Efektywne skalowanie
aplikacji
monolitycznej Symfony7

Inni programiści **wyśmiewają** nas!

Mówią, że
statystyki **php nabija**
tylko
wordpress

Opluwają nas,

że php jest

wolny

Szydzą,

że php **nie umie** dobrze

pracować wielowątkowo

Wytykają nam,

że długo żyjące procesy w php

umierają

bez zapowiedzi

Kpią z nas,

że

php nie żyje

już od 10 lat.

A tak naprawdę to Oni nam

zazdroszczą!

Bo nam błędy zgłasza

klient,

a im

KOMPILATOR!

Dzień dobry!

Jestem
Max.

I mam

zaszczyt,

Już

trzeci rok

z rzędu stać tutaj przed wami.

I nawijać farmazony.

Dzisiaj opowiem wam...

Jak **wyMaxować** monolit,

żeby robił:

Łu tu tu!

Jesteście

gotowi?

Na

głupie żarty

prowadzącego

i

żenujące meme’y?

Na wstępie muszę komuś

podziękować.

Wczoraj spotkałem się z

trzynastoma

wyjątkowo odważnymi ludźmi.

Którzy to ludzie zdecydowali się spędzić ze mną jedno całe popołudnie na warsztacie.

Narażając się na wieloletnią

terapię.

Robiliśmy takie eksperymenty...


PHP

że **trauma** zostanie z nimi na długo.

A oni jeszcze za to **zapłacili** **$$$**.

Byliście wielcy!

Dzięki!

Let’s roll!

Efektywne skalowanie aplikacji monolitycznej Symfony7

O czym nie będę mówić podczas tej prezentacji?

O mikroserwisach.

Prawie wcale.

O optymalizacji zapytań SQL.

O denormalizacji tabel w bazie danych.

O mongodb.

No I napewno o laravelu nic nie powiem;

dobrego ;)

Na początek zdekodujmy sobie tytuł prezentacji.

Efektywność

Skalowanie

Aplikacja Monolityczna

To w skrócie:

Jak pragmatycznie zmaksymalizować wydajność aplikacji sf7, którą wdraża się za pomocą jednej jednostki wdrożeniowej?

W większym skrócie:

Zbijamy czas renderu response, żeby first byte sent time był jak najkrótszy, bez wydawania fortuny na infrastrukturę.

Panie co pan mi tu imputuje?!

Tylko mikroserwisy!

Z mikroserwisami, to może być

pewien problem.

Serio serio.

I to żeby tylko jeden

problem.

Primo:

Mieliśmy być efektywni, nie efektowni

Secundo:

Mikroserwisy potrafią podnieść koszty operacyjne o

50%


MICROSERVICES
WILL BE DEPLOYED
imgflip.com

Badania mówią, że bez

2 milionów USD

przychodu rocznie nie mamy tu czego szukać.

To gdzie tu ta

efektywność?

A gwarantuje, że to dopiero

początek problemów.

Bo debuggowanie systemów rozproszonych nie należy do przyjemnych.

To jak?

Możemy już skończyć z mikroserwisami?

Wracając do monolitu.

Co

maxować

najpierw?

A co ogranicza nas najbardziej?

Infrastruktura?

Kod aplikacji?

Architektura Aplikacji?

Legacy?

Budżet?

Kompetencje zespołu?

Ja osobiście bym zaczął…

Tam gdzie najtaniej!

Zacznijmy od rewizji infrastruktury,
może gdzieś leży coś za darmo.

Często pomijamy oczywiste, oczywistości.

A nie zawsze przychodzi nam wdrożyć projekt na wymarzoną infrastrukturę.

Bo klient ma admina,

który ma szwagra,

który ma brata

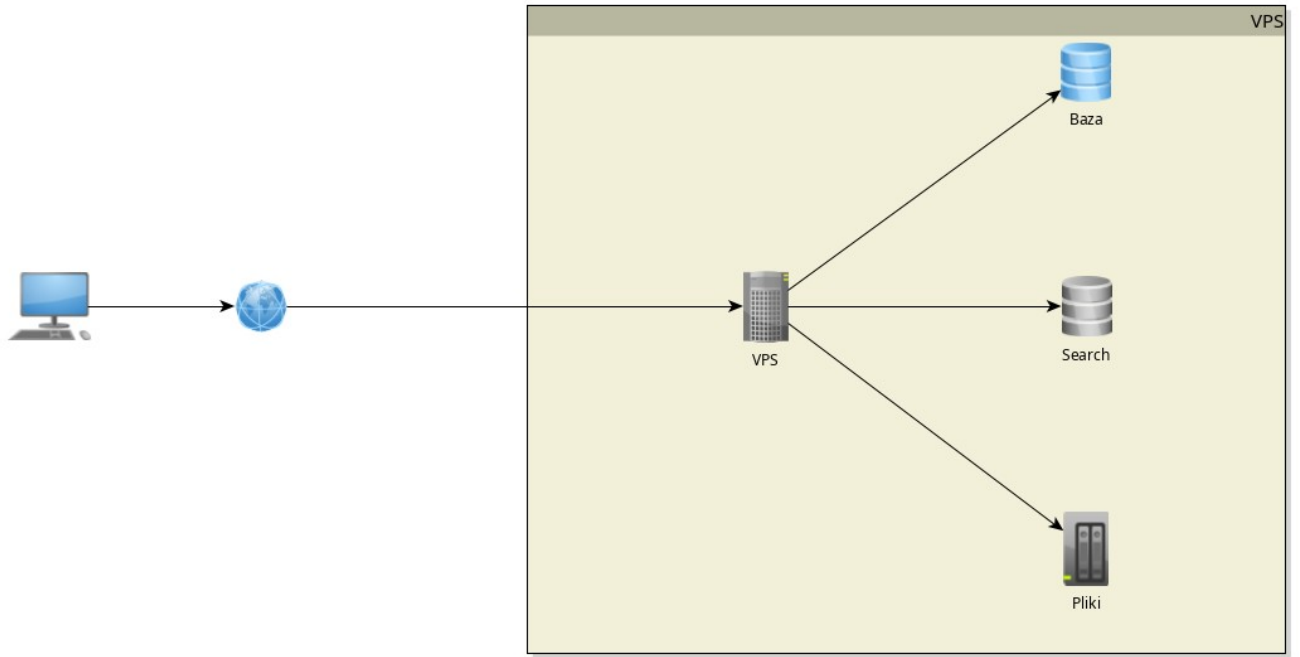
i właśnie ten kumpel z klasy jego brata,
co jedździ Golfem IV.

