

Command Query Responsibility Segregation

Schneller lesen als schreiben

DDD sagt: Daten sind Daten. CQRS sagt: Daten ändern und Daten lesen sind zwei ziemlich verschiedene Aufgaben. Wenn man DDD mit CQRS kombiniert, erreicht man eine effiziente und extrem skalierbare Architektur.

Auf einen Blick



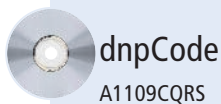
Dipl.-Inf. **Moukarram Kabbash** arbeitet als Senior Software Engineer bei der adesso AG in Dortmund. Seine Interessen erstrecken sich über Softwarearchitektur, nachrichtenbasierte Systeme bis hin zu funktionaler Programmierung. Sie erreichen ihn über moukarram.kabbash@adesso.de.



Dipl.-Ing. **Dominik Kania** ist Senior Software Engineer bei der adesso AG in Dortmund. Er betreut Kundenprojekte im .NET-Umfeld. Die Schwerpunkte seiner Arbeit liegen in der Konzeption verteilter Geschäftsanwendungen und der Entwicklung von Unternehmensportalen im SharePoint-Bereich. Sie erreichen ihn über dominik.kania@adesso.de.

Inhalt

- Lesevorgänge und Schreibvorgänge architektonisch voneinander trennen.
- Denormalisierte Datenbanken für Lesevorgänge bereithalten.
- Ein nachrichtenbasiertes System mit dem Enterprise Service Bus (ESB) MassTransit erstellen.



Der domänengesteuerte Entwurf (Domain-Driven Design) findet heutzutage in zahlreichen Softwareprojekten Anwendung [1]. Denn eine große Herausforderung bei der Entwicklung komplexer Softwaresysteme liegt darin, das System möglichst zukunftssicher und flexibel zu gestalten. Änderungs- oder Erweiterungswünsche sollen mit möglichst geringem Aufwand realisierbar sein. Um dieser Anforderung gerecht zu werden, ist es zwingend erforderlich, das Prinzip der Trennung der Anliegen (Separation of Concerns) [2] beim Entwurf der Systemarchitektur einzuhalten.

Dieses Prinzip definiert, dass ein Anliegen durch exakt ein Modul innerhalb des Softwaresystems repräsentiert wird und nicht über mehrere Module verstreut werden soll. Denn eine enge Kopplung führt dazu, dass bei Änderungen an einem Modul alle Abhängigkeiten richtig erkannt werden müssen, um ungewollte Nebeneffekte zu verhindern. Außerdem erweist es sich aufgrund der engen Verzahnung der Module untereinander oftmals als problematisch, automatisierte Tests zu erstellen.

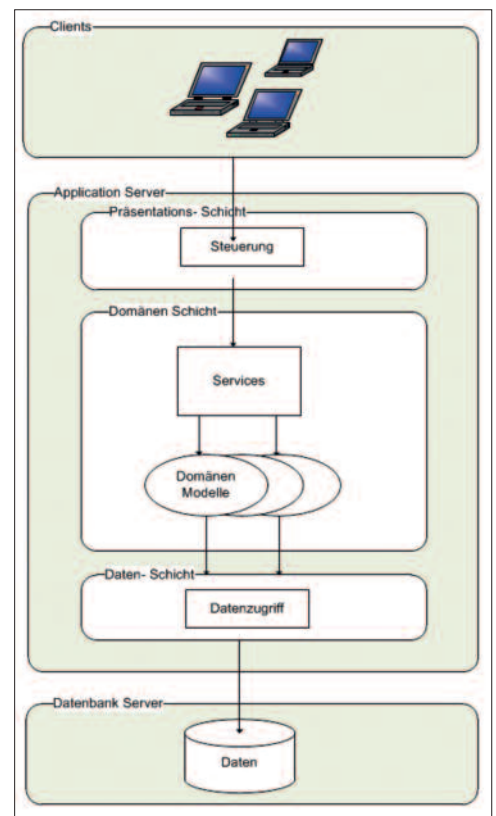
Die Folge ist, dass mögliche Auswirkungen einer Änderung manuell, also mit großem Zeitaufwand, getestet werden müssen. Dieser Zustand führt in manchen Softwareprojekten zu einer gewissen Phobie gegenüber Änderungs- oder Erweiterungswünschen.

