Noah Große Starmann (Matrikel-Nummer: 758077)

Ramona Plogmann (Matrikel-Nummer: 758046)

Expose

In dieser Hausarbeit zum Fach Rich Media Applications wird die Entwicklung einer App zum Zeigen von Informationen über Bahnhöfe mit Ionic beschrieben.

Entwicklung einer bahnhofs-app

Rich Media Applications Hausarbeit WS17/18, Dozent: Dipl.-Inf. Björn Plutka

**Inhaltsverzeichnis**

1 Einleitung

* 1. Ideenfindung
  2. Ziele
  3. Geplante Struktur

1. Verwendete Technologien
   1. Google Maps API
   2. Ionic Framework
   3. REST API
2. DBOpenData
   1. Allgemeines
   2. Bahnhofsfotos API
   3. FaSta API
   4. Parkplatz API
   5. Stada API
   6. Reisezentren API
3. Implementierung

4.1 Verwendete Entwicklungsumgebung

# 1. Einleitung

## 1.1 Ideenfindung

Zu Beginn dieser Arbeit soll erläutert werden, wie die Idee zur Bahnhofsapp entstanden ist. Aufgrund der Aufgabenstellung zur Entwicklung einer App, die auf Daten zugreift und diese dynamisch darstellt, haben wir uns zunächst umgesehen, welche Art von Daten von in Deutschland bekannten Firmen zur Verfügung gestellt werden. Wichtig ist uns dabei gewesen, dass diese Daten zum Einen den rechtlichen Aspekt der freien Nutzbarkeit erfüllen und zum Anderen möglichst gepflegt und aktuell sind, um so den größtmöglichen praktischen Nutzen aus der App ziehen zu können. Außerdem sollte der zur Verfügung gestellte Datensatz unter anderem Geodaten enthalten, beziehungsweise sollten die Daten einen Zusammenhang zur Geographie erkennen lassen um sie so für Google Maps nutzbar machen zu können. Den Anfang der Suche markierte dabei das „Geodaten Portal Niedersachen“[[1]](#footnote-1). Da die dort bereitgestellten Datensätze aber entweder keine praktische Relevanz oder keine Aktualität aufwiesen, oder aber nicht in unserem Interessengebiet lagen, haben wir uns noch einmal umorientiert. Eine weitere Suche führte uns schließlich zu den verwendeten „DBOpenData“[[2]](#footnote-2), wo im Showcase bereits interessante Projekte zu finden sind und außerdem eine vergleichsweise gute Dokumentation der einzelnen APIs gegeben ist. Die Wahl innerhalb der OpenData fiel dabei auf die Daten der einzelnen Bahnhöfe.

## 1.2 Ziele

Im Vordergrund soll zunächst einmal die einfache Bedienbarkeit liegen. Da die Zielgruppe der App aus der Natur der Daten heraus Bahnfahrer aller Altersgruppen sein sollen ist es wichtig, die App für alle Altersklassen optisch ansprechend und möglichst intuitiv bedienbar zu gestalten. Es muss sowohl für junge als auch für ältere Nutzer, die nicht mit Smartphones oder ähnlichem aufgewachsenen sind, auf den ersten Blick zu verstehen sein, wie die App funktioniert und welche Schritte getätigt werden müssen, um zum gewünschten Ergebnis zu gelangen. Daher verzichten wir möglichst auf eine Einstellungsseite, die mit verschiedenen Einstellungen, wie zum Beispiel dem Auswählen eines Intervalls für die Standortbestimmung, gerade beim älteren Teil der Zielgruppe für Verwirrung sorgt. Des Weiteren soll die App kostenlos sein, da wir von der Bahn öffentlich zur Verfügung gestellte Daten nutzen, und darüber hinaus einen praktischen Nutzen im Alltag darstellen.

# 2. Verwendete Technologien

## 2.1 Ionic Framework

Das Ionic-Framework ist ein Open-Source Framework zur Erstellung nativer Apps.[[3]](#footnote-3) Eine native App bezeichnet dabei eine Anwendung, die speziell für das Betriebssystem des jeweiligen Endgerätes gestaltet ist. Vermarktet werden diese meistens über die individuellen Verkaufsplattformen der Betriebssysteme (z.B. Google Market, App Store, etc.).[[4]](#footnote-4)

Die Komponenten der nativen App werden dabei mithilfe von HTML, CSS und JavaScript zusammengesetzt und können so auch von Webentwicklern schnell erstellt werden.[[5]](#footnote-5) Ein erklärtes Ziel des Frameworks ist es dabei, dass Web-Entwickler schon vorhandene Kenntnisse weiternutzen können und möglichst wenig Neues erlernen müssen.[[6]](#footnote-6) Ein weiterer Fokus des Frameworks liegt auf simpler UI Interaktion und dem plattformspezifischen Aussehen der zu erstellenden App.[[7]](#footnote-7) Die Realisierung der Funktionen erfolgt dabei in einer Art und Weise, die sich gut als „Stack“ beschreiben lässt.[[8]](#footnote-8) Auf der ersten Stufe steht das Ionic-Framework an sich. Hiermit kann eine gut aussehende Benutzeroberfläche erstellt werden, die im Browser getestet werden kann. Außerdem steht ein Kommandozeilenwerkzeug (CLI) zur Verfügung, um bequem Verwaltungsaufgaben, wie das Erstellen von Seiten, ausführen zu können.[[9]](#footnote-9) Beim Ionic Paket handelt es sich um ein npm-Modul[[10]](#footnote-10), welches über die Plattform Node.js installiert werden kann. Unter Windows ist es dafür nötig, sich ein Terminal herunterzuladen, über welches Node.js dann ausgeführt werden kann, unter Linux und macOS wird dieses nicht benötigt. Des Weiteren wird das Cordova Plugin verwendet, um aus dem erstellten Dokument schließlich eine native App zu konstruieren.

