

Entwicklung und Realisierung einer progressiven Webapplikation zur Datenaufnahme und Anzeige

Projektbericht

Martin Arendt, Niklas Herz, Ken Madlehn, Eduard Götz

FH Bielefeld

Inhaltsverzeichnis

1. Einleitung
2. Vorstellung der Gruppenmitglieder und ihre Rollen und Aufgaben im Team
3. Theoretische Grundlagen progressiver Webanwendungen
4. Offlinefunktion des Dateiuploads
5. Serviceworker
6. Visualisierungskomponente
7. Installationshinweise
8. Ausblick
9. Quellenverzeichnis

Einleitung

Motivation

Im Rahmen des Moduls Webengineering sollte an dem bestehenden *SmartMonitoring-System* gearbeitet werden.

Schwerpunkt sollte dabei sein, das SmartMonitoring als **progressive Webapplikation** verwenden zu können. Dies bedeutet konkret, dass man das System auch als App verwenden kann. Dazu zählen von Apps bekannte Funktionalitäten wie das Installieren der App auf das Gerät sowie Offlinefunktionalitäten.

Der Gedanke dahinter ist die Verwendbarkeit unterwegs. Man sollte falls man unterwegs Daten gemessen hat, diese leicht und intuitiv durch das mobile Gerät hochladen können. Auch wenn man keine Internetverbindung hat. Dies schließt auch ein, dass bisher besuchte Seiten und aufgerufene Daten offline angezeigt werden können.

Themenbeschreibung

Die Anforderungen beschreiben also entsprechend die Nutzung des Smart-Monitoring-System als progressive Webapplikation mit nativem Anwendungsgefühl auf mobilen Geräten.

Ein weitere großer Teil der progressiven Webapplikation sollte der Datenupload sein. Zusätzlich zu den vorhandenen Funktionen, sollte dieser nun auch offline funktionieren. Das bedeutet, dass man ohne Internetverbindung jederzeit Dateien zum Upload bereitstellen kann. Diese sollten dann automatisch hochgeladen werden sobald eine Internetverbindung verfügbar ist.

Darüber hinaus sollte eine SWAC- (*SmartWebApplicationComponents*) Komponente zur Datenanzeige bzw. Datenvisualisierung aus dem schon vorhandenen Code entwickelt werden. Diese Komponente sollte ebenfalls offline funktionieren und bei einer Internetverbindung wieder aktuelle Daten anzeigen. Die Daten sollten in Form eines Thermometers und Hygrometers visualisiert und angezeigt werden. Auf einer Anzeigeseite in der Smart-Monitoring Webapplikation sollten dann Daten zu einem beobachteten Objekt über diese Diagramme visualisiert dargestellt werden.

Vorstellung der Gruppenmitglieder und ihre Rollen und Aufgaben im Team

Unsere Gruppe besteht aus vier Gruppenmitgliedern, die folgende Rollen und Aufgabenbereiche haben:

Gruppenmitglied	Aufgabenbereich
Niklas Herz	<ul style="list-style-type: none">- Projektleiter- Planung des Projektablaufs (Meilensteine etc.)- Planung und Implementierung des Dateiuploads mit Locale Storage und IndexedDB
Ken Madlehn	<ul style="list-style-type: none">- Verwaltung der Repositories auf GitLab- Verwaltung der gruppeninternen Cloud- Recherche nützliche Frameworks für das Projekt- Projekt progressiv machen- Visuelle Aufwertungen der Uploadfunktion
Martin Arendt	<ul style="list-style-type: none">- Planung und Realisierung des Service Workers- Precaching-Strategien entwickeln
Eduard Götz	<ul style="list-style-type: none">- Erweiterung der Visualisierungskomponente- Diagramme dynamisch anzeigen lassen

Die Verteilung der Aufgabengebiete innerhalb der Gruppe war nicht von Anfang an fest. Beispielsweise hat Niklas Herz in den ersten beiden Wochen des Projekts angefangen den Service Worker zu entwickeln während Martin Arendt an der Uploadfunktion gearbeitet hat. Da beide Gruppenmitglieder in ihrem Aufgabenbereich Probleme hatten, haben sie diese getauscht, was sich als gute Entscheidung herausstellte.

Ebenso waren die Vorkenntnisse der Gruppenmitglieder nicht identisch. Ken Madlehn hat vor Anfang des Semesters Technische Informatik in Lemgo studiert und deswegen noch nicht das Modul Webbasierte Anwendungen absolviert, in welchem wir Service Worker bereits behandelt haben. Seine Kenntnisse in JavaScript waren jedoch fortgeschritten, da er bei seiner Arbeit häufig JavaScript benutzte. Deshalb haben wir die Aufgaben so aufgeteilt, dass die Arbeiten mit dem Service Worker größtenteils von Martin Arendt und Niklas Herz erledigt wurden und Ken Madlehn und Eduard Götz sich um die Implementierungen mit JavaScript sowie um die Recherche welche Frameworks nützlich sind erledigten.

Des Weiteren brauchten wir einen Projektleiter, da wir uns intern organisieren mussten und jemand dafür die Treffen und gruppeninternen Abgabefristen planen sollte. Wir haben für diesen

Zweck Niklas Herz zum Projektleiter ernannt, da dieser schon in den Modulen Softwareprojektmanagement im 3. Semester und im Softwareprojekt im 4. Semester Erfahrungen als Projektleiter bzw. stellvertretener Projektleiter gesammelt hat. Unsere Gruppe war ansonsten jedoch wenig hierarchisch und die meisten Entscheidungen wie zum Beispiel die Verteilung der Aufgaben wurden demokratisch und gemeinsam entschieden.

Theoretische Grundlagen Progressiver Webanwendungen (PWA)

Begriff

Der Begriff **Progressive WebApp**, **Progressive Webanwendung** oder auch **Progressive Web Application** beschreiben Web Apps, im deutschen auch Webanwendungen genannt.

Wie eine gewöhnliche Internetseite besitzt eine progressive Webanwendung eine Benutzeroberfläche. Diese Oberflächen werden durch einen Browser dargestellt. Für die Darstellung einer solchen Oberfläche muss die Webanwendung dem Browser durch Stylesheet-Sprachen mitteilen, wie einzelne Elemente dargestellt werden sollen. Den Aufbau einer Seite beschreibt eine Webanwendung mithilfe von Auszeichnungssprachen wie HTML. Für eine verbesserte Benutzererfahrung im Bereich der Oberflächengestaltung können Animationen verwendet werden. Rudimentäre Verbesserungen dieser Art können auch mit Stylesheet-Sprachen umgesetzt werden, wobei im professionellen Umfeld hierfür auf Scriptsprachen zurückgegriffen wie JavaScript zurückgegriffen wird.

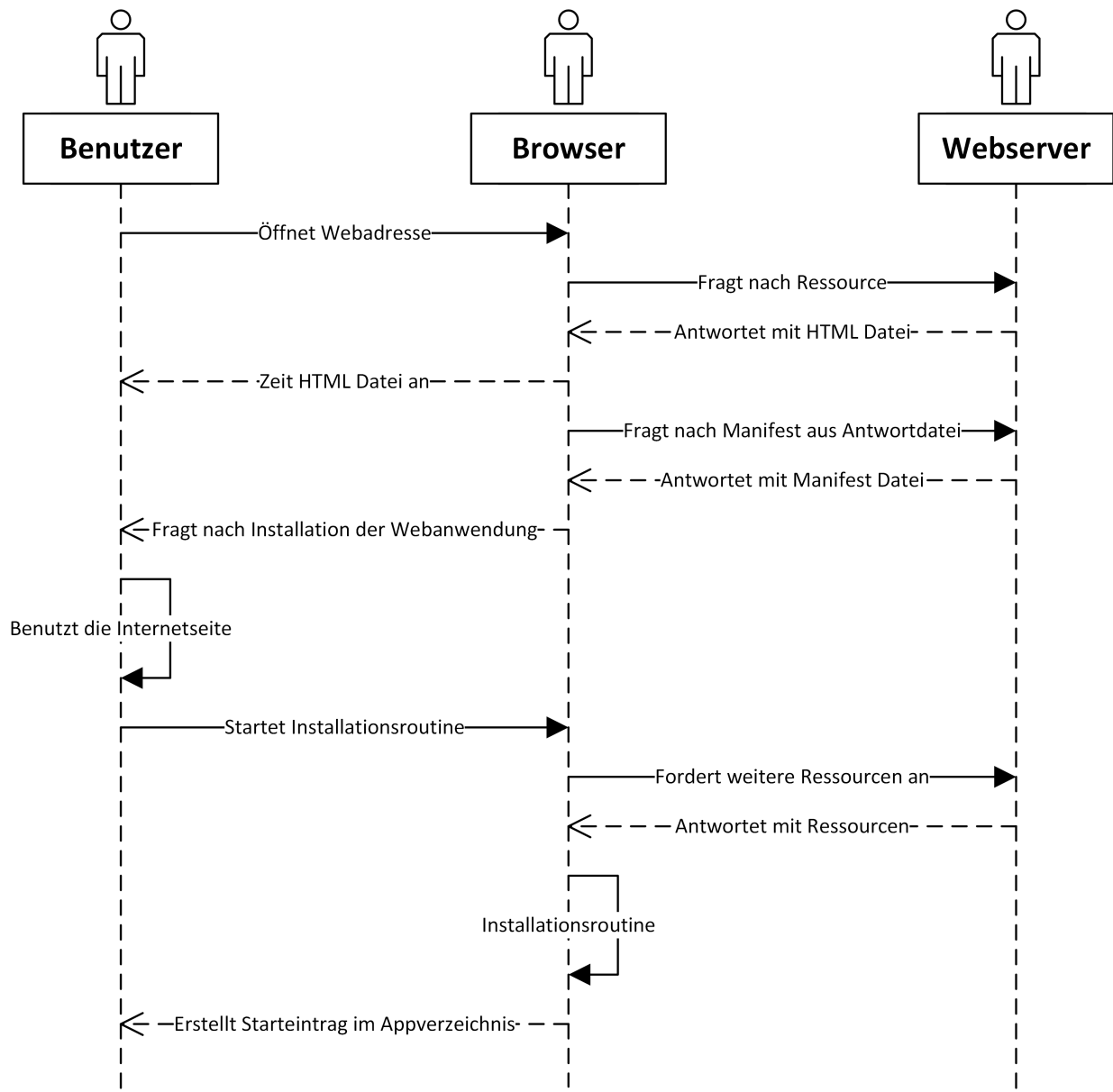
Neben den elementaren Bausteinen aus der Webentwicklung zeichnen sich progressive Webanwendungen durch andere Besonderheiten aus. Zum einen ist der Datenverkehr einer solchen Anwendung stets per HTTPS verschlüsselt, zum anderen ist sie in der Lage offline zu arbeiten.