Eine klassische DDD-Architektur

Abbildung 1 stellt den typischen Aufbau eines traditionellen Drei-Schichten-Systems dar.

Die Implementierung der Geschäftslogik findet hier in einer speziell dafür ausgelegten Domänenschicht statt. Diese Schicht ist von der technischen Infrastruktur (Datenbank, Logging etc.) und der Präsentationslogik unabhängig und bildet den Kern der Softwarelösung. Die Domänenschicht setzt sich aus den Domänenmodellen zusammen. Diese Modelle beinhalten sowohl Domänenobjekte (Entities) als auch die Geschäftsregeln eines bestimmten Aufgabengebietes. Sie werden häufig auch als Fachbereichsmodelle bezeichnet.

Die Schwierigkeit bei der Modellierung einer Domäne besteht darin, das Modell durchgehend handhabbar zu halten. Würde etwa ein Modell Aspekte aus mehreren Kompetenzen dieser Do-



[Abb. 1] Traditionelle Drei-Schichten-Architektur.

mäne beinhalten, würde es schnell zu komplex und unübersichtlich werden. Außerdem werden die Domänenobjekte innerhalb des Modells eine hohe Abhängigkeit voneinander aufweisen. Somit sollte ein Domänenmodell immer klare Grenzen haben und nur innerhalb eines bestimmten Kontexts (Bounded Context) verwendet werden.

Ein Kunde hat beispielsweise in der Finanzabteilung eine ganz andere Bedeutung als in der Logistikabteilung. Somit wäre es sinnvoll, dieses Objekt in zwei voneinander unabhängigen Modellen zu betrachten. Da aber auch immer Operationen existieren, welche nicht exakt einem Modellkontext zugeordnet werden können, wird in vielen DDD-Architekturen eine Applikationsschicht konstruiert. Über diese Schicht werden die eigentliche Kommunikation und Übersetzung der Domänenobjekte zwischen den unabhängigen Kontexten abgewickelt. Dabei wird die Datenübertragung zwischen dem UI und der Ap-

plikationsschicht durch Data-Transfer-Objekte (DTO) [3] realisiert.

DDD – Das Dilemma

Wie bereits beschrieben, ist ein Domänenmodell immer sehr spezifisch und nur für einen bestimmten Zweck optimiert. Ein Domänenmodell kann somit nicht so einfach wie ein Schweizer Messer für verschiedene Zwecke eingesetzt werden. Diese Tatsache kann aber nicht immer in einer klassischen DDD-Architektur berücksichtigt werden. Denn die Schreib- und die Leseoperationen verwenden das gleiche Modell. Das Modell muss dann gleichermaßen für Datenauswertungen und ein transaktionales Verhalten optimiert werden. Dafür müssen zahlreiche Transformationsoperationen implementiert werden. Um etwa die Kundendaten in einer Anwendung anzuzeigen, müssen zuerst folgende Transformationen durchgeführt werden:

Database -> Domain Object -> [DTO ->] View Model Object

Aber ist dieser Aufwand wirklich nötig? Denn beim Anzeigen der Daten wird keine Geschäftslogik gebraucht. Die Command Query Responsibility Segregation (CQRS) hilft, diese Transformationen zu reduzieren.

