Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

Umsetzung einer Schnellstartseite als Erweiterung für Google Chrome mit Dart

Praxisarbeit

Eingereicht von: Benjamin Jesuter
Lützner Straße 101
04177 Leipzig
Seminargruppe: CS13 -2
Matrikelnummer: 5000341

Betreuer: Dipl. Ing. Tino Müller
ibes AG
Bergstraße 55
09113 Chemnitz

Leipzig, 14.01.16

Hinweis: Diese Praxisarbeit kann unter
https://github.com/Linkmaster08/chrome-speedpad-research-paper
mit allen notwendigen Quelldokumenten und Quellcodedateien eingesehen und heruntergeladen werden.

Inhaltsverzeichnis

1 Einleitung 4

2 Grundlagen 5

2.1 Die Programmiersprache – Dart 5

2.2 Chrome Apps und Erweiterungen 6

3 Aufbau des Projektes 6

3.1 Lokale Speicherung der Icons und der Linkdaten 7

3.2 Synchronisation der Icons 8

4 Das Hauptmodul „chrome–speedpad“ 9

4.1 Das Design 10

4.2 Die Technik – AngularDart 11

4.3 Datenfluss zwischen den Komponenten 13

5 Das Dienstmodul „speedpad–service“ 16

6 Das Hilfsmodul „shared–classes“ 16

6.1 Serialisierung in Dart 17

7 Kommunikation zwischen „chrome–speedpad“ und „chrome–service“ 19

7.1 Übertragungsprotokoll 20

7.2 Verbesserung der Geschwindigkeit der Anwendung 20

8 Fazit 21

9 Verzeichnisse 22

9.1 Abbildungsverzeichnis 22

9.2 Listingverzeichnis. 22

9.3 Quellenverzeichnis - Informationsquellen. 22

10 Selbstständigkeitserklärung 24

1 Einleitung

Eine Schnellstartseite ist ein spezieller Bildschirm in einer Anwendung, welcher dem Benutzer schnellen Zugriff auf häufig benutzte Funktionen eines Programms geben soll. In einem Browser ist dies meistens über eine Erweiterung (auch Addon oder Extension genannt) realisiert, welche die Seite ersetzt, die in einem gerade neu geöffnetem Tab angezeigt wird. Dort erscheinen dann Lesezeichen, die visuell mit Icons oder Bildschirmfotos der verlinkten Webseite aufbereitet sind, um dem Nutzer das Auffinden eines speziellen Lesezeichens zu erleichtern. Diese werden dann meistens in einem anpassbaren Raster angezeigt. *Abb. 01 auf Seite 4* zeigt die Seite einer Schnellstart-Erweiterung für den Google Chrome Browser, welcher im Folgenden nur noch Chrome genannt wird.

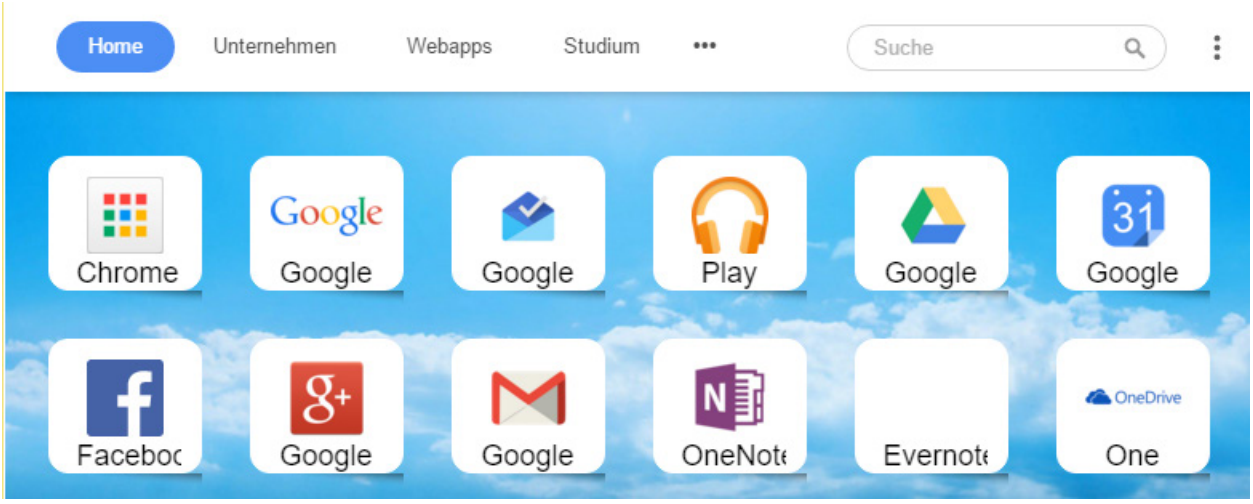

Abb. 01: Schnellstartseite in Google Chrome

Die in der Abbildung gezeigte Chrome Erweiterung heißt SpeedDial 2. Aber schon in diesem Bildschirmfoto fällt auf, was man verbessern könnte. Da ist zum einen die Darstellung der Titel der einzelnen Links, die auf dieser Skalierung eher nicht so gut aussieht. Zum anderen sieht man bei dem Evernote-Lesezeichen, dass das Icon fehlt. Was man hier jetzt nicht sehen kann, ist, dass die Vorschaubilder oder Icons nicht lokal auf dem PC des Anwenders gespeichert werden, sondern nur von ihrer eingerichteten Quelle im Internet abgerufen werden. Dies sorgt ab und zu für diese unschönen weißen Kästen, welche dadurch verursacht werden, dass das eingebundene Bild an seiner Quelladresse nicht mehr gefunden werden kann.

Aus diesen Beobachtungen heraus entstand der Wunsch, eine neue Schnellstart-Erweiterung zu entwickeln, welche die genannten Schwierigkeiten und Unschönheiten behebt.

Die Ziele dieser neuen Erweiterung sind somit

- die Offline-Speicherung der Icons jedes einzelnen Links (Ziel: lokal),
- die Synchronisierung dieser Icons mit allen Google Chrome Instanzen auf allen Geräten, in denen der Nutzer mit seinem Google-Konto angemeldet ist (Ziel: synchronisiert),
- sowie eine Verbesserung der Optik, um ansprechender zu sein und die Anzeige der Titel auch bei kleinen Skalierungen zu ermöglichen (Ziel: responsive und ansprechend).

2 Grundlagen

2.1 Die Programmiersprache – Dart

Die Entwicklung von Chrome Apps und Erweiterungen erfolgt normalerweise in Html, CSS und JavaScript. Man kann es jetzt natürlich direkt in JavaScript entwickeln, oder man bedient sich einer Sprache, die ihren Quellcode zu JavaScript kompilieren kann, um sich die Entwicklung zu vereinfachen. Da gibt es natürlich ein paar Sprachen, beispielsweise TypeScript [Q1] oder Dart [Q2]. Für dieses Projekt wurde Dart gewählt, da hier eine entsprechende Programmbibliothek verfügbar ist, die den Zugriff auf die, in JavaScript angebotenen, APIs von Chrome erlaubt. Diese Programmbibliothek heißt einfach nur „chrome“ [Q3] und ist im offiziellen pub repository zu finden.

