

Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

NJM - CRUD-Operationen via HTTP mit NodeJS und MongoDB



Überblick und Lernziele

In den vorangegangenen Lerneinheiten haben Sie die aktuellen Ausdrucksmittel von HTML, CSS und JavaScript kennengelernt und damit eine vorgegebene Designvorlage für eine mobile Anwendung partiell umgesetzt. Die im Design vorgesehenen Ansichten haben Sie darüber hinaus unter Verwendung von `XMLHttpRequest` und anderer JavaScript APIs dynamisch auf der Grundlage von JSON Daten aufgebaut, in denen die darzustellenden Inhalte beschrieben wurden.

Bisher wurden diese Daten in Form von JSON-Dateien statisch durch den von uns verwendeten Webserver bereitgestellt. Die vorliegende Lerneinheit setzt an dieser Stelle an und wird Ihnen zeigen, wie Sie die betreffenden Daten selbst auf Seiten der mobilen Anwendung erstellen und an eine serverseitige Anwendung übertragen können und wie die Daten serverseitig persistiert – d. h. dauerhaft gespeichert – werden können.

Darüber hinaus werden Sie gespeicherte Daten modifizieren und ggf. wieder aus dem Datenspeicher entfernen. Für alle diese *schreibenden* Datenzugriffe werden wir – wie für den bereits durchgeführten *lesenden* Zugriff auf serverseitige Daten – `XMLHttpRequest` verwenden.

Serverseitige Anwendungen

Bezüglich der Realisierung der serverseitigen Anwendungskomponente, auf die wir die mobile Anwendung zugreifen lassen, wie auch im Hinblick auf die Realisierung der Datenspeicherung selbst, bestehen zahlreiche Möglichkeiten. So lassen sich Anwendungen, die einer Client-Anwendung den lesenden und schreibenden Datenzugriff auf serverseitige Datenspeicher ermöglichen, u. a. in Java, PHP, Python, Ruby und zahlreichen anderen Sprachen entwickeln. Als Datenspeicher wäre zunächst der Einsatz einer relationalen Datenbank wie MySQL oder PostgreSQL denkbar, für deren Zugriff jeweils Bibliotheken für die genannten Sprachen zur Verfügung stehen.

NodeJS und MongoDB

Als Alternative hierzu werden wir in der vorliegenden Lerneinheit und für den weiteren Verlauf der Lehrveranstaltung eine unter dem Name NodeJS zunehmend verbreitete serverseitige Ausführungsumgebung für JavaScript vorstellen. Wir werden also serverseitig die gleiche Programmiersprache einsetzen, die wir bereits für die Umsetzung der im Browser ausgeführten Client-Anwendung verwendet haben. Dabei werden wir verstärkt mit einer API Bekanntschaft machen, die – vergleichbar `XMLHttpRequest` – in hohem Maße asynchron ausgeführte Funktionen bereitstellt, denen zur Weiterverarbeitung des Funktionsergebnisses Callback-Funktionen übergeben werden müssen.

Für die Datenspeicherung werden wir als Alternative zu relationalen Datenbanken mit der Datenbanktechnologie MongoDB eine sogenannte „NoSQL“-Datenbank einsetzen – dieser Begriff, der als „*not only SQL*“ gelesen werden kann, bringt zum Ausdruck, dass eine Datenbank nicht nur die Konstrukte verwendet, die aus relationalen Datenbanken bekannt sind, u. a. Tabellen, Spalten und Beziehungen zwischen Tabellen mittels Fremdschlüsseln. So ermöglicht MongoDB beispielsweise die einfache Speicherung von beliebig komplex „verschachtelten“ Objekten – wie sie z. B. für die Repräsentation der in einer Ansicht darzustellenden Inhalte der Beispielanwendung von uns genutzt wurden – in einer einzigen „Tabelle“.

Collection statt Tabelle

Der korrekte Begriff, den wir später einführen werden, ist der Begriff der Collection.

Auf MongoDB werden wir aus NodeJS mit einer JavaScript API zugreifen, die ihrerseits wesentlich aus einer Menge asynchroner Funktionen besteht.

Ein zentraler Begriff, der sich auch im Titel der vorliegenden Lerneinheit findet, ist der Begriff der *CRUD-Operationen*. Mit diesem Kürzel werden üblicherweise die folgenden vier grundlegenden lesenden und schreibenden Operationen bezeichnet, die für die Erstellung und Manipulation von Datenbeständen – Dateisysteminhalten oder Inhalten einer Datenbank – erforderlich sind:

- Erstellen von Daten (*create*)
- Auslesen von Daten (*read*)
- Aktualisierung bestehender Daten (*update*)
- Löschen von Daten (*delete*)

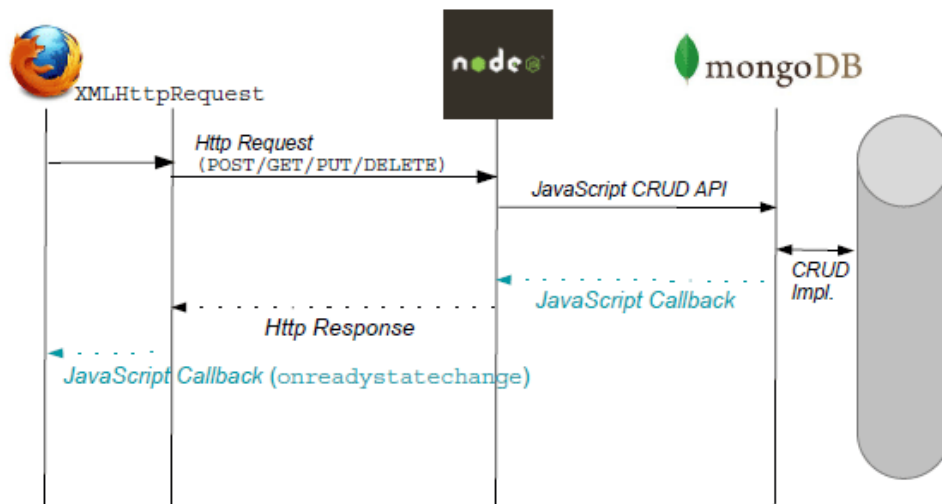
Zunächst werden wir einige Ideen, die NodeJS und MongoDB zugrunde liegen, vorstellen und die wichtigsten Aspekte der von uns jeweils zu verwendenden Funktionen, wie sie auch in den Implementierungsbeispielen genutzt werden, exemplarisch erläutern. So werden wir z. B. zeigen, wie in NodeJS auf sehr einfache Weise ein Webserver implementiert werden kann, auf den wir mittels `XMLHttpRequest` von unserer Client-Anwendung zugreifen können.

Wir illustrieren schließlich, wie die Ausdrucksmittel von HTTP verwendet werden können, um einem Client die Identifikation von CRUD-Operationen zu ermöglichen, die durch eine serverseitige Anwendung ausgeführt werden sollen. Damit legen wir die Grundlage für den clientseitigen Zugriff auf serverseitig ausgeführte CRUD-Operationen, deren Umsetzung mittels NodeJS und MongoDB erfolgt.

Das „Gesamtbild“ der Lerneinheit finden Sie in folgender Abbildung dargestellt. Ein Fokus liegt hier auf der serverseitigen Verarbeitung bzw. dem Zugriff darauf via `XMLHttpRequest`. Dabei blenden wir zunächst aus, wie die Daten für die auf einem Server aufzurufenden CRUD-Operationen durch den Nutzer einer mobilen Anwendung bereitgestellt werden können. Letzteren Aspekt wird dann die nachfolgende Lerneinheit ausführlich beleuchten.

Mit CRUD-Operationen werden wir uns darüber hinaus auch noch zu einem späteren Zeitpunkt befassen, wenn wir aufzeigen werden, welche Operationen die `IndexedDB` API zum Zweck der *lokalen Datenspeicherung* auf einem Endgerät bereitstellt.

Abb.: Zusammenführung von `XMLHttpRequest`, NodeJS Zugriff auf MongoDB und den auf der Datenbank durchgeführten CRUD Operationen





Lernziele

Lernziele

Nachdem Sie die Lerneinheit durchgearbeitet haben, sollten Sie in der Lage sein:

- Die client- und serverseitige Verwendung von CRUD-Operationen zu begreifen.
- Daten von einer mobilen Anwendung an eine serverseitige Anwendung zur dauerhaften Speicherung zu übertragen.
- Gespeicherte Daten zu modifizieren und ggf. zu löschen.
- Die Motivation zur Verwendung von NodeJS als Web Server zu erläutern.
- Eine Abgrenzung zwischen SQL und NoSQL Technologien zur persistenten Datenspeicherung vorzunehmen und dabei u. a. die Anforderungen der Typsicherheit und der Konsistenz von Datenbeständen zu berücksichtigen.
- Die Konsequenzen bezüglich der möglichen Einsatzszenarien für NoSQL-Technologien zu beurteilen
- Ausdrucksmittel von HTTP in geeigneter Form einzusetzen, um mittels ihrer die durch NodeJS auf MongoDB aufzurufenden lesenden und schreibenden Zugriffsoperationen zu initiieren.



Gliederung

Gliederung

- Serverseitige Datenverarbeitung
- NodeJS
- MongoDB
- Datenzugriffsoperationen via HTTP
- Zusammenfassung
- Prüfungsfragen
- Übungen



Zeitbedarf

Zeitbedarf und Umfang



Für die Bearbeitung der Lerneinheit benötigen Sie etwa 3 Stunden. Für die Bearbeitung der Übungen zur Beispielanwendung etwa 3 Stunden und für die Wissensfragen ca. 1,5 Stunden.

1 Serverseitige Datenverarbeitung

Nachfolgend werden Sie einige Hintergründe zu NodeJS und MongoDB kennenlernen sowie einen Einblick in die APIs erhalten, die die Implementierungsbeispiele verwenden und die Sie selbst im Rahmen des Übungsprogramms anwenden werden.

2 NodeJS

Herkunft und Historie

Wie viele quelloffenen und ohne Lizenzgebühren auch kommerziell nutzbare Technologien im Umfeld von Webanwendungen ist  NodeJS dem weit verzweigten „Ökosystem“ von Google entwachsen. Entwickelt wurde es seit 2009 durch den freiberuflichen Softwareentwickler Ryan Dahl in C++ auf Basis des JavaScript-Interpreters  V8, der in Googles Chrome Browser verwendet wird.

Erwähnt hatten wir vorher bereits, dass es sich bei NodeJS um eine Ausführungsumgebung für JavaScript handelt, die unabhängig von einem Browser lauffähig ist und damit z. B. für die Umsetzung serverseitiger Komponenten von Webanwendungen eingesetzt werden kann. So stellt NodeJS selbst eine umfangreiche JavaScript-Bibliothek zur Verfügung, mittels derer wesentliche Funktionen serverseitiger Anwendungen durch den Anwendungsentwickler umgesetzt werden können. Dazu gehören u. a. der Zugriff auf das Dateisystem des Rechners, auf dem NodeJS ausgeführt wird, die Unterstützung von Netzwerkkommunikationsfunktionen im Allgemeinen sowie im Besonderen die Handhabung von HTTP-Requests und Responses.

Mit NodeJS lässt sich also z. B. die Grundfunktionalität eines Webservers implementieren, wie wir sie in den vorangegangenen Lerneinheiten vorausgesetzt haben. Diese besteht in der Auslieferung statischer Ressourcen, die im Dateisystem vorliegen, an einen Client, der diese Ressourcen anhand der URL von HTTP `GET` Requests identifiziert.

2.1 Grundlagen

Eines der wichtigsten Leistungsmerkmale eines Servers stellt neben seiner Stabilität und Verfügbarkeit die Fähigkeit dar, gleichzeitig eine möglichst große Anzahl von Client-Requests mit minimaler Latenzzeit – d. h. möglichst verzögerungsfrei – zu bearbeiten und mit einem Response zu erwidern. Nicht immer liegt eine etwaige Verzögerung bei der Bearbeitung eines Requests jedoch in der Hand des Servers bzw. des Entwicklers des Servers. Treten beim Zugriff auf eine außerhalb des Servers laufende Datenbank beispielsweise Verzögerungen auf Seiten der Datenbank auf, dann wird diese Verzögerung durch den zugreifenden Server zwar an seine Clients weitergegeben, während des Datenbankzugriffs tut der Server jedoch selbst nichts, als auf eine Erwidern der Datenbank zu „warten“.

OS Threads

Problematisch wird ein solcher Wartezustand wenn er verhindert, dass der Server zur gleichen Zeit weitere Anfragen bearbeiten kann. Letzteres ist der Fall, wenn für die gleichzeitige Bearbeitung mehrerer Anfragen jeweils ein betriebssystemeigener Thread – OS Thread – pro Anfrage verwendet wird, da diesem eine fixe Speichergröße zugewiesen wird. Dieser zugewiesene Speicher ist so lange für den betreffenden Thread reserviert, wie dieser ausgeführt wird. Er steht also auch dann nicht für andere Verarbeitungsprozesse zur Verfügung, wenn der Thread nichts „tut“, außer darauf zu warten, dass ihm Daten geliefert werden – sei es von einer Datenbank, von einer anderen via HTTP zugreifenden Anwendung oder auch vom Dateisystem.

Während solcher Input/Output-Operationen ist der Thread jeweils nicht aktiv, sondern wartet lediglich auf das Ergebnis der jeweiligen Operation. Die an den Thread vergebenen Speicherressourcen liegen also während der Durchführung von I/O-Operationen brach und stehen nicht für etwaige gleichzeitig durchzuführende Verarbeitungsprozesse zur Verfügung. Resultat ist eine eingeschränkte Kapazität des Servers bezüglich der Menge verarbeitbarer Client-Requests und eine evtl. Verzögerung der Erwidern jedes einzelnen Requests, falls mehrere unabhängig voneinander durchführbare I/O-Operationen nacheinander ausgeführt werden, wie in folgender Abbildung angenommen.

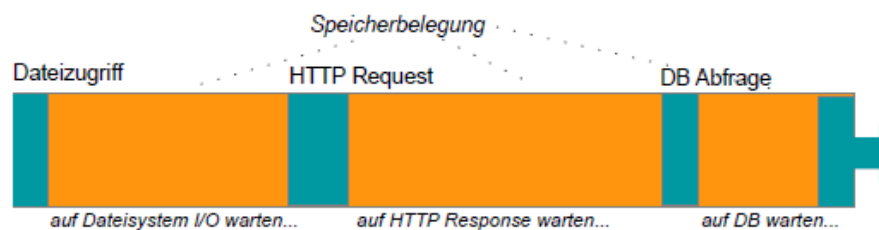


Abb.: Brachliegen von Speicherressourcen bei Verwendung von OS Threads

Quelle: [www. https://www.udemy.com/lectures/understanding-the-nodejs-event-loop-91298](https://www.udemy.com/lectures/understanding-the-nodejs-event-loop-91298)

Weshalb ist die geschilderte Problematik gerade für Anwendungsfälle wie den unseren relevant?

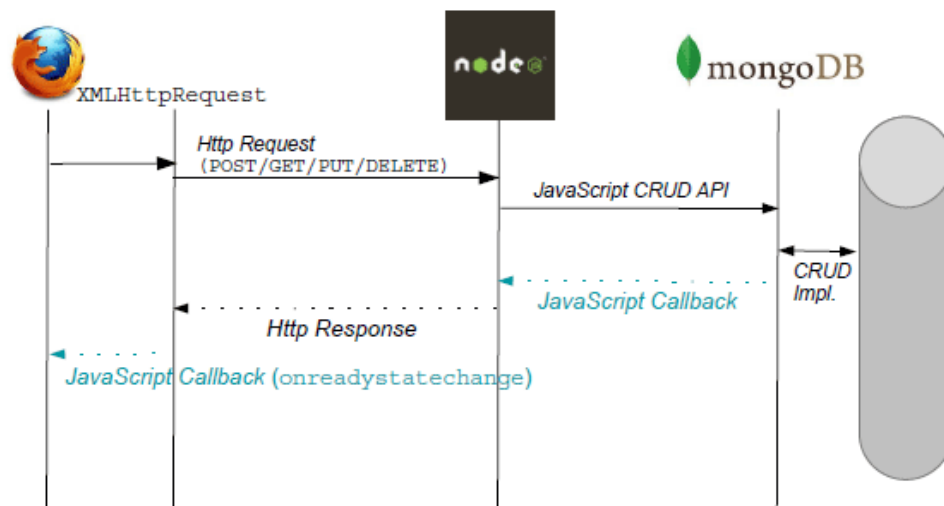


Abb.: Überblick über die Zusammenführung von XMLHttpRequest, NodeJS Zugriff auf MongoDB und den auf der Datenbank durchgeführten CRUD Operationen

Wie in der bereits gezeigten Abbildung zu sehen ist, wird in unserem Fall dem NodeJS Server eine bloße Mittlerfunktion zukommen. Diese besteht darin, dass HTTP-Requests in den Aufruf von CRUD-Operationen auf der MongoDB Datenbank „übersetzt“ werden, welche als eigenständige Anwendung betrieben wird und dass die Ergebnisse des Datenbankzugriffs im HTTP-Response an den Client zurück übermittelt werden. Die Server-Anwendung führt also nicht etwa ihrerseits komplexe Berechnungsvorgänge durch, für die beträchtliche Speicher und CPU-Ressourcen erforderlich wären. Sie übernimmt vielmehr eine Vermittlungsrolle zwischen lesenden und schreibenden Datenzugriffen, die jeweils über die Netzwerkschicht der verwendeten Rechnerhardware durchgeführt werden.

Angemerkt sei, dass eine solche Rolle exemplarisch für Server-Anwendungen ist, die dazu verwendet werden, die Funktionalität von Drittanwendungen ggf. öffentlich über HTTP zur Verfügung zu stellen. In unserem Fall ist dies lediglich die Funktionalität von MongoDB als Datenspeicher. Aber auch in Fällen, in denen z. B. geschäftslogische Operationen einer mehrschichtigen Unternehmensanwendung für den Zugriff via mobiler Endgeräte verfügbar gemacht werden sollen, ist der Einsatz einer vergleichbaren „Vermittler-Anwendung“ in Erwägung zu ziehen, die als zentraler Zugriffspunkt auf diejenigen Operationen der Unternehmensanwendung fungiert, die von der mobilen Anwendung genutzt werden sollen.

Die Ineffizienz bezüglich der Verwendung von Speicherressourcen, die in solchen Fällen bei Nutzung von OS Threads zu beobachten ist, ist in obiger Abbildung „Brachliegen von Speicherressourcen bei Verwendung von OS Threads“ noch einmal illustriert.