CQRS – Eine Evolution

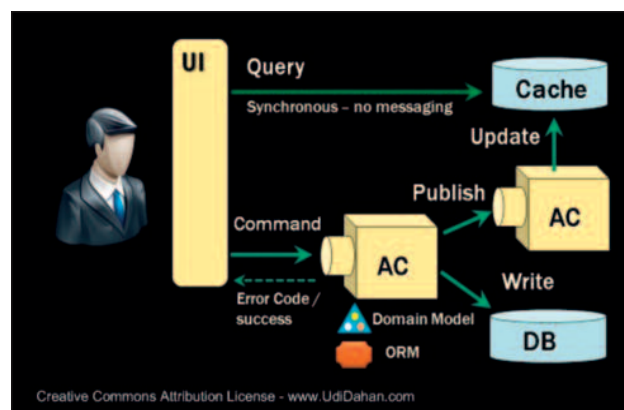
Abbildung 2 wurde aus dem Artikel *Clarified CQRS* von Udi Dahan [4] entnommen und demonstriert einen einfachen Aufbau eines nach den CQRS-Prinzipien entworfenen Systems. Die CQRS-Architektur zeichnet sich dadurch aus, dass Schreibvorgänge (Commands) und Lesevorgänge (Queries) in jeweils separaten Datenbanken ausgeführt werden. Durch die Separierung kann für die Ausführung der Queries eine skalierbare Infrastruktur implementiert werden. Commands, die eine Zustandsänderung bewirken, werden dagegen in einer transaktionalen, asynchronen Umgebung durchgeführt.

Lesevorgänge (Queries)

Bei der Darstellung von Daten stellt sich immer die Frage nach der Aktualität der Daten. Udi Dahan vertritt die Meinung, dass Daten direkt nach dem Rendern veraltet sind. Denn bevor sie über das Netzwerk übertragen und verwertet wurden, könnten sie schon wieder von einem anderen Prozess geändert worden sein.

Somit ist es auch legitim, dass Lesevorgänge auf eine denormalisierte Datenquelle zugreifen, die dem aktuellen Stand even-

[Abb. 2] Systemarchitektur bei Nutzung der Command Query Responsibility Segregation (CQRS).



tuell etwas hinterherhinkt, da sie nicht gleichzeitig mit der Ausführung einer Businessaktion aktualisiert werden kann.

Die Lesedatenquelle ist passend für die Anforderungen des UI ausgelegt. Es wird für jede Ansicht in dem UI eine Tabelle in der Datenbank angelegt, die den gleichen Aufbau wie die Ansicht aufweist. Dadurch entfällt der Transformationsvorgang bei jedem Lesezugriff. Außerdem werden die Daten denormalisiert persistiert, sodass komplexe Select-Anweisungen überflüssig werden und eine optimale Voraussetzung für Leseoperationen geschaffen wird.

Diese Technik findet heutzutage eher im Business-Intelligence-Bereich Anwendung. So liegen etwa die Daten in einem Data Warehouse meistens in einem sinnvollen Maße denormalisiert vor, um eine optimale Performance bei der Berichterstellung zu ermöglichen. Die Kehrseite der Medaille ist, dass zusätzlicher Aufwand betrieben werden muss, um die Konsistenz der Daten sicherzustellen, da auf Mechanismen wie die referenzielle Integrität bewusst verzichtet wird. Zudem muss ein Replikationsvorgang Schreib- und Lesedatenbank synchron halten.

Schreibvorgänge (Commands)

Bei den Schreibvorgängen ist ein wichtiges CQRS-Prinzip, dass diese Operationen niemals Daten zurückgeben sollten. Die Einhaltung dieses Prinzips ist die Voraussetzung dafür, bei den Schreibvorgängen eine asynchrone Kommunikation zu ermöglichen. Ein Schreibvorgang durchläuft folgende Zustände:

- Zuerst erzeugt der Client anhand der Daten, die geändert werden sollen, einen Befehl (Command).
- Dieser Befehl wird asynchron an eine zuständige Komponente gesandt. Die Commands werden unter Berücksichtigung der Geschäftsregeln ausgeführt.

- Nach der erfolgreichen Ausführung eines Commands wird ein Ereignis ausgelöst. Andere Komponenten können dieses Event dafür nutzen, um ihre Daten zu aktualisieren oder andere Events auszulösen.