Damit eine solche Funktionalität gewährleistet werden kann, müssen diese Anwendungen auf eine Vielzahl von Schnittstellen moderner Browser zurückgreifen, um einem Benutzer dieser Anwendung die Usability einer nativen Anwendung zu bieten.

Neben der Offlinefähigkeit lassen sich diese Anwendungen auch nicht nur hervorragend bedienen, sondern sind auch über den Browser auf einem Gerät installierbar.

Funktionsweise

Das erstmalige Starten einer progressiven Webanwendung geschieht über einen modernen Browser des Benutzers. Wie bei dem Öffnen einer regulären Internetseite muss die Webadresse der progressiven Webanwendung in die Adressleiste des Browsers eingegeben, und die Seite angefragt werden.



Nach dem Anfragen der Seite durch den Benutzer am Browser versucht dieser den Webserver zu erreichen. Dazu wird zuerst der Domainname über einen DNS Server zu einer IP-Adresse übersetzt. Über diese Adresse und der angefragten URI versucht der Browser den Webserver zu kontaktieren und die geforderte Ressource zu bekommen.

Der Webserver antwortet dem Browser mit einer HTML Datei, welche zunächst beim Browser den Rendervorgang einleitet. Während die Seite aufgebaut und dem Benutzer durch den Browser dargestellt wird, versucht der Browser zeitgleich weitere Ressourcen die zum Aufbau der Seite notwendig sind, anzufragen. Diese Abhängigkeiten sind in der HTML Datei beschrieben und beinhalten Dateien für das Styling der Oberfläche in Form von CSS Dateien und weitere Dateien für Animationen und Bibliotheken für weitere Funktionalitäten in Form von JavaScript Dateien.

Eine dieser Referenzierten Dateien ist bei einer progressiven Webanwendung die Manifestdatei, welche der Browser in diesem Prozess auch anfragt.

Wenn diese Datei vollständig und konsistent ist, zeigt der Browser dem Benutzer an, dass die aktuell angezeigte Seite auch als App installiert werden kann.

Das ist allerdings nur eine Option für den Benutzer die Anwendung direkt zu starten, und kann sich dadurch den Umweg über den Browser sparen. Sobald der Benutzer dem Browser mitteilt, dass die Installationsroutine gestartet werden kann, fordert der Browser am Webserver weitere Ressourcen an, wie zum Beispiel einige der Icons, die in der Manifestdatei referenziert sind.

Danach wird die Installationsroutine gestartet, wobei die Nachgeladenen und aktuellen Ressourcen auf dem Endgerät für eine spätere Nutzung vorgehalten werden. Dies geschieht über einen ServiceWorker. Eine genaue Beschreibung über die Funktionsweise ist weiter unten in diesem Dokument zu finden.

Außerdem wird ein Starteintrag der Anwendung mit einem der Icons aus der Manifestdatei als App-Icon und dem `short_name` aus der genannten Datei auf dem Startbildschirm des Benutzers erstellt.


```
1  {
2    "name": "SCL SmartMonitoring",
3    "short_name": "SCL",
4    "display": "standalone",
5    "start_url": "sites/login.html",
6    "background_color": "#34495e",
7    "theme_color": "#8e44ad",
8    "icons": [
9      {
10       "src": "img/favicon.png",
11       "type": "image/png",
12       "sizes": "48x48"
13     }, {
14       "src": "img/favicon.png",
15       "type": "image/png",
16       "sizes": "72x72"
17     }, {
18       "src": "img/favicon.png",
19       "type": "image/png",
20       "sizes": "96x96"
21     }, {
22       "src": "img/favicon.png",
23       "type": "image/png",
24       "sizes": "144x144"
25     }, {
26       "src": "img/favicon.png",
27       "type": "image/png",
28       "sizes": "168x168"
29     }, {
30       "src": "img/favicon.png",
31       "type": "image/png",
32       "sizes": "192x192"
33     }, {
34       "src": "img/favicon.png",
35       "type": "image/png",
36       "sizes": "256x256"
37     }, {
38       "src": "img/favicon.png",
39       "type": "image/png",
40       "sizes": "512x512"
41     }
42   ]
43 }
```

In dem Ausschnitt oben ist der Inhalt einer Manifestdatei zu sehen.

Attributname	Attributbeschreibung
name	Der vollständige Name der progressiven Webanwendung
short_name	Der kurze Name der Webanwendung. Wird in dem App-Launcher und in der Liste mit aktiven Anwendungen angezeigt
display	standalone : Lässt die Webanwendung wie eine native Anwendung auf einem mobilen Gerät darstellen. Die Anwendung bekommt ein eigenes Fenster, einen eigenen Namen, Starticon und optionale Oberflächen-Elemente. Weitere Möglichkeiten: fullscreen , minimal-ui und browser
start_url	Einstiegspunkt der progressiven Webanwendung. Nach dem Installieren und erstmaligem Öffnen der App wird diese URL geöffnet
background_color	Beim Starten der App können Verzögerungen durch das Laden von Ressourcen entstehen. Dabei wird ein Ladebildschirm generiert, der ein passendes Icon aus der Liste in der Manifestdatei enthält. Der Hintergrund wird mit der hier angegebenen Farbe gefüllt
theme_color	Hintergrundfarbe für die Anzeige im App-Manager
icons	Liste mit Icons für die Anzeige im App-Launcher, App-Manager, Titelleiste, etc. Hierfür sollten möglichst viele unterschiedliche Größen angeboten werden, auch wenn die selbe Datei referenziert wird. Dadurch können iOS Geräte dazu gebracht werden, Icons mit einer zu niedrigen Auflösung trotzdem anzuzeigen

Unterstützte Browser

Eine Vollständige Unterstützung aller Schnittstellen die im W3C definiert sind, bietet zum aktuellen Zeitpunkt keiner der bekannten Browser.

Die beste Unterstützung für solche Anwendungen bietet der Chrome Browser von Google, der mit allen in diesem Projekt verwendeten Technologien umgehen kann. Kleinere Einschränkungen bei der Nutzung hat der Browser Firefox von Mozilla. Hier lassen sich die Webanwendungen nicht wie bei Chrome über die Aufforderung oder das Menü installieren. Von der Nutzung von Microsoft Edge und dem Safari Browser wird abgeraten, da die Linux-Leute es nicht getestet haben ;) Es ist jedoch davon auszugehen, dass der Edge Browser in Zukunft auch mit diesen Anwendungen wie der Chrome Browser umgehen kann, da dieser einige Module aus Chrome bekommen wird.

Responsive WebApp vs. Progressive WebApp

Eine responsive WebApp ist eine Anwendung, die sich den Gegebenheiten eines Gerätes anpasst. Hier werden Prinzipien wie *mobile first* angewandt, bei denen eine Oberfläche zuerst für mobile Geräte optimiert wird. Danach wird erst die Kompatibilität der Oberfläche für Geräte mit großen Bildschirmen hergestellt. Für solche Anwendungen werden häufig Frameworks wie

Bootstrap verwendet, die neben ihren eigentlichen Designeigenschaften durch ihre Stylesheets auch die Usability durch Animationen in Form von JavaScript Code verbessern. Die Verbesserung der Usability kommt dadurch zustande, dass größere Elemente einer Internetseite an einer definierten Stelle umgebrochen werden. Dadurch werden diese Elemente die logisch gesehen nebeneinander gehören, untereinander dargestellt. Um untereinander stehende Elemente auch logisch voneinander trennen zu können, werden auf Techniken wie dem Grid-Design zurückgegriffen.

