ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Entwicklung einer Web Scraping Plattform für mobile Anwendungen

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Manuel Schmid
manuel-1.schmid@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Michael Stach

2020

Fassung 27. Juli 2020

Satz: PDF-LaTeX 2$_\varepsilon$

Abstract

Smartphones sind für viele Menschen zu alltäglichen Begleitern geworden. Damit einher geht auch eine breite Auswahl an Apps für diese. Für Forschungszwecke, Erstellung von Statistiken, Archivierung oder ähnliche Szenarien ist es nützlich, eine einfache Möglichkeit zu haben, um möglichst systematisch und automatisiert Metadaten über solche Apps zu erhalten. Da es für die großen App Stores keine oder nur unzulängliche öffentliche API's oder anderweitige Schnittstellen zur Datenabfrage gibt, ist es notwendig, auf Web Scraping zurückzugreifen.

In dieser Arbeit wird ein solcher Web Scraper für den Google Play Store und den Apple App Store entwickelt, welcher basierend auf den Webseiten der Stores die relevanten Daten extrahiert. Dabei wird zunächst darauf eingegangen, wie die Daten übertragen und repräsentiert werden und anschließend eine Implementierung entwickelt, die es möglichst einfach machen soll, neue Stores zu ergänzen und Änderungen seitens der Stores umzusetzen. Zuletzt wird die Funktionalität dieser Scrapers in Form einer REST-API zur Verfügung gestellt, um einen Ressourcen-zentrierten Zugriff zu erlauben und Programmiersprachen-Unabhängigkeit zu erlangen.

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Verfassung des Arbeit und des gesamten Studiums unterstüzt haben.

Ein großer Dank geht an Michael Stach, der diese Ausarbeitung betreut hat. Für seine hilfreichen Ratschläge und seine Unterstützung möchte ich mich herzlich bedanken.

Ebenso möchte ich mich beim Herrn Prof. Dr. Manfred Reichert für die Begutachtung meiner Arbeit bedanken.

Zuletzt möchte ich mich bei meiner Familie bedanken, vor allem bei meinen Eltern Andreas und Anja, die mich während des Studiums immer unterstützt haben.

Inhaltsverzeichnis

1

Einleitung

Die Zahl der Smartphone-Nutzer und der damit verbundenen Anwendungen wächst kontinuierlich, für die beiden dominierenden Betriebssysteme Android und iOS befanden sich Ende 2017 weltweit ca. 3,3 Milliarden Geräte im Gebrauch [32], in Deutschland gab es Ende 2019 ca. 57,7 Millionen Smartphone-Nutzer [33]. Im 2. Quartal 2019 gab es im Google Play Store ca. 2,4 Millionen Apps und im Apple App Store ca. 1,9 Millionen Apps [21].

Diese Zahlen zeigen, dass das Smartphone zu einem alltäglichen Begleiter für viele Menschen geworden ist und eine Vielzahl Apps für nahezu jede Tätigkeit, ob Spiele, Fitness, Kommunikation, etc. existiert. Durch die große Verbreitung lassen Größen wie Download-Zahlen oder Nutzer-Wertungen auch Rückschlüsse auf das Nutzungsverhalten zu. Sucht man im Play Store z.B. nach dem Begriff *Angst*, so erscheinen diverse Apps, die helfen sollen Angst- und Panikstörungen zu bewältigen. Für statistische Untersuchungen solcher Apps ist es nötig, möglichst automatisiert Daten über diese Apps zu erhalten, mit welchen sich dann Aussagen z.B. über die Verbreitung des Problems oder über die Nützlichkeit solcher Apps treffen lassen.

1.1 Problemstellung

Das Kern-Problem der Daten-Extraktion aus den Stores ist, dass es teils keine offiziellen öffentlichen API's für Datenabfragen gibt oder diese veraltet und/oder unzulänglich sind, daher muss auf Web Scraping zurückgegriffen werden.

Beim Web Scraping extrahiert man Daten direkt aus den Webseiten der Stores oder reproduziert HTTP-Requests, die diese Stores machen. Daraus folgt natürlich eine starke Abhängigkeit gegenüber dem Datenformat und den Request-Endpunkten. Allgemein bestehen beim Web Scraping unter Anderem die folgenden zentralen Herausforderungen:

- Robustheit gegenüber Änderungen seitens der Stores
- Erweiterbarkeit für neue Stores
- *Pagination*, d.h. die Extraktion von Daten, die über mehrere *Pages*, also *Seiten* verstreut sind
- Verhinderung von IP-Blockaden gei größeren Anzahlen von Requests
- Einfache Bedienbarkeit

Darüberhinaus soll der Scraper auch möglichst vollständig die jeweiligen Funktionen der Stores abbilden. Dies umfasst z.B. die Möglichkeit, Apps nach einem Begriff zu suchen (evtl. mit Filtern), Apps aus bestimmten Kategorien aufzulisten, detaillierte Informationen zu einer App auszugeben, etc.

1.2 Zielsetzung

Der entwickelte Scraper soll die Stores der beiden größten Smartphone-Betriebssysteme, Android (Google Play Store) und iOS (Apple App Store) scrapen können. Dabei soll er so konstruiert werden, dass er möglichst sinnvolle Lösungen für die in der Problemstellung genannten Herausforderungen bereitstellt.

Der Großteil des Scrapers (der *Scraper-Kern*) soll dabei als eine Programm-Bibliothek fungieren, die direkt in anderem Code genutzt werden kann. Für diese Blibliothek lassen sich dann andere Frontends zur einfachen Nutzung entwicklen. Spezifisch soll ein REST-Schnittstelle entwickelt werden, die einen einfachen, ressourcen-zentrierten Zugriff auf die Kern-Funktionen des Scrapers ermöglicht.

Der Scraper soll dabei möglichst präzise die Funktionalitäten der Stores abbilden, diese umfassen zur Zeit des Schreibens grob zusammengefasst die folgenden zentralen Aspekte:

- Abfrage von App-Details
- Abfrage von bestimmten Gruppen von Apps, darunter fallen z.B. Apps eines Entwicklers, ähnliche Apps zu einer anderen App oder bestimmte App-Sammlungen (top kostenlose Apps, neue Apps, etc.)
- Suche nach Apps und Such-Vervollständigung
- Abfrage von Nutzer-Wertungen

Eine exakte Auflistung von diesen *Scraper-Methoden* soll über eine entsprechende Dokumentation gegeben werden.

1.3 Struktur der Arbeit

Zu Beginn werden in Kapitel 2 die Anforderungen an den Scraper genauer ausgeführt, unterteilt in funktionale und nicht-funktionale Anforderungen, beides jeweils für den eigentlichen Scraper-Kern als auch die REST-Schnittstelle. In Kapitel 3 werden dann der Google Play Store und der Apple App Store hinsichtlich verfügbarer Daten und dem Zugriff auf diese Daten untersucht. Basierend darauf wird dann für beide Stores erarbeitet, welche *Scraper-Methoden* jeweils für die Stores angeboten werden. Eine *Scraper-Methode* ist dabei eine Schnittstelle mit Eingabe-Parametern und einem definierten Format für die gelieferten Daten. In Kapitel 4 wird das Konzept und der Entwurf des Scrapers genauer erörtert. In Kapitel 5 wird dann auf die Implementierung eingegangen und gezeigt, wie die größten Herausforderungen gelöst wurden. In Kapitel 6 wird die entstandene Anwendung hinsichtlich der Erfüllung der Anforderungen evaluiert. Kapitel 7 betrachtet vergleichbare Anwendungen und geht grob auf die signifikanten Unterschiede ein. Kapitel 8 fasst dann nochmals die Ergebnisse kurz zusammen und gibt einen Ausblick darauf, wie der Scraper noch erweitert werden könnte und wie er in Zukunft eingesetzt werden kann.

2

Allgemeine Anforderungen

Die Anforderungsanalyse ist ein klassischer und zentraler Aspekt der Software-Entwicklung. In dieser Phase wird möglichst präzise definiert, welche Anforderungen an die Software gestellt werden. Die Anforderungen stellen einen Referenzpunkt für die weitere Entwicklung dar, an dem man sich orientieren kann; man erhält ein klareres Bild, was genau die Software können soll. Gerade bei größeren Systemen kann eine unzulängliche Anforderungsanalyse später für Probleme sorgen, z.B. wenn diese nicht präzise definiert werden und später dann nicht so umgesetzt werden wie erwartet oder wenn Anforderungen komplett fehlen.

Klassisch teilt man Software-Anforderungen in zwei Kategorien auf: Funktionale und Nicht-funktionale Anforderungen. Funktionale Anforderungen beschreiben die Daten, Funktionen und das Verhalten der Software, wogegen Nicht-funktionale Anforderungen eher als *Attribute* der Software aufzufassen sind, diese umfassen z.B. Anforderungen an Dokumentation, Erweiterbarkeit, Robustheit, Tests, etc.

Da die entwickelte Software den eigentlichen Scraper-Kern und eine REST-Schnittstelle umfasst, werden hier Anforderungen für beide präsentiert. Da die angebotenen Funktionen stark von den Stores abhängen, werden hier nur allgemeine, store-unabhängige Anforderungen grob umrissen. Die detaillierten Funktionen sind abhängig von den einzelnen Stores und werden in Kapitel 3 genauer erläutert.

2.1 Funktionale Anforderungen

2.1.1 Scraper-Kern

ID	FK01
Titel	Store-Modul
Beschreibung	Die Scraper-Kern-Bibliothek muss ein Modul für jeden Store bereitstellen, das die Scraper-Methoden für diesen Store zu Verfügung stellt.

ID	FK02
Titel	Scraper-Methoden
Beschreibung	Ein Store-Modul enthält Scraper-Methoden. Dies sind Funktionen, die einen bestimmten Typ Daten aus dem Store extrahieren. Für jede Scraper-Methode sind festzulegen: • **Eingabeparameter**: Erforderliche und optionale Parameter, um eine bestimmte Ressource von einem Store abzufragen. Dies können z.B. IDs, Suchbegriffe, etc. sein. • **Datenformat**: Das Format und die Struktur der zurückgegeben Daten (z.B. JSON). • **Potentielle Fehler**: Eventuell Fehler, die bei der Datenabfrage auftreten können.

ID	FK03
Titel	Zu unterstützende Stores
Beschreibung	Es sind mindestens die folgenden beiden Stores zu unterstützen: Google Play Store, Apple App Store.

ID	FK04
Titel	Store-Kommunikation/Request-Backend
Beschreibung	Die Netzwerk-Kommunikation mit den Stores geschieht über HTTP. Dazu soll ein separates, austauschbares Netzwerk-Modul als Request-Backend fungieren, welches von einem Store-Modul zur Kommunikation mit einem Store genutzt wird.

ID	FK05
Titel	Store-Zugriff gewährleisten
Beschreibung	Wenn möglich, so sollen Maßnahmen ergriffen werden, die sicherstellen, dass der Scraper nicht seitens eines Stores blockiert wird (z.B. Eingabe eines CAPTCHA's, IP-Blockaden, etc.)

2.1.2 REST-API

ID	FR01
Titel	REST-Server
Beschreibung	Die REST-API muss über einen Webserver zur Vefügung gestellt werden. Zur genaueren Konfiguration des Server werden hier keine Anforderungen gestellt.

ID	FR02
Titel	Ressourcen-Anfragen
Beschreibung	Sämtliche Anfragen werden über HTTP-*GET*-Requests gestellt. Für jede Scraper-Methode muss dabei folgendes festgelegt werden: • **Anfrage-Pfad**: Kodiert den angefragten Store und die angefragte Ressource. • **Anfrage-Parameter**: Die URL-Parameter kodieren Eingabe-Parameter für die im Pfad kodierte Scraper-Methode. Sie müssen auf Korrektheit validiert werden (soweit möglich). • **geliefertes Datenformat**: Eine korrekte Antwort wird immer im JSON-Format geliefert. Der Aufbau ist zu dokumentieren. • **mögliche Fehlerfälle**: Ein Fehlerfall wird durch einen HTTP-Status-Code aus den 400/500er-Blöcken signalisiert. Genauere Details (Fehlernachricht, Bedeutung, etc.) sind zu dokumentieren.

ID	FR03
Titel	Scraper-Request-Backend
Beschreibung	Das für den Scraper-Kern gewählte Request-Backend muss die folgenden zwei Funktionen unterstützen: • **Request-Caching**: Anfragen an einen Store, die vom Scraper-Kern generiert werden müssen gecached werden, um eine Wiederholung derselben Requests zu vermeiden. Cache-Algorithmus und Speicher-Backend sind frei wählbar, es muss allerdings eine *TTL (time to live)* einstellbar sein, um eine zu lange Verweildauer zu vermeiden. • **Request Throttling**: Es muss möglich sein, eine Obergrenze für die Request-Senderate (Requests pro Sekunde) für einen Store einzustellen, um zu Blockaden seitens der Stores zu vermeiden.

2.2 Nicht-funktionale Anforderungen

2.2.1 Scraper-Kern

ID	QK01
Titel	Robustheit und Wartbarkeit gegenüber Store-Änderungen
Beschreibung	Der Scraper-Kern muss (soweit möglich) mit Store-Änderungen umgehen können und einfach anpassbar sein, um auf solche Änderungen reagieren zu können.

ID	QK02
Titel	Dokumentation
Beschreibung	Die Funktionsweise, der Code und die vom Scraper zur Verfügung gestellten Scraper-Methoden müssen ausführlich dokumentiert werden.

2.2.2 REST-API

ID	QR01
Titel	Auslieferung
Beschreibung	Um die Installation von Tools, Programmiersprachen, etc. zu vermeiden muss ein Docker Image für den REST-Webserver zur Verfügung gestellt werden.

ID	QR02
Titel	Dokumentation
Beschreibung	Wie auch beim Scraper-Kern muss die REST-API dokumentiert werden.

3

Untersuchung der Stores

Dieses Kapitel widmet sich der Untersuchung der beiden unterstützten Stores, dem Google Play Store und dem Apple App Store. Ziel ist es, die Scraper-Methoden für die beiden Stores wie in Anforderung FK02 definiert, festzulegen. Zunächst wird die generelle Vorgehensweise näher erläutert und anschließend auf die einzelnen Stores eingegangen.