Weiterhin eliminiert Dart viele Fallstricke, die bei der Programmierung einer umfangreichen App in JavaScript entstehen können. Dafür besitzt es zum Beispiel [Q4]

- eine ähnliche Syntax wie die C – Sprachfamilie (C, C++, Java, usw.), mit kleineren syntaktischen Vereinfachungen
- viele und gut strukturierte Standardbibliotheken, wie Dart HTML für den Zugriff auf DOM Manipulationen
- optionale Typisierung für die Flexibilität einer typenlosen Sprache kombiniert mit der schnellen Erkennung von Fehlern zur Entwicklungszeit mit einem statischen Analyzer
- First-Class Unterstützung für asynchrone Ausführung von Code mittels Futures und Streams, den async/await Schlüsselwörtern und asynchronen Generatoren mittels async* and yield* [Q5]
- Modularität mittles Bibliotheken und Paketen
- und viele weitere Dinge.

2.2 Chrome Apps und Erweiterungen

Die Entwicklung einer solchen Schnellstart-Erweiterung ist eigentlich recht einfach. Im Grunde handelt es sich dabei nämlich um eine einfache Single-Page-Application, wie man sie auch als normale Web Applikation bauen könnte. Der Unterschied zwischen der Chrome Erweiterung und einer normalen Web Applikation ist, dass die Chrome Erweiterung speziell an die Chrome Plattform gebunden ist. Dadurch kann man sie packen und sehr einfach über den Chrome Web Store publizieren, oder als gepackte und signierte „.crx“ Datei weitergeben. Weiterhin besitzen solche Chrome Erweiterungen bzw. Apps viel mehr Möglichkeiten als eine Web App, da sie auf spezielle Browser APIs zugreifen können. Zum Beispiel kann man mit der Omnibox API auf die Adresszeile in Chrome zugreifen, man bekommt Zugriff auf weitere APIs zum Speichern von Daten, wie Chrome Local Storage und Chrome Sync Storage, oder man kann sogar auf Hardware wie Bluetooth und USB zugreifen. [Q6]

Diese Zugriffe setzen allerdings voraus, dass man die entsprechenden Berechtigungen besitzt. Dazu besitzt jede App oder Erweiterung eine sogenannte Manifest Datei, welche verschiedene Daten zur jeweiligen App oder Erweiterung enthält. So enthält sie hauptsächlich Name, Beschreibung, Versionsnummer und, am wichtigsten, den bzw. die Einsprungspunkte der App oder der Erweiterung, wie z.B. ein JavaScript, was im Hintergrund ausgeführt wird, oder eine HTML Seite, die beim Aufrufen der App oder Erweiterung angezeigt wird. [Q7]

Zusätzlich enthält diese Manifest Datei, welche in JSON verfasst wird, ein Array mit den Rechten, welche die Anwendung anfordert. Die angeforderten Rechte werden dann bei der Installation der App oder Erweiterung in Chrome angezeigt und der Nutzer wird gefragt, ob er der App diese Rechte geben möchte. Dies ist für viele APIs notwendig, die nur für Chrome Erweiterungen oder Apps zur Verfügung stehen. Auf Web APIs kann grundsätzlich ohne weitere Rechte zugegriffen werden, da diese schon so restriktiv gebaut sind, dass sie gefahrlos von jeder beliebigen Webseite benutzt werden können. [Q8]

3 Aufbau des Projektes

Das Projekt ist in drei Module aufgeteilt. Das Hauptmodul heißt „chrome-speedpad“ und beinhaltet die Oberfläche des Schnellstarters, welche die „Neuer Tab“ Seite des Browsers ersetzt. Daneben gibt es ein Dienstmodul mit dem Namen „chrome-service“, welches hauptsächlich für die persistente Datenspeicherung zuständig ist. Für Klassen, die von beiden Modulen gebraucht werden, gibt es das Hilfsmodul „shared-classes“.

Technisch gesehen ist „chrome-speedpad“ eine Erweiterung (engl. Extension) für den Google Chrome Browser und „chrome-service“ eine Chrome Desktop App (im Folgenden Chrome App oder nur App genannt). Diese Trennung ist notwendig, um die genannten Ziele „lokal“ und „synchronisiert“ erreichen zu können. Der Zusammenhang wird in den folgenden beiden Unterkapiteln genauer erläutert.

3.1 Lokale Speicherung der Icons und der Linkdaten

Das Erste Ziel stellt die lokale Speicherung der Icons eines Schnellstarteintrags auf dem PC des Nutzers dar. Der Hauptvorteil davon ist, dass die Bilder immer verfügbar bleiben. Bindet man ein Vorschaubild direkt aus seiner Quelle im Internet als Icon ein, kann es das Problem geben, dass dieses irgendwann nicht mehr verfügbar ist. Dann bekommt der Nutzer unschöne leere Flächen zu sehen, wo eigentlich das Icon sein sollte, oder noch schlimmer, die ganze Schnellstartverknüpfung wird evtl. nicht mehr angezeigt.

Da Browser meist eingeschränkten Zugriff auf das Dateisystem des Nutzers haben, stellt sich die Frage, wie diese lokale Speicherung umsetzbar ist. Chrome stellt für das Speichern von Daten unterschiedliche APIs für unterschiedliche Anwendungsgebiete zur Verfügung. Die verschiedenen Möglichkeiten finden Sie in *Tabelle 01 auf Seite 7*.

Speichersystem oder API	**Speichertyp**	**Speicherbeschränkungen**
Chrome Sync Storage	Schlüssel-Wert Speicher	100 kB
Indexed DB	Objektspeicher	nicht bekannt, speziell für jeden Browser bzw. teilweise auch für jede Browserversion
Chrome Local Storage	Schlüssel-Wert Speicher	5 MB, oder unbegrenzt mit Erlaubnis
Chrome File System (nur Apps)	Dateisystem des Nutzers	bearbeitet beliebige Dateien und Ordner auf Anfrage und Erlaubnis durch den Nutzer
Chrome Synced File System	Spezieller Chrome-verwalteter Ordner	unbekannt, wahrscheinlich keine

Tabelle 01 [Q9]

Chrome Sync Storage ist für die geplante Anwendung als Bildspeicher wegen dem Limit auf 100 kB nicht zu gebrauchen, IndexedDB ist eigentlich für strukturierte nicht-binär Daten gedacht.

Chrome Local Storage ist eine gute Option, zumindest, solange man die Berechtigung auf unlimitierten Speicher vom Nutzer bekommen hat, da man die 5 MB Grenze mit der Speicherung von Bildern relativ schnell erreicht haben sollte.