In Bezug auf die geschilderte Problematik, die Verarbeitungsprozesse mit einem hohen Anteil von I/O-Operationen für Server-Anwendungen mit sich bringen, zeichnet sich NodeJS dadurch aus, dass es zur gleichzeitigen Ausführung von Verarbeitungsvorgängen, die z. B. durch HTTP-Requests initiiert werden, eine feiner granulare Threadverwaltung verwendet. Insbesondere wird damit eine reservierte Zuweisung von Speicherressourcen an Threads, unabhängig von deren tatsächlicher Tätigkeit, vermieden, und NodeJS ist dazu in der Lage, den gesamten verfügbaren Speicher jeweils denjenigen Verarbeitungsvorgängen zuzuweisen, die zum gegebenen Zeitpunkt tatsächlich aktiv sind.

Erreicht wird dies vor allem dadurch, dass I/O-Operationen nicht synchron auf den betreffenden Threads durchgeführt werden, sondern an einen zentralen „Verwaltungsprozess“ delegiert werden, der als Event-Loop bezeichnet wird. Liegt für eine I/O-Operation ein Ergebnis vor, wird der betreffende Vorgang aus dem Event-Loop heraus notifiziert und fortgesetzt. Dies erfolgt durch Aufruf einer Callback-Funktion, die beim Aufruf der I/O-Operation übergeben werden kann. Da I/O-Operationen asynchron ausgeführt werden, können von einem Verarbeitungsprozess auch mehrere unabhängig voneinander ausführbare Operationen hintereinander aufgerufen werden, deren Ergebnisse jeweils durch Callback-Funktionen verfügbar gemacht werden.

Verglichen mit einer synchron aufeinanderfolgenden Ausführung der betreffenden Operationen ergibt sich damit nicht nur ein geringerer Speicherverbrauch, sondern auch eine erheblich kürzere Gesamtbearbeitungszeit, illustriert in folgender Abbildung, in – plakativer – Abgrenzung von der Darstellung in Abbildung „*Brachliegen von Speicherressourcen bei Verwendung von OS Threads*“.

Es sei diesbezüglich aber darauf hingewiesen, dass auch in anderen Ausführungsumgebungen, z. B. in Java, bei Vorliegen unabhängig ausführbarer Operationen eine Reduktion der Gesamtverarbeitungszeit – wenn auch nicht der Speicherbelegung – durchaus auch durch verfügbare Ausdrucksmittel für Threads erzielt werden kann.

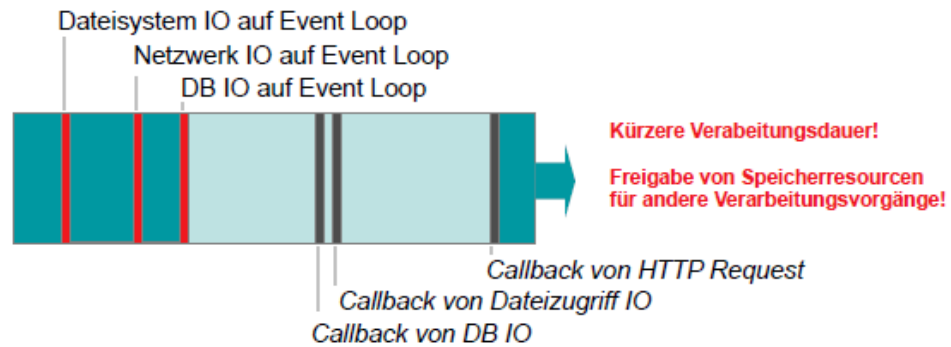


Abb.: Effiziente Nutzung von Speicherressourcen bei Verwendung von NodeJS

Effiziente Nutzung von Speicherressourcen bei Verwendung von NodeJS. Die hellgrün markierten Bereiche entsprechen den orangenen Abschnitten in vorheriger Abbildung „Brachliegen von Speicherressourcen bei Verwendung von OS Threads“, in denen bei Verwendung von OS Threads die durch einen Thread reservierten Ressourcen „brachliegen“. Im Ggs. dazu stehen die Ressourcen bis zum Aufruf und der Ausführung der betreffenden Callbacks – markiert in dunkelgrau – für parallele Verarbeitungsprozesse zur Verfügung.

Quelle: [www. https://www.udemy.com/lectures/understanding-the-nodejs-event-loop-91298](https://www.udemy.com/lectures/understanding-the-nodejs-event-loop-91298)

„Event Loop“

Die Funktionsweise des Event-Loops als zentrales Konzept von NodeJS ist vergleichbar der asynchronen Verarbeitung von Nutzereingaben durch ein User Interface. Wie wir oben gesehen haben, können für dessen Bedienelemente Event-Handler gesetzt werden, deren implementierte Verarbeitungsprozesse bei Auftreten einer Nutzereingabe initiiert werden. Dementsprechend können auf dem Event-Loop mit dem Aufruf asynchroner I/O-Zugriffe und anderer asynchron auszuführender Operationen Callback-Operationen assoziiert werden, die bei Abschluss der Operation aufgerufen werden und den aufrufenden Verarbeitungsvorgang ggf. aus einem Ruhezustand – nicht aber einem genuinen Warte-zustand – „aufwecken“.

In beiden Fällen besteht die Implementierungsleistung des Anwendungsentwicklers darin, die Funktionalität einer Anwendung weitgehend als reaktives Verhalten der Anwendung auf Ereignisse – seien es Nutzereingaben oder das Vorliegen eines Operationsergebnisses im Event-Loop – umzusetzen. Die Herausforderung bei der Verwendung von NodeJS besteht jedoch darin, dass insbesondere jegliche I/O-Funktionalität auf diesem Wege bereitgestellt wird und nicht, wie z. B. in Java, eine synchrone API auf Basis von Operationsaufrufen und Rückgabewerten verfügbar ist. Ggf. kann dies die Verwendung mehrerer ineinander eingebetteter – „verschachtelter“ – Callback-Funktionen erfordern, wie Sie anhand der Implementierungsbeispiele ersehen können, deren verwendete Ausdrucksmittel nachfolgend beschrieben werden.

2.2 Weitere Merkmale von NodeJS

Beim Blick auf NodeJS werden wir sehen, dass hier eine Form der Modularisierung von JavaScript-Komponenten zur Anwendung kommt, die durch das Modularisierungsframework [www RequireJS](#) ermöglicht wird. Dieses steht auch für clientseitig im Browser ausgeführtes JavaScript zur Verfügung und erlaubt es, JavaScript-Skripten Funktionen als „öffentlich“ zu „exportieren“, die dann von den Nutzern dieser Skripte mit einem *Namespace* versehen importiert werden können.

Namespace

In Java dienen Packages zur Angabe von Namespaces für die in einer Anwendung verwendeten Bezeichner, insbesondere für die Namen von Klassen und deren Mitgliedern.

Beispielsweise werden durch die folgende Anweisung die durch das Skript `http2mdb.js` exportierten Funktionen mit dem durch die Variable `hmdb` repräsentierten Namespace assoziiert, d. h. sie sind als Funktionen auf dieser Variable zugreifbar und könnten damit von anderen Funktionen gleichen Namens unterschieden werden, die ggf. durch andere Skripte bereitgestellt werden:



Quellcode

Import von http2mdb.js

```
001 // Import der durch das Skript http2mdb.js exportierten Funktionen
002 var hmdb = require("../njsimpl/http2mdb");
003
004 /* (...) */
005
006 // Aufruf einer exportierten Funktion
007 hmdb.processRequest(req, res);
```

Wenn wir einen Blick auf das im Rahmen des Lehrmaterials bereitgestellte Skript werfen, dann sehen wir, dass lediglich die hier aufgerufene Funktion `processRequest()` exportiert wird und alle anderen Funktionen des Skripts – wie die nachfolgend dargestellte Funktion `doGet()` – privat sind, d. h. sie stehen bei Verwendung von RequireJS nicht für den Aufruf von außen zur Verfügung:




Quellcode

Export einer öffentlichen Funktion

```
001 // Export einer öffentlichen Funktion
002 module.exports = {
003
004   processRequest : function processRequest(req, res) {
005     /* (...) */
006     if (req.method == "GET") {
007       doGet(uri, req, res);
008     }
009     /* (...) */
010   }
011 }
012
013 // private Funktion
014 function doGet(uri, req, res) {
015   /* (...) */
016 }
```

RequireJS stellt also Ausdrucksmittel für die Umsetzung einer modularen Softwarearchitektur in JavaScript bereit, auf die wir an dieser Stelle im Hinblick auf das Verständnis des Beispielcodes aber lediglich verweisen wollen.


Falls Sie bereits mit serverseitigen Technologien wie Java, PHP oder Ruby gearbeitet haben, fragen Sie sich vermutlich auch, inwiefern Ihnen in NodeJS komfortable Ausdrucksmittel zur serverseitigen Generierung von HTML oder anderen Markupssprachen zur Verfügung stehen. Diesbezüglich hatten wir ja eingangs erwähnt, dass die vorliegende Lehrveranstaltung – abgesehen von einer Ausnahme in der nachfolgenden Lerneinheit – auf jegliche Verwendung serverseitiger Markupgenerierung zugunsten einer plattformübergreifend wieder verwendbaren Client-Server-Schnittstelle verzichten wird. Zumindest hingewiesen sei aber darauf, dass auch für NodeJS zahlreiche Lösungen existieren, die es Ihnen erlauben, Nutzerschnittstellen mit serverseitiger Markupgenerierung unter Verwendung eines MVC-Ansatzes umzusetzen, so z. B. das Framework  Sails.js.

2.3 I/O Operationen und Verarbeitung von HTTP Requests

Nachfolgend werden wir unter Bezugnahme auf die Implementierungsbeispiele ausgewählte Aspekte der in NodeJS Server-Anwendung vorstellen. Beachten Sie bitte, dass hier wie in den folgenden Beispielen gewisse Abweichungen und Kürzungen gegenüber der tatsächlichen Implementierung vorgenommen werden, die wesentlich kosmetischen oder anderen darstellungsbedingten Zwecken dienen.

2.4 Webserver Implementierung

Das Skript `webserver.js` zeigt sehr deutlich, wie der bereits beschriebene *Event-Loop* von NodeJS für die Bereitstellung der grundlegenden Funktionalität eines HTTP-Servers verwendet wird. Nach einer Deklaration und Instantiierung verschiedener Variablen sowie der Ermittlung der IP-Adresse, unter der der Server bereitgestellt werden soll, enthält es lediglich zwei Funktionsaufrufe bezüglich des in NodeJS enthaltenen `http` Moduls, mittels derer der Server gestartet wird.

Ausgangspunkt der Implementierung war ursprünglich ein einfaches Tutorial, das auf  <http://jmesnil.net/weblog/2010/11/24/html5-web-application-for-iphone-and-ipad-with-node-js/> eingesehen werden kann.



Quellcode

webserver.js

```
001 // Deklaration von Modulen
002 var http = require("http");
003 /* (...) */
004
005 // Deklaration von Variablen
006 var port = 8380;
007
008 // Ermittlung der IP-Adresse
009 var ip = utils.getIPAddress();
010
011 // Initialisierung des Servers unter Angabe einer Callback-Funktion
012 var server = http.createServer(function(req, res) {
013     // Bearbeitung von HTTP Requests
014     /* (...) */
015 }
016
017 // Starten des Servers unter der angegebenen IP-Adresse und Port
018 server.listen(port, ip);
```

Hier wird zunächst durch den synchronen Aufruf von `createServer()` ein neues Server-Objekt erzeugt. Das Verhalten des Servers bei der Bearbeitung von HTTP-Requests wird dabei mittels der übergebenen Callback-Funktion beschrieben. Durch Aufruf der `listen` Funktion auf dem Server-Objekt wird der Server schließlich für ein durch die angegebenen IP-Adresse und Port identifiziertes TCP Socket gestartet, bezüglich dessen Client-Requests durch den Server bearbeitet werden können. Der Aufruf der `listen()` Methode bildet in der vorliegenden Server-Anwendung den Einstiegspunkt in den NodeJS Event-Loop. So wird, wann immer durch das angegebene TCP Socket ein Client-Request entgegengenommen wird, durch NodeJS ein Thread für die Bearbeitung dieses Requests gestartet und auf diesem Thread die in `createServer()` übergebene Callback-Funktion aufgerufen.


Beachten Sie, dass der Callback-Funktion zwei als `req` und `res` bezeichnete Objekte übergeben werden. Hierbei handelt es sich um eine serverseitige Repräsentation des HTTP-Requests bzw. des HTTP-Response, welcher nach erfolgter Bearbeitung an den Client übermittelt wird. Die durch NodeJS bereitgestellte API dieser Objekte ermöglicht es u. a., auf sämtliche Detailinformationen von Request und Response lesend bzw. schreibend zuzugreifen, insbesondere auf die Daten, die im Body von Request und Response übermittelt werden können. Durch die Verwendung dieser beiden Objekte – und nicht durch die Zuweisung eines reservierten Threads – ist die Zuordnung eines Verarbeitungsvorgangs zu dem spezifischen Paar aus Request und Response gewährleistet, bezüglich dessen die Verarbeitung erfolgt.

Im Verlauf der Request-Bearbeitung werden üblicherweise, wie wir gleich sehen werden, weitere asynchrone Funktionen aufgerufen, z. B. um die im Body des Requests übermittelten Daten auszulesen. Diese Funktionsaufrufe werden durch den Event-Loop verwaltet. Dabei gewährleisten die Objektreferenzen, die in den jeweils übergebenen Callback-Funktionen verwendet werden, dass bei Aufruf des Callbacks aus dem Event-Loop heraus die Verarbeitung bezüglich derselben Objekte erfolgt bzw. fortgesetzt wird, die vor Aufruf der asynchronen Funktion bearbeitet worden waren.

Entscheidend für die „durchgängige Bearbeitung“ beispielsweise eines HTTP-Requests ist in NodeJS also nicht die Assoziation mit einem für die Bearbeitung reservierten Thread, sondern vielmehr der spezifische, durch die Menge referenzierter Objekte gebildete Kontext, bezüglich dessen die Bearbeitung erfolgt. In Bezug auf HTTP-Requests kommt die wichtigste Rolle diesbezüglich den beiden oben als `req` und `res` bezeichneten Objekten zu, die durch dem initialen Callback bei Bearbeitung eines Requests übergeben werden.

Mit der Umsetzung dieses Callbacks in den Implementierungsbeispielen werden wir uns nachfolgend beschäftigen.

2.5 Auslieferung statischer Ressourcen

Als einfachster Fall der Bearbeitung eines HTTP-Requests kann die Auslieferung einer statischen, im Dateisystem des Servers vorliegenden Ressource angesehen werden, die durch die URL des Requests identifiziert wird. Ob dieser Fall vorliegt oder nicht, wird in den Beispielen anhand einer primitiven Überprüfung der URL festgestellt. So wird hier angenommen, dass jeglicher Request, dessen URL Pfad *nicht* mit einem festgelegten Segment – `/http2mdb/` – beginnt, eine statische Ressource identifiziert. Bei dem `url` Objekt, auf dem nachfolgend die `parse()` Funktion aufgerufen wird, handelt es sich um ein NodeJS Modul mit  Utility-Funktionen bezüglich URLs:




Quellcode


Implementierung der Callback-Funktion

```
001 // Implementierung der Callback-Funktion zur Request-Bearbeitung
002 function(req, res) {
003   // ermittle den Pfad der URL im Anschluss an die Domain/IP-Adresse und Port
004   var path = url.parse(req.url).pathname;
005
006   // überprüfe, ob der Pfad mit einem festgelegten Segment beginnt
007   if (path.indexOf("/http2mdb/") == 0) {
008     /* (...) */
009   } else {
010     /* nimm an, dass der Pfad eine Datei bezeichnet, deren Inhalt an
011        den Client übergeben werden soll und lies die Dateiinhalte aus */
012     /* (...) */
013   }
014 }
```

Zugriff auf Dateisystem

Auf die im `else`-Fall durchgeführten Schritte werden wir nachfolgend in Kürze eingehen. Mit der – vermutlich interessanteren – Bearbeitung der durch die Überprüfung „aussortierten“ Requests bezüglich `/http2mdb/` werden wir uns unten im Anschluss an die Darstellung der Ausdrucksmittel von MongoDB beschäftigen.

Für den Zugriff auf das Dateisystem stellt uns NodeJS ebenfalls  ein Modul zur Verfügung, auf dem wir, wie nachfolgend gezeigt, eine durchaus sprechend benannte Methode `readFile()` aufrufen können. Dieser übergeben wir zum einen den Pfad der auszulesenden Datei, zum anderen eine Callback-Funktion, welche nach erfolgtem Auslesen der Dateiinhalte – oder im Fall eines Fehlers – durch den Event-Loop aufgerufen wird.

Bei `__dirname` handelt es sich um eine  globale, in NodeJS verfügbare Variable, die (nicht wirklich überraschend) das Verzeichnis identifiziert, in dem das gerade ausgeführte Skript liegt. Unsere Beispielanwendung ist dabei so aufgebaut, dass alle durch den NodeJSWebserver ausgelieferten Ressourcen in einem Verzeichnis namens `webcontent` liegen, unterhalb dessen die Dateien entsprechend dem durch den HTTP Request identifizierten Pfad zugegriffen werden:



Quellcode

Dateisystemzugriff zur Auslieferung von Dateiinhalten

```
001 // Dateisystemzugriff zur Auslieferung von Dateiinhalten via HTTP Response
002 fs.readFile(__dirname + "/webcontent/" + path, function(err, data) {
003   // setze einen NOT FOUND Fehlercode, falls ein Fehler beim Zugriff auftritt
004   if (err) {
005     res.writeHead(404);
006     res.end();
007   }
008   // übermittle andernfalls die Dateiinhalte im Body des Response
009   else {
010     1 // setze einen OK Status Code und ermittelt den Content-Type
011       des Dateinhalt
012     res.writeHead(200, {
013       "Content-Type" : contentType(path)
014     });
015     res.write(data, 'utf8');
016     res.end();
017   }
018 });
```

Sie sehen hier deutlich, welche Funktionen in NodeJS für die Übermittlung von Daten im HTTP-Response verwendet werden können. So kann mittels `writeHead()` der HTTP *Status Code* sowie eine Menge von Response Headern geschrieben werden, die optional als Attribute eines Objekts – hier als Objektliteral markiert durch `{...}` – angegeben werden. Im vorliegenden Fall wird lediglich der *Content-Type* Header gesetzt, dessen Wert mittels einer von uns implementierten Funktion `contentType()` auf Basis der Dateiendung ermittelt wird. Der Aufruf der `end()` Funktion auf dem Response-Objekt ist obligatorisch. Er zeigt dem NodeJS-Webserver an, dass der Response abgeschlossen ist. Andernfalls wartet der Client „auf ewig“ auf den Abschluss des Response, und der Server betrachtet das Response-Objekt als fortdauernd in Gebrauch.