3.1 Vorgehensweise und Werkzeuge

Im Wesentlichen müssen die folgenden drei Schritte durchlaufen werden, um die Scraper-Methoden für einen Store festzulegen:

1. Identfikation der vom Store zur Verfügung gestellten Funktionen (über Webseite, native Anwendungen, etc.)
2. Für jede Funktion des Stores: Ermitteln der Daten, die von die von dieser Funktion geliefert werden und wie diese Daten zu erhalten sind. Z.B. kann dies der Text eines HTML-Tags sein oder es kann erforderlich sein, einen HTTP-Request zu senden und die Antwort erhält die gewünschten Daten.
3. Festlegung der in Anforderung FK02 geforderten Eigenschaften für die Funktion um diese dann als Scraper-Methode nutzen zu können.

Für Schritt 1 muss zunächst überlegt werden, wie man überhaupt auf die Stores zugreift, da dies die verfügbaren Funktionen und gelieferten Daten beeinflussen kann. Dazu gibt es im Wesentlichen die folgenden zwei Möglichkeiten:

- Zugriff über die Webseiten der Stores. Der Vorteil ist, dass dies vergleichweise einfach und außerdem plattform-unabhängig ist. Es werden z.B. keine Annahmen über ein bestimmtes Gerät gemacht und es sind keine Nutzer-Accounts erforderlich, die Einfluss nehmen könnten (z.B. Nutzer-Präferenzen). Außerdem lassen sich Webseiten recht einfach mittels Browser-Tools untersuchen.

- Zugriff über native Anwendungen. Dies umfasst z.B. die Apps für die entsprechenden Smartphone-Betriebssysteme oder im Fall des Apple App Stores die iTunes-Anwendung. Das größte Problem hier ist, dass man entsprechende Geräte besitzen muss (z.B. Android/iOS-Gerät) und es erheblich schwerer ist, die technische Umsetzung und Netzwerk-Kommunikation nachzuvollziehen. Prinzipiell wären hier Packet Sniffer (wie Wireshark) möglich, aber das Problem hierbei ist, dass diese Anwendungen ihre Daten verschlüsseln und der Packet Sniffer die Inhalte nicht entschlüsseln kann. Bei Browsern kann man sich die TLS-Schlüssel ausgeben lassen, aber nicht bei nativen Anwendungen.

Aufgrund der Schwierigkeiten mit den nativen Anwendungen werden die Stores hier basierend auf ihren Webseiten untersucht. Beim Apple App Store gibt es allerdings noch einige weitere Ressourcen (Partner-Programme), die eine Datenabfrage möglich machen, diese werden hier auch betrachtet.

Zur Untersuchung der Webseiten werden Browser-Debug-Tools eingesetzt. Damit lassen sich Webseiten recht ausführlich inspizieren und die Kommunikation mit dem Server nachvollziehen. Hier wird Mozilla Firefox eingesetzt, Abbildung 3.1 zeigt die Entwickler-Tools von Firefox. Zu sehen sind mehrere Reiter mit den folgenden Funktionen:

- Der **Inspektor** zeigt das DOM in einer Baumstruktur an und entält Stil-Informationen, Attribute, Text, etc. für jeden Knoten im Baum.

- Die **Konsole** ist einfach die JavaScript-Konsole für JavaScript-Output, man kann aber auch direkt JavaScript ausführen (als Read-Eval-Print-Loop, erlaubt z.B. Zugriff auf globale Variablen).

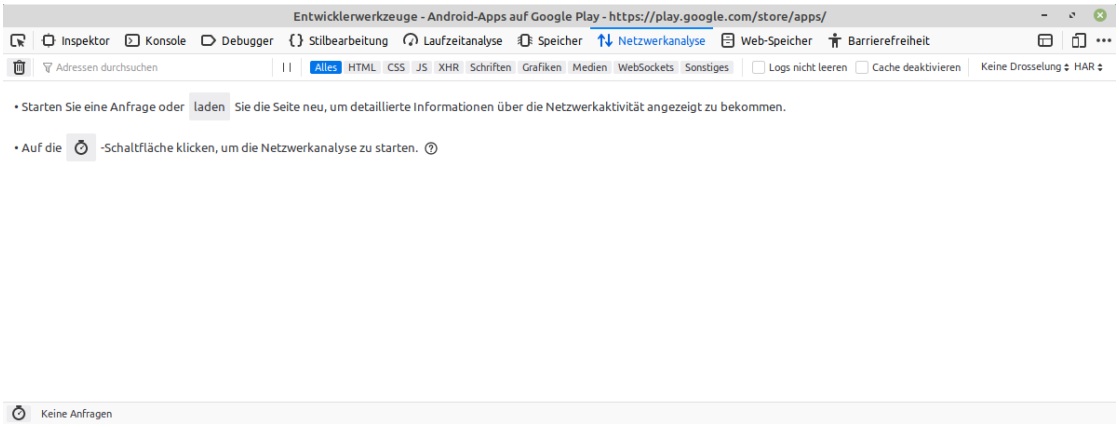


Abbildung 3.1: Debug-Tools von Mozilla Firefox (Version 68.8.0esr)

- Der **Debugger** erlaubt das Debugging von JavaScript mit den üblichen Funktionen (Break Points, Watches, etc.) und man kann den Quellcode der Skripte ansehen.
- Die **Stilbearbeitung** erlaubt das ansehen und manipulieren aller verwendeten Stylesheets.
- **Laufzeitanalyse** und **Speicher** erlauben das Überwachen vom Ressourcen-Verbrauch (Laufzeit-Verhalten und Speicher-Verhalten).
- Die **Netzwerkanalyse** nimmt alle gesendeten HTTP-Requests und Antworten auf.
- Der **Web-Speicher** liefert Informationen über Cache, Cookies und andere Formen von persistierten Daten.

Ein weiterer wichtiger Aspekt ist das *view-source*-Pseudoprotokoll. Hängt man vor eine URL in der Addressleiste das Protokoll-Präfix `view-source:`, so wird der Seitenquelltext angezeigt. Im Gegensatz zum Inspektor wird hier wirklich nur der Quelltext wie er empfangen wurde angezeigt. Der Inspektor liefert immer die aktuelle Sicht und da das DOM durch Skripte manipuliert wird entspricht dies nicht dem Originaldokument. Dies ist wichtig, da diese Manipulationen eine volle JavaScript-Ausführungsumgebung (z.B. einen automatisierten Browser) erfordern.

3.2 Google Play Store

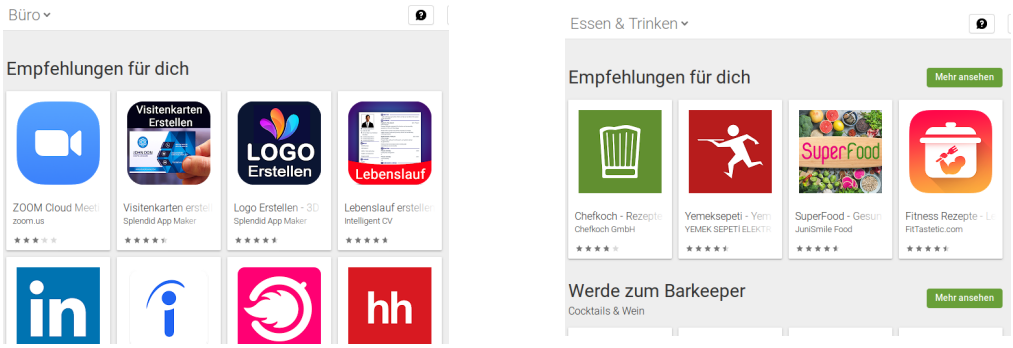
In den nächsten drei Abschnitten werden entsprechend der Vorgabe die Funktionen des Play Stores identifiziert, dann die Datenquellen und -formate erläutert und zu guter Letzt die Scraper-Methoden definiert.

3.2.1 Aufbau und Funktionen des Stores

Abbildung 3.2: Ausschnitt der Hauptseite des Google Play Stores) [15]

Der Google Play Store ist generell unter der URL `https://play.google.com/store` zu erreichen. Für Apps spezifisch (keine Filme, Serien, etc.) ist der zu verwendende Pfad `/store/apps`. Abbildung 3.2 zeigt einen Ausschnitt der Hauptseite des Play Stores, wobei die folgenden Elemente hervorgehoben sind:

- **App-Card**: Ein GUI-Element, welches Icon, Titel, Entwickler und Durchschnittswertung einer App anzeigt. Kann angeklickt werden und führt dann zur Detailseite einer App.
- **App-Cluster**: Ein Cluster ist eine Sammlung von Apps nach bestimmten Kriterien, z.B. hier die „beliebtesten“ Apps. Man kann sich den kompletten Cluster anschauen, indem man auf „Mehr ansehen“ klickt. Hier wird nur eine **Cluster-Vorschau** mit ein paar Apps (abhängig von der Fenstergröße) angezeigt.
- **Kategorien**: Ein Dropdown-Menü, welches alle App-Kategorien auflistet. Klickt man auf eine der Kategorien gelangt man zu der Übersichtsseite für die Kategorie.
- **Sammlungen**: Dies sind Sammlungen von mehreren Clustern unter einem Oberbegriff wie „Top-Apps“ oder „neue Apps“. Es scheinen nur diese beiden Sammlungen zu existieren.



(a) Cluster-Übersicht [15]

(b) Cluster-Sammlung [15]

Abbildung 3.3: Unterschied Cluster-Übersicht/Cluster-Sammlung

Die Hauptseite des Play Stores kann man als eine *Cluster-Sammlung* bezeichnen: Sie listet Vorschauen für mehrere Cluster auf, die Apps unter einem bestimmten Kriterium gruppieren. Insgesamt gibt es im Wesentlichen die folgenden vier zentralen Typen von Seiten:

- **Cluster-Sammlungen**: Eine Seite, die Vorschauen zu verschiedenen Clustern auflistet. Z.B. listet die in Abbildung 3.3b dargestellte Seite die Cluster

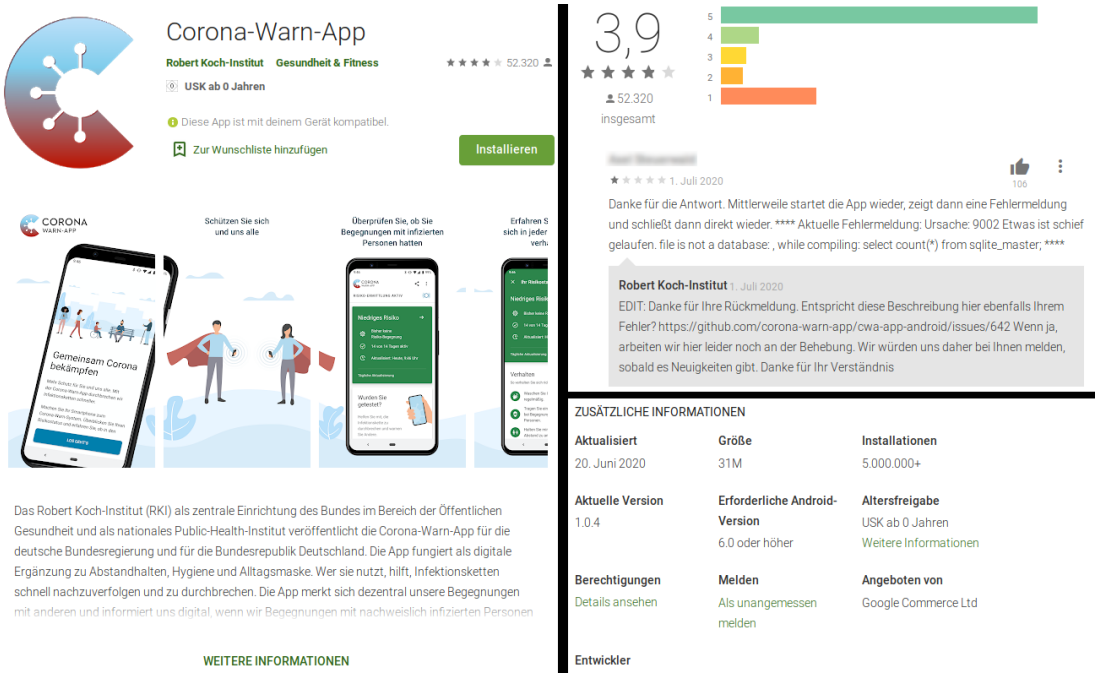


Abbildung 3.4: App-Detail-Seite im Play Store [15]

„Empfehlungen für dich“ und „Werde zum Barkeeper“ auf, die restlichen Cluster sind auf dem Bild nicht zu sehen. Diese Cluster-Sammlungen finden sich an verschiedenen Stellen, z.B. führen die „Top-Charts“ zu einer Cluster-Sammlung mit „Top-Apps“, „Bestseller-Apps“, etc.

- **Cluster-Übersicht**: Diese Seiten bekommt man angezeigt, wenn man sich alle Apps eines Clusters anzeigen lässt (z.B. über „Mehr ansehen“). Diese Seite beinhaltet nur App-Cards und evtl. den Namen des Clusters, siehe dazu z.B. Abblidung 3.3a, hier werden nur App-Cards des Clusters „Empfehlungen für dich“ angezeigt.
- **App-Detail-Seite**: Eine Seite, die Details zu einer App anzeigt, also Beschreibung, Titel, Nutzerwertungen, etc. (siehe zum Beispiel Abbildung 3.4).
- **Themenseite**: Der Play Store hat sog. „Redaktionsempfehlungen“ zu einem bestimmten Thema. Eine Themenseite besteht aus mehreren Abschnitten, die jeweils eine empfohlene App beinhalten. Die enthaltenen Informationen entsprechenden den Informationen aus dem Kopfbereich einer App-Detail-

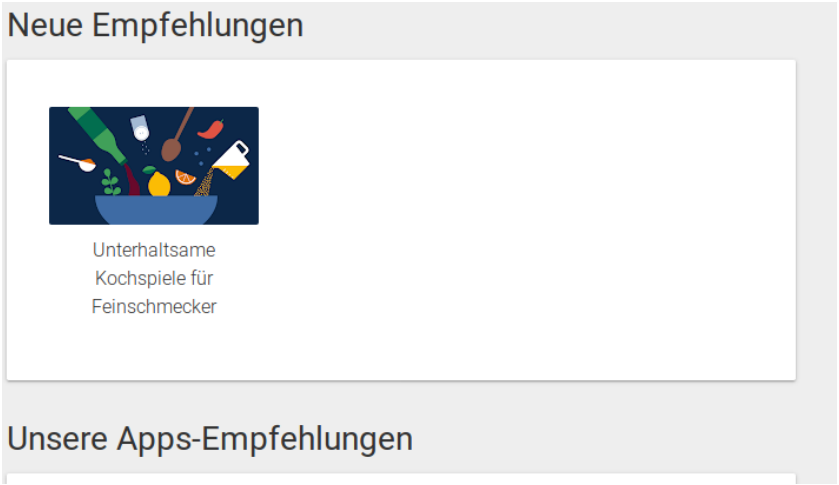


Abbildung 3.5: Redaktionsempfehlungen im Play Store [15]

Seite. Abbildung 3.5 zeigt eine Liste mit **Topic-Card**s, die zu Themenseiten führen, hier nur das Thema „Unterhaltsame Kochspiele für Feinschmecker“.