Progressive Webanwendungen sind von sich aus schon responsive wie oben beschrieben. Sie unterscheiden sich allerdings in der Art, wie sie auf mobilen Geräten genutzt werden können. Progressive Webanwendungen lassen sich über einen Webbrowser auf dem Zielgerät installieren. Der Vorgang ist vergleichbar mit dem Installieren einer Anwendung aus einem Appstore. Nach der Installation hat eine progressive Webanwendung allerdings mehr Möglichkeiten mit dem Benutzer durch das Endgerät Informationen darstellen zu können. Eine dieser Möglichkeiten ist das Benutzen der Webanwendung ohne aktive Internetverbindung. Eine weitere andere Möglichkeit ist das Senden von Push-Nachrichten, bei denen der Benutzer die Webanwendung nicht die ganze Zeit lang aktiv halten muss, um Nachrichten bekommen zu können. Das schont die Nerven des Benutzers und den Akku des Endgerätes.

Offlinefunktion des Dateiuploads

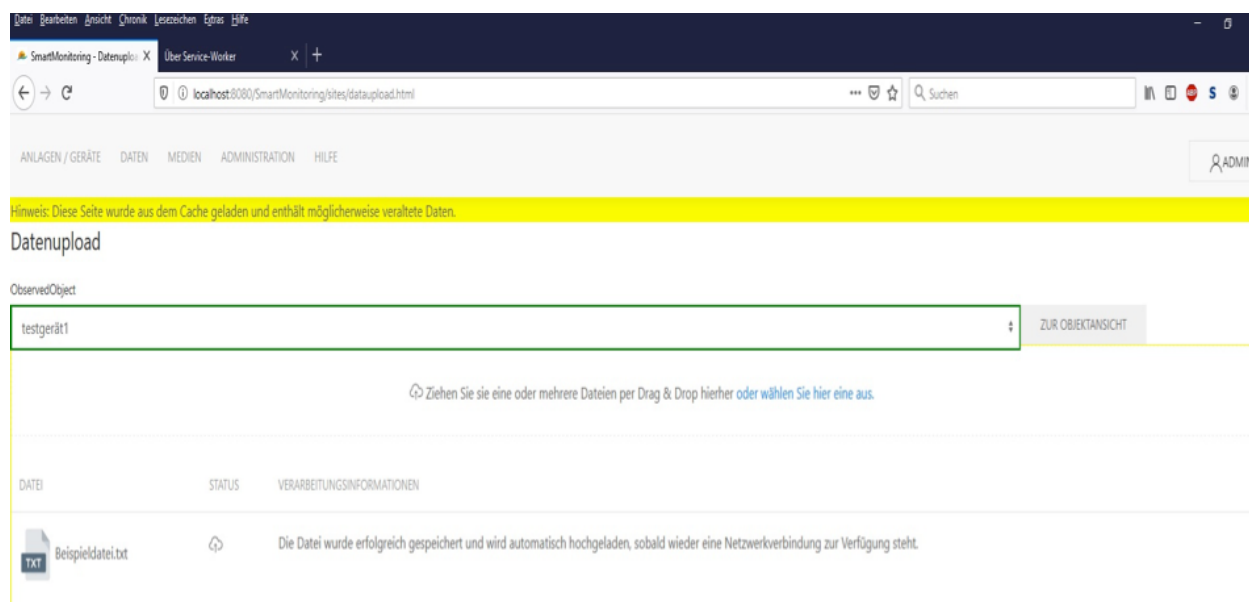
Problemstellung

Zweck der Aufgabe war es, dem Benutzer auch ohne Netzwerkverbindung das Hochladen von Dateien zu ermöglichen. Zuvor war das nicht möglich, da zum einen die HTML-Seite zum Dateiupload ohne Netzwerkverbindung nicht aufgerufen werden konnte und zum anderen, weil der Dateiupload ohne Netzwerkverbindung auch nicht durchgeführt werden kann.

Dadurch ist es den Benutzern nicht möglich zu jeder Zeit und an jedem Ort Dateien zum Server zu schicken. Somit muss der Benutzer warten bis er wieder eine Netzwerkverbindung hat und es kann passieren, dass das nachträgliche Hochladen der Dateien in Vergessenheit gerät.

Dateien im Browser zwischenspeichern

Durch die Implementierung eines Service Workers wird die Seite für den Datenupload im Cache gespeichert. Der Benutzer wird durch ein Banner unter der Menüleiste darüber informiert, dass er grade keine Internetverbindung hat. Wenn er nun eine Datei hochlädt, wird diese zunächst lokal im Browser gespeichert.



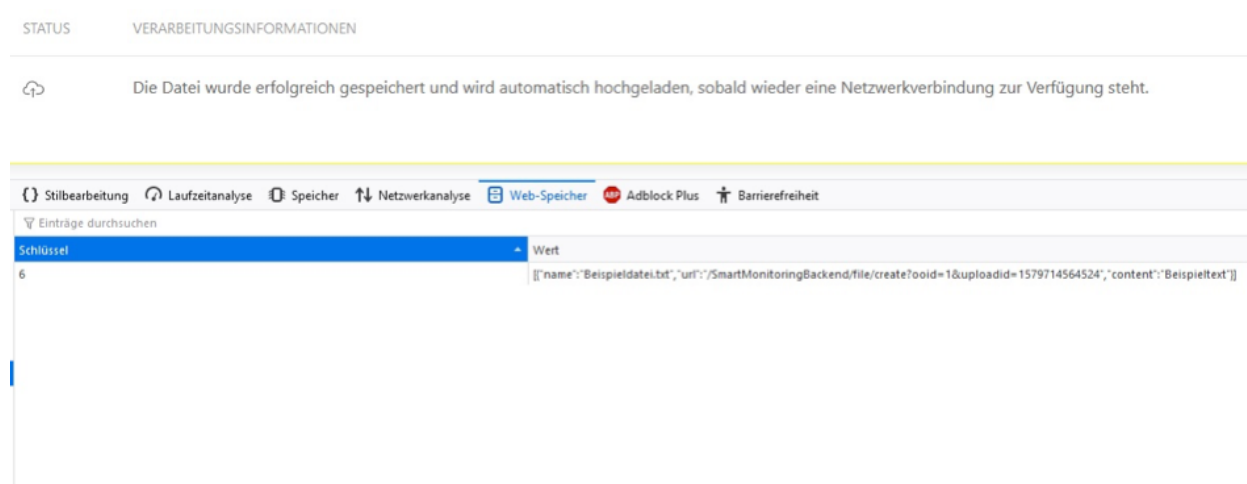
Zum Zwischenspeichern von Daten in einem Browser hat man als Entwickler mehrere Möglichkeiten. Zum einen gibt es die IndexedDB, welche von jedem gängigen Browser unterstützt wird. Diese hat die größte Speicherkapazität und beste Performance. Des Weiteren kann man noch den Local Storage oder den Cache benutzen. Diese sind aus Sicht eines Programmierers einfacher zu benutzen, bringen aber auch viele Nachteile mit sich. Beispielsweise ist der Speicherplatz kleiner und das Risiko, dass die Daten gelöscht werden ist höher, da viele Benutzer regelmäßig ihren Cache leeren.

Wir haben uns dafür entschieden vorzugsweise die IndexedDB zu benutzen und nur falls diese nicht vom Browser unterstützt wird den Dateiinhalte stattdessen in den Local Storage zu

speichern. Der nachfolgende Code zeigt die Umsetzung dieser Hierarchie.

```
1
2  if (!window.indexedDB) {
3    console.log("Your browser doesn't support a stable version of IndexedDB.
4    Such and such feature will not be available. Using Cache instead.");
5    let reader = new FileReader();
6    let fileobject = {};
7    let counter = 0;
8    if (localStorage.getItem("fileCounter") !== null)
9    {
10   counter = localStorage.getItem("fileCounter");
11   }
12   let strname = "file" + counter;
13   reader.onload = function (file) {
14     let filename = SWAC_upload.files[0].name;
15     fileobject.content = reader.result;
16     fileobject.url = uploadURL;
17     fileobject.name = filename;
18     localStorage.setItem(strname, JSON.stringify(fileobject));
19     localStorage.setItem("fileCounter", (parseInt(counter) + 1));
20   };
21   reader.readAsText(SWAC_upload.files[0]);
22   } else {
23     let reader = new FileReader();
24     let fileContent;
25
26     reader.onload = function (file) {
27       let filename = SWAC_upload.files[0].name;
28       fileContent = [{name: filename, url: uploadURL, content: reader.result}];
29       let db;
30       let request = indexedDB.open("UploadDateien", 1);
31       request.onerror = function (event) {
32         alert("Fehler beim Öffnen der Datenbank,
33         möglicherweise müssen Sie den Zugriff manuell erlauben.");
34       };
35       request.onsuccess = function (event) {
36         db = event.target.result;
37
38         let transaction = db.transaction(["Dateien"], "readwrite");
39         transaction.oncomplete = function (event) {
40           console.log("Datei erfolgreich in IndexedDB gespeichert");
41         };
42         transaction.onabort = function (event) {
43           console.log("Fehler beim Speichern in der IndexedDB");
44         };
45         let objectStore = transaction.objectStore("Dateien");
46         // Add uploadId to uploadURL
47
48         let request = objectStore.add(fileContent);
49         request.onsuccess = function (event) {
50
51         };
52
53       };
54       //Info: "on upgrade needed" wird vor "on success" ausgeführt
55       //Nur in "on upgrade needed" darf die Struktur der DB geändert werden
```