Theoretisch könnte man auch einen eigenen Dienst im Internet zur Speicherung dieser Icons bereitstellen. Dies würde auch das Problem beheben, dass die Bilder plötzlich weg sind, da man dies dann selbst kontrollieren kann. Allerdings kämen dadurch eine Reihe neuer Probleme hinzu, wie z.B. die Authentifizierung der Nutzer, die schwierige Skalierbarkeit und hohe Kosten, da man für jeden Nutzer der Applikation einen eigenen Speicherbereich auf dem Server bereitstellen müsste, und die Frage, ob die Nutzer einem unbekannten Entwickler einfach Daten überlassen wollen, die in Zusammenhang mit deren privaten Lesezeichen stehen.

3.2 Synchronisation der Icons

Das zweite Ziel ist die Synchronisierung der Icons mit allen Chrome-Browsern auf Desktop Computern. Dies wird dadurch notwendig, dass die Icons ja jetzt nicht mehr im Web gespeichert sind und dadurch bei einer Synchronisierung der Linkdaten nicht mehr automatisch abrufbar sind, da für diese ja keine öffentliche URL mehr existiert.

Kein Problem wäre dies wieder mit einem eigenen Datenspeicherdienst für diese Bilder im Web, da dieser dann die öffentliche URL bereitstellen würde, allerdings wurde diese Variante ja schon im vorherigen Kapitel über die lokale Speicherung der Icons wegen der vielen zusätzlichen Probleme verworfen.

Die bessere und genauso einfache Variante ist hier die Nutzung der sogenannten Sync Filesystem API von Chrome. Diese API stellt dem Nutzer ein Dateisystemobjekt zur Verfügung, welches auf einen Ordner zeigt, dessen Inhalt mit einem Ordner in der Google Drive Cloud des Nutzers synchronisiert ist. Voraussetzung dafür ist natürlich eine erfolgte Anmeldung in Google Chrome durch den Nutzer, weshalb ohne Anmeldung auf die Chrome Local Storage API zurückgegriffen wird, welche im vorherigen Kapitel als beste Lösung für die lokale Speicherung herausgefunden wurde.

Bei Nutzung der Sync File System API von Chrome wird auf Speicherplatz zurückgegriffen, welcher dem Nutzer bereits zugeordnet wurde. Dadurch spart man sich als Entwickler den Aufwand für einen eigenen Datenspeicher im Web und bekommt gleich noch die Synchronisierungsmechanik gratis, welche auch die meisten Synchronisierungskonflikte automatisch oder mit wenig Quellcode beheben kann.

Natürlich muss man sich trotzdem noch selbst um die Auflösung größerer Konflikte kümmern. Speziell der Fall, wenn sich ein Nutzer zum ersten Mal in Chrome anmeldet, die Anwendung sich synchronisieren will, aber schon Daten in dem Sync File System vorhanden sind, muss abgefangen werden. Eine zusätzliche Schwierigkeit kommt hinzu, wenn der Nutzer vorher einige Zeit ohne Synchronisation, das heißt im Fallback Modus mit der Chrome Local Storage API gearbeitet hat.

Dann muss die Anwendung den Nutzer auffordern, eine Entscheidung zu treffen, wie der Synchronisationskonflikt behoben werden kann und evtl. die lokalen Daten mit dem synchronisierten Stand zusammenfügen.

Ein weiterer „Nachteil“ dieser API ist auch, dass sie nur für Chrome Desktop Apps freigegeben ist, sie sozusagen von normalen Chrome Erweiterungen nicht genutzt werden kann, da sie sonst zu leicht missbraucht werden könnte.

Aus diesem Grund wurde die Applikation auch in ein Hauptmodul und ein Dienstmodul getrennt, wobei Zweiteres technisch als Chrome App ausgeführt ist und deshalb auf besagte API zugreifen darf.

4 Das Hauptmodul „chrome–speedpad“

Das Hauptmodul „chrome-speedpad“ ist die Chrome Erweiterung des Systems. In Abbildung 2 sieht man ein Bildschirmfoto mit einer beispielhaften Startseite. Die Abbildungen 3 bis 6 zeigen das Anlegen einer neuen Verknüpfung.