Im Nachfolgenden wird auf konkrete Funktionen aus Nutzersicht eingegangen. Viele von diesen Funktionen laufen darauf hinaus, dass ein bestimmter Cluster von Apps angezeigt wird. Der einzige Unterschied besteht darin, welche Apps angezeigt werden, der Aufbau ist aber derselbe.

ID	PL01
Titel	Kategorien auflisten
Beschreibung	Der Scraper muss die Liste der Play-Store-Kategorien auflisten können, so wie im Drop-Down-Menü in 3.2 gezeigt.

ID	PL02
Titel	Cluster auflisten
Beschreibung	Der Scraper muss die App-Cards aus einer Cluster-Übersicht und aus den Cluster-Vorschauen einer Cluster-Sammlung extrahieren können.

ID	PL03
Titel	Apps aus Standard-App-Sammlung auflisten
Beschreibung	In Abbildung 3.2 sind die Top/Neu-Sammlungen zu sehen. Die Anzahl der gezeigten Cluster unterscheidet sich abhängig von der gewählten Kategorie. Welche Cluster gezeigt werden hängt von dieser Anzahl ab (werden z.B. zwei Cluster in einer „Top“-Sammlung gezeigt, so sind dies immer „Top-Apps“ und „Bestseller-Apps“). Der Scraper soll Apps aus einem solcher Cluster extrahieren können, ohne dass die Cluster-URL angegeben werden muss.

ID	PL04
Titel	Apps eines Entwicklers auflisten
Beschreibung	Klickt man auf den Namen eines Entwicklers, so bekommt man eine Cluster-Übersicht mit allen Apps des Entwicklers angezeigt, die der Scraper mittels PL02 extrahieren können muss.

ID	PL05
Titel	Ähnliche Apps auflisten
Beschreibung	Auf der App-Detail-Seite zu einer App ist eine Cluster-Vorschau zu finden, die ähnliche Apps auflistet. Der Scraper muss mittels PL02 die App-Cards aus dem zugehörigen Cluster extrahieren können.

ID	PL06
Titel	App-Details ausgeben
Beschreibung	Der Scraper muss Daten aus der App-Detail-Seite extrahieren können (z.B. Screenshots, Beschreibung, etc.)

ID	PL07
Titel	Rezensionen zu einer App ausgeben
Beschreibung	Die App-Detail-Seite beinhaltet eine Vorschau mit drei Nutzer-Rezensionen. Durch Klicken auf „Alle Bewertungen lesen“ kann man alle Bewertungen mit verschiedenen Filtern anzeigen. Der Scraper muss die Rezensionen auflisten können.

ID	PL08
Titel	Suche nach Apps
Beschreibung	Sucht man im Play Store nach einem Begriff wird eine Cluster-Übersicht mit den Ergebnissen angezeigt. Diese müssen mittels PL02 extrahiert werden können.

ID	PL09
Titel	Suchvorschläge
Beschreibung	Tippt man einen unvollständigen Suchbegriff in die Suchleiste ein, so kommen bis zu fünf Vorschläge zur Vervollständigung. Der Scraper muss diese extrahieren können.

ID	PL10
Titel	Redaktionsempfehlungen
Beschreibung	Der Scraper muss Apps aus einer bestimmten Themenseite auflisten können und die Themenseiten aus den Redaktionsempfehlungen auflisten können.

3.2.2 Datenabfrage und Datenformat

In diesem Abschnitt wird detailliert darauf eingegangen, wie die Webseite des Play Stores die Daten erhält und wie diese strukturiert sind. Zu unterscheiden ist zwischen *statischen* und *dynamischen* Daten.

Mit *statischen* Daten sind alle Daten gemeint, die direkt im empfangenen HTML-Dokument bei Aufruf einer Play-Store-Seite enthalten sind. Diese Daten können direkt extrahiert werden.

Im Kontrast dazu sind *dynamische* Daten sämtliche Daten, die in irgendeiner Form durch Skripte erzeugt werden und die das DOM manipulieren, um sie dem Nutzer zu präsentieren. Um diese Daten zu erhalten benötigt man entweder einer Browser-Ausführungsumgebung, um dasselbe Verhalten wie in einem Webbrowser zu garantieren oder man muss die Erzeugung der Daten nachstellen.

Viele Daten aus dem Play Store bestehen aus einer Liste von einzelnen Datensätzen desselben Typs. Dies können z.B. App-Cards aus einer Cluster-Übersicht sein. Hier liegt ein Teil der Daten meistens schon statisch im HTML-Dokument vor. Werden durch eine Aktion des Nutzers weitere Datensätze benötigt (z.B. durch Scrollen in der Cluster-Übersicht), so werden durch ein Skript die nächsten Datensätze nachgeladen. Dies involviert meistens einen Request an einen Play-Store-Endpunkt. Man nennt einen Block von Datensätzen auch *Page (Seite)* und den gesamten Prozess des Aufteilens von Daten in solche Blöcke *Pagination*.

```
AF_initDataCallback({key: 'ds:<key>',
                     isError: false,
                     hash: '<hash>',
                     data: <data> OR function() {return <data>;}
                     });
```

Listing 3.1: Daten-Skript

Extraktion statischer Daten

Klassischerweise extrahiert man die statischen Daten aus dem HTML-Dokument, indem man das HTML-Dokument parst und dann z.B. XPath-Ausdrücke verwendet, um an ein bestimmtes Datum zu kommen. Interessanterweise liegen beim Play Store die gesuchten Daten in einer weitaus nützlicheren Form vor.

Das HTML-Dokument, welches vom Play Store empfangen wird beinhaltet einige HTML-Skript-Tags, die nichts weiter tun, als ein JavaScript-Funktion namens `AF_initDataCallback` aufzurufen. Der Code sieht in etwa so aus, wie in Listing 3.1 gezeigt. Diese Funktion wird mit einem JavaScript-Objekt aufgerufen, wobei zwei Eigenschaften dieses Objekts interessant sind:

- Die Eigenschaft `key` hat als Wert einen String, der immer nach dem Schema “ds:<key>” aufgebaut zu sein scheint, wobei *<key>* eine positive Ganzzahl ist. Dieser Schlüssel bestimmt, welche Art von Daten vorliegen.
- Die Eigenschaft `data` enthält entweder ein JSON-Element oder eine Funktion, die nichts anderes macht, als ein JSON-Element zurückzugeben. Dieses JSON-Element beinhaltet die eigentlichen gesuchten Daten.

Es gibt mehrere solche Skript-Tags, wobei je nach Schlüssel unterschiedliche Daten enthalten sind. Es ist schwer zu sagen, was diese Funktion bewirkt, da sämtliches JavaScript des Play Store extrem obfuskiert ist. Auch mit deaktiviertem JavaScript wird das Meiste immer noch korrekt angezeigt ist, weshalb diese Skript-Tags überflüssig wirken. Evtl. werden sie für serverseitiges Rendering (SSR) verwendet.

Tabelle 3.1 listet für die unterschiedlichen Seiten-Typen die relevanten Schlüssel auf und welche Daten diese beinhalten. Anhang A.1 ordnet den Store-Funktionen URLs und Seiten-Typ zu. Basierend auf dem Seiten-Typ wird in Anhang A.4 die Struktur der enthaltenen Daten definiert.

Seiten-Typ	*Schlüssel*	*Daten*
Alle	ds:0	Liste von allen Play-Store-Kategorien
Cluster-Sammlung	ds:3	Initiale Liste von App-Cards für jede Cluster-Vorschau der Sammlung.
Cluster-Übersicht	ds:3	Wie bei einer Cluster-Sammlung aber nur ein Cluster statt mehrere Cluster-Vorschauen.
Themenseite	ds:3	Ähnlich wie bei einer Cluster-Sammlung, Format etwas anders.
App-Detail-Seite	ds:12	App-ID
App-Detail-Seite	ds:8	App-Größe und Versionsinfos
App-Detail-Seite	ds:3	Preis-Informationen
App-Detail-Seite	ds:6	Informationen zur App-Bewertung
App-Detail-Seite	ds:5	Informationen zu Entwickler, Beschreibung, Titel, uvm.
App-Detail-Seite	ds:7	Cluster-Vorschau zu ähnlichen Apps. Nahezu gleich aufgebaut wie ds:3 bei einer Cluster-Sammlung.

Tabelle 3.1: Schlüssel für statische Daten in Abhängigkeit vom Seiten-Typ

Extraktion dynamischer Daten

Wie bereits erwähnt müssen manche Datensätze nachgeladen werden. Dazu gehören z.B. Rezensionen und App-Cards in einer Cluster-Übersicht. Das Nachladen geschieht über einen POST-Request an die URL `https://play.google.com/_/PlayStoreUi/batchexecute`. Dem HTTP-Header `Content-Type` entnimmt man, dass der Request-Körper als `application/x-www-form-urlencoded` kodiert ist, d.h. im selben Format wie auch die URL-Parameter. Eine detaillierte Auflistung der beobachteten und relevanten Parameter, sowohl in der URL als auch dem Request-Körper findet sich in Anhang A.2.

Relevant für den Request ist interessanterweise ausschließlich der Parameter `f.req` im Request-Körper, der als Wert einen Json-String hat, der den Request kodiert. Dieser Json-String enthält verschiedene Werte, es findet sich darin aber in jedem Fall der sog. `rpcids`-Parameter, welcher den Anfrage-Typ bestimmt. Tabelle 3.2 listet abhängig von diesem Parameter die Anfrage-Typen auf und welche weiteren Parameter erforderlich sind, um einen vollständigen Request zu bilden.

rpcids	*Anfrage-Typ*	*weitere erforderliche Parameter*
w3QCWb	Nächste Seite von Cluster-Sammlungen	Ein **Token**, das die angefragte Seite identifiziert und die **Anzahl** an angefragten Items.
qnKhOb	Nächste Seite von App-Cards	Ein **Token**, das die angefragte Seite identifiziert und die **Anzahl** an angefragten Items.
xdSrCf	Berechtigungen einer App	Die **App ID**.
UsvDTd	Nächste Seite von Rezensionen	Ein **Token**, die **Anzahl** an angefragten Items und **Filter-** und **Sortieroptionen**.
UsvDTd	Bearbeitungs-Historie einer Rezension	Die **App ID** und die **Review ID**.

Tabelle 3.2: Typen von batchexecute-Requests.

```
)]}'

[["wrb.fr","<rpcids>", "<JSON>", ...],
 ["di", ...],
 ["af.httprm", ...]
]
```

Listing 3.2: Struktur einer batchexecute-Antwort

Eine detaillierte Beschreibung des Json-Strings, der den Request kodiert und eine genauere Beschreibung der Parameter findet sich in Anhang A.2.

Die Antwort auf einen batchexecute-Request sieht wie folgt aus:

Zunächst kommen immer die Zeichen `)]}'`, gefolgt von einer Leerzeile. Wieso dies der Fall ist, ist unklar. Danach kommt ein Json-Array, welches wiederum mehrere Json-Arrays enthält. Das erste Json-Array enthält die eigentlichen Daten. Index 1 dieses Arrays wiederholt den Parameter `rpcids`, Index 2 enthält einen einen Json-String, der die eigentlichen Antwort-Daten enthält. Das Format hängt wieder vom Typ der angefragten Daten ab und ist in Anhang A.4 genauer erläutert.

Werden Daten mit einem solchen Request nachgeladen, so ist das Antwortformat großteils dasselbe wie das der initialen Daten im HTML-Dokument. Meistens ist eine Art Token erforderlich um anzugeben, welche Seite geliefert werden soll und die Antworten enthalten das Token für die nächste Seite.

Eine Ausnahme für eine Store-Funktion, die ihre Daten nicht auf diese Art und Weise erhält ist PL09 (Suchvorschläge). Hier wird ein GET-Request an die URL `https://market.android.com/suggest/SuggRequest` gemacht. URL-Parameter und Antwortformat sind in Anhang A.3 detailliert ausgeführt.

3.2.3 Definition der Scraper-Methoden

Im vorigen Abschnitt wurden die Datenquellen und -formate erörtert. Basierend darauf lassen sich nun die Scraper-Methoden definieren, die die Store-Funktionen

im Scraper zur Verfügung stellen. Hier wird nur grob beschrieben, welche Parameter erforderlich sind, welche Daten zurückgeliefert werden und welche Store-Funktion abgedeckt wird. In Anhang C.1 werden Parameter, Datenformat und potenzielle Fehler detailliert ausgeführt.

Es ist anzumerken, dass bei allen Scraper-Methoden die Eingabe-Parameter „Sprachkürzel“ und „Länderkürzel“ möglich sind, mit denen man die Sprache und das Land festlegen kann. Diese wurden in der Zusammenfassung hier weggelassen.

