Projektarbeit

Praktikumsersatzleistung

Titel: *Entwicklung einer Browser-Extension zur Analyse besuchter Internetseiten*

Institut für Informatik
Studienrichtung: Informatik
[Sommersemester 2018]

vorgelegt von:	Lars Kleinsteinberg
Matrikelnummer:	899391
Telefon:	0157 / 89358391
Eingereicht bei:	Marco Wiemer
Abgabetermin:	03.09.2018
Bearbeitungszeit:	fünf Monate

Inhaltsverzeichnis

Abbildungsverzeichnis

Ehrenerklärung

Ich, Lars Kleinsteinberg, versichere durch meine Unterschrift, dass ich den vorliegenden Bericht selbstständig erstellt habe. Andere als die angegebenen Hilfsmittel habe ich nicht verwendet. Soweit ich fremde Gedankengänge oder Texte verwendet habe, sind diese von mir als solche kenntlich gemacht und dem Urheber eindeutig zuordenbar. Dazu zählen sowohl wörtliche als auch nicht wörtliche Übernahmen.

Münster, 17.09.2018
Ort, Datum

Lars Kleinsteinberg
Unterschrift

1. Einleitung

Die hier dokumentierte Projektarbeit wurde im Rahmen des Bachelor-Studiengangs Informatik der Wilhelm-Büchner Fern-Hochschule erbracht. Während des Informatik Studiums ist es erforderlich eine Berufs-praktische-Phase abzuleisten. Allerdings kann dieses Praktikum unter bestimmten Umständen durch eine Praktikums-Ersatzleistung ersetzt werden. Bei einer solchen Ersatzleistung muss der Student eigenständig ein Software-Projekt erarbeiten und zusammen mit einem Projektbericht einreichen um zu bestehen.

Die vorliegende Projektarbeit erfüllt die Funktion einer solchen Praktikumsersatzleistung erbracht. Der Autor konnte wegen einer fachfremden Festanstellung kein reguläres Praktikum bestreiten. Marco Wiemer von der Wilhelm-Büchner-Hochschule begleitete das Projekt als Betreuer / Tutor.

Der Bearbeitungszeitraum begann am 06.03.2018 und endete am 06.08.2018 (als spätester Abgabetermin war der 06.09.2018 angegeben). Die Arbeit ist mit 15 CP gewichtet und wird bei Bestehen auf dem Bachelor-Zeugnis als „mit Erfolg teilgenommen“ dokumentiert.

Das Thema der Projektarbeit: „ ***Entwicklung einer Browser-Extension zur Analyse besuchter Internetseiten“*** wurde vom Autor selbst gewählt und durch das Dekanat der Wilhelm-Büchner Hochschule geprüft und bestätigt.

Um diese abstrakte Zielvorgabe in die Tat umzusetzen musste der Autor allerlei neue Technologien verstehen und anwenden. Die Erkenntnisse der Recherchen zu den Themen „Browser-Extensions“ und „Scraping“ stehen am Anfang des Projektberichts.

Das erstellte Add-on soll in der Lage sein, aus dem Bedienungs-Verhalten eines Benutzers eines Web-Browsers, Daten zu generieren und diese für eine spätere Verwendung zu speichern. Technisch bedeutet dies, dass während des Surfens ständig die Position des Mauszeigers auf der gerade aktiven Internet-Seite beobachtet werden soll. Aus der Position des Zeigers, soll dann das unter dem Zeiger liegende DOM-Element ermittelt und gespeichert werden.

Der Bericht stellt zunächst die technologischen Besonderheiten einer Browser-Extension vor. Ausserdem wird das Prinzip des Scraping vorgestellt, da sich heraus stellte, dass das Projekt zur entsprechenden Klasse von Anwendungen gehört.

Die eigentliche Planung baut auf den Ergebnissen der technischen Recherche auf und stellt eine Verbindung zwischen der angestrebten Funktionalität und den Entwicklungs-Werkzeugen her. Die Ergebnisse dieser Überlegungen führen zu einer Reihe von Meilensteinen welche zur Umsetzung des Projekts abzuarbeiten sind.

Es folgt eine detaillierte Beschreibung der einzelnen Entwicklungsschritte mit Ausschnitten des erstellten Quellcodes und Erklärungen.

Aus der Perspektive des abgeschlossenen Projekts wird auf potenzielle Anwendungsmöglichkeiten für verschiedene Zielgruppen und Rahmenbedingungen eingegangen. Speziell die weitere Verwendung der Ergebnisse in einer anschliessenden Bachelor-Arbeit werden besprochen.

Zum Ende des Berichts folgt eine Schlussbetrachtung, bzw. Resümee des Projekts und seines Verlaufs. Dabei wird auf die Herausforderungen während der Umsetzung eingegangen, auf den Erkenntnisgewinn für den Autor und es werden Bezüge zu den bisherigen Studieninhalten hergestellt.

2. Grundlagen

Im folgenden werden die für die Umsetzung und das Verständnis der Projektarbeit notwendigen Grundlagen vorgestellt. Für das Thema Browser-Extensions wird ein Überblick über den Stand der Technik gegeben. Des weiteren wird auf das Thema „Scraping“ eingegangen um dem Leser die thematische Einordnung, bzw. den Vergleich mit anderen Produkten zu erleichtern.

2.1. Browser-Extensions

Browser Extensions(auch Add-ons oder Plug-ins genannt) sind Programme, welche die Funktionalität von Webbrowsern erweitern oder modifizieren. Dabei kann sowohl die Funktionalität und das Aussehen des Browsers selbst, als auch Funktionalität und Aussehen der dargestellten Inhalte (Content) verändert und erweitert werden. Theoretisch kann jeder der möchte sich einen eigenen Browser schreiben und es gibt eine Vielzahl solcher Projekte oder auch weniger verbreiteter kommerzieller Browser.
Im folgenden wird der Browser Firefox im Zentrum der Arbeit stehen. Die Hersteller der Browser Chrome, Edge, Firefox und Opera haben sich auf einen gemeinsamen Standard für die Entwicklung von Browser-Extensions geeinigt. Es gibt jedoch noch immer kleine Unterschiede bei den jeweiligen Schnittstellen, was zu Fehlern führen kann, wenn eine Extension direkt von einem Browser auf den anderen übertragen werden soll.

2.1.1. Geschichte der Browser-Extensions (Standardisierung)

Es lässt sich nicht sicher sagen, wann zum ersten mal Nutzer auf die Idee gekommen sind Web-Browser um neue Funktionen oder Designs zu ergänzen. Irgendwann war der Bedarf auf Seiten der Nutzer für standardisierte Schnittstellen zur individuellen Anpassung ihrer Web-Browser gross genug, dass viele Hersteller damit begannen, speziell zu diesem Zweck Entwicklungs-Schnittstellen in ihre Produkte zu integrieren. Diese ersten Schnittstellen waren zunächst jeweils Eigenentwicklungen der Browser-Produzenten und dementsprechend nicht miteinander kompatibel. Für einen bestimmten Web-Browser geschriebene Erweiterungen mussten also komplett umgeschrieben werden, wenn man sie in einem Konurrenz-Browser nutzen wollte.
Im Jahr 2013 wurden erste Schritte hin zu einem gemeinsamen Standard für Web-Extensions unternommen. Damals waren es die Mitarbeiter von Opera, welche das von Google genutzte Format für Browser-Extensions in einer leicht modifizierten Version, in ihr eigenes Produkt integrierten und vorschlugen, das Ergebnis als branchenweiten Standard zu nutzen.

Das Ergebnis der folgenden Bemühungen war die Gründung der Extensions Community Group im Jahre 2016 [w3ecg], durch Opera, Microsoft und Mozilla. Im verlaufe des Projekts schloss sich auch Google selbst dem Vorhaben an, sodass man heute, abgesehen von kleinen Unterschieden,

Web-Extensions für Chrome, Edge, Firefox und Opera mit den gleichen API‘s schreiben und in den meisten Fällen mit nur geringem Aufwand für die anderen Browser nutzbar machen kann. Neben dem Begriff Browser-Extension, existieren auch noch andere Bezeichnungen, z.B. Plug-Ins, Add-ons etc., welche seit der Standardisierung jedoch an Bedeutung verlieren und in vielen Fällen synonym für Browser-Extensions stehen. Vor allem Mozillas Dokumentation kann auf den ersten Blick verwirrend wirken, es wird von Browser- Web-Extensions und Add-ons gesprochen. Davon sollte man sich nicht verwirren lassen, gemeint ist eigentlich immer der gleiche Standard.

2.1.2. Technologie von Browser-Extensions / Stand der Technik

Da sich der durch die Extensions Community Group festgelegte Standard weitest gehend an Google orientiert, benutzen heute alle 4 Browser die gleichen oder sehr ähnliche Formate. Im folgenden wird vorraus gesetzt, dass es sich um Extensions für Mozillas Firefox-Plattform handelt.
Eine Datei Manifest.json muss in jeder Extension enthalten sein. Content- und Background-Scripte sind theoretisch optional. Sie realisieren die Funktionen einer Extension. Es gibt noch weitere Möglichkeiten, die jedoch hier nicht weiter erwähnt werden, da es sich eigentlich um Spezialfälle der obigen Scripte handelt und sie für dieses Projekt uninteressant sind. Diese Bestandteile bilden die eigentliche Browser-Extension die innerhalb der Browser-Extension läuft.
Für das vorliegende Projekt soll eine native Komponente hinzu kommen. Nativ heisst hier, dass sie ein natürlicher Einwohner des Betriebssystems ist, also direkten Zugriff auf dessen Komponenten hat, statt innerhalb der Browser-Umgebung zu laufen. Damit eine Interaktion zwischen diesen beiden Teilen statt finden kann, muss auch für die native Komponente ein Steckbrief, also eine JSON-Datei erstellt werden, welche als Schnittstelle und Bedienungsanleitung für die im Browser laufende Komponente dient.
Im folgenden wird die im Browser-arbeitenden Komponente „Add-on“ und die native Komponente einfach „App“ genannt. Die Komponenten der App werden im späteren Verlauf des Berichts beschrieben.