Verzögerung einer Response-Beendigung

Nur erwähnt sei hier, dass die Verzögerung einer Response-Beendigung im Rahmen der Umsetzung eines als Long Polling bezeichneten Verfahrens auch bewusst eingesetzt werden kann.

Sollen im Body des Response Daten an den Client übermittelt werden, kann dies durch Aufruf der – synchron ausgeführten – Funktion `write()` erfolgen. Es ist dabei nicht notwendig, dass alle Daten, wie im vorliegenden Fall, in einem Schritt übermittelt werden. Beispielsweise könnte bei großen Datenmengen das Auslesen von Dateiinhalten und die Übermittlung im HTTP-Response auch portionsweise vorgenommen werden – durch mehrfache Aufrufe von `write()` und Übergabe der jeweils zu übermittelnden Datenportion.

Auslesen von Dateiinhalten

Für das portionsweise Auslesen von Dateiinhalten kann eine Funktion namens `read()` als Alternative zu `readFile()` verwendet werden.

Nun wissen Sie also bereits, wie mittels NodeJS ein einfacher Webserver entwickelt werden kann, der nach Auslesen einer URL aus einem HTTP-Request zur Übermittlung von Dateiinhalten an einen auf den Server zugreifenden Client in der Lage ist. Auf Grundlage dieser Funktionalität könnten z. B. die in den vorangegangenen Lerneinheiten verwendeten Implementierungsbeispiele durch „unsere eigene“ NodeJS Webserver-Implementierung für den Zugriff aus dem Browser bereitgestellt werden. Beachten Sie aber, dass die hier gezeigte Umsetzung in verschiedener Hinsicht stark vereinfacht ist. So nehmen wir nach der oben gezeigten Fallunterscheidung bezüglich des Startsegments des URL-Pfads an, dass abgesehen von den darin „ausortierten“ Requests jeglicher Request den Zugriff auf einen Dateiinhalt zum Ziel hat. Entsprechend werden Fehler beim Zugriff auch undifferenziert durch den Status Code `404 NOT FOUND` erwidert. Dies ist jedoch nicht nur hinsichtlich der Fehlerursache unscharf, sondern auch hinsichtlich der Tatsache, dass zur Feststellung, ob tatsächlich ein lesender Zugriff auf eine Ressource vorliegt, eigentlich die im Request verwendete *HTTP-Methode* berücksichtigt werden müsste – üblicherweise würde in den vorliegenden Fällen jeweils ein `GET` Request verwendet.

Für eine differenzierte Request-Behandlung und -Erwiderung wäre außerdem ggf. die Berücksichtigung von *Content-Type*-Headern im Request sowie die Anzeige der übermittelten Datenmenge mittels eines *Content-Length*-Headers im Response erforderlich. Nur erwähnt sei an dieser Stelle, dass auch Caching-bezogene Informationen aus dem Request ausgelesen bzw. im Response übermittelt werden könnten.

Die vorliegende Lerneinheit verfolgt jedoch nicht das Ziel, eine solch umfassende Umsetzung eines Webserver entsprechend den im HTTP-Protokoll spezifizierten Ausdrucksmitteln durchzuführen, auch wenn die Möglichkeiten von NodeJS und eine weitergehende Nutzung der [www](#) Dateizugriff API durchaus auch zu diesem Zweck eingesetzt werden könnten. In Bezug auf den Zugriff auf die Inhalte des HTTP Requests, von dem wir bisher lediglich die URL ausgelesen haben, werden wir jedoch im Anschluss an die Darstellung von MongoDB weitere Ausdrucksmittel vorstellen, die ihrerseits die für NodeJS charakteristische Nutzung asynchroner Funktionen mit Callbacks illustrieren werden.

3 MongoDB

Haben Sie schon jemals Bekanntschaft mit dem englischen Wort „*humongous*“ gemacht? Falls dem nicht so ist – aber auch selbst wenn Sie dieses Wort kennen – dürfte die Erschließung der Namensherkunft der hier betrachteten Datenbanktechnologie [www MongoDB](http://www.mongodb.com) eine gewisse Herausforderung darstellen. So ist der Namensbestandteil *mongo* tatsächlich nur ein Teilsegment des erwähnten Worts „*hu-mongo-us*“ – und damit nicht unbedingt intuitiv aus diesem ableitbar.

Die Wortbedeutung – ins Deutsche lässt sich „*humongous*“ mit „riesig“, „gigantisch“ übersetzen – erlaubt es jedoch schon eher, einen – wenn auch nicht gerade bescheidenen – Bezug zu Datenbanken herzustellen. Und genau dieser Anspruch, eine Technologie für die Speicherung potentiell riesiger Datenmengen bereitzustellen, steht hinter der Wortwahl durch die Entwickler der MongoDB Technologie, die diese seit 2007 konzipiert und entwickelt haben – lange bevor der Begriff „Big Data“ mit verschiedenen Assoziationen von E-Commerce bis NSA in aller Munde war.

Zunächst sollte MongoDB – letzteren Vorstellungen durchaus angemessen – die Grundlage für einen „cloud-basierten“ Speicherdienst bilden, welcher entsprechend einem „Platform-as-a-Service“ Geschäftsmodell vermarktet werden sollte.

Definition of Cloud Computing

Siehe für eine präzise Fassung dieses im alltäglichen Gebrauch ebenfalls eher unscharfen Begriffs eine offizielle Definition unter

<http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>.

Seit 2009 ist MongoDB jedoch als Open Source Software mit GNU Lizenz für verschiedene Betriebssysteme verfügbar und mithin auch dezentral installierbar – beispielsweise auf Ihren jeweiligen Entwicklungsrechnern. Eigener Aussage zufolge ist MongoDB derzeit die „*leading NoSQL Database*“ und spätestens hier wird ein weiterer Buzzword-verdächtiger Begriff verwendet, auf den wir im folgenden näher eingehen, bevor wir uns mit den konkreten Ausdrucksmitteln von MongoDB beschäftigen wollen.

<http://www.mongodb.com/leading-nosql-database>).

3.1 NoSQL

Wie wir eingangs erwähnt haben, wird das Kürzel „*NoSQL*“ üblicherweise als „*not only SQL*“ aufgelöst. Zur Erläuterung sei hier in Kürze und vereinfachend dargestellt, was unter „*SQL*“ aus der Sicht der zunehmend zahlreicher werdenden Verfechter von „*NoSQL*“ verstanden wird.

Datenbanken können aufgefasst werden als Speicher für Instanzen komplexer Datentypen, wobei Instanzen untereinander durch Assoziationen verknüpft sein können – bestimmt haben Sie schon von den vier Assoziationstypen *one-to-one*, *one-to-many*, *many-to-one* und *many-to-many* gehört. Ein wesentlicher Unterschied zwischen SQL und NoSQL besteht zum einen darin, welche Einschränkungen der verwendete Datenspeicher bezüglich der konkreten Struktur der zu speichernden Instanzen mit sich bringt.

Ein anderer Unterschied betrifft die Art und Weise, wie die Assoziationen zwischen Instanzen gehandhabt werden. Bezüglich dieser Aspekte sind „SQL“-basierte Datenbanken dadurch gekennzeichnet, dass Instanzen komplexer Datentypen jeweils in eigenen typspezifischen Tabellen repräsentiert werden. Attribute von Datentypen entsprechen den Spalten einer Tabelle, sofern es sich um Attribute mit primitiven Datentypen handelt oder falls ein Attribut auf maximal eine Instanz eines komplexen Datentypen verweist.

Für Verweise werden – als sogenannte Fremdschlüssel – die Identifikatoren verwendet, die eine assoziierte Instanz in ihrer jeweiligen Tabelle eindeutig identifizieren. Falls Verweise bezüglich mehrerer Instanzen vorliegen, werden zu deren Repräsentation Relationstabellen eingesetzt, die die Instanzen der betreffenden Daten zueinander in Beziehung setzen – exemplarisch werden diese Zusammenhänge mit Blick auf das Datenmodell der Implementierungsbeispiele in folgender Abbildung dargestellt.

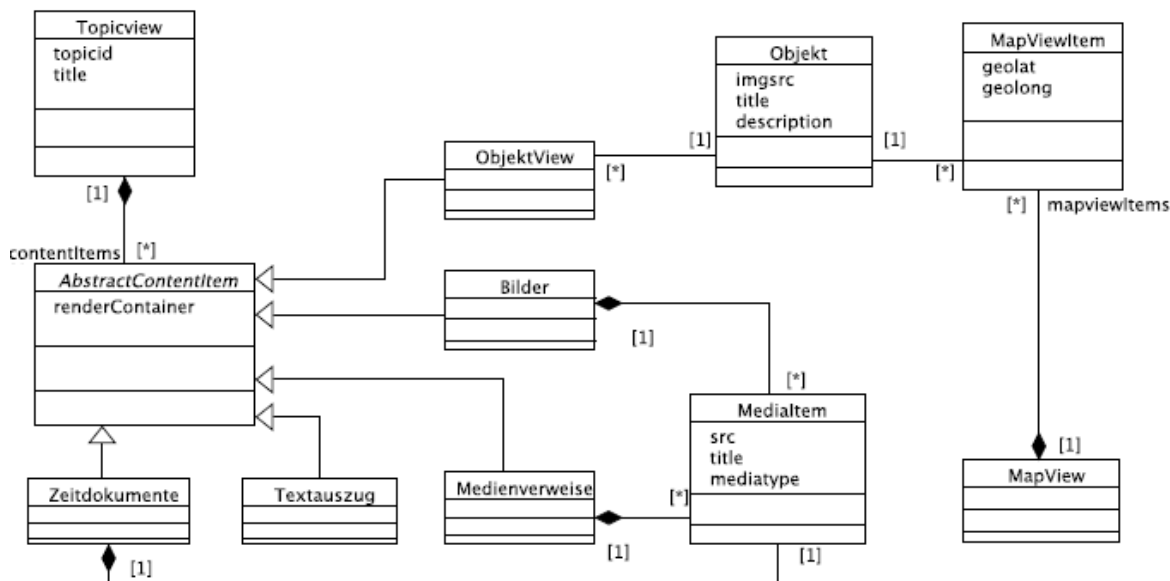


Abb.: Datenmodell der Beispielanwendung

Ein relationales Datenbankschema würde für alle dargestellten Klassen (mit Ausnahme ggf. der Unterklassen von *Abstract-ContentItem*) jeweils eine Tabelle deklarieren und Assoziationen, bei denen ein Assoziationsende mehrfach auftreten kann, z. B. *contentItems*, mittels Relationstabellen abbilden.

Sollen in einem einzigen Datenbankzugriff ggf. aufeinander verweisende Instanzen mehrerer Datentypen identifiziert werden, sind dafür ggf. komplexe Join-Anfragen erforderlich, die sich auf die verschiedenen Tabellen erstrecken, die zur Repräsentation der Instanzen und ihrer Beziehungen verwendet werden.

Vielleicht haben Sie schon mit sogenannten ORM-Technologien gearbeitet, die eine teilweise automatische Abbildung objektorientierter Datenmodelle und relationaler Datenbankschemata ermöglichen, z. B. mit einer Implementierung der *Java Persistence Architecture* wie *Hibernate*. In diesem Fall bleibt die Komplexität solcher Abfragen bei Datenzugriff evtl. vor Ihnen als Entwickler zunächst verborgen. Nicht auszuschließen ist jedoch, dass sie sich in Form von Performancedefiziten bemerkbar macht.

Unweigerlich weisen SQL-basierte Datenbanken eine Eigenschaft auf, die man wahlweise als Grundlage der Wohldefiniertheit und Konsistenz der repräsentierten Daten ansehen, auf die man aber auch eine gewisse Inflexibilität zurückführen kann, die SQLbasierte Datenspeicherung kennzeichnet. So erfordern die meisten Änderungen der Struktur der zu repräsentierenden Daten – evtl. abgesehen von einer Reduktion der Menge der Attribute eines Datentypen – eine entsprechende Modifikation des Datenbankschemas und ggf. eine Anpassung der vorhandenen Datenbestände.

Einem Datenbankschema kommt in SQL auch insofern eine einschränkende Funktion zu, als schreibende Datenzugriffe gegen das Schema verifiziert werden und zur Vermeidung von Inkonsistenzen ein Zugriff ggf. unterbunden wird – so wird beispielsweise die Existenz von Instanzen, die via Fremdschlüssel referenziert werden, grundsätzlich garantiert und die Typsicherheit der zugegriffenen Daten gewährleistet – d. h. „eine Tabelle enthält nur und genau das, was im Schema vorgesehen ist“. Die Gewährleistung der Datenkonsistenz, die man durchaus auch als einen Vorteil von SQL ansehen kann, erfordert es jedoch, dass Datenbestände jeweils „als Komplettpaket“ berücksichtigt werden und als solches zugreifbar sein müssen. Daraus resultiert eine geringere Flexibilität bezüglich der Verteilung von Datenbeständen über mehrere Datenbankserver hinweg und ggf. Aufwände bezüglich der Replikation von Datenbeständen.

NoSQL vs. SQL

Zur Abgrenzung von SQL vs. NoSQL lässt sich, vereinfacht ausgedrückt, eine im Umfeld unserer Veranstaltung durchaus angemessene Analogie herstellen. So verhält sich SQL zu NoSQL mit Blick auf die Typsicherheit bzw. Flexibilität der Typisierung ungefähr so wie Java zu JavaScript. Auch hinsichtlich der Assoziation von Instanzen eines oder verschiedener Datentypen erlauben NoSQL-Ansätze durchweg flexiblere Lösungen – dies jedoch unter Preisgabe der garantierten Konsistenz von Datenbeständen hinsichtlich referentieller Integrität und der Vollständigkeit erforderlicher Attribute oder Assoziationen.

Erwähnt sei aber auch, dass letztere Aspekte mittlerweile durch optionale Erweiterungen bestehender NoSQL-Technologien adressiert werden. So steht für MongoDB mit dem Framework [www.Mongoose](#) beispielsweise eine Lösung zur Verfügung, die einige der aus SQL bekannten Aspekte fixer Datenbankschemata für die an sich flexibel schematisierte Datenhaltung von MongoDB zur Verfügung stellt – was NoSQL Puristen davon halten, wurde bisher durch den Autor noch nicht in Erfahrung gebracht...

So kann denn gerade der grundsätzliche Verzicht auf ein fixes und normatives Datenbankschema als gemeinsame Eigenschaft verschiedener im Umfeld von NoSQL anzusiedelnder Ansätze angesehen werden, die wir nachfolgend in Kürze vorstellen.

Dokumentenorientierte Datenbanken und Objekt Stores

Dokumentenorientierte Datenbanken und „*Object Stores*“ erlauben es, Instanzen komplexer Datentypen komplett – inklusive aller ggf. in ihnen enthaltenen Objekte – in „Tabellen“ bzw. analogen Datenbehältern zu speichern. Eine Verteilung der miteinander assoziierten Instanzen auf ggf. verschiedene Tabellen eines Datenbankschemas ist damit nicht erforderlich. Für das in der vorherigen Abbildung dargestellte Datenmodell der Beispielanwendung könnten z. B. Instanzen von `Topicview` inklusive aller in ihnen enthaltenen `AbstractContentItem`-Objekte und der ggf. in diesen enthaltenen `MediaItem`-Objekte gespeichert werden.

Dies bedeutet eine klare Abgrenzung von SQL, wo für Assoziationen jeglicher Art zwischen Instanzen komplexer Datentypen ein Datensatz pro Instanz ggf. in verschiedenen Tabellen persistiert werden muss. Die Abgrenzung von Objekt Stores und dokumentenorientierten Datenbanken ist dadurch gegeben, dass letztere es erlauben, lesende und schreibende Zugriffe bezüglich beliebiger Teilstrukturen eines ggf. komplexen Objekts durchzuführen und z. B. auch partielle Aktualisierungen durchzuführen. Objekt Stores erfordern hingegen die Identifikation von Objekten mittels eines eindeutigen Identifikators und sind weniger flexibel hinsichtlich zugreifbarer oder abfragbarer Teilstrukturen. Die in unserer Veranstaltung betrachteten Technologien MongoDB und [IndexedDB](#) für server- bzw. client-seitige Datenspeicherung gehören zu diesem Typ von NoSQL-Ansätzen und können als exemplarische Vertreter eines dokumentenorientierten Ansatzes bzw. eines Objekt Stores angesehen werden.

Graphdatenbanken

Graphdatenbanken wie z. B. [www.Neo4J](#) oder [www.RDF](#)-basierte Implementierungen verfolgen einen entgegengesetzten Ansatz, indem sie jegliche Information bezüglich einfacher Attribute einer Datentypinstanz, wie auch bezüglich der Assoziationen zwischen Instanzen als Menge elementarer Aussagen repräsentieren. Solche Aussagen haben die Form `Instanz-ID - Attribut - Attributwert`, wobei `Attributwert` entweder ein Literalwert oder eine `Instanz-ID` ist. Auf Basis einer `Instanz-ID` kann also die Menge der bezüglich der bezeichneten Instanz vorliegenden Aussagen ermittelt und damit eine Instanz hinsichtlich aller Attribute und Assoziationen beschrieben werden.

Abgesehen von der Struktur elementarer Aussagen treffen Graphdatenbanken keine restriktiven Annahmen bezüglich der Menge möglicher Attribute und deren Wertebereich, d. h. sie erlauben in vergleichbar flexibler Weise, wie die vorgenannten Ansätze, eine flexible Typisierung der repräsentierten Daten. Erklärungsbedürftig ist evtl. die Bezeichnung Graphdatenbank. Diese ist dadurch motiviert, dass die Menge der Instanz-IDs und die Menge der elementaren Attributwerte eines Datenbestands als Knoten und die Menge der in elementaren Aussagen verwendeten Attribute bzw. Assoziationen als Kanten eines Graphen aufgefasst werden können, wie es in folgender Abbildung illustriert wird.