Postawił wam ten serwer.

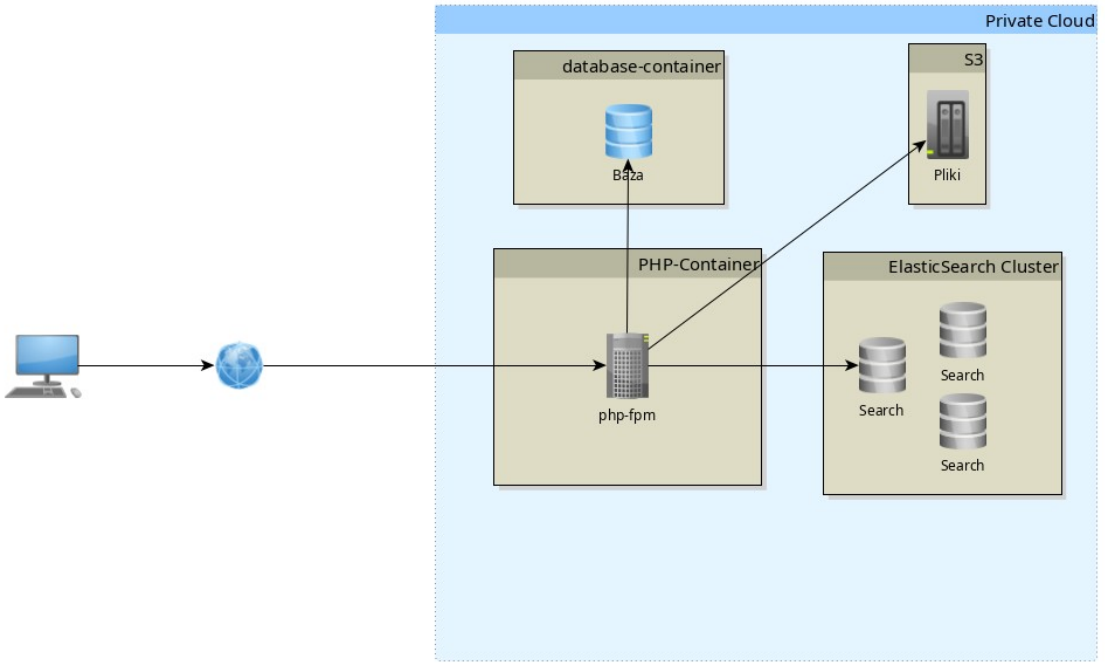
Tier lista biedy

ifrastrukturalnej!

