Webanwendungen beschleunigen mit CQRS

Webanwendungen beschleunigen mit CQRS

[expand title="mehr lesen…"]

Webanwendungen beschleunigen mit CQRS

Getrennt siegen

Arne Blankerts, Sebastian Heuer

Typische PHP-Anwendungen verbringen einen Großteil ihrer Zeit damit, Daten aus einer Datenbank abzurufen, zu verarbeiten und als HTML auszugeben. Command Query Responsibility Segregation (CQRS) kann die Zahl der Datenbankabrufe drastisch reduzieren und so die Anwendung erheblich beschleunigen.

iX-TRACT

CQRS (Command Query Responsibility Segregation) trennt lesende und schreibende Zugriffe auf den Datenbestand strikt voneinander. Dadurch lassen sich datenbankbasierte Webanwendungen deutlich beschleunigen.

Der für Schreibzugriffe zuständige Prozess erzeugt bei Änderungen eine neue HTML-Repräsentation des aktuellen Zustands der geänderten Daten.

Bei Lesezugriffen muss die Webanwendung lediglich die vorproduzierten HTML-Schnipsel ausliefern, statt die Daten aus der Datenbank abzufragen und selbst für die Darstellung im Browser aufzubereiten.

Da das zeitaufwendige Generieren der HTML-Repräsentation nicht mehr bei jedem Zugriff stattfindet, sondern nur noch bei Änderungen der Daten, profitieren vor allem Anwendungen mit deutlich mehr Lese- als Schreibzugriffen.

PHP gehört nach wie vor zu den gängigsten Sprachen bei der Webentwicklung. Stand früher besonders die einsteigerfreundliche Lernkurve im Vordergrund, werden inzwischen auch umfangreiche Webanwendungen mit der Skriptsprache entwickelt. Die aktuelle Version 7 brachte PHP eine deutlich optimierte Laufzeitumgebung – Grund genug, sich näher anzusehen, wie sich heutzutage hochperformante Webseiten mit PHP umsetzen lassen.

Anders als Java oder Node.js folgt PHP dem „shared nothing“-Prinzip: Die einzelnen Requests an den Server teilen sich erst einmal keinerlei Daten. Auch wenn moderne PHP-Frameworks komplexe Implementierungen möglich machen, läuft es technisch doch erschreckend häufig auf die folgenden Schritte hinaus: den für die URL zuständigen Controller ermitteln, mit SQL aus der Datenbank die für das Model notwendigen Daten abfragen, den View mit HTML erzeugen und das Ergebnis ausliefern.

Was hier so einfach klingt, ist in der Praxis natürlich deutlich aufwendiger und wird bei komplexen Datenbankabfragen und steigender Systemlast schnell zum Problem. Auf den ersten Blick wäre dann ein dauerhaft laufender Application Server wie bei Java oder Node.js von Vorteil, der bereits geholte Daten im Speicher hält. Ob und wann geänderte Daten in der Persistenz aktualisiert werden, wäre dann bloß ein Implementierungsdetail.

Problemfall zentrale Datenbank

Der so erzielte Performancegewinn wird jedoch im wahrsten Sinne des Wortes teuer erkauft: Nimmt der Besucherandrang zu, muss man den Server immer weiter aufrüsten. Irgendwann geht es nicht mehr sinnvoll weiter und es bleibt nur eine horizontale Skalierung. War bislang ein einziger Serverprozess für die Datenhaltung verantwortlich, müssen sich jetzt mehrere Prozesse und Maschinen synchronisieren, was entweder eine ausgefeilte Interprozess-Kommunikation oder doch wieder eine zentrale Persistenz etwa in Form einer permanent aktualisierten Datenbank erfordert.

Bei einer derartigen Architektur, egal ob mit PHP oder einer anderen Plattform genutzt, hat selbst eine deutliche Beschleunigung der Laufzeitumgebung nur eine marginale Auswirkung auf die Gesamtperformance: Wenn die Anwendung einen signifikanten Anteil der Zeit mit Warten auf Antworten der Datenbank verbringt, beschleunigt das, nüchtern betrachtet, lediglich das Warten.