Tabelle 3.4: Scraper-Methoden des Play Stores

Methode	*Funkt.*	*Parameter*	*Daten*
categories	PL01	-	Eine Liste aller Play-Store-Kategorien.
clusterList	PL02	URL-Pfad max. Item-Anzahl	Informationen zur Cluster-Sammlung inklusive einer Liste von extrahierten Cluster-Vorschauen und Themen-IDs.
cluster	PL02	URL-Pfad max. Item-Anzahl	Informationen zum Cluster inklusive einer Liste von extrahierten App-Cards aus dem Cluster.
listApps	PL03	Sammlungs-ID Kategorie-ID max. Item-Anzahl	Wie bei **cluster**, hier sind die Apps aus dieser speziell ausgewählten Sammlung.
developer	PL04	Entwickler-ID max. Item-Anzahl	Wie bei **cluster**, hier sind die Apps die des angegebenen Entwicklers.
similar	PL05	App-ID max. Item-Anzahl	Wie bei **cluster**, hier sind es die ähnlichen Apps zu der angegebenen App.

Tabelle 3.4: Scraper-Methoden des Play Stores

Methode	*Funkt.*	*Parameter*	*Daten*
details	PL06	App-ID	Detaillierte Informationen zu der App.
reviews	PL07	App-ID max. Item-Anzahl Sortieroptionen Filteroptionen	Eine Liste von Rezensionen zu der angegebenen App.
reviewHistory	PL07	App-ID Review-ID	Eine Liste von Rezensionen, die die Bearbeitungs-Historie einer Rezension darstellt.
search	PL08	Suchbegriff Filteroptionen	Wie bei **cluster**, hier sind die Apps die Suchergebnisse.
suggest	PL09	Suchbegriff	Eine Liste von vervollständigten Suchbegriffen.
editorsChoice	PL10	-	Eine Liste von Themen-IDs zu redaktionell empfohlenen Apps.
topic	PL10	Themen-ID	Eine Liste von Informationen zu den Apps für das gegebenen Thema.

3.3 Apple App Store

In den nächsten drei Abschnitten werden entsprechend der Vorgabe die Funktionen des App Stores identifiziert, dann die Datenquellen und -formate erläutert und zu guter Letzt die Scraper-Methoden definiert.

3.3.1 Aufbau und Funktionen des Stores

Abbildung 3.6: Ausschnitt der Hauptseite des Apple App Stores [10])

Der Apple App Store hat keine vollständige Web-Oberfläche und ist eher zur Nutzung über Endgeräte und native Anwendungen gedacht. Die „Hauptseite“ des App Stores (Abbildung 3.6) besteht im Wesentlichen aus einer Übersicht der verschiedenen Genres an Apps und ist unter `https://apps.apple.com/de/genre/ios/id36` zu erreichen. Steuert man dort ein bestimmmtes Genre an, so erhält man eine Liste beliebter Apps zu diesem Genre. Folgt man dem Link zu der App, so landet man auf der *App-Detail-Seite* der App.

Die App-Detail-Seite (s. Abbildung 3.7) enthält wie beim Play Store Informationen über eine App, darunter fallen z.B. Icon, Beschreibung, Changelog, etc.

Neben der Web-Oberfläche gibt es verstreut noch ein paar weitere Ressourcen, die zusätzliche Daten liefern, die die Web-Oberfläche nicht liefert:

- Die **iTunes-Anwendung** ist eine native Anwendung für den App Store. Es gibt ein paar weitere Funktionen, die aber nicht weiter relevant sind, mit Ausnahme der Suchfunktion, die die Web-Oberfläche nicht hat.
- **Affiliate-Ressourcen**: Apple hat ein Programm namens „Performance Partners“, das unter anderem eine öffentliche API umfasst mit der Daten abgefragt

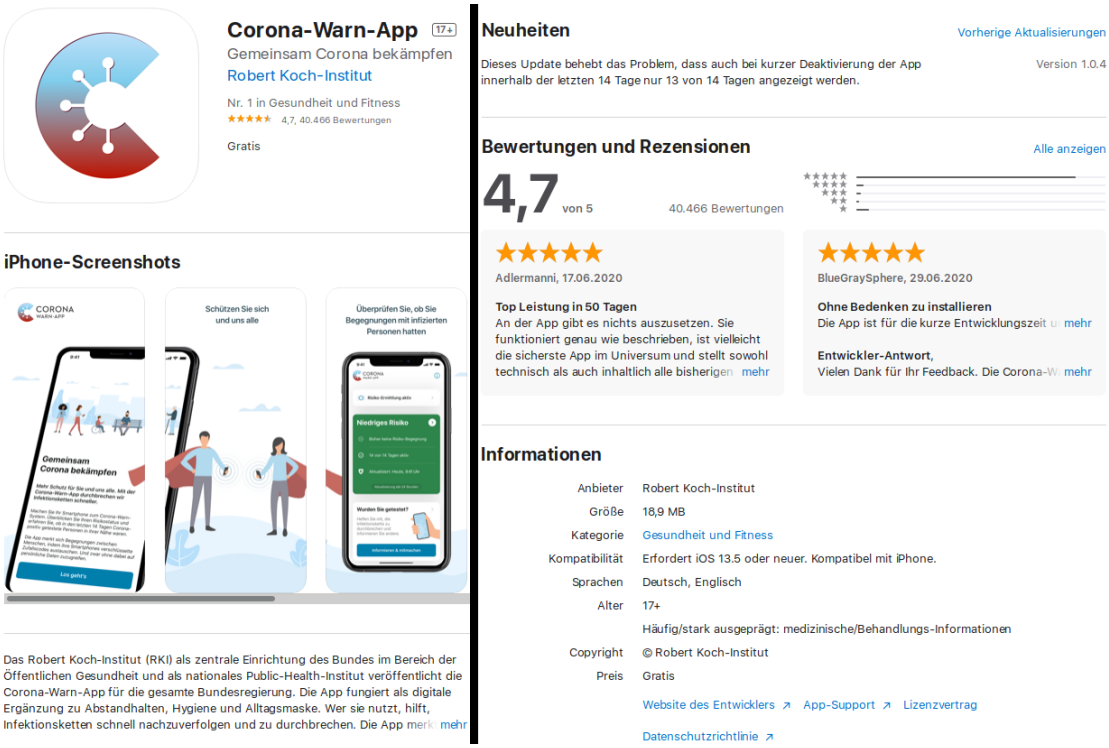


Abbildung 3.7: App-Detail-Seite im App Store [10]

werden können. Die API ermöglicht Suche und Lookup von App Store IDs. Siehe [11] für mehr Details.

- **RSS-Feeds**: Apple bietet RSS-Feeds für verschiedene App-Sammlungen. Dabei findet sich in [12] eine Liste verschiedener RSS-Feeds (z.B. „Top 25 Free Apps“) und [13] stellt einen Generator für RSS-Feed-URLs zur Verfügung. Interessanterweise sind die URLs aus den vorgegeben Listen anders als die generierten URLs und die gelieferten Daten sind auch in einem anderen Format.

Im Nachfolgenden wird auf konkrete Funktionen aus Nutzersicht eingegangen. Dabei werden neben der Web-Oberfläche auch die Funktionen aus den zusätzlich genannten Quellen berücksichtigt.

ID	AS01
Titel	App-Details ausgeben
Beschreibung	Der Scraper muss Details zu einer App ausgeben können.

ID	AS02
Titel	Apps eines Entwicklers auflisten
Beschreibung	Klickt man auf der App-Detail-Seite auf den Namen des Entwicklers, so erhält man eine Übersicht über alle Apps des Entwicklers. Der Scraper muss diese auflisten können.

ID	AS03
Titel	Auflisten von Apps aus bestimmten Sammlungen
Beschreibung	Die RSS-Feeds liefern Apps zu bestimmten Sammlungen (Top-Apps, Neue Apps, etc.). Der Scraper muss die Apps aus diesen Feeds auflisten können.

ID	AS04
Titel	Auflisten von Rezensionen zu einer App
Beschreibung	Auf der App-Detail-Seite finden sich auch Rezensionen zu einer App. Der Scraper muss diese auflisten können.

ID	AS05
Titel	Suche nach Apps
Beschreibung	In der iTunes-Anwendung kann man nach Apps suchen. Der Scraper muss diese Möglichkeit ebenfalls bieten.

ID	AS06
Titel	Auflisten ähnlicher Apps
Beschreibung	Auf einer App-Detail-Seite gibt es einen Link zu einer Liste von ähnlichen Apps zu der betrachteten App. Der Scraper muss diese ebenfalls auflisten können.

ID	AS07
Titel	Suchvervollständigung
Beschreibung	Bei Eingabe eines Suchbegriffs in der iTunes-Anwendung wird dieser vervollständigt. Der Scraper muss diese Funktion ebenfalls bieten.

3.3.2 Datenabfrage und Datenformat

In diesem Abschnitt wird darauf eingegangen, wie man Daten aus dem Apple App Store erhält. Ähnlich wie beim Play Store ist es nicht notwendig, Daten aus dem HTML-Dokument zu extrahieren, auch wenn dies durchaus möglich wäre. Es ist allerdings auch hier so, dass für einige Datensätze *Pagination* erforderlich ist. Dies betrifft insbesondere die Rezensionen zu einer App und ähnliche Apps. Beim Play Store ist es erforderlich, dass zunächst Daten aus den HTML-Skript-Tags extrahiert werden, der Grund dafür ist, dass für die batchexecute-Requests ein Token erforderlich ist,

das man nur so erhalten kann. Beim App Store ist dies nicht der Fall, man kann die Requests an den Server direkt nachbilden und so alle Informationen erhalten, die man benötigt.

Der X-Apple-Store-Front Header und WebObjects

Beobachtet man Requests, die von älteren iTunes-Anwendungen gesendet werden, so stellt man drei Dinge fest:

1. Die Requests sind an Subdomains von `itunes.apple.com` gerichtet, die Pfade folgen dem Schema:

   `/WebObjects/MZ<TYPE>.woa/wa/<FUNC>.`

   Diese Art von Links nennt man *WebObject*-Links, wobei **TYPE** das WebObject identifiziert und **FUNC** die Funktion des WebObjects. Z.B. gibt es die WebObjects *MZStore.woa* und *MZSearchHints.woa*.

2. Die Requests haben einen Header namens `X-Apple-Store-Front`. Dieser Header legt die *Storefront* fest. Es gibt keine exakte, vollständige Dokumentation für diesen Header, aber er spezifiert eine sog. *Store-ID*, die landesabhängig ist. Dies hat Einfluss auf Sprache und Verfügbarkeit der gelieferten Daten. Ein zweiter Wert spezifiert eine Art Plattform-ID. Der Aufbau des Headers folgt dem Schema:

   `X-Apple-Store-Front:<SID>,<PID>`

   Hierbei steht `SID` für die Store ID und `PID` für die Plattform ID. Z.B. ist 143441 die Store ID für die USA. Manchmal sind im Header nocht weitere Daten, z.B. `143441,24 t:native`, hier ist es nicht ganz klar, wofür “t:native” steht. Anhang B.7.4 findet sich eine Liste bekannter Storefront-IDs.

3. Die Sprache wird mittels des Headers `Accept-Language` festgelegt. Der Wert muss Sprach- und Länderkürzel, getrennt durch einen Unterstrich beinhalten, z.B. “de_de”).

Funktion	*WebObject*
App-Suche	`search.itunes.apple.com/WebObjects/MZStore.woa/wa/search`
Suchvervollständigung	`search.itunes.apple.com/WebObjects/MZSearchHints.woa/wa/hints`

Tabelle 3.5: App Store WebObject-Funktionen

Es gibt einige Funktionen, die nur über die Nutzung von WebObject-URLs realisiert werden können. Tabelle 3.5 listet die zwei bekannten WebObject-URLs und deren Funktionalität auf, eine detaillierte Beschreibung ist in Anhang B.5 zu finden. Der Header lässt sich außerdem bei Abfrage von App-Details anwenden, sodass man die HTML-Seite in einem anderen Format erhält als gewöhnlich, in diesem sind die Daten direkt eingebettet, s. dazu Anhang B.2.

Die Web-Oberfläche

Die Web-Oberfläche trägt den Namen *web-experience*. Im Gegensatz zum Play Store ist das JavaScript im App Store nicht obfuskiert. Schaut man sich dises etwas genauer an, so kann man erkennen, dass die Oberfläche für den App Store mittels EmberJS [27] entwickelt wurde und man könnte prinzipiell die Funktionweise zurückentwickeln.

Beobachtet man über die Browser-Netzwerkanalyse die Requests, die auf der App-Detail-Seite getätigt werden, so stellt man fest, dass die Requests, die die eigentlichen Daten liefern alle nach dem Schema `amp-api.apps.apple.com/v1/catalog/<GL>/<RSRC>` aufgebaut sind. Dabei ist `GL` ein Länder-Kürzel und `RSRC` ein Pfad-Suffix, welches abhängig von der angefragten Ressource ist. Dieser Server wird als **AppHost** bezeichnet, daher wird dieser Begriff ab jetzt für Datenzugriff über diese URL verwendet.

Eine Ressource besteht aus einem Ressourcen-Typ und einer numerischen ID. Der Ressourcen-Pfad hat dann die folgenden Erscheinungsformen:

1. `/<BASE-TYPE>`
   Für die Abfrage mehrer IDs wird nur eine Pfadkomponente – der Ressourcen-Typ – angegeben. Die IDs werden dann über den URL-Parameter `ids` angegeben. Dieser hat als Wert komma-getrennt die abgefragten IDs.

2. `/<BASE-TYPE>/<ID>`
   Für die Abfrage einer einzelnen ID. Dies hat den Vorteil, dass der Pfad weiter fortgesetzt werden kann und keine URL-Parameter notwendig sind. Dies ist vor allem relevant für die Abfrage von Beziehungen zu anderen Ressourcen wie sie im nächsten Punkt erläutert wird.

3. `/<BASE-TYPE>/<ID>/<REL-TYPE>`
   Hier wird ein Datensatz einer bestimmte Ressource abgefragt, die Bezug auf eine andere Ressource nimmt oder sogar nehmen muss. Z.B. fallen darunter Rezensionen, ähnliche Apps oder die Genres einer App.

Für Basis-Typen sind der Bezeichner des Ressourcen-Typs und die Pfadkomponente identisch. Für relative Typen kann dies aber unterschiedlich sein. Z.B. ist der Ressourcen-Typ für Nutzer-Rezensionen „user-reviews“, aber die Pfadkomponente ist „reviews“. Tabelle 3.7 listet die bekannten Ressourcen-Typen auf und ob sie Basis-Typen sind.