Nach Ausführung dieses Codes ist der Dateiinhalt entweder im Local Storage oder in der IndexedDB. Je nach Anwendungsfall werden jedoch unterschiedlich viele Daten abgespeichert. Sowohl in der IndexedDB als auch im Local Storage werden neben dem Dateiinhalt auch die Upload URL und der Dateiname gespeichert. Die Upload URL enthält sowohl einen Zeitstempel als auch die ID des Observed Objects, zu dem die Datei gehört. Im Local Storage wird für jede neue Datei ein neuer Eintrag angelegt. Deshalb muss man im Local Storage noch eine Zählervariable mitspeichern, die sich jedes Mal um eins erhöht, wenn man eine neue Datei abspeichert und sich um eins verringert, wenn man eine Datei erfolgreich hochgeladen hat. Ansonsten weiß der Uploadalgorithmus später nicht wie viele Dateien im Local Storage zwischengespeichert sind. Dieses Problem hat man in der IndexedDB nicht, da dort alle Dateien im ObjectStore „Dateien“ gespeichert werden und man problemlos den ObjectStore auslesen und durch diesen iterieren kann.



Wenn der Benutzer eine Datei zwischenspeichern ließ wird diese auch beim Seitenaufruf der Uploadseite aufgelistet. Die meistens Endbenutzer sind keine Informatiker und benutzen weder die Browserkonsole noch den Browserspeicher, um zu prüfen, ob der Upload funktioniert hat oder nicht. Da wir dem Nutzer ein Feedback geben wollen, dass die Uploadfunktion auch offline verfügbar ist und die Dateien nicht verloren gehen, wenn er diese benutzt, generieren wir beim Seitenaufruf eine Tabelle, die alle zwischengespeicherten Dateien aus der IndexedDB und dem Local Storage ausliest und auflistet. Damit sind auch mehrfache Uploads der gleichen Dateien unwahrscheinlich.

Dateien auslesen und hochladen

Nachdem die Dateien nun zwischengespeichert worden sind, müssen sie auch wieder hochgeladen werden. Wir haben uns dafür entschieden die Überprüfung der Netzwerkverbindung und der anschließenden Ausführung der Uploadfunktion in der `swac.js` zu implementieren, da diese in jeder HTML-Seite des Projekts eingebunden ist. Dies hat den Vorteil, dass die Uploadfunktion auch dann ausgeführt wird, wenn der Benutzer sich auf anderen Seiten (innerhalb unserer App) aufhält.

```
1
2 let db;
3 let request = indexedDB.open("UploadDateien",1);
4 request.onerror = function(event) {
5 alert("Fehler beim Öffnen der Datenbank");
6 };
7 request.onsuccess = function(event) {
8 db = event.target.result;
9
10 let transaction = db.transaction(["Dateien"], "readwrite");
11 transaction.oncomplete = function(event) {
12
13 };
14 transaction.onabort = function(event) {
15 console.log("Fehler beim Lesen der IndexedDB");
16 };
17 let objectStore = transaction.objectStore("Dateien");
18
19 let request = objectStore.getAll();
20 request.onsuccess = function(event) {
21 let results = request.result;
22 if(results.length===null || results.length===0)
23 {
24     return;
25 }
26
27 for(let i=0;i<results.length;i++)
28 {
29     console.log("versuche Upload für: "+ request.result[i][0].name);
30     let tmpfile = new File([request.result[i][0].content],
31                             request.result[i][0].name);
32     tmpfile.uploadid=Date.now()+(i+1);
33     let formData = new FormData();
34     formData.append('file', tmpfile);
35     // Upload data
36     $.ajax({
37         url: request.result[i][0].url,
38         type: "POST",
39         async: false,
40         cache: false,
41         data: formData,
42         processData: false,
43         contentType: false,
44         mimeType: "multipart/form-data",
45         success: function (response) {
46             console.log("Upload erfolgreich für Datei: "
47                         + request.result[i][0].name);
48         },
49         timeout:
50             50000,
51         error: function (xhr, ajaxOptions, thrownError) {
52             try {
53                 console.log("Fehler beim Upload der Datei");
54                 let upResponse = JSON.parse(xhr.responseText);
55                 console.log(upResponse);
```


Der oben stehende Code zeigt wie die Überprüfung und der Upload realisiert wurden. Zuvor wurde überprüft, ob überhaupt eine Internetverbindung besteht. Falls diese vorhanden ist, wird zunächst die IndexedDB geöffnet und der ObjectStore „Dateien“ ausgelesen. Falls dieser Ergebnisse liefert, wurden Dateien in der IndexedDB gespeichert. Für jede dieser Dateien wird anschließend der Upload mit Ajax ausgeführt. Die dargestellte Funktion ist identisch mit der Uploadfunktion, die im Normalfall bei bestehender Internetverbindung ausgeführt wird. Die meisten Parameter sind statisch, lediglich die Upload URL und der Dateiinhalt werden benötigt, welche beide gespeichert wurden. Zuletzt werden die Einträge aus der IndexedDB entfernt.

Diese Prozedur wird für den Local Storage ebenfalls ausgeführt:

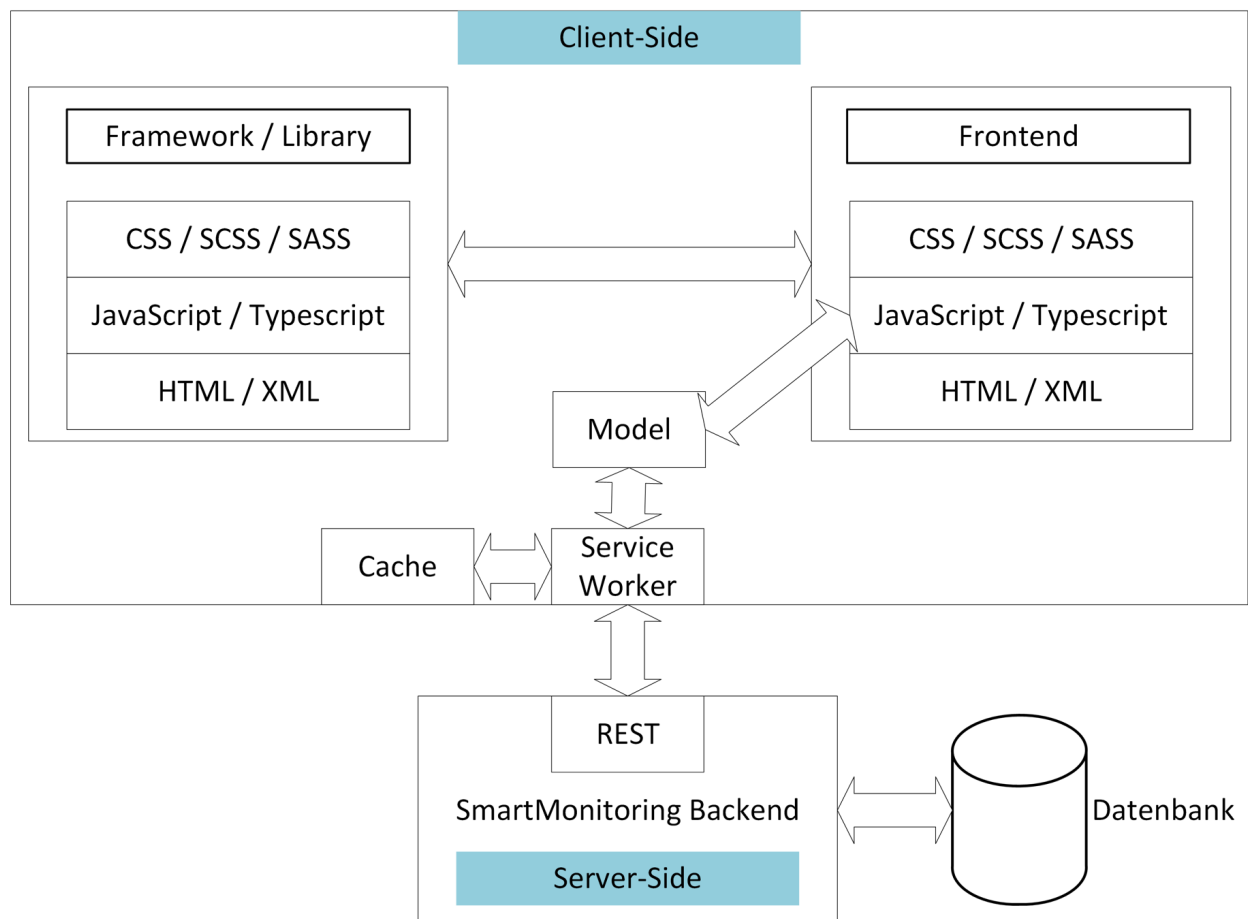
```
1
2 let counterLS = localStorage.getItem("fileCounter");
3 if(counterLS===null || counterLS===0)
4 {
5     return;
6 }
7 else {
8     for(let i=0;i<parseInt(counterLS);i++)
9     {
10         let jsonstr = localStorage.getItem("file"+(i));
11         let obj=JSON.parse(jsonstr);
12         console.log(i);
13         console.log(obj);
14         let tmpfile = new File([obj.content],obj.name);
15         tmpfile.uploadid=Date.now()+(i+1);
16         let formData = new FormData();
17         formData.append('file', tmpfile);
18         $.ajax({
19             url: obj.url,
20             type: "POST",
21             async: false,
22             cache: false,
23             data: formData,
24             processData: false,
25             contentType: false,
26             mimeType: "multipart/form-data",
27             success: function (response) {
28                 console.log("Upload erfolgreich für Datei: "+ obj.name);
29             },
30             timeout:
31                 50000,
32             error: function (xhr, ajaxOptions, thrownError) {
33                 try {
34                     console.log("Fehler beim Upload der" +
35                         "Datei aus Local Storage");
36                     let upResponse = JSON.parse(xhr.responseText);
37                     console.log(upResponse);
38                 } catch (ex) {
39                     console.log("Fehler beim Upload aus Local Storage");
40                 }
41             }
42         });
43         localStorage.removeItem("file"+(i));
44     }
45     localStorage.removeItem("fileCounter");
46 }
47 }
48
```