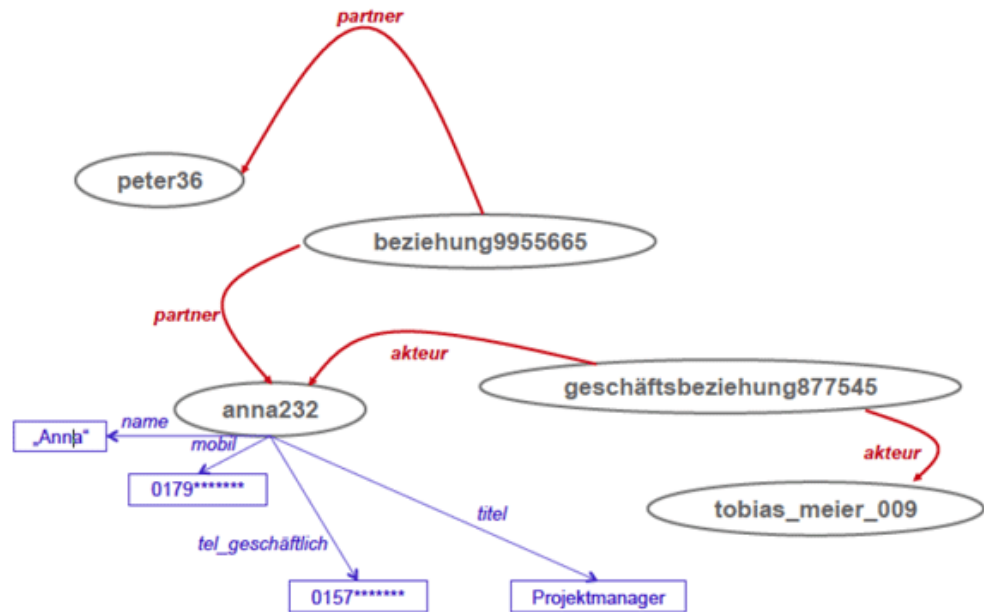


Abb.: Beispiel für die Repräsentation von Daten mittels Graphen auf Grundlage des im W3C Standard RDF verwendeten Graphenmodells

Die grau markierten Knoten des Graphen repräsentieren Instanzen von Datentypen, bezüglich derer elementare Aussagen in Form von Assoziationen mit anderen Instanzen (rot) und Attributzuweisungen (blau) getroffen werden können. Instanzen werden durch – im vorliegenden Fall „sprechende“ – eindeutige Bezeichner identifiziert. Beachten Sie, dass in RDF vergleichbar der Verwendung von Assoziationsklassen in einem objektorientierten Datenmodell Assoziationen, denen ihrerseits Eigenschaften zugewiesen werden sollen, als Instanzen modelliert werden müssen. Beispielsweise könnten den beiden hier als „private Beziehung“ und „Geschäftsbeziehung“ modellierten Individuen Attribute wie beginn, ende, etc. zugewiesen werden.

Key Value Stores

Key Value Stores sehen zunächst nichts weiteres vor als die Möglichkeit, Instanzen einfacher und ggf. komplexer Datentypen ungeachtet ihrer Typzugehörigkeit durch einen eindeutigen Bezeichner – den Key – zugreifbar zu machen. Ein Beispiel hierfür ist z. B. der Key Value Store [www Redis](https://www.redis.io/).

Merkmale von NoSQL

Die bereits oben erwähnte Abwesenheit fixer Datenbankschemata, die allen genannten NoSQL-Ansätzen gemeinsam ist, wie auch der Verzicht auf eine „harte Verdrahtung“ assoziierter Daten und deren „Überwachung“ durch die verwendete Datenbanktechnologie ermöglichen gegenüber SQL nicht nur eine höhere Flexibilität der speicherbaren Daten.

Erhöht wird gegenüber SQL auch die Flexibilität bezüglich des Speicherorts, d. h. assoziierte Daten können über mehrere – evtl. sehr viele... – Datenbankserver verteilt werden und müssen nicht notwendigerweise durch jeden der genannten Server in Gänze zugreifbar sein. Grundlage dafür ist die Möglichkeit einer Partitionierung der Gesamtdatenmenge anhand der jeweils verwendeten Identifikatoren, d. h. für den Zugriff auf einen konkreten Datensatz auf Basis seines Identifikators ist dann nur die Information erforderlich, auf welchem oder welchen Datenbankserver(n) die betreffende Partition verfügbar ist. Erwähnt sei, dass die Konsistenz der Gesamtdatenmenge damit nicht in vergleichbarer Weise garantiert werden kann, wie dies bei Verwendung von SQL möglich ist.

Daraus ergeben sich indes Konsequenzen bezüglich der möglichen Einsatzszenarien für NoSQL-Technologien. So erscheinen letztere für kritische Geschäftsfunktionen, z. B. in Finanzwesen, Logistik oder bezüglich der Haltung von Stammdaten von Kunden, Mitarbeitern oder Produkten, nicht notwendigerweise geeignet – zumindest nicht, falls ein etwaiger Verlust oder die Inkonsistenz von Daten oder deren Nichtverfügbarkeit zu einem gegebenen Zeitpunkt als Schadensfall angesehen werden muss.

3.2 Konzepte von MongoDB und ihre Anwendung

Konzept der Collection

MongoDB ist eine dokumentenorientierte Datenbank, die es u. a. erlaubt, komplexe Objekte mit objektwertigen Attributen in beliebiger Einbettungstiefe / „Verschachtelung“ als Ganzes zu repräsentieren. Ein wesentliches Konzept, das der Datenspeicherung in MongoDB zugrunde liegt, ist dabei das Konzept der *Collection*.

Collections sind benannte Mengen von Objekten, innerhalb welcher jedes Objekt durch einen eindeutigen Bezeichner identifiziert werden kann, der durch MongoDB zugewiesen wird. Eine Collection ist insofern mit einer Tabelle einer relationalen Datenbank vergleichbar, als sie verwendet werden kann, um die Menge aller Instanzen eines Datentyps zu speichern. Die Abgrenzung ist jedoch dadurch gegeben, dass für Collections kein Schema angegeben wird, d. h. eine Collection kümmert sich nicht darum, ob die in ihr gespeicherten Objekte eine bestimmte Struktur haben, wie sie durch ein Schema deklariert werden kann.

Die in einer Collection gespeicherten Objekte können außerdem, wie oben mit Blick auf Dokumentendatenbanken beschrieben, beliebig viele assoziierte Objekte in beliebiger Einbettungstiefe enthalten – vergegenwärtigen Sie sich diesbezüglich noch einmal die Struktur von `Topicview`-Objekten, wie wir sie in der vorangegangenen Lerneinheit für den dynamischen Aufbau von Ansichten verwendet haben. In einer Collection können z. B. jeweils die kompletten Objekte persistiert werden, die in den JSON-Konfigurationsdateien der Aufgaben zu JSR enthalten sind.

Verwendung von Collections

Collections kommt in MongoDB auch insofern eine zentrale Rolle zu, als lesende und schreibende Datenzugriffe immer bezüglich einer Collection durchgeführt werden. Auch wenn MongoDB die Behandlung von Collections als Mengen der Instanzen eines wohldefinierten Datentyps nicht forciert, erscheint es daher mit Blick auf einen klar strukturierten und nachvollziehbaren Aufbau einer Anwendung empfehlenswert, Collections analog zu Tabellen zu verwenden und verschiedenartige Objekte in verschiedenen Collections zu persistieren.

So werden Sie im Rahmen des Übungsprogramms beispielsweise Instanzen des Typs `Objekt` in einer anderen Collection speichern als Instanzen von `Topicview`. Wie Sie in unserer nachfolgenden Betrachtung der NodeJS für MongoDB sehen werden, können Sie in MongoDB recht flexibel auf die Inhalte einer Collection zugreifen – und zwar sowohl lesend, als auch schreibend. So können Abfragebedingungen beispielsweise bezüglich aller Attribute von Objekten formuliert werden, inklusive der Attribute eingebetteter Objekte. Auch erlaubt MongoDB die partielle Aktualisierung von Objekten in einer Collection, inklusive des Zugriffs auf eingebettete Objekte oder Arrays. Im Rahmen der Beispielanwendung ist es z. B. möglich, in der Abfragebedingung einer Update-Operation ein `Topicview` Objekt eindeutig zu identifizieren und dem in diesem Objekt enthaltenen `contentItems`-Array ein neues Element hinzuzufügen.

Assoziationsarten

Was die Vorteile der Verwendung von MongoDB gegenüber SQL basierten Datenbanken angeht, so macht unser Verweis auf das Beispieldatenmodell deutlich, dass Instanzen von Datentypen, die *one-to-one* und *one-to-many* Kompositionsbeziehungen verwenden, sehr einfach mit MongoDB gespeichert werden können. So zeichnet sich eine Komposition ja dadurch aus, dass die auf diese Weise assoziierten Objekte nicht nur „ineinander enthalten“ sind, sondern dass die Lebensdauer eines enthaltenen Objekts an die Lebensdauer des „Behälters“ gebunden ist.

Im gegebenen Datenmodell existieren Instanzen von `AbstractContentItem` und `MediaItem` beispielsweise nie unabhängig von der `Topicview`-Instanz, die die betreffenden Elemente verwendet. Entsprechend reicht hierfür in MongoDB eine einzige Collection aus, um Instanzen des „obersten Behälters“ einer ggf. eingebetteten Komposition zu speichern – in unserem Fall `Topicview`.

Mehr Aufwände bereitet hingegen die Verwendung von Datentypen, die durch „schwächere“ Assoziationen als eine Kompositionsbeziehung aufeinander bezogen sind. In unserem Fall trifft dies für den Datentyp `Objekt` zu, den wir in der vorangegangenen Lerneinheit bereits verwendet haben. Für diesen sieht das Datenmodell vor, dass ein und dieselbe Instanz von `Objekt` in verschiedenen `Topicview`-Instanzen verwendet werden kann – und ggf. auch in Instanzen von `MapViewItem` zur Darstellung des aktuellen oder eines früheren Orts eines Objekts in einer Kartenansicht. Instanzen von `Objekt` müssen also unabhängig von den Instanzen von `Topicview` gespeichert werden können, in welchen die Darstellung eines Objekts via `ObjektView` ggf. eingebunden wird.

Entsprechend unseren Überlegungen zur Verwendung von Collections empfiehlt es sich daher, für `Objekt` eine eigene Collection zu verwenden und die zu darzustellenden Gegenstände in geeigneter Form aus `ObjektView` zu referenzieren. Bei Einsatz einer relationalen Datenbank könnte hierfür auf der zu verwendenden Tabelle für `ObjektView` (bzw. der Tabelle für dessen abstrakte Oberklasse `AbstractContentItem`) ein Fremdschlüssel bezüglich der `Objekt`-Tabelle verwendet werden. In MongoDB hingegen müssen wir eine solche Beziehung zwischen den beiden Tabellen manuell „nachbauen“ und beispielsweise Objekte des Typs `ObjektView` wie folgt repräsentieren:



Quellcode

Objektrepräsentation von `ObjektView`

```
001 // Objektrepräsentation von ObjektView
002 {
003   type: "objektview",
004   render_container: "left",
005   objektid: <ID der referenzierten Objekt Instanz>
006 }
```

Zur Assoziation von Instanzen aus verschiedenen Collections können wir also Attribute verwenden, als deren Wert der durch die betreffende Collection vergebene eindeutige Identifikator – in unserem Fall der Identifikator bezüglich der Collection für `Objekt` – gesetzt wird. Beachten Sie aber, dass diese Verknüpfung im Ggs. zu Fremdschlüsseln insofern „lose“ ist, als MongoDB nicht für uns überprüft, ob das referenzierte Objekt tatsächlich existiert. So ist ja nicht einmal die Tatsache, dass es sich bei dem betreffenden Identifikator um eine ID aus der Collection für `Objekt` handelt, für MongoDB transparent, sondern lediglich eine Konvention, die wir auf Ebene unserer Implementierung des Zugriffs auf MongoDB verwenden. Ggf. müssten bei Entfernung eines Objekts alle `ObjektView`-Instanzen, die auf das zu entfernende Objekt verweisen, ihrerseits entfernt werden – und auch diese Entfernung würde allein in unserer Implementierung vollzogen und würde nicht – wie bei Verwendung von Fremdschlüsseln – durch die zugriffene Datenbank forciert.

Die konkreten Ausdrucksmittel, die Ihnen auf Ebene von NodeJS für die Ausführung der vorstehend beschriebenen lesenden und schreibenden Zugriffe auf Inhalte einer MongoDB zur Verfügung stehen, werden wir im nachfolgenden Abschnitt erläutern.

3.3 Zugriff auf MongoDB in NodeJS

Sie werden nun einen Einblick in die JavaScript API erhalten, die uns für den Zugriff auf MongoDB in NodeJS zur Verfügung steht. Diese wird uns durch das externe NodeJS Modul [www.mongoosejs.com](#) bereitgestellt, das bereits in unseren Implementierungsbeispielen enthalten ist. Wir beschreiben zunächst, wie einfache CRUD-Operationen auf Collections umgesetzt werden können. Danach zeigen wir u. a. mit Blick auf die im Rahmen des Übungsprogramms umzusetzende Funktionalität, wie in MongoDB auf eingebettete Objekte der in einer Collection enthaltenen Objekte zugegriffen werden kann.

Erwähnt sei, dass auf die in einer MongoDB Datenbank enthaltenen Inhalte natürlich auch direkt – und ohne Einsatz von NodeJS – zugegriffen werden kann. Hierfür steht uns die sogenannte Mongo Shell zur Verfügung, bezüglich derer Sie unter <http://docs.mongodb.org/manual/tutorial/getting-started-with-the-mongo-shell/> einführende Hinweise finden.

Im Rahmen der Lehrveranstaltung werden wir auf dieses Werkzeug nicht näher eingehen – angemerkt sei lediglich, dass auch hier eine JavaScript-Syntax zur Formulierung der Zugriffe auf MongoDB verwendet wird.

Die meisten der nachfolgend vorgestellten Ausdrucksmittel werden im Skript `http2mdb.js` der Implementierungsbeispiele zur aktuellen Lerneinheit verwendet.

3.4 Einfache CRUD-Zugriffe

Um auf eine MongoDB-Datenbank zuzugreifen, benötigen Sie zunächst das Modul `mongoosejs`, das nachfolgend mittels `require()` importiert wird. Auf diesem gewährt Ihnen dann die `connect()` Funktion Zugriff auf eine laufende Instanz der Datenbank. Dieser übergeben Sie als erstes Argument die URL der Datenbank, welche die folgende Form hat:

<Rechnername|IP-Adresse>:<MongoDB-Port>/Datenbankname

Falls Sie beim Starten von MongoDB keine spezifischen Einstellungen angeben, wird der Datenbankserver unter `localhost:27017` gestartet, in diesem Fall reicht für den Zugriff via `mongoosejs` die Angabe des Datenbanknamens aus. Sollte eine Datenbank dieses Namens auf dem Server noch nicht existieren, wird sie bei Aufruf von `connect()` erstellt – die Inbetriebnahme von MongoDB und das Zusammenspiel mit NodeJS gestalten sich damit sehr niedrigschwellig.

Die Funktion `connect()` bringt auch bereits Collections als zentrales Konzept von MongoDB ins Spiel. So können Sie ihr als zweites Argument einen Array mit Collectionnamen übergeben, auf die Sie zugreifen wollen. Auch hier gilt, dass Collections, die beim Zugriff noch nicht existieren, dynamisch erstellt werden, ohne dass dafür eine Sonderbehandlung erforderlich wäre. Rückgabewert der – synchron ausgeführten! – `connect()` Funktion ist dann ein Objekt, das für die angeforderten Collections jeweils ein Attribut bereitstellt, welches für den späteren Zugriff auf die Collection verwendet wird:



Quellcode

mongoosejs Modul

```
001 // importiere das mongoosejs Modul
002 var mongoosejs = require("mongoosejs");
003
004 /* stelle eine Verbindung zur Datenbank her und gib die zu verwendenden
005    Collections an */
006 var db = mongoosejs.connect("localhost:27017/iamdb", ["topicviews",
007    "gegenstaende"]);
008
009 // zeige die Objekte, mit denen auf die Collections zugegriffen werden kann
010 console.log("topicviews will be accessed via: " + db.topicviews);
011 console.log("gegenstaende will be accessed via: " + db.gegenstaende);
```

Falls Sie in den von Ihnen ausgeführten Code die beiden hier verwendeten Logmeldungen einfügen, werden Sie lediglich die Meldung bekommen, dass hier jeweils ein JavaScript-Objekt verwendet wird, das nicht weiter dargestellt wird. Der Nutzen dieser Objekte wird jedoch offensichtlich, wenn wir sie verwenden, um lesende und schreibende Zugriffe auf den betreffenden Collections durchzuführen.

create

Wir beginnen mit einem schreibenden Zugriff auf eine der betreffenden Collections und übergeben dieser ein `Topicview`-Objekt der Form, wie wir sie in der vorangegangenen Lerneinheit bereits verwendet haben, das heißt z. B.:



Quellcode

Topicview Objekt

```
001 // Topicview Objekt
002 var topicviewObj = {
003   topicid: "die_umsiedlerin",
004   title: "Die Umsiedlerin",
005   content_items: []
006 };
```

Durch Aufruf der Funktion `save()` können wir dieses Objekt zu einer Collection hinzufügen. Diese Funktion wird asynchron ausgeführt und übergibt im Erfolgs- oder Fehlerfall ihr Resultat an eine Callback-Funktion. Die `mongojs` API orientiert sich hier an anderen I/O-Operationen in NodeJS, wie wir sie selbst auch bereits im Fall von `readFile()` verwendet haben und übergibt dem Callback wahlweise ein Error-Objekt oder das „erwünschte“ Operationsergebnis. Im Fall von `save()` ist dies das der Funktion übergebene Objekt, auf dem bei erfolgreichem Hinzufügen zur Collection ein eindeutiger Identifikator als Wert eines `_id` Attributs gesetzt wird:



Quellcode

Objekt in die Collection schreiben

```
001 //schreibe ein Objekt in die topicviews Collection
002 db.topicviews.save(topicviewObj, function(err, saved) {
003   /* übergeben wird dem Callback entweder ein Fehler oder das um
004     ein _id Attribut erweiterte Objekt */
005   if (err || !saved) {
006     console.error("object could not be saved: " + err);
007     /* (...) */
008   } else {
009     console.log("saved object with id: " + saved._id);
010     /* (...) */
011   }
012 });
```

read

Um ein bestimmtes Objekt oder eine Menge von Objekten aus einer Collection auszulesen, können Sie eine gleichermaßen sprechend benannte Funktion namens `find()` aufrufen. Dieser übergeben wir als erstes Argument die Abfragekriterien, anhand derer das Auslesen erfolgen soll – und für welche in SQL die Ihnen vermutlich vertraute `WHERE` Clause verwendet wird. Abfragebedingungen können in MongoDB bezüglich aller Attribute der auszulesenden Objekte formuliert werden, inklusive bezüglich der Attribute eingebetteter Objekte, wie wir im nachfolgenden Teilkapitel zeigen werden. Abgesehen von exakten Übereinstimmungen von Attributwerten können Bedingungen auch mittels verschiedener Vergleichsoperatoren formuliert werden, die in der MongoDB Dokumentation [www.mongodb.org](#) beschrieben werden.