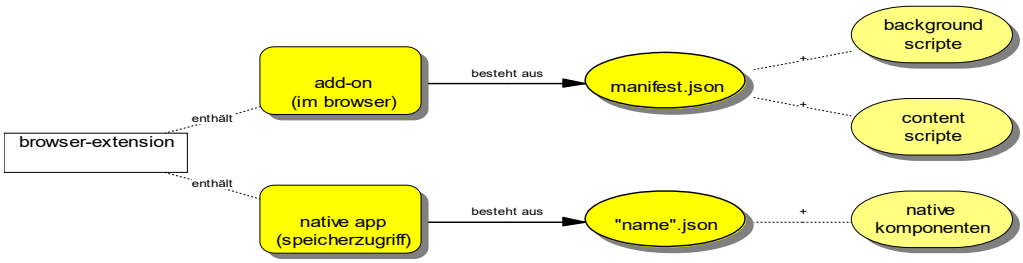

Abbildung 1: Aufbau: Browser-Extension mit nativer Komponente

2.1.3. Das zentrale Element: Manifest.Json

Wie die Endung bereits andeutet handelt es sich um eine Datei, deren Inhalt im JSON-Format kodiert ist. Sie dient dazu die Extension und ihre Fähigkeiten (also vor allem die Browser-Schnittstellen, mit denen sie interagieren soll) gegenüber dem Browser zu legitimieren. Die Einträge im Manifest gleichen dabei einem Steckbrief, der Funktionen, Dateien und Rechte der jeweiligen Extension. Diese Codierung dient dem Browser auch als Bedienungsanleitung für die

funktionalen Komponenten der Extension. Eine einfaches Beispiel für eine Manifest Datei bietet Googles Dokumentation [mExtDoc01]:
https://developer.chrome.com/extensions/overview
Die Beispiel-Datei in Googles Dokumentation enthält Meta-Daten über die Extension: Name, Versionsnummer und Beschreibung (Description). Die Liste „Icons“ enthält schlicht die Dateinamen, der für die Extension global zugelassenen Icons. Das Background-Feld enthält den Dateinamen des Scripts „background_script“ und gibt an dass die Sub-Eigenschaft „persistent“ dieses Scriptes mit „false“ initialisiert wird.
Im Feld „Permissions“ werden der Extensions die Rechte eingeräumt auf die „Activetab“ APIs und auf alle URLs die mit „https//“ beginnen und mit „.google.com“ enden zuzugreifen.
Das Feld „Browser_action“ enthält die Dateien Popup.html und icon_16.png. Dadurch wird das default Symbol für die Schaltfläche in Browser-Oberfläche durch das icon ersetzt und festgelegt, dass bei einem Klick auf das Symbol die Datei Popup.html innerhalb des Browsers geöffnet wird.

Ein weiteres Beispiel aus Googles online Dokumentation findet man unter [gExtDoc02]:
https://developer.chrome.com/extensions/manifest
Diese Variante gibt einen Einblick in die Vielfalt der möglichen Einstellungen und in die Hierarchie der Attribute.
Man beachte Kommentierung der einzelnen Felder mit „Required“, „Recommended“, „Pick one (or none)“ und „Optional“. In seiner minimalistischsten Konfiguration muessen in einer Manifest-Datei lediglich die mit „Required“ markierten Attribute ausgefüllt sein. Für eine detailliertere Übersicht über die Möglichkeiten der Manifest Datei, sind die entsprechenden Dokumentationen von Google und Mozilla zu empfehlen.
Unter folgender URL findet man eine gute Übersicht über die möglichen Eigenschaften der Manifest-Datei, bzw. der Javascript APIs die sie repräsentieren [mExtDoc01]:
https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions/manifest.json

Eines der wichtigsten Attribute: Permissions [mExtDoc02]:
https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions
Allgemein gibt es 3 Typen von Permissions: Host-Permissions, die Activetab-Permission und API-Permissions. Host-Permissions erlauben es vor allem der Extension Zugriff auf URLs oder Gruppen von URLs zu gewähren. Im ersten Beispiel repräsentiert "https://*.google.com/" eine Host-Permission. Die Activetab-Permission gewährt priviligierte Rechte für die im aktiven Tab des Browsers angezeigte Seite. API-Permissions erlauben die Nutzung der jeweiligen gleichnamigen Webextension APIs.

2.1.4. Dokumentationen

Sowohl Google als auch Mozilla pflegen ausführliche online Dokumentationen, einschliesslich Tutorials zum Thema Browser-Extensions.
Einstiegspunkt in die Google-Dokumentation[gExtDoc03] :
https://developer.chrome.com/extensions
Einstiegspunkt in die Mozilla Dokumentation [mExtDoc03]:
https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions

Hinweis: Es gibt ein paar kleine aber entscheidende Unterschiede bei den Attributen der Manifest-Dateien der verschiedenen Hersteller, bzw. der entsprechenden APIs. Manchmal handelt es sich lediglich um minimale Unterschiede, wie andere Namen für ansonsten gleiche APIs, dies reicht

jedoch bereits um im Ergebnis zu verhindern, dass man Extensions nicht ohne weiteres vom einer Browser-Umgebung in einer andere portieren kann.

2.1.5. Die Sprachen

Bei der Entwicklung des urprünglich hauseigenen Frameworks beschlossen die Google-Entwickler sich soweit wie möglich auf gängige Web-Technologien zu konzentrieren. Das Browser-Extension-Framework erlaubt die direkte Einbindung von CSS-, HTML- und Javascript-Dateien. Die steuernden API‘s des Frameworks sind dabei alle in Javascript gehalten, während CSS und HTML vor allem zur Gestaltung der Benutzer-Oberfläche bzw. zur Modifikation von Web-Inhalten dienen.

2.1.6. Die Scripte

Bei den Scripten herrscht Aufgabenteilung, Content-Scripten ist vorbehalten direkt mit aufgerufenen Seiten zu interagieren, während Background-Scripte im Hintergrund laufen und Programmlogik steuern oder eigenen Inhalt, in einem vom Web-Seiten Content unabhängigen Rahmen darstellen. Die beiden Script-Typen können untereinander mit der Messaging-API, direkt kommunizieren.
Theoretisch kann man Extensions schreiben die keine Scripte enthalten, allerdings haben sie dann auch keine richtige Funktion. Erst durch (meist in Javascript geschriebene) Content- und/oder Background-Scripte entfalten sie ihre Magie. Die Präambeln Content und Background geben dabei an, auf welche Daten die jeweiligen Scripte zugreifen können, bzw. mit welchen sie interagieren können.

2.1.7. Content-Script (Javascript, HTML)

Content-Scripte erlauben die direkte Interaktion mit dem im Browser dargestellten Inhalt, also meistens der geöffneten Web-Seiten. Theoretisch können auch CSS Scripte verwendet werden, um z.B. das Aussehen von Inhalten zu ändern, in der Praxis steht der Begriff jedoch Synonym für Javascript-Dateien. Sie erlauben es eigenen Javascript Code in beliebige geladene HTML-Seiten zu injizieren. Man kann also dem Quellcode einer Seite auf diesem Wege eigenen Quellcode hinzufügen. Dabei können Extensions auf die allgemeinen APIs von Javascript zur DOM-Manipulation etc. zugreifen. Content-Scripte haben im Gegensatz zu Background-Scripten nur einen sehr begrenzten Zugriff auf Webextension APIs.

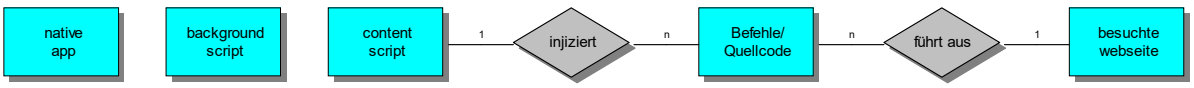

Abbildung 2: Zugriff auf Webseiten durch Content-Scripte

2.1.8. Background-Script (Javascript, HTML)

Background-Scripte werden in dem Augenblick in dem die Extension geladen wird ebenfalls geladen und bleiben es normalerweise auch, bis die Extension ebenfalls beendet wird. In Background Scripten wird der Programmfluss der Extension gesteuert, die meisten der

Webextension-APIs . In diesen Scripten kann nicht direkt auf den Inhalt von Web-Seiten zugegriffen werden. Die Kommunikation mit Content-Scripten oder Nativen Apps erfolgt über die Massaging-APIs.