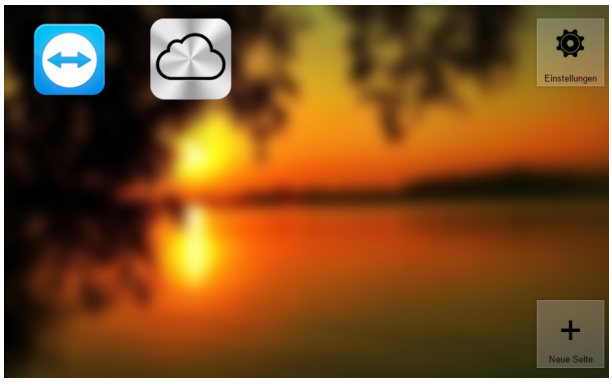

Abb. 02: Chrome Speedpad - Hauptbildschirm

Abb. 03: Chrome Speedpad - Neue Seite anlegen

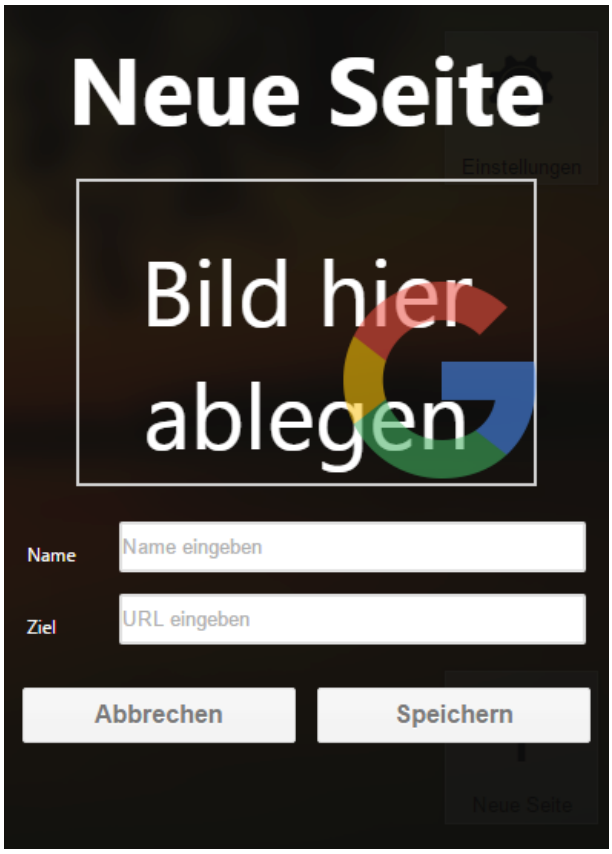

Abb. 04: Einfügen des Icons per Drag and Drop

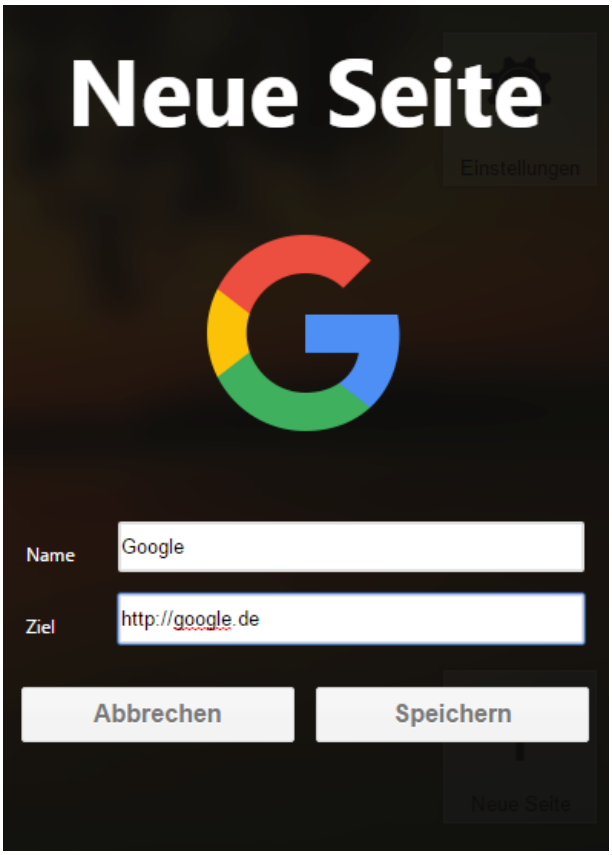

Abb. 05: Details der Verknüpfung ausfüllen

4.1 Das Design

Da die Applikation noch nicht final ist, ist die Funktionalität an einigen Stellen noch unvollständig oder gar nicht vorhanden. Dies wird auch in dem teilweise nicht harmonierenden Design sichtbar, was aber nicht davon abhalten soll, die generelle Designperspektive zu beleuchten.

Eine Schnellstartseite ist ein sehr persönliches Werkzeug zur Navigation im Web. Sie ist eine Seite, die sehr häufig aufgerufen wird und sollte deshalb sowohl funktional sein, als auch optisch ansprechend, um praktisch und mental bei der Arbeit zu unterstützen. Dazu wurde ein Bildschirmfüllendes Hintergrundbild verwendet, welches später von dem Benutzer noch geändert werden kann. Der leichte Unschärfeeffekt sorgt dafür, dass die Symbole für die einzelnen Links auch auf schwierigen Hintergründen noch leicht zu sehen und zu treffen sind.

Die Darstellung der Links soll in großen Teilen personalisierbar werden, vor allem bezogen auf die Fragen:

- Soll der Titel des Links angezeigt werden und wenn ja, wo?
  (unter dem Icon, über dem Icon, usw.)
- Wie groß sollen die Icons angezeigt werden?
- Sollen die Icons einen generierten Hintergrund bekommen, um sie noch besser vom Hintergrund abzuheben?

Zur besseren Strukturierung sind noch Ordner und Tabs geplant, diese funktionieren dann ähnlich wie der Startbildschirm unter Android oder iOS. Dort gibt es ja auch Ordner und ein oder mehrere Startbildschirme, wobei letztere dann durch die Tabs symbolisiert werden würden.

Wichtig sind zusätzlich noch verschiedene kleine Details, die das Arbeiten mit dieser Anwendung entsprechend einfacher gestalten. Ein Beispiel dafür ist der Import des Bildes für einen neuen Link.

Anstatt einen komplizierten Auswahldialog anzuzeigen, kann das Bild einfach per Drag and Drop von der eigenen Festplatte oder direkt aus dem Internet eingefügt werden. Ein zweites Beispiel dafür ist, dass man die Seitenleisten für Einstellungen oder neue Links einfach wegklicken kann, indem man bei geöffneter Seitenleiste auf eine freie Fläche außerhalb der Seitenleiste klickt.

4.2 Die Technik – AngularDart

Da sich an diesem Punkt die Entwicklung einer Chrome Erweiterung nicht mehr von der Entwicklung einer normalen Single-Page-Application(SPA) unterscheidet, kann man auch alle Tools und Frameworks benutzen, welche man normalerweise für die Entwicklung einer SPA einsetzt. In diesem Fall wurde als Basis für die Oberfläche das AngularDart Framework benutzt. [Q10] AngularDart ist ein Framework, welches verschiedene Möglichkeiten zur sinnvollen Strukturierung einer SPA zur Verfügung stellt.

Die grundlegenden Bausteine einer AngularDart Anwendung sind Components. Components sind wiederverwertbare Komponenten, die in sich abgeschlossen sind, was bedeutet, sie besitzen ihr eigenes Gerüst als HTML, ihre eigene Logik in Form von Dart Quellcode und ihre eigenen Stilinformationen in Form von eingebetteten <style>-Tags, Inline Styles oder CSS-Dateien. Im HTML werden sie einfach als Tag eingebunden, der Tagname ist ihr Selektor. Sie können auch wie normale HTML-Tags mit Attributen ergänzt werden, auf welche dann im Script der jeweiligen Komponente zugegriffen werden kann. *Listing 02 auf Seite 14* zeigt den grundlegenden Aufbau der Komponente, welche den Button zum Hinzufügen von neuen Verknüpfungen zum Chrome Speedpad repräsentiert.

```dart
library add_button;
import „dart:html“;

import „package:angular/angular.dart“;
import „package:event_bus/event_bus.dart“;
import „package:speedpad/event_classes/events.dart“;

@Component(
    selector: „add-button“,
    templateUrl: „packages/speedpad/add_button/add_button.html“,
    useShadowDom: false
)
class AddButton {
  EventBus _eventBus;

  AddButton(this._eventBus) {}

  void addLink (MouseEvent e) {
    e.stopImmediatePropagation();
    _eventBus.fire(new OpenGenericPanel(PanelView.NewLink));
  }
}
```

Listing 01: Aufbau einer AngularDart-Komponente am Beispiel der AddButton-Komponente

Das wichtigste hier ist die Annotation @Component, die durch das AngularDart Framework bereitgestellt wird. Der Selektor bestimmt den Tag Namen der Component im HTML, und muss nach HTML5 Spezifikation einen Bindestrich enthalten, um es als nutzerdefinierte Komponente zu kennzeichnen. templateUrl erlaubt das Hinzufügen einer HTML Datei, welche die innere HTML Struktur der Komponente enthält. Stattdessen kann man für kleinere Komponenten auch direkt das template Schlüsselwort verwenden und als Wert einen String mit der entsprechenden Struktur übergeben. UseShadowDom sagt dem Framework, dass für diese Komponente kein ShadowDom zum Einsatz kommen soll.