Doch wie ist das Framework lizenzrechtlich gestaltet? Zunächst einmal ist zu sagen, dass das Framework unter der sogenannten „permissive MIT license“ entwickelt wird.[[11]](#footnote-11) Die Dokumentation und die Website hingegen stehen unter der Apache 2 Lizenz.[[12]](#footnote-12)

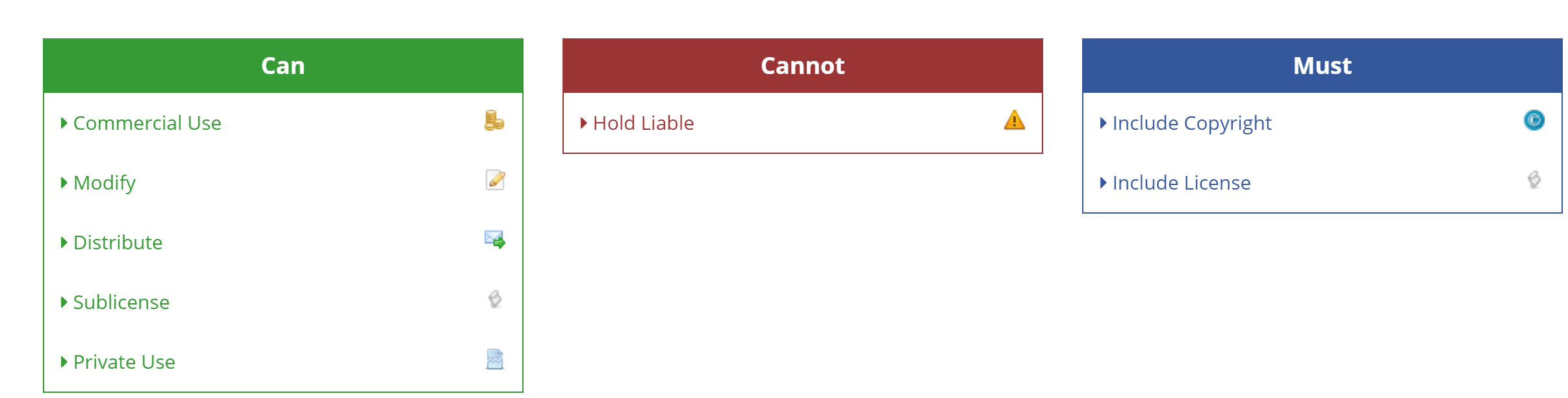
Bei Benutzung der MIT Lizenz ist dem Benutzer prinzipiell eine freie Nutzung des unter Lizenz gestellten Inhalts möglich. Zu beachten ist dabei nur, dass der Lizenzhinweis in die erstellte Arbeit zu integrieren ist. Außerdem muss das originale Copyright zu finden sein.[[13]](#footnote-13)

Abbildung 1: Lizenzrechtliche Möglichkeiten bei Verwendung eines unter MIT Lizenz stehenden Inhalts, Quelle: https://tldrlegal.com/license/mit-license

Trotz der Open Source Lizenz bietet Ionic auch kostenpflichtige Möglichkeiten in seinem Framework an. So lässt sich beispielsweise ein Dienst nutzen, der Ionic Apps auch zum Testen auf Devices verfügbar macht, ohne dass man diese mit Cordova übertragen muss. Außerdem lassen sich Services, wie eine längere Fehlerhistorie kostenpflichtig hinzubuchen. Dabei gibt es verschiedene Modelle für einzelne Entwickler, Teams oder Firmen.[[14]](#footnote-14)

## 2.2 REST API

In diesem Abschnitt soll die im Verlauf der App Entwicklung benutzte REST API zur Sprache kommen und genauer erläutert werden.

Die Abkürzung „REST API“ steht für „Representational State Transfer Application Programming Interface“.[[15]](#footnote-15) Eine Programmierung einer Schnittstelle nach dem REST Programmierparadigma setzt sich dabei zum Ziel, eine Kommunikation zwischen Client und Server nach den Prinzipien des World Wide Web zu ermöglichen. [[16]](#footnote-16) Vornehmlich handelt es sich bei dieser Kommunikation um den Zugriff auf Dateien, der gemanagt werden soll. Bevorzugtes Protokoll ist HTTP.

Ein Dienst muss, um das Paradigma zu erfüllen, dabei genau sechs Eigenschaften besitzen, die ihn als REST API auszeichnen.

Zum ersten verlangt das Paradigma ein Client-Server System im üblichen Sinne, dass der Server auf Anfrage Daten an den Client liefert. Die zweite Bedingung ist es, dass Client und Server zustandslos miteinander kommunizieren. Konkret bedeutet dies, dass der Server nicht selber auf gespeicherte Daten zugreift. Jede Anfrage eines Clients muss so formuliert sein, dass der Server daraus die für den Datenzugriff nötigen Informationen gewinnen kann. Dadurch sollen Zuverlässigkeit in der Datenhaltung und Skalierbarkeit verbessert werden. Jedoch treten auch Nachteile in Bezug auf die Netzwerk-Performance durch diese Richtlinie auf. [[17]](#footnote-17) Eine weitere Bedingung ist, dass für Anfragen eines Clients http-Caching verwendet wird, um so unnötige Abfragen zu umgehen und den Netzwerk Traffic zu schonen. Problematisch und wichtig zu wissen für die Anwender, die auf die REST Schnittstelle zugreifen, ist es, dass durch eben dieses Caching die Möglichkeit besteht, dass auf bereits veraltete Daten aus dem Cache zugegriffen wird. Daher werden gespeicherte Informationen mit speziellen Flags versehen, die sie danach kennzeichnen, ob sie kurzfristig gespeichert werden können oder nicht. Schnittstellentechnisch setzt die REST API auf eine einheitliche Schnittstelle für alle durch den Server verwalteten Daten. Nachteilig ist dabei, dass durch diese Vereinheitlichung Informationen oft nicht optimal angepasst dargestellt werden können. Des Weiteren sind REST Systeme mehrschichtig aufgebaut, sodass jede Schicht nur die notwendigen Informationen bekommt. Dadurch wird die Kommunikation entscheidend vereinfacht, da keine unnötigen Informationen dargestellt werden. Die sechste und letzte Bedingung ist optionaler Natur. Sie legt fest, dass sogenannter Code-on-Demand möglich sein muss. Dieser wird zur lokalen Ausführung „beantragt“ und erweitert die Funktionalität eines Clients. Dies kann beispielsweise ein Skript sein, dass von einer REST API heruntergeladen wird.

