LDS - Lokale Datenspeicherung

Überblick und Lernziele

Zu Beginn des Studienmoduls haben wir uns in den Lerneinheiten HTM, CSS und JSL mit Anwendungen beschäftigt, die mittels HTML, CSS und JavaScript einem Nutzer den Zugriff auf eine feststehende Menge von Inhalten – Texte und Bilder in unserem Fall – ermöglichten. „Dynamisch“ war hier lediglich das Verhalten der Nutzerschnittstelle unserer Anwendung, beispielsweise konnten wir Übergänge zwischen den Ansichten der Anwendung ohne Neuladen von serverseitigen Inhalten mittels JavaScript und CSS umsetzen. Auf Grundlage von Daten, die uns durch den Server bereitgestellt wurden, konnten wir außerdem Ansichten selbst dynamisch aufbauen.

Im Anschluss daran haben wir es den Nutzern unserer Anwendung in den Lerneinheiten NJM, FRM, MFM und MME ermöglicht, durch Verfügbarmachung schreibender Operationen auf einem serverseitigen Datenspeicher selbst Inhalte dauerhaft zu erstellen und auf diese Inhalte aus der verwendeten Anwendung zuzugreifen. Die Funktionsfähigkeit der Anwendung war hier jedoch vollkommen abhängig von der Zugreifbarkeit und Verfügbarkeit des NodeJS Servers und der durch diesen verwendeten MongoDB Datenbank.

Die letzten beiden Lerneinheiten unserer Veranstaltung, deren erste Sie hier bearbeiten werden, werden in Ergänzung und Abrundung des bisherigen Programms Ausdrucksmittel vorstellen, die unsere Anwendung schrittweise unabhängig von einem laufenden und zugreifbaren Webserver machen. So werden wir hier u. a. mit *IndexedDB* die Verwendung einer lokalen, auf dem Endgerät laufenden Datenbank als Alternative zur serverseitigen Datenspeicherung in MongoDB vorstellen.

Als Abschluss des Semesterstoffs werden Sie schließlich nach Absolvierung der nachfolgenden Lerneinheit dazu in der Lage sein, nicht nur die von der Anwendung verwendeten *Inhalte*, sondern auch „die Anwendung selbst“ – d. h. den von uns in HTML, CSS und JavaScript entwickelten Quellcode der Anwendung – unabhängig von der Verfügbarkeit eines Web Servers auszuführen. Ein Zugriff auf den Server wird dann lediglich noch für die erstmalige Installation der Anwendung sowie für deren Aktualisierung erforderlich sein. Davon abgesehen wird Ihre Anwendung aber vollständig im Offline-Betrieb lauffähig sein.

Mit dem auf diese Weise beschrittenen Weg von statischen Anwendungen, über dynamische Anwendungen mit serverseitiger Datenspeicherung hin zu offline-fähigen Anwendungen werden Sie das gesamte Spektrum an Nutzungsfällen kennengelernt haben, das native mobile Anwendungen seit ihrem Entstehen abdecken und für das die ihnen zugrunde liegenden APIs die erforderlichen Ausdrucksmittel zur Verfügung stellen. Für mobile Webanwendungen hingegen konnte auch im Hinblick auf die Entkoppelbarkeit einer Anwendung von einem Server, der einerseits als „Quellcodequelle“ und andererseits als Speicher für dynamische Inhalte fungiert, ein Defizit konstatiert werden, das erst durch die aktuellen Standardisierungsbemühungen behoben wird. Im Sinne unserer Veranstaltung wird es also hier und in der folgenden Lerneinheit darum gehen, aufzuzeigen, wie „Internetanwendungen für mobile Geräte“ unabhängig „vom Internet“ gemacht werden können.

Nachfolgend werden wir zunächst einen Überblick über verschiedene Verfahren zur lokalen Datenspeicherung geben, die von Webanwendungen genutzt werden können. Danach werden wir die Ausdrucksmittel der IndexedDB API vorstellen, mittels derer die Speicherung größerer Mengen strukturierter Daten auf dem verwendeten Endgerät möglich ist.


Lernziele

Lernziele

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

- Die verschiedenen Möglichkeiten zur clientseitigen Datenspeicherung, die von Webanwendungen genutzt werden können zu nennen.
- Die Same Origin Policy für Webanwendungen zu erläutern.
- Die Ausdruckmittel des HTTP-Protokolls für Cookies und die Einschränkungen bezüglich der Verwendung von Cookies zu beschreiben.
- Den Unterschied zwischen sessionStorage und localStorage darzustellen.
- IndexedDB zur lokalen Speicherung von Instanzen komplexer Datentypen auf einem Endgerät einzusetzen.


Gliederung

Gliederung

- Sicherheitsaspekte
- Traditionelle lokale Speicherverfahren
- Aktuelle Verfahren zur lokalen Datenspeicherung
- Zusammenfassung
- Wissensüberprüfung
- Übungen


Zeitbedarf

Zeitbedarf und Umfang

Für die Bearbeitung der Lerneinheit benötigen Sie etwa 180 Minuten. Für die Bearbeitung der Übungen der Beispielanwendung etwa 2,5 Stunden und für die Fragen zur Wissensüberprüfung ca. 1 Stunde.

1 Sicherheitsaspekte

Vielleicht fragen Sie sich, weshalb die Frage der lokalen Datenspeicherung bzw. generell die bereits oben aufgeworfene Frage des Zugriffs von Webanwendungen auf das lokale Dateisystem des Endgeräts überhaupt relevant ist. Stellen Sie sich aber einmal vor, dass jede Webanwendung unbeschränkt und ohne Ihre steuernde Einflussnahme auf Ihr Dateisystem zugreifen könnte – würde dies nicht ein erhebliches Sicherheitsrisiko bedeuten? Wenn dem aber so ist, was ist dann das unterscheidende Merkmal zwischen nativen Anwendungen – seien sie für mobilen oder stationären Gebrauch konzipiert – und Webanwendungen, auf dessen Grundlage ersteren ein solcher Zugriff üblicherweise möglich ist?

Diese Frage führt uns zu einer Maßnahme, die für gewöhnlich als Hürde bezüglich der Nutzung einer Softwareanwendung angesehen wird, nämlich zum Schritt der Installation einer Anwendung. Im Hinblick auf die beschriebene Problematik ist die Installation der entscheidende von Ihnen als Nutzer eines Endgeräts vollzogene Schritt, mittels dessen Sie einer Anwendung das Recht geben, nicht nur auf dem Endgerät betrieben zu werden, sondern ggf. auch lesend und schreibend auf das Dateisystem des Geräts zuzugreifen.

Aufgrund des Gewichts, das diesem Schritt zukommt, wird Ihnen üblicherweise empfohlen, nur „vertrauenswürdige Anwendungen“ bzw. „Anwendungen aus vertrauenswürdigen Quellen“ zu installieren – oder ist gar die Berechtigung zur Installation einem ausgewählten Personenkreis von Administratoren vorbehalten. So machen denn auch native Anwendungen, die Sie auf Ihrem Smartphone installieren, aber auch Anwendungen für Facebook, die auch als Webanwendungen bereitgestellt werden können, üblicherweise in Form eines „Beipackzettels“ – siehe folgende Abbildung – transparent, in welchem Maße sie die Ressourcen und Funktionen, die das Gerät zur Verfügung stellt, nutzen. Sie haben es dann selbst in der Hand, zu entscheiden, ob Sie die Anwendung installieren und nutzen wollen oder nicht.

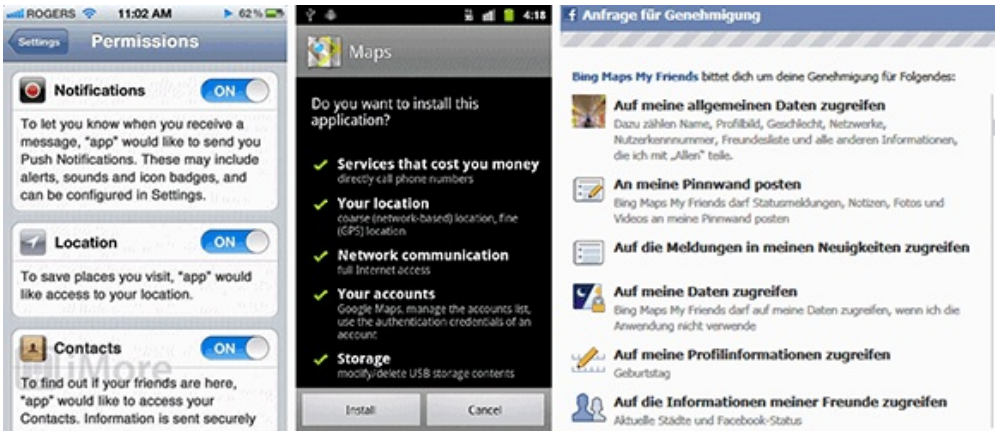


Abb.: „Beipackzettel“ von Apps für iOS, Android und Facebook (von links nach rechts)

Sandboxing

Im Gegensatz zu Anwendungen wie den gerade beschriebenen bedürfen Webanwendungen, die von einem Browser geladen werden, keines Installationsprozesses. Die Ausführung der Anwendung erfolgt vielmehr durch bloßes Laden der dafür erforderlichen Ressourcen beim Zugriff auf die entsprechende Website. Aus diesem Grund gelten Webanwendungen aber auch als grundsätzlich weniger vertrauenswert als installierbare Anwendungen und unterliegen daher spürbaren Restriktionen. So müssen Browser verhindern – oder zumindest versuchen zu verhindern, dass Webanwendungen unkontrolliert auf lokale Daten bzw. auf Daten anderer Webanwendungen zugreifen können, die etwa in einem anderen Tab oder Browserfenster ausgeführt werden.

Die Umsetzung dieser Anforderung obliegt freilich den Browserherstellern. Beispielsweise führt Chrome jeden Tab in einem separaten Prozess aus, der als „Sandbox“, d. h. als „geschützte Umgebung“ jeglichen unkontrollierten Zugriff von außen und nach außen unterbindet. Kommuniziert werden diese Merkmale u. a. durch ein Video, dessen Zielgruppe nicht eine reine Entwickler-Community, sondern ein deutlich darüber hinaus reichende Nutzergruppe ist. Dies verdeutlicht, dass die genannten Sicherheitsanforderungen von den Entwicklern von Chrome durchaus auch als Differenzierungsmerkmal gegenüber anderen Browsern angesehen werden. Für eine deutlich weniger publikumswirksame Darstellung des offensichtlich weniger ausgereiften Sandbox-Konzepts von Firefox siehe eine Dokumentationsseite sowie eine umfangreiche Forendiskussion zur Gegenüberstellung der Sicherheitskonzepte der beiden Browser.

