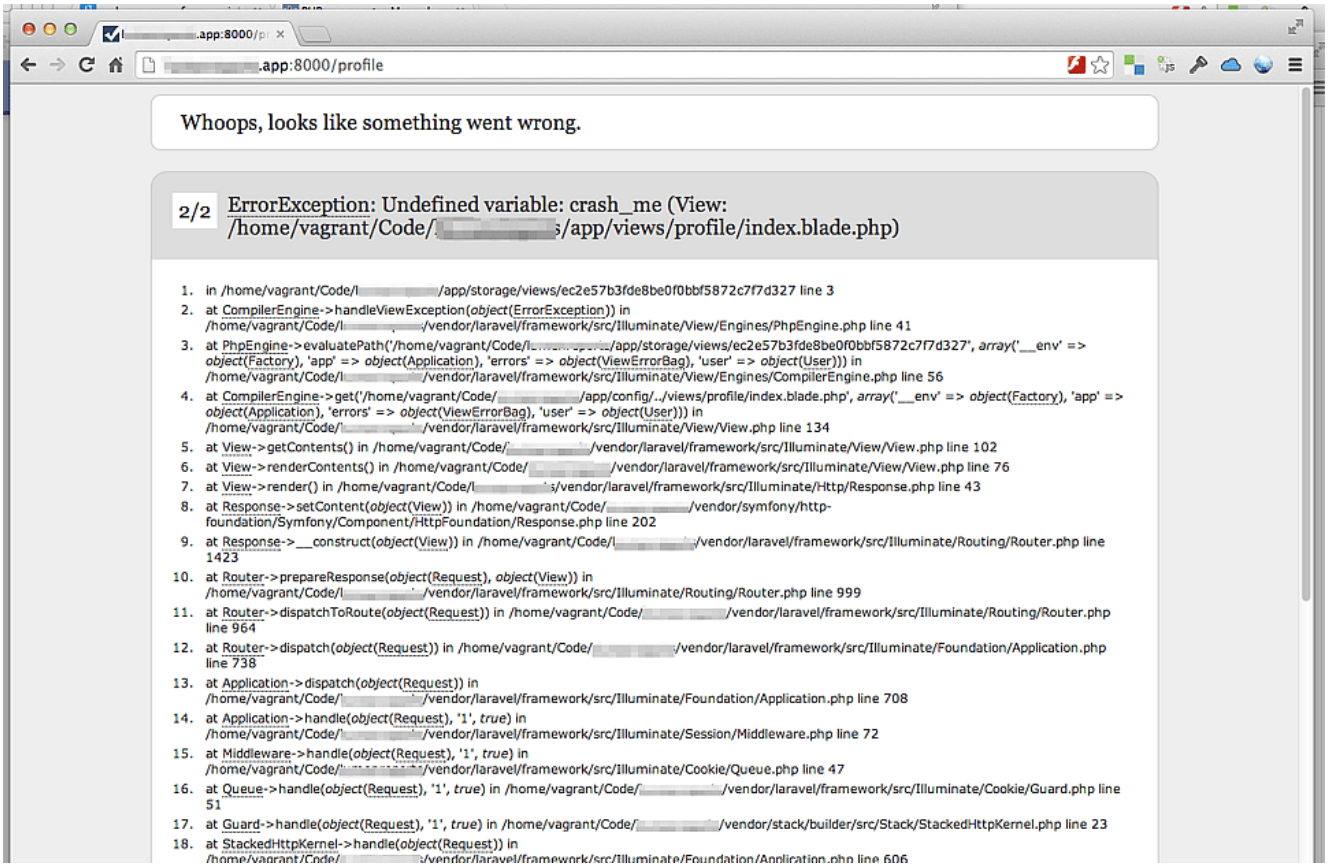
Как оптимизировать веб-приложение PHP Laravel для высокой производительности?

22.02.2024

Laravel — это много чего. Но быстрота — не одна из них. Давайте научимся некоторым хитростям, чтобы сделать его быстрее! В наши дни ни один PHP-разработчик не сталкивается с Laravel. Это либо разработчик младшего или среднего уровня, которому нравится быстрое развитие, предлагаемое Laravel, либо разработчик старшего уровня, который вынужден изучать Laravel из-за давления рынка. В любом случае, нельзя отрицать, что Laravel оживил экосистему PHP (я, например, давно бы ушел из мира PHP, если бы не было Laravel). Однако, поскольку Laravel изгибается, чтобы облегчить вам жизнь, это означает, что под ним скрывается тонна и тонна работы, чтобы обеспечить вам комфортную жизнь как разработчику. Все “волшебные” функции Laravel, которые, как кажется, просто работают, имеют множество слоев кода, которые необходимо создавать каждый раз, когда функция запускается. Даже простой Exception показывает, насколько глубока кроличья нора (обратите внимание, с чего начинается ошибка, вплоть до основного ядра):



Для того, что кажется ошибкой компиляции в одном из представлений, необходимо отследить 18 вызовов функций. Я лично столкнулся с 40, и их может быть больше, если вы используете другие библиотеки и плагины. Дело в том, что по умолчанию эти слои кода делают Laravel медленным.