Vielleicht liegt die Lösung des Problems also weniger im Beschleunigen der Ausführung als im Vermeiden derselben. Beachtet man zudem, dass die meisten Webseiten deutlich häufiger gelesen als neu geschrieben werden, bietet es sich an, hier zuerst anzusetzen.

Mehr Tempo durch Caching

Wenig überraschend ist die gängige Reaktion auf Performanceprobleme bei Lesezugriffen: die Einführung von Caches. Sie sollen das erneute Generieren einer Antwort unnötig machen, sofern die Anfrage innerhalb eines bestimmten Zeitfensters schon einmal beantwortet wurde.

Naheliegend und technisch am einfachsten umzusetzen ist das Vorschalten eines Reverse Proxy wie Varnish, der die HTML-Ausgabe der Applikation zumeist im Arbeitsspeicher

zwischenspeichert und sie bei ähnlichen Requests direkt an den Client zurücksendet, ohne dass die Applikation Arbeit damit hat. Das sorgt zwar für eine sehr schnelle Beantwortung von Requests, für die bereits eine passende Antwort im Cache liegt, bringt aber neue Probleme mit sich.

Daten in einem Cache sind immer potenziell veraltet, da sie der Reverse Proxy lediglich anhand ihres Alters aus dem Cache löschen kann – die Anwendung soll ja gerade nicht in die Beantwortung eingebunden werden. Zudem kommt es schnell zu Inkonsistenzen, wenn verschiedene Ansichten der gleichen Daten zu unterschiedlichen Zeiten abgerufen wurden und so in unterschiedlichen Ständen im Cache zwischengespeichert sind.

Auch die Auslieferung personalisierter Inhalte unter der gleichen URL ist ein Problem, da die HTML-Ausgabe dann eben gerade nicht identisch ist. Um dennoch vom Caching zu profitieren, müsste die Antwort in personalisierte und nicht personalisierte Fragmente zerlegt werden. Die personalisierten Bereiche müssten dabei für jeden eingehenden Request von der Applikation neu geliefert werden.

Varnish bietet mit Edge Side Includes (ESI) eine in diese Richtung gehende Option. Allerdings bringt das einen zusätzlichen Frontend-Layer außerhalb der Applikation mit sich, was zu neuen Problemen führt. Und gerade die Fragmente mit dynamischen Inhalten, deren Erzeugung teuer ist, werden nicht vom Cache abgedeckt, sodass das Performanceproblem letztlich bestehen bleibt.

Nicht zuletzt erschwert die Abhängigkeit der Anwendung von einem gefüllten Cache auch noch das Deployment neuer Versionen. In der Praxis erweist es sich als sehr kompliziert, zu ermitteln, welche Daten veraltet sind; daher löscht man meist einfach alle Daten aus dem Cache. Das bedeutet allerdings, dass die Anwendung jetzt wieder alle Anfragen bearbeiten muss, was zwangsläufig zu einem Performanceengpass führt.

Caching kann nicht die Antwort sein

Caching lindert also nur Symptome, ohne jedoch die eigentliche Ursache von Performanceproblemen zu adressieren. Die vermeintlich einfache und schnelle Verbesserung der Antwortzeiten wird mit zusätzlicher Komplexität, neuen Ausfallszenarien und der Auslieferung potenziell veralteter Daten erkauft. Es lohnt sich daher, einige Schritte zurückzutreten und erneut einen Blick auf die Softwarearchitektur zu werfen.

Eines der wichtigsten Ziele jeder Website ist die schnellstmögliche Beantwortung von Anfragen. Um das zu erreichen, muss der Server mit der ohnehin schon knappen Ressource Zeit extrem sparsam umgehen. Geht es um Antwortzeiten von 100 Millisekunden und weniger, wird die Luft beim Einsatz gängiger Fullstack-Frameworks auch und gerade im PHP-Umfeld schnell dünn: Die schiere Größe der Codebasis, die zur Laufzeit viele Entscheidungen treffen muss, zusammen mit nur selten benötigter Flexibilität fordert ihren Preis.