Nutzerkontrolle

Ein weiteres Merkmal, das Webanwendungen im Hinblick auf die Absicherung gegenüber unbefugten Zugriffen kennzeichnet, ist die Delegierung von Zugriffen an den Nutzer. Sehr deutlich wird dies anhand der Verwendungsweise des bereits mehrfach betrachteten `<input>` Elements mit `type="file"`. Anhand dieses Elements hatten wir in der vorangegangenen Lerneinheit gesehen, dass es uns auf Ebene des JavaScript Codes einer Anwendung durchaus möglich ist, Dateiinhalte auszulesen. Allerdings musste hier dem Auslesen die Auswahl der auszulesenden Datei durch den Nutzer vorausgehen.

In der Lerneinheit zu Multimedia Elementen wurde aber noch ein weiteres Beispiel für die Delegierung von sicherheitsrelevanten Entscheidungen an den Nutzer während der Nutzung einer Anwendung gezeigt, nämlich die Rückbestätigung sicherheitsrelevanter Zugriffe durch den Browser, die wir in Lerneinheit MME, Kapitel 3.2 anhand eines Beispiels gezeigt haben, und die uns auch an verschiedenen anderen Stellen im Rahmen der Verwendung aktueller Ausdrucksmittel von HTML begegnen, z. B. unten bei der Nutzung von `LocalStorage`, aber auch bei der Verwendung der API für Geolocations. In Ermangelung eines obligatorischen Installationsprozesses, der – wie im Fall nativer mobiler Apps – als eine pauschale Gewährung von Berechtigungen angesehen werden kann, werden wir hier ggf. für jeden durch eine Anwendung unternommenen Zugriff gefragt, ob wir diesen gewähren möchten. Angemerkt sei, dass Firefox OS als „Betriebssystem für mobile Webanwendungen“ u. a. verschiedene Vertrauensstufen für Webanwendungen vorsieht sowie einen Installationsvorgang, der dem von nativen Anwendungen durchaus vergleichbar ist. Siehe dafür die Ausführungen zum Sicherheitskonzept.

Same Origin Policy

Im Grunde kann aber bereits die Eingabe einer URL in die Adresszeile des Browsers oder das Verfolgen eines Hyperlinks bezüglich eines Dokuments, das von einer anderen Domain als der des aktuell geladenen und dargestellten Dokuments bereitgestellt wird, als ein Schritt angesehen werden, bezüglich dessen dem Nutzer eine wesentliche Verantwortung im Hinblick auf die sicherheitsbezogenen Konsequenzen zugewiesen wird. So obliegen denn vom Moment des Ladens eines Dokuments aus einer Domain jegliche nicht durch den Nutzer selbst initiierten Zugriffe auf serverseitige Funktionen der Beschränkung durch die *Same Origin Policy*. Diese schränkt beispielsweise die Ausführung von Formularaktionen wie von „unter der Haube“ ablaufenden Server-Zugriffen via `XMLHttpRequest` auf die Ursprungsdomain des aktuell geladenen Dokuments ein (für die Definition einer Ursprungsdomain bzw. der Übereinstimmung von *Ursprungsdomains* siehe http://tools.ietf.org/html/rfc6454).

Der Ursprungsdomain kommt aber auch eine wesentliche Rolle im Hinblick auf alle nachfolgend betrachteten lokalen Speicherverfahren zu. So sieht die HTML-Spezifikation vor, dass ein Speicherzugriff grundsätzlich nur bezüglich derjenigen Daten erfolgen kann, die mit derselben Ursprungsdomain assoziiert sind wie das aktuell geladene Dokument. Für jede Ursprungsdomain existiert also eine eigene Instanz – sei sie physikalisch oder virtuell – des betreffenden Speichern, die gegen Zugriffe aus anderen Domains geschützt ist – bzw. mit Blick auf die Anforderungen an Browser geschützt sein sollte.

2 Traditionelle lokale Speicherverfahren

Web Storage / Local Storage

Wir geben hier zunächst einen kurzen – eher theoretischen – Überblick über verschiedene alternative – und lange in Gebrauch befindliche – Verfahren zur lokalen Datenspeicherung, die von Webanwendungen genutzt werden können. Danach werden wir uns mit den beiden in der aktuellen HTML-Version vorgesehenen und durch Browser weitgehend oder teilweise unterstützten APIs für `WebStorage` – oft auch als `LocalStorage` bezeichnet – sowie `IndexedDB` als browserseitiger Datenbank beschäftigen.

FileSystem API

Nur erwähnen möchten wir an dieser Stelle die `FileSystem` API. Diese beschreibt ein von Webanwendungen nutzbares virtuelles Dateisystem auf Grundlage der Dateiverwaltung des Endgeräts, das den Anforderungen einer „Sandbox“ im oben genannten Sinne genügt. Hierbei handelt es sich aber laut Mozilla Developer Network um eine „experimentelle Technologie“, für die Firefox – im Ggs. zu Chrome und Opera – auch noch keine Unterstützung anbietet und die auch aus diesem Grund nicht für den praktischen Stoff dieser Lerneinheit in Betracht gezogen wurde.

Betrachten werden wir im folgenden mit Caching und Cookies zwei Verfahren zur lokalen Datenspeicherung, die bereits lange verfügbar sind. So kann Caching, wie wir gleich darlegen werden, als zum „Grundmechanismus des WWW“ gehörig angesehen werden, insofern als es nicht unwesentlich zur Funktionsfähigkeit des Webs beitrug und – auch in Zeiten scheinbar unbegrenzter Bandbreite – noch beiträgt.

2.1 Caching

In Bezug auf Webanwendungen bezeichnet Caching die lokale Speicherung der Inhalte von HTTP Responses durch einen Client – im hier betrachteten Rahmen ist dies der Browser eines Endnutzer, Caching kann jedoch auch durch einen Proxy-Server vorgenommen werden. Ziel des Cachings ist die Verhinderung überflüssigen Netzwerkverkehrs bzw. des redundanten Datentransports über ein Netzwerk. Ausgangspunkt dafür bildet die durchaus nachvollziehbare Annahme, dass Ressourcen, die über einen bestimmten Zeitraum unverändert sind, nicht bei jedem Zugriff innerhalb dieses Zeitraums von neuem übertragen werden müssen. Beachten Sie, dass damit gerade in den Anfangsjahren des „Web 1.0“ gleichermaßen eine positive Auswirkung auf die User Experience erzielt werden konnte wie Caching eine optimale Nutzung der insgesamt im Netz verfügbaren Bandbreite ermöglichte und ermöglicht.

Caching und HTTP

Aus diesen Sachverhalten motiviert sich unsere Rede von Caching als einem „Grundmechanismus“ des WWW, und so ist Caching denn auch Bestandteil der HTTP Spezifikation. Diese legt u. a. Caching-bezogene Header für Requests und Responses sowie Status Codes fest, anhand derer ein Server einem Client, der auf eine Ressource zugreift, anzeigen kann, ob und ggf. wie das Caching der Ressource umgesetzt werden soll. Als Identifikator dient jeweils die *vollständige* URL mit der die betreffende Ressource identifiziert wird, inklusive etwaiger Query Parameter.

Caching Varianten

Bezüglich des Cachings lassen sich die folgenden beiden Umsetzungsvarianten unterscheiden:

1. Der Server zeigt einem Client an, dass eine Ressource bis zu einem *Ablaufdatum* bzw. bis zum *Ablauf eines Zeitintervalls* ohne erneuten Zugriff auf den Server verwendet werden kann. In diesem Fall wird der Client die betreffende Ressource im Cache ablegen und tatsächlich keinen erneuten Zugriff unternehmen, bis der jeweils identifizierte Zeitpunkt erreicht ist, d. h. hier wird jegliche Client-Server Interaktion bezüglich der gecachten Ressource unterbunden – es sei denn Sie als Nutzer führen explizit ein Refresh bezüglich der Ressource oder einer anderen Ressource – etwa eines HTML-Dokuments – durch, in die diese eingebunden ist.

2. Die zweite Variante für Caching ist dahin gehend flexibler, dass sie serverseitig keine Festlegung bezüglich eines absoluten oder relativen Ablaufdatums erfordert, innerhalb dessen keine relevante Änderung einer Ressource stattfindet oder zu erwarten ist. Stattdessen ermöglicht sie eine jederzeitige Aktualisierung von Ressourcen mit der Gewährleistung, dass eine Ressource neu geladen wird, wenn sie tatsächlich aktualisiert worden ist. Zu diesem Zweck werden Ressourcen vom Server mit einem im Header gesetzten Schlüssel – einem sogenannten *ETag* – ausgeliefert, der den Zustand einer Ressource zum Zugriffszeitpunkt identifiziert. Anhand dieses Schlüssels kann der Server bei einem erneuten Zugriff durch den Client erkennen, ob seit dem letzten Zugriff, anlässlich dessen der Schlüssel ausgeliefert wurde, eine Änderung an der Ressource vorgenommen wurde oder nicht.

Im Gegensatz zum ersteren, „verfallsdatumsbasierten“ Verfahren ist für das zweite Verfahren für jede auf Clientseite initiierte Verwendung einer Ressource ein Zugriff auf den Server erforderlich, der dem Abgleich des ETag dient. Kann auf Grundlage dessen keine Änderung der Ressource festgestellt werden, übermittelt der Server den Status Code `304` und zeigt damit an, dass die Ressource `NOT MODIFIED` ist. In diesem Fall wird der Client die Ressourceninhalte aus dem Cache laden und nicht erneut vom Server auslesen. Das Verfahren spart damit zwar nicht die Anzahl an Client-Server Interaktionen gegenüber einem Szenario ohne Caching ein, wohl aber die übertragene Datenmenge. Zugleich ist es hinsichtlich möglicher Änderungen der Ressourcen deutlich flexibler als das erstgenannte Verfahren.

Wie bei allen Festlegungen der HTTP-Spezifikation gilt auch hier, dass es die Aufgabe der jeweiligen Implementierung eines Webservers wie auch der Client-Komponenten ist, Caching gemäß der Spezifikation zu unterstützen. Im Rahmen einer Eigenentwicklung eines Servers, wie wir sie in NodeJS umgesetzt haben, muss daher u. a. auch der Unterstützung für Caching Rechnung getragen werden – es sei denn die besagte Anwendung möchte auf Caching verzichten. Einblick in den Cache Status auf Clientseite erhalten Sie unter Firefox über die URL `about:cache`. Diese gibt Ihnen, wie Sie in folgender Abbildung sehen, sowohl Zugriff auf etwaige Cache-Inhalte im Arbeitsspeicher und auf der Festplatte („Memory cache device“ bzw. „Disk cache device“) als auch Einblick in das Offline Cache Ihrer Anwendung, das Sie in der folgenden Lerneinheit kennen lernen werden.