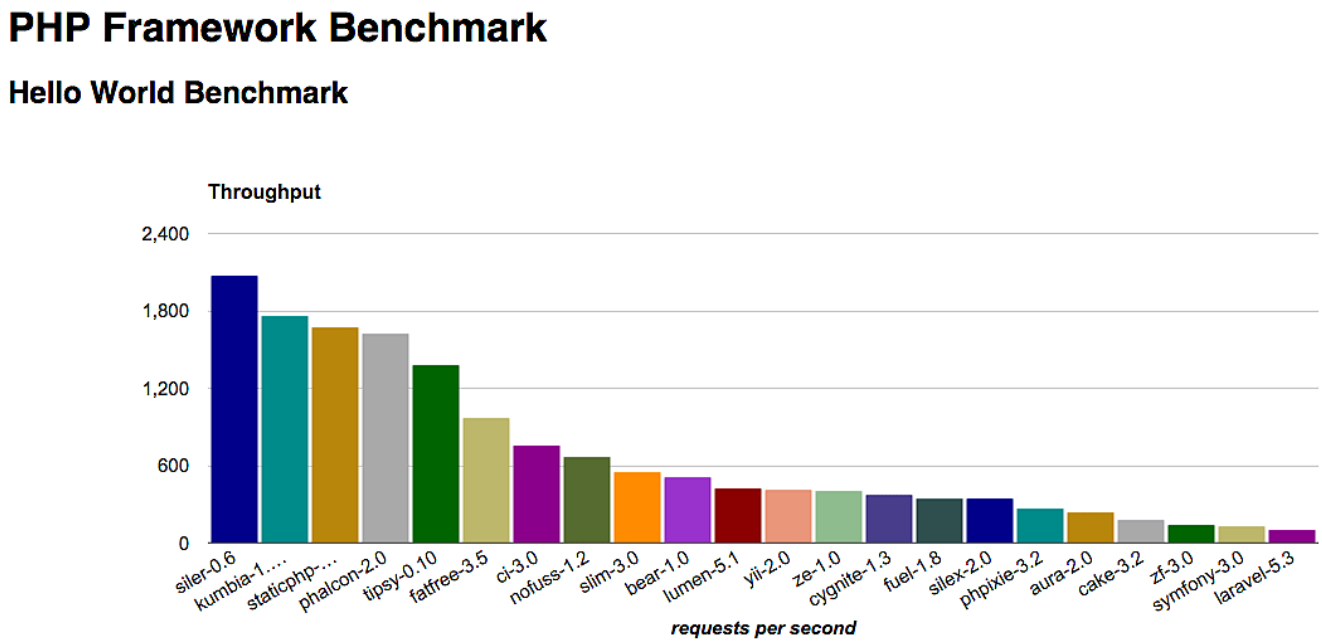
Насколько медленным является Laravel? Честно говоря, ответить на этот вопрос просто невозможно по нескольким причинам.

Во-первых, не существует общепринятого, объективного, разумного стандарта для измерения скорости работы веб-приложений. Быстрее или медленнее по сравнению с чем? В каких условиях?

Во-вторых, веб-приложение зависит от стольких вещей (база данных, файловая система, сеть, кэш и т. д.), что говорить о скорости просто глупо. Очень быстрое веб-приложение с очень медленной базой данных — это очень медленное веб-приложение . Но именно эта неопределенность и является причиной популярности эталонов. Даже если они ничего не значат, они дают *некую* систему координат и помогают нам не сойти с ума.

Поэтому, приготовив несколько щепоток соли, давайте получим неверное, приблизительное представление о скорости работы PHP-фреймворков.

Судя по этому довольно уважаемому источнику на GitHub, вот как выглядят эти PHP-фреймворки в сравнении:



Вы можете даже не заметить Laravel (даже если сильно прищуритесь), если только не бросите свой взгляд в самый конец хвоста. Да, дорогие друзья, Laravel на последнем месте! Конечно, большинство из этих “фреймворков” не очень практичны и даже полезны, но это говорит о том, насколько Laravel медлительнее других, более популярных. Обычно эта “медлительность” не проявляется в приложениях, потому что наши повседневные веб-приложения редко достигают высоких показателей. Но как только это происходит (скажем, до 200-500 параллелей), серверы начинают задыхаться и умирать. Это тот случай, когда даже установка дополнительного оборудования не помогает решить проблему, а счета за инфраструктуру растут так быстро, что ваши высокие идеалы облачных вычислений рушатся.