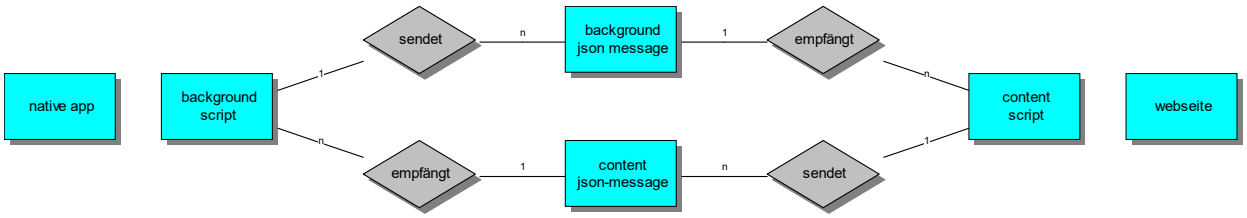

Abbildung 3: Kommunikation zwischen Content- und Background-Scripten

2.1.9. Die native Komponente:

„Nativ“ heisst hier, dass die entsprechende Komponente direkt auf dem Betriebssystem des Nutzers läuft, also ein „natürlicher Bewohner“ desselben ist. Der Vorteil einer solchen Komponente ist, dass man nicht den Restriktionen unterliegt, die die Standardisierung einer Web-Extension mit sich bringt. Z.B. ist es eigentlich nicht möglich mit einer Web-Extension direkt und uneingeschränkt auf den Nutzer-Speicher zuzugreifen. Diese Regelung wurde wie die meisten verwendeten Technologien von den Standards für allgemeine Web-Anwendungen übernommen. Sie soll z.B. verhindern, dass es möglich ist, dass besuchte Webseiten direkt auf den Speicher eines Besuchers zugreifen.
Die Native-Messaging Schnittstelle erlaubt es diese Regelung zu umgehen und direkt auf das Betriebssystem des Nutzers zuzugreifen.

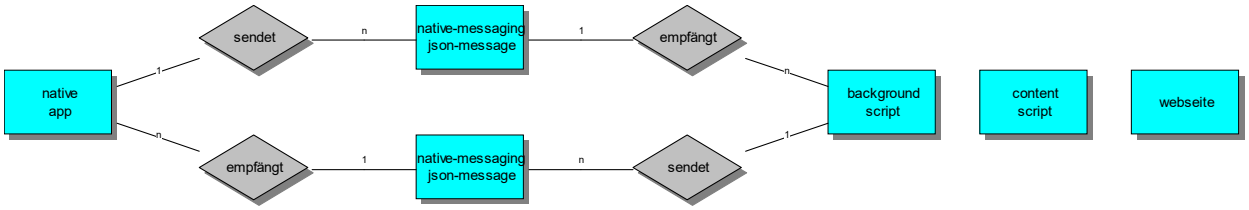

Abbildung 4: Kommunikation zwischen Background-Scripten und nativen Komponenten

2.1.10. Veröffentlichung / Installation:

Alle 4 in der Extensions Community Group vertretenen Hersteller unterhalten einen Store genannte Distributionsplattform für Browser-Extensions. Um eine Expansion in einem dieser Stores anzubieten müssen die Entwickler sie zunächst von den Betreibern des jeweiligen Stores zertifizieren lassen. Dies soll unter anderem verhindern, dass schädliche Software in den Stores vertrieben wird. Gleichzeitig bieten manche Stores auch die Möglichkeit, die angebotene Software zu monetarisieren.
Prinzipiell müssen Extensions die im Chrome oder Firefox Store veröffentlicht werden sollen immer erst gepackt werden. Die Archive können dann auf den dafür vorgesehenen Seiten der Hersteller hochgeladen und geprüft/signiert werden. Nach erfolgreicher Prüfung und Signierung können die Archive über den entsprechenden Store öffentlich verfügbar gemacht werden. Die

Installation ist dann mit ein paar Clicks kinderleicht. In der entpackten Form können Add-ons in beiden Browsern nur im Entwickler-Modus installiert werden.
Komplizierter wird es unter Umständen, wenn die Extensions im Konzert mit sogenannten Native-Apps, also Programmen auf dem Speicher des Nutzer-Computers welche über die Native-App-API mit der Extension arbeiten sollen.
Im vorliegenden Projekt wurde die entsprechende Native-App-Komponente als Python-Script verwirklicht. Also muss z.B. eine kompatible Python-Version auf dem Nutzer-System vorhanden sein und korrekt angesprochen werden.

2.1.11. Beispiele für verbreitete Browser Extensions

Eine Übersicht über populäre Extensions kann man z.B. dem Chrome Web-Store entnehmen [gExtStr01].
(siehe: https://chrome.google.com/webstore/category/extensions)

Der Autor nutzt unter anderem folgende Browser-Extensions:

Disconnect [gExtStr02]:
(siehe :https://chrome.google.com/webstore/detail/disconnect/jeoacafpbcihiomhlakheieifhpjdfeo)
Dieses kleine Programm dient dazu vor allem dazu die Überwachung durch besuchte Webseiten, dass sogenannte Tracking zu verhindern, was den angenehmen Nebeneffekt hat, dass durch die Blockade der entsprechenden Prozesse, das Laden der eigentlichen Inhalte beschleunigt wird.

Add-Block [gExtStr03]:
(siehe: https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom?hl=de)
Ein Programm zur Unterdrückung unerwünschter Werbung auf Internet Seiten.

2.2. Web Scraping

2.2.1. Allgemein

Während der Arbeit an dem Projekt, entdeckte der Autor, dass die Extraktion von Daten aus Web-Seiten über die Benutzer-Oberfläche unter dem Fachbegriff „Scraping“ ein weit verbreitetes Thema ist.
Dabei werden die gesuchten Daten metaphorisch von der Benutzer-Oberfläche der Web-Seiten abgekratzt (engl „to scrape“ = de „kratzen“).
Dabei geht es jedoch meistens nicht darum, die Interaktion des Nutzers mit verschiedenen Web-Seiten zu untersuchen. Statt dessen wird oft versucht, aus der automatisierten Abfrage der öffentlichen Präsenzen von Unternehmen, Informationen zu gewinnen, die diese so eigentlich nicht veröffentlichen möchten. Z.B. kann man eine automatisierte Abfrage installieren um die Preisentwicklung aller Angebote eines Online-Haendlers zu dokumentieren um besonders günstig einzukaufen, oder die Preisentwicklung des eigenen Shops der Konkurrenz anzupassen. Die Bandbreite der Anwendungsmöglichkeiten ist gross, das abstrakte Prinzip lässt sich auch nur schwer eingrenzen.

2.2.2. Rechtliche Bedenken / Kontroverse

Wie bereits im vorigen Absatz angedeutet, ist das Abschöpfen von Daten auch aus öffentlich zugänglichen Quellen nicht immer erwünscht und in bestimmten Fällen auch rechtlich bedenklich. Die veröffentlichten Inhalte in Summe erlauben es Schlüsse zu ziehen, Muster zu finden, die einem menschlichen Surfer verborgen geblieben und deren Veröffentlichung eben nicht beabsichtigt war.
Die dem Autor zugänglichen Quellen deuten darauf hin, dass die Rechtssprechung in der BRD im Moment davon ausgeht, dass Scraping prinzipiell legal ist.
„Komplette Datenbanken und wesentliche Teile solcher Datenbanken sind urheberrechtlich geschützt, nicht aber einzelne Inhalte.“
Das vorliegende Projekt analysiert nur kleine Teile einer besuchten Web-Seite, im wesentlichen die DOM-Elemente unter dem Mauszeiger. Die Einsatz wäre demnach nur dann nicht erlaubt, wenn der Nutzer die Extension nutzen würde um Stück für Stück eine komplette urheberrechtlich geschützte Datenbank zusammen zu setzen. Hinweis: theoretisch wäre auch denkbar, dass eine urheberrechtlich geschützte Datenbank als ein einziges DOM-Element repräsentiert wird. Um die Frage der Legalität abschliessend zu klären müsste der Autor den Rat eines Juristen einholen [lScrap01].
(siehe: http://www.rechtswissenschaft-verstehen.de/lexikon/screen-scraping/)

2.2.3. Beispiele für Scraping-Anwendungen

Das prominenteste Beispiel für Scraping-Anwendungen stellt wohl das Open-Source-Framework Selenium da und verdeutlicht gleichzeitig die Dialektik der Anwendungsmöglichkeiten von Scraping. Der offiziell zugedachte Anwendungszweck des Framework ist das Testen von Web-Applikationen, also Programmen nach dem Server-Client-Modell deren Benutzer-Oberflächen über Browser erreichbar sind. Der Übergang von Test und Missbrauch ist fliessend und hängt vor allem von der Beziehung von Selenium-Anwender und Inhaber der „getesteten“ Web-Seiten ab [hqSel01].
(siehe https://www.seleniumhq.org/)

2.3. Der Gegenstand der Analyse

Im folgenden soll ein kurzer Überblick über die Codierung von Information in Webseiten gegeben werden. Dabei soll aufgezeigt werden, welcher Zusammenhang zwischen der erzeugten grafischen Oberfläche und der zugrunde liegenden Struktur von Webseiten besteht.

2.3.1. Informationen in Webseiten-Oberflächen

Das Internet kann man sich auch metaphorisch als eine gewaltige Stadt vorstellen. Die Häuser aus denen diese Stadt besteht entsprechen den verschiedenen Websites. Die Häuser selbst haben x Stockwerke, welche einzelnen Webseiten also Dokumenten entsprechen.
Die zugrunde liegende Technologie wird an dieser Stelle als gegeben voraus gesetzt. Gegenstand der Funktionen dieses Projekts sollen Benutzeroberflächen besuchter Webseiten(bildlich also der Stockwerke der Häuser) sein.
Für Menschen wahrnehmbare Signale können derzeit visuell und akustisch durch Computer übermittelt werden. Akustische Signale sollen an dieser Stelle nicht weiter behandelt werden, da der Umfang des Projekts dies nicht zulässt und ihre Bedeutung im Vergleich zu visuellen Signalen geringer ist.