Tabelle 3.7: Ressourcen-Typen für den App Store

Ressourcen-Typ	*Basis-Typ?*	*Beschreibung*
apps	Ja	Liefert Details zu einer App.
user-reviews	Nein	Liefert Rezensionen für eine App.
developers	Ja	Liefert Details über einen Entwickler.
genres	Ja	Liefert Details über ein Genre.
in-apps	Ja	Liefert Daten über einen In-App-Kauf.

Die ID, die dann im Pfad verwendet werden muss ist dann natürlich vom Ressourcen-Typ abhängig, z.B. eine App ID für *apps*, eine Genre ID für *genres*, etc. ID's sind generell immer numerisch beim App Store. Basis-Typen können direkt als Pfadkomponente verwendet werden, um eine bestimmte ID abzufragen. Tabelle 3.9 listet die Namen der Pfadkomponenten für Abfragen von Daten relativ zur Basis-Ressource *apps*.

Tabelle 3.9: Relative Ressourcen für die Ressource *apps*

Pfadkomponente	*Ressourcen-Typ*	*Beschreibung*
customers-also-bought-apps	apps	Liefert ähnliche Apps zu einer App.
developer-other-apps	apps	Liefert andere Apps desselben Entwicklers.
genres	genres	Liefert die Genres, zu der die App gehört.
developer	developers	Liefert den Entwickler der App.
top-in-apps	in-apps	Liefert Daten bzgl. In-App-Käufen in der App.
reviews	user-reviews	Liefert Nutzer-Rezensionen für die App.

Generell ist es so, dass bei Abfrage eines Basis-Typs immer nur eine feste Anzahl an Datensätzen abgefragt wird (erste oder zweite Form des Ressourcen-Pfads). Bei relativen Daten kann es aber auch der Fall sein, dass man eine ganze Liste von Datensätzen als Ergebnis erhält, die aber nicht alle auf einmal geliefert werden können. Hier kommt dann wieder *Pagination* ins Spiel. Bevor darauf detaillierter eingeangen wird, wird zunächst grob das Format der Antwort-Daten besprochen.

Als Antwort erhält man immer eine *Datenseite*. Dadurch umgeht man die Notwendigkeit, Daten mit und ohne *Pagination* zu unterscheiden. Eine Datenseite ist ein JSON-Objekt mit den folgenden Schlüsseln:

Schlüssel	*Wert*
next	Existiert nur, wenn es eine „nächste“ Datenseite gibt. Angegeben ist der URL-Pfad (inklusive URL-Parameter) für die nächste Seite.
data	Ein Json-Array, der die Liste von Datensätzen beinhaltet (nur ein Einträg für Abfrage einer einzelnen ID). Die Einträge werden *Items* genannt.
href	Der URL-Pfad, von dem die abgefragte Ressource stammt. Findet sich nur bei den Datenseiten im `relationships`-Objekt eines Items.

Jedes *Item* ist ein JSON-Objekt, das die Daten für eine einzelne ID beinhaltet. Die Schlüssel sind die folgenden:

Schlüssel	*Wert*
id	Die ID des Items (z.B. App ID, Review ID, Genre ID, etc.)
type	Der Bezeichner für den Ressourcen-Typ.
attributes	Ein JSON-Objekt, dass die *Attribute* des Items beinhaltet. Dies sind die eigentlichen Daten.
href	Der Pfad für eine AppHost-URL, die die Daten zu genau diesem einen Item liefert. Existiert nur bei Ressourcen-Typen, die ein Basis-Typ sind.
relationships	Ein JSON-Objekt, mit dem man sich Daten einer relativen Ressource direkt „mitliefern“ lassen kann. Ein Schlüssel ist jeweils die Pfadkomponente einer relativen Ressource und der Wert ist die zugehörige Datenseite.

Für AppHost-Requests gibt es einige relevante URL-Parameter, die teilweise auch erforderlich sind, um einen Request korrekt zu bilden. Eine ausführlichere Beschreibung findet sich in Anhang B.1.

Darüberhinaus gibt es aber noch zwei weitere Hürden, um AppHost-Requests korrekt senden zu können. Erstens ist es wichtig, den `Accept`-Header auf `application/json` zu setzen, damit der Server die Daten korrekt liefert. Zweitens ist es erforderlich, den `Authorization`-Header anzugeben.

Der `Authorization`-Header dient zur Client-Autorisierung und erlaubt unterschiedliche Autorisierungs-Methoden. Der Aufbau ist generell wie folgt:

`Authorization: <type> <credentials>` [3].

Der vom App Store genutzte Typ ist *Bearer*, welcher zur Autorisierung mittels OAuth 2.0-Token einsetzt wird [16]. Spezifiziert man diesen Header nicht oder gibt ein ungültiges Token an, so erhält man HTTP-Fehler 401 (Unauthorized).

Man kann dieses Token wie folgt erhalten: Der Abruf der Detail-Seite einer App ist ohne Probleme möglich, das Token wird nur verwendet, um das AppHost-Backend zu kontaktieren. Die Abfrage von Daten wird durch Skripte auf der Seite getätigt, daher müssen diese irgendwie das Token erhalten. Auf einer App-Detail-Seite gibt es im HTML-Dokument ein *meta*-Tag mit dem Attribut `name` und dem Wert `web-experience-app/config/environment`. Dieses Tag hat ein weiteres Attribut `content`, dessen Wert ein Json-String ist, welcher diverse Konfigurations-Daten beinhaltet, darunter auch das Token. Der Aufbau ist in Anhang B.1.3 erläutert. Wichtig ist, dass dieses HTML-Tag durch Skripte entfernt wird, man kann es also nur mit deaktiviertem JavaScript erhalten!

Datenquellen für Store-Funktionen

Die Web-Oberfläche kann nicht alle Store-Funktionen bedienen. Für manche Store-Funktionen lassen sich aber auch auf mehrere Arten und Weisen Daten erhalten. Tabelle 3.11 stellt die verschiedenen Datenquellen pro Store-Funktion vor und geht

auf einige Unterschiede ein. Anhang B beschreibt die verschiedenen Datenquellen detailliert.

Tabelle 3.11: Mögliche Datenquellen für App-Store-Datenextraktion

Funktion	*Quelle*	*Anmerkung*
AS01	AppHost	Liefert die ausführlichsten Informationen
AS01	WebObjects	Etwas weniger detaillierte Informationen
AS01	Lookup-API	Am wenigsten detaillierte App-Informationen
AS02	AppHost	Liefert nur App IDs, Lookup über AS01
AS02	Lookup-API	Liefert Daten im selben Format wie AS01+Lookup-API
AS03	Legacy RSS	Liefert Daten über alte RSS-Feeds
AS03	RSS	Liefert Daten über aktuelle RSS-Feeds
AS04	AppHost	Rezensionen über die Web-Oberfläche
AS04	RSS	Rezensionen über RSS-Feeds
AS05	WebObjects	Suche über Legacy WebObjects. Liefert nur App IDs, Lookup mittels AS01
AS05	Lookup-API	Liefert Daten im selben Format wie AS01+Lookup-API
AS06	WebObjects	Ähnliche Apps über Legacy WebObjects. Liefert nur App IDs, Lookup mittels AS01
AS06	AppHost	Liefert Daten im selben Format wie AS01+AppHost
AS07	WebObjects	Einzig bekannte Möglichkeit für Suchvervollständigung

Die Lookup-API und RSS-Feeds

Auf die Lookup-API und RSS-Feeds wird hier nicht näher eingegangen. Die öffentlich bekannten Informationen finden sich für die Lookup-API in Abschnitt B.3 und für RSS-Feeds in Abschnitt B.6.

3.3.3 Definition der Scraper-Methoden

Im vorigen Abschnitt wurden die Datenquellen und -formate erörtert. Basierend darauf lassen sich nun die Scraper-Methoden definieren, die die Store-Funktionen im Scraper zur Verfügung stellen. Eingabeparameter und zurückgegebene Daten werden hier nur kurz beschrieben, in Anhang C.2 werden Parameter, Datenformat und potenzielle Fehler detailliert ausgeführt.

Es ist anzumerken, dass bei allen Scraper-Methoden die Eingabe-Parameter „Sprachkürzel“ und „Länderkürzel“ möglich sind, mit denen man die Sprache und das Land festlegen kann. Diese wurden in der Zusammenfassung hier weggelassen. Bei AppHost-Methoden gibt es außerdem noch die Möglichkeit, das Autorisierungs-Token und Optionen für den Detailgrad der extrahierten Daten anzugeben. Auch diese Parameter wurden wier weggelassen.

Tabelle 3.13: Scraper-Methoden des App Stores

Methode	*Funkt.*	*Parameter*	*Daten*
details_apphost **details_itunes** **details_api**	AS01	App-ID	Liefert detaillierte App-Informationen via AppHost, WebObjects oder über die Such-API.
developer_apphost **developer_api**	AS02	Entwickler-ID	Liefert detaillierte App-Informationen für die Apps eines Entwicklers.

Tabelle 3.13: Scraper-Methoden des App Stores

Methode	*Funkt.*	*Parameter*	*Daten*
listApps_old **listApps_new**	AS03	Sammlungs-ID max. Item-Anzahl Kategorie-ID	Liefert Informationen über Apps aus einer Sammlung.
reviews_rss **reviews_apphost**	AS04	App-ID max. Item-Anzahl Sortieroptionen	Liefert eine Liste von Daten über Rezensionen für die App.
search_api **search_apphost**	AS05	Suchbegriff	Liefert eine Liste von detaillierten App-Informationen für alle Apps, die für den Suchbegriff gefunden wurden.
similar_apphost **search_api**	AS06	App-ID	Liefert eine Liste mit detaillierten App-Informationen für ähnliche Apps zu der gegebenen App.
suggest	AS07	Suchbegriff	Liefert eine von Vervollständigungen des Suchbegriffs.

4

Entwurf und Design

Dieses Kapitel widmet sich dem Design und Entwurf des Scraper-Kerns. In Abschnitt 4.1 wird zunächst auf die generelle Struktur des Scraper-Kerns eingegangen, d.h. in welche Komponenten er unterteilt wird und welche Beziehungen diese untereinander haben. In Abschnitt 4.2 wird auf das Interface *RequestGenerator* eingegangen, das auch die Struktur des Datenfluss im Scraper-Kern beschreibt. Die Abschnitte 4.3, 4.4 und 4.5 beschreiben dann noch wie Antworten auf Requests verarbeitet werden, wie Requests einfacher erzeugt werden können und wie Pagination funktioniert.

4.1 Grobe Architektur des Scraper-Kerns

Abbildung 4.1 stellt die drei zentralen Komponenten des Scraper-Kerns dar:

- **request-backend** ist die Komponente, die für das eigentliche Senden und Empfangen von HTTP-Requests zuständig ist. Sie fungiert als Abstraktion von den eigentlichen Bibliotheken, die das Senden übernehmen. Dadurch ist es möglich, unterschiedliche Bibliotheken zu verwenden und Funktionalität einfach zu ergänzen, z.B. Request Caching. Die Interaktion mit anderen Komponenten geschieht über das Interface `RequestGenerator`, welches in Abschnitt 4.2 näher erläutert wird.
- **common** ist die Komponente, die store-unabhängige Funktionalität bereitstellt. Dazu gehören insbesondere Hilfsfunktionen für die Generierung von HTTP-Requests (das `request-generation`-Modul), für die Verarbeitung

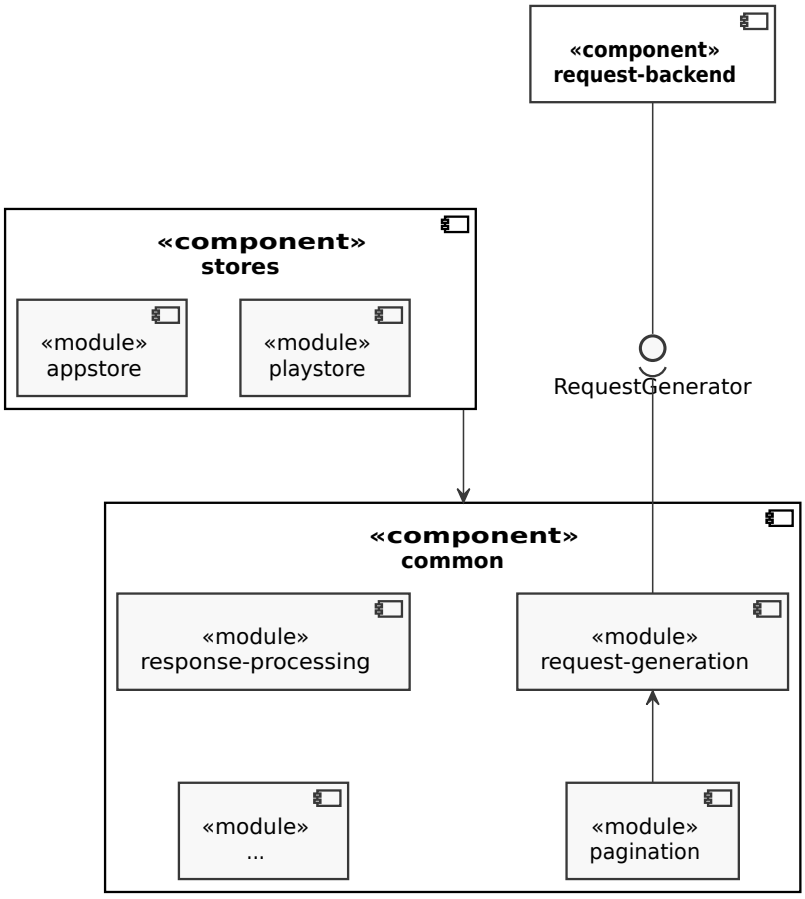


Abbildung 4.1: Komponentendiagramm für den Scraper-Kern

von HTTP-Antworten (das `response-processing`-Modul), für die Umsetzung von Pagination (das `pagination`-Modul) und weitere implementierungsabhängige Aspekte.

- **stores** ist die Komponente, die die Store-Module beinhaltet. Die jeweiligen Module enthalten die Scraper-Methoden und nutzen dann die `common`-Komponente, um HTTP-Requests zu generieren und Antworten zu verarbeiten.