Doch auch ein bewusst schlankes, im Zweifel selbst geschriebenes Framework verschenkt noch viel Potenzial. Betrachtet man beispielsweise die Produktseite zu einem Artikel in einem E-Shop, beginnt deren Aufbau meist damit, die Artikelstammdaten – Bilder, Texte, tagesaktuelle Preise – unter Einsatz eines ORMs wie Doctrine aus der Datenbank zu ziehen und die zugehörigen Models zu instanziieren. Die so vorbereiteten Daten gehen dann an eine Template-Engine zur Umwandlung in HTML.

Aber sind all diese Schritte überhaupt notwendig? Aus Sicht des Browsers hat das empfangene HTML mit dem Model „Produkt“ nichts mehr zu tun. Nimmt man hinzu, dass sich die Artikelstammdaten nicht bei jedem Request ändern, folgt daraus: Das Erzeugen der HTML-Repräsentation muss gar nicht während der Beantwortung der Anfrage geschehen, sondern nur dann, wenn sich die zugrunde liegenden Daten ändern.

CQRS to the Rescue

CQRS (Command Query Responsibility Segregation) beschreibt ein Architekturmuster, das lesende und schreibende Operationen strikt voneinander trennt. Konkret bedeutet das, dass ein generisches Model aufgeteilt wird in eines für die lesende und eines für die schreibende Seite. Aus einer Klasse, die eine API zum Abrufen und eine zum Verändern von Daten anbietet, werden also zwei Klassen mit spezialisierten APIs. Das Read-Model ist dann nur noch eine unveränderbare Repräsentation eines Zustandes, während das Write-Model Methoden zu dessen Veränderung bereitstellt.

Diese explizite Trennung eröffnet spannende Möglichkeiten für die Performanceoptimierung. Da lediglich Schreiboperationen den Zustand verändern können, reicht es, wenn sie die Neugenerierung der Zustandsrepräsentation triggern. Bei Abfragen muss lediglich die bei der letzten Änderung erzeugte Repräsentation ausgeliefert werden.

Interessanterweise findet sich diese konzeptuelle Trennung schon im HTTP-Standard: Der lesende *GET*-Request verändert den Zustand der Applikation nicht, während die schreibenden Request-Methoden wie *DELETE, PATCH, POST* und *PUT* eine Zustandsänderung bewirken. CQRS und das Web scheinen also bestens zusammenzupassen.

Für einen E-Shop heißt das: Die HTML-Repräsentation einer Produktseite muss nur dann neu generiert werden, wenn beispielsweise neue Artikeldaten vom ERP-System kommen. Daher kann ein von den eingehenden Requests unabhängiger Prozess diese Daten entgegennehmen und neue HTML-Repräsentationen erzeugen. Listen mehrerer Artikel beispielsweise einer Kategorie kann man aus HTML-Schnipseln für die einzelnen Artikel zusammensetzen.

Key-Value-Store statt SQL

Diese Schnipsel müssen nun so persistiert werden, dass sie sich bei der Verarbeitung eines Requests möglichst schnell laden lassen. Dafür eignet sich ein Key-Value-Store, der beliebige Daten unter einem eindeutigen Schlüssel ablegt. Statt eine SQL-Query an die Datenbank zu schicken, lädt die Anwendung die Daten über den Schlüssel, was viel schneller geht.

Der quelloffene Key-Value-Store Redis beispielsweise beantwortet bei geringem Ressourcenverbrauch mühelos mehrere Tausend Requests pro Sekunde. Dass der Key-Value-Store die generierten Daten dabei nicht wie eine relationale Datenbank in normalisierter Form, sondern im Gegenteil mehrfach in verschiedenen Varianten speichert, gehört dabei zum Konzept. Als Datenquelle für die Generierung kann natürlich weiterhin eine relationale Datenbank dienen.

Das Frontend liefert lediglich HTML-Snippets aus, die beim Anlegen und Ändern von Einträgen in der Produktdatenbank generiert werden. Die Suchmaschine ermöglicht beliebige Kombinationen der Schnipsel (Abb. 1).
Erfahrungen aus Projekten der Autoren zeigen, dass auch der Arbeitsspeicherbedarf geringer ausfällt, als man vielleicht vermuten würde. Und wenn der Key-Value-Store in einer sehr großen Anwendung einmal mehr als ein paar Gigabyte belegt, lässt sich der HTML-Code leicht komprimiert ablegen, was den Speicherbedarf drastisch verringert – um den Preis einer etwas höheren CPU-Belastung durch das Dekomprimieren beim Lesen.