Die Visuellen Signale von Webseiten werden über das Medium eines Computer-Bildschirms, bzw. des abgestrahlten Lichts an den menschlichen Wahrnehmungsapparat übertragen. Sie entsprechen also einer Kanal-Codierung für die zugrunde liegende Quellcodierung einer Webseite, dem Quelltext. Sowohl Quelltext also auch die Ausgabe am Bildschirm sind also letztlich lediglich Codierungen der gleichen Informationsmenge.
Für die Analyse von Webseiten bedeutet dies, dass man die entsprechend codierten Informationen vergleichen kann. Während die visuellen, direkt für die menschliche Wahrnehmung codierten Informationen keine digitale Schnittstelle besitzen, ist der Quellcode über das Document-Object-Modell hierarchisch organisiert und abrufbar. Gerade diese doppelte Codierung erlaubt eine vergleichende Analyse zwischen den beiden Varianten.
Schliesslich kann bei einer Analyse noch die „Umgebung“ betrachtet werden, was man heute Meta-Daten nennt. Also die URL, die reine Menge an Daten einer Seite, Schriftgrösse, das Datum des Abrufs, die genutzten Technologien, Scripte, etc.. Zu den Meta-Daten kann auch die Aufzeichnung der Befehlseingabe durch den Nutzer gezählt werden, der mit den doppelt codierten Informationen interagiert.
Welche dieser Daten für eine Analyse letztlich heran gezogen werden, hängt letztlich immer von der jeweiligen Fragestellung ab.

2.3.2. DOM-Bäume

Die Abkürzung DOM steht für Document Object Modell und ist ein Standard des World Wide Web Consortiums. Dieses „Modell“ ist eigentlich eine Plattform- und Sprach-unabhängige Schnittstelle für den Zugriff auf Inhalt, Struktur und Stil eines Dokuments. Diese Schnittstelle setzt das Objektmodell der objektorientierten Programmierung für die entsprechenden Dokumente um. Es wird unterschieden zwischen CORE-DOM, XML-DOM und HTML-DOM. Im folgenden verallgemeinernd, davon ausgegangen, dass ausschliesslich HTML-Seiten betrachtet werden, die meisten Erläuterungen sind für DOM-Objekte universell gültig.
Jedes Tag oder HTML-Element kann über seine DOM-Schnittstelle angesprochen werden. So kann z.B. mit CSS der Stil einer Menge von HTML-Elementen geändert werden, indem sie über ihre DOM-Schnittstellen angesprochen werden. Mit Javascript kann der Inhalt einer Webseite dynamisch angesprochen werden, es können also Objekte über ihre DOM-Schnittstelle abgefragt, hinzugefügt, verändert oder entfernt werden.
Hierarchisch sind diese Objekte eben eine objektorientierte Repräsentation der Baum-Modells von HTML-Seiten. Bildlich gesprochen sind Webseiten nicht als Seiten in einem Buch zu verstehen, sondern vielmehr als eine Collage von sich überschneidenden Zetteln auf einer Seite (oder als ein Haufen von Post-its). Die Wurzel des Baums entspricht der untersten Seite des Stapels. Der Mauszeiger zeigt immer auf das oberste Blatt des Baums/Stapels an der jeweiligen Stelle.
Letztere Funktion ist die Grundlage für die vorliegende Projektarbeit.

3. Technische Analyse

Zentraler Bestandteil der zu erstellenden Extension-Software muss entsprechend des Standards für Browser-Extensions die Manifest-Dartei sein.
Weiterhin sollen Daten aus dem Quelltext besuchter Webseiten extrahiert werden, diese direkte Interaktion mit besuchten Web-Seiten wird mit Hilfe eines Context-Scripts realisiert.
Die eigentliche Bearbeitung der durch das Context-Script gesammelten Daten wird durch ein Background-Script ausgeführt. Die Daten einer Internet-Seite bilden eine Baumstruktur, wobei jeder Knoten ein Objekt repräsentiert. Jedes dieser Objekte lässt sich über die DOM-Schnittstelle ansteuern. Ein Context-Script kann jedoch nicht direkt mit Applikationen ausserhalb der Browser-Extension kommunizieren. Diese Rolle ist den Background-Scripten vorbehalten. Daten-Pakete aus einer Web-Seite müssen vom Context-Script erfasst, von dort an das Background-Script geschickt werden, von wo aus sie wiederum an eine externe Instanz weiter geleitet werden können. Die Richtung des Kommunikationssflusses kann dabei auch umgekehrt, oder an einer Stelle unterbrochen werden.
Man kann dabei nach dem aktuellen Standard für Browser-Extensions nicht direkt auf den lokalen Speicher eines Nutzer-Computers zugreifen. Um Daten auf einem lokalen Speicher abzulegen, gibt es 2 Optionen. Die erste und einfachere ist die Verwendung von XMLhttp-Requests zur Kommunikation zwischen Background-Script und einem beliebigen Server. Das Modell hat den Nachteil, dass die Partner-App nicht aus der Browser-Oberfläche gestartet werden kann. Es ist jedoch möglich dieses Problem zu umgehen, indem man zusätzlich zu der eigentlichen Extension eine Native App schreibt, welche die Interaktion mit dem lokalen Speicher übernimmt. Die Native App empfängt/schickt Daten von/zu der Extension und interagiert direkt mit dem lokalen Speicher. Diese native App, kann über die Native-Messaging-Schnittstelle auch direkt durch die im Browser arbeitende Komponente gestartet werden.
Die Namensgebung an dieser Stelle kann etwas verwirrend sein. Während die API für die Kommunikation zwischen den Scripten „Messaging“ genannt wird, wird die API zur Kommunikation des Background-Scripts mit nativen Anwendungen als „native-Messaging“ bezeichnet.
Um die entsprechenden Schnittstellen nutzen zu können, müssen ihre Nutzung im Manifest erlaubt werden, es müssen die entsprechenden Permissions eingetragen werden. Nach der Installation der Extension muss der Nutzer einmalig bestätigen, dass er der Extension den Zugriff auf diese Schnittstellen erlaubt.
Das Resultat ist ein Aufbau ähnlich der Geschäftsprozessmodellierung, die einzelnen Scripte und die Native-App entsprechen Geschäftsprozessen, die untereinander über die beiden Messaging-APIs kommunizieren.
Zu beachten ist hierbei, dass laut Dokumentation, beide Formen des Messaging (zwischen den Scripten und zwischen Extension und Native-App) sehr ähnlich sind und das bereits durch das Manifest bekannte JSON-Format für die übermittelten Nachrichten-Pakete nutzen, jedoch nur das Background-Script direkt mit der Native-App kommunizieren kann.
Die Native-App selbst kann prinzipiell in jeder Sprache geschrieben werden, welche die in der Native-Messaging-API implementierte STDIO-Schnittstelle nutzen kann. Ihr Aufgabenbereich umfasst neben der Kommunikation mit der Extension, das Speichern und Lesen von/auf dem lokalen Speichermedium. Entsprechende Sprachen sind z.B. Python, C, C++ usw..

4. Planung (Ressourcen und Meilensteine)

Für die Umsetzung des Projekts, standen folgenden Ressourcen zur Verfügung:

4.1. Ressourcen

4.1.1. Hardware

Zur Bearbeitung der Aufgabe, stand dem Autor ein handelsüblicher Laptop mit Internet-Verbindung zur Verfügung.

4.1.2. Human Ressources

Die Aufgabe ist als Einzelleistung des Autors entstanden. Bei einzelnen Fragen zur Aufgabenstellung war Herr Wiemer als Betreuer so freundlich mit Rat und Tat zu helfen.
Im Laufe der Bearbeitung, trat an einer Stelle ein Probelm auf, bei dem der Autor versuchte sich über das Stack-Overflow Forum Hilfe zu suchen. Der Entsprechende Versuch blieb jedoch unbeantwortet, der Autor war in der Lage das Problem selbststaendig zu lösen.

4.1.3. Zeit

Der allgemeine Rahmen war natürlich durch die Aufgabenstellung mit einer Dauer von 5 Monaten vorgegeben. Da die Programmierung mit und für Browser-Plattformen für den Autor komplett neu war, kam der thematischen Einarbeitung / Recherche in die Thematik ein entsprechend grosser Anteil zu. Nach der allgemeinen Einarbeitung in das Thema war es vorgesehen sich der Umsetzung des Software-Projekts zu widmen, indem die Aufgabe in weitestgehend unabhängig voneinander erfüllbare Teilschritte unterteil wurde (siehe Meilensteine).
Aufbauend auf den bisherigen Ergebnissen, stand die Erstellung des Projektberichts am Ende der Bearbeitung.

4.1.4. Software

Das gesamte Projekt wurde auf einem Rechner mit dem Betriebssystem Windows 10 bearbeitet. Weiterhin wurden im Verlaufe des Projekts die Web-Browser Chrome und Firefox in ihren aktuellen Iterationen verwendet. Das Projekt wurde vom Autor zu Beginn für Googles Chrome Browser entwickelt. Die Implementierung der Native-Messaging-API führte jedoch auch nach einer Woche Entwicklungszeit zu keinem funktionierenden Ergebnis, im Gegensatz zu entsprechenden Versuchen mit Firefox. Die Programmierung selbst wurde mit Hilfe des Atom-Editors von Github realisiert. Dazu wurden die Programmiersprachen HTML, Javascript und Python verwendet. Während die ersten beiden als teil des Standards für Browser Extensions direkt von den Web-Browsern interpretiert werden, musste für Python die entsprechende Funktions-Umgebung installiert werden. Für die Erstellung des Berichts wurde die Open-Source-Software Libre-Office verwendet.