Но не унывайте! Эта статья не о том, что нельзя сделать, а о том, что можно сделать .

Хорошая новость заключается в том, что вы можете сделать многое, чтобы ваше приложение на Laravel работало быстрее. В несколько раз быстрее. Да, без шуток. Вы можете заставить одну и ту же кодовую базу работать быстрее и сэкономить несколько сотен долларов на инфраструктуре/хостинге каждый месяц. Как? Давайте перейдем к делу.

Четыре типа оптимизаций

На мой взгляд, оптимизация может проводиться на четырех разных уровнях (если речь идет о PHP-приложениях):

- **На уровне языка:** Это означает, что вы используете более быструю версию языка и избегаете специфических особенностей/стилей кодирования в языке, которые делают ваш код медленным.
- **Уровень фреймворка:** Это те вещи, которые мы будем рассматривать в этой статье.

- **Инфраструктурный уровень:** Настройка менеджера процессов PHP, веб-сервера, базы данных и т. д.
- **Аппаратный уровень:** Переход на более качественное, быстрое и мощное оборудование хостинг-провайдера.

Все эти типы оптимизаций имеют свое место (например, оптимизация PHP-fpm является довольно важной и мощной). Но в этой статье речь пойдет об оптимизациях исключительно второго типа: тех, что связаны с фреймворком. Кстати, нумерация не имеет никакого обоснования и не является общепринятым стандартом. Я просто придумал их. Пожалуйста, никогда не цитируйте меня и не говорите: “Нам нужна оптимизация третьего типа на нашем сервере”, иначе руководитель вашей команды убьет вас, найдет меня, а потом убьет и меня. И вот, наконец, мы добрались до земли обетованной.

Будьте в курсе n+1 запросов к базе данных

Проблема с n+1 запросами часто встречается при использовании ORM. В Laravel есть мощный ORM под названием Eloquent, который настолько красив, настолько удобен, что мы часто забываем посмотреть, что там происходит. Рассмотрим очень распространенный сценарий: отображение списка всех заказов, сделанных заданным списком клиентов. Это довольно часто встречается в системах электронной коммерции и вообще в любых интерфейсах отчетности, где нам нужно отобразить все сущности, связанные с некоторыми сущностями. В Laravel мы можем представить себе функцию контроллера, которая выполняет эту работу следующим образом:

```
class OrdersController extends Controller { // ...

    public function getAllByCustomers(Request $request, array $ids) { $customers = Customer::findMany($ids); $orders = collect(); // новая коллекция foreach ($customers as
```

```
$customer) { $orders = $orders->merge($customer->orders); }
return view('admin.reports.orders', ['orders' => $orders]); }
}
```

Мило! А главное, элегантно, красиво. □□ К сожалению, это *катастрофический* способ написания кода в Laravel. Вот почему. Когда мы просим ORM найти заданных клиентов, формируется SQL-запрос, подобный этому:

```
SELECT * FROM customers WHERE id IN (22, 45, 34, . . .);
```

Что в точности соответствует ожиданиям. В результате все возвращенные строки сохраняются в коллекции $customers внутри функции контроллера. Теперь мы перебираем всех клиентов по очереди и получаем их заказы. Для этого выполняется следующий запрос…

```
SELECT * FROM orders WHERE customer_id = 22;
```

. … столько раз, сколько есть клиентов.

Другими словами, если нам нужно получить данные о заказе для 1000 клиентов, общее количество выполняемых запросов к базе данных составит 1 (для получения данных обо всех клиентах) + 1000 (для получения данных о заказе для каждого клиента) = 1001. Отсюда и происходит название n+1. Можем ли мы добиться большего? Конечно! Используя так называемую “нетерпеливую загрузку”, мы можем заставить ORM выполнить JOIN и вернуть все необходимые данные в одном запросе! Например:

```
$orders = Customer::findMany($ids)->with('orders')->get();
```

Результирующая структура данных, конечно, вложенная, но данные о порядке можно легко извлечь. Результирующий единый запрос в этом случае выглядит примерно так:

```
SELECT * FROM customers INNER JOIN orders ON customers.id =
orders.customer_id WHERE customers.id IN (22, 45, . . .);
```

Один запрос, конечно, лучше, чем тысяча лишних запросов. Представьте себе, что было бы, если бы нужно было обработать

10 000 клиентов! Или, не дай бог, если бы мы захотели отобразить товары, содержащиеся в каждом заказе! Помните, что техника называется eager loading, и это почти всегда хорошая идея.