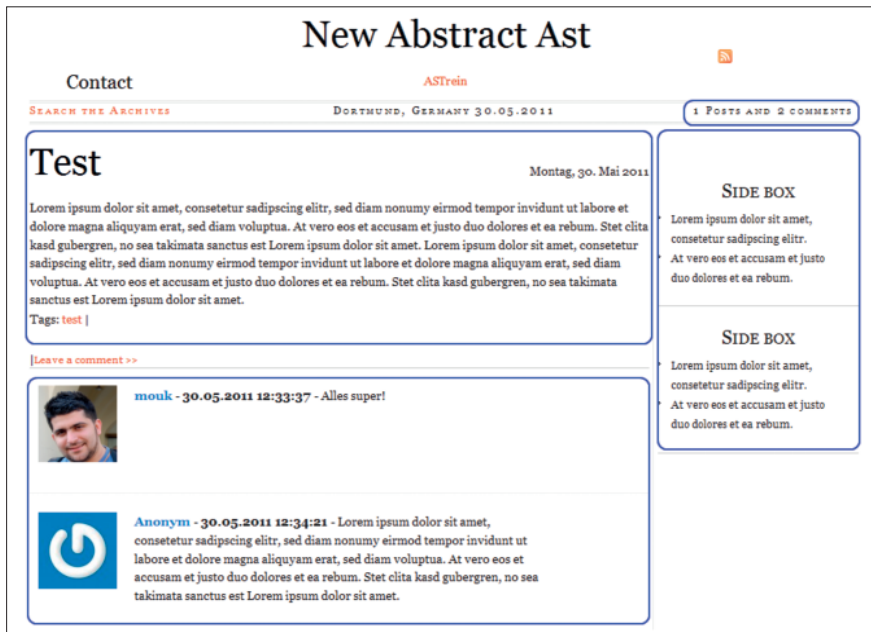
CQRS – Event Sourcing

Fowler definiert das Event-Sourcing-Pattern so: „... Captures all changes to an application state as a sequence of events.“ [5] Dieser Ansatz persistiert nicht wie bislang nur den aktuellen Zustand einer Anwendung, sondern speichert alle Aktionen, die zu Zustandsänderungen führen, in einem sogenannten Event Store. Neue Operationen werden immer nur hinzugefügt, sodass keine wichtigen Informationen verloren gehen können.

Dieses Pattern war allerdings bislang in traditionellen Architekturen kaum praktikabel umsetzbar. Bereits eine einfache Kundenabfrage, wie beispielsweise „Alle Kunden mit dem Vornamen Gustav“, hätte zu einem nicht tragbaren Aufwand geführt. Diese Problematik entfällt in CQRS, da alle Abfragen in dem dafür optimierten Lese-Modell ausgeführt werden.

Die Umsetzung dieses Patterns ist also in einer CQRS-Architektur mit wenig Aufwand möglich und kann, da keine Informationen verloren gehen, einige Vorzüge bringen. So kann etwa die Lesedatenbank jederzeit aus dem Event Store rekonstruiert werden. Dieser Umstand führt dazu, dass eventuelle Synchronisationsprobleme zwischen den separierten Datenbanken beherrschbar bleiben. Wird beispielsweise wegen eines Programmierfehlers nach dem Anlegen eines neuen Kunden die Lesedatenbank nicht synchronisiert, so kann dieses Event nach Behebung des Fehlers erneut ausgelöst werden.

Zudem können in einer CQRS-Architektur neue Möglichkeiten genutzt werden, um



[Abb. 3] Die Benutzeroberfläche kombiniert den Output verschiedener Services.

Konflikte zu behandeln, da durch das Event Sourcing transparent wird, welche Akteure welche Aktionen ausgeführt haben. Das erlaubt Rückschlüsse auf die jeweiligen Intentionen, was die Grundlagen für ein elegantes Konfliktmanagement schafft.

In einer CQRS-Architektur findet ein Umdenken in Bezug auf die Modellierung

der Businessaktionen statt. So müssen diese nicht wie gewohnt als CRUD-Operationen wahrgenommen werden. Stattdessen können sie in der Businesssprache modelliert werden, da das Schreibmodell nicht für Abfragen optimiert werden muss. Diese SQL-freie Modellierung ermöglicht eine enge Kopplung des Modells an die domä-

nenspezifische Sprache. Man spricht von „Beitrag erstellen“ oder „Titel ändern“ statt von INSERT- und UPDATE-Anweisungen.