Das Ergebnis eines `find()` Aufrufs können wir wiederum mittels einer Callback-Funktion entgegennehmen. Nachfolgend sehen Sie einen Fall, in dem wir versuchen, auf ein Objekt einer Collection durch Abgleich des Attributs `topicid` auszulesen, das in unserem Datenmodell und in der `topicviews`-Collection als eindeutiger Identifikator fungiert:




Quellcode

Objekt aus einer Collection auslesen

```
001 // lies ein Objekt aus einer Collection aus
002 db.topicviews.find({
003   topicid : "die_umsiedlerin"
004 }, function(err, elements) {
005   if (err || !elements) {
006     console.error("error reading from topicviews: " + err);
007     /* (...) */
008   } else if (elements.length == 0) {
009     console.error("object with topicid " + topicid + " could
010       not be found.");
011     /* (...) */
012   } else {
013     console.log("found " + elements.length + " elements for
014       topicid " + topicid);
015     /* (...) */
016   }
017 });
```

Vielleicht fragen Sie sich bereits, welche weitere Verwendung – abgesehen von der Ausgabe von Logmeldungen – wir für die Resultate der Datenbankzugriffe im Sinn haben. Hierfür verweisen wir auf den abschließenden Abschnitt dieser Lerneinheit, in dem wir den Zugriff auf MongoDB mit der in NodeJS implementierten Webserver-Funktion aus dem vorangegangenen Abschnitt zusammenführen werden.

Dass „flexible Typisierung“ in JavaScript keineswegs bedeutet, dass der Typ eines Variablenwerts für die Verarbeitung keine Rolle spielt (sondern lediglich, dass u. a. Variablen Werte unterschiedlicher Typen zugewiesen werden können), wird uns auch durch die JavaScript API für MongoDB deutlich vor Augen geführt. So führt MongoDB für den eindeutigen Identifikator eines Objekts den speziellen Typ  `ObjectId` ein, der auch in einer Abfragebedingung bezüglich des `_id` Attributs verwendet werden muss. Wird anstelle von `ObjectId` z. B. eine String-Repräsentation des Identifikators verwendet, kann das betreffende Objekt nicht gefunden werden.

Nun ist es durchaus möglich, dass die `_id` beim Zugriff auf MongoDB in Form eines Strings vorliegt – z. B. wenn die Id als Segment einer URL übermittelt wird, wie wir unten sehen werden. Für solche Fälle können wir aber auf `mongojs` eine Funktion namens `ObjectId` – beachten Sie die korrekte Schreibweise! – aufrufen, die einen String in die erforderliche Objektrepräsentation umwandelt:

```
001 //wir an, dass die _id eines Objekts als String vorliegt, z. B.:
002 var objectid = "5304e3954f2375a806000001";
003
004 // wandle den String in einen Wert des Typs ObjectId um
005 var convertedid = mongojs.ObjectId(objectid);
006
007 // ... und führe damit den Zugriff durch:
008 db.topicviews.find({
009   _id : convertedid
010 }, callback);
```

delete

Für das Löschen von Objekten aus einer Collection können die zu löschenden Objekte wie für den Zugriff via `find()` durch Abfragebedingungen identifiziert werden. Der Callback-Funktion wird hier im Erfolgsfall allerdings nur die Anzahl der tatsächlich gelöschten Objekte übermittelt. Im folgenden Beispiel sollte dies maximal ein Objekt sein, da wir als Abfragebedingung die `_id` als eindeutigen Identifikator verwenden:



Quellcode

Objekt aus einer Collection entfernen

```
001 // entferne ein Objekt aus einer Collection
002 db.topicviews.remove({
003   _id : convertedid
004 }, function(err, update) {
005   if (err || !update || update < 1) {
006     console.log("topicview with id " + convertedid + " could
007       not be deleted. Got: " + err);
008     /* (...) */
009   } else {
010     console.log("topicview with id " + convertedid + " was
011       deleted. Got: " + update);
012     /* (...) */
013   }
014 });
```

Sollen tatsächlich alle Objekte entfernt werden, die eine gegebene Bedingung erfüllen, muss dies allerdings mittels eines weiteren Arguments zum Ausdruck gebracht werden.

```
// entferne alle Objekte, auf die condition zutrifft
db.topicviews.remove(condition, {multi: true}, callback);
```

update

Die Aktualisierung von Objekten ist in `mongojs` insofern mit dem Löschen vergleichbar, als auch im Erfolgsfall dem übergebenen Callback die Anzahl der modifizierten Objekte übergeben wird. Sollen alle Objekte geändert werden, auf die die angegebene Abfragebedingung zutrifft, ist auch hier die Angabe von `{multi: true}` erforderlich. Was die Angabe der zu aktualisierenden Inhalte angeht, so erlaubt es MongoDB, partielle Updates auf den ausgewählten Objekten durchzuführen, d. h. der `update()` Funktion müssen lediglich die tatsächlich zu aktualisierenden Attribute übergeben werden.

Die Art des Updates kann in MongoDB durch Aktualisierungsoperatoren angegeben werden, z. B. wird für das Überschreiben bzw. Ergänzen von Attributen z. B. der `$set` Operator verwendet. Nachfolgend nehmen wir an, dass der Wert der Variable `update` ein Objekt mit einer beliebigen Anzahl von Attributen ist, die auf dem in der Abfragebedingung identifizierten Objekt gesetzt werden:



Quellcode

Objekt in einer Collection updaten

```
001 /* füge die in update enthaltenen Attribute einem durch topicid
002   identifizierten Objekt hinzu */
003 db.topicviews.update({
004   topicid : topicid
005 }, {
006   $set : update
007 }, function(err, updated) {
008   if (err || !updated || updated < 1) {
009     console.error("topicview with id " + convertedid + "
010       could not be updated: " + err);
011     /* (...) */
012   } else {
013     console.log("topicview with id " + convertedid + " has
014       been updated. Got: " + updated);
015     /* (...) */
016   }
017 });
```

Vielleicht wollen Sie aber – im Ggs. zum vorstehenden Beispiel – die Situation, dass ein zu aktualisierendes Objekt noch nicht existiert, aber gar nicht als Fehlerfall, sondern als „Variante des Normalfalls“ betrachten und in diesem Fall das Objekt mit den übergebenen Daten einfach erzeugen. Dafür können Sie der `update()` Funktion als optionales drittes Argument die Rahmenbedingung `{upsert: true}` übergeben:

```
001 // aktualisiere ein vorhandenes Objekt oder verwende update zur Erstellung
002 eines neuen Objekts
003 db.topicviews.update(condition, {$set: update}, {upsert: true}, callback);
```


Erwähnt sei außerdem, dass MongoDB neben dem Setzen bzw. Überschreiben von Attributen noch zahlreiche weitere Aktualisierungsoperationen vorsieht, z. B. kann die Inkrementierung eines Zahlenwerts mittels des Operators `$inc` oder das Löschen eines Attributs mittels `$unset` durchgeführt werden. Auch können in einem Aufruf von `update()` mehrere Update-Operationen gleichzeitig veranlasst werden. Dafür kann das Update-Objekt, für welches wir oben lediglich ein `$set`-Attribut angegeben haben, um weitere „Operator-Attribute“ erweitert werden. Eine [vollständige Übersicht](#) der in MongoDB verwendbaren Operator-Attribute finden Sie in der Dokumentation.

3.5 Zugriffe auf eingebettete Objekte

Als Abrundung unserer Darstellung der Ausdrucksmittel von MongoDB, von denen wir vermittelt durch die `mongojs` API in NodeJS Gebrauch machen, zeigen wir hier anhand verschiedener Beispiele, wie der Zugriff auf eingebettete Objekte der in einer Collection enthaltenen Objekte durchgeführt werden kann. Auch hier beziehen wir uns wieder auf das den Implementierungsbeispielen zugrunde liegende Datenmodell für `Topicview` als einen exemplarischen Fall für eine solch komplexe Objektstruktur. Wir konzentrieren uns insbesondere auf die Verwendung eingebetteter Arrays als eine Datenstruktur, deren Verwendung bei einer strukturierten Repräsentation von Objekten in einem SQL-Schema die Nutzung von Relationstabellen erforderlich machen würde.

Hinzufügen

Es wurde bereits erwähnt, dass für die Darstellung eines `Objekt-Elements` in einem `Topicview` ein `Objektview-Element` dem `contentItems`-Array des `Topicview` hinzugefügt werden muss. Hierfür können wir in MongoDB den Update-Operator `$push` verwenden, dem wir entsprechend der nachfolgend gezeigten Notation den Namen des zu erweiternden Arrays sowie das hinzuzufügende Element angeben:



Quellcode

newitem dem content_items Array hinzufügen

```
001 // füge newitem dem content_items Array eines Topicview hinzu
002 db.topicviews.update(condition, {
003   $push : {
004     content_items : newitem
005   }
006 }, callback);
```

Ist das zu aktualisierende Array-Attribut noch weiter in das mittels `condition` identifizierte Objekt eingebettet, dann kann der Attributpfad als String angegeben werden. Wäre `contentItems` z. B. auf einem Objekt gesetzt, das als Wert eines Attributs `content` auftritt, könnte im obigen Beispiel anstelle von `content_items` der Pfad `content.content_items` verwendet werden.

Entfernen

Das Entfernen von Objekten aus einem eingebetteten Array erfolgt analog zur Hinzufügung unter Verwendung eines `$pull` Operators. Diesem wird ein Objekt übergeben, das die Abfragebedingungen bezüglich der aus dem Array zu löschenden Objekte repräsentiert.

Im nachfolgenden Beispiel liegen also zwei Abfragebedingungen vor. So identifiziert `condition1` eine Menge von Objekten und `condition2` eine Menge von Elementen im `content_items` Array eines durch `condition1` identifizierten Objekts. Unter Angabe von `{multi: true}` veranlasst die `update()` Operation also, dass aus den `content_items` Arrays der durch `condition1` identifizierten Objekte jeweils die durch `condition2` identifizierten Elemente entfernt werden:



Quellcode

Elemente aus dem content_items Array entfernen

```
001 // entferne Elemente des Typs objektview aus dem content_items Array der
002 // durch condition1 identifizieren Objekte
003 var condition2 = {type: "objektview"};
004
005 db.topicviews.update(condition1, {
006   $pull : {
007     content_items : condition2
008   }
009 }, callback);
```


Identifizieren

Im verwendeten Datenmodell wäre eine solche Entfernung von Elementen des Typs `ObjektView` aus `contentItems` z. B. für den Fall erforderlich, dass ein Objekt `Objekt` gelöscht wird – so nimmt unser Datenmodell ja an, dass `Objekt`-Instanzen unabhängig von `Topicview` existieren und ggf. in mehr als einem `Topicview` verwendet werden. In diesem Fall wollen wir die Entfernung für alle diejenigen `Topicview`-Instanzen durchführen, deren `contentItems`-Array ein `ObjektView`-Element bezüglich der zu löschenden oder bereits gelöschten – in SQL wäre diese Reihenfolge nicht beliebig! – `Objekt`-Instanz enthält.

Diese `Topicview`-Instanzen können wir aber nicht anhand ihrer eindeutigen Identifikatoren identifizieren, sondern anhand der Tatsache, dass in ihnen ein entsprechendes `ObjektView`-Element enthalten ist. Es ist also erforderlich, dass wir innerhalb einer Abfragebedingung auf Arrays Bezug nehmen können, die in die Objekte einer Collection eingebettet sind. Auch diese Anforderung wird in der JavaScript API für MongoDB durch die Bereitstellung eines durch `$` gekennzeichneten Operators – `$elemMatch` – erfüllt:



Quellcode

Referenzierte Elemente aus content_items entfernen

```
001 // entferne alle Elemente aus content_items, die eine Referenz auf das
002 // durch deletedid identifizierte Objekt enthalten
003 var condition = {objektid: deletedid};
004
005 db.topicviews.update({
006   content_items : {$elemMatch: condition}
007 }, {
008   $pull : {
009     "content_items" : condition
010   }
011 }, { multi: true },
012   callback
013 });
```

Aus dem Blickwinkel der Anforderungen der Beispielanwendung und des Übungsprogramms haben wir bisher exemplarische Ausdrucksmittel von NodeJS und MongoDB anhand von Codebeispielen dargestellt. Offen blieb bisher aber die Frage, wie die im vorliegenden Abschnitt eingeführten Zugriffsfunktionen auf MongoDB clientseitig initiiert und dann durch den bereits vorgestellten NodeJS Server aufgerufen werden können. Dieses Bindeglied zwischen den clientseitig und serverseitig ausgeführten Komponenten unserer Anwendung werden wir nun im abschließenden Teilkapitel der Lerneinheit einführen.

4 Datenzugriffsoperationen via HTTP

Wie bereits erwähnt, geht dieser Abschnitt der Frage nach, in welcher Form wir die Datenzugriffsoperationen auf der serverseitig laufenden MongoDB von unserer im Browser ausgeführten mobilen Anwendung aufrufen können. Da wir aus clientseitigem JavaScript nicht direkt über eine Netzwerkverbindung auf den MongoDB Datenbankserver zugreifen können – und dies aus architektonischen Erwägungen heraus auch nicht wollen! – werden wir dafür die bereits ansatzweise entwickelte NodeJS Server-Anwendung verwenden. Mit dieser kann der Browser, wie bereits gezeigt, durch Übermittlung von HTTP-Requests kommunizieren.

Konkret werden wir `XMLHttpRequest` verwenden, da uns der Server in seinen Erwiderungen ja lediglich die „rohen“ Ergebnisse der Zugriffsoperationen, z. B. in Form von JSON-Objekten, zur Verfügung stellen soll, die wir dann in jeweils geeigneter Weise für den Aufbau oder die Manipulation des User Interfaces unserer Anwendung verwenden werden – möglicherweise wollen wir z. B. den Erfolg oder Fehlschlag eines durch den Nutzer gewünschten schreibenden Datenzugriffs auch durch geeignete Überlagerungselemente, wie *Popup-Dialoge* oder *Toasts*, kommunizieren.

CRUD in HTTP

Wie aber können wir die Ausdrucksmittel von HTTP in geeigneter Form einsetzen, um mittels ihrer die durch NodeJS auf MongoDB aufzurufenden lesenden und schreibenden Zugriffsoperationen zu identifizieren?

Bisher hatten wir lediglich `GET`-Requests für den lesenden Zugriff auf serverseitig vorliegende statische Inhalte genutzt, deren Dateisystempfad durch eine URL identifiziert wurde. Lassen Sie uns nun daher an dieser Stelle noch einmal vergegenwärtigen, welche weiteren Ausdrucksmittel uns in HTTP zur Verfügung stehen und wie wir diese für den hier beschriebenen Zweck zum Einsatz bringen können. Von besonderem Interesse hierfür ist das in der HTTP-Spezifikation verwendete Konzept einer serverseitig vorliegenden Ressource, deren Identifikation eine URL als *Uniform Resource Locator* dient, sowie durch die in der Spezifikation vorgesehenen Methoden, von denen wir bisher ja nur `GET` verwendet haben. Gemäß der Spezifikation wird durch die in einem Request verwendete Methode zum Ausdruck gebracht, in welcher Form der Zugriff auf die durch die URL identifizierte Ressource erfolgt.

Ziel des vorliegenden Abschnitts ist es nun, geeignete Verbindungen von URLs und HTTP Methoden zu erarbeiten, anhand derer die durch den Server auszuführenden Datenzugriffe – inklusive der partiellen Zugriffe auf eingebettete Objekte eines Objekts – identifiziert werden können, und außerdem zu klären, in welcher Form Argumente und Rückgabewerte des Datenzugriffs übermittelt werden.

Grundsätzlich liegt die „Interpretation“ eines HTTP-Requests zwar vollkommen in den Händen des Entwicklers, der den HTTP-Server implementiert. Bereits die Namen der spezifizierten Methoden geben uns aber zumindest einen Hinweis darauf, wie die entsprechende Behandlung von Requests erfolgen könnte. So wird beispielsweise ein Request mit `GET`-Methode üblicherweise als Anfrage bezüglich einer „Auslieferung“ der Ressource an den Client aufgefasst. Den Server-Entwickler hindert jedoch grundsätzlich nichts daran, die zugriffene Ressource vor der Auslieferung zu löschen, sodass ein erneuter `GET`-Zugriff fehlschlagen würde. Dass dieser Fall jedoch nicht konform mit der HTTP-Spezifikation ist, zeigt ein Blick auf das Konzept der Idempotenz, das dort zur weiteren Charakterisierung von HTTP-Methoden verwendet wird.

Idempotenz

Idempotenz bezüglich einer HTTP-Methode besagt, dass sich das Resultat einer mehrfachen Ausführung eines Requests nicht von dem einer einfachen Ausführung des Request unterscheiden soll.



Definition

Idempotente Funktion

Eine Funktion f ist *idempotent* genau dann, wenn für alle x im Definitionsbereich von f gilt:
 $f(f(x)) = f(x)$.

Zweifelsohne wird dieses Kriterium durch die erwähnte Umsetzung eines `GET`-Requests als Löschen der zugegriffenen Ressource nicht erfüllt, da nach einmaliger Bearbeitung des Requests gar kein Ressourceninhalt mehr ausgeliefert werden kann. Da `GET` jedoch durch die HTTP-Spezifikation als idempotente Methode definiert wird, steht diese Implementierung im Widerspruch zu den Regeln der Spezifikation und könnte damit als ungültig ausgeschlossen werden.

HTTP Methoden

Welche Methoden sind nun aber in der HTTP-Spezifikation vorgesehen? Welche Annahmen werden bezüglich deren Idempotenz getroffen? Und welche Anhaltspunkte ergeben sich daraus für die Identifikation von CRUD Datenzugriffsoperationen via HTTP?