4.1.5. Recherche

Sowohl Mozilla als auch Google pflegen kostenlose Online-Dokumentationen über die Entwicklung von Browser-Extensions auf ihren jeweiligen Plattformen. Bei der Google-Variante gibt es allerdings Unklarheiten, manche der gepflegten Dokumentationen beziehen sich auf Legacy Code, was nicht immer ausreichend gekennzeichnet ist. Google bietet mit Chromium ein Betriebssystem an. Die Tutorials von Google lassen darauf schliessen, dass der Aufbau der Anwendungen und die Bezeichnungen von Chromium Entwicklungs-APIs in vielen Punkten mit den Entsprechungen für Web-Extensions für den Chrome Browser übereinstimmen. Jedoch nicht in allen, die Unterschiede in der Dokumentation sind nicht immer auf den ersten Blick erkennbar.
Die entsprechende Dokumentation von Mozilla war in dieser Hinsicht leichter zu entschlüsseln.
Als Basis für viele Detail-Recherchen diente schliesslich Google‘s gleichnamige Suchmaschine, oder die Entwickler-Plattform Stack-Overflow, bzw. die Code/Projekt-Sharing-Seite Github.

4.2. Meilensteine

Vor der eigentlichen Entwicklung hat der Autor einen groben Projektplan mit entsprechenden Meilensteinen erstellt. Eine detailliertere zeitliche Unterteilung war kaum möglich, da zu viele Teilbereiche für den Autor komplett unbekannt waren, eine Abschätzung der notwendigen Zeit also unrealistisch.

Teilaufgabe 1.1: Recherche: Add-Ons allgemein, Aufbau, Technologien
Teilaufgabe 1.2: Entwicklung: Add-On allgemein, Funktionstest
Teilaufgabe 2.1: Recherche: Maus-Dom-Interaktion
Teilaufgabe 2.2: Entwicklung: Add-On Maus-Dom-Interaktion Funktionstest
Teilaufgabe 3.1: Recherche: Daten auf Nutzer-System lokal speichern.
Teilaufgabe 3.2: Entwicklung: Daten auf Nutzer-System lokal speichern
Teilaufgabe 4.1: Recherche: Assembly: Verknüpfung der Ergebnisse aus 2 und 3
Teilaufgabe 4.2: Entwicklung: Assembly: Verknüpfung der Ergebnisse aus 2 und 3
Teilaufgabe 5.1: Recherche: NominalPhrasenAnalyse
Teilaufgabe 5.2: Entwicklung: Nominalphrasen-Analyse
Teilaufgabe 5.3: Entwicklung: Assembly: Integration von 5.3 in das Projekt
Teilaufgabe 6.1: Recherche: GUI für eine Wiedergabe der gewonnenen Daten
Teilaufgabe 6.2: Entwicklung: GUI für eine Wiedergabe der gewonnenen Daten (allein stehend)
Teilaufgabe 6.3: Entwicklung: Integration der vorigen Ergebnisse in die GUI
Teilaufgabe: Projektbericht schreiben (1. Struktur, 2. Inhalt, 3. Grafiken)

5. Realisierung (Beschreibung des Vorgehens, der gefällten Entscheidungen etc.)

Entsprechend der in Kapitel 4 dargestellten Meilensteine, hat der Autor die Projektaufgabe in mehrere kleinere Schritte unterteilt und diese nacheinander abgearbeitet. Während der Bearbeitung musste die Planung leicht angepasst werden.

5.1. Teilaufgabe: Basis der Browser-Extension

Der Schluessel zu jeder Browser-Extension ist die Manifest.json Datei. Unabhängig von Funktion und Inhalt der Extension muss das Manifest die Attribute „manifest_version“, „name“ und „version“ enthalten („description“ ist optional). Um die Ergebnisse der Analyse umzusetzen werden weiterhin ein Content-Script und ein Background-Script benötigt (content.js und background.js). Entsprechend muss das Manifest um die Attribute „content_scripts“ und „background“ erweitert werden. „content_script“ bekommt das Sub-Attribut in dem der Dateiname „content.js“ als Wert eingetragen wird. Schliesslich wird dem Attribut „background“ das Sub-Attribut „scripts“ mit dem Wert des Dateinamen „background.js“ angehängt. Die Scripte sind zu diesem Zeitpunkt leer, enthalten keinen Code. Alle 3 Dateien werde in einem Ordner mit dem Namen „C:\praktWebExt“ gespeichert.

Content.js : leer
Background.js: leer

Manifest.json:

```
{
 "description": "Saving Stuff example",
 "manifest_version": 2,
 "name": "PraktikumExtension",
 "version": "1.0",

 "content_scripts":
   {
    "js": ["content.js"]
   },

 "background": {
  "scripts": ["background.js"]
 },
 }
```

Hinweis: Browser-Konsole:
Für das Testen der einzelnen Funktionsabschnitte wird im folgenden die Browser-Console von Firefox benötigt. Diese lässt sich über das Menü öffnen: „Menü öffnen“ → „Web-Entwickler“ → „Browser-Konsole“, oder über den Shortcut: Strg+Umschalt+J. Der Javascript Befehl „console.log(...“ soll in der Browser-Konsole zu Ausgaben führen.

5.2. Teilaufgabe: Auslösen eines mousemove-Events

Um die Mausbewegungen des Nutzers auf der besuchten Web-Seite zu erfassen, muss mit Hilfe des Content-Scripts der Quellcode der besuchten Seiten um einen Event-Listener für das „mousemove“-Event erweitert werden. Dabei handelt es sich nicht um eine Web-Extension spezifische Schnittstelle, „mousemove“ ist teil des W3C-Standards [w3MoMve].
https://www.w3.org/TR/uievents/#event-type-mousemove

Dem „window“-Objekt wird ein EventListener für ein „mousemove“-Event hinzugefügt, welches die Funktion „notifyExtension“ auslöst. Dazu dient folgende Zeile im Content-Script:
window.addEventListener("mousemove", notifyExtension);

Beim Auftreten eines mousemove-Events wird die „notifyExtension“ Funktion ausgelöst, welche in der Browser-Konsole (siehe Hinweis Browser-Konsole) den String „Content-Script Mouse-Event!“ ausgibt.

Content.js:
```
window.addEventListener("mousemove", notifyExtension);
function notifyExtension(e) {
   console.log("Content-Script Mouse-Event!“);
}
```

5.3. Teilaufgabe: Zugriff auf das auslösende DOM-Element

Das „mousemove“-Event wird ausgelöst, wenn das unter dem Zeiger liegende DOM-Element durch eine Bewegung der Maus wechselt. Im vorliegenden Fall repräsentiert der Parameter „e“ das auslösende Event. Mit „e.target“(event.target) kann das entsprechende auslösende DOM-Element der besuchten Seite adressiert werden und mit „e.target.innerHTML“ der HTML-Inhalt des DOM-Elements.
Content.js:
```
window.addEventListener("mousemove", notifyExtension);
function notifyExtension(e) {
   console.log(„content-script“ + e.target.innerHTML);
}
```

5.4. Teilaufgabe: Vermeiden von unnötigen Wiederholungen

Bisher unterscheidet die Funktion nicht, ob das auslösende Element sich ändert. Um Wiederholungen zu verhindern, wird eine globale Variable „var knownElement=0“ am Anfang des Scriptes initialisiert und die Funktion „notifyExtension“ um ein paar Zeilen ergänzt. Falls e.target den gleichen Inhalt hat, wie „knownElement“, wird die Funktion zurück gegeben. Andernfalls, wird „knownElement“ der Wert von „e.target“ zugewiesen und der Rest der Funktion ausgeführt, sodass die Funktion nur dann eine Ausgabe auf der Konsole verursacht, wenn das DOM-Element wechselt.

Content.js:
```
var knownElement=0;

window.addEventListener("mousemove", notifyExtension);
function notifyExtension(e) {
  if (e.target == knownElement) {
    return;
  }
  else {
    knownElement= e.target;
    console.log(„content-script“ + e.target.innerHTML);
}}
```

5.5. Teilaufgabe: Messaging (Content-Script → Background-Script)

Die Daten aus dem Content-Script sollen nun an das Background-Script weiter gereicht werden. Wie bereits beschrieben dient die einfache Messaging-API der Kommunikation zwischen den Scripten einer Browser-Extension. Das Content-Script muss dazu die Funktion browser.runtime.SendMessage() implementieren, welche bei jedem Aufruf eine Nachricht versendet. Wenn die Funktion wie hier ohne Übergabe einer ExtensionId als Parameter aufgerufen wird, sendet sie die Message an alle anderen Scripte der Extension, ausgenommen des Absender-Scipts(content.js). Da das vorliegende Projekt nur 2 Scripte enthalten soll, erfüllt der Parameter keine Funktion und wird nicht benötigt [mSendMe].
https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/sendMessage

Content.js:

```
var knownElement=0;

window.addEventListener("mousemove", notifyExtension);
function notifyExtension(e) {
  if (e.target == knownElement) {
    return;
  }
  else {
    knownElement= e.target;
    console.log(„content-scirpt“ + e.target.innerHTML);

    browser.runtime.sendMessage(
    {"innerHTML": e.target.innerHTML});
}}
```

Wenn das Content-Script die Nachrichten verschickt, soll das Background-Script sie empfangen. Dazu muss wieder ein entsprechender Event-Listener und eine verarbeitende Funktion implementiert werden. Ergo bekommt das vormals leere Background-Script an dieser Stelle seine ersten Zeilen Code.

Background.js:

```
browser.runtime.onMessage.addListener(notify);
function notify(message) {
  var jsonString= JSON.stringify(message)
  console.log(„background-script: „ + jsonString);
}
```

Hinweis: Background→ Content:
Um Nachrichten umgekehrt vom Background- an das Content-Script zu schicken, muss das Vorgehen leicht geändert werden. Statt der browser.runtime.sendMessage Funktion muss dann die Funktion tabs.sendMessage verwendet werden.