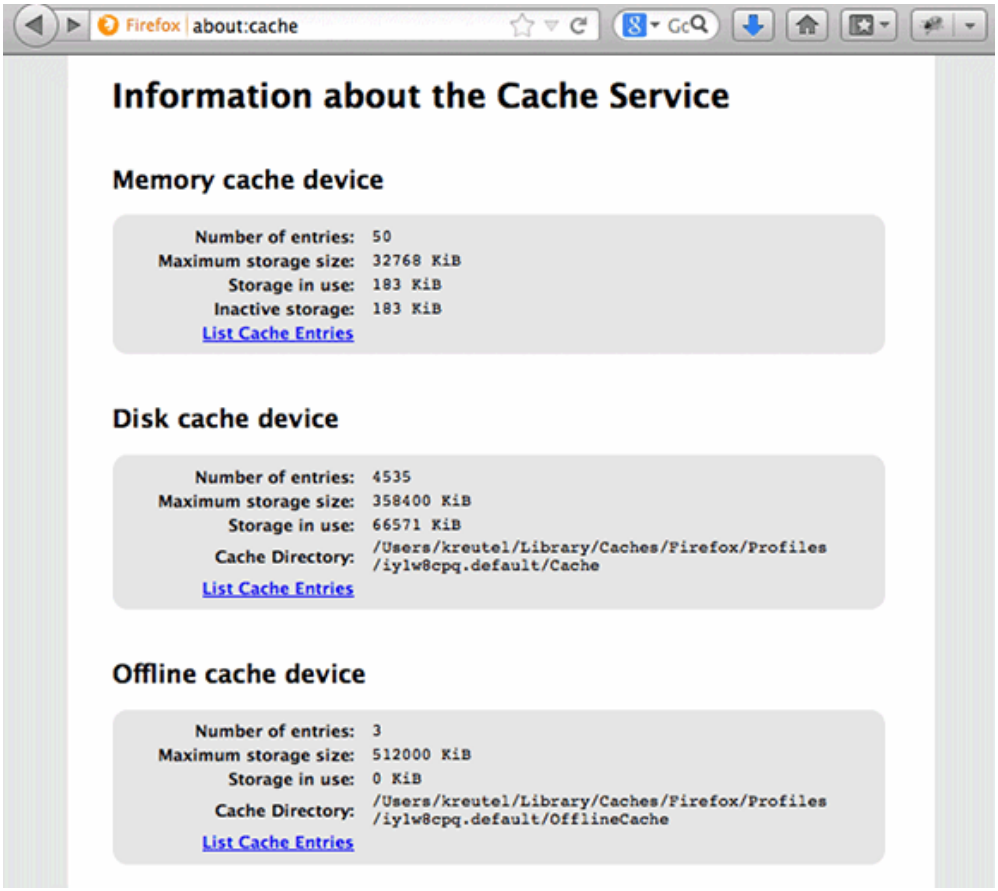


Abb.: Hauptansicht der `about:cache` Zugriffsfunktionen von Firefox

Caching und JavaScript

Clientseitig liegt bei Verwendung eines Browsers jegliche Caching-bezogene Funktionalität exklusiv in dessen Händen, und auch mittels Workarounds scheint keine Zugriffsmöglichkeit auf das Browser Cache in JavaScript zu bestehen. Auch bei Verwendung von `XMLHttpRequest` ist die Umsetzung des Cachings unterhalb der JavaScript API verborgen, es existieren hier wie dort aber „offizielle Tricks“, wie das Cache umgangen werden kann – diese verlassen sich dann z. B. darauf, dass Caching jeweils bezüglich exakter und vollständiger URLs durchgeführt wird. Jegliche Modifikation der URL – und sei es auch hinsichtlich irrelevanter Bestandteile – wird daher ein Neuladen gecachter Ressourcen erzwingen.

Es existieren auch proprietäre Lösungen, die Caching unabhängig vom Browser umsetzen, die an dieser Stelle aber nur erwähnt und nicht empfohlen werden sollen. Caching ist denn auch nicht zuletzt serverseitig eine nicht triviale Funktionalität, für deren Unterstützung aber „gewachsene“ Softwarekomponenten wie der Apache Webserver bereitstehen.

2.2 Cookies

Ein weiteres Ausdrucksmittel zur clientseitigen Datenspeicherung, auf das Sie als Entwickler deutlich mehr Einfluss haben als bezüglich des Cachings von Ressourcen, sind Cookies. Auch diese stützen sich auf Ausdrucksmittel von HTTP und wurden bereits in den ersten Jahren des Web konzipiert und durch Browser unterstützt.

Spezifiziert wurden Cookies 1997 als State Management Mechanism für HTTP; der die Identifikation der Zustände einer Client-Server Interaktion erlauben sollte – bedenken Sie, dass HTTP von seiner Konzeption her ein genuin zustandsloses Protokoll ist. D. h. unter Verwendung von Cookies können Clients einen Interaktionszustand über eine Folge von HTTP Requests oder Anwendungsnutzungen hinweg identifizieren. Gebräuchlich sind Cookies u. a. zur reinen *Identifikation* von Nutzern – wichtig: nicht zu deren Authentifizierung – die es u. a. ermöglichen, einen Warenkorb über eine Folge von Requests hinweg zu befüllen und auch nach Neustart des Browsers oder Endgeräts die betreffenden Daten verfügbar zu haben. Cookies dienen also dazu, clientseitig nutzerbezogene Daten zu speichern, die von Webanwendungen verwendet werden sollen.

Cookies können von Browsern – so der Nutzer ihre Verwendung zulässt – generisch behandelt werden, indem sie ein einheitliches Format verwenden. Dieses umfasst eine Menge einfacher Werte oder Attribut-Werte-Paare, die mit einer URL bzw. einer mittels Wildcards identifizierten Menge von URLs assoziiert werden, die der vorher genannten *Same Origin Policy* genügen müssen. Darüber hinaus können für Cookies Gültigkeits- und Sicherheitsbedingungen angegeben werden, die ihre Verwendbarkeit gegenüber einer unbeschränkten Verwendung deutlich einschränken.

Server können also das clientseitige Setzen von Cookies als Form der lokalen Datenspeicherung veranlassen. Erfüllt ein HTTP Request die in einem Cookie angegebenen Anwendungsbedingungen wird der Cookie durch den Browser den Request Headern hinzugefügt, ohne dass dafür anwendungsspezifische Entwicklungsmaßnahmen erforderlich wären. U. a. durch die üblicherweise durch Browser angewandte Größenbeschränkung von ca. 4 KB ist die Verwendung von Cookies jedoch eingeschränkt. Ein Beispiel für Cookies beim Zugriff auf eine weit verbreitete E-Commerce Website sehen Sie in folgender Abbildung.

Abb.: Verwendung von Cookies

Dargestellt ist der `Cookie` Header, in dem der Client Cookies an den Server übermittelt. Der Server veranlasst in seiner Erwiderung mittels `Set-Cookie` die Setzung weiterer Cookies, z. B. der `session-id`, die dann in einem anschließenden Client Request wieder via `Cookie` an den Server übermittelt werden und so den aktuellen Interaktionszustand identifizieren können. Anhand der `expires` Angaben sehen Sie, dass die hier zugegriffene Website sich durchaus großzügig im Hinblick auf die Dauer einer Session gibt.

Cookies in JavaScript

Wir wir oben gesehen haben, ist der Zugriff auf Cache Inhalte gegenüber JavaScript abgeschottet. Hingegen kann auf die mit dem geladenen HTML-Dokument und dessen URL Pfad assoziierten Cookies lesend und schreibend aus JavaScript zugegriffen werden. Zwar können Cookies nicht durch den Entwickler veranlasst auf Headern von `XMLHttpRequest` gesetzt und an einen Server übertragen werden; sie sind jedoch grundsätzlich verwendbar, um rein clientseitig Daten über eine Browsernutzung hinweg zu speichern und erfüllen damit eine Minimalvoraussetzung für die von uns betrachteten lokalen Datenspeicher. Ihre Verwendbarkeit ist jedoch insofern eingeschränkt, als sie aufgrund ihrer String- Repräsentation und der hierfür zu verwendenden Syntax gewisse Aufwände erfordert (siehe jedoch einen verallgemeinernden Lösungsvorschlag aus dem Firefox-Umfeld).

Anstelle von Cookies ist daher für nur clientseitig zu nutzende Daten mittlerweile die Verwendung von *Local Storage* empfehlenswert, auf die wir im folgenden Abschnitt eingehen werden.

3 Aktuelle Verfahren zur lokalen Datenspeicherung

Nachfolgend werden wir zwei Möglichkeiten vorstellen, die im Rahmen der aktuellen Version von HTML für die lokale Speicherung von Daten verwendet werden können. Wie Sie sehen werden, unterscheiden sich diese erheblich im Hinblick auf ihre Ausdrucksstärke, aber auch im Hinblick auf die Komplexität ihrer APIs. Mithin stellen sie keine echten Alternativen für die Umsetzung einer Speicheranforderung dar, sondern können eher als sich einander ergänzende Lösungen mit unterschiedlichen Einsatzgebieten angesehen werden.

Verwiesen sei hier darauf, dass verschiedene Browser darüber hinaus auch die Möglichkeit zur Verwendung lokaler relationaler Datenbanken im SQLite Format ermöglichen, das u. a. auch auf den nativen mobilen Plattformen Android und iOS verfügbar ist. Siehe dafür z. B. die `Storage` API von Firefox – diese sollte nicht verwechselt werden mit der unten beschriebenen gleichnamigen Schnittstellenbeschreibung, die die HTML-Spezifikation für `WebStorage` vorsieht.

3.1 WebStorage

Die `WebStorage` API – auch als *DOM Storage* bezeichnet – positioniert sich eigener Aussage zufolge als eine Alternative zu Cookies, die u. a. die geringe Speicherkapazität und unbequeme Handhabbarkeit der letzteren, aber auch Sicherheitsprobleme, die mit Cookies in Verbindung gebracht werden, adressiert:

Definition

DOM Storage
"DOM Storage is designed to provide a larger, more secure, and easier-to-use alternative to storing information in cookies."
(Siehe https://developer.mozilla.org/en-US/docs/Web/Guide/API/DOM/Storage.)

Spezifiziert wird die API als Bestandteil des Funktionsumfang der aktuellen HTML-Version, und weiterführende Dokumentation und Verwendungshinweise finden Sie wie üblich u. a. auf der Mozilla Entwickler Website.