Die Module der `common`-Komponente implementieren den Großteil der eigentlichen Funktionalität des Scraper-Kerns. Ihre Aufgaben und in Teilen auch ihre Datenstrukturen und Funktionen werden in den nachfolgenden Abschnitten genauer beschrieben. Eine besondere Bedeutung kommt dem Interface `RequestGenerator` zu, welches

die Schnittstelle zwischen dem `request-backend` und der `common`-Komponente definiert.

4.2 Das *RequestGenerator*-Interface

Dieser Abschnitt beschäftigt sich damit, wie mit Hilfe des `RequestGenerator`-Interfaces die `request-backend`-Komponente mit einem Server-Endpunkt eines Stores kommuniziert. Abbildung 4.2 beschreibt die Funktionen, die das Interface zur Verfügung stellen muss und mit deren Hilfe sich das Abflussprotokoll definieren lässt.

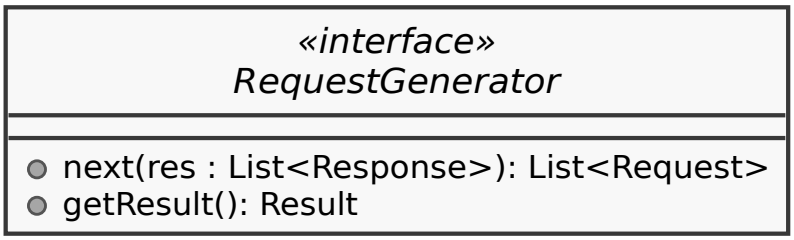


Abbildung 4.2: `RequestGenerator`-Interface

Die generelle Idee hinter dem Interface ist, dass es wiederholt einen Request erzeugt, dann die Antwort zu diesem Request erhält und basierend darauf den nächsten Request erzeugt. Aus den Antworten auf die Requests kann somit nach und nach ein Ergebnis aufgebaut werden. Dies ist nützlich, da viele Scraper-Methoden nicht nur einen einzelnen, sondern mehrere Requests für ein vollständiges Ergebnis erfordern. Insbesondere lässt sich so auch Pagination umsetzen. Im Detail funktionieren die beiden Methoden des Interfaces wie folgt:

1. Die Methode **next** erzeugt eine Liste von Requests, die als nächstes gesendet werden sollen. Die Requests in dieser Liste dürfen parallel gesendet werden, es besteht keine Abhängigkeit dieser Requests untereinander. Als Parameter wird die Liste mit den Antworten auf die zuvor erzeugten Requests verlangt. Dabei müssen die Antworten in derselben Reihenfolge vorliegen wie die Requests, damit diese auch zugeordnet werden können. Für die initialen Requests wird

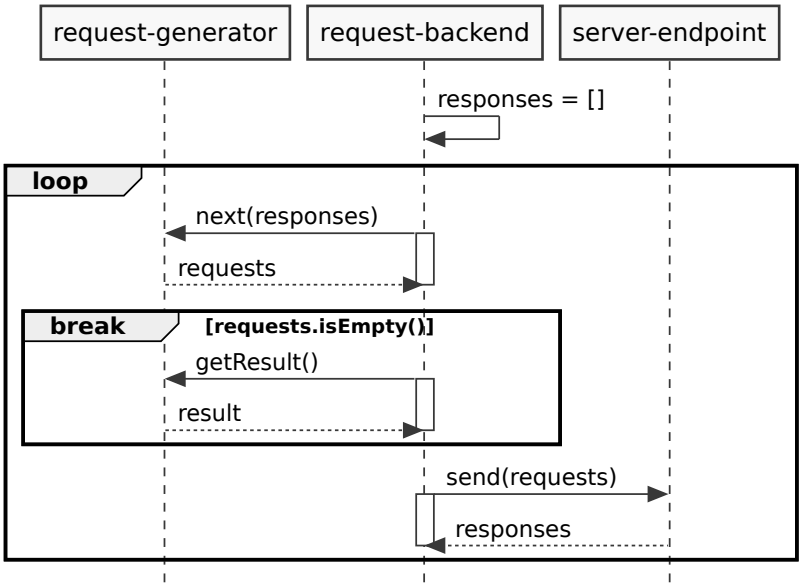


Abbildung 4.3: `RequestGenerator`-Datenfluss

eine leere Liste als Argument verwendet. Um zu signalisieren, dass keine weiteren Requests mehr folgen wird die leere Liste zurückgegeben.

2. Die Methode **getResult** liefert das Ergebnis. Mit den Antworten auf die Requests lassen sich nach und nach die erforderlichen Daten akkumulieren. Wenn alle notwendigen Daten angekommen sind, dann signalisiert die Methode `next`, dass keine weiteren Requests mehr nötig sind und das Ergebnis lässt sich mittels `getResult` abfragen.

Abbildung 4.3 demonstriert nochmals als Sequenzdiagramm das Prinzip des Interfaces.

Dieses Modell ermöglicht es z.B. Pagination umzusetzen: Die `next`-Methode generiert kontinuierlich den Request für die nächste Seite und verarbeitet dann die Daten für diese Seite. Beim Erreichen einer Abbruchbedingung erhält man dann via `getResult` die Daten für alle Seiten.

Bei Pagination wie sie z.B. im Play Store vorkommt kann der Request für die nächste Seite nur generiert werden, wenn man die Daten aus der vorigen Seite hat, da ein

Token benötigt wird. Manchmal ist es aber auch der Fall, dass die Requests für alle Seiten gleichzeitig getätigt werden können, d.h. es bestehen keine Abhängigkeiten zwischen den Requests. Aus diesem Grund liefert die `next`-Methode eine Liste von Requests, so besteht auch die Möglichkeit mehrere Requests gleichzeitig zu senden (so wie es z.B. auch Webbrowser machen). So kann man mit dem Interface Parallelität und Sequentialität gleichzeitig umsetzen.

4.3 Die Verarbeitungs-Pipeline

Das `RequestGenerator`-Interface legt selbst nicht fest, wie die empfangenen Daten verarbeitet werden. Das Modul `response-processing` soll genau hierbei helfen. Generell lässt sich die Verarbeitung der empfangenen Daten im Zusammenhang mit dem *RequestGenerator*-Modell in drei Schritte einteilen:

1. **Parsing**: Verarbeitet die im Körper der Antwort empfangenen Daten in eine passende Datenstruktur (z.B. String als JSON parsen).
2. **Transformation**: Transformiert die geparste Datenstruktur in ein einheitliches Schema, das für die weitere Verarbeitung geeignet ist (z.B. so dass die Struktur dem Datenformat der Scraper-Methoden entspricht).
3. **Akkumulation**: Zusammenführen der transformierten Daten mit den Daten, die aus den vorigen Requests extrahiert wurden (z.B. Anhängen an eine Datensatz-Liste für Pagination).

Natürlich lassen sich zwischen diesen drei Schritten bei Bedarf noch weitere Schritte einfügen oder man kann Schritte austauschen. Z.B. könnte man anstelle der Akkumulation auch einfach die neu erhaltenen Daten in eine Datenbank einfügen. Da eine Scraper-Methode am Ende aber ein Ergebnis liefern soll, ist dies hier nicht relevant.

All diese einzelnen Schritte bilden zusammen eine Art Pipeline. Das `response-processing`-Modul ist dafür verantwortlich, Hilfsfunktionen für einzelne Pipeline-Schritte zu Verfügung zu stellen und diese zu einer Pipeline zu kombinieren, wobei

die Pipeline am Ende einfach nur eine Funktion sein soll, die den bisherigen akkumulierten Wert und die Antwort auf den letzten Request annimmt. Bei Aufruf sollen dann die Pipeline-Schritte durchlaufen werden und am Ende soll der neue Akkumulator-Wert zurückgegeben werden.

Eine einfache Pipeline, die den Körper einer HTTP-Antwort in JSON parst, einen Schlüssel extrahiert und diese an die Akkumulator-Liste anhängt, könnte in Python z.B. so aussehen:

```
def pipeline(acc, response):
  json = JSON.parse(response.body) # Step 1: Parse
  json = json["data"]              # Step 2: Transform
  acc.append(json)                 # Step 3: Accumulate
  return acc                       # Return new accumulator
```

Das *response-processing*-Modul muss diesen Vorgang jetzt in einzelne, kombinierbare Schritte aufteilen, diese zu einer Pipeline zusammensetzen können und natürlich auch komplexere Transformationen, Parsing, etc. erlauben. Nähere Details zur Umsetzung muss die Implementierung festlegen.

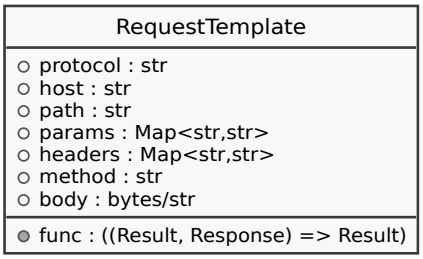
4.4 Request-Generierung und -Templates

Das `RequestGenerator`-Interface selbst ist die rohe Schnittstelle zum *request-backend*. Um Redundanz zu vermeiden und die Implementierung der Scraper-Methoden simpler zu gestalten hilft das `request-generation`-Modul zur Erzeugung von `RequestGenerator`-Instanzen.

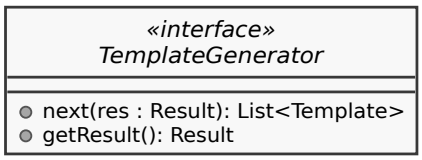
Dazu wird das Konzept der sog. *Request Templates* verwendet. Ein *Request Template* ist eine Datenstruktur, die einen unvollständigen HTTP-Request beschreibt und die um verschiedene Informationen erweitert werden kann, um so einen vollständigen HTTP-Request zu bilden. Sie definiert die üblichen Attribute eines HTTP-Requests (also Host, Pfad, URL-Parameter, HTTP-Methode, Header, Request-Körper, etc.),

wobei nicht für alle Attribute ein Wert angegeben werden muss. Die Store-Module können diese Request-Templates nutzen, um die unveränderlichen Daten einer Scraper-Methode festzulegen (z.B. Host oder URL-Pfad) und bei Verwendung der Methode kann das Template dann zu einem vollständigen Request ergänzt werden.

Zusätzlich soll ein Template eine Verarbeitungsfunktion haben, d.h. eine Funktion, die den bisherigen Akkumulatorwert und die HTTP-Antwort annimmt und daraus einen neuen Akkumulatorwert berechnet (könnte z.B. eine Pipeline-Funktion sein).



(a) Request-Template



(b) Template-Generator

Abbildung 4.4: Konzept von Request-Templates

Das *request-generation*-Modul stellt die Datenstrukturen für solche Templates zur Verfügung. Analog zum *RequestGenerator*-Interface lässt sich dann ein *TemplateGenerator* definieren, der dann nicht direkt Requests erzeugt, sondern nur Request Templates liefert. Aus einem *TemplateGenerator* lässt sich dann ein *RequestGenerator* erzeugen, der dann für die Antworten auf die HTTP-Requests immer die Verarbeitungsfunktion anwendet. Nähere Details zur Umsetzung muss die Implementierung festlegen.

4.5 Pagination

Das *pagination*-Modul nutzt das Konzept der Pagination, um Implementierungen des *TemplateGenerator*-Interfaces zur Verfügung zu stellen, die Pagination realisieren. Dies läuft dabei so ab, dass ein einzelnes Request Template zur Verfügung gestellt wird, das einen Request für eine Ressource beschreibt, die Pagination erfordert.

Dazu wird dann noch spezifiziert, welche Eigenschaft des Templates modifiziert werden soll und auf welche Art und Weise.

Z.B. könnte verlangt werden, dass ein URL-Parameter die Anzahl der bereits empfangenen Datensätzen wiederspiegelt, um so ein Offset zu erzeugen. Der Generator würde dann das Template solange wiederholen, bis ein Abbruchkriterium eintrifft (z.B. max. Anzahl an Items erreicht) und bei jedem mal das Offset um die Seitengröße erhöhen.

Es gibt generell zwei Typen von Pagination, die unterschieden werden müssen:

1. **Fixe Pagination**: Hier ist z.B. eine Seitennummer oder ein Offset erforderlich und alle Seiten haben eine feste Größe. In diesem Fall lassen sich sämtliche Requests auf einmal generieren. Haben wir z.B. eine Seitengröße von 50 Items und wir wollen 200 Items erhalten, so können wir 4 Requests mit den Seitennummern 1-4 oder den Offsets 0,50,100 und 150 erzeugen und diese parallel senden. Das *RequestGenerator*-Interface ist so konzipiert, dass es das auch erlaubt.
2. **Token-basierte Pagination**: Hierbei erhält man aus der Antwort auf einen Request ein Token (z.B. eine ID), mit der man den Request für die nächste Seite erzeugen kann. Dies hat den Nachteil, dass man die Requests nur sequenziell tätigen kann und man kein Start-Offset angeben kann.

Das *pagination*-Modul bietet die Möglichkeit für beide Typen von Pagination *TemplateGenerator*-Instanzen basierend auf einem Request-Template zu erzeugen.

Z.B. könnte die `next`-Methode eines Template-Generators für token-basierte Pagination wie folgt aussehen, wenn wir annehmen, dass der Akkumulator ein JSON-Objekt mit einem Schlüssel „token“ ist und das Token als URL-Parameter angegeben wird:

```
def next(acc : JSON) : List<Template>
  template.params["token"] = acc["token"]
  return [template]
```

Das *pagination*-Modul muss beide Typen von Pagination unterstützten und mehr Funktionalität zur Verfügung stellen, insbesondere die Festlegung des zu modi-

fizierenden Attributes und die Definition von Abbruchkriterien (ungültiges Token, maximale Anzahl an Datensätzen erreicht, etc.)

5

Implementierung

In diesem Kapitel wird auf die zentralen Aspekte der Implementierung des Scraper-Kerns und der REST-API eingegangen. Als Programmiersprache wird hier Python 3.8 verwendet. Zunächst werden einige grundlegende Konzepte von Python erläutert, die relevant für die Implementierung sind, dann einige Technologien, die eingesetzt wurden und zuletzt wird dann separat für Scraper-Kern und REST-API demonstriert, wie diese mittels dieser Technologien umgesetzt werden können.