5.6.1. Teilaufgabe: Native-Messaging: Vorbereitungen

Die Implementation der Native-Messaging API ist notwendig um die Browser-Extension mit einem Programm auf dem lokalen Speicher des Nutzers interagieren zu lassen. Im folgenden hat sich der Autor weitest gehend an dem Beispiel orientiert, welches auch in der Dokumentation zu dieser API im entsprechenden Mozilla-Tutorial verwendet wird [mNativeMe]:
https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging

5.6.2. Teilaufgabe: Python (Version 3.7.0)

Um das Programm in der vorliegenden Form nutzen zu können ist es notwendig einen Python Interpreter und die Standard-Bibliotheken zu installieren. Für dieses Projekt wurde die Version 3.7.0 verwendet, frühere Versionen wurden nicht getestet.
Eine Installations-Anleitung für Python auf dem Windows10 Betriebssystem findet man unter[oPythDoc]:
https://docs.python.org/2/using/windows.html

5.6.3. Teilaufgabe: Dateien und Ordner der nativen App

Im selben Ordner „C:\praktWebExt“ werden 3 neue Darteien angelegt: app.json, app.py und app.bat. Die Datei app.json erfüllt eine sehr ähnliche Funktion wie die manifest.json Datei der Extension selber. Im Detail gibt es jedoch Unterschiede, z.B. ist der Name frei wählbar.

app.json:

```
{
  "name": "file_test",
  "description": "Example host for native messaging",
  "path": "C:\\mysstuff\\app\\ping_pong_win.bat",
  "type": "stdio",
  "allowed_extensions": [ "file_test@example.org" ]
}
```

- name: Der hier eingetragene Wert (file_test) muss gleich dem Wert sein, welcher der Funktion browser.runtime.connectNative("file_test") im Background-Script übergeben wird. Ausserdem muss der erstellte Registry-Key für die Native App mit diesem Wert übereinstimmen.
- path: Der Dateipfad der beim Aufruf auszuführenden Datei. Man beachte die doppelten Backslashes.
- type: Für diesen Schluessel gibt es im Moment nur einen möglichen Wert, nämlich stdio. Die App kann über die Schnittstellen STDIO(bzw. stdin) Daten empfangen und über stdout Daten verschicken.
- allowed_extensions: Enthält die Ids aller Extensions welche berefchtigt sind mit der nativen App zu kommunizieren.

app.bat:
Eine Windows-Batch Datei, welche als Schnittstelle zwischen der geforderten Syntax des app Manifests app.json und der für die Ausführung von Python-Programmen. Dieser Zwischenschritt ist scheinbar nicht immer notwendig, aber dem Autor ist es nicht gelungen ihn zu umgehen.

```
@echo off
call python C:\praktWebExt\app.py
```
Offensichtlich ist es essenziell, dass die Dateien der Python App (app.bat, app.json und app.py) im richtigen Verzeichnis liegen.

Hinweis:
An dieser Stelle hatte der Autor lange zu kämpfen. Scheinbar ist es unter bestimmten Bedingungen notwendig die sogenannte Kurz-Schreibweise des Pfadnamen zu verwenden. Im Falle des Projekts konnte Python keine Pfadnamen mit Leerzeichen verarbeiten. Der Pfad C:\praktWebExt\app.py in der hier vorgestellten Version sollte keine Probleme machen, der Hinweis bezieht sich auf eventuelle Änderungen.
Eine Anleitung wie man den entsprechenden „short path name“ heraus bekommt, findet man z.B. hier [shortPath]: https://superuser.com/questions/348079/how-can-i-find-the-short-path-of-a-windows-directory-file

5.7. Teilaufgabe: Implementierung der Native-Messaging API

Die am Datenaustausch direkt beteiligten Scripte sind in diesem Fall „Background.js“ und „App.py“. Die Native-Messaging API nutzt die STDIO Schnittstelle um JSON-Datenpakete zu transportieren.

Background.js:
Auf der Extension Seite der Kommunikation, kann die API „connectNative“ genutzt werden. Diese dient zur Initialisierung eines Ports welcher als Schnittstelle für das Senden und Empfangen von JSON-Paketen an und von App.py dient.

```
// Initialisierung des Ports:
var port = browser.runtime.connectNative("app");

// Verknüpfung von Message-Event-Listener und Port
port.onMessage.addListener((response) => {
  console.log("Backgroud Received: " + response);
});
```

App.py:
Im Python Code der nativen App, wird die Stdio Schnittstelle über sys.stdin angesprochen. Der entsprechende Code ist in der Methode „getMessage“ enthalten, welche auch die Decodierung aus dem JSON-Format übernimmt. Die Methode „encodeMessage“ dient zur Codierung eines Datums für den umgekehrten Weg, von App.py zu Background.js. Entsprechend codierte Daten können dann mit der Funktion „sendMessage“ über die Stdio-Schnittstelle an das Background-Script übergeben werden.
Die getMessage Methode wird in einer Endlosschleife ausgeführt um zu gewährleisten, dass eingehende Nachrichten auch erkannt werden. Ausserdem wird jede eingehende Nachricht mit einem String („pong3“) quittiert.

```
#!/usr/bin/env python

import sys
import json
import struct
```

```
try:
    # Reaktion auf den Eingang einer JSON-Message über die Stdin-Schnittstelle
    def getMessage():
        rawLength = sys.stdin.buffer.read(4)
        if len(rawLength) == 0:
            sys.exit(0)
        messageLength = struct.unpack('@I', rawLength)[0]
        message = sys.stdin.buffer.read(messageLength).decode()#'utf-8')
        jsonMessage= json.loads(message)
        return jsonMessage

    # Die Antwort der Python App an das Background-Script muss wieder in
    # JSON codiert werden.
    def encodeMessage(messageContent):
        encodedContent = json.dumps(messageContent).encode('utf-8')
        encodedLength = struct.pack('@I', len(encodedContent))
        return {'length': encodedLength, 'content': encodedContent}

    # Die entsprechende NAchricht wird über die Stdout-Schnittstelle verschickt.
    def sendMessage(encodedMessage):
        sys.stdout.buffer.write(encodedMessage['length'])
        sys.stdout.buffer.write(encodedMessage['content'])
        sys.stdout.buffer.flush()

    # Falls eine Nachricht eingegangen ist, soll eine Quittung an das
    # Background-Script geschickt werden.
    while True:
        receivedMessage = getMessage()
        if receivedMessage:
            sendMessage(encodeMessage("pong3"))
except AttributeError:
    # wenn Error, dann kaputt
    sys.exit(0)
```

5.8. Teilaufgabe: Eingehende Nachrichten lokal speichern

Um die eingehenden Daten zu speichern, soll bei jeder Sitzung ein neuer Ordner angelegt werden. In diesem Ordner soll für jede Sitzung eine neue Datei im Format CSV angelegt werden, in welcher die eingehenden Nachrichten abgespeichert werden.

Um sicher zu stellen, dass jede Datei einzigartig und systematisch benannt ist, wird der entsprechende Datei- und Pfad-Name mit Hilfe der System-Zeit Schnittstelle generiert. Die CSV-Datei selbst wird mit einem Header versehen um die erwarteten Datenpakete aufzunehmen.

App.py:

```
import pathlib
import time
import csv
```

```
# Erstelle einen String mit der codierten Systemzeit.
ts = time.strftime("%Ss%Mm%Hh%p%d%b%y", time.localtime())
# derzeitiger Arbeitsordner
currentpath=os.getcwd()
# Systemzeit-String wird neuer Arbeitsordner.
pathlib.Path(currentpath+'/'+ts).mkdir(parents=True, exist_ok=True)
# Erstelle CSV.Datei mit entsprechendem Header, innerhalb des Ordners zum Speichern der
#Nachrichten.
with open(ts+'/'+ts+".csv",'w+', newline='') as csvfile:
    fieldnames = ['messageId', 'url', 'innerHTML', 'outerHTML', 'tagName', 'currentURL']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
```

Zum speichern der eingehenden Nachrichten wird die Methode getMessage entsprechend angepasst.

App.py:

```
    # Die Werte der JSON-Message sollen in die zu Beginn
    # generierte CSV-Datei geschrieben werden.
    with open(ts+'/'+ts+'.csv', 'a', newline='') as csvfile:
        fieldnames = ['messageId', 'url', 'innerHTML', 'outerHTML', 'tagName', 'currentURL']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writerow(jsonMessage)
```

5.9. Teilaufgabe: Screenshots

Für eine spätere Verwendung der Software hielt der Autor es für sinnvoll, während einer Sitzung neben dem HTML-Code Bild-Daten zu erfassen. Z.B. können Algorithmen des maschinellen Lernens diese Bilddaten mit den DOM-Informationen in Verbindung bringen und auf Muster untersuchen. Dazu müssen lediglich die entsprechenden Bibliotheken eingebunden und die getMessage Methode um 2 Zeilen ergänzt werden.

App.py:

```
from PIL import ImageGrab

    im=ImageGrab.grab(bbox=(10,10,500,500))
    im.save(ts+'/'+time.strftime("%Ss%Mm%Hh%p%d%b%y", time.localtime())+'.png')
```

Hinweis:
Die Position und Grösse der Screenshots sind in diesem Stadium zufällig gewählt und müssen bei einer Weiter-Entwicklung den Erfordernissen angepasst werden.