Für die clientseitige Speicherung von Daten werden in `WebStorage` zwei verschiedene Ebenen angenommen, die als *Session Storage* bzw. *Local Storage* bezeichnet werden. Obsolet ist mittlerweile eine noch vor wenigen Jahren vorgesehene dritte Ebene, die dafür gedacht war, als `globalStorage` anwendungsübergreifend Daten verfügbar zu machen. Zugreifbar sind die beiden unterstützten Ebenen jeweils durch gleichnamige Attribute des `window` Objekts, d. h. sie bedürfen keiner weiteren Präfigierung, sondern können direkt mittels der Ausdrücke `sessionStorage` und `localStorage` referenziert werden.

`sessionStorage`

Das Session Storage erlaubt es, Daten mit einem *Browsing Context* zu assoziieren. Damit wird die im aktuellen Browserfenster oder Browsertab zugegriffene Menge von Dokumenten bezeichnet, für die eine *History* existiert und welche je nach Browserhersteller auch über Neustarts des Browsers hinweg verfügbar ist. So verfügen die meisten Browser über Möglichkeiten, einen Browsing Context nach einem Neustart wieder herzustellen, und erlauben dem Nutzer damit, falls gewünscht, die unmittelbare Anknüpfung an eine vorangegangene Anwendungsnutzung. Das Session Storage ist gebunden an diesen Browsing Context, d. h. die Verfügbarkeit der im Storage enthaltenen Daten hängt von der Handhabung des Browsing Contexts durch den Browser ab. Beachten Sie aber, dass jede Ursprungsdomain innerhalb eines Browsing Contexts ihr eigenes Session Storage hat, d. h. ein Domain-übergreifender Zugriff auf Daten ist trotz der Assoziation mit einem gemeinsamen Kontext nicht möglich – und dürfte aus Sicherheitsgründen auch nicht erwünscht sein.

Aufgrund der Bindung an einen *Browsing Context* ist die Ebene des Session Storage nicht für die *dauerhafte* lokale Datenspeicherung geeignet, sondern nur dafür, Daten innerhalb eines Nutzungsfalls einer Anwendung vorzuhalten. So könnte das Session Storage z. B. für die Übergabe von Daten zwischen verschiedenen Ansichten einer Anwendung verwendet werden, die mittels verschiedener HTML-Dokumente realisiert werden. Beispielsweise könnte es genutzt werden, um aus einer Übersichtsansicht Argumente an eine Detailansicht zu übergeben – d. h. als Alternative zur Nutzung von URL Query Parametern, die Sie in den Implementierungsbeispielen finden.

`localStorage`

Im Gegensatz zum *Session Storage* dient die Ebene des *Local Storage* tatsächlich der dauerhaften Speicherung von Daten, die mit einer Ursprungsdomain assoziiert sind, d. h. sie ist unabhängig von der Lebensdauer jeglicher Browsing Kontexte, in denen die Domain enthalten ist. Local Storage eignet sich damit grundsätzlich für die nutzungsübergreifende Speicherung nutzerbezogener Daten auf einem Endgerät.

Zu beachten ist allerdings zum einen, dass einer Ursprungsdomain, über die mehrere Anwendungen bereitgestellt werden, ein einziges Local Storage zugewiesen wird, sofern die Identifikation der Anwendungen mittels Pfaden unterhalb der Ursprungsdomain erfolgt. Zum anderen wird ein und dieselbe Anwendung mit verschiedenen Ursprungsdomains assoziiert, falls sie über unterschiedliche IP-Adressen, z. B. in einem Heim- vs. Hochschul- vs. Firmennetzwerk zugegriffen wird. Beide Fälle dürften aber nur im Rahmen der Anwendungsentwicklung und nicht im Echtbetrieb auftreten. Falls Sie den Aptana Webserver verwenden oder Webanwendungen in Apache Tomcat entwickeln, sollten Sie sich dessen aber bewusst sein, um etwaige Irritationen zu vermeiden.

Storage API

Session Storage und Local Storage verwenden eine gemeinsame API, die der API von Maps bzw. Dictionaries nachempfunden ist. So können Daten unter einem eindeutigen Bezeichner – `key` in den folgenden Funktionssignaturen – ablegt und ausgelesen werden, wobei jeweils nur Strings als Werte für `key` und `value` zulässig sind:

- `getItem(key)`
- `setItem(key,value)`
- `removeItem(key)`
- `clear()`

Für die Iteration über die Inhalte eines Storage Objekts können dann ggf. ein `length` Attribut sowie eine Funktion `key()` verwendet werden, die für einen Index im Bereich von `length` den Namen des damit assoziierten `key` zurückgibt:

Quellcode

Iteration über die Inhalte eines Storage Objekts

```
001 // iteriere über localStorage
002 for (var i=0;i<localStorage.length;i++) {
003    // ermittle den key für den aktuellen Index
004    var key = localStorage.key(i);
005    console.log("found key value pair: " + key + "=" + localStorage.
006    getItem(key));
007 }
```

Storage Events

Für den Fall, dass ein Nutzer aus zwei verschiedenen Browsing Kontexten – seien es Tabs oder Fenster – auf eine *gemeinsame* Anwendung zugreift, erlaubt es die WebStorage API, Änderungen an Local Storage, die in einem der Kontexte vorgenommen wurden, an die anderen Kontexte zu kommunizieren, um so z. B. eine Aktualisierung der jeweils dargestellten Ansichten zu ermöglichen. So könnten Sie beispielsweise im Rahmen einer Shopping-Anwendung einen clientseitigen Warenkorb verwenden und dem Nutzer den gleichzeitigen Zugriff auf Produktdetailseiten aus mehreren Tabs bezüglich dieses gemeinsamen Warenkorbs ermöglichen.

Als Kommunikationsmittel verwendet die API-Ereignisse, die für das betreffende Local Storage Objekt ausgelöst werden. Diese Ereignisse werden ohne implementatorisches Zutun Ihrerseits durch den Browser bei jedem schreibenden Zugriff auf Local Storage ausgelöst – bei Interesse an einem solchen Ereignis müssen Sie als Entwickler daher nur einen entsprechenden Event Handler für das Ereignis `storage` auf dem `window` Objekt registrieren. Das `event` Objekt, das bei Auftreten eines Ereignisses dem Event Handler übergeben wird, beinhaltet dann die sprechenden Attribute `key`, `oldValue` und `newValue`, die `url` des Dokuments, aus welchem heraus der Zugriff auf Local Storage erfolgt ist, sowie das Attribut `storageArea`, das eine Referenz auf das `localStorage` Objekt selbst enthält:

Quellcode

storage Event aus einem anderen Browsing Kontext

```
001 // reagiere auf ein storage Event aus einem anderen Browsing Kontext
002 window.addEventListener("storage", function(event) {
003    // gib Logmeldungen aus, die die Attribute des event Objekts illustrieren
004    console.log("got storage event " + event + ": " + event.key + "
005     changed from " + event.oldValue + " to " + event.newValue);
006    console.log("new value (read out from event.storageArea): " + event
007     .storageArea.getItem(event.key));
008    // behandle das Event, aktualisiere z. B. die dargestellte Ansicht
009    /* (...) */
010 });
```

Einschränkung von `Storage`

Durch die oben erwähnte Beschränkung auf Strings als Datentyp für speicherbare Werte, eignet sich Local Storage jedoch nur eingeschränkt zur lokalen Datenspeicherung. So müssen auch primitive Datentypen ggf. in das jeweils erwünschte Repräsentationsformat umgewandelt werden, und für das Schreiben und Lesen von JSON-Objekten ist eine Serialisierung bzw. Deserialisierung erforderlich:


Quellcode

JSON-Objekte in Local Storage

```
001 // Schreiben von JSON Objekten in Local Storage
002 function writeJSONToLocalStorage(key, value) {
003    localStorage.setItem(key, JSON.stringify(value));
004 }
005
006 // Auslesen von JSON Objekten
007 function readJSONFromLocalStorage(key) {
008    return JSON.parse(localStorage.getItem(key));
009 }
```

Sie sehen allerdings anhand dieser Beispiele, dass die Implementierungsaufwände für das Lesen und Schreiben von JSON-Daten auch durchaus überschaubar und in hohem Maße generalisierbar sind, so existieren hierfür auch zahlreiche „Komfort-Lösungen“, die Ihnen die Berücksichtigung verschiedener Datentypen bezüglich Session Storage und Local Storage abnehmen. Problematisch erscheinen diese Lösungen aber z. B., wenn mittels ihrer Datenzugriffsoperationen auf Mengen von Instanzen eines Datentyps – „Collections“ im Sinne von MongoDB – umgesetzt werden.

So könnten Sie bei einer einfachen Lösung zwar jede Collection unter einem Schlüssel abspeichern, müssten dann aber für jeden Zugriff die gesamte Collection einer Deserialisierung unterziehen bzw. für schreibende Zugriffe – z. B. das Hinzufügen eines Objekts – die gesamte Collection erneut serialisieren. Falls Sie andererseits jedes Objekt unter seinem Identifikator direkt auf Ebene des Storage abspeichern, müssen Sie zumindest für das Auslesen aller Objekte jeweils über das gesamte Storage iterieren.

Beide Lösungen erscheinen mit Blick auf die Performance Ihrer Anwendungen allenfalls als Fallback geeignet – und sollten ggf. über eine asynchrone API mit Callbacks bereitgestellt werden. So fällt den auch auf, dass der im Standard vorgesehene Zugriff auf Local Storage über eine synchrone API erfolgt und damit darauf hindeutet, dass die hier vorgestellten Ausdrucksmittel nicht für den Nachbau einer Datenbankfunktionalität konzipiert wurden. Einer solchen Verwendung steht nicht zuletzt auch die maximale Gesamtgröße von Local Storage entgegen, die zwar browser-spezifisch ist, sich jedoch browserübergreifend im Rahmen von 5 MB zu bewegen scheint – für einen Kapazitätstest sei auch die Website http://arty.name/localstorage.html empfohlen.

Wofür ist Local Storage angesichts der hier geschilderten Einschränkungen dann aber überhaupt brauchbar? Geeignet erscheint es insbesondere für die Umsetzung jeglicher Anforderungen, welche nicht einen weit gehenden CRUD-Zugriff auf eine größere Menge von Instanzen eines oder mehrerer komplexer Datentypen erfordern, sondern sich mittels Handhabung einer überschaubaren Menge von Key-Value Paaren bewältigen lassen – seien die Werte nun Werte primitiver Datentypen oder Objekte. Darunter fallen z. B. die Speicherung von Konfigurationseinstellungen oder Nutzerpräferenzen, aber auch für den oben erwähnten endgeräteseitigen „Warenkorb“ erscheint die Verwendung von Local Storage nicht abwegig – eine Einkaufshistorie, die Produkt- und Rechnungsdaten über einen längeren Zeitraum hinweg zugreifbar macht, erscheint als Anwendungsfall für Local Storage hingegen problematischer. Für solche Fälle wäre dann die Verwendung von `IndexedDB` als gleichermaßen standardisiertes Ausdrucksmittel zur lokalen Datenspeicherung geeigneter.