Ukradli mi DEVa

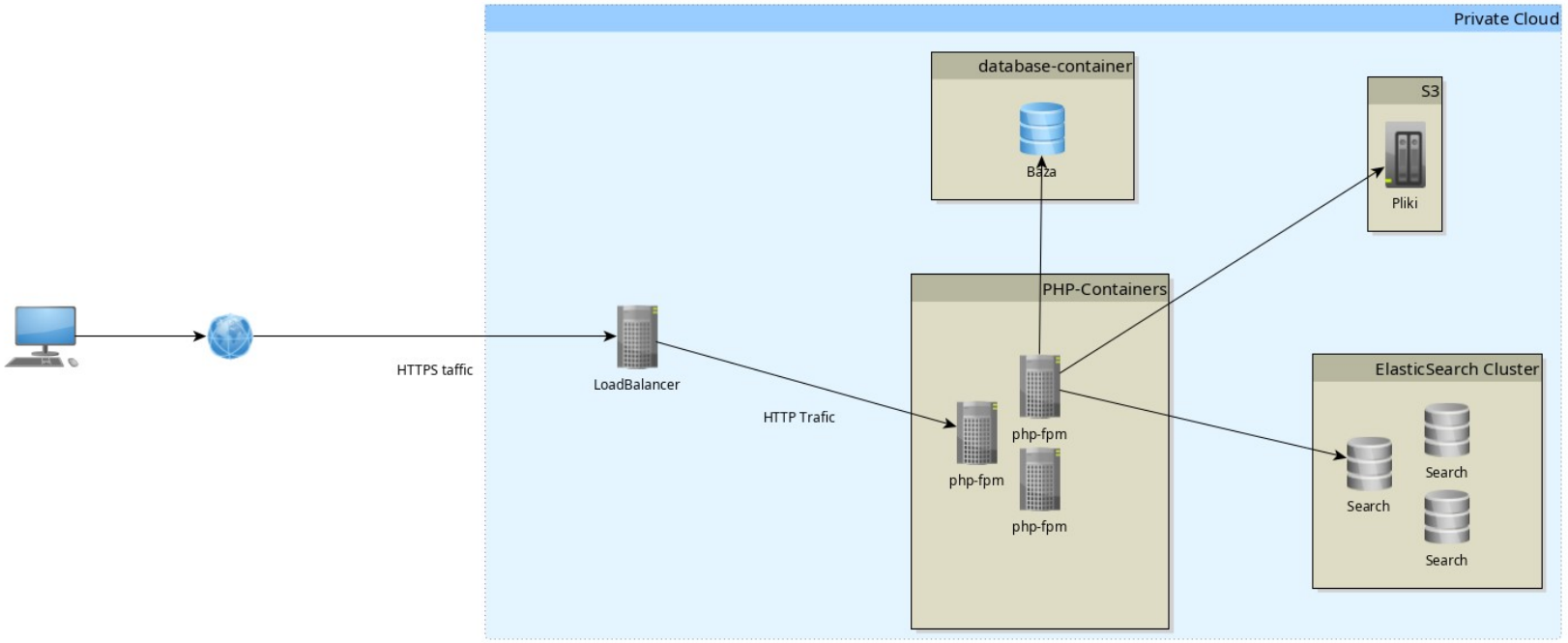


Konteneryzowany RAW z Bazy



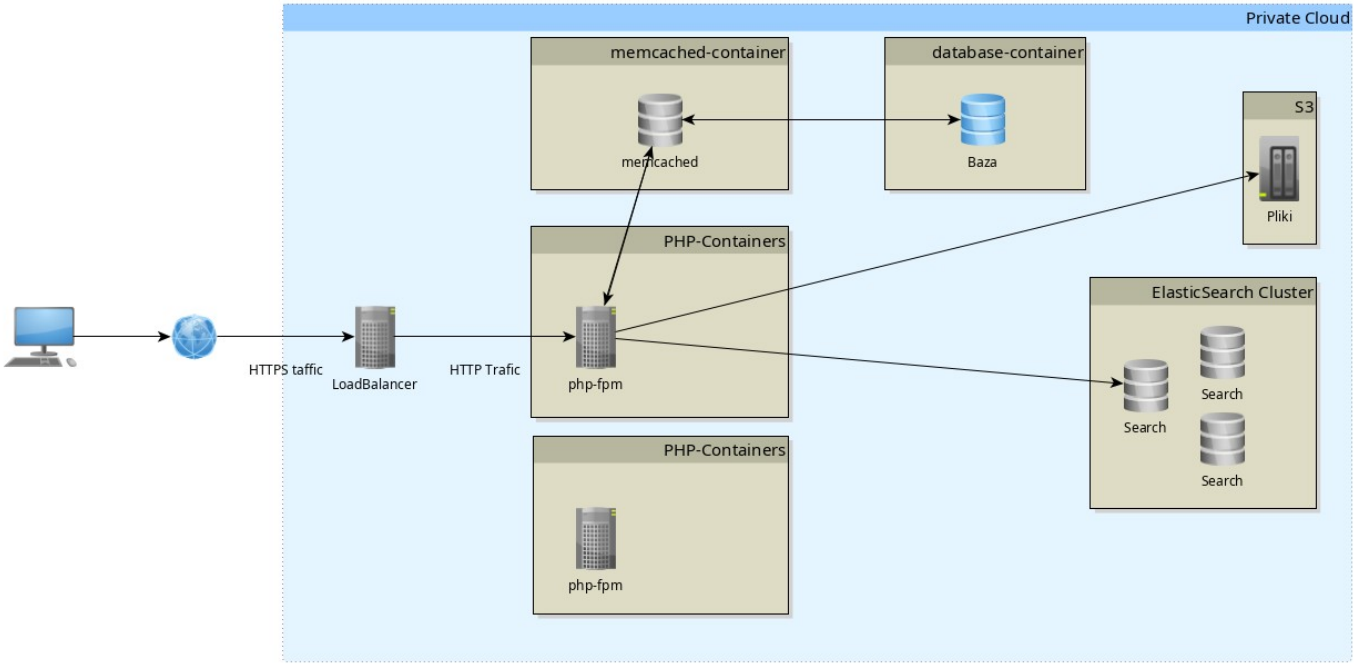
Jak wygląda loadbalancing?

Zbalansowany RAW z bazy



Czy w naszym systemie działa jakiś result chache?