6. Ausblick (Anwendungsmöglichkeiten)

Die Extension war von Anfang an vom Autor als Plattform-Konzept für weitere Anwendungen gedacht. Die Bedeutung des Internet nimmt seit Jahrzehnten praktisch ungebremst zu. Über alle Betriebssysteme hinweg dienen Web-Browser als Rahmen für die Darstellung der Web-Inhalte, bzw. der Benutzeroberflächen. Technologische Fortschritte erlauben es immer kompliziertere Anwendungen in Browsern dar zu stellen, mit Googles Chrome-Os gibt es heute sogar ein Betriebssystem, welches sich an Web-Browsern orientiert.
Die Browser-Extension Schnittstelle, welche durch die Web-Extension-Community-Group standardisiert wurde erlaubt es nun, mit einfachsten Mitteln, Verbindungen zwischen beliebigen Programmen, Sprachen und Systemen und des global akzeptierten WWW-Standards herzustellen.

Für den Autor sind vor allem Anwendungen interessant, die maschinelles Lernen auf Web-Inhalte anwenden. Die Überlegung lautet: die GUIs von Webseiten bestehen aus 2 Informations-Mengen. Dem Seiten-Quelltext und der visuellen Repräsentation. Dabei besteht stets eine direkte Verbindung zwischen visueller Repräsentation und codierter Funktionalität. Also dem semantischen Inhalt und der Codierung. Dieser Zusammenhang macht das WWW zu einer idealen Umgebung um Assistenz-Systeme mit künstlicher Intelligenz zu trainieren. Der Autor hat vor in dieser Richtung weiter zu forschen.

6.1. Anwendungsmöglichkeiten für Unternehmen im internen Einsatz

Voraussetzung für den Einsatz ist natürlich zunächst einmal, dass eine Firma HTML-Anwendungen und Firefox nutzt. Dann wäre es möglich aufbauend auf den Ergebnissen aus diesem Projekt Daten über die Arbeitsabläufe der Angestellten zu sammeln und auszuwerten.
Diese Daten könnten sowohl zu einer Analyse der Mitarbeiterleistung als auch zu einer Verbesserung der Software selber genutzt werden. Z.B. liesse sich so leicht feststellen, an welcher Stelle in einem Arbeitsablauf bestimmte Befehle immer wieder in der gleichen Reihenfolge gemacht werden müssen. Diese könnten dann durch eine „Abkürzung“ ersetzt werden.
Dieser Ansatz könnte auch in der Verhaltens- und Aufmerksamkeitsforschung ohne ein direktes wirtschaftliches Interesse genutzt werden.

6.2. Anwendungsmöglichkeiten für Privatanwender

Die Idee zu diesem Projekt entstand aus der Überlegung, dass sich das WWW als Umgebung gut dafür eignet, virtuelle Assistenten für Privatanwender zu trainieren. Ein Kritikpunkt an den vielen Assistenz-Systemen von Firmen wie Google, Amazon, Apple usw. ist die Praxis, die Verhaltensdaten der Nutzer auf Firmeneigenen Servern im Netz auszuwerten und zu speichern. Der Missbrauch dieser umfassenden Überwachungsdaten kann zu einer digitalen Totalüberwachung führen, deren Folgen kaum zu unterschätzen wären.
Dies ist vor allem dann widersprüchlich, wenn man bedenkt, dass ein digitaler Assistent umso besser funktioniert, umso besser er seinen Besitzer „kennt“. Dem Autor schien es logisch die Lösung dieses Dilemmas in einem Ansatz zu suchen, der dem Benutzer die vollkommene Kontrolle über seine Daten erlaubt. Die Entwicklung einer entsprechenden Anwendung soll zu einem späteren Zeitpunkt fortgeführt werden.

6.3. Möglichkeiten der Kommerzialisierung

Eine der am weitesten verbreiteten Wege moderne Web-Anwendungen zu finanzieren ist Werbung. Dadurch gibt ein Nutzer unweigerlich wieder Informationen über sich preis, jedoch deutlich weniger, als wenn ein Assistenzssystem dessen Funktionsprinzip eine möglichst umfassende Verhaltensanalyse ist, die entsprechenden Daten weiter gibt. Eine Web-Extension könnte z.B. zusätzliche Werbeinhalte während des Surfens einblenden. Oder auch eine eigene GUI für die Extension bereit erstellen, welche dann mit Werbebannern versehen ist. Es wäre auch denkbar, dass ein Nutzer sich bereit erklärt auf seinem Rechner eine Software zum Bitcoin-Mining laufen zu lassen.
Die duale Natur einer Extension mit nativen Programm-Anteilen (Anteilen die auf dem Nutzer Computer installiert sind), erlaubt es sowohl die Möglichkeiten zur klassischen dirketen Kommerzialisierung von Software (also direkte Zahlungen eines Kunden an den Hersteller), als auch indirekte Finanzierung über Werbung zu nutzen. Die ist ein grosser Vorteil für kleine Entwickler-Teams, welche originelle Ideen / Ansätze verfolgen die für einen grossen Nutzermarkt interessant sind, die jedoch wenig Ressourcen in die Vermarktung investieren können.

7. Ergebnis (Anwendungsbeispiel)

Funktion und Ablauf soll folgend an einem Beispiel vorgestellt werden. Dazu wurde nach der Installation der Software im Firefox Fenster die Seite https://de.wikipedia.org/wiki/Wikipedia:Hauptseite [wikiDe]aufgerufen.

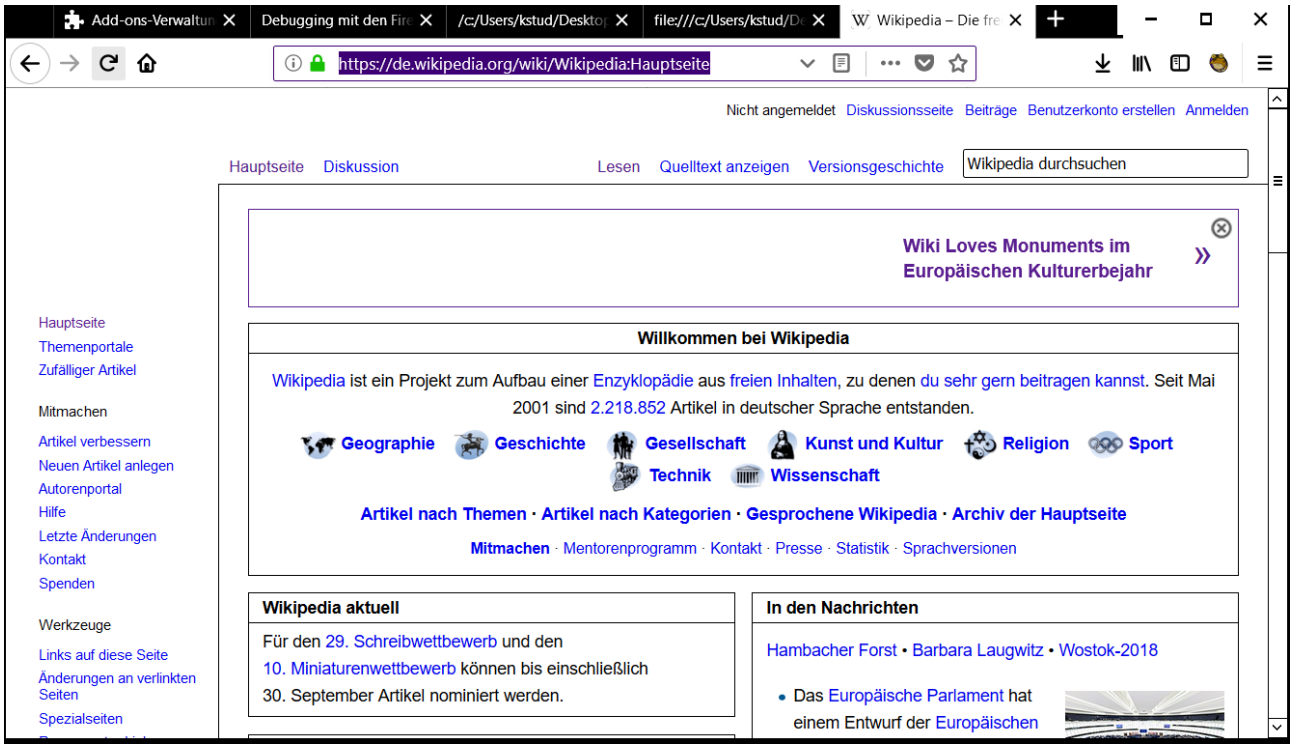

Abbildung 5: Beispiel Webseite: Wikipedia

In der rechten oberen Ecke erkennt man das Frosch-Symbol der Browser-Extension. Die aktive Extension reagiert auf Bewegungen des Mauszeiger über der Oberfläche der Webseite, immer dann wenn sich das „berührte“ DOM-Element ändert. Die Vorgänge werden durch die im Content und Background Script stehenden „Console“ Anweisungen in der Web-Entwickler Browser-Konsole von Firefox angezeigt.

Abbildung 6: Auszug der Browser-Konsole für die Beispiel-Webseite

Das Python Script legt schliesslich Ordner und Dateien an, in denen die Ergebnisse gespeichert werden.