Кэшируйте конфигурацию!

Одна из причин гибкости Laravel – множество конфигурационных файлов, входящих в состав фреймворка. Хотите изменить, как/где хранятся изображения? Ну, просто измените файл config/filesystems.php (по крайней мере, на момент написания статьи). Хотите работать с несколькими драйверами очередей? Не стесняйтесь описать их в config/queue.php. Я только что подсчитал и обнаружил, что существует 13 конфигурационных файлов для различных аспектов фреймворка, что гарантирует, что вы не будете разочарованы независимо от того, что вы хотите изменить.

Учитывая природу PHP, каждый раз, когда поступает новый веб-запрос, Laravel просыпается, загружает все и разбирает *все* эти конфигурационные файлы, чтобы понять, как сделать все по-

другому на этот раз. Вот только это глупо, если за последние несколько дней ничего не изменилось! Перестраивать конфигурацию при каждом запросе — это лишние траты, которых можно (на самом деле, нужно) избежать, и выход из этой ситуации — простая команда, которую предлагает Laravel:

```
php artisan config:cache
```

Это позволяет объединить все доступные файлы конфигурации в один и кэшировать его для быстрого получения. В следующий раз, когда поступит веб-запрос, Laravel просто прочитает этот единственный файл и начнет работу. Тем не менее, кэширование конфигурации — это очень тонкая операция, которая может взорваться у вас на глазах. Самая большая загвоздка заключается в том, что после выполнения этой команды вызовы функции env() отовсюду, кроме файлов конфигурации, будут возвращать null! В этом есть смысл, если подумать. Если вы используете кэширование конфигурации, вы говорите фреймворку: “Знаете что, я думаю, что все хорошо настроил, и я на 100 % уверен, что не хочу, чтобы это менялось”. Другими словами, вы ожидаете, что окружение останется статичным, для чего и нужны файлы .env. С учетом сказанного, вот несколько железных, священных, нерушимых правил кэширования конфигураций:

1. Делайте это только на рабочей системе.
2. Делайте это только в том случае, если вы действительно уверены, что хотите заморозить конфигурацию.
3. Если что-то пойдет не так, отмените настройку с помощью php artisan cache:clear
4. Молитесь, чтобы ущерб, нанесенный бизнесу, не был значительным!

Сократите количество служб в

автозагрузке

Чтобы быть полезным, Laravel загружает тонну сервисов, когда просыпается. Они доступны в файле config/app.php как часть ключа массива 'providers'. Давайте посмотрим, что есть в моем случае:

```
/* |---------------------------------------------------------- | Автозагрузка поставщиков услуг |---------- ---------------------------------------------------------- - | | | Перечисленные здесь поставщики услуг будут автоматически загружаться при | запросе к вашему приложению. Не стесняйтесь добавлять в этот массив свои собственные сервисы,| чтобы расширить функциональность ваших приложений.| */ 'providers' => [ /* * Laravel Framework Service Providers...
            */ IlluminateAuthAuthServiceProvider::class, IlluminateBroadcastingBroadcastServiceProvider::class, IlluminateBusBusServiceProvider::class, IlluminateCacheCacheServiceProvider::class, IlluminateFoundationProvidersConsoleSupportServiceProvider::class,     IlluminateCookieCookieServiceProvider:::класс, IlluminateDatabaseDatabaseServiceProvider::класс, IlluminateEncryptionEncryptionServiceProvider::класс, IlluminateFilesystemFilesystemServiceProvider::класс, IlluminateFoundationProvidersFoundationServiceProvider::класс, IlluminateHashingHashServiceProvider::класс, IlluminateMailMailServiceProvider::class, IlluminateNotificationsNotificationServiceProvider::class, IlluminatePaginationPaginationServiceProvider::class, IlluminatePipelinePipelineServiceProvider::class, IlluminateQueueQueueServiceProvider::class, IlluminateRedisRedisServiceProvider:::class, IlluminateAuthPasswordsPasswordResetServiceProvider::class, IlluminateSessionSessionServiceProvider::class, IlluminateTranslationTranslationServiceProvider::class, IlluminateValidationValidationServiceProvider::class, IlluminateViewViewServiceProvider::class, /* * Пакет Service Providers.... */ /* * Application Service Providers... */ AppProvidersAppServiceProvider::class, AppProvidersAuthServiceProvider::class,          //
```

```
AppProvidersBroadcastServiceProvider::class,
AppProvidersEventServiceProvider::class,
AppProvidersRouteServiceProvider::class, ],
```