Ein Beispiel aus der Praxis

Im Folgenden zeigen wir die Funktionsweise von CQRS in PHP am Beispiel der E-Commerce-Plattform hinter www.kartenmacherei.de, einem Shop für personalisierbare Print-Produkte wie Einladungskarten oder Fotokalender. Das Team der

kartenmacherei hat 2016 ein komplett neues Shop-Frontend vor eine existierende Standardsoftware gesetzt, das konsequent den Grundsätzen von CQRS folgt. Die Anwendung auf Grundlage dieser Architektur erreicht ohne vorgeschaltetes Caching Antwortzeiten von unter 35 Millisekunden und ermöglicht eine problemlose horizontale Skalierung.

Die Applikation ist in mehrere Komponenten aufgeteilt. Das in PHP geschriebene Frontend ist für die Beantwortung von Requests zuständig und soll daher möglichst wenig zu tun haben. Bei Lesezugriffen ermittelt es anhand des Request-URI die XHTML-Snippets, die zum Aufbau der Seite erforderlich sind, und lädt sie aus dem Key-Value-Store. Anschließend werden sie nach einer vorgegebenen Logik in ein XHTML-Template eingefügt und als HTML an den Client geschickt.

Eine Kategorienseite ist aus fertigen HTML-Snippets zusammengesetzt (Abb. 2).

Listing 1: Kategorienseite

```
<?php
class CategoryPage {
  private $template;
  private $searchEngine;
  private $snippetStore;

   public function __construct(SearchEngine $searchEngine, SnippetStore $snippetStore,  PageTemplate $template) {
    $this->searchEngine = $searchEngine;
    $this->snippetStore = $snippetStore;
    $this->template = $template;
  }

   public function asHtml(CategoryName $categoryName, FilterCollection $filters): HTML {
                                $productSkus = $this->searchEngine->getProductsInCategory($categoryName, $filters);
```

```
                                        $snippets        =
$this->snippetStore->getListTiles($productSkus);
     $template = $this->template->inject('category-items',
$snippets);
    return $template->asHtml();
  }
}
```

Die zu ladenden Produkt-Snippets beispielsweise für eine Kategorieseite im Shop ermittelt eine Suchmaschine. Bei kartenmacherei.de ist das Elasticsearch, eine Alternative wäre Apache Solr. Beide Open-Source-Projekte basieren auf der Lucene-Bibliothek und beantworten Anfragen auch unter hoher Last innerhalb weniger Millisekunden. Elasticsearch gibt für eine Kategorie unter Berücksichtigung von Filterkriterien („nur die Farben Rot und Pink“) eine Liste von Artikelnummern zurück, mit denen das Frontend die HTML-Snippets aus dem Key-Value-Store lädt. Listing 1 zeigt, wie wenig Code im Frontend dafür nötig ist.

Es mag so klingen, als sei der Key-Value-Store nur ein besserer Cache, doch es gibt fundamentale Unterschiede. Aus Sicht des Frontend ist der Store als primäre Datenquelle an die Stelle der zuvor genutzten relationalen Datenbank getreten. Fehlt dort ein Eintrag, verhält sich die Anwendung nicht anders, als wenn ein Eintrag in einer MySQL-Tabelle fehlt. Somit haben die Daten im Key-Value-Store auch keine TTL, da sie per Definition immer den aktuellen Datenstand repräsentieren. Dass eine separate Komponente diese Daten von Zeit zu Zeit durch neuere Versionen ersetzt, weiß das Frontend nicht. Auch das bei Caches übliche Verdrängen älterer Einträge durch neuere gibt es nicht: Geht dem Key-Value-Store der Arbeitsspeicher aus, quittiert er schlichtweg den Dienst oder muss sich mit Swappen behelfen.