Die Komponenten von AngularDart werden intern in sogenannte WebComponents umgesetzt. Die WebComponent Spezifikation besteht aus den neuen Web APIs Custom Elements, HTML Imports, Templates und Shadow Dom. [Q11] Die Custom Elements API erlaubt jedem Web Programmierer, eigene HTML Tags hinzuzufügen, die HTML Import API erlaubt das importieren einer HTML Datei über ein <link> Tag mit einem type=“html“ Attribut, die Template API erlaubt das Deklarieren von HTML Teilbäumen und erlauben die einfache mehrfache Instanziierung, und Shadow Dom stellt eine Kapselung um die eigenen Komponenten bereit, damit z.B. Style Definitionen nicht von innen nach außen, aber vor allem nicht von außen nach innen gelangen, da CSS Klassen und IDs ja normalerweise global sind. Für einige Komponenten in diesem Projekt wurde Shadow Dom aber abgeschaltet, weil zurzeit noch eine einfache Möglichkeit fehlt, umfassendes Styling der Komponente außerhalb bereitzustellen.

Alle visuellen Komponenten der Erweiterung sind nach diesem Muster gebaut. Es können aber auch unsichtbare Komponenten entwickelt werden, die z.B. Datentransformationen übernehmen oder Elemente gruppieren und anordnen. Solche Elemente können auch als Attribut an ein HTML Element gehängt werden, um dessen Funktionsweise zu verändern oder zu ergänzen. Zuletzt gibt es noch die Services, auf Deutsch: „Dienste“. In diesen wird die Programmlogik strukturiert abgelegt, sie stellen z.B. Schnittstellen mit externen Systemen bereit, abstrahieren Datenspeicher oder kümmern sich um notwendige Hintergrundarbeiten.Die folgende Abbildung zeigt alle Komponenten, welche von der Chrome Speedpad Erweiterung benutzt werden. Die Ordner event_ classes, global und util gehören nicht dazu.

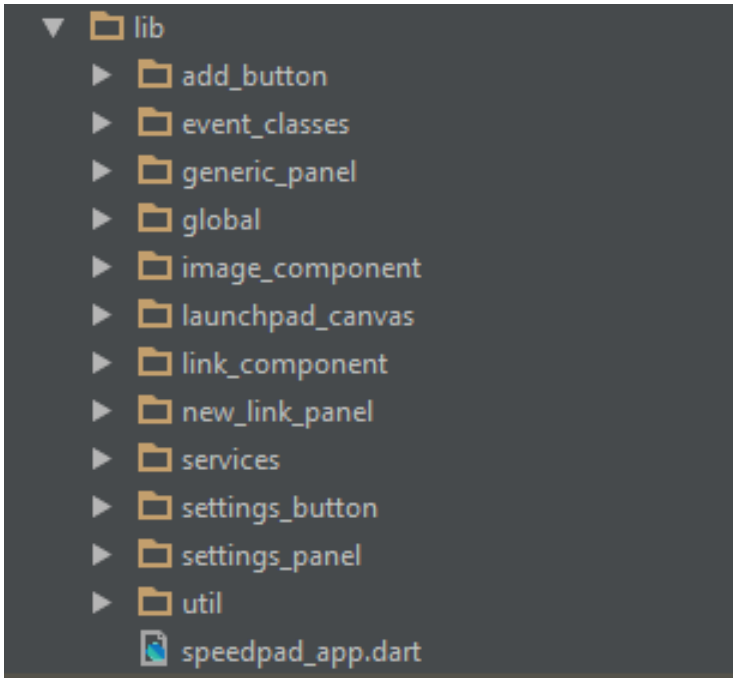

Abb. 06: Chrome Speedpad - Komponentenübersicht

4.3 Datenfluss zwischen den Komponenten

Ein interessanter Aspekt ist die Kommunikation zwischen den einzelnen Komponenten. In Swing aus der Java Welt wird diese Kommunikation über das Observer-Observable Pattern abgewickelt. Dies könnte hier natürlich genau so implementiert werden, allerdings wäre es sehr umständlich, umständlicher als es sein müsste. Das Observer Pattern zwingt den Entwickler, viel unnötigen, aufgeblähten Code, sogenannten Boilerplate Code, zu schreiben, wie z.B. die Listen in jedem Observable für unterschiedliche Typen an Observern. Eine andere Lösung ist hier die Verwendung eines Event Bus Systems. Ein Event Bus stellt eine Publish-Subscribe Funktionalität zur Verfügung und fördert so die lose Kopplung zwischen den Elementen. Nachteilig daran ist natürlich, dass man nicht mehr so einfach im Quellcode sehen kann, wo ein spezielles Event ausgewertet wird bzw. wo ein spezielles Event herkam.

Diesen Teil muss dann eine gute Dokumentation übernehmen, falls mehrere Programmierer an diesem Projekt arbeiten. Der eindeutige Vorteil ist allerdings, dass man sich nicht mehr darum kümmern muss, verschiedene Event-Objekte durch die ganze Anwendung zu schleifen. Man legt es einfach auf den Event Bus, und alle Observer suchen sich ihr Event Objekt heraus. Wie das funktioniert, soll an dem Anlegen eines neuen Links gezeigt werden.

Um einen neuen Link anzulegen, klickt der Benutzer zuerst auf die Schaltfläche „Neue Seite“. Der entsprechende Event Handler „addLink“ ist in *Listing 01 auf Seite 12* zu sehen. Dort wird ein „OpenGenericPanel“-Event erzeugt, welchem als Parameter ein Wert eines Enums mitgegeben wird, welcher bestimmt, welcher Inhalt in die Seitenleiste geladen werden soll. Dies ist zur Wiederverwendung des Codes der Seitenleiste gedacht, da man ansonsten die ganzen Event Handler für die Interaktion mit der Seitenleiste für jeden unterschiedlichen Inhalt der Seitenleiste duplizieren müsste. Dieses „OpenGenericPanel“-Event wird nun direkt in der Komponente des GenericPanel aufgefangen und ausgewertet, was in *Listing 02 auf Seite 14* dargestellt ist.

```
//react on open events on the event bus
_eventBus.on(OpenGenericPanel).listen((OpenGenericPanel e) {

  if (e.view != currentPanel) {

    switch (e.view) {
      case PanelView.Settings:
        currentPanel = PanelView.Settings;
        break;
      case PanelView.NewLink:
        currentPanel = PanelView.NewLink;
        break;
      case PanelView.None:
        currentPanel = PanelView.None;
        break;
    }
  }
```

Listing 02: Das Abfangen und Reagieren auf das OpenGenericPanel-Event in der GenericPanel -Komponente

Die Variable currentPanel wurde hier als Bedingung zum Austausch des Inhalts des Panels verwendet. Das AngularDart DataBinding sorgt für den weiteren Austausch, als Inhalt wird nun eine NewLinkPanel-Component verwendet.