Wenn wir dafür die Übersicht in folgender Tabelle betrachten, fallen neben `GET` insbesondere die folgenden drei Methoden ins Auge:

- `PUT`: Übergabe von Inhalten an eine Ressource
- `DELETE`: Löschen des Inhalts einer Ressource
- `POST`: Übergabe von Inhalten an eine Ressource

Methode	Semantik	idempotent?
<code>GET</code>	Zugriff auf den Inhalt einer Ressource ohne weitere Seiteneffekte	ja
<code>HEAD</code>	Bloße Abfrage der Metadaten einer Ressource (Response enthält keine Nutzdaten)	ja
<code>PUT</code>	Übergabe von Inhalten an eine Ressource	ja
<code>DELETE</code>	Löschen des Inhalts einer Ressource	ja
<code>POST</code>	Übergabe von Inhalten an eine Ressource	nein
<code>OPTIONS</code>	Abfrage der verfügbaren Methoden bezüglich einer Ressource	(ja)
<code>TRACE</code>	Übermittlung des beim Server angekommenen Requests (zur Detektion etwaiger Manipulationen)	(ja)
<code>CONNECT</code>	Initiiert eine gesicherte SSL-Verbindung über einen in der Ressource bezeichneten Proxy-Server	-
<code>PATCH</code>	Bewirkt eine partielle Modifikation des Inhalts der bezeichneten Ressource (Vorschlag, siehe http://tools.ietf.org/html/rfc5789)	ja

Tab.: Übersicht über HTTP-Methoden

Offensichtlich scheint neben `GET` für den lesenden Zugriff auf eine Ressource die `DELETE` Methode sehr gut geeignet, um das Löschen eines Ressourceninhalts zu veranlassen. Mit Blick auf die verbleibenden CRUD-Operationen dienen die beiden Operationen *create* und *update* aber gleichermaßen dem schreibenden Zugriff durch Übermittlung von Inhalten, und die Methoden `PUT` und `POST` stimmen ihrerseits hinsichtlich des Wortlauts ihrer Funktionsbeschreibung überein. Damit stellt sich aber die Frage wie die Unterscheidung zwischen der Erzeugung einer neuen Instanz eines Datentypen (*create*) von der Aktualisierung einer bestehenden Instanz (*update*) auf Ebene von HTTP-Requests abgebildet werden soll.

Auch bezüglich dieser Frage liefert das Konzept der Idempotenz einen eindeutigen Hinweis. So ist `PUT` im Ggs. zu `POST` als idempotent spezifiziert, d. h. die wiederholte Ausführung eines `PUT`-Requests soll keinen anderen Effekt haben als die einmalige Ausführung. Wenn wir die Funktionsweise eines Updates – inklusive des oben mit Blick auf MongoDB erwähnten „Upserts“ – betrachten, dann scheint diese durchaus dieser Vorstellung von `PUT` als idempotent zu entsprechen. So verändert ein wiederholtes Update unter Übergabe derselben Daten nichts an dem Zustand, der aus der einmaligen Ausführung des Updates resultiert.

Für die Erzeugung neuer Objekte mittels `create` gilt dies jedoch nicht: eine wiederholte Ausführung z. B. der `save()` Funktion von MongoDB oder die wiederholte Ausführung eines `insert` auf einer SQL-Datenbank wird mit jeder Ausführung ein neues Objekt bzw. einen neuen Datensatz in einer Collection bzw. Tabelle erzeugen. Im Ggs. zu `update` ist die `create`-Operation also nicht idempotent.

Betrachten wir nun wieder die Spezifikation, dann sehen wir, dass selbiges für die `POST`-Methode gilt. Damit ist zumindest hinsichtlich der zu verwendenden HTTP-Methoden eindeutig, wie die Zuordnung zu den vier CRUD-Operationen erfolgen soll – wir stellen nachfolgend zusätzlich die jeweils auszuführenden Funktionen von MongoDB dar:

CRUD	HTTP	MongoDB
<code>create</code>	POST	<code>save()</code>
<code>read</code>	GET	<code>find()</code>
<code>update</code>	PUT	<code>update()</code>
<code>delete</code>	DELETE	<code>delete()</code>

Tab.: Funktionen von MongoDB

Für das Ziel dieses Abschnitts, nämlich die tatsächliche implementatorische Umsetzung von clientseitig initiierten CRUD-Zugriffen, ist damit bereits eine grobe Struktur vorgegeben. So sollte der Client für die Ausführung einer `create`-Operation einen `XMLHttpRequest` mit `POST`-Methode an den Server übermitteln, und dieser sollte bei Eintreffen eines `POST`-Requests die `save()` - Funktion auf MongoDB ausführen. Mit Blick auf letztere stellt sich dann aber die Frage, auf welche Collection zugegriffen werden soll, welche Daten in die Collection geschrieben werden sollen und welche Daten dem Client als Rückgabewert übermittelt werden sollen.

URLs und Ressourcen

Zur weiteren Bearbeitung der geschilderten Problematik wollen wir das Konzept der Ressource in HTTP betrachten und überlegen, wie wir dieses auf die Inhalte beziehen können, auf welche in unserer Anwendung lesend und schreibend zugegriffen werden soll. Was den Ressourcen-Begriff angeht, so sollte dieser losgelöst von einer bestimmten „Verkörperung“ einer Ressource, z. B. als einer Datei im Dateisystem, betrachtet werden, auch wenn die bisher mittels `GET` zugegriffenen und durch unseren NodeJS-Server ausgelieferten Ressourcen allesamt auf diese Weise verkörpert waren. Ein verallgemeinerter Ressourcenbegriff ist jedoch durchaus auch auf die folgenden Inhalte anwendbar, die uns hier besonders interessieren, z. B.:

- Die Menge aller `Topicview`-Elemente
- Ein spezifisches Element aus dieser Menge
- Die Menge der `AbstractContentItem`-Elemente, die in einem spezifischen `Topicview` enthalten ist.

In gleicher Weise, wie das von der HTTP verwendete Konzept der Ressource abstrakt ist, können wir URLs als *abstrakte Identifikatoren* von Ressourcen begreifen, die insbesondere agnostisch gegenüber der Verkörperung der betreffenden Ressourcen als Dateisysteminhalt, Datenbankinhalt o. ä. sind. Diese ermöglichen als Uniform Resource Locators zugleich den Zugriff auf die bezeichneten Ressourcen.

Dementsprechend erscheinen die folgenden URL-Pfade geeignet, um die vorstehend genannten Ressourcen zu identifizieren – wir lassen hier das initiale Segment aus, das den konkreten Server identifiziert, über welchen der Ressourcenzugriff erfolgt:

- `/topicviews`
- `/topicviews/5304e3954f2375a806000001`
- `/topicviews/5304e3954f2375a806000001/content_items`

CRUD mit HTTP

Wir haben also bereits ermittelt, welche HTTP-Methoden zur Identifikation der CRUD-Datenzugriffsoperationen verwendet werden sollen und wie die zugegriffenen Inhalte mittels URLs identifiziert werden können. Vor der Implementierung des hier erarbeiteten Konzepts müssen wir also nur noch festlegen, in welcher Form die von den Operationen zu verwendenden Argumente und Rückgabewerte übermittelt werden sollen. Diesbezüglich sei darauf hingewiesen, dass z. B. für die Ausführung von lesenden und schreibenden Zugriffen gar keine weiteren Daten erforderlich sind, da die Zugriffe allein auf Grundlage der Kombination von HTTP-

Methode und URL durchgeführt werden können – überzeugen Sie sich selbst:

- `GET /topicviews`
- `GET /topicviews/5304e3954f2375a806000001`
- `DELETE /topicviews/5304e3954f2375a806000001`

Für den gezielten lesenden und löschenden Zugriff auf existierende Ressourcen reichen also URLs der vorgeschlagenen Form aus, deren letztes Segment für den Zugriff auf ein einzelnes Element einer Menge der eindeutige Identifikator dieses Elements ist. Wird kein Identifikator angegeben, dann kann ein `GET`-Zugriff als Zugriff auf die Gesamtmenge der Elemente angesehen werden. Soll als Alternative zu einem Zugriff auf alle vs. ein einzelnes Element hingegen eine Teilmenge von Elementen identifiziert werden, ist die Verwendung von Query-Parametern denkbar, die ebenfalls ein möglicher Bestandteil von URLs sind. So könnte die Entfernung von Elementen eines bestimmten Typs aus dem eingebetteten `content_items`-Array eines identifizierten Topicview z. B. wie folgt initiiert werden:

- `DELETE /topicviews/5304e3954f2375a806000001/content_items/?type=objektview`

Was die Rückgabewerte der betrachteten Operationen angeht, so kann hier für `GET` der Inhalt der zugegriffenen Ressource in geeigneter Repräsentation verwendet werden, z. B. als JSON-Stringserialisierung. Für `DELETE` reicht andererseits ggf. ein `boolean` Wert aus, um anzuzeigen, ob ein `DELETE`-Zugriff erfolgreich war oder nicht. Ob entsprechend der MongoDB-API die Anzahl der ggf. gelöschten Objekte übermittelt werden sollte – bzw. für die Ausführung von `update` mittels `PUT` die Anzahl der modifizierten Objekte – sei hier dahingestellt. Denkbar wäre auch eine Unterscheidung mittels der Verwendung von HTTP Status Codes, z. B. `200` für den Erfolgsfall und ein geeigneter Fehlercode für den Fall, dass das Löschen fehlgeschlagen ist. Ob jedoch z. B. eine aufgrund der Nichtexistenz eines referenzierten Objekts effektlose Löschaktion als Fehlschlag betrachtet werden sollte, kann nur im Einzelfall mit Blick auf die konkreten Anforderungen einer Anwendung beurteilt werden.

So gilt generell für alle hier betrachteten Zugriffsoperationen, dass die an den Client übermittelten HTTP-Responses durchweg den Status Code `200` verwenden können, es sei denn es tritt serverseitig ein Fehler auf oder eine referenzierte Ressource kann aufgrund von Nichtvorhandensein nicht ausgeliefert werden – im letzteren Fall erscheint der Fehlercode `404 NOT FOUND` sehr naheliegend.

Da eine Menge von Elementen als Ressource aufgefasst und für ein hinzuzufügendes neuen Elements der Menge nicht von der Existenz eines eindeutigen Identifikators ausgegangen werden kann, ergibt sich für den Aufbau eines entsprechenden HTTP-Requests zur Realisierung einer `create`-Operation mittels `POST` die nachfolgend dargestellte Form. Die Notation `{content}` bringt hier zum Ausdruck, dass die Inhalte des zu erstellenden Elements im Body des Requests übertragen werden:

- `POST /topicviews {content}`

Die Aktualisierung eines Elements erfordert hingegen das Vorliegen eines Identifikators, entsprechend wird hierfür ein URL-Segment verwendet und die Inhalte des ggf. partiellen Updates werden ihrerseits im Body des Requests übertragen:

- `PUT /topicviews/5304e3954f2375a806000001 {content}`

Rückgabewert ist im ersteren Fall wahlweise nur der dem Element zugewiesene eindeutige Identifikator oder das gesamte Objekt. Für die `update`-Operation kann, wie bereits erwähnt, ein `boolean` Wert ausreichend erscheinen. Die Ergebnisse des hier erarbeiteten Aufbaus von HTTP Requests zur Identifikation von CRUD-Operationen finden Sie für Zugriffe auf nicht-eingebettete Objekte noch einmal zusammenfassend und verallgemeinert dargestellt.

CRUD	Methode	URL	Request Body	Response Body
<i>create</i>	POST	/<collection-name>	{content}	{content+ID}
<i>read (all)</i>	GET	/<collection-name>		{content}
<i>read (single)</i>	GET	/<collection-name>/<element-id>		{content}
<i>update</i>	PUT	/<collection-name>/<element-id>	{update}	boolean
<i>update</i>	DELETE	/<collection-name>/<element-id>		boolean

Generalisierte Darstellung des Aufbaus von HTTP-Requests und -Responses, mit denen die Datenzugriffsoperationen bezüglich der Elemente einer Collection – im Sinne von MongoDB – realisiert werden können.

Für die Gestaltung des Zugriffs auf eingebettete Objekte, auf den wir vorstehend eingegangen sind, halten wir eine fallweise Gestaltung mit Blick auf die konkret durch eine Client-Anwendung zu nutzende Funktionalität für zielführender als eine Generalisierung – so müssten z. B. je nach Funktionsumfang verschiedene Ausdrucksmittel, die uns MongoDB für den Zugriff auf eingebettete Strukturen zur Verfügung stellt, auf der Ebene von URLs „nachgebildet“ werden.

REST

Bevor wir uns nun abschließend mit der client- und insbesondere serverseitigen Umsetzung des vorliegenden Konzepts beschäftigen, sei jedoch noch darauf hingewiesen, dass eine durch einen Server bereitgestellte Datenzugriffsschnittstelle, die zur Identifikation der durchzuführenden Operationen in hohem Maße von den Ausdrucksmitteln des HTTP-Protokolls Gebrauch macht, darauf hinweist, dass die Client-Server-Architektur entsprechend den Kriterien eines sogenannten RESTful Architekturstils gestaltet wurde.

So wird denn eine unseren Vorschlägen entsprechende Schnittstelle üblicherweise auch als „REST-Schnittstelle bezeichnet“. Ohne an dieser Stelle auf zu viele Details einzugehen sei jedoch darauf hingewiesen, dass der Aufbau von URLs und HTTP-Requests einen eher oberflächlichen Aspekt einer RESTful Architektur darstellt. So zielt REST (Representational State Transfer) denn insbesondere darauf ab, die Interaktion zwischen Client und Server zustandslos zu realisieren. Insbesondere ist damit gemeint, dass die Bearbeitung einer vorliegenden Client-Anfrage durch einen Server nicht davon abhängig sein sollte, ob der betreffende Server bereits die vorangegangene Anfrage des betreffenden Clients bearbeitet hat oder nicht – wird dieses Prinzip berücksichtigt, können zu bearbeitende Requests sehr flexibel und ressourceneffizient den Servern einer „Server Farm“ zugewiesen werden, wie sie von den großen Spielern der aktuellen Web-Generation verwendet werden. Dabei kann das Prinzip der Zustandslosigkeit aber z. B. auch dadurch eingelöst werden, dass ein „Zustand“, z. B. ein über eine Folge von Requests angereicherter „Warenkorb“, als Ressource modelliert und durch eine URL oder ein URL-Segment identifizierbar gemacht wird.

4.1 CRUD HTTP Requests und Responses in NodeJS

Hier werden Sie als Abschluss dieser stark auf serverseitige Verarbeitung fokussierten Lerneinheit einen Einblick in diejenigen Funktionen der NodeJS Server-Anwendung erhalten, die über die bloße Auslieferung von statischen Ressourcen hinausgehen und die Funktionalität der vorstehend erarbeiteten HTTP-Schnittstelle für CRUD-Operationen umsetzen. Den Einstiegspunkt in diese Funktionalität haben wir im Abschnitt zu NodeJS bereits vorweggenommen. Er liegt vor in der Fallunterscheidung bezüglich des initialen Segments des URL-Pfads, die wir zu Beginn des Callbacks vornehmen, welcher für die Bearbeitung von HTTP-Requests durch den HTTP-Server von NodeJS aufgerufen wird. Nachfolgend geben wir diese noch einmal wieder:



Quellcode

Callback-Funktion zur Request-Bearbeitung

```
001 // Implementierung der Callback-Funktion zur Request-Bearbeitung
002 function(req, res) {
003   // ermittle den Pfad der URL im Anschluss an die Domain/IP-Adresse und
004   // Port
005   var path = url.parse(req.url).pathname;
006
007   // überprüfe, ob der Pfad mit einem festgelegten Segment beginnt
008   if (path.indexOf("/http2mdb/") == 0) {
009     // rufe die Implementierung der HTTP CRUD Schnittstelle auf
010     hmdb.processRequest(req, res);
011   } else {
012     // nimm an, dass der Pfad eine Datei bezeichnet, deren Inhalt an
013     // den Client übergeben werden soll und lies die Dateiinhalte aus
014     /* (...) */
015   }
016 }
```

Initiales Pfadsegment

Führt ein Client einen HTTP-Request durch, dessen Pfad mit dem hier überprüften Segment beginnt, dann wird hier die Bearbeitung des Requests an die `processRequest()`-Methode delegiert, die in den Implementierungsbeispielen durch das Skript `http2mdb.js` bereitgestellt wird. Beachten Sie, dass diese Weiterleitung allein auf Grundlage der URL erfolgt und z. B. keine Überprüfung der im Request verwendeten Methode durchgeführt wird. Angenommen, unser Server ist unter den verwendeten Namen bzw. IP-Adressen und Ports erreichbar, kann z. B. mit URLs wie den folgenden der Aufruf von `processRequest()` veranlasst werden:

- `http://192.168.177.93:8380/http2mdb/topicviews/die_umsiedlerin`
- `http://localhost:8080/http2mdb/objects/5304e3954f2375a806000001/`

Durch Verwendung des URL-Segments `/http2mdb/` kann ein Client also anzeigen, dass er die durch den Server bereitgestellte HTTP-Schnittstelle für CRUD-Operationen nutzen möchte. Beachten Sie dabei, dass diese Annahme bereits zu unserer spezifischen Implementierung der oben vorgestellten Schnittstelle gehört und dass sie allen Clients bekannt sein muss, welche auf die Schnittstelle zugreifen, was z. B. im Rahmen der Dokumentation der Schnittstelle erfolgen kann.

Wie bereits erwähnt orientieren sich auch alle anderen Strukturmerkmale der zu verwendenden HTTP-Requests zwar an den Konventionen eines REST-Architekturstils. In ihrer konkreten Ausprägung stellen sie aber eine spezifische Eigenschaft unserer spezifischen HTTP-Schnittstelle für CRUD-Operationen dar. Die Merkmale, die erforderlich sind um die Assoziation von HTTP-Requests und Operationsaufrufen zu beschreiben, sind aber zum einen überschaubar, zum anderen in ihrem Umfang beschränkt. Im Wesentlichen bestehen sie, wie in der [Tabelle „Generalisierte Darstellung des Aufbaus von HTTP-Requests und Responses“](#) im vorherigen Abschnitt gezeigt, in einer Assoziation von Funktionsaufrufen mit einer Kombination von HTTP-Requests und URLs bzw. URL-Mustern sowie in der Festlegung, in welcher Form die Funktionsargumente übergeben werden.

Dementsprechend existieren für zahlreiche Ausführungsumgebungen wiederverwendbare Softwarekomponenten – zumeist als „REST-Frameworks“ bezeichnet – die eine solche Assoziation mittels weniger Zeilen Programmcode zu beschreiben erlauben. Für NodeJS wird diese Funktionalität z. B. durch das Framework [Express](#) bereitgestellt, für Java existieren verschiedene Implementierungen der [JAX-RS-API](#), welche es erlauben, die genannten Assoziationen mittels Annotationen auszudrücken.