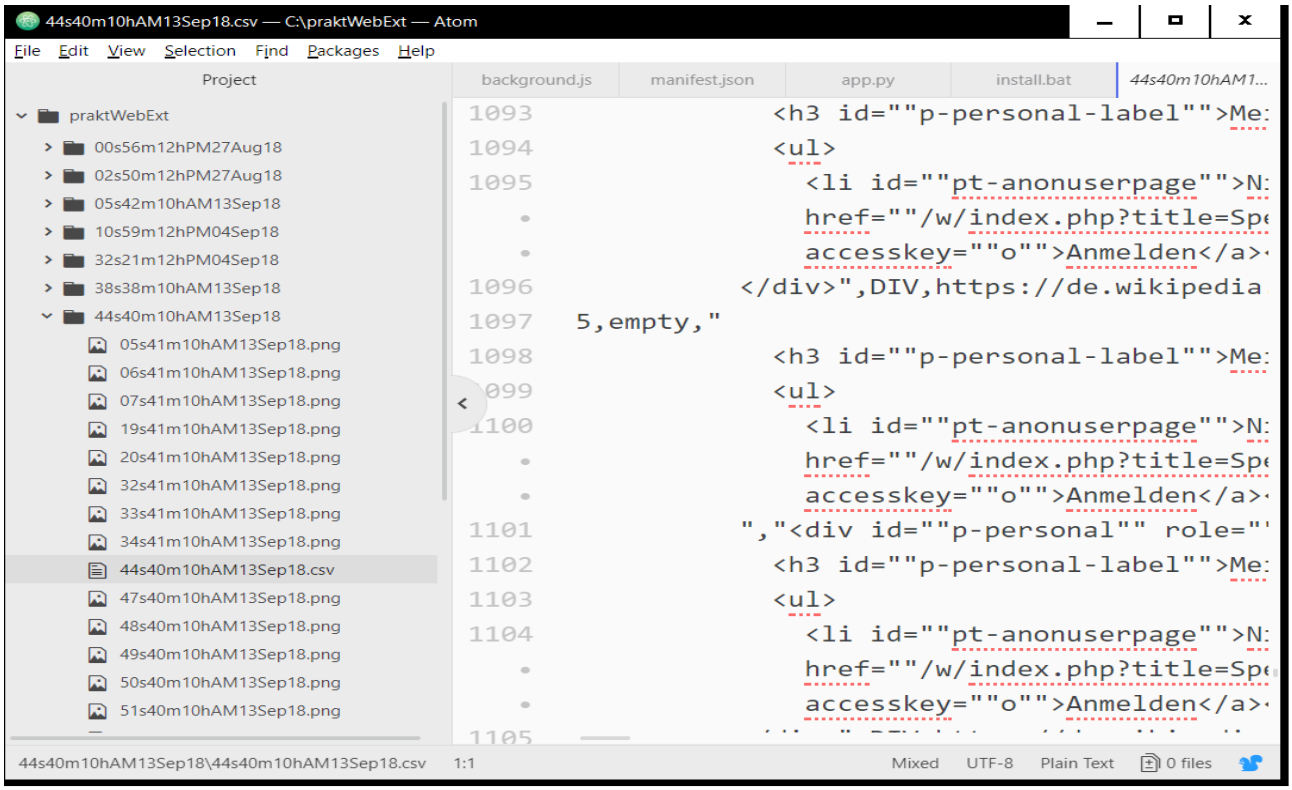

Abbildung 7: Atom-Editor Ansicht des Resultats der Speichervorgänge für die Beispiel-Webseite

8. Schlussbetrachtungen

Folgend wird auf Überschneidungen zwischen bisherigen Inhalten des Studiums und der Umsetzung der Projektarbeit eingegangen. Schliesslich folgt ein persönliches Fazit des Autors.

8.1. Bezüge zum Studiengang Informatik

Durch die durch den Autor bisher bearbeiteten Inhalte des Studiums wurden bereits die meisten Aspekte des Projekts abgedeckt. Natürlich würde es den Rahmen dieser Arbeit sprengen alle Symmetrien aufzuzeigen. Es folgt eine Übersicht über die Module die den grössten direkten Einfluss auf die Umsetzung hatten.

8.1.1. Wissenschaftliches Arbeiten, Qualitäts- und Projektmanagement

Dieses Modul diente als Rahmen für dieses Projektberichts. Die Basis wissenschaftlicher Arbeiten muss demnach die Einhaltung ethischer und formaler Rahmenbedingungen sein.
Ethisch in dem Sinne, dass die Arbeit selbst den Zweck verfolgen sollte der Menschheit gutes zu tun, als auch dadurch, dass geistige Urheberschaft klar gekennzeichnet sein sollte, man sich nicht mit anderer Leute Federn schmücken sollte.
Auch formal sollten wissenschaftliche Arbeiten, sich nach den in dem Modul beschriebenen Anforderungen richten. Das Modul stellt also eine ganze Reihe von impliziten Vorgaben und Standards für wissenschaftliche Arbeiten im jeweiligen Kontext / Umfang bereit. Im vorliegenden Fall handelt es sich z.B. um einen Praktikumsbericht / Projektarbeit mit dem Hintergrund einer Literatur- bzw. Experimentellen Arbeit.
Neben diesen relativ abstrakten Vorgaben, enthält das Modul auch sehr detaillierte Anweisungen zu Zitaten, Quellenangaben usw., welche ebenfalls Verwendung in der vorliegenden Arbeit fanden.

8.1.2. Software Engineering: agile Entwicklung, Entity Relationship

Direkt diesem Modul entnommen wurden die Hinweise zu Entity-Relationship Modellen für die Grafiken. Ansonsten hat der Autor überlegt sich an eines der Entwicklungsmodelle zu halten (Phasenmodell, V-Modell, usw.) sich letztlich jedoch für eine eigene Interpretation entschieden, welche agile Methoden mit geplanten Meilensteinen verknüpft.
Die Tatsache, dass der Autor sowohl die Technologien als auch den Aufbau erst recherchieren / erlernen musste, machte eine vorherige Planung unmöglich. Statt dessen, war es meist möglich die erlernten Prinzipien direkt in das Projekt einzufügen.

8.1.3. Informationsmanagement und Prozessmodellierung

Die APIs für die Kommunikation zwischen den verschiedenen Scripten, bzw. zwischen nativen Komponenten und der Extension Komponente entsprechen in ihrem Aufbau den Darstellungen aus der Geschäftsprozessmodellierung. Die jeweiligen „messages“ entsprechen den „Bestellungen“ aus den Modellen dieses Moduls, die verschiedenen Scripte und Komponenten den Abteilungen und Bereichen. Dieses Wissen erleichterte natürlich das Verständnis des zugrunde liegenden Ordnungsgedankens der Browser-Extensions.

8.1.4. Multimedia

In diesem Modul konnte der Autor erste Erfahrungen mit Javascript und HTML machen, was ein hervorragender Einstieg in das Arbeiten an Browser-Extensions war.

8.2. Fazit

Das Format des Projekts war eine Herausforderung. Auch wenn das technische Endergebnis nicht sonderlich Umfangreich ist, waren recht viele verschiedene für den Autor unbekannte Komponenten notwendig, was eine breite Recherche notwendig machte. Wenn man ihn einmal erschlossen hat, bietet der Rahmen einer Browser-Extension sehr viele Möglichkeiten für Entwickler, mit relativ wenig Aufwand praktisch sinnvolle Projekte umzusetzen. Durch die Anbindung an den Inhalt von Webseiten stehen einem Entwickler ohne weitere Anstrengungen gewaltige Mengen an Daten zu jedem denkbaren Thema zur Verfügung. Dabei muss bedacht werden, dass ein Abkratzen von Daten unter bestimmten Bedingungen illegal sein kann.
Die Native-Messaging Schnittstelle erlaubt es dann diese Daten direkt mit jedem beliebigen Programm über die Stdio-Schnittstelle weiter zu verarbeiten und die für Extensions sonst geltenden Beschränkungen zu umgehen, um direkt auf das Betriebssystem des Anwenders zuzugreifen. Die Hersteller-Übergreifende Standardisierung und Orientierung an vorhandenen Webstandards geht einher mit einer umfassenden und professionellen Dokumentation der APIs. Die Möglichkeit Programme auf die gleiche Art über Werbung zu monetarisieren wie Webseiten, ist ein sehr interessantes Feature für kleine aber innovative Software-Entwickler.

Das angestrebte Ziel des Autors eine Plattform für spätere Projekte zu schaffen wurde erreicht. Alle notwendigen Funktionen um ein digitales Assistenz-System umzusetzen scheinen gegeben. Im nächsten Schritt muss der Autor erst einmal erarbeiten, wie genau KI-Systeme funktionieren und ob eine Realisierung des Vorhabens überhaupt realistisch ist. Dann könnte z.B. die Bachelor-Arbeit auf den Ergebnissen dieses Projekts aufbauen.

Quellenverzeichnis

w3ecg: , , , https://www.w3.org/community/browserext/
mExtDoc01: , , , https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions/manifest.json
gExtDoc02: , , , https://developer.chrome.com/extensions/manifest
mExtDoc02: , , , https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions
gExtDoc03: , , , https://developer.chrome.com/extensions
mExtDoc03: , , , https://developer.mozilla.org/de/docs/Mozilla/Add-ons/WebExtensions
gExtStr01: , , , https://chrome.google.com/webstore/category/extensions
gExtStr02: , , ,
https://chrome.google.com/webstore/detail/disconnect/jeoacafpbcihiomhlakheieifhpjdfeo
gExtStr03: , , ,
https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom?hl=de
lScrap01: , , , http://www.rechtswissenschaft-verstehen.de/lexikon/screen-scraping/
hqSel01: , , , https://www.seleniumhq.org/
w3MoMve: , , , https://www.w3.org/TR/uievents/#event-type-mousemove
mSendMe: , , , https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/runtime/sendMessage
mNativeMe: , , , https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging
oPythDoc: , , , https://docs.python.org/2/using/windows.html
shortPath: , , , https://superuser.com/questions/348079/how-can-i-find-the-short-path-of-a-windows-directory-file
wikiDe: , , , https://de.wikipedia.org/wiki/Wikipedia:Hauptseite