Wie oben erwähnt, setzt eine REST API meistens HTTP zur Kommunikation ein. Dabei werden die Befehle GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS, CONNECT und TRACE dem Client zur Verfügung gestellt. Davon werden einige im Rahmen dieser Arbeit verwendet.

# 3. DBOpenData

## 3.1 Allgemeines

Im Fokus dieses Abschnittes sollen die DBOpenData [[18]](#footnote-18) stehen, die den Grundstein des Projektes bilden.

Hierbei handelt es sich um von der Deutschen Bahn öffentlich zur Verfügung gestellte Daten, die sich rund um den Bahnbetrieb drehen. Zur Verfügung gestellt werden sie in einem Online Portal. Um die Entwicklung voranzutreiben finden regelmäßig Hackathons statt, bei denen die Daten mit anderen Interessierten gemeinsam genutzt werden können.

Die gebotenen Daten werden unter einer freien Lizenz verwaltet. Sie stehen jedem „in unterschiedlichen Formaten, dauerhaft und kostenlos“[[19]](#footnote-19) zur Verfügung, können also problemlos weiterverwendet werden.

Für den Zugriff auf eine der angebotenen APIs wird ein Bearer Token benötigt, der dann an die jeweilige HTTP Anfrage angehängt wird. Um diesen Token zu bekommen muss man einen Account auf der Seite der OpenData erstellen[[20]](#footnote-20), mit dem sich dann die jeweiligen APIs abonnieren lassen. Durch das Subscriben erhält man dann auch den nötigen Key für die jeweilige API. Jede API hat außerdem eine Konsole, mithilfe derer man im Browser zu Testzwecken Abfragen starten kann, um einen Überblick über die gelieferten Daten zu erhalten.

## 3.2 Bahnhofsfotos

Die Bahnhofsfotos API stellt Fotos der jeweiligen Bahnhöfe über eine Schnittstelle zu Verfügung. Das Konzept ist, dass Reisende ihre eigenen Fotos zu den Bahnhöfen online zur Verfügung stellen können, die dann über die API verwendet werden können. Dazu wird eine Internetseite[[21]](#footnote-21), sowie auch eine App für Android Mobiltelefone[[22]](#footnote-22) angeboten.

Die Zuordnung der einzelnen Stationen zu den Fotos erfolgt über eine stationID. Bei der Bahnhofsfotos API ist es zusätzlich notwendig, einen Ländercode bei der GET Abfrage anzugeben, um so das richtige Land zu erhalten. Die in der App verwendete GET Abfrage über die Bahnhofs ID gibt den Titel des Fotos, den Fotografen, dessen URL, die Lizenz und die Koordinaten zurück. Das Foto wird nicht als Datei dazu geliefert, es wird nur eine URL zum Inhalt zurückgegeben. Verschiedene Abfragen erlauben es, sowohl die Fotografen der jeweiligen Fotos über eine separate Anfrage herauszufiltern, als auch sich Statistiken über die Bilder zu beschaffen. Des Weiteren besteht die Möglichkeit, Fotos per POST hochzuladen und diese auch zu registrieren. Wir nutzen diese API jedoch lediglich zum Anzeigen des zugehörigen Fotos eines Bahnhofs. Letztendlich haben wir uns dazu entschieden, statt id-basiert für den aktuellen Bahnhof jeweils eine neue Anfrage zu starten, beim Start der App direkt alle Fotos zu laden, da so während der Nutzung Wartezeiten und plötzlich auftauchende Fotos, die erst angezeigt werden, wenn die Anfrage beendet ist, vermieden werden können. Mit der von uns genutzten Anfrage „de/stations?hasPhoto=true“ lässt sich eine zusätzliche Beschleunigung erzielen, da nur die Einträge der Stationen geladen werden, die tatsächlich ein Foto enthalten.