Danach ist das Panel offen und kann ausgefüllt werden. Dies ist in der *Abb. 07 auf Seite 17* dargestellt. Nachdem alles ausgefüllt ist, klickt der Benutzer auf Speichern. Dabei wird die save-Funktion der NewLinkPanel-Component verwendet, welche in *Listing 03 auf Seite 15* zu sehen ist.

```
void save() {
  if (nameBuffer.isEmpty) {
    missingName = true;
    return;
  } else {
    missingName = false;
  }

  if (urlBuffer.isEmpty) {
    missingUrl = true;
    return;
  } else {
    missingUrl = false;
  }

  if (imgBuffer.isEmpty) {
    //TODO: add empty image placeholder image
    //TODO: add possibility for website screenshot later
  }

  //Pack all Buffer vars to a LinkEntry and give it to _storage.save
  var linkEntry = new LinkEntry(nameBuffer, parentTab, urlBuffer, imgBuffer);
  _eventBus.fire(new SaveLink(linkEntry));
  _eventBus.fire(new CloseGenericPanel(PanelView.NewLink));
}
```

Listing 03: Die save() Funktion der NewLinkPanel-Komponente

Hier wird zuerst überprüft, ob alle Felder richtig ausgefüllt sind, danach werden alle wichtigen Daten in ein LinkEntry Objekt zusammengepackt, welches danach dem Event Objekt SaveLink übergeben wird. Zum Schluss wird noch das Event CloseGenericPanel angestoßen, damit sich die Seitenleiste wieder schließt. Dieses Event wird im GenericPanel abgefangen.

Das interessantere Event ist SaveLink. Dieses wird an zwei Stellen abgefangen. Die erste ist die LaunchpadCanvas Komponente, die zweite ist der Dienst zur Verwaltung der gespeicherten Linkdaten – StorageService. In der LaunchpadCanvas Komponente wird das neue LinkEntry Objekt dazu benutzt, die Anzeige der Links zu aktualisieren. Dadurch wirkt die Anwendung schön flüssig.

Im Hintergrund wird das LinkEntry Objekt durch den StorageService an das Dienstmodul speedpad-service weitergegeben, da nur dieses die persistente Speicherung vornehmen kann. Einen genaueren Blick auf die Kommunikation zwischen dem Hauptmodul und dem Dienstmodul kann man in *„Kommunikation zwischen „chrome-speedpad“ und „chrome-service““ auf Seite 19* weiter unten lesen.

5 Das Dienstmodul „speedpad–service“

Das Dienstmodul „speedpad-service“ besteht nur aus einem Hintergrundscript. Theoretisch kann es auch eine in HTML geschriebene Oberfläche besitzen, da es aber nur für die Datenspeicherung zuständig ist, ist dies nicht notwendig.

Es beantragt im Manifest die Rechte für unbegrenzten Speicher, Zugriff auf die Chrome Storage APIs, wie Chrome Storage Local und Chrome Storage Sync, sowie Zugriff auf das Sync File System. Sollte das Sync File System nicht erreichbar sein, weil der Nutzer nicht in Chrome angemeldet ist, wird als Rückfallmöglichkeit die Chrome Storage Local API benutzt. Der Vorteil dieser ist, dass der mögliche Speicherplatz größer als nur 5 MB ist, solange das Recht für unlimitierte Speicherung gewährt wurde.

Der Aufbau der Speicherung ist relativ simpel. Es gibt eine Hauptdatei, die „structure.json“. Diese enthält eine Liste aus TabEntry Objekten und stellt damit die oberste Hierarchieebene der Links dar. Die TabEntry Objekte besitzen immer einen Namen, der für die Anzeige der Tabs genutzt wird. Optional können sie ein Vorschaubild oder ein Icon besitzen. Der Name dient gleichzeitig auch der Identifizierung aller zugehöriger Dateien zu diesem Tab. Dazu wird der MD5 Hash aus dem Namen gebildet, womit alle Zeichen entfernt werden, die evtl. ungültig sind für Dateinamen. Damit werden Probleme mit Leerzeichen, Umlauten und Sonderzeichen umgangen. Dieser Hash wird im Folgenden als Tab ID bezeichnet.

Damit nicht immer sofort alle Linkdaten gleichzeitig übertragen werden müssen, werden die Linkdaten in separaten Dateien abgelegt. Der Dateiname dieser Dateien entspricht der Tab ID. Inhaltlich werden in diesen Dateien einfach nur Listen aus LinkEntry Objekten gehalten. Ein LinkEntry Objekt besitzt einen Namen, der z.B. als Titel des Links angezeigt werden kann, ein Vorschaubild, welches hier als Base64 kodierter String vorliegt, eine URL, wohin der Link zeigt, und natürlich eine Referenz des TabEntry Objektes, zu welchem der Link gehört. Diese Referenz wird benötigt, um neue Links oder Links, die bearbeitet wurden, dem richtigen Tab zuordnen zu können. Die geplanten Ordner-Elemente, die zur weiteren Gruppierung der Links auf den einzelnen Seiten gedacht sind, sind noch nicht implementiert.

6 Das Hilfsmodul „shared–classes“

Das Hilfsmodul „shared-classes“ ist eine Dart-Programmbibliothek, welche von den beiden anderen Modulen eingebunden wird. Da beispielsweise die Objekte der Datenspeicherung sowohl im Hauptmodul, als auch im Dienstmodul gebraucht werden, sind diese hier im Hilfsmodul definiert. Abbildung 7 zeigt die Dateien im Hilfsmodul „shared-classes“. Der Ordner data_classes enthält die entsprechenden Represäntationsklassen für Tabs, Ordner und Links. Die speedpad_entry.dart ist die Basisklasse für die anderen drei.