Um das Frontend möglichst einfach zu halten, werden schreibende Requests aufgrund von Benutzeraktionen im Frontend an Microservices delegiert. Diese kapseln die gesamte Geschäftslogik und sind unter anderem auch für die inhaltliche

Validierung zuständig. So prüft der Warenkorb-Service beispielsweise, ob ein Artikel überhaupt verfügbar ist, bevor er im Warenkorb landet. Diese Zustandsänderungen werden in der Regel synchron ausgeführt, das Frontend wartet also, bis der Service die Aufgabe erledigt hat. Da Nutzer bei solchen Aktionen eine gewisse Verarbeitungszeit erwarten, sind hier etwas längere Antwortzeiten vertretbar. Tatsächlich gibt es Fälle, in denen eine zu schnelle Beantwortung einer abgesendeten Bestellung zu der Annahme führte, etwas habe nicht funktioniert.

Das Backend erzeugt die HTML-Snippets

Listing 2: Einfügen eines neuen Produkts

```
class ProductSnippetRenderer {
  private $snippetStore;

  public function __construct(SnippetStore $snippetStore)  {
    $this->snippetStore = $snippetStore;
  }

  public function render(Product $product) {
    $this->snippetStore->startTransaction();
$this->snippetStore->addProductListSnippet($this->renderProduc
tListSnippet($product));
$this->snippetStore->addProductDetailPageSnippet($this->render
ProductDetailPageSnippet($product));
$this->snippetStore->addCartItemSnippet($this->renderCartItemS
nippet($product));
    // (...)
    $this->snippetStore->commit();
  }
}
```

Listing 3: Schnittstelle zu Elasticsearch

```
class ElasticsearchProductSearchIndexer {
  private $elasticsearchClient;
```

```
    public function __construct(ElasticsearchClient
$elasticsearchClient) {
      $this->elasticsearchClient = $elasticsearchClient;
  }

  public function index(Product $product) {
$this->elasticsearchClient->index($this->mapProductToElasticSe
archDocument($product));
  }
}
```

Das ebenfalls in PHP geschriebene Backend befüllt den Key-Value-Store mit den vom Frontend benötigten HTML-Snippets und sonstigen Datenstrukturen (Listing 2). Dazu nimmt es neue Produktdaten entgegen (Push) oder fragt regelmäßig nach Änderungen etwa im ERP-System (Pull). Zusätzlich wird die Suchmaschine mit neuen Daten versorgt: Neue Artikeldaten gehen als JSON-Objekte an die REST-API von Elasticsearch (Listing 3).

Die Idee der vorgenerierten HTML-Snippets lässt sich auf die Pflege von Content in einem CMS übertragen: Beim Veröffentlichen von Inhalten generiert das CMS als Backend die passenden Snippets und schreibt sie in den zentralen Key-Value-Store. Das Frontend muss nur noch wissen, unter welcher URL welche Snippets auszuliefern sind. Diese Information kann das CMS als zusätzlichen Eintrag in den Key-Value-Store schreiben.

Dabei muss das CMS gar nicht über das Internet erreichbar sein, da es an der Auslieferung von Seiten nicht beteiligt ist – in Anbetracht der häufigen Sicherheitslücken in den populären Content-Management-Systemen dürfte das manchem Administrator einen ruhigeren Schlaf bescheren. Zudem lässt sich das CMS als interne Komponente leichter austauschen: Die einzige Anforderung ist die Fähigkeit, veröffentlichte Seiten oder Seitenbestandteile als (X)HTML zu rendern und inklusive der dazugehörigen URL zu exportieren.

Da Frontend und Backend nicht voneinander abhängig sind, lassen sie sich problemlos getrennt skalieren. So genügt es, bei steigender Besucherzahl einfach zusätzliche Frontend-Instanzen hochzufahren.