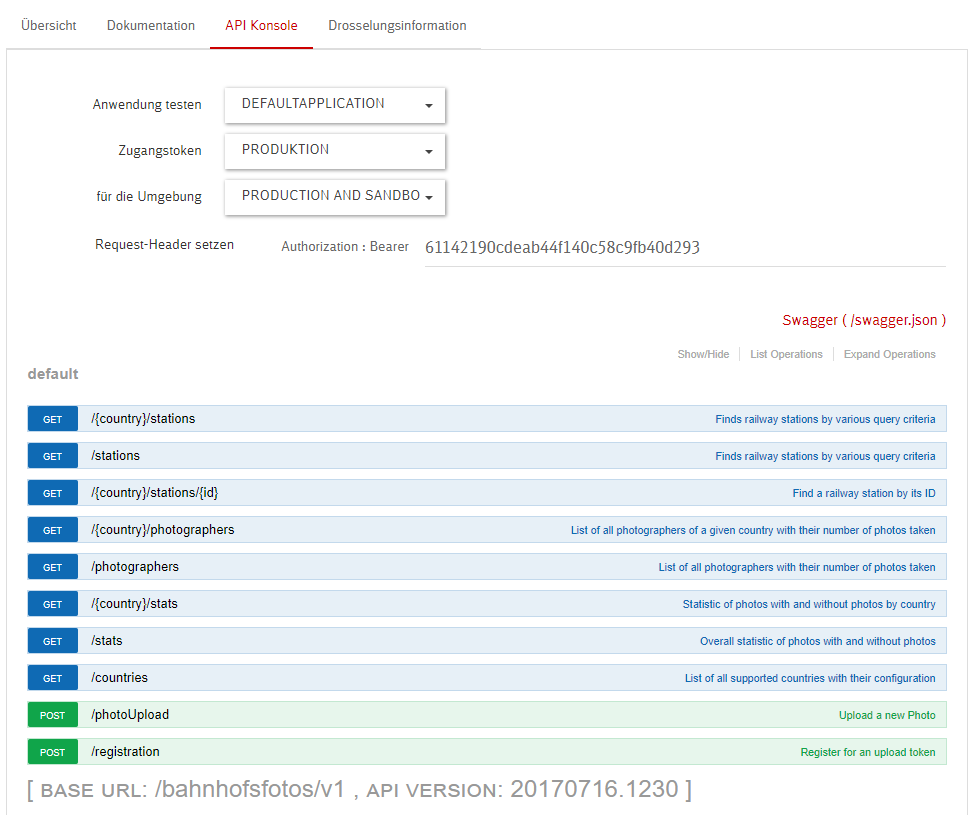


Abbildung 3: Screenshot der API Konsole, mit den zu Verfügung stehenden Möglichkeiten, Quelle: selbst aufgenommen, 21.02.18, 16:19

## 3.3 FaSta API

FaSta steht für Facilities Status und dient dazu, über den Betriebszustand von Rolltreppen und Aufzügen zu informieren. Verwendet worden ist hier die FaSta Version 2, da die erste Version nicht mehr lange unterstützt wird. Insgesamt stellt die FaSta API drei Möglichkeiten für eine Datenabfrage zur Verfügung. Zum einen lassen sich alle „facilities“ ausgeben. Dabei werden die Equipmentnummer, der Typ, eine Beschreibung, die Koordinaten, der Status und die jeweilige Station ID mitgegeben. Die zweite Möglichkeit ist, die gesuchte Rolltreppe anhand ihrer Equipmentnummer zu finden und die Daten dazu auszugeben. Das gelieferte Datenpaket beinhaltet dabei dieselben Daten, wie auch bei der 1. Möglichkeit der Abfrage. Beide Möglichkeiten sind im Rahmen dieser App nicht verwendet worden, da keine Relevanz besteht, entweder alle Daten abzufragen, oder den Bezug über die jeweilige Equipmentnummer herauszustellen. Die von uns genutzte, dritte Möglichkeit ist es, passend zur ID der Bahnhöfe die Aufzüge und Rolltreppen zu finden.

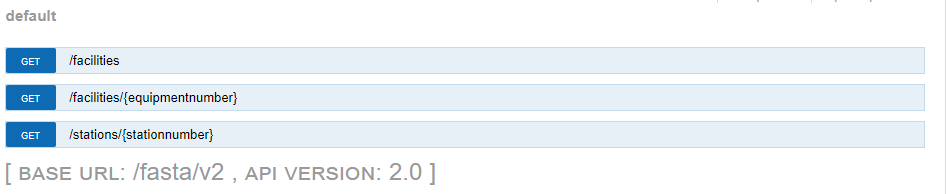


Abbildung 4: Screenshot der API Konsole der Fasta API mit den jeweiligen Links für die oben beschriebenen Verwendungsmöglichkeiten, Quelle: selbst erstellt, 21.02.18, 16:37

## 3.3 Parkplatz API

Als nächstes soll die Parkplatz API beschrieben werden. Sie ist online unter dem Namen „BahnPark- V1“ zu finden und stellt Daten über die an den Bahnhöfen vorhandenen Parkplätze und deren Belegung (z.T. auch voraussichtliche Belegung) zur Verfügung. Für einen Bahnhof können mehrere Parkplätze existieren.

Jedem einzelnen Parkplatz ist eine eigene ID zugeordnet um ihn so identifizieren zu können. Es gibt 9 Möglichkeiten für eine Abfrage (s. Abb. @TODO). Zum einen lassen sich alle Parkplätze ausgeben. Dabei wird unterschieden zwischen „spaces“ und „spaces/pit“. Für den Betrachter ist mangels einer Dokumentation der Daten nicht herauszufinden, wo genau der Unterschied besteht. Die Daten der Antwort sind bei beiden identisch. Als nächste Möglichkeit kann auf die Belegungsdaten der jeweiligen Parkplätze zugegriffen werden. Wir tun dies bei Aufruf der Parkplatzseite ID-basiert für jeden Parkplatz der aktuellen Station. Zuletzt werden für einige wenige Parkplätze Prognosen bereitgestellt. Auf diese kann nur gezielt per Parkplatz-ID zugegriffen werden.

Die letzte Möglichkeit, die die Parkplatz API bietet, ist eine extrem verkürzte Version der Stationsdaten API, die im nächsten Abschnitt beschrieben wird.

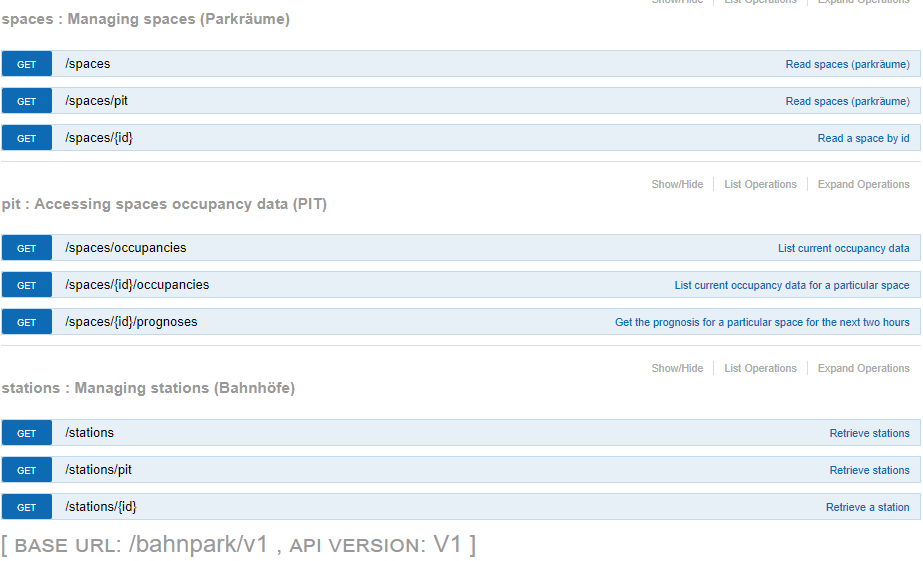


Abbildung 5: Screenshot der API Konsole mit den Abfragemöglichkeiten der Parkplatz API, Quelle: selbst erstellt, 21.02.18. 16:53

## 3.4 StaDa API

Die meisten der von uns verwendeten Daten liefert die StaDa API. „StaDa“ steht für „Station Data“ und stellt Informationen über einzelne Bahnhöfe zur Verfügung.