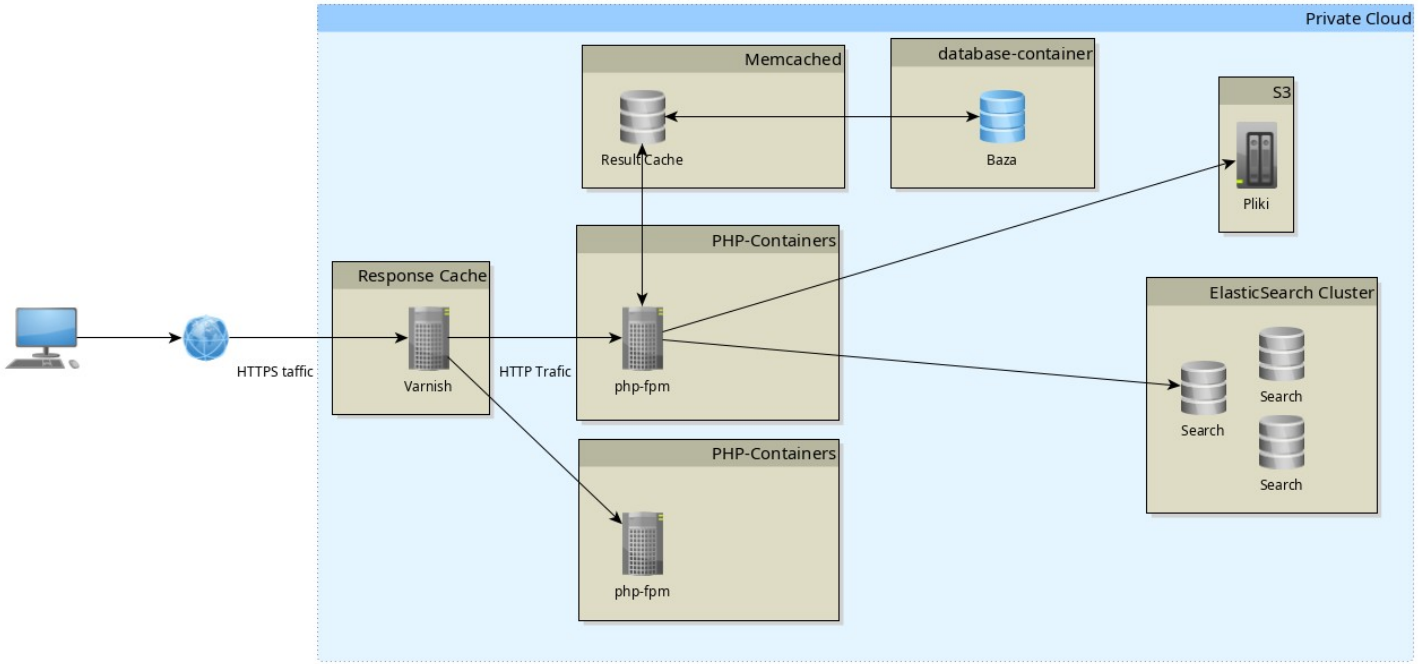
Tato dowiedziałem się o memcached



Czy używamy headerów „etag” i „Cache-control”?

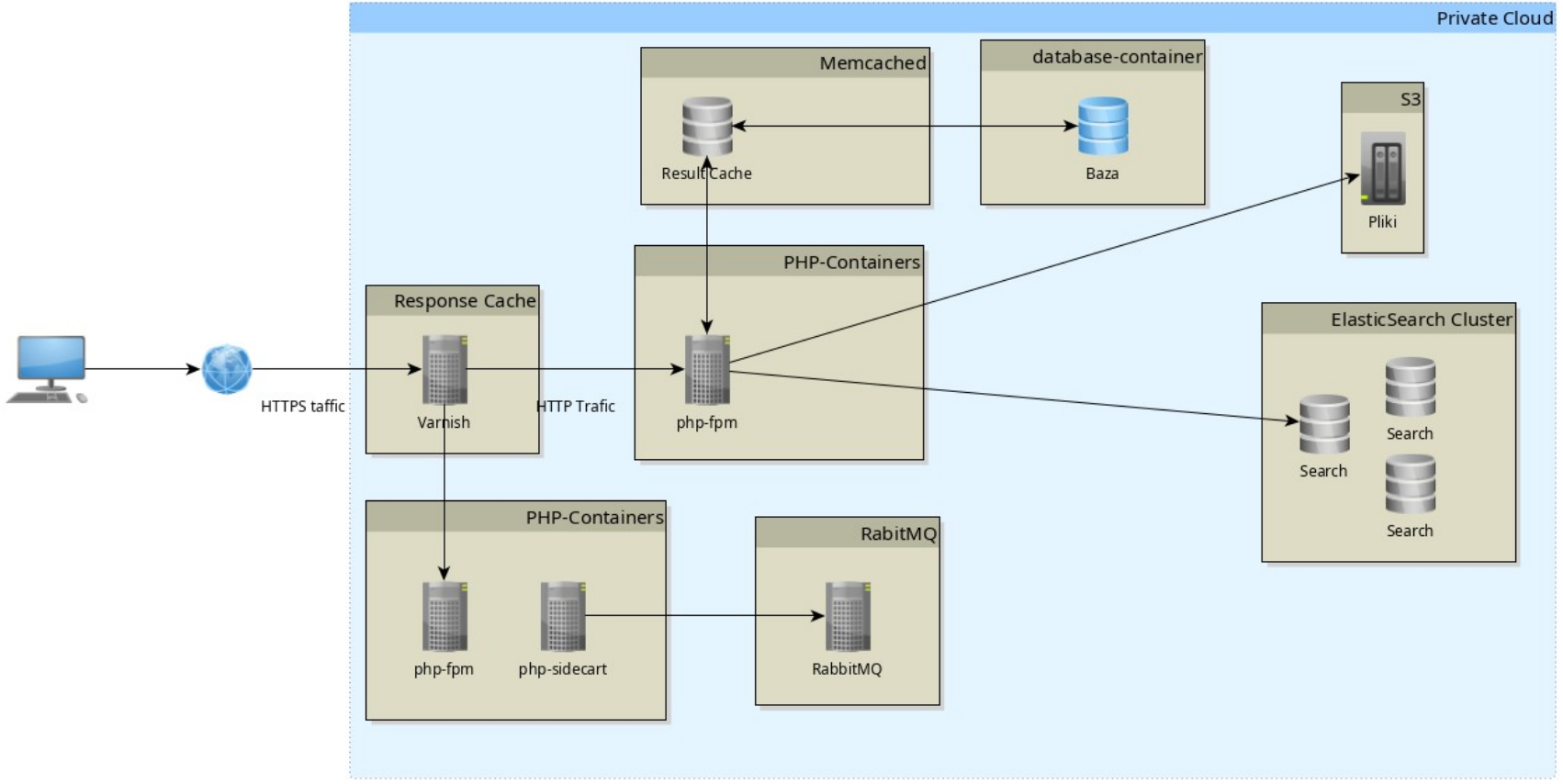
To może Varnish.