Я снова посчитал, и в списке оказалось 27 услуг! Возможно, вам понадобятся все из них, но это маловероятно. Например, в данный момент я создаю REST API, а значит, мне не нужны Session Service Provider, View Service Provider и т. д. А поскольку я делаю некоторые вещи по-своему, а не следую установкам фреймворка по умолчанию, я также могу отключить Auth Service Provider, Pagination Service Provider, Translation Service Provider и так далее. В общем, почти половина из них не нужна для моего случая использования. Внимательно изучите свое приложение. Нужны ли ему все эти поставщики услуг? Но ради Бога, пожалуйста, не комментируйте эти сервисы вслепую и не запускайте в производство! Проведите все тесты, проверьте все вручную на dev- и staging-машинах и будьте очень параноидальны, прежде чем нажимать на курок.

Будьте мудры при выборе стеков промежуточного ПО

Если вам нужна пользовательская обработка входящего веб-запроса, создайте новое промежуточное ПО. Заманчиво открыть app/Http/Kernel.php и поместить промежуточное ПО в стек web или api; так оно станет доступно во всем приложении и если не будет делать ничего навязчивого (например, протоколировать или уведомлять). Однако по мере роста приложения эта коллекция глобального промежуточного программного обеспечения может стать тихой обузой для приложения, если все (или большинство) из них будут присутствовать в каждом запросе, даже если для этого нет никаких бизнес-причин. Другими словами, будьте осторожны с тем, где вы добавляете/применяете новое промежуточное ПО. Может быть удобнее добавить что-то глобально, но в долгосрочной перспективе это приведет к большим потерям

производительности. Я знаю, как больно вам придется, если вы будете выборочно применять промежуточное ПО каждый раз, когда происходит новое изменение, но я с готовностью приму эту боль и рекомендую!

Избегайте ORM (в некоторых случаях)

Хотя Eloquent делает многие аспекты взаимодействия с БД приятными, за это приходится расплачиваться скоростью. Будучи маппером, ORM должен не только получать записи из базы данных, но и инстанцировать объекты модели и гидратировать (заполнять) их данными из колонок. Так, если вы сделаете простой запрос `$users = User::all()` и найдется, скажем, 10 000 пользователей, фреймворк получит 10 000 строк из базы данных и внутренне создаст 10 000 новых `User()` и заполнит их свойства соответствующими данными. Это огромный объем работы, выполняемый за кулисами, и если база данных является узким местом в вашем приложении, обход ORM иногда является хорошей идеей. Это особенно актуально для сложных SQL-запросов, где вам пришлось бы перепрыгивать через множество обручей и писать закрытия за закрытиями, но в итоге все равно получился бы эффективный запрос. В таких случаях предпочтительнее сделать `DB::raw()` и написать запрос вручную. Судя по этому исследованию производительности, даже при простой вставке Eloquent работает гораздо медленнее, чем при увеличении количества записей:

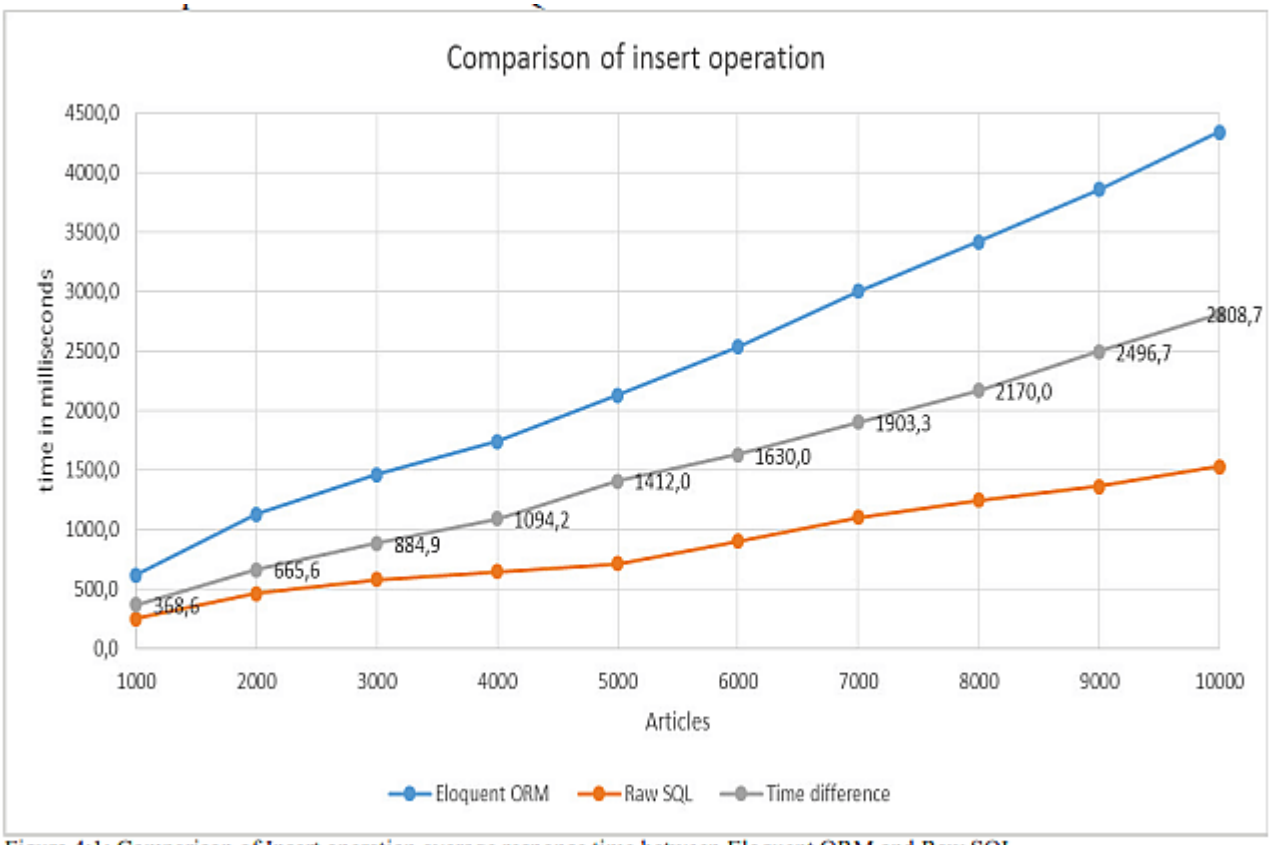


Figure 4:1: Comparison of Insert operation average response time between Eloquent ORM and Raw SQL.

Используйте кэширование как можно чаще

Один из самых секретных секретов оптимизации веб-приложений – это кэширование. Для непосвященных кэширование означает предварительное вычисление и хранение дорогостоящих результатов (дорогостоящих с точки зрения использования процессора и памяти) и простое их возвращение при повторении одного и того же запроса. Например, в магазине электронной коммерции может оказаться, что из 2 миллионов товаров людей чаще всего интересуют те, что есть в наличии, в определенном ценовом диапазоне и для определенной возрастной группы. Запрашивать базу данных для получения этой информации расточительно – поскольку запросы меняются нечасто, лучше хранить эти результаты в месте, к которому можно быстро получить доступ. Laravel имеет встроенную поддержку нескольких типов кэширования. Помимо использования драйвера кэширования и создания системы кэширования с нуля, вы можете использовать

некоторые пакеты Laravel, которые облегчают кэширование моделей, кэширование запросов и т. д. Но учтите, что за пределами определенного упрощенного сценария использования готовые пакеты кэширования могут создать больше проблем, чем решить.

Предпочитайте кэширование в памяти

Когда вы кэшируете что-то в Laravel, у вас есть несколько вариантов того, где хранить результирующие вычисления, которые нужно кэшировать. Эти варианты также известны как драйверы кэша. Итак, хотя использование файловой системы для хранения результатов кэширования возможно и вполне разумно, это не совсем то, чем должно быть кэширование. В идеале вы хотите использовать кэш in-memory (живущий полностью в оперативной памяти), например Redis, Memcached, MongoDB и т. д., чтобы при повышенных нагрузках кэширование служило жизненно важным целям, а не становилось узким местом. Вы можете подумать, что использование SSD-диска — это почти то же самое, что и использование RAM-накопителя, но это даже не так. Даже неофициальные бенчмарки показывают, что RAM превосходит SSD в 10-20 раз, когда речь идет о скорости. Моя любимая система, когда речь идет о кэшировании, — Redis. Она до смешного быстра (100 000 операций чтения в секунду — обычное дело), а для очень больших систем кэширования ее можно легко превратить в кластер.