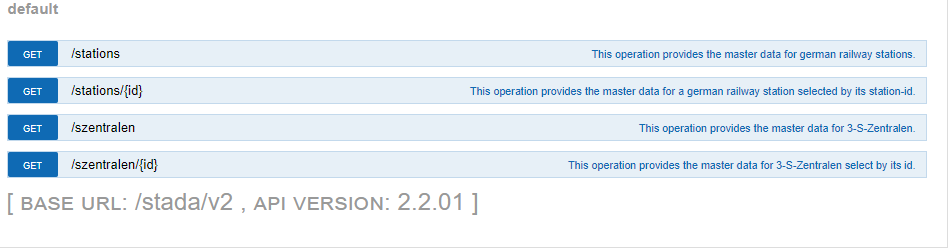
Die API stellt dem Nutzer vier verschiedene Möglichkeiten für die Abfrage von Daten zur Verfügung. Zum ersten lassen sich alle Bahnhöfe mit ihren Daten abrufen. Das Datenpaket beinhaltet dabei eine Vielzahl an Informationen, unter anderem die Nummer, den Namen, die Adresse, das Bundesland, verschiedene Angaben zur Infrastruktur (z.B. ist WLAN vorhanden, existiert ein Taxistand oder eine Anbindung an den öffentlichen Nahverkehr, etc.), sowie die Öffnungszeiten des Bahnhofs und die Verfügbarkeit der dortigen Servicemitarbeiter. Des Weiteren kann man über die Station ID obige Infos zu einem speziellen Bahnhof erfragen und Details über 3-S-Zentralen[[23]](#footnote-23) herausfinden. Dies hat für unser Projekt jedoch keine Relevanz.

Abbildung 6: Screenshot der API Konsole der StaDa API mit den möglichen Abfragen, Quelle: selbst aufgenommen, 21.02.18, 17:19

## 3.5 Reisezentren API

Die Reisezentren API stellt Daten über die Reisezentren in Deutschlands Bahnhöfen zu Verfügung.

Hier existieren vier Möglichkeiten der Abfrage der Daten. Zunächst können, wie auch bei den vorhergegangen APIs die Daten aller Reisezentren abgefragt werden. Das zurückkommenden Datenpaket enthält den Namen, die Adresse und die Öffnungszeiten. Des Weiteren existiert die Möglichkeit, über eine Station ID die dazugehörigen Reisezentren zu finden. Diese korreliert jedoch nicht mit dem ID-Feld „number“ des StaDa-Response-Bodys, das von den anderen APIs zur Identifikation der Station genutzt wird. Welche ID die Reisezentren API hier verlangt, konnten wir nicht in Erfahrung bringen, weshalb wir die Möglichkeit nutzen, das zugehörige Reisezentrum über die Koordinaten dem entsprechenden Bahnhof zuzuordnen, um in der Detailansicht des aktuell ausgewählten Bahnhofs die Öffnungszeiten des Reisecenters anzeigen zu lassen.



Abbildung 7: Screenshot der API Konsole der Reisezentren API mit den zur Verfügung stehenden Möglichkeiten zur Abfrage, Quelle: selbst aufgenommen, 21.02.18, 17:33

# 4. Implementierung

## 4.1 Verwendete Entwicklungsumgebung

Wir nutzen vorrangig nicht den in der Vorlesung verwendeten Editor Sublime Text, sondern die Entwicklungsumgebungen WebStorm und phpStorm der Firma JetBrains. Sie sind für Studenten kostenlos nutzbar. Sie bieten vorinstallierte Bibliotheken für die Entwicklung mit Ionic und verfügen außerdem über ein integriertes Kommandozeilen-Interface, mit dessen Hilfe sich direkt im Codefenster Konsolenbefehle ausführen lassen. Des Weiteren ist ein durch die Verwendung der genannten IDEs entstehender Vorteil, dass die Entwicklungsumgebungen über eine unserer Meinung nach gut konzipierte Code-Vervollständigung verfügen, die den Workflow vereinfacht. Außerdem existieren viele sehr nützliche Shortcuts für Dateinavigation und Codebearbeitung.

Zusätzlich installiert worden ist natürlich das eigentliche Ionic Framework und Android Studio inklusive der nötigen SDKs zum Testen.