Tak. Response cahce działa w REST’ach



Może jakiś worker / cron / consumer pracuje na tej samej instancji?

Asynchroniczna Konsumpcja na delegacji



Niestety tu nie wszystko od nas zależy.

- Dopasuj rozwiązanie do budżetu klienta przeznaczonego na hostowanie aplikacji.
- Jeżeli widzisz że aplikacja się dusi, negocjuj rozbudowę architektury.
- Zrób stress-testy żeby określić miejsce gdzie aplikacja umrze. Jasno zakomunikuj klientowi gdzie znajduje się punkt krytyczny wydajności.

Skalowanie infrastruktury jest efektywne kosztowo.

Przy założeniu, że godzina pracy serwera jest tańsza od godziny pracy programisty.

Trzeba zabrać się za

benchmark.

Cel benchmarku?

Odnalezienie **Bottlenecków** na poziomie infrastruktury.

Benchmark / Load Test

- Przeanalizuj logi i znajdź miejsca, których czas odpowiedzi odbiega od pozostałych.
- Napisz scenariusz load testu, tak by pokrył część krytycznych scieżek systemu oraz dotknął najbardziej obciążonych miejsc.
- Wybierz termin, który nie sparaliżuje biznesu klienta i poinformuj go o planowanym teście.

Benchmark / Load Test

- 

Interpretacja Wyników testu

- Access log to złoto przy identyfikacji wolno odpowiadających kontrolerów.
- Error log to złoto przy identyfikacji umierających usług.

Wnioski po benchmarku:

1) Jeśli baza jest przeciążona, to brakuje Cache
2) nginx jest przeciążony powinniśmy zadbać o C.D.N.
3) php-fpm – musimy sprawdzić co się dzieje z logiką aplikacji

Dosyć!

Tej infrastruktury.

Suchary siadły?

Mocne

3/10?

Tak myślałem.

Kolejna rzecz za darmo:

Audyt konfiguracji.

O konfiguracji PHP8 mówiłem już rok temu.

Kto był?

Kto słuchał?

A kto usłyszał?

No to czas

przypomnieć!

1. Wyłącz xdebug na prod.

+5%

2. Composer autoloader

+2%

```
composer install
        --no-progress
        --no-interaction
        --no-dev
        --classmap-authoritative
        --prefer-dist
```

Jeśli posiadasz włączone APCu :

```
composer install
        --no-progress
        --no-interaction
        --no-dev
        --apcu-autoloader
        --prefer-dist
```

3. Włącz Opcache

+10%

```
opcache.enable=1
opcache.revalidate_freq=10
opcache.validate_timestamps=0
opcache.max_accelerated_files=20000
opcache.memory_consumption=256
opcache.max_wasted_percentage=10
opcache.interned_strings_buffer=1
opcache.fast_shutdown=1
```

4. Włącz Preload

+2%

```
opcache.preload=/path/to/project/
config/preload.php
opcache.preload_user=www-data
```

5. Włącz i skonfiguruj JIT

+2%

```
opcache.jit_buffer_size=256M
opcache.jit=1255
opcache.jit=tracing
```

6. Sprawdź czy twoje consumery oraz workery pracują w “--env=prod”

7. Popraw ustawienia pm w php-fpm

```
pm.max_children = 10
pm.start_servers = 5
pm.min_spare_servers = 2
pm.max_spare_servers = 6
pm.max_requests = 1000
```

8. Sprawdź czy coś nie sieje Warningami, Deprecated po logach.

10. Włącz cache realpath

+1%

```
realpath_cache_size=4096K
realpath_cache_ttl=600
```

Potencjalne przyrosty wydajności

Wniosek z tego jest taki:

Często zabieramy sobie **~20-30%**

wydajności aplikacji przez błędy / zaniechania w konfiguracji.

Dosyć darmochy.

Czas zabrać się za

Aplikacje

Poniekąd,

po to tu przyszliście.

Profilowanie!

Stress test dał nam dane:

1) Access Log
2) Średni czas response.

Dzięki temu mamy punkt zaczepienia dla profilera.

Zaczynamy od bezkosztowej opcji:

Symfony Profiler

Rozpoczynamy od najdłuższych response time.