Wenn der Upload erfolgreich war, dann werden sämtliche Einträge für die Dateien sowie den Zähler für die Dateien aus dem Local Storage gelöscht.

Serviceworker

Der Serviceworker ist ein im Hintergrund laufendes Skript, welches die Offline-Funktionalität ermöglicht. Somit ist der Serviceworker die Hauptkomponente einer PWA. Während einer bestehenden Internetverbindung kann der Serviceworker je nach Implementierung Daten in dem Serviceworker-Cache ablegen. Wenn keine Internetverbindung mehr besteht, können Anfragen des Clients weiterhin bedient werden, in dem der Serviceworker aus seinem Cache die erforderlichen Daten liefert.

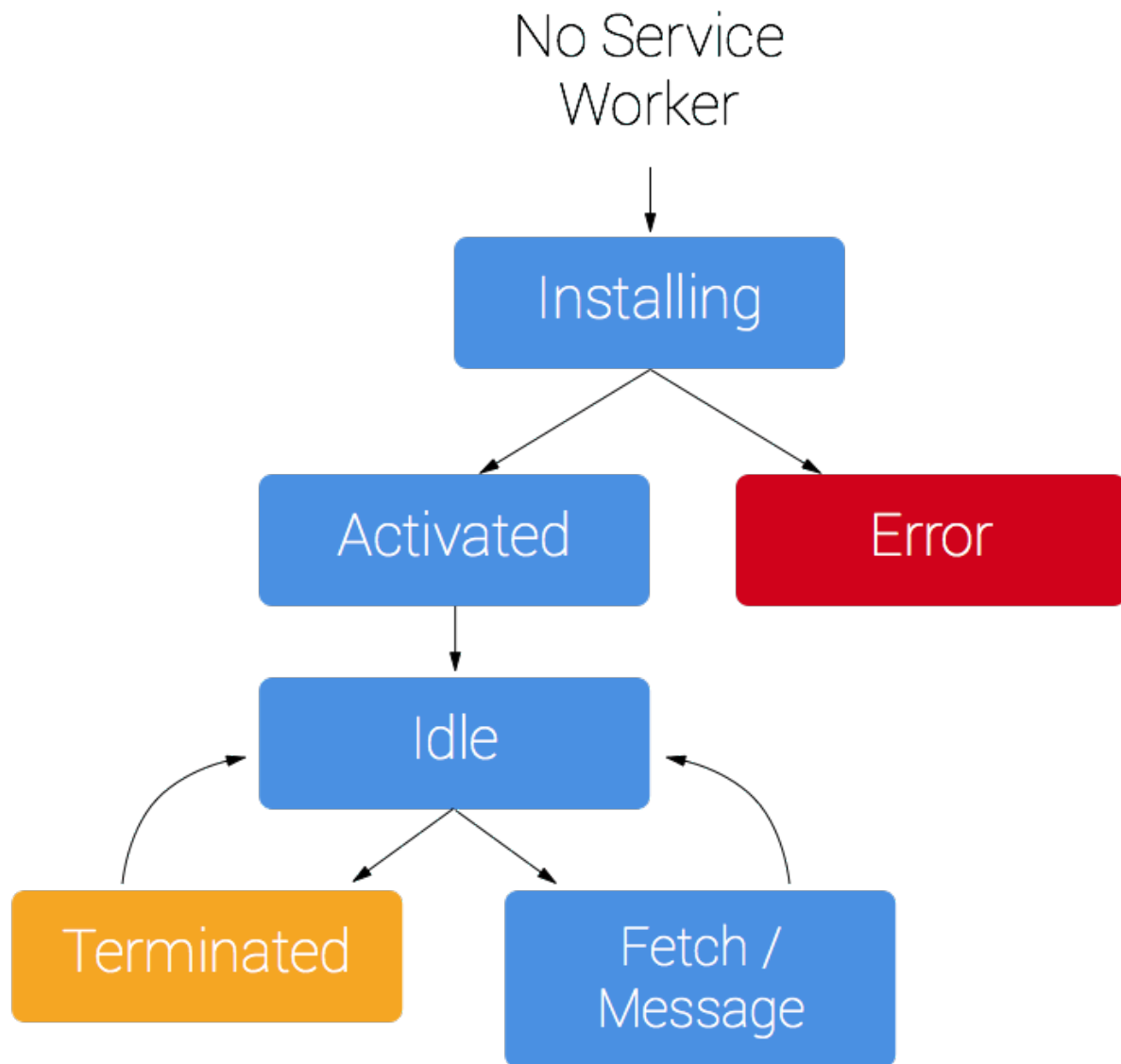
Das unten stehende Bild zeigt, an welcher Stelle sich der Serviceworker aus Sicht der Softwarearchitektur befindet.



Es ist gut zu erkennen, dass der Serviceworker den gesamten Datenverkehr der Anwendung kontrollieren kann.

Serviceworker Lifecycle

Der Serviceworker durchläuft verschiedene Phasen, bevor er bereit ist, seine reguläre Arbeit aufzunehmen. Das unten stehende Bild zeigt, den typischen Serviceworker-Lifecycle.



Die konkrete Implementierung der Events im Serviceworker-Lifecycle wird nachfolgend erklärt.

Activate-Event

Der Event-Listener für das activate-Event hat die Aufgabe, den Cache des Serviceworker aufzuräumen. Für jede URL, die im Cache abgelegt wurde, existiert ein Eintrag mit Zeitstempel in der indexedDB. Für den Zugriff auf die indexedDB wird die Bibliothek localForage verwendet. LocalForage bietet eine API die wie der Local Storage funktioniert. Der local Storage selber ist aus dem Serviceworker heraus nicht verwendbar.

Die Bibliothek wird dabei über die Service-Worker eigene importScripts-Funktion importiert.

```
1 //import localforage lib, this lib provides an api
2 //for easy key-value indexedDB access
3 self.importScripts('SWAC/libs/localForage-1.7.3/dist/localforage.js');
4
```

Um den Cache des Servicewokers aufzuräumen, wird mit der localForage Funktion iterate() über die indexedDB iteriert.

```
1 //iterate through list of urls in indexedDB
2 localforage.iterate(function (value, key, iterationNumber) {
```

Auf jeden Zeitstempel in der indexedDB wird dann die DeletionTime (also die maximale Verweilzeit einer URL) temporär aufaddiert. Wenn dann der Zeitstempel immer noch kleiner als die aktuelle Zeit ist, wird erst der Eintrag aus dem Cache gelöscht und dann aus der indexedDB selber.

```
1 //get date object from value in indexedDB
2   var timestamp = new Date(value);
3
4   //add number of days from DELETIONTIME
5   timestamp.setTime(timestamp.getTime()
6   + (DELETIONTIME * 24 * 60 * 60 * 1000));
7
8   var currentDate = new Date();
9
10  //if url is older then DELETIONTIME,
11  //delete first from cache, then from indexedDB
12  if (timestamp.getTime() < currentDate.getTime()) {
13
14      //console.log("OUTDATED FILE: " + [key, value]);
15
16      var request = new Request(key);
17
18      //remove file from cache
19      caches.open(CACHE).then(function (cache) {
20          cache.delete(request);
21      })
22
23      //remove entry from indexedDB
24      localforage.removeItem(key).then(function () {
25          // Run this code once the key has been removed.
26          //console.log('Key is cleared!');
27      }).catch(function (err) {
28          // This code runs if there were any errors
29          console.log(err);
30      });
```

Fetch-Event

Hier werden Anfragen an den Server bearbeitet. Da der Serviceworker sich im Rootverzeichnis befindet, aber mehrere Frontendordner sich dort befinden können, müssen Anfragen gefiltert werden.