Wie Sie im folgenden Abschnitt sehen werden, wird uns hier auch wieder eine ausgeprägte asynchrone API zur Verfügung gestellt, die den Grundsatzerwägungen bezüglich nicht blockierender I/O Zugriffe Rechnung trägt.

3.2 IndexedDB

IndexedDB ist eine clientseitige Datenbank, die die dauerhafte Speicherung von Objekten in *Object Stores* ermöglicht. Objekte können über die uns zur Verfügung gestellte standardisierte JavaScript API als Objekte geschrieben, gelesen und aktualisiert werden, und für verschiedene Typen von Objekten können verschiedene Stores erstellt werden – vergleichbar den Collections, die wir im Rahmen von MongoDB für denselben Zweck nutzen konnten, verwenden aber auch Object Stores kein bestimmtes „Schema“ von Objekten, sondern sind agnostisch gegenüber der internen Struktur eines Objekts. Die API für IndexedDB ist als CRUD API konzipiert und bietet sich damit für die Umsetzung all jener Anforderungen an, die wir oben aus dem Anwendungsbereich von Local Storage ausgeschlossen hatten.

Wie letzteres ist aber auch IndexedDB an die Ursprungsdomain des im Browser geladenen HTML-Dokuments gebunden, aus dem heraus die Datenbank erstellt wird, d. h. pro Ursprungsdomain existiert eine Datenbankinstanz.

IndexedDB vs. MongoDB

Wie wir in der Lerneinheit NJM in unseren Ausführungen zu NoSQL Ansätzen bereits erwähnt hatten, erlauben es Object Stores, Objekte in beliebiger Komplexität unter einem eindeutigen Bezeichner abzuspeichern und ähneln in dieser Hinsicht einer dokumentenorientierten Datenbank wie MongoDB. Auch die grundsätzlichen Vor-und Nachteile gegenüber der Verwendung relationaler Datenbanken, die wir oben in Bezug auf MongoDB dargelegt haben, treffen auf IndexedDB zu.

Wie wir später sehen werden, sind die Zugriffsmöglichkeiten auf Objekte in IndexedDB dabei allerdings weniger flexibel als im Fall von MongoDB. Während dort z. B. Abfragebedingungen bezüglich beliebiger Attribute von Objekten – inklusive eingebetteter Objekte und Arrays – formuliert werden konnten, muss für den Zugriff auf IndexedDB entweder der eindeutige Identifikator oder ein Index verwendet werden, der bezüglich eines Attributs eines Objekts deklariert werden kann. Auch stellt die API keine Möglichkeit für die partielle Aktualisierung von Objekten zur Verfügung – vielmehr muss für eine Update-Operation immer das gesamte Objekt und nicht nur die zu modifizierenden Attribute übergeben werden.

Eine wichtige Gemeinsamkeit der beiden APIs zur implementatorischen Handhabung von IndexedDB und MongoDB hatten wir allerdings schon erwähnt: auch IndexedDB macht in hohem Maße von asynchronen Funktionsaufrufen Gebrauch und erfordert die Angabe von Callback-Funktionen, um auf den Abschluss einer asynchron aufgerufenen Funktion reagieren und deren Resultat weiter bearbeiten zu können.

Objekttypen

Falls Sie in JavaScript verschiedene Objekttypen verwenden, sei darauf hingewiesen, dass IndexedDB nur die Attributwerte von Objekten, inklusive etwaiger eingebetteter Objekte, speichert und dass auf Objekten definierte Funktionen wie auch die Prototyp-Information bezüglich eines Objekts nicht persistiert werden. Ggf. müssen Sie also beim Auslesen eines Objekts aus IndexedDB dieses um die erforderlichen Funktionen des von Ihnen verwendeten Typs anreichern.

Die folgenden Ausführungen zur Handhabung der IndexedDB API orientieren sich an den Implementierungsbeispielen, welche ihrerseits auf Grundlage der Dokumentation im Mozilla Netzwerk erstellt wurden.

3.2.1 Initialisierung von Object Stores

Zugreifen können Sie auf eine IndexedDB Datenbank aus JavaScript unter Angabe eines Namens und einer Versionsnummer. Letztere ist anwendungsspezifisch und erlaubt es Ihnen, unterschiedliche Versionen einer Datenbank mit ggf. unterschiedlichem Funktionsumfang bei der Umsetzung der von Ihrer Anwendung bereitgestellten Funktionalität zu berücksichtigen bzw. dem Nutzer eine Aktualisierung seiner Datenbank anzubieten. Falls noch keine Datenbank existiert oder die existierende nicht in der gewünschten Version vorhanden ist, wird eine Callback-Funktion aufgerufen, die Sie als Wert des Attributs `onupgradeneeded` auf dem `request` Objekt setzen können, das Ihnen die `open()` Funktion als Rückgabewert gibt.

Beachten Sie, dass trotz Bereitstellung eines Rückgabewerts die Ausführung dieser Funktion asynchron erfolgt. Das `request` Objekt dient denn auch gerade dazu, die für die Funktionsausführung relevanten Callbacks zu setzen. Sie sehen hier bereits einige Merkmale der IndexedDB API, die Ihnen immer wieder begegnen werden: Zugriffsfunktionen liefern Ihnen ein `request` Objekt, auf dem Sie je nach Funktion verschiedene Callback-Funktionen setzen können, und diesen Funktionen wird das Ergebnis der Zugriffsfunktion als Wert des Attributs `event.target.result` übergeben.

Im vorliegenden Fall haben wir es außerdem mit einem mehrstufigen Callback zu tun: falls die Versionsnummer der vorliegenden Datenbank nicht der angegebenen entspricht oder falls die Datenbank neu erstellt wird, wird zunächst der Callback bezüglich `onupgradeneeded` aufgerufen und nach dessen Abschluss der als `onsuccess` angegebene Callback. Liegt die Datenbank in der erforderlichen Version vor, wird sofort der `onsuccess` Callback ausgeführt. Aus dessen event Argument kann in beiden Fällen ein vollständig initialisiertes IDBDatabase Objekt ausgelesen werden, das alle erforderlichen Funktionen für den Datenzugriff bereitstellt – in diesem Sinne verwenden wir für seine Repräsentation eine Variable namens `db` und bezeichnen es nachfolgend auch informell als „Datenbank-Objekt“:

Quellcode

IDBDatabase Objekt

```
001 // eine Variable, die mit dem IDBDatabase Objekt initialisiert wird
002 var db;
003
004 /* versuche, die Datenbank des angegebenen Namens in der angegebenen
005    Version zu öffnen */
006 var request = indexedDB.open(dbname, version);
007
008 // setze einen Callback für den Fehlerfall
009 request.onerror = function(event) { /* (...) */}
010
011 // setze einen Callback für den erfolgreichen Abschluss von open()
012 request.onsuccess = function(event) {
013    /* lies das IDBDatabase Objekt aus dem event aus und instantiiere die db
014       Variable */
015    db = event.target.result;
016 }
017
018 /* setze einen Callback, in dem die Initialisierung/Aktualisierung der
019    Datenbank vorgenommen werden kann */
020 request.onupgradeneeded = function(event) {
021    var db = event.target.result;
022    // initialisiere Datenbank und Object Stores
023    /* (...) */
024 }
```

Erstellung von Object Stores

Die Initialisierung einer IndexedDB Datenbank innerhalb der `onupgradeneeded` Callback-Funktion beinhaltet insbesondere die Erstellung von Object Stores, die für die Speicherung der von einer Anwendung genutzten Daten verwendet werden. Dafür steht die Funktion `createObjectStore()` auf dem Datenbank Objekt zur Verfügung. Dieser Funktion wird neben dem Namen des Stores insbesondere ein Parameter-Objekt übergeben, in welchem festgelegt wird, ob die eindeutigen Bezeichner für die zu speichernden Objekte durch die Datenbank zugewiesen werden sollen – angezeigt wird dies ggf. durch das Attribut `autoincrement`. Als Alternative hierzu kann ein sogenannter *Key Path* angegeben werden, der ein Attribut identifiziert, dessen Werte als eindeutige Identifikatoren dienen. Das folgende Beispiel illustriert beide Fälle. Der für den zweiten Fall angegebene Key Path bezüglich des Attributs `_id` könnte z. B. verwendet werden, um in einer IndexedDB Datenbank Objekte zu speichern, denen zuvor durch eine auf Serverseite verwendete MongoDB ein Identifikator zugewiesen wurde – diese Anforderung finden Sie auch im Übungsprogramm zur vorliegenden Lerneinheit:

Quellcode

Object Store

```
001 /* erzeuge einen Object Store wahlweise mit eigener ID-Vergabe oder unter
002    Verwendung einer extern ermittelten ID */
003 var parameters = new Object();
004 if (useExternalIds) {
005    /* verwende als keyPath das _id Attribut, wie es z. B. durch MongoDB
006       gesetzt wird */
007    parameters.keyPath = "_id"
008 }
009 else {
010    parameters.autoIncrement = true
011 }
012 // hier wird die Store Erzeugung mit Parameterübergabe veranlasst
013 var objectStore = db.createObjectStore(currentStore, parameters);
```

Erstellung von Indizes

Auf einem auf diese Weise erzeugten Object Store Objekt haben Sie dann die Möglichkeit, weitere Einstellungen vorzunehmen. Beispielsweise können Sie für ein Attribut der darin enthaltenen Objekte einen *Index* setzen. Für den effizienten Zugriff auf die Inhalte eines Object Stores stellen Indizes eine Alternative zum Zugriff via eindeutigem Identifikator dar. Dies kann durchaus sinnvoll sein, da im Gegensatz zum Zugriff auf MongoDB in IndexedDB keine beliebigen Abfragebedingungen über den Attributen von Objekten formuliert werden können. Die Erstellung eines Index ist auch für Object Stores möglich, die bereits Daten enthalten, sie muss allerdings immer im Rahmen eines `onupgradeneeded` Callbacks ausgeführt werden. Ob der Index ein eindeutiger Bezeichner ist oder ob mehrere Objekte mit demselben Index-Wert assoziiert sein können, können Sie bei der Erstellung des Index ebenfalls angeben:

Quellcode

Nicht eindeutiger Index

```
001 // Setzen eines nicht eindeutigen Index bezüglich eines Attributs namens title
002 objectStore.createIndex("titleindex", "title", {
003    unique : false
004 });
```

Wurde eine Datenbank und die durch sie zu verwaltenden Object Stores auf dem hier gezeigten Weg erfolgreich erstellt, dann können Sie, wie oben gezeigt, das für den Zugriff zu verwendende `IDBDatabase` Objekt aus dem Argument des `onsuccess` Callbacks auslesen und in geeigneter Form Ihrer Anwendung für die Ausführung von CRUD Operationen zur Verfügung stellen.