Przeglądamy timeline

Performance metrics

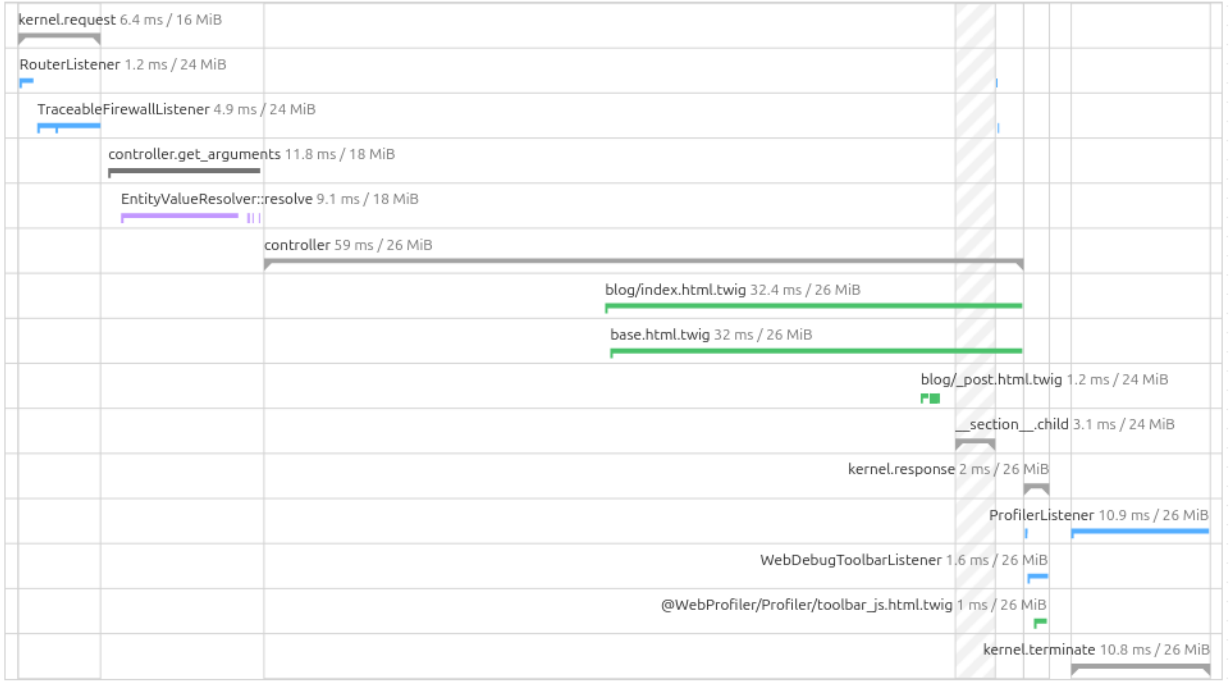
Total execution time	Symfony initialization	Peak memory usage	Sub-Request	Sub-Request time
135 ms	44 ms	28.00 MiB	1	3.1 ms

Execution timeline

Threshold 1 ms (timeline only displays events with a duration longer than this threshold)

Main Request 90.8 ms

default · section · event_listener · template · doctrine · controller.argument_value_resolver



Co kosztowało nas najwięcej czasu?

Logika kontrolera?

Wykonanie zapytania?

Serializacja odpowiedzi?

Może jakiś middleware?

Na localu chodzi dobrze. WTF?

Za mało danych.
Dump z produkcji rozwiąże sytuacje.

We need to dig

deeper.

xhprof

~~Instalacja przez paczkę z PECL'a~~

R.I.P.

xhprof from PECL
Php < 8.0
Never forgetti.

Zajęło mi to jakieś...

żeby zrozumieć dlaczego composer nie działa na dockerze po instalacji tego pecla.

Peszek.

Xdebug3

FTW!

Konfigurujemy zbieranie danych profilera w php.ini kontenera.

```
xdebug.mode = profile
xdebug.start_with_request = trigger
xdebug.output_dir = /tmp/profiles
xdebug.profiler_output_name = cachegrind.out.%p.%R.%u
```

Ustawiamy ciastko albo parametr get żeby wywołać profilowanie.

W katalogu
/tmp/profiles utworzony został plik cachegrind.

Teraz ten plik wrzucamy do Kcachegrind albo PhpStorma w Tools

Server: ... Time: ms REFRESH

Execution Statistics | Call Tree

Callable	Time	Own Time	Memory (B)	Own Memory (B)	Calls
/var/www/symfony/public/index.php	258 100.0%	0 0.0%	5,388,272	464	1 0.0%
require_once::/var/www/symfony/vendor/autoload_runtime.php	258 100.0%	0 0.1%	5,494,520	0	1 0.0%
Symfony\Component\Runtime\Runner\Symfony\HttpKernelRunner->run	256 99.1%	0 0.1%	5,334,704	0	1 0.0%
Symfony\Component\HttpKernel\Kernel->handle	204 78.9%	0 0.1%	3,718,400	0	1 0.0%
Symfony\Component\HttpKernel\HttpKernel->handle	191 73.8%	0 0.0%	3,502,192	0	2 0.0%
Symfony\Component\HttpKernel\HttpKernel->handleRaw	191 73.8%	0 0.0%	3,503,640	0	2 0.0%
Twig\Environment->render	139 53.8%	0 0.0%	2,521,520	0	4 0.0%
Twig\TemplateWrapper->render	138 53.5%	0 0.3%	2,526,480	0	10 0.0%
Twig\Template->render	138 53.3%	1 0.5%	2,792,568	48,040	10 0.0%
Twig\Template->yield	137 53.2%	1 0.7%	2,750,272	16,736	469 0.4%

Callees | Callers

Callable	Time	Calls
Symfony\Component\HttpKernel\HttpKernel->handleRaw	191 73.8%	2 0.0%
Symfony\Component\HttpKernel\Controller\TraceableControllerResolver->getController	0 0.2%	2 0.0%
Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent->getController	0 0.0%	2 0.0%
Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent->__construct	0 0.0%	2 0.0%
Symfony\Component\HttpKernel\HttpKernel->filterResponse	19 7.5%	2 0.0%
Symfony\Component\HttpKernel\Event\ControllerArgumentsEvent->getArguments	0 0.0%	2 0.0%
App\Controller\BlogController->postShow	135 52.4%	1 0.0%
Symfony\Component\EventDispatcher\Debug\TraceableEventDispatcher->dispatch	94 36.4%	16 0.0%
Symfony\Component\HttpKernel\Controller\TraceableArgumentResolver->getArguments	31 12.3%	2 0.0%
Symfony\Component\HttpKernel\Event\ControllerEvent->__construct	0 0.0%	2 0.0%
Symfony\Bundle\FrameworkBundle\Controller\TemplateController->templateAction	1 0.6%	1 0.0%

Teraz przeglądamy call stack,
szukamy wywołań które zajęły

dużo czasu

lub/i

dużo zasobów cpu.

et voilà !