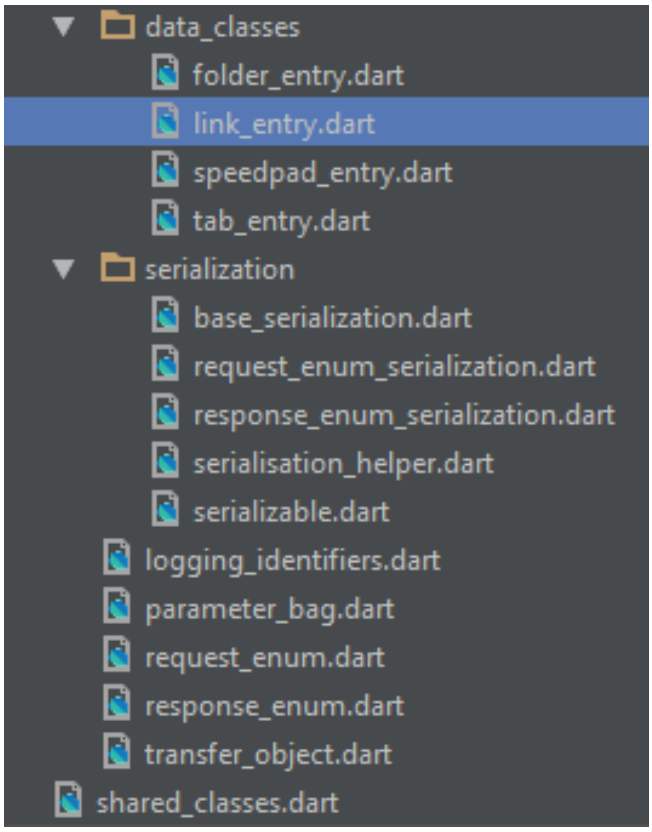

Abb. 07: Inhalt des Hilfsmoduls shared-classes

6.1 Serialisierung in Dart

Serialisierung in Dart ist ein nicht ganz einfaches Thema. Für eine einfache Serialisierung zu JSON besitzt Dart zwar einen JSON Codec, der die benötigten JSON.encode() bzw. JSON.decode() Methoden zur Verfügung stellt. Leider funktionieren diese nur mit primitiven Werten, Listen und Maps. Um beliebige Objekte nach JSON kodieren zu können, müssen vom Programmierer entsprechende toJSON Methoden bereitgestellt werden, welche die spezifischen Felder eines Objektes als Map abbilden. Schwierig ist allerdings das Auslesen, da JSON.decode() nur Maps und Listen zurückgeben kann, statt eines komplett typisierten Objektes.

Grundsätzlich kann man diese Schwierigkeiten auf das Fehlen einer performanten Reflection zurückführen. JSON Bibliotheken in Java suchen sich die benötigten Felder für eine Serialisierung einfach per Reflection selbst heraus und können so die Inhalte von Variablen speziellen Schlüsseln im JSON zuordnen. Da Reflection aber in JavaScript sehr schwierig und damit langsam ist, ist dies keine praktikable Lösung für Dart.

Die Lösung in diesem Projekt ist die sogenannte „serialization“ Programmbibliothek in Dart. Diese erlaubt es, eine Modellklasse mit Funktionen für die Enkodierung und die Dekodierung zur Verfügung zu Stellen. *Listing 04 auf Seite 18* zeigt die Basisklasse für die Serialisierung in diesem Projekt.

Eine solche spezielle Regel für die Serialisierung eines Typs erfordert das Überschreiben verschiedener Methoden aus dem Basistyp „CustomRule“. Die appliesTo-Methode testet, ob ein gelesenes oder übergebenes Objekt zu dieser Regel passt oder nicht. Die Methode getState() ruft automatisiert auf dem zu serialisierenden Objekt die toMap() Methode auf, welche als interne Konvention für jede serialisierbare Klasse verfügbar sein muss.

```
class BaseSerialisation<Type extends Serializable> extends CustomRule {
  Function _createImplementation;

  BaseSerialisation(this._createImplementation(Map state));

  bool appliesTo(instance, Writer w) => instance.runtimeType == Type;

  getState(Serializable instance) => instance.toMap();

  void setState(object, List state) {
    //ignore -nothing to do
  }

  create(state) => _createImplementation(state);
}
```

Listing 04: Die allgemeine Klasse zum Erzeugen von Serialisierungsregeln

Für die Umkehrung kann man einen benannten Konstruktor von Dart verwenden. Unglücklicherweise gibt es keine Möglichkeit, eine Klasse ohne Reflection zu instanziieren, die zur Kompilierzeit noch nicht bekannt ist. Aus diesem Grund bekommt der Konstruktor dieser BaseSerialization-Klasse einen Lambda-Ausdruck übergeben, welche das Anlegen des entsprechenden Objekts übernimmt. Dieser Lambda-Ausdruck wird dann in der create() Methode aufgerufen und bekommt den State als Map übergeben. Dadurch kann die zu serialisierende Klasse einfach einen Class.fromMap() Konstruktor bereitstellen, der die restliche Arbeit übernimmt. Aufrufe dieses BaseSerialisation-Konstruktors ist in *Listing 05 auf Seite 19* zu sehen.

```
final Serialization _serialization = new Serialization()
  ..addRule(new BaseSerialisation<TabEntry>((state) => new TabEntry.
fromMap(state)))
  ..addRule(new BaseSerialisation<LinkEntry>((state) => new LinkEntry.
fromMap(state)))
  ..addRule(new BaseSerialisation<FolderEntry>((state) => new FolderEntry.
fromMap(state)))
  ..addRule(new BaseSerialisation<TransferObject>((state) => new TransferObject.
fromMap(state)))
  ..addRule(new BaseSerialisation<ParameterBag>((state) => new ParameterBag.
fromMap(state)))
  ..addRule(new RequestEnumSerialization())
  ..addRule(new ResponseEnumSerialization());
```

Listing 05: Erzeugung der konkreten Serialisierungsregeln mittels der BaseSerialisation-Klasse

7 Kommunikation zwischen „chrome–speedpad“ und „chrome–service“

Die Übertragung von Daten zwischen dem Hauptmodul und dem Dienstmodul erfolgt hier über eine Schnittstelle in der Chrome Runtime, welche Teil der Chrome Messaging API ist. Die Chrome Messaging API erlaubt es Erweiterungen, Chrome Apps und sogenannten ContentScripts miteinander zu kommunizieren. ContentScripts sind JavaScript Dateien, welche im Kontext einer speziellen Webseite ablaufen, um diese Webseite mit erweiterten Funktionalitäten auszustatten. Es gibt zwei Möglichkeiten, die Chrome Messaging API zu benutzen. Die erste Möglichkeit ist ein Nachricht-Antwort Zyklus. Eine Komponente sendet eine Nachricht über Chrome Runtime, die zweite Komponente empfängt diese Nachricht und sendet direkt eine Antwort zurück. Die Zweite Möglichkeit zur Kommunikation, welche auch in dieser Anwendung benutzt wird, ist die Möglichkeit, eine sogenannte „Long-lived Connection“, also eine langlebige Verbindung, aufzubauen. Bei dieser Methode öffnet eine Seite eine Verbindung und die zweite Seite akzeptiert dies. Beide Seiten besitzen dann ein Port Objekt, welches eine Methode zum Senden, und einen Stream aus empfangenen Nachrichten bereitstellt.

Die Nachricht selbst wird als String übertragen. Sollte ein Objekt übertragen werden, welches kein String ist, wird es nach JSON konvertiert und danach übertragen. Da die JavaScript APIs von Chrome kein Wissen über die darüber liegenden typisierten Dart Objekten haben, werden alle nicht-String Objekte vorher selbst serialisiert. Dies erfolgt mithilfe der „serialize“-Bibliothek und den Serialisierungsregeln, welche in Abschnitt 6.1 vorgestellt wurden.

7.1 Übertragungsprotokoll

Das eigentliche Übertragungsprotokoll ist properitär, da die Chrome Messaging API nur das Verschicken von JSON Nachrichten beherrscht. Das grundlegende übertragene Objekt ist ein Objekt der Klasse TransferObject, welche im Hilfsmodul „shared-classes“ zu finden ist, da sie auf beiden Seiten der Kommunikation benötigt wird. Diese Klasse beinhaltet zwei benannte Konstruktoren, namentlich TransferObject.Request und TransferObject.Response. Diese Konstruktoren bekommen jeweils einen Wert aus einem RequestType-Enum bzw. aus einem ResponseType-Enum. Das RequestType-Enum enthält alle Befehle, die der Hauptteil der Anwendung an das Dienstmodul schicken kann, das ResponseType-Enum enthält dementsprechend die möglichen Antworten.