Durch die Trennung der einzelnen Kompetenzen können Komponenten unabhängig voneinander entwickelt und vorangetrieben werden. Ändert sich das Domänenmodell einer Komponente, um beispielsweise weitere Daten aufzunehmen, so hat das keinen Einfluss auf die Entwicklung der restlichen Komponenten. Wird etwa in der Hauptanzeige eines Blogs eine neue Eigenschaft *LastModified* benötigt, so muss diese nur zum entsprechenden Modell hinzugefügt werden.

MLog: CQRS in Action

Es folgt ein Beispiel, das die Prinzipien und Ideen von CQRS praxisnah erläutert. Die hier vorgestellte Anwendung ist ein Forum namens MLog. Registrierte Mitglieder können in diesem Forum Beiträge verfassen, editieren und wieder entfernen. Anonyme Benutzer sollen Beiträge nicht bearbeiten, sondern nur kommentieren dürfen. Die Applikation stellt in einer Übersicht Beiträge und Kommentare dar. Eine wichtige Anforderung an die Anwendung ist, dass sie skalierbar sein soll. Auch bei einer hohen Belastung soll ein optimales Nutzererlebnis gewährleistet sein.

MassTransit. Ein Crashkurs

MassTransit ist ein Enterprise Service Bus (ESB). Es ist ein Framework, das eine für den Nachrichtenaustausch geeignete Infrastruktur zur Verfügung stellt. Mithilfe von MassTransit können Services transaktionell Nachrichten untereinander austauschen. Nachrichten können mittels Microsoft Message Queues an alle Abonnenten ausgeliefert werden. Bei den Empfängern dieser Nachrichten sucht MassTransit die passenden Handler, instanziiert sie und führt sie aus. Diese Handler sind Klassen, welche durch ein bestimmtes MassTransit-Interface markiert wurden.

Im Folgenden lernen Sie die wichtigsten Konzepte von MassTransit kurz kennen. Weiterführende Ressourcen finden Sie auf der offiziellen Homepage des Projekts [6] oder in diversen Blogs.

Nachrichten

Nachrichten sind in MassTransit – und in der ESB-Kultur allgemein – ein sehr wichtiges Konzept. Wie herkömmliche Briefe sind ESB-Nachrichten ein Einwegkommunikationsmodell. Eine Nachricht kann jedoch eine Antwortadresse enthalten, falls eine Antwort gewünscht ist. Das Antworten ist eine eigenständige Aktion, die auf der technischen Ebene keinen Bezug zur originalen Nachricht enthält. Fachlich gesehen, können die Nachrichten durch vereinbarte Felder aufeinander verweisen. Für den Postboten sind diese Verweise jedoch irrelevant.

Der Vorteil dieses Verfahrens liegt darin, dass die Kommunikation asynchron stattfindet, was zu einer erhöhten Unabhängigkeit zwischen den Services führt. Wenn Ihnen ein Bekannter beispielsweise einen Brief schickt, während Sie im Urlaub sind, dann können Sie den Brief trotzdem lesen, wenn Sie zurück sind. Im Gegensatz dazu kann der Bekannte Sie nur dann anrufen, wenn Sie zu Hause sind. Ähnlich läuft es bei Services. Ein Service

kann einem anderen eine Nachricht zuschicken, auch wenn dieser offline ist. Die Realisierung der Nachrichten erfolgt in der Regel durch Message Queueing. Eine Queue ist eine Art Postfach, in die entfernte oder lokale Prozesse Nachrichten schreiben können. Diese Nachrichten werden der Reihe nach aus der Queue entfernt und abgearbeitet. Queues sind persistent, das heißt, bereits empfangene Nachrichten bleiben erhalten, bis sie aus der Queue entfernt werden.

Subscription

Komponenten können Nachrichten entweder gezielt zugeschickt bekommen oder selbst welche abonnieren. Wenn eine Komponente eine Nachricht abonniert, will sie jede publizierte Instanz dieser Nachricht erhalten.