Mamy **bottleneck** zlokalizowany!
Poprawiamy i możemy jechać z nim na **produkcje**.

Takie proste to jest zawsze na slajdach?

Przy profilowaniu, trzeba nastawić się na

cierpienie.

Bo na tym miejscu zbrodni jesteś:
detektywem, sprawcą i ofiarą

jednocześnie

A jak monitorować tę produkcję?

Najtaniej jest: Analizować logi.

Drugi Najtańszy będzie selfhosted

Prometheus

Jak go zintegrować z Symfony?

https://github.com/artprima/prometheus-metrics-bundle

Używa redisa lub Apcu do trzymania metryk

Jeżeli chodzi o płatne narzędzia, to Datadog jest przy pewniej skali doskonałym rozwiązaniem.

Konfiguracja observability, jest dość żmudna.

Alarmy, które możesz zintegrować ze Slackiem, często ratują sytuacje.

A co z Symfony Command w CLI?

Tam też są bottlenecki?

Tak! I to całkiem sporo.

Output to Twój wróg.

$em→flush();
też.

Jak komenda odpalana jest przez crontab, może nie zakończyć się przed jej kolejnym wywołaniem.

Same problemy!

My tu przyszliśmy po rozwiązania.

Ogranicz verbosity

Dodaj do komendy parametr -v.

A następnie wytnij ifami cały output, gdy parametr nie jest obecny w wywołaniu

Batch processing

Zawsze w komendzie, posiadaj parametry limit oraz offset.
Zapisuj paczki.
Flushuj co 50,100,200 encji.

Dobieranie rozmiaru batch’a

Pamiętaj że im większy batch tym więcej pamięci zje komenda.

Zbijamy tym czas oczekiwania na nową transakcje, kosztem pamięci Ram.

Wielowątkowość

Twoja komenda ma zrobić coś szybko?

Pracuje na osobnym workerze, który ma kilka rdzeni, które się nudzą?

Sprawdź Process Component w Symfony.

Dzięki niemu możesz, utwożyć fork dowolnej komendy.

```
$process = new Process(['ls', '-lsa']);
$process->start();

while ($process->isRunning()) {
    // waiting for process to finish
}

echo $process->getOutput();
```

Możesz rozpocząć ich tyle ile masz dostępnych wątków w procesorze.

```
$threads =8;
$perPage = ceil(count($users) / $threads);
$processes = [];
$command = 'php app/console user:duplicates:merge --env=prod --
flush=true %s';
for($i=0; $i < $threads; $i++) {
    $skip = $i * $perPage;
    $commandUsers = implode(
      ' ',
      array_slice($users, $skip,    $perPage)
    );
    $spawn = sprintf($command, $commandUsers);
    $process = new Process($spawn);
    $process->start();
    $processes[] = $process;
}
```

Teraz tylko musimy sprawdzić czy wszystkie procesy się zakończyły.

```
$start = new \DateTime();
while (false === empty($processes)) {
    foreach ($processes as $key => $process) {
        if (false === $process->isRunning()) {
            $output->write($process->getOutput());
            $output->writeln('Finished process: '. $key);
            $end = new \DateTime();
            $took = $end->diff($start);
            $output->writeln('took: '.$took->format('%I:%S'));
            unset($processes[$key]);
        }
    }
}
```

Tworzymy z tego nową komendę, która zrównolegli nasze Batch processing.

Uwaga Tradeoff!

Komendy, które zabierają całe CPU nie wpływają korzystnie na działanie php-fpm w trakcie ich działania.

Taką komendę najlepiej odpalić na osobnej instancji.

A co gdy mam tylko jedną instancje na fpm i workery?

Kolejka na ratunek.

1) Mamy już batch processing,
2) zamiast spawnować nowe procesy tworzymy message zawierające batche do wykonania,
3) które consumer kolejki przetworzy w wolnym czasie.
4) Zabraliśmy tylko jeden wątek z procesora.
5) FPM jest szczęśliwy

A co jeśli komenda jest consumerem kolejki?

Zastosuj wszystkie powyższe sugestie dla komend.

Za wyjątkiem wielowątkowości.

Zadbaj by “zdemonizować” proces, bo sam lubi sobie czasem bez ostrzeżenia umrzeć.

Recepta: Supervisord

```
[program:rabbitmq_consumer]
process_name=%(program_name)s_%(process_num)02d
command=php /app/src/console.php worker:queue-message-consumer


numprocs=1
autostart=true
autorestart=true
startsecs = 0
user=root


stdout_logfile=/dev/fd/1
stdout_logfile_maxbytes=0
stderr_logfile=/dev/fd/1
stderr_logfile_maxbytes=0
```

Zadbaj by wiadomość była idempotentna.

Gdy ją wyślesz wielokrotnie, to zostanie przetworzona maksymalnie jeden raz.

Tak jak mnożenie razy

jeden.

A co z integracjami?