Кэшируйте маршруты

Как и конфигурация приложения, маршруты не сильно меняются со временем и являются идеальным кандидатом для кэширования. Это особенно актуально, если вы, как и я, не переносите большие файлы и в итоге разбиваете web.php и api.php на несколько файлов. Одна команда Laravel собирает все доступные маршруты и хранит их под рукой для будущего доступа:

```
php artisan route:cache
```

И если вам нужно добавить или изменить маршрут, просто сделайте это:

```
php artisan route:clear
```

Оптимизация изображений и CDN

Изображения — это сердце и душа большинства веб-приложений. По совпадению, они также являются крупнейшими потребителями полосы пропускания и одной из главных причин медленной работы приложений/веб-сайтов. Если вы просто храните загруженные изображения на сервере и отправляете их обратно в HTTP-ответах, вы упускаете огромную возможность оптимизации. Прежде всего, я рекомендую не хранить изображения локально — это чревато потерей данных, а в зависимости от географического региона, в котором находится ваш клиент, передача данных может быть очень медленной. Вместо этого воспользуйтесь таким решением, как Cloudinary, которое автоматически изменяет размер и оптимизирует изображения на лету. Если это невозможно, используйте что-то вроде Cloudflare для кэширования и обслуживания изображений, пока они хранятся на вашем сервере. А если даже это невозможно, то небольшая настройка программного обеспечения вашего веб-сервера, чтобы сжимать ресурсы и направлять браузер посетителя на кэширование, имеет большое значение. Вот как выглядит фрагмент конфигурации Nginx:

```
server { # file truncated # gzip compression settings gzip on; gzip_comp_level 5; gzip_min_length 256; gzip_proxied any; gzip_vary on; # browser cache control location ~* .(ico|css|js|gif|jpeg|jpg|png|woff|ttf|otf|svg|woff2|eot)$ { expires 1d; access_log off; add_header Pragma public; add_header Cache-Control "public, max-age=86400"; } }
```

Я знаю, что оптимизация изображений не имеет никакого отношения к Laravel, но это настолько простой и мощный прием (и им так часто пренебрегают), что я не смог удержаться.