5.1 Wichtige Konzepte in Python

Python ist eine interpretierte und objektorientierte Programmiersprache, die aber auch andere Paradigmen wie z.B. prozedurale und funktionale Programmierung unterstützt [5]. Darüberhinaus gibt es einige besondere Konzepte von denen in der Implementierung Gebrauch gemacht wird, daher werden diese hier näher erläutert.

5.1.1 *Iterable*, *Iterator* und *for*-Schleifen

Ein fundamentales Konzept in Python ist das der iterierbaren Objekte. Dies sind Objekte, die eine sequentielle Abfolge von anderen Objekten produzieren können. Das können z.B. Listen sein, die einfach ihre Einträge liefern oder Bäume, die ihre Knoten traversieren, etc.

```
##############################
# Create range Iterable
rangeIterable = range(2, 8, 2) # start=2, stop=8 (excl.), step=2
# Create range Iterator
rangeIterator = iter(rangeIterable)
# Iterate through the sequence
print(next(rangeIterator)) # Prints "2"
print(next(rangeIterator)) # Prints "4"
print(next(rangeIterator)) # Prints "6"
print(next(rangeIterator)) # Throws StopIteration exception
##############################
for i in range(2, 8, 2):   # Equivalent for loop
  print(i)
##############################
```

Listing 5.1: Beispiel: Python range-Iterator

In Python wird eine Klasse als **Iterable** bezeichnet, wenn sie die Methode `__iter__` implementiert. In Python nennt man Methoden, die mit zwei Unterstrichen anfangen und aufhören *Magic Methods* und sie werden teilweise etwas anders gehandhabt (das sog. *Special Method Lookup*)[4]. Diese spezielle magische Methode hat keine Parameter und muss ein `Iterator`-Objekt liefern.

Ein **Iterator**-Objekt wird für eine einmalige Iteration durch die erzeugte Sequenz verwendet. Dazu gibt es die magischen Methode `__next__`, die bei Aufruf den nächsten Wert der Sequenz liefert. Das Ende der Sequenz wird durch das Werfen einer **StopIteration**-Exception signalisiert. Ein Iterator ist selbst auch ein Iterable-Objekt und hat daher auch die `__iter__`-Methode, diese liefert meistens den Iterator selbst ohne Änderungen, kann ihn aber z.B. auch zurücksetzen.

Üblicherweise werden die magischen Methoden `__iter__` und `__next__` nicht direkt verwendet. Stattdessen gibt es die built-in-Funktionen `iter` und `next`. Diese Methoden sollten auch verwendet werden, da sie eine Abstraktionsschicht bilden und auch noch auf andere Arten und Weise Iteratoren erzeugt werden können [6].

Ein Beispiel für dieses Konzept ist die `range`-Funktion, diese liefert ein Iterable-Objekt, mit dem man ein Intervall von natürlichen Zahlen mit einer gegebenen Schrittweite durchlaufen kann. Listing 5.1 demonstriert das Beispiel.

```
# Die folgenden beiden Code Snippets sind äquivalent:
##############################
for i in iterable:
  do_someting(i)
##############################
itr = iter(iterable)
while True:
  try:
    i = next(itr)
    do_something(i)
  except StopIteration:
    break
##############################
```

Listing 5.2: Python: Arbeitsweise von for-Schleifen

Dieses Konzept wird in Python für fast alle Formen der Iteration verwendet (Ausnahme: while-Schleifen). In Python ist eine `for`-Schleife eine Schleife, die einen Iterator vollständig durchläuft, d.h. sie ruft zuerst `iter` auf, um den Iterator zu erzeugen und durchläuft dann die Werte mit `next` solange, bis eine `StopIteration` erzeugt wird. Die Syntax ist dabei generell immer: `for <VAR> in <ITER>`. Listing 5.2 demonstriert die Arbeitsweise von *for*-Schleifen.

5.1.2 *Generatoren*

Generatoren sind spezielle Iteratoren, die die folgenden drei zusätzlichen Methoden unterstützen: **send**, **throw** und **close**.

Die letzteren beiden Methoden dienen dazu, Exceptions in den Generator zu senden und den Generator vorzeitig zu „beenden“. Dabei liefert `throw` wie `next` den nächsten Wert oder wirf selbst wieder eine Exception. `close` ist ein Spezialfall von `throw`, wobei hier die sog. `GeneratorExit`-Exception verwendet wird und kein Wert zurückgeliefert wird. Der Sinn hinter diesem Mechanismus wird im weiteren Verlauf ersichtlich.

```
################################################################
def request_gen(requests):
  responses = []
  for req in reqs:
    try: res = yield req
    except IOError as e: responses.append(e)
    else: responses.append(res)
  return responses
################################################################
def requst_sender(gen):
  try:
    res = None
    while True:
      try:
        res = send_request(gen.send(res))
      except RequestException as e:
        res = send_request(gen.throw(e))
  except StopIteration as e:
    return e.value
################################################################
```

Listing 5.3: Beispiel für eine Generator-Funktion

Die `send`-Methode ist im Prinzip wie `__next__`, nur dass man hier nicht nur den nächsten Wert erhält, sondern auch noch einen Wert in den Generator hinein sendet. Dabei ist dieser Wert als eine Art „Antwort“ auf den vorher erhaltenen Wert gedacht (z.B. wenn er irgendwie transformiert wurde), als erster Wert muss `None` (Python's *null*) gesendet werden. Die Methode liefert dann wie `next` den nächsten Wert. Generell kann man sich die Verwendung von `next` als äquivalent zur Verwendung von `send` vorstellen, wobei immer `None` gesendet wird.

Das Konzept ist so wichtig, dass es in Python dafür eine extra Syntax gibt: Die sog. **Generator Functions**. Das sind Funktionen, die bei Aufruf einen Generator erzeugen. Eine Generator-Funktion erhält man automatisch, indem man das Schlüsselwort **yield** verwendet. Der Code läuft immer bis zu diesem Schlüsselwort und hält dort an. Der Wert bei einem `yield`-Ausdruck wird beim Aufruf von `next` zurückgegeben. Mit einer Zuweisung kann man gesendete Werte empfangen und eine `try-except`-Konstruktion kann Exceptions abfangen, die mittels `throw`

ausgelöst werden. Die Nutzung von `return` ist äquivalent zum Erzeugen einer `StopIteration`-Exception mit dem Wert des return-Ausdrucks.

Listing 5.3 demonstriert ein *Producer-Consumer*-Modell: `request_gen` ist eine Generator-Funktion (agiert als *Producer*), die Requests generiert, Antworten auf diese Requests empfängt und mittels `try-except` mögliche Fehler abfängt. Die zweite Funktion (`request_sender`, agiert als *Consumer*) nimmt einen Generator an und sendet dem Generator immer die Antwort auf den vorherigen Request (initial `None`). Falls ein Fehler beim Durchführen des Requests auftritt, wird die Exception an den Generator weitergeleitet. Dies wird solange wiederholt, bis eine `StopIteration`-Ausnahme auftritt, die das Ergebnis beinhaltet.

Dieses Konzept wurde sogar auf Listenkomprehensionen angewandt. Normale Listenkomprehensionen erzeugen eine Liste, man kann aber auch eine *Generator Comprehension* schreiben, die statt einer Liste einen Generator produziert, der dieselbe Sequenz liefert (s. Listing 5.4).

```
listExpr = [map_func(x) for x in iterable if filter_func(x)] # list
genExpr  = (map_func(x) for x in iterable if filter_func(x)) # gen
```

Listing 5.4: Beispiel für eine Generator-Funktion

Zu guter Letzt gibt es noch die Möglichkeit, an Subgeneratoren zu delegieren. Dabei wird ein **yield from**-Ausdruck verwendet. Hierbei wird der Kontrollfluss an den Subgenerator abgegeben, als Ergebnis erhält man den Wert, den der Subgenerator mittels `return` bzw. `StopIteration` zurückgibt.

Listing 5.5 zeigt ein Beispiel hierzu: Eine Generator-Funktion sendet zuerst einen initialen Request, als nächstes wird dann eine weitere Generator-Funktion aufgerufen, die basierend auf dem initialen Request einen Generator für paginierte Requests erzeugt. An diesen Subgenerator wird dann delegiert und am Ende das Ergebnis von diesem zurückgegeben.

```
def request_gen(initReq,paginatedReq):
  res = yield initReq
  res = yield from paginatedReq(res)
  return res
```

Listing 5.5: Beispiel für Subgeneratoren

5.1.3 Asynchrone Ausführung mittels *async/await*

Das Konzept der Generatoren wurde als Ausgangspunkt für sog. *Coroutinen* verwendet, die eine große Rolle bei der asynchronen Programmierung spielen, die z.B. mit NodeJS an Popularität gewonnen hat und in Python seit Version 3.5 funktional und seit Version 3.7 syntaktisch unterstützt wird.

Bei der asynchronen Programmierung geht es darum, dass langsame Operationen (primär I/O-Operationen) nicht die Programmausführung blockieren, so wie es bei klassicher I/O der Fall ist. Bei normaler, blockierender I/O wird der Thread, der eine I/O-Operation durchführt blockiert. Möchte man z.B. mehrere Requests gleichzeitg senden, so benötigt man Multithreading. Asynchrone I/O nutzt kein Multithreading, sondern Proxy-Objekte für das Ergebnis der Operation mittels derer man bei Bedarf auf das Ergebnis warten kann.

Bei Python nennt man diese Proxy-Objekte **Awaitable**s. Diese Objekte können zusammen mit dem **await**-Schlüsselwort verwendet werden, um auf ein Ergebnis zu warten. Es gibt im Wesentlichen drei Typen von `Awaitable`: **Coroutine**, **Task** und **Future**. Relevant sind hier für uns jetzt nur die Coroutinen.

Coroutinen sind eine Weiterentwicklung von Generatoren: Bei einer Generatorfunktion wird die Ausführung immer bei einem `yield`-Ausdruck pausiert und bei einem `next`-Aufruf läuft der Code bis zum nächsten `yield` weiter. Das `await`-Schlüsselwort funktioniert hier ähnlich wie `yield from`: Der Kontrollfluss wird an ein `Awaitable` weitergegeben und man erhält das Ergebnis von diesem. Während man auf das Ergebnis wartet, befindet sich die Coroutine in einem pausierten Zustand und wird nicht ausgeführt. Coroutinenen werden mit dem Schlüsselwort **async** definiert.

```
##################################
async def some_request():
  awaitable = make_request(...)
  response = await awaitable
  return process_response(response)
##################################
async def send_multiple_requests():
  import asyncio as aio
  reqs = [some_request() for i in range(10)]
  return await aio.gather(*reqs)
##################################
```

Listing 5.6: Beispiel für eine Coroutine

Listing 5.6 zeigt ein Beispiel für zwei Coroutinen. Die erste sendet einen Request, wartet auf die Antwort und verarbeitet diese. Für einen einzelnen Request hat man dadurch noch nicht viel gewonnen. Die zweite Funktion ruft die erste zehn mal auf, erzeugt also zehn `Awaitable`-Objekte, die alle einen Request senden werden. Die `gather`-Funktion fasst diese zu einem einzigen `Awaitable` zusammen, wartet man hier auf das Ergebnis, so erhält man eine Liste mit den einzelnen Ergebnissen. Die I/O-Operationen für das Senden der Requests werden zwar sequentiell initiiert, aber sobald die erste Coroutine auf ein `await` stößt, wird die nächste Coroutine bis zum ersten `await` ausgeführt, dann die nächste, etc. Der Thread ist niemals *idle*, es wird immer etwas ausgeführt (im Gegensatz zu blockierender I/O), es sei denn alle Couroutinen müssen warten.

Die „Parallelität“ entsteht durch den sog. *Event Loop*. Wenn eine Coroutine auf ein `await` stößt wird die Ausführung pausiert und eine andere Couroutine, die gerade nicht bei einem `await` wartet wird ausgeführt. Ist eine Operation fertig, auf die gerade gewartet wird, so wird die zugehörige Coroutine als „nicht wartend“ markiert. Der Event Loop hat eine Liste aller nicht-wartenden Couroutinen. Wenn also bei Beispiel 5.6 die Antwort für einen Request eintrifft, so wird die zugehörige Coroutine für die weitere Ausführung markiert und irgendwann auch ausgeführt. Es gibt keine garantierte Reihenfolge der Ausführung, es können also immer noch Formen von Synchronisation nötig sein. Erreicht eine Coroutine ein `return`, so wird sie als

„fertig“ markiert und alle anderen Coroutinen, die auf das Ergebnis gewartet haben können fortgesetzt werden.

In gewisser Hinsicht ist eine Coroutine also ähnlich zu einem Generator, der *Event Loop* ist dabei die Komponente, die die Ausführung der Coroutine fortsetzt, bei Generatoren macht dies der Programmierer selbst durch den Aufruf von `send` oder `next`. Es gibt auch asynchrone Generatoren, Schleifen, etc. auf die hier nicht eingegangen wird. Weitere Details zum Thema „asynchrone Programmierung in Python“ findet sich bei [7].

5.2 Eingesetzte Technologien

Dieses Kapitel widmet sich den eingesetzten Bibliotheken und Technologien für Scraper-Kern und die REST-API.

5.2.1 Die *JMESPath*-Query-Sprache

JMESPath ist eine Sprache zur Transformation von JSON-Daten, die auch durch eine formale Grammatik (ABNF-Notation) beschrieben ist [25]. Eine Implementierung der Sprache existiert in Form von Bibliotheken für viele Programmiersprachen, darunter auch Python. Die Transformation der Daten aus den Stores in das für die Scraper-Methoden definierte Format wird in der Implementierung ausschließlich durch JMESPath realisiert. Im Nachfolgenden wird grob die Funktionsweise von JMESPath erläutert.

Sämtliche JSON-Daten in diesem Abschnitt werden in Python-Notation geschrieben. Der einzige Unterschied ist, dass es in Python kein *null* gibt, denselben Zweck erfüllt das Schlüsselwort `None`. Außerdem werden `True` und `False` mit großen Anfangsbuchstaben geschrieben.