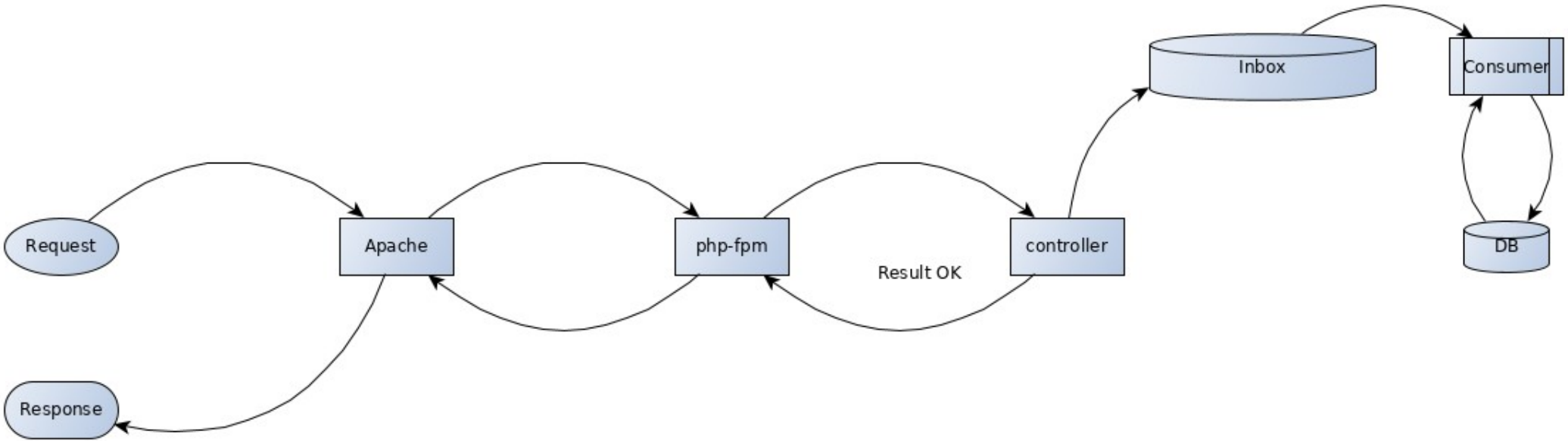
Racja. Wpuszczenie integracji zewnętrznej do naszego systemu, bez ograniczeń może być kosztowne.

Gdy przyjmujemy dane. Możemy zamiast integracji otrzymać atak DDoS.

Do przyjmowania danych zastosuj Inbox pattern.

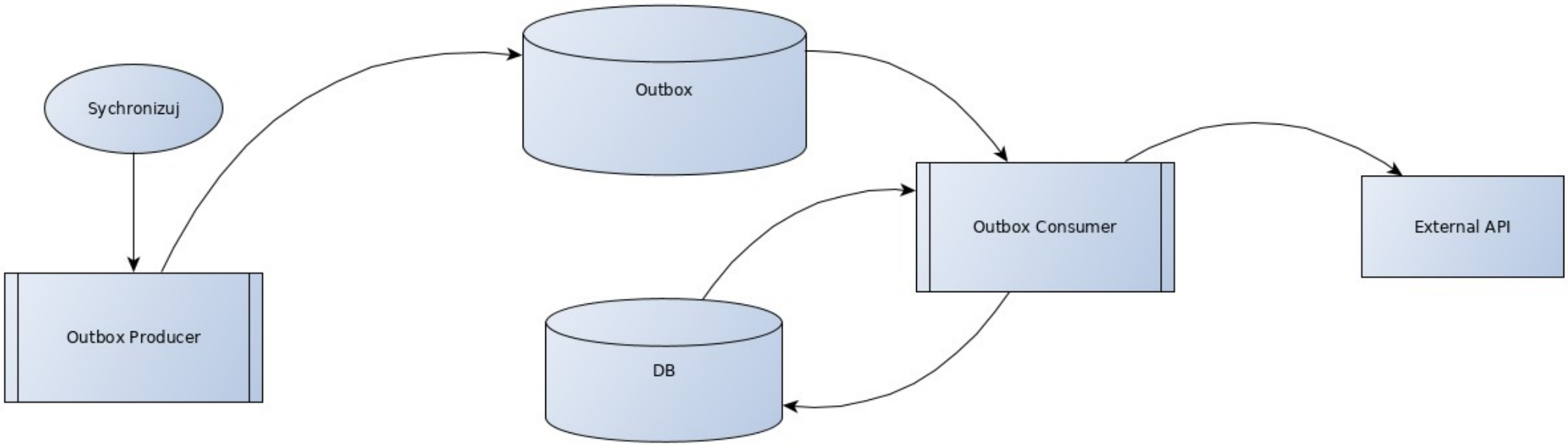
Mówiłem o tym na phpers poznań #12

Endpoint do przyjmowania danych możemy tak zamodelować.


Request
Apache
php-fpm
Result OK
controller
Inbox
Consumer
DB
Response

A co z integracjami gdzie ja wysyłam dane?

Dokładnie odwrotnie. Zastosować Outbox pattern.


Sychronizuj
Outbox
Outbox Producer
DB
Outbox Consumer
External API

Pozbyliśmy się bottlenecków!

For now.

A jak robić żeby nie wprowadzić kolejnych?

Żeby widzieć zmiany w środowisku produkcyjnym polecam:

Monitoring response time.

Ustalenia pułapu wydajności z klientem, który chcemy utrzymać.

Pytania?

Dzięki!

Złap mnie

- X: @mgz
- Github: @emgiezet

Post Scriptum

Aktualnie szukam nowego projektu.

Call me ;)