо вьювера использовали какой-нибудь фронтенд фреймворк. В итоге поддерживать такой BeLa3 становилось все сложнее, а ввести нового программиста в команду было очень сложно, ибо не все рождаются Эйнштейнами. Добавим сюда тотальное отсутствие документации разработчика (камменты в 9000 файлах считаешь - там все есть!) и в итоге, смотря на это все, становилось по-настоящему грустно...

Но тут произошли ряд событий, которые в корне изменили ситуацию. Во-первых, наконец-то, вышли стандарты PSR и Symfony внезапно перестал быть единственным модульным фреймворком. во-вторых, вышел ReactJS, который позволил полноценно разделить фронтенд от бэкенда и заставить их общаться через API. И добивая последний гвоздь в крышку гроба старой системы разработки (MVC - это наше все!) выходит OpenAPI 3.0, собственно который и регулирует стандарты этого общения через API между фронтендом и бэкендом.

И в мире PHP стало возможно делать следующее:

1. Разделить бизнес-модель на сервисы и микросервисы, и не поднимать для этого весь BeLa3, а обслуживать запросы микросервисов буквально в пару строк кода - самый банальный пример: похожие товары (отдельный запрос GET - отдельный, малюпаський API, который его обработал и вернул, и мгновенный вывод этой информации ReactJS в браузере пользователя. Основной BeLa3 даже и не узнал о том, что произошло...
2. Писать API стандартизировано по стандарту OpenAPI 3.0 (<https://swagger.io/>) в виде YAML или JSON файла, когда каждый программист не лезет в грязных сапогах в ядро системы, а например, культурно дописывает свою часть в общий YAML файл. тем самым устраняя вероятность ошибки от человека к человеку и уменьшая количество седых волос у тестировщиков. Просто потом из готового YAML сгенерировал полностью рабочий и даже с мидлваре сервер. На каком угодно языке и фреймворке.
3. Теперь не надо стало нанимать кого-то, чтобы он, этот кто-то писал для вашего API библиотеки, которыми ваши клиенты будут обращаться к вашему API:
<https://github.com/OpenAPITools/openapi-generator> - я насчитал генерацию более 40 серверов для API и даже не стал считать библиотеки для доступа к ним, ибо единственный язык программирования который я там не нашел - Dlang)

Итак с API думаю разобрались. Пишем YAML файл в сваггере или инсонии, через OpenAPITools генерируем сервер и пользовательские библиотеки. Абстрактные классы не

трогаем (папочка lib) а всю бизнеслогику выносим в наследуемые классы (папочка scr), для того чтобы при последующей регенерации сервера мы ничего не сломали, а просто тупо скопировали папочку lib к себе в корень фреймворка и добавили новую бизнес логику в не перемещаемой папочке scr. API готов - Быстро, просто, функционально. Клиентский библиотеки тоже готовы. Директор даже не успел вернуться с Мальдивов....

Теперь становится вопросы, точнее два вопроса, а что у нас перед API и соответственно, что у нас "под хвостом" после API.

Ответы:

1. "Там вдали за рекой", далеко перед API у нас цветет, расползается на новую функциональность и картинки - ФРОНТЕНД (Предпочтительнее ReactJS, но Vue тоже сойдет. Хотя там из коробки всего столько много, что утяжелять процесс он будет, а вот насколько в реальной жизни это понадобится - не совсем понятно и зависит напрямую от бизнес модели). И НЕТ! Я к этому зверю даже близко подходить не буду, ибо с детства у меня на него аллергия. Тут нужен отдельный специалист. Я не фулстак и не пишу фронтенд.

2. Прямо вот перед самим API у нас.... НЕ УГАДАЛИ... не NGINX, а RoadRunner <https://roadrunner.dev/features> . Заходим, читаем, понимаем, что это быстрее и вокееры расписываются по количеству процессоров, а посему никогда не будет таблички "МЫ НА ПРОФИЛАКТИКЕ", ибо просто надо вокееры переключить.

И на этом моменте хочу остановиться подробнее. Ибо в моем понимании есть три пути "как мух ловить", запросы то бишь...

1. В случае если весь API написан уже, и будет написан в дальнейшем, на PHP - голову ломать не зачем, ставим RoadRunner с <https://prometheus.io>

2. В случае если система собирается из разных кусков, разные сервисы написаны на разных языках и дальше тоже не понятно на чем их писать будут:

2.1. Ставим NGINX UNIT - пользуемся поддерживаемыми языками.

2.2. Поднимаем ВСЕ РАВНО КАКУЮ систему контейнеров, Docker, LXC, LXD. Выбор опять же зависит от размера проекта - поддерживать сборку PROXMOX-LXC на хостинге в 12 процессоров, с 32Гб памяти, за 40 евро в месяц будет в разы дешевле, чем Docker сборки на Google Cloud Platform. В каждый контейнер ставим подходящий к языку сервер, и связываем все это HAProxy <http://www.haproxy.org> . HAProxy - шикарный балансир и прокси сервер который в корпоративной среде, не менее популярен чем NGINX. Что он делает, а чего нет читаем тут <https://cbonte.github.io/haproxy-dconv/2.3/intro.html> пункт 3.1. При такой архитектуре сервисы или микросервисы могут писаться на чем угодно и никто не зависит от ограничений накладываемыми RoadRunner или NGINX UNIT.

3. "Под хвостом" - Cycle ORM. Не ленимся смотрим видео -

https://www.youtube.com/watch?v=o1wzzSoJJHg&ab_channel=fwdays , что будет стоять за ней конкретно MySQL или Postgres - опять, я бы оставил на после того, как будет понятна бизнес схема проекта. MySQL проще масштабируется, в Postgres - больше бизнес логики перенесенной внутрь самой базы.

4. Пример который можно посмотреть и пощупать -

<https://bitbucket.org/rumatakira/api-example/src/master/> . Там за основу взято тестовое задание. Все самые полезные вещи находятся в папке EXTRAS. Там уже сть jar file генератора, YAML swagger файл API, сгенерированный API stub через OpenAPITools в SLIM4. Даже с аутентификацией и мидлваре. документация на API сгенерированная, правда swagger, не

OpenAPITools. Предполагается, что некоторые юзеры залогинены и им выдан токен. Там уже стоит RoadRunner впереди. Стек - PHP 7.4.10, PostgreSQL 12.4.

После Git clone, composer install в файле /bootstrap.php прописываем юзера и пароль к базе, которую вначале создаем, потому что это PostgreSQL, по умолчанию сервер слушает локальный порт 8888, если нужно - меняем в файле /.rr.yaml, и выполняем команду: composer run-script fill-database . Все - никаких миграций)). Пользуемся.

P.S. Всем залогиненым пользователям присвоен одинаковый токен. Вообще в примере нет никаких валидаций ввода пользователя и почти нет защит - это пример в основном нацеленный на архитектуру.

С уважением,
Кирилл Лапчинский
api-studio.com
mail@api-studio.com (рабочий для клиентов)
rumatakira74@gmail.com (личный)
telegram @rumatakira