Der übergebene Wert wird dann in einem Feld namens params vom Typ ParameterBag in der TransferObject-Klasse gespeichert. Damit wird der Empfangsseite des jeweiligen Objektes ermöglicht, das Objekt richtig auszuwerten. ParameterBag ist auch eine eigene Klasse, welche ein Set an Parametern für einen Anfragebefehl oder eine Antwort aufnehmen kann.

Der Vorteil dieser Architektur liegt darin, dass nicht für jeden Befehl eine eigene Klasse zur Übertragung des Befehls mit seinen Parametern implementiert werden muss. Normalerweise klingt das zwar auch nach einer guten Lösung, da man so Fehler durch die genauere Typisierung schneller erkennt. Aber da die Serialisierungslösung in dieser Anwendung für jeden serialisierbare Klasse eine Regel zum Serialisieren benötigt, stiege der Aufwand bei vielen Kommandoklassen um einiges an. So kann die Empfangsseite je nach gespeichertem Request- oder ResponseType die entsprechenden Parameter selbst aus dem ParameterBag extrahieren.

7.2 Verbesserung der Geschwindigkeit der Anwendung

Leider ist die Anwendung zurzeit nicht besonders schnell und reaktiv. Bei einem Test der Hauptanwendung in den Chrome Developer Tools konnte man feststellen, dass die Anwendung bei jedem Neuladen der Oberfläche längere Wartezeiten durchläuft. „Längere Wartezeiten“ bezieht sich hier jetzt auf eine bis mehrere Sekunden, was im Kontext einer Schnellstartseite schon extrem lange ist. Diese Wartezeiten resultieren vermutlich auf der Kommunikation zwischen „chrome-speedpad“ und dem „speedpad-service“, welche aus mehreren Gründen nicht performant sein könnte.

Der erste Grund liegt in der Architektur von Chrome. Jede Webseite, jede Erweiterung und jede Chrome App bekommt einen eigenen Prozess für die Kommunikation. Das bedeutet, dass die Chrome Messaging API ihre Nachrichten per Inter-Process-Communication (IPC) verschicken muss, was nicht die schnellste Lösung sein dürfte. Dazu kommt noch der Overhead aus der Serialisierung der Objekte zu JSON sowie die Zeiten, die zum Lesen und Schreiben von Dateien auf dem Sync File System benötigt werden.

Eine Lösung für dieses Problem wäre der Austausch des Übertragungskanals. Statt einer IPC zwischen den Modulen der Anwendung könnte es effizienter sein, den „speedpad-service“ einen Webserver implementieren zu lassen und das Hauptmodul ganz normal per HTTP daruf zugreifen zu lassen. Der Overhead dieser Lösung sollte schon kleiner sein, als die jetzige Lösung, aber die Verwendung von zustandslosem HTTP für eine lokale Verbindung klingt nicht viel besser, im Hinblick auf die Leistung.

Eine bessere Lösung ist eine weitere API, welche von Chrome bereitgestellt wird. Diese nennt sich chrome.sockets und stellt schlanke TCP/IP Sockets zur Verfügung. Damit hätte man einen schlanken bi-direktionalen Datenkanal, auf welchem man beispielsweise mit dem „Tether-Protokoll“ operieren könnte. Das „Tether Protokoll“ ist eine Nachrichtenabstraktion für bi-direktionale Kommunikation, welches auf JSON basiert. [Q12] Glücklicherweise gibt es außerdem eine Dart Bibliothek zur einfachen Implementierung in die eigene Anwendung.

Ein weiterer Punkt zur Geschwindigkeitsoptimierung ist die Haupterweiterung selbst. Für jeden neuen Tab, der vom Benutzer geöffnet wird, wird eine neue Übertragung zum Dienstmodul aufgebaut. Dieses Verhalten könnte man minimieren, indem man ein Hintergrundscript in „chrome-speedpad“ laufen lässt, welches permanent die Verbindung zum Datendienst hält. Nebenbei könnte man an dieser Stelle auch sinnvoll eine Zwischenspeicherung der Daten implementieren.

8 Fazit

Die Entwicklung dieses Anwendungssystems war eine sehr interessante Aufgabe. Es wurden Lösungen für die Ziele der lokalen Datenspeicherung und der synchronisierten Datenbasis ausgearbeitet und in einer ersten Version implementiert. Im Folgenden können die Verbesserungen aus Kapitel 7.2 implementiert werden.

9 Verzeichnisse

9.1 Abbildungsverzeichnis

Abb. 01: Schnellstartseite in Google Chrome .png 4

Abb. 02: Chrome Speedpad - Hauptbildschirm.png 9

Abb. 03: Chrome Speedpad - Neue Seite anlegen.png 9

Abb. 04: Einfügen des Icons per Drag and Drop.png 10

Abb. 05: Details der Verknüpfung ausfüllen .png 10

Abb. 06: Chrome Speedpad - Komponentenübersicht.png 13

Abb. 07: Inhalt des Hilfsmoduls shared-classes.png 17

9.2 Listingverzeichnis

Listing 01: Aufbau einer AngularDart-Komponente am Beispiel der AddButton-Komponente . 12

Listing 02: Das Abfangen und Reagieren auf das OpenGenericPanel-Event in der GenericPanel -Komponente 14

Listing 03: Die save() Funktion der NewLinkPanel-Komponente 15

Listing 04: Die allgemeine Klasse zum Erzeugen von Serialisierungsregeln 18

Listing 05: Erzeugung der konkreten Serialisierungsregeln mittels der BaseSerialisation-Klasse19

9.3 Quellenverzeichnis - Informationsquellen

[Q1] http://www.typescriptlang.org/

[Q2] http://www.dartlang.org

[Q3] https://pub.dartlang.org/packages/chrome

[Q4] https://www.dartlang.org/

[Q5] https://www.dartlang.org/docs/tutorials/futures/

[Q6] https://developer.chrome.com/extensions/api_index

[Q7] https://developer.chrome.com/extensions/manifest

[Q8] https://developer.chrome.com/extensions/api_other

[Q9] https://developer.chrome.com/extensions/storage
https://developer.chrome.com/apps/app_storage
https://developer.chrome.com/apps/fileSystem
https://developer.chrome.com/apps/syncFileSystem
http://www.html5rocks.com/en/tutorials/webdatabase/websql-indexeddb/?redirect_from_locale=de

[Q10] https://angulardart.org/

[Q11] http://webcomponents.org/, im Kasten „Spec“

[Q12] https://pub.dartlang.org/packages/tether

10 Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

Leipzig, 14.01.16 Unterschrift:_____