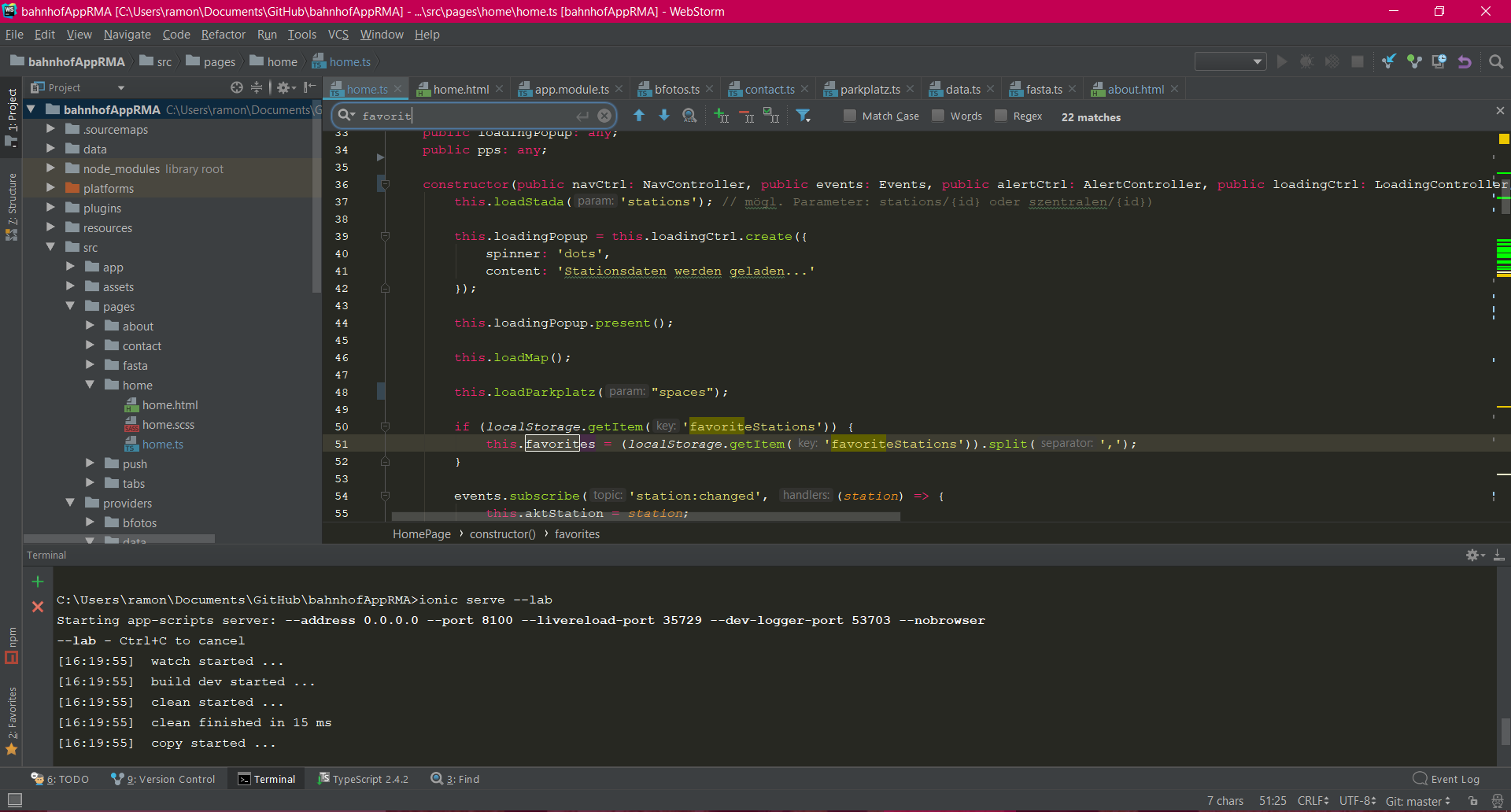


Abbildung 8: Screenshot der WebStorm Entwicklungsumgebung, Quelle: selbst aufgenommen, 23.02.18, 16:20

## 4.2 Nutzung der Google Maps API

Eine Google Maps Karte wird an zwei Stellen unserer App verwendet.

Zum einen stellt sie die Hauptkomponente der Home Page dar, über die man Stationen findet und auswählt.

Die Methode zum Finden nahegelegener Stationen macht sich die sogenannten „Bounds“ [[24]](#footnote-24) zunutze. Diese stellen je ein LatLng-Objekt für die Nordost- und eines für die Südwestecke des derzeitigen Kartenausschnittes zur Verfügung. Diese Koordinaten werden dann einzeln mit den Stationskoordinaten verglichen, um so diejenigen Stationen herauszufiltern, die sich im betreffenden Ausschnitt befinden. Für jede der gefundenen Stationen wird ein Marker gesetzt. Eventuell vorher vorhandene Marker werden entfernt. Dies ist besonders wichtig für den Fall, dass der Kartenausschnitt nach der letzten Suche verkleinert wurde und die neue Suche demnach nicht auf alle derzeit gesetzten Marker zutrifft. Bei dieser Art, Stationen zu suchen, wird der Kartenausschnitt ausschließlich durch den Nutzer manipuliert.

Über das Suchfeld lässt sich gezielt eine Station ihrem Namen nach auswählen. Nach Klick auf diese Station wird ein Marker für sie hinzugefügt. In diesem Fall wird nur ein Marker entfernt, der sich unter Umständen bereits auf ebendieser Stelle befindet, um Duplikate zu vermeiden. Die restlichen Marker werden an ihrem Ort belassen. Des Weiteren wird die Karte bei gleichbleibendem Zoom auf den gewählten Bahnhof zentriert. Das entsprechende Detailfenster wird eingeblendet. Google Maps stellt eine Pan-By-Methode zur Verfügung, mithilfe derer die Karte dann nach oben verschoben wird, damit sich auch nach Öffnen des Detailfensters der neu gesetzte Marker im sichtbaren Kartenausschnitt befindet.

Der zweite Verwendungsort einer Google Maps Karte ist die Fasta Page. Hier wird die Karte auf den aktuellen Bahnhof zentriert geöffnet und zeigt die für diesen vorhandenen Aufzüge und Rolltreppen. Dabei werden die beiden Anlagentypen jeweils mit spezifischen Markern gekennzeichnet. Dem Nutzer wird so auf den ersten Blick deutlich, um was für eine Anlage es sich handelt und ob sie derzeit genutzt werden kann. Hierbei nutzen wir die von der API zur Verfügung gestellte Funktion, eine benutzerdefinierte Grafik als Markericon zu verwenden. Je nach Anlage wird dann dynamisch über den Dateinamen der Grafik der passende Marker für jede Anlage hinzugefügt. Auch die Größe der Icons lässt sich individuell einstellen. So ist das Markericon beim Anzeigen eines Parkplatzes etwas größer abgebildet, da es zu diesem Zeitpunkt die einzige Information auf der Karte ist und dem Nutzer so schneller ins Auge fällt.