Für den Frontendordner, auf die der Serviceworker reagieren soll, ist die Variable FRONTENDFOLDER angelegt. Aus diesem Grund muss dann auch angegeben werden, auf welche Art von URLs der Serviceworker noch reagieren soll. Zusätzlich soll der Serviceworker

auf POST-Request nicht reagieren, da dies zu Problemen bei der Upload-Funktion führt.

```
1  if ((evt.request.url.includes(FRONTENDFOLDER)
2      || evt.request.url.includes('Backend')
3      || evt.request.url.includes('SWAC'))
4      && evt.request.method !== 'POST') {
```

Zusätzlich werden bei URL, die einen GET-Parameter vom Typ „_“ haben, der GET-Parameter abgeschnitten. Dies ist erforderlich, da sich bei diesen Parametern der Wert sich bei jeder Anfrage ändert und somit ein cachen dieser URLs nicht sinnvoll ist.

```
1  if (evt.request.url.includes("?_")) {
2
3      var url = evt.request.url.toString();
4
5      //slice of GET-Parameter
6      url = url.slice(0, url.indexOf("?_"));
7
8      var modifiedRequest = new Request(url);
```

Anschließend wird die Anfrage wie folgt bearbeitet. Zuerst wird eine Antwort entweder aus dem Cache oder vom Server geliefert. Danach wird der Cache des Serviceworkers für die URL geupdated. Anschließend wird der Zeitstempel der URL in der indexedDB gespeichert bzw. aktualisiert.

```
1  evt.respondWith(fromNetworkOrCache(evt.request));
2  evt.waitUntil(update(evt.request));
3  evt.waitUntil(saveCacheDate(evt.request));
4
```

Funktion fromNetworkOrCache()

Diese Funktion liefert eine Antwort auf einen Fetch. Für Anfragen an das Backend wird immer versucht, aktuelle Daten vom Server zu liefern. Nur wenn eine solche Anfrage fehlschlägt, wird versucht die Anfrage aus dem Cache zu liefern. Für das Backend ist dies eine gute Strategie, da aus dem Backend sich schnell ändernde Werte abgefragt werden und diese somit möglichst aktuell sein sollten. Für alle anderen Anfragen ist die Strategie genau andersherum. Aus Performance Gründen wird dort zuerst eine Antwort aus dem Cache geliefert. Für URLs die z.B. das Frontend betreffen ist eine veraltete Seiten i.d.R. unproblematisch und diese Strategie erhöht die Geschwindigkeit der PWA.

```
1
2   if (request.url.includes("backend")) {
3       return fetch(request) || matching;
4
5   } else {
6       return matching || fetch(request);
7   }
```

Falls sowohl eine Antwort vom Server als auch aus dem Cache fehlschlägt, wird eine Fallback-Seite geliefert.

```
1   }).catch(function () {
2       //in case of no possible fetch an no match in cache, use fallback site
3       console.log("use fallback");
4       return useFallback();
5   });
```

Funktion useFallback()

Diese Funktion liefert eine Html-Seite aus dem Cache, im Falle dass eine Anfrage weder aus dem Cache noch vom Server beantwortet werden konnte. Diese Fallback-Seite wird bei der Installation geprecached.

```
1   function useFallback() {
2
3       var fallbackRequest = new Request('./'
4           + FRONTENDFOLDER + '/sites/offline.html');
5
6       return caches.open(CACHE).then(function (cache) {
7           return cache.match(fallbackRequest).then(function (matching) {
8               return matching;
9           });
10      });
11  }
```

Funktion update()

Diese Funktion macht eine Anfrage an den Server und packt die Antwort in den Cache.

```
1   function update(request) {
2       return caches.open(CACHE).then(function (cache) {
3           return fetch(request).then(function (response) {
4               return cache.put(request, response);
5           });
6       });
7   }
```

Funktion saveCacheDate()

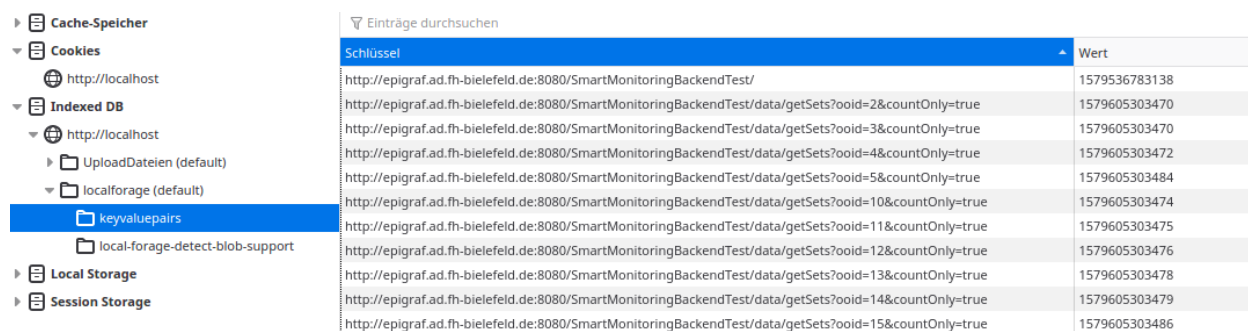
Diese Funktion speichert den Zeitstempel einer gecachten URL in der indexedDB mithilfe der Bibliothek localForage.

```

1 function saveCacheDate(request) {
2
3     //get current date
4     var d1 = new Date();
5     //safe url of request with timestamp of current date
6     localforage.setItem(request.url.toString(), d1.getTime());
7 }

```

Das unten stehende Bild zeigt die Zeitstempel in der indexedDB.



The screenshot shows the IndexedDB interface with a search bar and a list of entries. The left sidebar shows the database structure, and the main area displays a table of key-value pairs.

Einträge durchsuchen	Schlüssel	Wert
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/	1579536783138
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=2&countOnly=true	1579605303470
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=3&countOnly=true	1579605303470
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=4&countOnly=true	1579605303472
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=5&countOnly=true	1579605303484
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=10&countOnly=true	1579605303474
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=11&countOnly=true	1579605303475
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=12&countOnly=true	1579605303476
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=13&countOnly=true	1579605303478
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=14&countOnly=true	1579605303479
	http://epigraf.ad.fh-bielefeld.de:8080/SmartMonitoringBackendTest/data/getSets?oid=15&countOnly=true	1579605303486

Funktion preCacheUrlsFromFile()

Diese Funktion hat die Aufgabe, aus einer Datei URLs auszulesen und in den Cache zu packen. Als Übergabeparameter bekommt die Funktion den Namen der Datei und, optional, einen Prästring für relative Pfadangaben in der Datei.

##Registrieren des Serviceworkers

Der Serviceworker wird im Skript swac.js mit der Funktion registerServiceWorker registriert.

```

1 function newSWAC() {
2     // Load configuration
3     let firstSlashPos = window.location.pathname.indexOf("/", 2);
4     let approot = window.location.pathname.substr(0, firstSlashPos);
5     var tag = document.createElement("script");
6     tag.id = 'config';
7     tag.src = approot + "/configuration.js";
8     document.getElementsByTagName("head")[0].appendChild(tag);
9     tag.addEventListener('load', function (evt) {
10         SWAC.loadGlobalComponents();
11     });
12     registerServiceWorker();
13     checkIfOnline();
14 }