MassTransit verwaltet für jede Nachricht eine Liste aller entsprechenden Subscriber. Sobald ein Event einer bestimmten Art bei einem Service eintritt, erstellt dieser Service eine neue Instanz des entsprechenden Nachrichtentyps und publiziert sie. Der Bus sendet die Nachricht an alle Subscriber dieses Nachrichtentyps. Auf diese Weise entsteht das bekannte Event-Verhalten, jedoch verteilt und persistent.

Handler

Handler sind Klassen, die ankommende Nachrichten abarbeiten. Durch das Implementieren eines generischen Interface können Klassen als Handler markiert werden. Der Typparameter gibt den Typ der zu bearbeitenden Nachricht an. MassTransit ist dafür verantwortlich, die Handler einer Nachricht zu verwalten und, beim Ankommen einer Nachricht, zu instanzieren und auszuführen. Für einen Nachrichtentyp kann es mehrere Handler geben. Dies bringt einen großen architektonischen Vorteil: Handler können sehr granular und unabhängig voneinander entwickelt werden.

Architektonisch ist die Applikation in zwei unabhängige Komponenten geteilt: die Beitrags- und die Benutzerverwaltung.

Der Beitragsverwaltungsservice verwaltet Beiträge und Kommentare unter Berücksichtigung der jeweiligen Benutzerrechte. Da ausschließlich registrierte Benutzer Beiträge verfassen dürfen, muss dieser Service auch in der Lage sein, auf die Daten des Benutzerverwaltungsservice zuzugreifen. Dies geschieht in einer serviceorientierten Umgebung durch Benachrichtigungen und nicht durch direkte Datenbankzugriffe. Abbildung 3 zeigt, wie sich die Benutzeroberfläche aus den Daten verschiedener Komponenten zusammensetzt.

Der Benutzerverwaltungsservice benötigt wiederum Beitragsinformationen, beispielsweise um den Status des Benutzers aktualisieren zu können. Um einen gewissen Grad an Unabhängigkeit zu gewährleisten, kommunizieren die Services nicht direkt miteinander, sondern über Events. An dieser Stelle empfiehlt es sich, den Infokasten *MassTransit. Ein Crashkurs* zu lesen. Da die Demoapplikation MassTransit zum Übertragen von Nachrichten verwendet, ist ein elementares Verständnis der Terminologie und Arbeitsweise sehr wichtig.

Der Beitragsverwaltungsservice

Dieser Service besteht aus einer Menge von Handlern, die verschiedene Businessaktionen durchführen. Dieser Service verwendet Event Sourcing als Persistenzmechanismus. So ist gewährleistet, dass der Zustand des Systems zu jedem beliebigen Zeitpunkt wiederhergestellt werden kann. Dafür müssen lediglich alle Events bis zum gewünschten Zeitpunkt abgespielt werden.

Die Businessaktionen werden in Form von Commands im verantwortlichen Teilsystem als Nachrichten der Reihe nach abgearbeitet. Um den Kern herum befinden sich mehrere Komponenten, die auf Events warten und, beim Empfangen einer geeigneten Nachricht, ihre Daten entsprechend aktualisieren.

Durch die Trennung der einzelnen Kompetenzen und die asynchrone Natur der Kommunikation lässt sich das System besser skalieren. Nachrichten können sich in der Warteschlange aufstauen, während die Views, und damit das Benutzererlebnis, unbeeinflusst davon weiterfunktionieren.

Die View-Komponenten befinden sich nahe dem Webserver, idealerweise auf dem gleichen physischen Rechner, sodass die Daten fürs Generieren einer View schnell aufbereitet werden können. Listing 1 zeigt

Listing 1

Ein Handler verarbeitet eine Nachricht und schickt eine Benachrichtigung.

```
public class NewPostHandler : HandlerBase<CreateNewPost>
{
    //.....
    protected override void Process(CreateNewPost message) {
        var newPost = new NewPostHasBeenCreated
        {
            Post = message.Post,
            CreatedOn = DateTime.Now,
            Slug = PostFactory
                .GetSlug(message.Post.Title)
        };
        Bus.Publish(newPost);
    }
}
```