## 4.3 Grundfunktionen der Home Page

Nach dem erfolgreichen Abrufen der Stationsdaten war unser erstes Ziel, eine Station zu suchen und auf der Karte anzeigen zu lassen. Die eventbasierte Funktionalität des Ionic-Suchfeldes . Darunter wird eine Liste „filteredStations“ angezeigt, die den Inhalt eines zusätzlichen Arrays aller Stationsnamen ausgibt. Die Wahl, zum Hauptarray mit allen Stationsdaten noch ein zusätzliches zu speichern, welches nur die Namen enthält, trafen wir aus zwei Gründen: Die Stationen müssen bei jeder Veränderung des Suchfeldinhaltes neu gefiltert werden. Demnach ist es notwendig, dass ein Array zu jedem Zeitpunkt alle Stationen enthält, um das Ergebnis bei verändertem Input zurücksetzen zu können, weshalb für die gefilterten Stationen ein zweites Feld gebraucht wird. Dieses entschieden wir nur mit den Namen der Stationen zu befüllen, nachdem wir zu dem Schluss gekommen waren, dass das wiederholte Filtern umso schneller geht, je weniger Daten das Array enthält.

Dem Nutzer wird die Möglichkeit geboten, Stationen als Favoriten zu markieren. Diese werden dann in einem dritten Array gespeichert, welches ebenfalls nur die Stationsnamen enthält. Um die Favoriten dauerhaft zu speichern, wird das Array im Localstorage des Geräts gesichert und bei jeder Änderung auch dort aktualisiert. Beim Starten der App wird es im Konstruktor der Home Page aus dem Localstorage geladen, falls ein entsprechender Eintrag vorhanden ist.  
In der Ergebnisliste einer Stationssuche werden die Favoriten mit einem Herz gekennzeichnet. Mithilfe eines Knopfes in der oberen rechten Ecke der Karte lässt sich eine Liste der Favoriten anzeigen. Hierbei werden die Favoriten in das Stationsnamen-Array kopiert, da sich so die bereits vorhandenen Funktionen der App möglichst effektiv nutzen lassen. Zusätzlich wird die Searchbar verkleinert, um Platz für einen Knopf zu machen, der es erlaubt, die Favoriten wieder zu schließen.

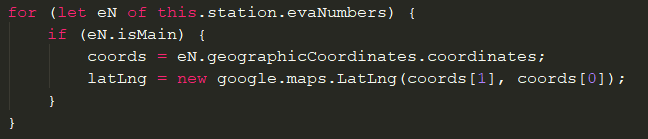
Die Auswahl einer bestimmten Station – nach einer Suche oder der Favoritenliste – ruft die Funktion foundStation auf. Die Javascript-Funktion find erlaubt eine einfache Auswahl der aktuellen Station über deren Namen, welcher von der View übergeben wird. Zunächst wird ein Marker für die ausgewählte Station hinzugefügt. Bei dem Durchsehen der Daten, die die Stada-API liefert, konnten wir jedoch zunächst keine Koordinaten finden, die hier zwingend benötigt werden, weshalb wir auf Reverse-Geocoding zurückgriffen. Dies erlaubt eine Gewinnung der Koordinaten aus der Bahnhofsadresse. Später haben wir tiefer geschachtelt in sogenannten evaNumbers des Datenpaketes doch noch Koordinaten der Bahnhöfe entdeckt und den Code dementsprechend angepasst. Teilweise gibt es für einen Bahnhof aber mehrere evaNumbers, von denen nur eine die gesuchten Koordinaten enthält. Dies lässt sich jedoch über das Attribut isMain herausfinden:

Abbildung @TODO: Finden der Koordinaten des aktuellen Bahnhofs

Für jeden neuen Marker wird ein Listener hinzugefügt. Dieser bewirkt, dass bei einem Klick auf den Marker die zugehörige Station als aktuelle Station bekannt gemacht und das Detailfenster geöffnet wird. So kann man zwischen mehreren auf der Karte angezeigten Station ohne erneute Suche hin- und herwechseln.

## 4.4 Aktualisierung des aktuellen Bahnhofs

Ein Problem auf das wir bei der App-Programmierung stießen, war die seitenübergreifende Bekanntmachung des aktuellen Bahnhofs. Die Schwierigkeit lag darin, dass man nicht nur über Buttons auf den Seiten selbst zwischen diesen wechseln kann, sondern auch über die Tabs am unteren Bildschirmrand, weswegen die Übergabe als Navigation Parameter bei einem Push nicht in Frage kam.

Eine naheliegende Lösung stellten unserer Meinung nach Events dar. Dieses Ionic-Feature bietet die Möglichkeit des seitenübergreifenden Nachrichtenaustauschs inklusive Parameterübergabe. Eine Seite veröffentlicht ein Event, beispielsweise die Änderung des aktuellen Bahnhofs, und liefert dabei den neuen Bahnhof gleich mit. Die anderen Seiten abonnieren diese Events und aktualisieren dann ihre aktuelle Station auf die mitgelieferte. Auch gibt es die Möglichkeit, in diesem Zuge noch andere Funktionen auszuführen, so wie zum Beispiel auf der Detailseite das Laden, ob Parkplatzinformationen zum aktuellen Bahnhof existieren, wovon abhängt, ob der Button, der auf die Parkplatzseite verlinkt, angezeigt wird oder nicht.

Später stellte sich heraus, dass die Lösung nicht ausreichend war, da Seiten erst Events abonnieren können, wenn der Konstruktor bereits aufgerufen wurde. Wählte man also einen Bahnhof, und öffnete dann den Tab „Details“, fehlte dort die Information, dass schon ein aktueller Bahnhof existiert, da das Event, das bei der Bahnshofsauswahl ausgelöst worden war, noch nicht von der Detailseite registriert worden war. Der Konstruktor war nämlich erst nach Klick des Tabs aufgerufen worden und somit wurde auch dort erst das Event abonniert.