```

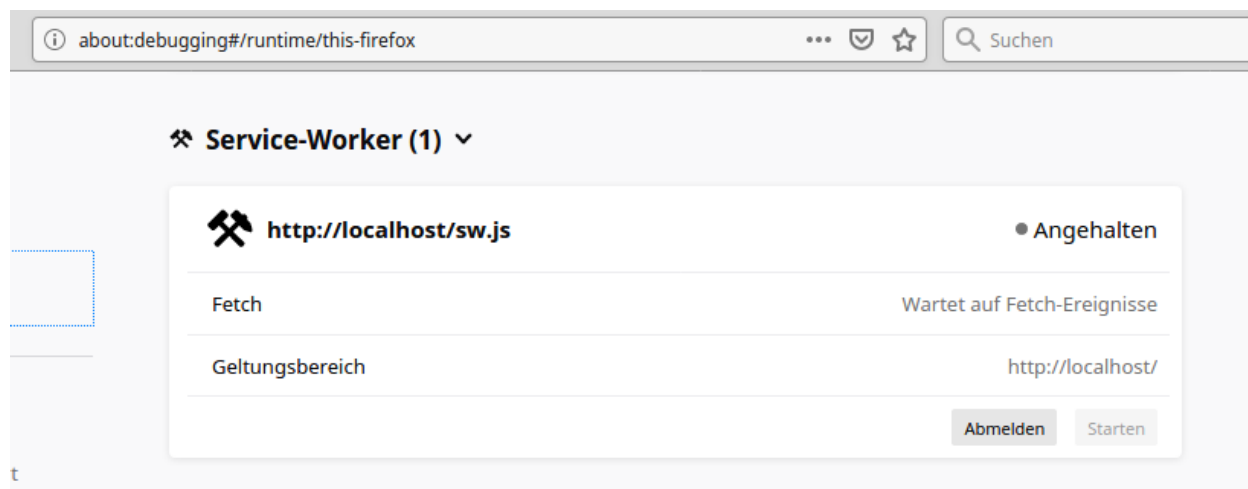
Die Funktion registerServiceWorker sieht wie folgt aus.


```
1 function registerServiceWorker() {  
2  
3   if ('serviceWorker' in navigator) {  
4     navigator.serviceWorker.register('/sw.js')  
5       .then((reg) => {  
6         console.log('Service worker registered.' , reg);  
7       }).catch((error) => {  
8         console.log('Service worker failed to register.', error);  
9       });  
10  }  
11 }
```

Der registrierte Serviceworker lässt sich wie folgt verwalten:


In Firefox über die URL:

<about:debugging#runtime/this-firefox>



In Chrome über die URL:

<chrome://serviceworker-internals>

 Chrome | chrome://serviceworker-internals

ServiceWorker

☐ Open DevTools window and pause JavaScript execution on Service Worker startup for debugging.

Registrations in: /home/martin/.config/google-chrome/Default (3)

Scope: <http://localhost/>
Registration ID: 168
Navigation preload enabled: false
Navigation preload header length: 0
Active worker:
Installation Status: ACTIVATED
Running Status: STOPPED
Fetch handler existence: EXISTS
Script: <http://localhost/sw.js>
Version ID: 664
Renderer process ID: 0
Renderer thread ID: -1
DevTools agent route ID: -2
Log:

Unregister

Start

)

Visualisierungskomponente

Eine weitere Anforderung war es gewesen, eine Visualisierungskomponente zu erstellen. Diese Komponente sollte eine SWAC- Komponente sein. Man sollte also in der Lage sein diese, wie auch die anderen SWAC-Komponenten, einfach im HTML überall einbinden zu können.

Einbindung

Die Einbindung der Komponente erfolgt mit einem allgemeinen Schema:

```
[Komponente] FROM [Datenquelle]
```

Dieses Schema gibt man bei der Einbindung in HTML mit an. Bei dieser Komponente sieht die Einbindung in diesem Projekt folgendermaßen aus:

```
1 <div id="thermometer" swa="SWAC_visualise
2 FROM dataview/findByDataviewConfigurationId
3 WHERE dataviewConfiguration_id=1 OPTIONS swac_visualise_thermometer"></div>
```

Wie man sieht gibt man unter dem swa-Attribut das obige Schema an. Die Bezeichnung der Visualisierungskomponente ist *SWAC_visualise*.

Die Datenquelle ist in diesem Fall `dataview/findByDataviewConfigurationId`. An dieser Adresse im Backend findet sich eine Liste mit Objekten die Daten, und weitere für die Verarbeitung notwendige Informationen, enthalten.

Mithilfe von *WHERE* lassen sich solche Datensätze weiter filtern, in diesem Fall durch `WHERE dataviewConfiguration_id=1`.

Der Teil letzte des Schemas *OPTIONS* ist bei vielen Komponenten optional und verschieden. Unter *OPTIONS* gibt man Optionen und nötige Informationen für die Komponente an. Bei der Visualisierungskomponente ist die Angabe eines Optionsobjektes erforderlich. Diese werden als JavaScript Objekt definiert und hier angegeben. Ein für die Visualisierungskomponente erforderliches Objekt sieht folgendermaßen aus:

```
1 var swac_visualise_thermometer = {};
2 swac_visualise_thermometer.dtype = 'thermometer';
3 swac_visualise_thermometer.colorangePath = {
4   path: 'dataview/colorRange/getByDataviewId?dataview_id=',
5   id: '2'
6 };
7 swac_visualise_thermometer.dataPath = {
8   path: 'data/getSets?ooid=',
9   filter: '&limit=1&orderBy=ts&order=DESC',
10  id: '38'
11 };
```

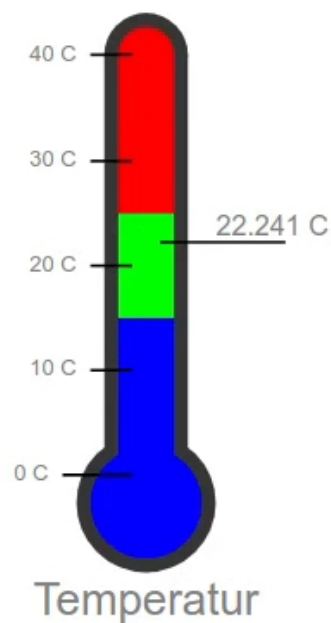
Das Objekt hat, in der Datei *visualise.js* definierte, Attribute. In jener Datei werden die in diesem

Objekt festgelegten Werte eingelesen und verarbeitet. Die Angabe des Attributs `dtype` ist hier zwingend erforderlich und wirft andernfalls einen Fehler.

Die erstellte Komponente, in diesem Fall Diagramme, werden unter diesem aufrufendem HTML-Element eingehängt.

Diagramme

Die Komponente kann Daten als zwei verschiedene Diagramme darstellen. Zum einen als Thermometer zur Darstellung der Temperatur:



Temperatur: 22.241C
Stand: 11:27:32.496 Uhr
2018-10-30

Weiterhin als Hygrometer zur Darstellung der Luftfeuchtigkeit:



Luftfeuchtigkeit: 38.27%
Stand: 11:27:32.496 Uhr
2018-10-30

Die Diagramme werden in JavaScript generiert und die angezeigten Daten werden aus der Datenbank gefetcht.

Beide Diagrammklassen, definiert in *Thermometer.js* und *Meter.js*, erben von der Klasse *Diagram* in *Diagram.js*. Generiert wird ein Diagramm in der `drawDiagram()` Funktion, welches ein SVG-Element zurückliefert. Jeweils pushen beide Diagrammtypen auch ihre Konstruktoren in die in der *Diagram.js* definierten `diagram_types`. Diese sind hinterher nötig, um die Diagrammobjekte zu erstellen.

```
1 | class Diagram {  
2 |     constructor(name, width, height, diagramRange, unit, data) {
```

Wie am Konstruktor zu sehen ist werden alle nötigen Daten direkt übergeben. Das nötige `diagramRange`, ein `DiagramRange` Objekt, ist in der gleichnamigen Datei definiert. In diesem Objekt ist auch ein `colorRange` Objekt gespeichert. Das übergeben von den Farb- und Diagrammgrenzen ist hier wichtig, da die Diagramme generiert werden.

Wie alle SWAC-Komponenten besitzt auch diese Komponente ein HTML Template. Dieses legt zusammen mit einer CSS Datei das Layout der Komponente fest und definiert Stellen wo weitere Elemente, Daten oder Sprachdaten eingefügt werden können.

```

1  <div class="diagram_div">
2    <div class="diagram_hook">
3    </div>
4    <div class="diagram_info uk-width-1-2">
5      <div class="diagram_info_data_div">
6        <b>{name} : </b>{value}<br>
7        <div class="diagram_info_state_div">
8          <b>##visualise.state##</b>
9          <div>{time}  ##visualise.clock##<br>{date}</div>
10       </div>
11     </div>
12   </div>
13 </div>

```

Unter dem div-Element mit der `diagram_hook` Klasse wird ein fertiges Diagramm als SVG-Element eingehangen.

An Stellen wie `{value}` oder `{time}` werden hinterher entsprechende Daten eingefügt.

`##visualise.state##` und `##visualise.clock##` werden durch die angegebenen Attribute im Sprachobjekt ersetzt.

Erstellungsablauf

Es beginnt in der `visualise.js` in der in `SWAC_visualise.init` definierten Funktion. Diese erhält eine `requestor` Variable. Sie ist das Element, welches die Komponente aufgerufen hat.

```

1  SWAC_visualise.init = function(requestor) {
2    return new Promise((resolve, reject) => {
3      // Create data binding
4      let bind = SWAC_bind.bind(requestor, SWAC_visualise);
5
6      // Wait for view and bind to load
7      bind.then(function(resolveObj) {
8        if (requestor.swac_comp.options.dtype !== null) {
9          if (requestor.swac_comp.options.dtype === 'thermometer') {
10             loadDiagrams(resolveObj.data[0], requestor);
11           } else if (requestor.swac_comp.options.dtype === 'hygrometer') {
12             loadDiagrams(resolveObj.data[1], requestor);
13           }
14         } else {
15           throw "No diagram type has been picked";
16         }
17       });
18     });
19   };

```

In SWAC sind Model und View getrennt. Im Model sind die Daten welche man im Requestor angegeben hat und für die View steht das HTML Template mit dem dazugehörigen CSS. Das Bind bindet Model und View mit `SWAC_bind.bind(requestor, SWAC_visualise);`. Die `bind` Funktion liefert ein Promiseobjekt zurück, welches eine Resolvefunktion mit einem

resolveObj ausführt.