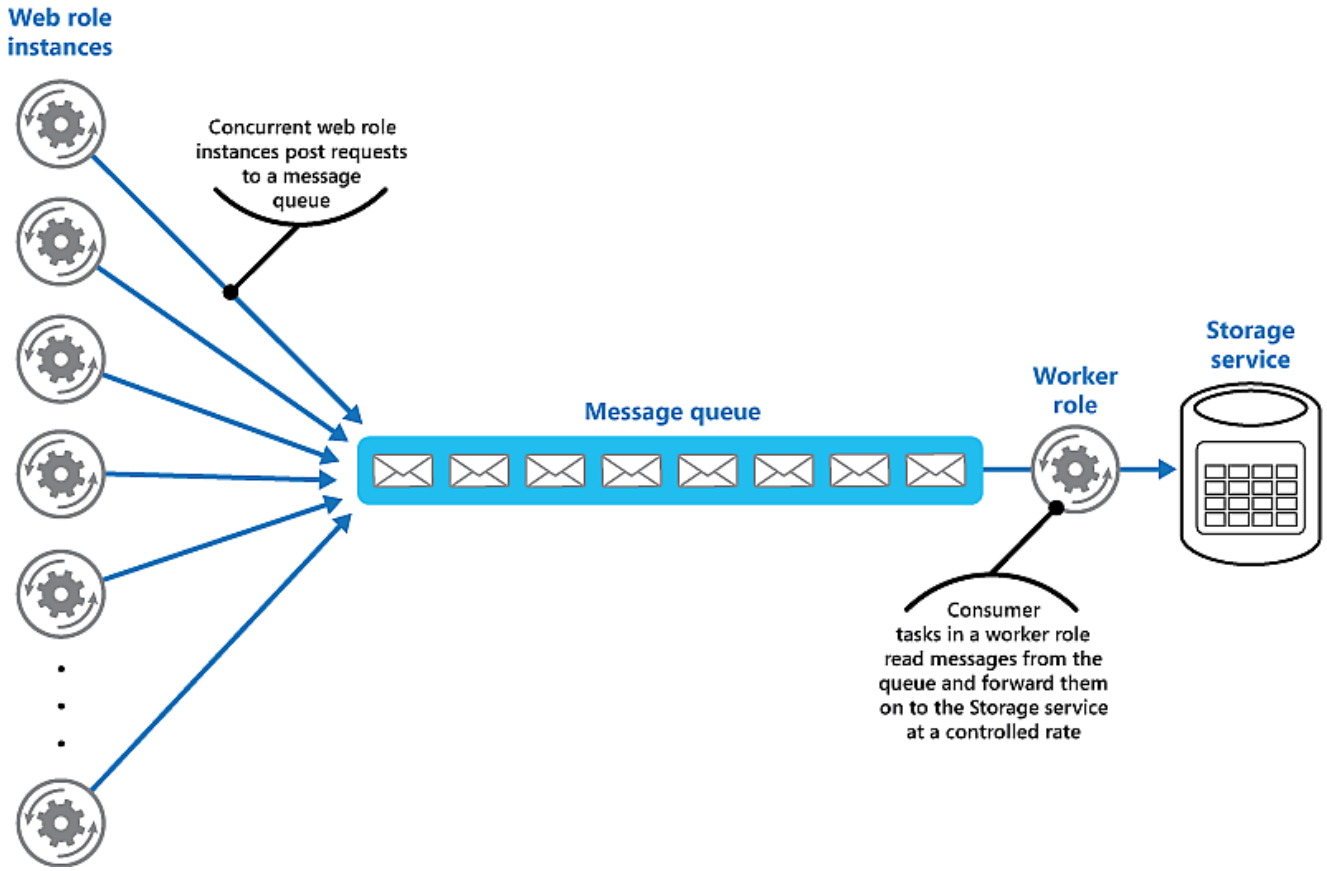
Оптимизация автозагрузки

Автозагрузка — это изящная, не так давно появившаяся функция в PHP, которая, возможно, спасла язык от гибели. Тем не менее, процесс поиска и загрузки соответствующего класса путем расшифровки заданной строки пространства имен занимает много времени, и его можно избежать в производственных развертываниях, где желательна высокая производительность. И снова Laravel предлагает однокомандное решение этой проблемы:

```
composer install --optimize-autoloader --no-dev
```

Подружитесь с очередями

Очереди — это способ обработки действий, когда их много, и каждое из них занимает несколько миллисекунд. Хорошим примером является отправка электронных писем — широко распространенный случай использования веб-приложений заключается в отправке нескольких писем с уведомлениями, когда пользователь совершает определенные действия. Например, при запуске нового продукта вы хотите, чтобы руководство компании (около 6-7 адресов электронной почты) получало уведомление каждый раз, когда кто-то делает заказ на сумму выше определенной. Если предположить, что ваш почтовый шлюз может ответить на SMTP-запрос за 500 мс, то речь идет о 3-4 секундах ожидания для пользователя, прежде чем появится подтверждение заказа. Очень плохой UX, я уверен, вы согласитесь. Выход — сохранять задания по мере их поступления, сообщать пользователю, что все прошло успешно, и обрабатывать их (через несколько секунд). Если произошла ошибка, поставленные в очередь задания можно повторить несколько раз, прежде чем они будут признаны неудачными.



Кредиты: Microsoft.com

Хотя система очередей немного усложняет настройку (и добавляет накладные расходы на мониторинг), она незаменима в современных веб-приложениях.

Оптимизация активов (Laravel Mix)

Для любых фронтенд-активов в вашем Laravel-приложении убедитесь, что есть конвейер, который компилирует и минифицирует все файлы активов. Тем, кому удобно работать с такими бандлерами, как Webpack, Gulp, Parcel и т. д., можно не беспокоиться, но если вы еще не делаете этого, то Laravel Mix — это надежная рекомендация. Mix — это легкая (и восхитительная, по правде говоря!) обертка вокруг Webpack, которая позаботится обо всех ваших CSS, SASS, JS и т.д. файлах для производства. Типичный файл .mix.js может быть таким же маленьким, как этот, и при этом творить чудеса:

```
const mix = require('laravel-mix');
mix.js('resources/js/app.js', 'public/js')
.sass('resources/sass/app.scss', 'public/css');
```

Это автоматически позаботится об импорте, минификации, оптимизации и прочей ерунде, когда вы будете готовы к производству и запустите npm run production. Mix позаботится не только о традиционных файлах JS и CSS, но и о компонентах Vue и React, которые могут быть в рабочем процессе вашего приложения.

Заключение

Оптимизация производительности — это больше искусство, чем наука: знать, как и сколько делать, важнее, чем что делать. Тем не менее, нет предела тому, сколько и чего можно оптимизировать в приложении Laravel. Но что бы вы ни делали, я хотел бы дать вам напутственный совет: оптимизация должна проводиться при наличии веских причин, а не потому, что это хорошо звучит, или потому, что у вас паранойя по поводу производительности приложения для 100 000+ пользователей, в то время как на самом деле их всего 10. Если вы не уверены, нужно ли вам оптимизировать свое приложение, не стоит разворошить пресловутое осиное гнездо. Работающее приложение, которое кажется скучным, но делает именно то, что должно, в десять раз желаннее, чем приложение, которое оптимизировано до уровня гибридной супермашины-мутанта, но время от времени падает.