Jeder Onlineshop hantiert mit Session-abhängigen Daten wie der Anzahl der Artikel im Warenkorb eines Benutzers oder personalisierten Preisen. Solche Informationen lassen sich nicht ohne Weiteres als fertiges HTML ablegen: Sämtliche HTML-Schnipsel mit allen möglichen Preisen für jeden existierenden Benutzer zu rendern, ist nicht praktikabel. Daher kann es notwendig sein, einige Informationen weiterhin während der Verarbeitung eines Requests zu ermitteln. Doch auch hier können optimierte Datenstrukturen, etwa einfache Listen mit sämtlichen Preisen je User, vorbereitet und im Key-Value-Store bereitgestellt werden, um dem Frontend die Arbeit zu erleichtern.

Personalisierung im Shop

Listing 4: Personalisierte Preise

```
class PriceInjector {
  private $priceStore;
  private $template;

  public function __construct(PriceStore $priceStore, Template
$template) {
    $this->priceStore = $priceStore;
    $this->template = $template;
  }

  public function updatePrices(User $user) {
    if (!$user->hasCustomPrices()) {
      return;
    }
                                $updatedTemplate       =
$this->template->inject($this->priceStore->getUserPrices($user
));
    return $template;
```

```
  }
}
```

Personalisierte Preise kann das Frontend nach dem Laden der HTML-Snippets abfragen und mittels DOM-Operationen anstelle der Standardpreise in die Snippets einsetzen. Der Aufwand dafür ist minimal und fällt bei einer Performancemessung kaum ins Gewicht. Da das HTML-Snippet die Standardpreise enthält, ist die zusätzliche Arbeit nur erforderlich, wenn der Benutzer eingeloggt ist und abweichende Preise haben kann. Darüber hinaus kann es keine fehlerhafte Artikeldarstellung geben, die wegen einer fehlgeschlagenen Ersetzung keinen Preis enthält: Im Zweifelsfall bleibt einfach der Standardpreis stehen (Listing 4).

Auch bei sich schnell ändernden Daten wie Lagerbeständen kann es sinnvoll sein, diese dynamisch in die Snippets zu injizieren, wenn das Backend ansonsten die Snippets zu häufig neu generieren müsste. Hier ist eine Abwägung auf Basis der spezifischen Eigenheiten der Anwendung nötig.

Fazit

Die hier vorgestellte Softwarearchitektur ist nicht auf den Einsatz von PHP beschränkt, sondern mit jeder Sprache umsetzbar. Sogar die Verwendung eines Key-Value-Store wie Redis ist im Grunde ein Implementierungsdetail. PHP mit seinem „shared nothing“-Prinzip eignet sich jedoch besonders gut für die Trennung der Verarbeitung von Requests und Änderungen an den zugrunde liegenden Daten nach dem CQRS-Ansatz.

Patterns wie CQRS sind dann besonders vorteilhaft, wenn ein starkes Ungleichgewicht zwischen lesenden und schreibenden Operationen herrscht, da sich jede der beiden Seiten unabhängig von der anderen optimieren lässt. In typischen Webanwendungen gilt es, die Performance bei Lesezugriffen zu verbessern, während schreibende Requests deutlich seltener vorkommen und nicht so zeitkritisch sind.

Zustandsrepräsentationen vorzugenerieren, um das Frontend zu entlasten, ist keine neue Idee: Konzerne wie Facebook oder Twitter nutzen dieses Prinzip schon lange, um User Generated Content an Millionen Nutzer in aller Welt ausliefern zu können.

Die Anwendung von CQRS ist in der Regel mit einem Umdenken verbunden, das einige Zeit brauchen kann. Auch einige gewohnte Entwicklungs-Workflows ändern sich. So werden Änderungen an Template-Dateien erst nach dem Neugenerieren der Snippets im Frontend sichtbar. Dafür erhält man eine modular skalierbare, hochperformante Applikation, die durch eine konsequente Aufteilung in verschiedene Komponenten auch langfristig beherrsch- und wartbar bleibt. (odi) Arne Blankerts ist Mitbegründer und Principal Consultant der thePHP.cc Consulting Company. Er berät kleine und große Unternehmen unter anderem bei Fragen zum Aufbau und Betrieb von hochperformanten PHP-Umgebungen. Sebastian Heuer ist Developer Advocate bei der kartenmacherei GmbH. Zusätzlich unterstützt er als freiberuflicher Consultant Teams bei der Entwicklung langlebiger und wartbarer Software.

[/expand]