Überprüfung der Request Methode

Wie Sie wissen, versucht unsere Veranstaltung einige der Mechanismen aufzuzeigen, die „unterhalb der Haube“ von Frameworks verwendet werden. In diesem Sinne verwenden wir für die Umsetzung unserer CRUD-HTTP-Schnittstelle kein Framework, sondern setzen die – grundsätzlich durch ein Framework verallgemeinerbare – Abbildung von HTTP-Requests und CRUD-Operationsaufrufen mit den „Bordmitteln“ um, die uns hierfür in JavaScript und NodeJS zur Verfügung stehen. Dafür betrachten wir nun die Implementierung der Funktion `processRequest()`, die im oben gezeigten Codefragment aufgerufen wird. Wie Sie nachfolgend sehen können, nimmt diese eine Fallunterscheidung auf Grundlage der im HTTP-Request verwendeten Methode vor und ruft abhängig davon Funktionen auf, die für die Behandlung von Requests der betreffenden Methoden zuständig sind.

Benennung der Funktionen

Es sei angemerkt, dass die Benennung dieser Funktionen nach dem Muster `do<Methodenname>` durch die Java Servlet API inspiriert ist, die für die Klasse [HttpServletRequest](#) Methoden gleichen Namens und gleicher Signatur deklariert.

An diese reichen wir die uns im initialen Callback übergebenen Objekte für HTTP-Request und Response weiter sowie den Pfadbestandteil der URL, der entsprechend unserer Überlegungen oben für die Bearbeitung der Requests berücksichtigt werden muss:



Quellcode

Objekte für HTTP Request und Response

```
001 function processRequest(req, res) {
002   // ermittle den URL Pfad im Anschluss an das http2mdb
003   // Schlüsselsegment
004   var uri = utils.substringAfter(req.url, "/http2mdb");
005
006   // rufe in Abhängigkeit von der verwendeten Request Methode eine
007   // entsprechende Bearbeitungsfunktion auf:
008   if (req.method == "GET") {
009     doGet(uri, req, res);
010   } else if (req.method == "POST") {
011     doPost(uri, req, res);
012   } else if (req.method == "PUT") {
013     doPut(uri, req, res);
014   } else if (req.method == "DELETE") {
015     doDelete(uri, req, res);
016   } else {
017     // andere Methoden werden nicht unterstützt...
018     res.writeHead(405);
019     res.end();
020   }
021 }
```

Abgleich der URL

In den hier aufgerufenen Funktionen `doGet()`, `doPost()` etc. werfen wir schließlich einen Blick auf den übergeben URL-Pfad und ermitteln auf dieser Grundlage, welche CRUD-Operation aufgerufen werden soll. Damit wird eine wesentliche Teilfunktion unserer Implementierung erfüllt, nämlich die Zuordnung der im Request verwendeten Methode und URLs zu spezifischen CRUD-Funktionsaufrufen. Hier wie im Folgenden verwenden wir einfache Funktionen wie `startsWith()` und `substringAfter()` zum Abgleich bzw. zur Segmentierung des URL-Strings. An deren Stelle könnte alternativ auch ein Abgleich z. B. mittels regulärer Ausdrücke vorgenommen werden.

Nachfolgend zeigen wir zwei Funktionen, die den Fall eines lesenden bzw. schreibenden Zugriffs exemplifizieren. Für den Fall, dass die verwendete URL keinem der unserer Implementierung bekannten Muster entspricht – und mithin die zuzugreifende Ressource nicht identifiziert werden kann – bringen wir dies durch aussagefähige Responses mit Status Code 404 NOT FOUND zum Ausdruck. Ist die URL bekannt, wird auch hier das für die vorgenommene Fallunterscheidung verwendete URL-Segment „abgeschnitten“ und der Rest der URL an die CRUD-Funktion übergeben. Wir arbeiten uns so gewissermaßen „Segment-für-Segment“ zum „Ziel“ des HTTP-Requests vor.

Für `doGet()` erlaubt uns die gezeigte Verwendung von `substringAfter()`, den Fall eines Zugriffs auf alle Elemente der betreffenden Collections vom Zugriff auf ein einzelnes Element zu unterscheiden – so wird bei korrekter Verwendung der HTTP-Schnittstelle durch den Client im ersteren Fall ein leerer String an `readTopicview()` bzw. `readObject()` übergeben, im letzteren Fall der Identifikator des betreffenden Elements. Beachten Sie außerdem, dass die nachfolgend gezeigten Codeausschnitte bereits Bestandteile des Übungsprogramms vorwegnehmen:




Quellcode

Bearbeitung von GET und PUT Requests

```
001 // Bearbeitung von GET Requests
002 function doGet(uri, req, res) {
003   if (utils.startsWith(uri, "/topicviews")) {
004     readTopicview(utils.substringAfter(uri, "/topicviews/"), req, res);
005   } else if (utils.startsWith(uri, "/objects")) {
006     readObject(utils.substringAfter(uri, "/objects/"), req, res);
007   } else {
008     res.writeHead(404);
009     res.end();
010   }
011 }
012
013 // Bearbeitung von PUT Requests
014 function doPut(uri, req, res) {
015   if (utils.startsWith(uri, "/topicviews")) {
016     updateTopicview(utils.substringAfter(uri, "/topicviews/"), req, res);
017   } else if (utils.startsWith(uri, "/objects")) {
018     updateObject(utils.substringAfter(uri, "/objects/"), req, res);
019   } else {
020     res.writeHead(404);
021     res.end();
022   }
023 }
```

Zugriff auf den Request Body

Die einzige noch verbleibende Maßnahme, die wir vor dem jeweiligen Zugriff auf die CRUD-API von MongoDB nun ggf. noch umsetzen müssen ist das Auslesen der im Body der HTTP-Requests übermittelten Daten, die ein zu erstellendes Objekt bzw. die für die Aktualisierung eines Objekts erforderlichen Attribute übermitteln. Hierfür stellt uns die HTTP-API von MongoDB eine sehr unspezifisch benannte Funktion namens `on()` zur Verfügung, die asynchron ausgeführt wird und mit der wir auf verschiedene Phasen bzw. Ereignisse beim Zugriff auf den HTTP-Request-Body reagieren können – sie sehen hier wieder sehr deutlich die Analogie, die zwischen der Funktionsweise und Handhabung des NodeJS Event-Loops und der Reaktion auf clientseitige Interaktionsereignisse besteht.

Von Interesse für uns sind diesbezüglich die beiden Ereignisse `data` und `end` bezüglich derer wir nachfolgend jeweils eine Callback-Funktion deklarieren. Oben hatten wir bereits darauf hingewiesen, dass die `write()` Funktion bezüglich des HTTP-Response Objekts in NodeJS die „portionsweise“ Übermittlung des Response Body erlaubt. Umgekehrt erfolgt auch das Auslesen von Daten aus dem Body eines HTTP-Requests in „Portionen“ – im Englischen wird hierfür üblicherweise der Begriff des  „chunk“ verwendet.

Diesbezüglich wird die Callback-Funktion bezüglich `data` aufgerufen, wann immer eine „Portion“ Daten aus dem Body ausgelesen wurde. Wenn wir also wissen, dass uns die Daten vom Client als String übergeben werden, können wir diese Funktion verwenden, um die jeweils ausgelesenen Daten an den Wert einer String-Variable – `alldata` im nachfolgenden Beispiel – hinzuzufügen. Den Abschluss eines Auslesevorgangs zeigt uns NodeJS dann durch Aufruf der Callback-Funktion bezüglich des `end`-Ereignisses an, d. h. hier können wir auf die `alldata`-Variable zugreifen und die gewünschte Weiterverarbeitung der Daten implementieren:



Quellcode

alldata Variable

```
001 function updateTopicview(uri, req, res) {
002   // deklariere eine Variable, deren Wert die ausgelesenen Datenportionen
003   // hinzugefügt werden
004   var alldata = "";
005   req.on("data", function(data) {
006     alldata += data;
007   });
008
009   // reagiere auf das Ende des Auslesevorgangs
010   req.on("end", function() {
011     // verwende die in alldata vorliegenden Daten
012     /* (...) */
013   });
014 }
```

Bei kleineren Datenmengen erfolgt das Auslesen von Daten in NodeJS in einem Schritt, d. h. die Callback-Funktion für `data` wird nur einmal aufgerufen. Für diese Fälle wäre die Verwendung zweier Callbacks bezüglich `data` und `end` nicht notwendigerweise erforderlich. Würden jedoch größere Datenmengen übertragen, würde diese Lösung unweigerlich einen Fehler hervorrufen, da z. B. ein beliebiger ausgelesener Teilstring eines im Request übermittelten JSON-Objekts nur zufälligerweise erfolgreich geparkt werden kann und in jedem Fall nicht die Gesamtheit der durch den Client übergebenen und durch den Server zu verarbeitenden Daten repräsentiert.

Die hier gezeigte Verwendung zweier Callbacks bezüglich `data` und `end` ist daher die durch die NodeJS-Dokumentation als kanonisch empfohlene Umsetzung eines lesenden Zugriffs auf die Daten eines HTTP-Request Body. Für den Fall, dass wir in der vorstehenden Funktionen ein Update bezüglich eines durch `uri` identifizierten `Topicview`-Elements durchführen wollen, liegen uns nun also alle Daten vor, die wir für die erfolgreiche Ausführung der `update()`-Funktion von MongoDB benötigen. So repräsentiert der Wert von `uri` den eindeutigen Identifikator des Objekts und `alldata` enthält die zu aktualisierenden Attribute. Wir brauchen nun also lediglich noch den aus dem URL-String ausgelesenen Identifikator in die erforderliche `ObjectId`-Repräsentation und den `alldata`-String in ein JSON-Objekt zu überführen und können damit das Update veranlassen:



Quellcode

alldata String in ein JSON-Objekt überführen

```
001 function updateTopicview(uri, req, res) {
002   var alldata = "";
003   /* (...) */
004
005   // reagiere auf das Ende des Auslesevorgangs
006   req.on("end", function() {
007     // verwende die in alldata vorliegenden Daten
008     db.topicviews.update({
009       _id : mongoose.ObjectId(objectid)
010     }, {
011       $set : JSON.parse(data)
012     },
013     // behandle den Erfolgs- bzw. Fehlerfall
014     function(err, updated) {
015       if (err || !updated) {
016         respondError(res);
017       } else {
018         respondSuccess(res, true);
019       }
020     })
021   });
022 }
```

Die beiden Methoden `respondError()` und `respondSuccess`, die die Callback- Funktion für eine fehlerhafte bzw. erfolgreiche Durchführung von `update()` aufruft, verwenden ihrerseits, wie nachfolgend gezeigt, die Ausdrucksmittel für den Zugriff auf HTTP-Responses, die wir bereits im einleitenden Abschnitt zu NodeJS kennengelernt haben. Vereinfachend und auf die Funktionalität unserer Anwendung beschränkt, nehmen wir hier für den Erfolgsfall nur die Unterscheidung zwischen der Übermittlung von JSON-Objekten und anderen Daten – z. B. `boolean` Werten – vor, und übermitteln im Fehlerfall den Status-Code 500 für „*Internal Server Error*“, falls der Funktion kein spezifischer Status-Code übergeben wird:



Quellcode

Übermittle Status Code

```
001 // übermittle Status Code 200 und ggf. Daten und Header
002 function respondSuccess(res, data) {
003     if (data) {
004         var header = {};
005         if (data instanceof Object) {
006             data = JSON.stringify(data);
007             header["Content-Type"] = "application/json";
008         }
009         res.writeHead(200, header);
010         res.write(data);
011     } else {
012         res.writeHead(200);
013     }
014     res.end();
015 }
016
017 // übermittle einen Fehlercode
018 function respondError(res, code) {
019     res.writeHead( code ? code : 500 );
020     res.end();
021 }
```

Wenn Sie auf eingebettete Objekte eines Collection-Elements zugreifen wollen, können dafür geeignete URLs konzipiert werden, die Sie in den Zugriffsfunktionen analysieren, wie in `updateTopicview()` im folgenden Beispiel.

Dort nehmen wir an, dass für den direkten Zugriff auf einen in einen `Topicview` eingebetteten `content_items`-Array URL-Pfade der Form `/topicviews/<topicview-id>/content_items` verwendet werden. Damit kann beispielsweise innerhalb von `updateTopicview()` das letzte Segment der `uri` überprüft werden und ggf. der eindeutige Identifikator aus dem ersten Segment ausgelesen werden:



Quellcode

Update mit eingebetteten Objekten

```
001 // Update mit Berücksichtigung von eingebetteten Objekten
002 function updateTopicview(uri, req, res) {
003     var alldata = "";
004     /* (...) */
005
006     req.on("end", function() {
007         // überprüfe anhand des letzte URL-Segments, ob ein "eingebettetes"
008         // Update vorliegt
009         if (utils.endsWith(uri, "/content_items")) {
010             // ermittle die id auf Grundlage des ersten Segments
011             var id = mongojs.ObjectId(uri.split("/")[0]);
012             // führe das Update auf content_items durch
013             db.topicviews.update({
014                 _id : id
015             }, {
016                 $push : {
017                     "content_items" : alldata
018                 }
019             }, callback);
020         }
021         // führe andernfalls ein "normales" Update durch
022         else {
023             /* (...) */
024         }
025     });
026 }
```

Wie bereits im vorangegangenen Abschnitt erwähnt, verwendet die hier vorgestellte HTTP-Schnittstelle für den Zugriff auf eingebettete Objekte von Collection-Elementen im Gegensatz zu den grundlegenden CRUD-Operationen bezüglich dieser Objekte einen stärker „pragmatisch“ denn „puristisch“ gekennzeichneten Ansatz, der sich an den konkreten Anforderungen unserer Anwendung orientiert. Dementsprechend besteht auch für die serverseitige Umsetzung der Zugriffsoperationen ein Spielraum, der weit über die Möglichkeiten des vorstehenden Beispiels hinausreicht.

Implementierungsalternativen

Entsprechend der hier gezeigten Vorgehensweise können wir also NodeJS verwenden, um HTTP-Requests als Aufrufe von CRUD-Operationen zu interpretieren und die Request-Bearbeitung als Aufruf der betreffenden Operationen auf MongoDB umzusetzen. In vergleichbarer Weise könnte die betreffende Funktionalität auch für andere server-seitige Laufzeitumgebungen und Datenhaltungstechnologien realisiert werden. In Java könnte z. B. auf annähernd analoge Weise ein Servlet mit Zugriff auf eine MySQL-Datenbank via *JDBC* oder unter Verwendung einer JPA-Implementierung wie [www.wj Hibernate](#) verwendet werden.

Unsere konkrete Umsetzung der CRUD-Operationen mittels NodeJS und MongoDB weist insofern besondere Merkmale auf, als sie in hohem Maße durch die Verwendung asynchroner Funktionsaufrufe und die Einbettung von Callback-Funktionen ineinander gekennzeichnet ist. Deutlich wird dies insbesondere anhand der oben gezeigten Varianten der Funktion `updateTopicview()`, die jeweils einen Callback bezüglich der `update()`-Funktion von `mongojs` in die Callback-Funktion einbetten, mit der wir auf das Auslesen des HTTP-Request-Body reagieren. Bedenken Sie aber auch, dass der Aufruf von `updateTopicview()` seinerseits aus der Callback-Funktion heraus erfolgt, die wir beim Starten unseres HTTP-Servers an NodeJS übergeben haben.

Client-seitiger Zugriff

Vor dem Client, der einen HTTP-Request initiiert, bleiben jegliche Details bezüglich dessen serverseitiger Verarbeitung verborgen. So unterscheiden sich die Requests, die wir auf Seiten unserer Client-Anwendung für den Zugriff auf die hier behandelten CRUD-Operationen verwenden, nur in Details von den Requests, mit denen wir in der vorangegangenen Lerneinheit unter Aufruf der von uns bereit gestellten Utility-Funktion `xhr()` einen `XMLHttpRequest` an den damals verwendeten Webserver initiiert haben. Diese betreffen die HTTP-Methode und die URL, mit denen der Request ausgeführt werden soll, sowie die Frage, ob und welche Daten im Request Body übertragen bzw. aus dem Response ausgelesen werden sollen. Nachfolgend seien daher lediglich exemplarisch Zugriffsmöglichkeiten auf die hier spezifisch behandelten *read*- und *update*-Operationen dargestellt:



Quellcode

Topicview Element auslesen und hinzufügen

```
001 // lies ein Topicview Element aus
002 function readTopicview(id) {
003     xhr("GET", "http2mdb/topicviews/" + id, null, function(xmlhttp) {
004         // parse den Response Body
005         var json = JSON.parse(xmlhttp.responseText);
006         // verwende das Topicview Element
007         /* (...) */
008     },
009     // Fehlerbehandlung
010     function(xmlhttp) {
011         if (xmlhttp.status == 404) {
012             /* (...) */
013         }
014         else {
015             /* (...) */
016         }
017     });
018 }
019
020 // füge für einen Topicview ein Element zu content_items hinzu
021 function addContentItem(topicviewid, item) {
022     xhr("PUT", "http2mdb/topicviews/" + topicviewid + "/content_items",
023     item, function(xmlhttp) {
024         /* (...) */
025     },
026     // Fehlerbehandlung
027     function(xmlhttp) {
028         /* (...) */
029     });
030 }
```

An dieser Stelle endet die aktuelle Lerneinheit und beginnt Ihr Übungsprogramm. In diesem werden Sie den hier in verschiedenen Etappen dargestellten Zugriffsweg für CRUD-Operationen für Objekte des Typs `Objekt` nachbauen bzw. in Erweiterung der vorhandenen Implementierung umsetzen. Sie beginnen damit mit Funktionsaufrufen auf der Client-Seite, in denen Sie „hartkodierte“ `Objekt`-Instanzen via `XMLHttpRequest` zur Persistierung in MongoDB an den NodeJS-Server übermitteln bzw. auf existierende Objekte modifizierend oder zum Zweck der Entfernung eines ausgewählten Objekts zugreifen. Die nachfolgende Lerneinheit wird uns dann mit den Ausdrucksmitteln ausstatten, die wir benötigen, um einem beliebigen Nutzer die Erstellung und Modifikation solcher Objekte anhand von Attributwerten seiner Wahl zu ermöglichen. Die Ebene der client- und serverseitigen Model-Komponenten einer Anwendung, auf der wir uns hier ausschließlich bewegt haben, werden wir dort anhand der Beschäftigung mit Formularen um die View- und Controller-Komponenten erweitern, die wir für eine Nutzerschnittstelle mit CRUD-Funktionen benötigen.