3.2.2 CRUD Operationen auf Object Stores

Auf Object Stores existieren die folgenden CRUD-Operationen, die wir hier zusammenfassend nennen und unten anhand von Implementierungsbeispielen erläutern werden:

CRUD Operation	IndexedDB API
create	`add()`
read *read all*	`get()` `openCursor()`
update	`put()`
delete	`delete()`

Tab.: CRUD-Operationen auf Object Stores

Verallgemeinert werden die genannten Operationen auf einem Object Store `objectStore` wie folgt aufgerufen, wobei `arguments` hier für die jeweils zu übergebenden Argumente steht.

```
var request = objectStore.<crud-operation>(<arguments>);
```

Auf dem `request` Objekt werden dann, wie oben bereits anhand der `open()` Funktion gezeigt, die Callbacks für `onerror` und `onsuccess` gesetzt. Im Erfolgsfall kann das jeweilige Ergebnis der Operation auf dem `event` Objekt wahlweise über den Attributpfad `event.target.result` oder über den Pfad `request.result` direkt aus dem `request` Objekt ausgelesen werden – da das `success` Event jeweils bezüglich des ausgeführten Requests geworfen wird, sind beide Varianten äquivalent.

Transaktionen

Für die Ausführung von CRUD-Operationen auf IndexedDB ist jedoch zu beachten, dass diese grundsätzlich innerhalb einer Transaktion erfolgt. Dieses Konzept ist Ihnen vermutlich im Rahmen der Beschäftigung mit relationalen Datenbanken bereits begegnet. Es bezeichnet eine Menge von schreibenden Zugriffsoperationen auf einen Datenspeicher, die so gehandhabt werden, dass eine durch eine einzelne Operation bewirkte Datenmanipulation nur dann tatsächlich umgesetzt wird, wenn alle schreibenden Operationen innerhalb der betreffenden Transaktion erfolgreich ausgeführt werden können.

Umgekehrt besagt dies, dass überhaupt kein schreibender Zugriff erfolgt, falls eine der Operationen fehlschlägt bzw. dass etwaige bereits vorgenommene schreibende Zugriffe automatisch rückgängig gemacht werden – d. h. ohne implementierende Intervention des Entwicklers, dessen Anwendung die Datenbank nutzt. Transaktionen gewährleisten auf diese Weise, dass aus einer Verknüpfung mehrerer elementarer schreibender Operationen immer ein konsistenter Zustand der zugegriffenen Datenbestände resultiert, da sie bei Auftreten eines Fehlers verhindern, dass partielle Modifikationen der Daten, „die für sich alleine keinen Sinn ergeben“, Bestand haben.


Hinweis

Lebensweisheit zu Transaktionen

Um eine leicht moralisierende Lebensweisheit zu zitieren: *Transaktionen sorgen dafür, dass „wann immer A gesagt wird, auch B gesagt wird“, und „wenn B nicht gesagt werden kann,die Aussage von A rückgängig gemacht wird“.*

In IndexedDB werden Transaktionen durch Aufrufe der `transaction()` Funktion auf einem Datenbank-Objekt initiiert. Dieser Funktion werden die Namen der zuzugreifenden Objekt Stores als Argumente übergeben sowie ein weiteres Argument, das die Art des Zugriffs als wahlweise „`readwrite`“ oder „`readonly`“ kennzeichnet.

Transaktion

Intern wird bei Ausführung der initialen `open()` Funktion ggf. eine Transaktion im Modus „`versionchange`“ ausgeführt.

Die Anzeige der Zugriffsart ist insofern relevant, als IndexedDB zwar die gleichzeitige Ausführung mehrerer `readonly` Transaktionen erlaubt, aber zu jedem Zeitpunkt nur eine einzige `readwrite` Transaktion bearbeitet:

```
var transaction = db.transaction([store1, store2,...,storen], "readwrite")
```

create

Auf dem `transaction` Objekt kann u. a. eine Callback Funktion für `oncomplete` gesetzt werden, die bei erfolgreichem Abschluss mehrerer elementarer CRU- Operationen innerhalb der Transaktion aufgerufen wird. In den folgenden Beispielen, die jeweils einen elementaren Zugriff verwenden, machen wir davon aber keinen Gebrauch. Auch für diese einfachen Zugriffe ist jedoch zu beachten, dass der jeweils zu verwendende Object Store grundsätzlich über das `transaction` Objekt zugegriffen wird, und zwar mittels der Funktion `objectStore()`. Auf deren Rückgabewert können dann die oben erwähnten CRUD-Operationen ausgeführt werden. Nachfolgend wird dies für die Hinzufügung von Objekten zu einem Store am Beispiel eines Objekts `topicview` gezeigt:

Quellcode

Object Store Transaktion

```
001 // deklariere eine Transaktion bezüglich des Stores topicviews
002 var transaction = db.transaction(["topicviews"], "readwrite");
003
004 /* in einem oncomplete Handler auf transaction kann ein Callback gesetzt
005 werden, der bei Abschluss aller zur Transaktion gehörenden Zugriffe
006 aufgerufen wird */
007 transaction.oncomplete = function(event) { /* (...) */}
008
009 // für den Fall eines Fehles wird der onerror Callback aufgerufen
010 transaction.onerror = function(event) { /* (...) */}
011
012 // greife auf den Store über das transaction Objekt zu
013 var objectStore = transaction.objectStore("topicviews");
014
015 // rufe die add() Funktion auf und übergib das hinzuzufügende Objekt
016 var request = objectStore.add(topicview);
017
018 /* deklariere einen onsuccess Callback - diesem wird als Ergebnis die id
019 des Objekts übergeben */
020
021 request.onsuccess = function(event) {
022    var newObjectId = event.target.result;
023    // verwende die id wie gewünscht
024    /* (...) */
025 }
```

Der an den `onsuccess` Callback übergebene Wert ist hier der durch die Datenbank selbst generierte Identifikator, falls der Object Store im `autoIncrement` Modus betrieben wird. Falls er einen `keyPath` für einen bereits existierenden Identifikator verwendet, wird dessen Wert übergeben.

update

Unterschiede zwischen den beiden genannten Fällen bestehen bei der Ausführung von Update Operationen mittels `put()`. Hier wird im `autoIncrement` Modus die generierte `id` des Objekts sowie das zu aktualisierende Objekt selbst angegeben, während im `keyPath` Modus nur das Objekt übergeben wird – der eindeutige Identifikator muss für `autoIncrement` also nicht notwendigerweise auf dem Objekt selbst persistiert werden (in den Implementierungsbeispielen ist dies allerdings der Fall).

Beachten Sie für `put()` außerdem, dass, wie oben erwähnt, keine partiellen Updates durchgeführt werden können, sondern jeweils das gesamte zu aktualisierende Objekt – hier in Form des Werts der `update` Variable – übergeben werden muss. Wie in MongoDB ist bei Übergabe des Identifikators außerdem darauf zu achten, dass dieser mit dem korrekten Typ übergeben wird – aus diesem Grund führen wir unten für die angenommene `id` Variable die Funktion `parseInt()` aus:

```
001 /* für put() wird überprüft, ob externe ids verwendet werden oder nicht.
002 update bezeichnet das zu aktualisierende Objekt */
003 var request;
004 if (useExternalId) {
005    request = objectStore.put(update);
006 }
007 else {
008    // verwende den korrekten Typ für das id Argument
009    request = objectStore.put(update, parseInt(id));
010 }
011 request.onsuccess = function(event) {/* (...) */}
```

Angemerkt sei, dass `put()` grundsätzlich als „upsert“ ausgeführt wird, d. h. falls ein Objekt noch nicht existiert, wird es analog zur `add()` Methode hinzugefügt. Im Erfolgsfall wird dem entsprechend für den Fall externer Identifikatoren das zu aktualisierende Objekt als `request.result` gesetzt, für den Fall generierter `ids` die `id` des betreffenden und ggf. neu hinzugefügten Objekts.

delete

Die Ausführung der `delete()` Operation ist denkbar einfach. Beachtet werden muss auch hier lediglich der korrekte Typ der id, d. h. insbesondere bei durch IndexedDB generierten IDs ist die Übergabe eines Integer-Werts erforderlich. Ein Wert für `request.result` bzw. `event.target.result` wird hier im Erfolgsfall nicht gesetzt, d. h. es wird lediglich der `onsuccess` Event Handler aufgerufen:

```
001 // lösche ein Objekt mit numerischer id
002 var request = objectStore.delete (parseInt(id));
003
004 request.onsuccess = function(event) {/* (...) */}
```

read

Was das Auslesen von Objekten angeht, so muss hier der gezielte Zugriff auf ein einzelnes Objekt mittels dessen `id` vom Zugriff auf eine Menge von Objekten unterschieden werden. Ersterer erfolgt analog zu `delete()` durch Übergabe der id. Das ausgelesene Objekt, falls vorhanden, wird dann als Wert von `request.result` gesetzt:

```
001 // lies ein Objekt mit numerischer id aus
002 var request = objectStore.get(parseInt(id));
003
004 // greife im onsuccess Callback auf das ausgelesene Objekt zu
005 request.onsuccess = function(event) {
006    var object = event.target.result;
007    /* (...) */
008 }
```

Für den Zugriff auf eine Menge von Objekten wird ein Cursor Objekt verwendet, wie es auch in anderen APIs für Datenzugriff gebräuchlich ist.

Cursor Objekt
Z. B. das gleichnamige Objekt `android.database.Cursor` in Android oder `java.sql.ResultSet` in JDBC.

Dieses funktioniert wie ein `Iterator` in Java und zeigt jeweils auf ein Objekt, das sich in der Gesamtmenge der ausgelesenen Objekte befindet. Wie Sie unten sehen, stellt IndexedDB dieses als Key-Value Paar auf dem Cursor Objekt zur Verfügung, wobei der key als Wert den Identifikator des Objekts enthält. Um alle Objekte auszulesen muss der Cursor „nach vorne beweg“t werden. In IndexedDB wird dafür die Funktion `continue()` verwendet.