Aus diesem Grund erstellten wir den Data Provider. Immer wenn das „station:changed“-Event veröffentlicht wird, wird der neue Bahnhof auch in das aktStation-Attribut des Data Providers geschrieben. Dort ist dann der aktuelle Bahnhof schon verfügbar, auch wenn die Detail- oder die Parkplatzseite das erste Event verpasst hat.

# Quellen- und Abbildungsverzeichnis

<https://www.geodaten.niedersachsen.de/startseite/gdini/open_data_portale/open-data-136000.html>, Zeitpunkt des Zugriffes: 06.02.18, 18:40

<http://data.deutschebahn.com/>, Zeitpunkt des Zugriffs: 06.02.18, 19:00

<https://de.wikipedia.org/wiki/Ionic_(Framework)#cite_note-2>, Zeitpunkt des Zugriffs: 10.02.18, 12:20

<https://ionicframework.com/docs/v1/overview/>, Zeitpunkt des Zugriffs: 10.02.18, 12:25

https://de.ryte.com/wiki/Native\_App, Zeitpunkt des Zugriffs: 10.02.18, 12:20

Chris Griffith: Mobile App Development with Ionic 2, First Edition April 2017

<https://www.thoughtworks.com/de/radar/languages-and-frameworks/ionic-framework>, Zeitpunkt des Zugriffs: 10.02.18, 12:40

<https://opensource.org/licenses/MIT>, Zeitpunkt des Zugriffs: 13.02.18, 16:19

<https://ionicframework.com/docs/intro/concepts/>, Zeitpunkt des Zugriffs: 13.02.18, 16:30

<https://tldrlegal.com/license/mit-license>, Zeitpunkt des Zugriffs: 13.02.18, 16:40

<https://ionicframework.com/pro/pricing>, Zeitpunkt des Zugriffs: 13.02.18, 16:45

<https://ionicframework.com/about>, Zeitpunkt des Zugriffs: 13.02.18, 17:00

<https://developers.google.com/maps/documentation/javascript/reference?hl=de>, Zeitpunkt des Zugriffs: 15.02.18, 18:12

https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/, Zeitpunkt des Zugriffs: 18.02.2018, 12:00

https://de.wikipedia.org/wiki/Representational\_State\_Transfer, Zeitpunkt des Zugriffs: 18.02.2018, 12:00

<https://developer.deutschebahn.com/store/apis/info?name=Bahnhofsfotos&version=v1&provider=DBOpenData&>, Zeitpunkt des Zugriffs: 21.02.2018, 14:04

<https://developer.deutschebahn.com/store/apis/info?name=FaStaStation_Facilities_Status&version=v2&provider=DBOpenData&>, Zeitpunkt des Zugriffs: 21.02.2018, 16:00

<https://developer.deutschebahn.com/store/apis/info?name=BahnPark&version=v1&provider=DBOpenData&#/>, Zeitpunkt des Zugriffs: 21.02.2018, 16:52

<https://developer.deutschebahn.com/store/apis/info?name=StaDa-Station_Data&version=v2&provider=DBOpenData&#/>, Zeitpunkt des Zugriffs: 21.02.18, 17:17

<https://developer.deutschebahn.com/store/apis/info?name=Reisezentren&version=v1&provider=DBOpenData&#/>, Zeitpunkt des Zugriffs: 21.02.18, 17:27

1. https://www.geodaten.niedersachsen.de/startseite/gdini/open\_data\_portale/open-data-136000.html [↑](#footnote-ref-1)
2. http://data.deutschebahn.com/ [↑](#footnote-ref-2)
3. https://de.wikipedia.org/wiki/Ionic\_(Framework)#cite\_note-2 [↑](#footnote-ref-3)
4. https://de.ryte.com/wiki/Native\_App [↑](#footnote-ref-4)
5. https://www.thoughtworks.com/de/radar/languages-and-frameworks/ionic-framework [↑](#footnote-ref-5)
6. https://ionicframework.com/about [↑](#footnote-ref-6)
7. https://ionicframework.com/docs/v1/overview/ [↑](#footnote-ref-7)
8. Chris Griffith: Mobile App Development with Ionic 2, S.5 [↑](#footnote-ref-8)
9. Chris Griffith: Mobile App Development with Ionic 2, S.6 [↑](#footnote-ref-9)
10. https://de.wikipedia.org/wiki/Ionic\_(Framework)#cite\_note-2 [↑](#footnote-ref-10)
11. https://opensource.org/licenses/MIT [↑](#footnote-ref-11)
12. https://ionicframework.com/docs/intro/concepts/ [↑](#footnote-ref-12)
13. https://tldrlegal.com/license/mit-license [↑](#footnote-ref-13)
14. https://ionicframework.com/pro/pricing [↑](#footnote-ref-14)
15. https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/ [↑](#footnote-ref-15)
16. https://de.wikipedia.org/wiki/Representational\_State\_Transfer [↑](#footnote-ref-16)
17. https://www.cloudcomputing-insider.de/was-ist-eine-rest-api-a-611116/ [↑](#footnote-ref-17)
18. siehe auch: https://twitter.com/dbopendata?lang=de [↑](#footnote-ref-18)
19. http://data.deutschebahn.com/ [↑](#footnote-ref-19)
20. https://developer.deutschebahn.com/store/site/pages/sign-up.jag [↑](#footnote-ref-20)
21. s. http://www.railway-stations.org/ [↑](#footnote-ref-21)
22. s. https://play.google.com/store/apps/details?id=de.bahnhoefe.deutschlands.bahnhofsfotos [↑](#footnote-ref-22)
23. Für weitere Informationen siehe https://de.wikipedia.org/wiki/3-S-Zentrale [↑](#footnote-ref-23)
24. https://developers.google.com/maps/documentation/javascript/reference?hl=de [↑](#footnote-ref-24)