Zusammenfassung

- NodeJS ist eine Ausführungsumgebung für JavaScript, die unabhängig von einem Browser lauffähig ist und kann z. B. für die Umsetzung serverseitiger Komponenten von Webanwendungen eingesetzt werden.
- Eines der wichtigsten Leistungsmerkmale eines Servers stellt neben seiner Stabilität und Verfügbarkeit die Fähigkeit dar, gleichzeitig eine möglichst große Anzahl von Client-Requests mit minimaler Latenzzeit – d. h. möglichst verzögerungsfrei – zu bearbeiten und mit einem Response zu erwidern.
- In NodeJS werden diese Anforderungen durch die Funktionsweise des Event-Loops und die asynchrone Verarbeitung von I/O-Operationen unterstützt.
- Ein wesentlicher Unterschied zwischen SQL und NoSQL besteht darin, welche Einschränkungen der verwendete Datenspeicher bezüglich der konkreten Struktur der zu speichernden Instanzen mit sich bringt und wie die Assoziationen zwischen Instanzen gehandhabt werden.
- Die im HTTP-Protokoll vorgesehenen Methoden `POST`, `GET`, `PUT` und `DELETE` sind zur Identifikation von CRUD-Operationen auf serverseitigen Datenbeständen sehr gut geeignet.

Sie sind am Ende dieser Lerneinheit angelangt. Auf den folgenden Seiten finden Sie noch Übungen.

Übungen

Die Implementierungsbeispiele für die vorliegende Lerneinheit finden Sie im Projekt `org.dieschnittstelle.iam.njm_frm_mfm`. Dieses verwenden wir auch in den folgenden beiden Lerneinheiten zu Formularen. Für den hier vermittelten Stoff und die Bearbeitung der Übungsaufgaben sind die beiden JavaScript-Implementierungen `TopicviewController.js` sowie `TopicviewCRUDOperations.js` sowie das Markup in `topicview.html` bis einschließlich des `<footer>` Elements relevant.

Mit den anderen Inhalten des HTML-Dokuments sowie dem Skript `EditviewController.js` werden wir uns im weiteren Verlauf der Veranstaltung beschäftigen.

Die prüfungsverbindlichen Übungen und deren Bepunktung werden durch die jeweiligen Lehrenden festgelegt.



Programmieren

Übung NJM-00

Vorbereitende Maßnahmen

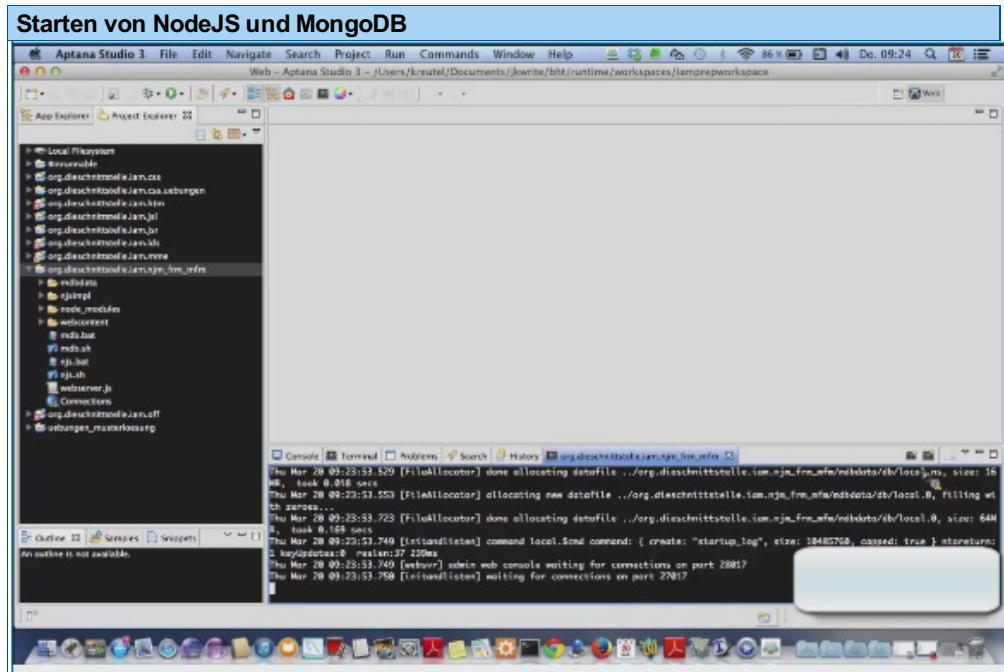
- Laden Sie die NodeJS Binaries aus dem Moodlekurs und importieren Sie diese in Eclipse. Es wird ein Projekt `Binrunnable` erstellt. Falls Sie Linux nutzen, dann laden Sie NodeJS von <http://nodejs.org/download/> herunter.
- Laden Sie von www.mongodb.org/downloads die für Ihre Plattform geeignete Version von MongoDB des Production Releases herunter und entpacken Sie die Archivdatei in Ihr Projekt `Binrunnable`.
- Überprüfen Sie das für Ihre Plattform zu nutzende Skript `mongodb.sh/mdb.bat`. Dieses sollte das Executable `mongod` aus der MongoDB-Installation im Projekt `Binrunnable` referenzieren. Nehmen Sie hier ggf. eine Anpassung vor.
- Führen Sie zunächst das Skript `mongodb` im Beispielpunkt aus – dafür können Sie die Konsolenansicht von Aptana verwenden. Damit wird der MongoDB-Server gestartet.
- Führen Sie dann das `njs` aus. Dieses startet den NodeJS-Webserver.
- Öffnen Sie die URL, unter der Ihre Webanwendung in NodeJS gestartet wird, im Browser. Ihnen sollte eine Liste mit Werkstiteln angezeigt werden.
- Wählen Sie einen Titel aus. Ihnen sollte dann eine „leere Gesamtansicht“ für den betreffenden Titel angezeigt werden.
- Falls Sie einen Fehler bekommen, dann ersetzen Sie im Skript `webserver.js` in den Implementierungsbeispielen die `ip` Variable im Aufruf von `server.listen(ip,port)` durch den String `127.0.0.1` und starten Sie `njs` neu.

Bearbeitungszeit: 30 Minuten

Das folgende Video zeigt Ihnen die einzelnen Schritte zur Übung NJM-00



Film



© Beuth Hochschule Berlin - Dauer: 03:14 Min. - Streaming Media 6 MB



Übung NJM-01

Beispielanwendung

Schauen Sie sich noch einmal das „Big Picture“ der Interaktion zwischen den Komponenten der Beispielanwendung in [Abbildung „Überblick über die Zusammenführung von XMLHttpRequest“](#) an. Versuchen Sie, den kompletten Durchlauf eines Aufrufs einer CRUD Methode nachzuvollziehen, beginnend mit der Ausführung einer Aktion durch den Nutzer auf Ebene der graphischen Nutzeroberfläche. Schauen Sie sich dafür die nachfolgend genannten Skripte und Funktionen an.

1. `webcontent/js/controller/TopicviewViewController.js` **und**
`TopicviewCRUDOperationsRemote.js: createTopicview(),`
`readTopicview(), updateTopicview(), deleteTopicview()`

Wann wird die Ansicht aktualisiert? Unmittelbar nach der Nutzereingabe oder erst, wenn die Aktion erfolgreich auf Serverseite durchgeführt wurde?

2. `webserver.js: Callback-Funktion von createServer()`

Auf welcher Grundlage wird entschieden, ob ein HTTP-Request mit der Auslieferung eines statischen Dokuments erwidert wird, oder ob die Bearbeitung des Requests durch das Skript `http2mdb` erfolgt?

Was muss auf Seiten von `TopicviewViewController.js` getan werden, damit die CRUD-Operationen durch `http2mdb` bearbeitet werden können?

3. `njsimpl/http2mdb.js: processRequest()` **sowie** `doGet(), doPost(), doPut()` **und** `doDelete()`

Nach welchem Kriterium erfolgt die Weiterverarbeitung der Anfrage, die an `processRequest()` übergeben wird?

4. `http2mdb.js: readTopicview(), createTopicview, etc.`

Wie erfolgt in diesen Funktionen das Zusammenspiel zwischen ggf. dem Auslesen des HTTP-Request-Body, dem Aufruf der betreffenden CRUD-Funktion der MongoDB-API, sowie der Erzeugung des HTTP-Response?

In welchen Funktionen liegt eine „Verschachtelung“ mehrerer Callback-Funktionen vor, d. h. die Verwendung von Callbacks in Callbacks? Weshalb ist dies an den betreffenden Stellen erforderlich?

[Demovideo](#)

Bearbeitungszeit: 45 Minuten



Übung NJM-02

CRUD für „Imgbox“

Aufgabe

Implementieren Sie die Operationen `create`, `update` und `delete` für `Imgbox`-Elemente und zeigen Sie `Imgbox`-Elemente in der `Topic`-Ansicht an. Erforderlich sind hierfür nur clientseitige Implementierungsmaßnahmen.

Alle serverseitigen Operationen sind bereits umgesetzt. Diese Aufgabe ist eine "Vorstufe" für die Aufgabe NJM3, die Sie in NJM3 erweitern werden. Insbesondere werden Sie dort auch die `read`-Operation bezüglich der mit `Topic`-Ansichten assoziierten `Imgbox`-Elementen umsetzen.

Anforderungen

1. Die `Imgbox`-Elemente sollen über die folgenden Attribute verfügen: `src`, `title` und `description`.
2. `Imgbox`-Elemente sollen anhand der serverseitig zugewiesenen `id` identifiziert werden.
3. Setzen Sie die Operationen durch Ausführung eines `XMLHttpRequest` bezüglich des mittels `webserver.js` gestarteten NodeJS Servers um.
4. Weisen Sie beim Aufruf von `create` einen Wert Ihrer Wahl für `src` zu und weisen Sie bei Ausführung von `update` einen davon verschiedenen Wert zu.
5. Nachdem ein `Imgbox`-Element serverseitig erfolgreich erstellt worden ist, soll dieses entsprechend dem in CSS3 vorgesehenen Styling auf der `Topic`-Ansicht dargestellt werden.
6. Nach Ausführung der Operationen `update` und `delete` soll die Darstellung des `Imgbox`-Elements modifiziert bzw. entfernt werden.
7. Die Operationen `update` und `delete` brauchen für diese Aufgabe nur dann ausführbar zu sein, wenn unmittelbar zuvor `create` ausgeführt wurde. Unmittelbar nach einem `reload` der Ansicht oder unmittelbar nach Zugriff aus der Titelliste müssen die Aktionen nicht funktionsfähig sein. Diese Einschränkung werden Sie in NJM3 aufheben.

Bearbeitungshinweise

- Verwenden Sie für die Umsetzung eine Kopie des Beispielprojekts `org.dieschnittstelle.iam.njm_frm_mfm`. In diesem Projekt können Sie dann alle folgenden Aufgaben umsetzen.
- Um die von MongoDB verwendeten Daten Ihrer Anwendung getrennt von den Daten der Implementierungsbeispiele zu handhaben, setzen Sie bitte den Namen Ihres Projekts im `-dbpath` Parameter im `mdb` Skript.
- **Anforderung 2:** Die `id` ist Ihnen bis auf Weiteres nur dann bekannt, wenn Sie die `create` Operation ausgeführt haben. Bei Zugriff auf die `Topic`-Ansicht kennen Sie die `id` noch nicht. Dafür werden Sie in NJM3 Erweiterungen vornehmen.
- **Anforderung 3:** Zur Auslösung der CRUD Operationen stehen Ihnen bereits Aktionen in der Fußleiste von `topicview.html` zur Verfügung, die Sie bei Betätigung des Pfeils am rechten Rand der Fußleiste angezeigt bekommen.

Demoversion

Bearbeitungszeit: 45 Minuten



Übung NJM-03

Verknüpfung „Topicview“ - „Imgbox“

Aufgabe

Assoziieren Sie Imgbox-Elemente auf Ebene der Datenbank mit den Topic-Ansichten in denen die Imgbox-Elemente erstellt werden. Auch hierfür sind keine serverseitigen Implementierungsmaßnahmen erforderlich.

Anforderungen

1. Erweitern Sie die `create` Operation aus NJM2 wie folgt: falls ein `Imgbox` erfolgreich erstellt wurde, soll der `contentItems` Liste des `Topicview`, von dem aus `createImgbox()` aufgerufen wurde, eine Referenz der folgenden Form hinzugefügt werden: `{type: "imgbox", renderContainer: "left", imgboxid: <imgboxid>}`, wobei `<imgboxid>` die `id` des neu erstellen `Imgbox-Elements` ist.
2. Setzen Sie die Aktualisierung von `contentItems` so um, dass nur das neu hinzuzufügende Element vom Client an den NodeJS Server und von dort an MongoDB übermittelt wird und nicht das gesamte `Topicview` Objekt aktualisiert wird.
3. Erweitern Sie die `delete` Operation aus NJM2 wie folgt: falls ein `Imgbox-Element` erfolgreich gelöscht wurde, soll auch die Referenz auf dieses `Imgbox-Element` aus `contentItems` entfernt werden. Auch hierfür soll nur ein partielles Update durchgeführt werden.
4. Wenn für einen `Topicview` bereits ein `Imgbox-Element` existiert, soll dieses bei Darstellung des `topicview.html` Dokuments entsprechend dem dafür vorgesehenen Styling (siehe Übung CSS3) dargestellt werden.
5. Falls ein `Imgbox-Element` bereits existiert oder neu erstellt wurde, soll bei erneuter Ausführung von `createImgbox()` ein Warnhinweis angezeigt werden und die `Imgbox-Erstellung` unterbunden werden.
6. Alle anderen Funktionen aus NJM2 sollen weiterhin verfügbar sein. Lediglich die Einschränkung aus **Anforderung 7** wird durch die Umsetzung von NJM3 aufgehoben. Falls dies nicht erfüllt ist, wird für NJM3 maximal die Hälfte der möglichen Punktzahl vergeben.

Bearbeitungshinweise

- **Anforderung 1, Anforderung 2, Anforderung 3:** Die Funktionen in `http2mdb.js` stellen bereits die Funktionalität des partiellen Updates für die Hinzufügung von Elementen zu `content_items` gemäß **Anforderung 1** und das Löschen gemäß **Anforderung 3** zur Verfügung. Um auf diese Funktionalität zuzugreifen, müssen Sie lediglich auf Ebene des browserseitig ausgeführten JavaScript-Codes HTTP-Requests der folgenden Form initiieren:
 - Hinzufügung: `PUT /topicviews/<topicid>/content_items` und Übertragung des hinzuzufügenden Elements im Body des `Requests.<topicid>` identifiziert den `Topicview`, dessen `content_items` Liste das Element hinzugefügt werden soll.
 - Löschen: `DELETE /topicviews/<topicid>/content_items/<elementtype>`. Auch hier identifiziert `<topicid>` den `Topicview`, auf dessen `content_items` Liste zugegriffen werden soll. `elementtype` bezeichnet den Typ des Elements / der Elemente, die dabei aus `content_items` entfernt werden sollen. In Ihrem Fall sieht die URL also wie folgt aus: `/topicviews/<topicid>/content_items/objekt`

- **Anforderung 3** Für das Löschen, aber auch für das Auslesen oder die Aktualisierung eines Objekts benötigen Sie `_id` des Objekts, die Sie aus der Objektreferenz aus `contentItems` auslesen können. Dafür steht Ihnen in `TopicviewViewController.js` die Funktion `getObjektIdForTopicview()` zur Verfügung.
- **Anforderung 4** Die Funktion zum Auslesen eines evtl. bereits existierenden `Imgbox-Elements` können Sie aufrufen, nachdem ein `Topicview` ausgelesen wurde und falls dessen `contentItems` Liste eine `Imgbox-Referez` enthält.

Bearbeitungszeit: 45 Minuten

Wissensüberprüfung

Versuchen die hier aufgeführten Fragen zu den Inhalten der Lerneinheit selbständig kurz zu beantworten bzw. zu skizzieren. Wenn Sie eine Frage noch nicht beantworten können, kehren Sie noch einmal auf die entsprechende Seite in der Lerneinheit zurück und versuchen sich die Lösung zu erarbeiten.



Formulieren

Übung NJM-04

NodeJS

Versuchen Sie die hier aufgeführten Fragen selbständig kurz zu beantworten bzw. zu skizzieren.

1. Was ist der Nachteil bei der Verwendung von Betriebssystem-Threads zur Bearbeitung der Anfragen an einen Server?
2. Was ist der Nachteil von synchron ausgeführten I/O-Operationen, insbesondere falls mehrere Operationen nacheinander ausgeführt werden sollen?
3. Nennen Sie drei Typen von I/O-Operationen, die in NodeJS asynchron ausgeführt werden können.
4. Welche Vorteile erwachsen aus der Handhabung von I/O-Operationen in NodeJS?
5. Was ist die Aufgabe des „Event-Loop“ in NodeJS?

Bearbeitungszeit: 30 Minuten



Formulieren

Übung NJM-05

MongoDB

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten bzw. zu skizzieren.

1. Nennen Sie drei Typen von Datenbanken, die als „NoSQL“ bezeichnet werden können.
2. Worin liegen die Vorteile relationaler Datenbanken gegenüber NoSQL Ansätzen?
3. Worin liegen die Vorteile von NoSQL Datenbanken gegenüber relationalen Datenbanken?
4. Was ist eine „Collection“ in MongoDB?
5. Für welchen Typ von Beziehung zwischen Instanzen komplexer Datentypen erleichtert eine Datenbank wie MongoDB die Repräsentation, verglichen mit relationalen Datenbanken? Welcher Typ von Beziehung erfordert ggf. höhere manuelle Aufwände bei der Implementierung?
6. Was für ein Programmierkonstrukt müssen Sie verwenden, um auf Ebene der JavaScript-API für MongoDB auf das Ergebnis einer CRUD-Operation zu reagieren?
7. Was bezeichnet der Begriff „upsert“ in MongoDB?

Bearbeitungszeit: 45 Minuten



Formulieren

Übung NJM-06

CRUD-Operationen

Versuchen die hier aufgeführten Fragen selbständig kurz zu beantworten bzw. zu skizzieren.

1. Was ist die Bedeutung der Bestandteile der Abkürzung CRUD?
2. Mittels welcher HTTP-Methoden können CRUD-Operationen identifiziert werden? Ordnen Sie die Methoden der jeweiligen Operation zu?
3. Für welche CRUD-Operationen müssen Sie bei Identifikation durch HTTP-Requests keinen Request-Body verwenden?
4. Angenommen, `/elements/` identifiziert eine Menge von Objekten: Wie können Sie durch eine URL ohne Verwendung von URL-Query Parametern ein konkretes Element dieser Menge mit Identifikator ID identifizieren?
5. Beschreiben Sie die Schritte, die ausgeführt werden, wenn eine Webanwendung browserseitig via HTTP auf eine serverseitige CRUD-Operation zugreift.

Bearbeitungszeit: 15 Minuten