Etwas erklärungsbedürftig ist vermutlich die Verwendung des `onsuccess` Callbacks: dieser wird für jedes ausgelesene Objekt aufgerufen, wobei als Wert von `event.target.result` das Cursor Objekt selbst gesetzt wird. Auf diesem kann dann sowohl das betreffende Objekt mittels Zugriff auf value und ggf. key ausgelesen, als auch mittels `continue()` das Auslesen des nächsten Objekts veranlasst werden. Falls aber das letzte Element der ausgelesenen Objektmenge erreicht worden ist, führt der nochmalige Aufruf von `continue` dazu, dass `event.target.result` nicht gesetzt ist. Anhand der Überprüfung von `event.target.result` kann also der Abschluss der Auslese-Operation veranlasst werden:


Quellcode

Auslesen aller Objekte aus einem Object Store

```
001 /* Auslesen aller Objekte aus einem Object Store mittels Cursor und "
002 Sammeln" der Objekte in einem Array */
003 var allobjects = new Array();
004 var cursor = objectStore.openCursor();
005
006 /* onsuccess Callback: dieser wird für jedes ausgelesene Objekt und am Ende
007 des Auslesevorgangs aufgerufen */
008 cursor.onsuccess = function(event) {
009    // solange Objekte ausgelesen werden, ist event.target.result gesetzt
010    cursor = event.target.result;
011    if (cursor) {
012       // gib den key des Objekts aus
013       console.log("read object with key: " + cursor.key);
014       // füge das Objekt dem Array hinzu
015       allobjects.push(cursor.value);
016       // rücke den Cursor vor
017      cursor.continue();
018    } else {
019       // hier ist der Auslesevorgang abgeschlossen
020      /* (...) */
021    }
022 }
```

Cursor Objekte werden nicht nur verwendet, um alle Objekte aus einem Object Store auszulesen, sondern können auch in Verbindung mit einem Index genutzt werden. Hierfür wird mittels `index()` der betreffende Index aus dem Object Store ausgelesen und auf diesem die `openCursor()` Funktion aufgerufen. Die weitere Verwendung des von dieser Funktion zurück gegebenen Cursors erfolgt wie im vorangegangenen Beispiel.

Der `openCursor()` Funktion, die wir im obigen Beispiel verwendet haben, kann als optionales Argument eine Key Range übergeben werden. Damit ist es möglich, die Menge der auszulesenden Objekte in Abhängigkeit vom verwendeten Identifikator – sei es der eindeutige Identifikator des Object Store oder ein Index Attribut der Objekte – einzugrenzen. Hierfür stehen u. a. die Funktionen `upperBound()`, `lowerBound()`, `range()` und `only()` zur Verfügung, mittels derer nach unten oder oben offene Intervalle, beidseitig begrenze Intervalle bzw. ein einziger Wert als Range angegeben werden kann. Für Details diesbezüglich verweisen wir auf die Mozilla Dokumentation. Im nachfolgenden Bespiel lesen wir mittels eines Index alle Titel aus, die mit den Anfangsbuchstaben „A“ und „B“ beginnen - das Intervall hierfür ist zu lesen als: „`A ≤ key < C`“, d. h. die beiden booleschen Argumente drücken hier die Nicht-Exklusion bzw. die Exklusion der angegebenen Grenzwerte aus:


Quellcode

Greife auf Objekte über einen Index zu

```
001 // greife auf Objekte über einen Index zu
002 var index = objectStore.index("titleindex");
003
004 // gib den Wertebereich für den key an, der mit dem Index zugegriffen wird
005 var boundKeyRange = IDBKeyRange.bound("A", "C", false, true);
006
007 /* erzeuge einen Cursor bezüglich des Wertebereichs, um Objekte auf dem
008 Index auszulesen */
009 var cursor = index.openCursor(boundKeyRange);
010
011 /* gib den onsuccess callback an, der für jedes ausgelesene Objekt
012 aufgerufen wird */
013 cursor.onsuccess = function(event) {/* (...) */}
```

Einschränkungen

Als Abschluss unserer allgemeinen Darlegungen zur Verwendung von IndexedDB und unter Rückbezug auf die anderen hier behandelten Verfahren zur lokalen Datenspeicherung sei noch erwähnt, dass auch bezüglich der Größe von Datenbanken bzw. einzelner Einträge Einschränkungen existieren. Die Mozilla Dokumentation trifft hierzu die folgende allgemeine Aussage, die nicht auf einen einzelnen Browser bezogen ist:


Hinweis

IndexedDB databases size

"There isnt any limit on a single database items size. However there may be a limit on each IndexedDB databases size".

Was Firefox im speziellen angeht, so ist hier keine Obergrenze vorgesehen. Sollen allerdings Datenmengen von mehr als 50 MB gespeichert werden, ist ggf. die Rückbestätigung des Nutzers erforderlich. Für die praktische Verwendung von IndexedDB ergibt sich daraus die Einschränkung, dass z. B. Objekte, die Mediendaten aus Data URLs enthalten, nicht in IndexedDB gespeichert werden sollten. Empfiehlt es sich generell, Datenbanken nur für die Speicherung strukturierter Daten bzw. Metadaten zu verwenden, Medieninhalte selbst aber auf ein Dateisystem auszulagern, so ist eine solche Lösung bei ausschließlicher Verwendung von Web Technologien derzeit mangels Verfügbarkeit der oben angesprochenen `FileSystem` API noch nicht umsetzbar.

Eine Alternative dazu könnte tatsächlich aber die Verwendung eines geeigneten Caching Mechanismus sein, bei dem Medieninhalte selbst serverseitig gespeichert und clientseitig gecacht werden und die Metadaten bezüglich dieser Inhalte evtl. serverseitig, auf jeden Fall aber clientseitig in einer IndexedDB gespeichert werden. Eine solche Lösung würde zwar eine initiale Übertragung der Inhalte an die serverseitige Anwendung erfordern, wäre danach aber grundsätzlich offline-fähig. Die dafür noch erforderlichen Ausdrucksmittel werden in der folgenden, abschließenden, Lerneinheit vermittelt.

3.2.3 Anmerkung zu den Implementierungsbeispielen

Vielleicht ist Ihnen beim Betrachten der Beispiele zur Verwendung der IndexedDB API aufgefallen, dass die Anwendung der API im Detail zwar gewisse Eigenarten mit sich bringt, abgesehen davon aber ein weitgehend einheitliches Muster für einfache CRUD-Datenzugriffe angewendet werden kann. Neben den spezifischen Argumenten und Callbacks, die Sie für die einzelnen Operationen benötigen und hinsichtlich derer sich die Operationen voneinander unterscheiden, sind nur noch das Datenbank-Objekt und der Name des zuzugreifenden Objekt Stores erforderlich.

Mit den genannten Daten lassen sich also für beliebige Datenbanken und Object Stores elementare CRUD-Zugriffe bezüglich eines Stores Ihrer Wahl umsetzen, wie Sie sie für die Übungsaufgaben benötigen. Um die Verwendung von IndexedDB zu vereinfachen, beinhalten die Implementierungsbeispiele zur vorliegenden Lerneinheit daher in der Datei `ixdb.js` eine Bibliothek, die Ihnen entsprechend den vorstehenden Überlegungen generische Zugriffsoperationen auf eine IndexedDB Datenbank ermöglicht und auch die Erstellung der Datenbank vereinfacht. So steht für diese eine Funktion `openOrCreateSimpleDB()` zur Verfügung, der die folgenden Argumente übergeben werden können:

- Der Name der Datenbank
- Die gewünschte Version der Datenbank
- Ein Array von Strings, der die Namen der zu erzeugenden Object Stores enthält.
- Einen Array, der für jedes Element des Namens-Arrays ein Parameter-Objekt für die Erzeugung des Stores enthält.
- Eine Callback-Funktion, der bei erfolgreicher Erstellung bzw. beim Öffnen der Datenbank das Datenbank-Objekt übergeben wird.
- Optional einen Array von Funktionen, die jeweils nach Erzeugung eines Object Stores aus dem Namens-Array unter Übergabe des Namens sowie des Store Objekts aufgerufen werden und es z. B. erlauben, auf dem Store einen Index zu setzen.

Sie sind also selbst dafür verantwortlich, das im Erfolgs-Callback übergebene Datenbank- Objekt an geeigneter Stelle in Ihrer JavaScript Implementierung abzulegen. Unter Übergabe dieses Objekts können Sie dann aber die in folgende Tabelle gezeigten generischen CRUD-Operationen nutzen, um elementare Zugriffe auf den Object Stores auszuführen.

CRUD-Operation	CRUD Funktion und Argumente	Callback Argumente
create	`createObject(db, objectstore, object, onsuccess, onerror)`	das erstellte Objekt inklusive der ggf. durch die Datenbank zugewiesenen `_id`
read	`readObject(db, objectstore, id, onsuccess, onerror)`	das ausgelesene Objekt
read all	`readAllObjects(db, objectstore, onsuccess, onerror)`	einen Array mit den ausgelesenen Objekten
update	`updateObject(db, objectstore, id, object, onsuccess, onerror)`	`true` oder `false`
	`updateObjectWithKeypath(db, objectstore, id, object, onsuccess, onerror)`	`true` oder `false`
delete	`deleteObject(db, objectstore, id, onsuccess, onerror)`	`true` oder `false`

Die Argumente `db`, `objectstore` und `id` bezeichnen das Datenbank-Objekt bezüglich dessen der Zugriff erfolgen soll, den Namen des zuzugreifenden Object Store sowie den eindeutigen Identifikator eines zuzugreifenden Objekts. `object` ist jeweils – auch für *update* – das zu speichernde Objekt.

Da Sie außerdem Zugriff auf das Datenbank-Objekt selbst haben, steht es Ihnen darüber hinaus frei, komplexere Zugriffe mit Transaktionen oder das Auslesen von Daten via Indizes selbst zu implementieren.

Zusammenfassung

- Webanwendungen gelten als weniger vertrauenswert als installierbare Anwendungen und unterliegen daher spürbaren Restriktionen. Browser versuchen zu verhindern, dass Webanwendungen unkontrolliert auf lokale Daten bzw. auf Daten anderer Webanwendungen zugreifen können.
- Ein Merkmal, das Webanwendungen im Hinblick auf die Absicherung gegenüber unbefugten Zugriffen kennzeichnet, ist die Delegierung von Zugriffen an den Nutzer.
- In Bezug auf Webanwendungen bezeichnet Caching die lokale Speicherung der Inhalte von HTTP Responses durch einen Client – Caching kann jedoch auch durch einen Proxy-Server vorgenommen werden.
- Für die clientseitige Speicherung von Daten werden in WebStorage zwei verschiedene Ebenen angenommen, die als *Session Storage* bzw. *Local Storage* bezeichnet werden.
- Im Gegensatz zum *Session Storage* dient die Ebene des *Local Strorage* der dauerhaften Speicherung von Daten, die mit einer Ursprungsdomain assoziiert sind.
- Durch die Beschränkung auf Strings als Datentyp für speicherbare Werte, eignet sich Local Storage nur eingeschränkt zur lokalen Datenspeicherung.
- Zur lokalen Speicherung von Instanzen komplexer Datentypen stellt IndexedDB geeignete Ausdrucksmittel bereit.