Listing 2

Das Lesemodell wird anhand der Nachricht aktualisiert.

```
public class PostListHandler : HandlerOf<NewPostHasBeenCreated>
{
    public PostListHandler(ISession session)
        : base(session)
    { }

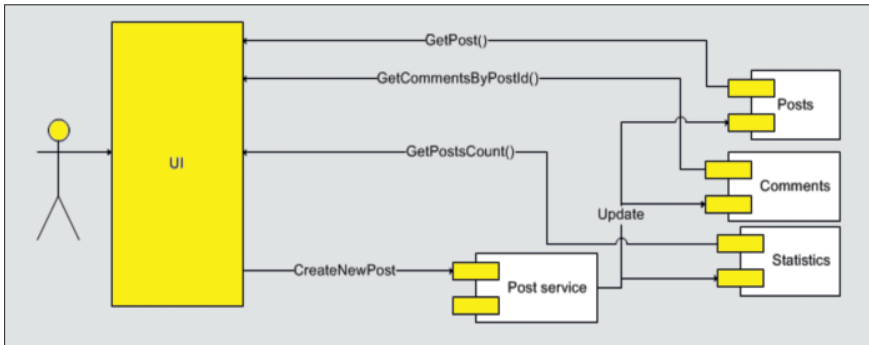
    public override void Consume(NewPostHasBeenCreated message)
    {
        var post = new Post
        {
            Id = message.Post.Id,
            Title = message.Post.Title,
            Slug = message.Slug,
            Body = message.Post.Body,
            CreatedOn = message.CreatedOn,
            Tags = message.Post.Tags,
            LastModifiedOn = DateTime.Now
        };
        SaveEntity(post);
    }
}
```

einen Handler, der sein Modell als eine Serie von Aktionen in der Datenbank persistiert und bei jeder Änderung Abonnenten durch Events benachrichtigt.

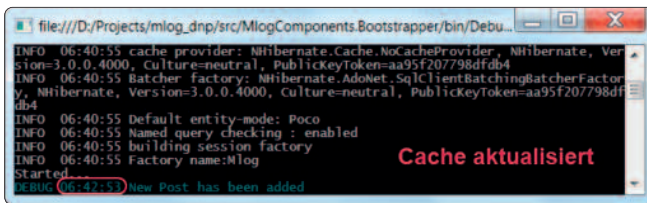
Eine Businesstransaktion soll am Beispiel der Beitragserfassung untersucht und analysiert werden. Folgendes Szenario liegt zugrunde: *Ein eingeloggtter Benutzer verfasst einen neuen Beitrag und speichert ihn ab. Daraufhin aktualisiert sich die Anzeige, und der neu verfasste Artikel ist, eventuell nach einer kurzen Verzögerung, sichtbar.*

Zwischen dem Speichern und der Anzeige auf der Website geschehen folgende Aktionen im Hintergrund:

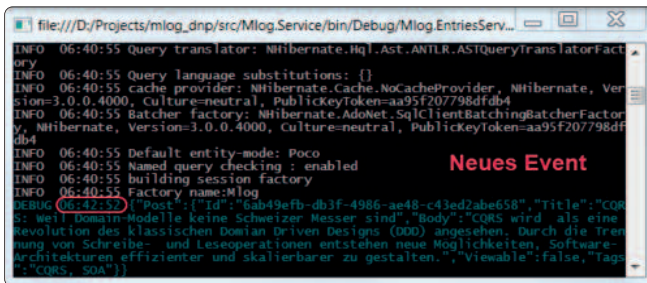
Der Webservice empfängt die eingegebenen Daten, prüft sie auf die syntaktische Korrektheit und kapselt die Daten in einem Command. Commands sind im Wesentlichen DTOs (Data-Transfer-Objekte). Sie enthalten in der Regel kein Verhalten, sondern lediglich Daten. Der Typ eines Commands bestimmt die auszuführende Aktion. So heißt das in diesem Szenario verwendete Command *CreateNewPost*. Das Command enthält sämtliche für die Ausführung benötigten Daten. *CreateNewPost* enthält folglich unter anderem den Titel, den Text und die Daten des Verfassers.



[Abb. 4] Ein Command veranlasst die Aktualisierung der Lesemodelle.