Dieses ist das Objekt welches man im Requestor als Datenquelle angegeben hat. In ihm stecken die nötigen Daten und Informationen.

Da der Requestor das aufrufende Element ist, kann man über ihn auch auf die angegebenen Optionen zugreifen. Je nachdem welcher dtype gesetzt wurde, bzw. welches Diagramm erstellt werden soll, wird ein anderes Objekt aus dem Datenobjekt resolveObj an die weitere Funktion gegeben.

Für die Visualisierungskomponente im SmartMonitoring-System sieht das Datenobjekt so aus:

```
1  {
2  "list" : [
3    {
4      "diagramType" : 1,
5      "dataviewConfiguration" : "ref://dataviewconfiguration/get/1",
6      "id" : 2,
7      "ooTypeJoinMType" : "ref://observedobjecttypejoinmeasurementtype/get/861",
8      "observedObject" : "ref://observedobject/get/38",
9      "weatherDiagram" : false
10   },
11   {
12     "dataviewConfiguration" : "ref://dataviewconfiguration/get/1",
13     "diagramType" : 0,
14     "id" : 3,
15     "observedObject" : "ref://observedobject/get/38",
16     "ooTypeJoinMType" : "ref://observedobjecttypejoinmeasurementtype/get/859",
17     "weatherDiagram" : false
18   }
19 ]
20 }
```

Das erste Objekt in der Liste beschreibt das Thermometer, das zweite das Hygrometer. Beide Objekte definieren wo einzelne Daten zu finden bzw. gespeichert sind. So findet man z.B. die eigentlichen Temperatur und Luftfeuchtwerte unter observedObject oder Informationen zum Datentyp oder die Bezeichnungen in ooTypeJoinMType .

Die Referenz zu den Farbgrenzen fehlt hier aber. Da die Komponente jedoch weiterhin nicht anwendungsspezifisch sein soll, kann man den Verweis zu den Farbgrenzen als Option in colorrangePath definieren. Das selbe gilt auch für die eigentlichen Datenwerte, dessen Verweis hier nicht ganz korrekt ist und die daher unter Option dataPath mit angegeben werden können.

```
1 function loadDiagrams(diagramDescription, requestor) {
2   if (diagramDescription.weatherDiagram === false) {
3     fetch(SWAC_config.datasources[0].replace('[fromName]', ''))
4       + requestor.swac_comp.options.colorrangepath.path
5       + requestor.swac_comp.options.colorrangepath.id)
6     .then((response) => {
7       return response.json();
8     }).then((colorRanges) => {
9       if (colorRanges === undefined
10         || colorRanges.errors !== undefined
11         || colorRanges.list.length === 0)
12         throw "No data could be loaded";
13       let diagramRange = new DiagramRange();
14       colorRanges.list.forEach((colorRange) => {
15         let colorRangeObject = new ColorRange(colorRange.minValue,
16           colorRange.maxValue, colorRange.color);
17         diagramRange.addColorRange(colorRangeObject);
18       });
19     });
20 }
```

In dieser Funktion `loadDiagrams` werden zunächst die Farbgrenzdaten aus dem Backend gefetcht. Dafür wird der in den Optionen mitgegebener Pfad und Id in dem `colorrangepath` Objekt verwendet. Es wird ein `DiagramRange` Objekt erstellt, dem die Farbgrenzen hinzugefügt werden. Jenes wird für die Erstellung des Diagramms gebraucht und daher an die nächste Funktion übergeben.

```
1 var promise;
2 if (diagramDescription.observedObject !== undefined) {
3   promise = fetch(SWAC_config.datasources[0].replace('[fromName]', ''))
4     + requestor.swac_comp.options.dataPath.path
5     + requestor.swac_comp.options.dataPath.id
6     + requestor.swac_comp.options.dataPath.filter)
7     .then((response) => {
8       return response.json();
9     }).then(dataList => {
10       createDiagramm(dataList, diagramDescription,
11         diagramRange, requestor);
12     });
13 }
```

Im zweiten Teil dieser Funktion werden die Daten unter Verwendung des angegebenen Pfads, Id und Filter im `dataPath` gefetcht. Die nun erhaltenen Daten sowie die Diagrammgrenzwerte werden an die letzte Funktion mit weitergegeben.


```
1 let typestr = diagramDescription.ooTypeJoinMType.replace('ref://', '');
2   fetch(SWAC_config.datasources[0].replace('[fromName]', ''))
3   + typestr.substring(0, typestr.lastIndexOf('/')) + '?id='
4   + typestr.substring(typestr.lastIndexOf('/')+1))
5   .then((response) => {
6     return response.json();
7   }).then((joiner) => {
8     [...]
9     if (dataList.list[0][joiner.aliasname] !== undefined)
10      data = dataList.list[0][joiner.aliasname];
11    else
12      data = dataList.list[0][joiner.mtype_name];
13  }
```

Nun in der letzten Funktion `createDiagramm` werden die Beschreibungsdaten mithilfe des Verweises im Datenobjekt gefetcht. Unter anderem kommt man nun an den `aliasname`, den man braucht um die eigentlichen Temperatur oder Luftfeuchtigkeitswerte aus der `dataList` zu erhalten. Auch angegeben sind dort die Darstellungen für die Einheiten, z.B. Celsius oder %, welche dann eingefügt werden können.

```
1 let diagram = new diagram_types[diagramDescription.diagramType]
2   (SWAC_language.visualise[requestor.swac_comp.options.dtype],
3   window.innerHeight / 2, window.innerHeight / 2, diagramRange,
4   joiner.unit, data);
5 let elem = requestor.getElementsByClassName('diagram_info_data_div')[0];
6 SWAC_view.searchAndReplace('name', diagram.name, elem);
7 SWAC_view.searchAndReplace('value', diagram.data + diagram.unit, elem);
8
9 SWAC_view.searchAndReplace('time', dataList.list[0].ts.split('T').pop(),
10 elem);
11 SWAC_view.searchAndReplace('date', dataList.list[0].ts.split('T')[0],
12 elem);
13
14 let svgNode = diagram.drawDiagram();
15 svgNode.classList.add('uk-card');
16 svgNode.classList.add('uk-card-body');
17
18 requestor.querySelector('.diagram_hook').appendChild(svgNode);
```

Das Diagramm wird jetzt erstellt. Die in den jeweiligen Diagrammdateien gepushten Konstruktoren werden nun mit den gefetchten Daten und einer Höhe und Breite aufgerufen. Die markierten Bereichen wie `name` oder `time` werden entsprechend mit Daten ersetzt.

```
1  {  
2    "id": 1000000,  
3    "humidity": 41.8500000000000014,  
4    "pressure": 1011.59000000000003,  
5    "temperature": 27.3299999999999983,  
6    "ts": "2018-07-06T05:09:02.551"  
7  }
```

Beispielsweise wird hier die Zeit aus dem `dataList` Objekt ausgelesen und entsprechen für `time` und `date` ersetzt. Die Werte wie z.B. die Temperatur wurden schon als `data` dem Diagramm übergeben und werden von dort aus verwendet.

Das Diagramm kann nun gezeichnet und an dem entsprechenden Hook unter dem Requestor eingefügt werden.

Installation der PWA

Chrome

Menü -> Zum Startbildschirm hinzufügen

Firefox

Menü -> Zum Startbildschirm hinzufügen

Ausblick

Intelligente Cache-Verwaltung

Momentan werden gewisse URLs geprecached, ansonsten werden nur die URLs in den Cache gepackt, die auch aufgerufen werden. Denkbar ist hier eine Lösung, bei der gewisse Daten vorgeladen werden. Z.B. nach häufiger Nutzung gewisser Daten könnten diese automatisch aktualisiert werden, ohne dass sie jedes Mal neu aufgerufen werden müssen. Auch eine Art über HTML verfügbare Konsole für das Einstellen der Cache-Verwaltung wäre denkbar.

Quellenverzeichnis

Kapitel Theoretische Grundlagen progressiver Webanwendungen:

<https://developer.mozilla.org/de/docs/Web/Manifest> (<https://developer.mozilla.org/de/docs/Web/Manifest>)

https://de.wikipedia.org/wiki/Progressive_Web_App (https://de.wikipedia.org/wiki/Progressive_Web_App)

Kapitel Serviceworker:

<https://developers.google.com/web/fundamentals/primers/service-workers>

(<https://developers.google.com/web/fundamentals/primers/service-workers>)

Bild Serviceworker-Lifecycle: <https://developers.google.com/web/fundamentals/primers/service-workers/images/sw-lifecycle.png> (<https://developers.google.com/web/fundamentals/primers/service-workers/images/sw-lifecycle.png>)