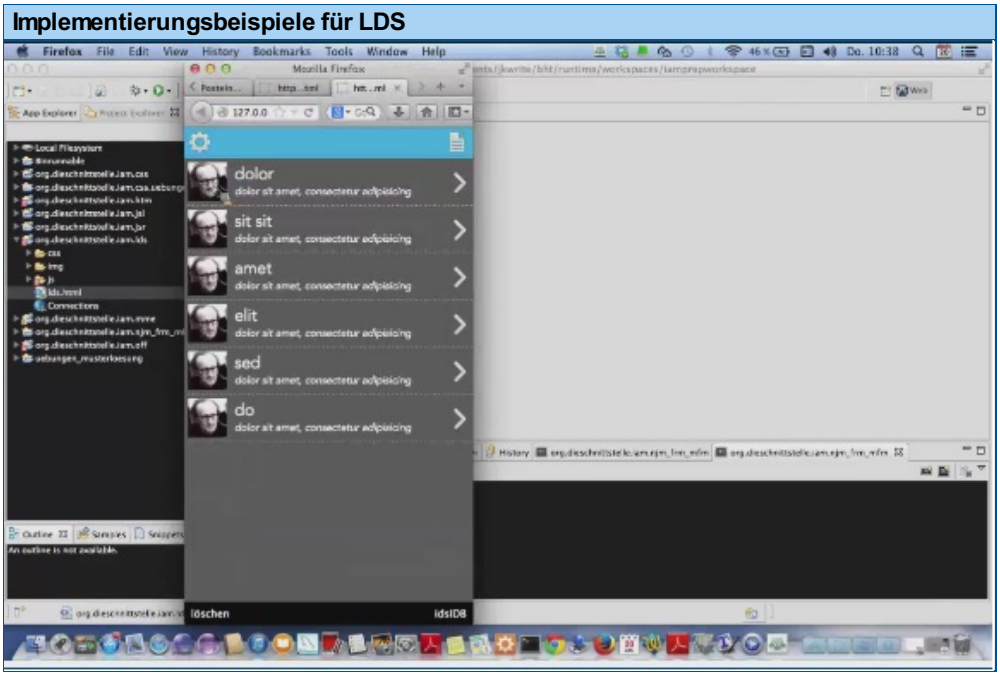
Sie sind am Ende dieser Lerneinheit angelangt. Auf den folgenden Seiten finden Sie noch Übungen.

Übungen

Im Projekt `org.dieschnittstelle.iam.lds` finden Sie die Implementierungsbeispiele für die vorliegende Lerneinheit. Die prüfungsverbindlichen Übungen und deren Bepunktung werden durch die jeweiligen Lehrenden festgelegt.

Film

Implementierungsbeispiele für LDS



© Beuth Hochschule Berlin - Dauer: 04:25 Min. - Streaming Media 8 MB


Formulieren

Übung LDS-01

Fragen zur Beispielanwendung

Die Error-Meldungen, die Ihnen Aptana evtl. für die Datei `js/lib/ixdb.js` anzeigt, können Sie auf Defizite des JavaScript Editors zurückführen.

ListviewViewController.js/showItemDialog(), mwf.css

- Wie wird bei Einblenden eines Popup-Dialogs der Übergang der Listenansicht in den Hintergrund bewirkt?
- Welche Ausdrucksmittel von CSS und JavaScript werden hierfür eingesetzt?

IListviewViewController.js/addNewItemViewToListView()

- Wie werden die einzelnen Listenelemente aufgebaut und mit den darzustellenden Daten befüllt?
- Ändern Sie die Wertzuweisung für `newItemView` wie folgt:
  ```
  var newItemView = itemView;
  ```
- Laden Sie die Ansicht neu. Wie hat sich die Darstellung geändert und weshalb?
- Was können Sie daraus hinsichtlich der Funktionalität von `cloneNode()` folgern?

`ListviewViewController.js/deleteItem()/updateItem()`

- Wie wird das aus der Listenansicht zu entfernende bzw. zu aktualisierende Element identifiziert?
- Wo wird die Voraussetzung dafür geschaffen, dass diese Identifikation möglich ist?

Local Storage

- Öffnen Sie die Beispielanwendung, geben Sie `localStorage` auf der Console in Firebug ein und führen Sie `Run` aus.
- Welche Inhalte werden Ihnen für Ihr `localStorage` angezeigt?
- Falls Inhalte aus anderen Beispielanwendungen enthalten sind, weshalb ist dies der Fall?

`lib/ixdb.js/readAllObjects()`

- Wie wird der Array der ausgelesenen Objekte aufgebaut und wann erfolgt deren Darstellung in der Listenansicht?
- Welche Änderung müsste vorgenommen werden, um die Objekte einzeln unmittelbar nach Auslesen eines Objekts darzustellen?

`DataItemCRUDOperations.js/readAllItems()`

- Weshalb wird in der abschließenden for-Schleife die Variable `countdown` verwendet, um den Abschluss der Iteration zu detektieren?
- Rufen Sie die Logmeldung “all objects have been read out!” alternativ nach Abschluss der for-Schleife auf und schauen Sie im Log, wann die Ausgabe der Meldung erfolgt.
- Weshalb wird nun zuerst diese Logmeldung ausgegeben und erst danach die Meldungen bezüglich der ausgelesenen Objekte?

Bearbeitungszeit: 45 Minuten

Programmieren

Übung LDS-02

Verwendung von IndexedDB f´üur Topicview und Imgbox

Aufgabe

Erweitern Sie die in den vorangegangenen Aufgaben entwickelte Implementierung durch einen lokalen Datenspeicher mit IndexedDB und ermöglichen Sie eine (ansatzweise) Synchronisierung von server-seitiger und lokaler Datenhaltung.

Anforderungen

1. Erstellen Sie eine alternative Implementierung aller bisher verwendeten CRUD Operationen für Topicview und Imgbox, bei der anstelle des Zugriff auf eine server-seitige MongoDB Datenbank eine lokale IndexedDB Datenbank verwendet wird.
2. Als Identifikatoren sollen durch Ihre Anwendung zugewiesene IDs und nicht die IDs von IndexedDB verwendet werden.
3. Erstellen Sie eine weitere Implementierung der CRUD Operationen für Topicview und Imgbox, die die lokale Implementierung aus **Anforderung 1** sowie die in den vorangegangenen Aufgaben genutzte Implementierung mit server-seitigem Zugriff verwendet. Wird eine schreibende CRUD Operation server-seitig erfolgreich ausgefüuhrt, soll die betreffende Operation auch lokal ausgeführt werden. Als IDs für IndexedDB sollen die IDs aus dem server-seitigen Zugriff genutzt werden.
4. Ermöglichen Sie in Ihrer Anwendung das "Umschalten" zwischen den gemäß **Anforderung 1** und **Anforderung 3** entwickelten lokalen bzw. "synchronisierten" Datenzugriffsoperationen und der bisher verwendeten Implementierung mit ausschließlich remote Datenzugriff.
5. Falls **Anforderung 4** nicht umgesetzt wird, wird für **Anforderung 1-Anforderung 3** maximal die Hälfte der genannten Punktzahl vergeben.

Bearbeitungshinweise

- **Anforderung 1-Anforderung 3:**Für die Umsetzung der lokalen und synchronisierten Datenzugriffsoperationen können Sie die Codegerüste `TopicviewCRUDOperationsLocal.js` und `TopicviewCRUDOperationsSynced.js` verwenden. In `TopicviewCRUDOperationsLocal.js` ist bereits die Verwendung der Bibliothek `lib/indexeddb.js` vorgesehen, die generische Funktionen für den Zugriff auf IndexedDB bereit stellt.
- **Anforderung 3:** Ein Beispiel für die generische Umsetzung dieser Anforderung können Sie der `RemoteMasterSyncedCRUDOperationsImpl.js` Implementierung aus OFF entnehmen.
- **Anforderung 4:** Diese Funktionalität ist in den aktuellen Implementierungsbeispielen bereits umgesetzt.

Bearbeitungszeit: 90 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 LDS-03

Lokale Datenspeicherung allgemein

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

1. Nennen Sie 4 verschiedene Möglichkeiten zur clientseitigen Datenspeicherung, die von Webanwendungen genutzt werden können.
2. Nennen Sie 3 Möglichkeiten zur Eindämmung des Sicherheitsrisikos beim Zugriff auf lokale Daten aus Webanwendungen?
3. Was besagt die *Same Origin Policy* für Webanwendungen und wo kommt sie im Bereich der lokalen Datenspeicherung zum Einsatz?

Bearbeitungszeit: 10 Minuten


Formulieren

Übung LDS-04

Caching

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

1. Wozu dient Caching?
2. Welche Ausdrucksmittel des HTTP-Protokolls werden für Caching genutzt?
3. Welche beiden Ansätze können in Bezug auf Caching voneinander unterschieden werden?

Bearbeitungszeit: 10 Minuten


Formulieren

Übung LDS-05

Cookies

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

1. Wozu dienen Cookies?
2. Welche Ausdrucksmittel des HTTP-Protokolls werden für Cookies verwendet?
3. Welche Einschränkungen bestehen bezüglich der Verwendung von Cookies in JavaScript?

Bearbeitungszeit: 10 Minuten


Formulieren

Übung LDS-06

Web Storage

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

1. Was ist ein Browsing Context?
2. Was ist der Unterschied zwischen `sessionStorage` und `localStorage`?
3. Was für eine Datenstruktur liegt der `Storage` API zugrunde?
4. Was zeigt das `storage` Event an und wozu kann es verwendet werden?
5. Welche Einschränkung bringt `Storage` bezüglich der Speicherung von Objekten mit sich und was ist dafür ggf. erforderlich?

Bearbeitungszeit: 15 Minuten


Formulieren

Übung LDS-07

IndexedDB

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

1. Was ist IndexedDB?
2. Was sind Object Stores in IndexedDB?
3. Was brauchen Sie, um das Ergebnis einer CRUD-Operation in IndexedDB weiter verarbeiten zu können?
4. Was ist der Vorteil der Verwendung von Transaktionen in IndexedDB?
5. Wozu dient ein Cursor in IndexedDB und mit welchem Konstrukt aus Java ist seine Handhabung vergleichbar?

Bearbeitungszeit: 15 Minuten