[Abb. 5] Schreib- und Lesemodelle sind nicht sofort synchron.



Das generierte Command wird mittels MassTransit an einen bestimmten Service weitergeleitet, danach ist der Ablauf auf dem Webserver beendet. Die lokale Queue nimmt die versandten Nachrichten entgegen und speichert sie. Der Absender verliert von diesem Zeitpunkt an die Kontrolle über die Nachricht. Im Hintergrund versucht MSMQ, die Nachricht aus der Out-

going-Queue zu lesen und an die Ziel-Queue zu schicken. Scheitert der Vorgang, etwa weil die Zieladresse nicht erreichbar ist, wird es zu einem späteren Zeitpunkt erneut versucht. Für den Webserver bedeutet das, dass er für weitere Anfragen zur Verfügung steht, auch wenn der Empfänger nicht erreichbar sein sollte. Diese Technik wird als „Store and Forward“ bezeichnet.

Listing 3

Kommentare getrennt von den Beiträgen verwalten.

```
public class CommentsController : ControllerBase {
    public ActionResult Get(Guid postId)
    {
        IList<Comment> comments;

        using (NHibernateSession.BeginTransaction())
        {
            comments = NHibernateSession
                .QueryOver<Comment>()
                .Where(a => a.PostId == postId)
                .OrderBy(c => c.CreatedOn).Asc
                .List();
        }
        return Json(comments, JsonRequestBehavior.AllowGet);
    }
}
```

Nachdem der neue Beitrag zum Schreibmodell hinzugefügt wurde, werden die damit verbundenen Lesemodelle aktualisiert. Das soll anhand von zwei Lesemodellen demonstriert werden, siehe Abbildung 4 und Listing 2. Das erste Modell enthält die aktuellen Beiträge und wird für die Startseite verwendet. Die Artikel in diesem Modell sind in einer denormalisierten Form abgelegt und enthalten alle für die Darstellung relevanten Daten. So werden auch aggregierte Daten wie etwa die Anzahl der Kommentare mitgespeichert. Der Benutzername und die ID werden auch denormalisiert mit dem Artikel zusammen gespeichert. Zusätzlich können die für eine Aktualisierung benötigten IDs ebenfalls gespeichert werden, siehe Abbildung 5.

Im zweiten Modell werden die Kommentare verwaltet. Dabei werden diese getrennt von den dazugehörigen Beiträgen persistiert, sodass an dieser Stelle keine relationale Beziehung verwaltet werden muss. Denn der Bezug zu einem Beitrag wird lediglich als ein weiteres Attribut verwaltet und nicht als eine relationale Beziehung. Die Kommentare werden durch Ajax-Aufrufe nachträglich aus der Datenbank geladen und dargestellt. An dieser Stelle nutzt man den Umstand, dass ein Leser zuerst den Artikel liest, bevor er die Kommentare erreicht, siehe Listing 3.

Weitere Funktionen können als weitere unabhängige Komponenten dazuimplementiert werden.

Fazit

Der Artikel hat einen ersten Zugang zum Thema CQRS geboten. Weiterführende Ressourcen finden Sie unter [7], [8] und [9]. [ml]

- [1] Eric Evans, Domain-Driven Design, Tackling Complexity in the Heart of Software, Addison-Wesley 2003
- [2] Separation of Concerns, http://en.wikipedia.org/wiki/Separation_of_concerns
- [3] Martin Fowler, Data Transfer Object, www.dotnetpro.de/SL1109CQRS1
- [4] Udi Dahan, Clarified CQRS, www.dotnetpro.de/SL1109CQRS2
- [5] Martin Fowler, Event Sourcing, www.dotnetpro.de/SL1109CQRS3
- [6] MassTransit, Lean Service Bus for .NET, <http://masstransit-project.com>
- [7] Linksammlung über CQRS, <http://cqrsinfo.com>
- [8] CQRS à la Greg Young, www.dotnetpro.de/SL1109CQRS4
- [9] OREDEV 2010, Session: Unleash Your Domain, www.dotnetpro.de/SL1109CQRS5