Pfade

Ein *Pfad* ist ein Ausdruck, der innerhalb der JSON-Struktur eine bestimmte enthaltene Teilstruktur „addressiert“. Betrachten wir z.B. das folgende JSON-Objekt:

```
{
  "key1": ["val1_1", None, 3123, True],
  "key2": [{"test": [1, 2, 3]}, False]
}
```

Ein JMESPath-Ausdruck beginnt immer beim Wurzel-Element, in diesem Fall das JSON-Objekt. Man addressiert einen Schlüssel in einem Objekt einfach durch Angeben des Schlüssel-Namens. Auf den Wert greift man mit einem Punkt zu. Falls ein Schlüssel-Name kein gültiger Bezeichner ist (darf nicht mit Ziffern anfangen, keine Bindestriche, etc.), so kann man ihn in Anführungszeichen setzen. Einen Array-Eintrag erhält man, indem man den Index in eckige Klammern schreibt. Falls der Pfad in irgendeiner Form ungültig sein sollte (nicht existierende Indizes, Schlüssel, o.Ä.), dann wird der Wert *null* (bzw. `None` in Python) zugeordnet. Die nachfolgende Tabelle demonstriert dieses Verhalten mit ein paar Beispiel-Pfaden.

Ausdruck	*Wert*
key1	`["val_1", None, 3123, True]`
key1[2]	`3123`
key2.test[0]	`1`
key5	`None`
key2.test2[4][3]."key-1"	`None`

Projektionen

Bei einer Projektion wird ein Array von Werten erzeugt und dann ein JMESPath-Ausdruck auf alle Einträge des Arrays angewandt anstatt auf den Array selbst. Dabei werden auch automatisch immer alle *null*-Werte herausgefiltert. Die einfachsten

Projektionen sind *Wildcard*-Projektionen, die einfach eine Projektion für einen Array oder die Werte aus einem Objekt erschafft. Eine Wildcard wird durch ein Asterisk dargestellt (als Schlüssel bei Objekten und in Klammern für Arrays). Betrachtet man z.B. das folgende JSON-Objekt:

```
{
  "key1": [[1,2,3],[4,5,6],[7,8,9]],
  "key2": [{"test": [1,2,3]}, {"test": [4,5,6]}],
  "key3": [True, 3213, None, "value"]
}
```

Dann listet die folgende Tabelle einige Beispiel für Projektionen auf:

Ausdruck	*Wert*
*	`[ [[1,2,3],[4,5,6],[7,8,9]], [{"test":[1,2,3]},{"test":[4,5,6]}], [True,3213,None,"value"] ]`
*[0]	`[[1,2,3],{"test": [1,2,3]},True]`
*[*].test	`[[],[[1,2,3],[4,5,6]],[]]`
key1[0]	`[1,2,3]`
key1[*][0]	`[1,4,7]`
key2[*].test	`[[1,2,3],[4,5,6]]`
key2[*].test[0]	`[1,4]`

Filter

Filter sind spezielle Projektionen, bei denen nicht alle Werte aus einem Array projiziert werden, sondern nur diejenigen, die ein Filter-Ausdruck akzeptiert. Ein Filter-Ausdruck ist ebenfalls ein JMESPath-Ausdruck, aber er muss einen booleschen Wert als Ergebnis haben. Dazu gibt es Operatoren wie *&&* (UND), // (ODER), etc. Ein

Filter wird ebenfalls in eckigen Klammern geschrieben, das erste Zeichen muss ein Fragezeichen sein, gefolgt von dem Ausdruck, der einen booleschen Wert ergibt. Mittels einfacher Anführungszeichen kann man ein String-Literal angeben und mittels Backticks beliebige JSON-Literale. Mittels `@` kann man den aktuellen Knoten referenzieren (jeweils der Eintrag aus dem Array, der geprüft wird). Die nachfolgende Tabelle zeigt Beispiele bezogen auf das JSON-Objekt aus dem vorigen Abschnitt.

Ausdruck	*Wert*
*[?[0]==‘4‘||test[0]==‘4‘]	`[[[4,5,6]],[{"test":[4,5,6]}],[]]`
*[?@==’value’]	`[[],[],["value"]]`
key1[?[0]<‘5‘]	`[[1,2,3],[4,5,6]]`

Slicing

Mit *Slicing* wird das „Zuschneiden“ eines Array bezeichnet. Wie Indizes werden Slice-Ausdrücke in eckige Klammern geschrieben (allerdings wird eine geschnittene Kopie geliefert). Die Syntax ist generell `<start>:<stop>:<step>`. Es muss mindestens ein Doppelpunkt vorkommen, bei nur einem Doppelpunkt wird `step` weggelassen. Die Zahlen für Anfang bzw. Ende können auch weggelassen werden um keine Einschränkungen zu machen. Negative Indizes werden vom Ende aus gezählt (z.B. ist *-1* der letzte Eintrag), bzgl. der Schrittweite heißt das, dass der Array von hinten durchlaufen wird. JMESPath hat dieselbe Syntax und Semantik, der Rest des Ausdrucks wird dann auf die Elemente des geschnittenen Arrays projiziert. Die nachfolgende Tabelle zeigt Beispiel bezogen auf das JSON-Objekt aus dem vorigen Abschnitt:

Ausdruck	*Wert*
key1[:2]	`[[1,2,3],[4,5,6]]`
key1[1:]	`[[4,5,6],[7,8,9]]`
key1[::2]	`[[1,2,3],[7,8,9]]`
key1[::-1]	`[[7,8,9],[4,5,6],[1,2,3]]`
key1[::-1][0]	`[7,4,1]`

Flattening

Flattening heißt, dass man einen Array „entschachtelt“. D.h. wenn ein Array selbst wieder Arrays enthält werden aus den Einträgen die Elemente genommen und zu einem Array zusammengefügt. Dies erreicht man in JMESPath, indem man eckige Klammern ohne Inhalt schreibt. Die nachfolgende Tabelle zeigt Beispiel vor und nachdem sie „geflattened“ wurden:

Ausdruck	*Wert*
*[*].test	`[[],[[1,2,3],[4,5,6]],[]]`
*[*].test[]	`[[1,2,3],[4,5,6]]`
*[*].test[][]	`[1,2,3,4,5,6]`
key1[?[0]<‘5‘]	`[[1,2,3],[4,5,6]]`
key1[?[0]<‘5‘][]	`[1,2,3,4,5,6]`
key1[-2:-3:-1]	`[[4,5,6]]`
key1[-2:-3:-1][]	`[4,5,6]`

Pipes

Irgendwann gelangt man an den Punkt, bei dem man eine Projektion stoppen möchte. Betrachtet man z.B. das folgende JSON-Objekt:

```
{
  "people": [
    {"age": 25, "name": "Hans Müller"},
    {"age": 42, "name": "Max Mustermann"},
    {"age": 12, "name": "Tina Wagner"}
  ]
}
```

Das Ziel sei jetzt, dass man das Alter der ersten Person über 20 Jahren ausgeben möchte. Ein Ausdruck, um alle Personen über 20 Jahren zu erhalten wäre:

```
people[?age>'20']
```

Jetzt könnte man meinen, einfach `[0]` anhängen zu müssen, dies ist aber falsch. Man würde den leeren Array erhalten. Der Grund dafür ist, dass wir immer noch eine Projektion anwenden, wir würden also `[0]` auf alle Elemente des gefilterten Arrays anwenden (also auf Person-Objekte), nicht auf den Array selbst.

Wir müssen die Projektion stoppen, dies geht mit der *Pipe*, die als `|` geschrieben wird. Der korrekte Ausdruck wäre daher:

```
people[?age>'20'] | [0].age
```

Die Leerzeichen sind optional und erhöhen die Lesbarkeit.

JSON-Ausdrücke

JMESPath erlaubt es auch, komplett neue JSON-Strukturen zu erzeugen. Für Arrays gibt man einfach als Werte die JMESPath-Ausdrücke als Einträge an und bei Objekten sind die Schlüssel Konstanten und die Werte JMESPath-Ausdrücke. Betrachten wir z.B. die folgende JSON-Struktur:

```
{
  "good_people": [[25,"Person A"], [42, "Person B"]],
  "bad_people": [[17, "Person Y"], [67, "Person Z"]]
}
```

Eine Person ist jetzt in einem Array gespeichert, wobei Index 0 das Alter und Index 1 der Name ist. Zusätzlich wurden noch „gute“ und „schlechte“ Personen getrennt. Wir wollen jetzt alle Personen in einer Liste haben, wobei eine Person ein Objekt mit den Schlüssel `age` und `name` sein soll, also so:

```
[
  {"age": 25, "name": "Person A"},
  {"age": 42, "name": "Person B"},
  {"age": 17, "name": "Person Y"},
  {"age": 67, "name": "Person Z"}
]
```

Der zugehörige JMESPath-Ausdruck ist:

```
[good_people, bad_people][].{age: [0], name: [1]}
```

Zunächst wird mit `[good_people, bad_people]` ein Array mit zwei Elementen erzeugt, der die Werte der entsprechenden Schlüssel beinhaltet (also die zwei Arrays mit Personen A und B bzw. Y und Z), dann wird „geflattened“ was eine Projektion erschafft, bei der wir auf jedes Element den Ausdruck `{age: [0], name: [1]}` anwenden, was zur Folge hat, dass ein Objekt mit den Schlüsseln *age* und *name* erschaffen wird, die die Werte aus den Indizes 0 und 1 des entsprechenden Array-Eintrags haben, also genau Alter und Name.

Funktionen

Zu guter Letzt bietet JMESPath die Möglichkeit, Funktionalität durch Funktionen zu ergänzen, z.B. lassen sich so Arrays sortieren, das Maximum aus einem Array bestimmen, etc. Betrachten wir z.B. die folgende JSON-Struktur:

```
[
  {"age": 5, "name": "Person A"},
  {"age": 17, "name": "Person B"},
  {"age": 67, "name": "Person C"},
  {"age": 23, "name": "Person D"}
]
```

Wir wollen jetzt die älteste erwachsene Person (mindestens 18 Jahre) bestimmen. Dazu müssen wir also zuerst für Personen über 18 filtern und dann das Maximum anhand des Schlüssels `age` bestimmen. Der JMESPath-Ausdruck dazu ist:

```
[?age>=`18`] | max_by(@, &age).
```

Zunächst wird ganz normal für Personen über 18 gefiltert. Dann wird eine Pipe verwendet, um die Projektion zu beenden, sonst würde der nachfolgende Ausdruck auf die einzelnen Personen angewandt werden und nicht auf den Personen-Array. Es wird dann die `max_by`-Funktion aufgerufen. Diese hat zwei Parameter: Der Array aus dem das Maximum bestimmt werden soll und ein Ausdruck, der den Vergleichswert aus einem Array-Eintrag liefert.

Der Ausdruck muss hier mit einem `&` anfangen. Damit signalisiert man eine verzögerte Auswertung. Schreibt man das Zeichen nicht, so wird der Ausdruck ausgewertet und der zugehörige Wert als Argument übergeben, nicht der Ausdruck selbst. In diesem Fall würde also versucht werden, den Wert des Schlüssels `age` aus dem aktuellen JSON-Element (also dem gefilterten Personen-Array) zu bestimmen. Da dieser nicht existiert, wäre der Wert dann *null*, die `max_by`-Funktion benötigt aber einen Ausdruck, den sie auf jeden Array-Eintrag anwenden kann, um den Vergleichsschlüssel zu erhalten und dafür existiert das Referenz-Symbol `&`.

5.2.2 *FastAPI*, *uvicorn* und *gunicorn*

FastAPI ist ein Python-Webframework mit dem sich REST-API's leicht aufbauen lassen. Es wird in diesem Abschnitt nicht auf die Details eingangen, sondern nur ein paar zentrale Eigenschaften genannt und ein paar Beispiele demonstriert.

Einige der Vorteile von *FastAPI* sind [23]:

- **Schnelligkeit**, vergleichbar mit *NodeJS* und *Go*
- **Automatische Argument-Validierung** mittels *Pydantic* und *Type Hints*
- **Interaktive, automatische** Dokumentation basierend auf Quellcode-Dokumentation

```
from fastAPI import FastAPI
app = FastAPI()

@app.get("/")
def get_root():
  return "Hello, World"
```

Listing 5.7: FastAPI-Beispiel 1

- **Basiert auf Standards** wie *OpenAPI* und *JSON.Schema*
- **intuitive** Nutzung, die **schnelle Entwicklung** erlaubt

Listing 5.7 demonstriert ein Beispiel für eine *FastAPI*-Anwendung, die bei Anfrage an den Wurzel-Pfad einfach den Text „Hello, World!“ liefert.

Das zentrale Objekt ist eine `FastAPI`-Instanz, die als Dekorator verwendet werden kann, um *Request Handler* zu definieren. Für Parameter werden Python Type Hints verwendet, um Argumente zu validieren. Default-Werte erlauben optionale Parameter bzw. über die `Query`-Klasse auch feinere Argument-Validierung (z.B. via regulären Ausdrucken). Pfad-Angaben können Python-Format-Strings sein, um Parameter in Pfaden zuzulassen. Listing 5.8 demonstiert einen Request-Handler, der eine JSON-Struktur basierend auf einer ID und optionalem Parameter liefert.

```
@app.get("/app/{app_id}/reviews")
def reviews(app_id : int, count : Optional[int] = 40):
  return get_reviews(id, count) # Get Reviews
```

Listing 5.8: FastAPI-Beispiel 2

Mehr Details zu FastAPI finden sich auf der Website unter [23].

Um eine FastAPI-Anwendung nutzen zu können braucht man einen **ASGI-Server** (*Asynchronous Server Gateway Interface*, Details s. [26]). **uvicorn** ist ein solche Server-Software, wobei hierbei Minimalismus angestrebt wird [30]. *uvicorn* eignet sich gut als Entwicklers-Server. Hat man z.B. ein *FastAPI*-Objekt namens *app* im

Modul `main.py`, so kann man mittels `uvicorn main:app --reload` einen Entwicklungsserver (Standard-Port 8000) starten. Mehr Details finden sich auf der Website [30].

Für einen Deployment-Server ist *uvicorn* selbst nicht so gut geeignet [31], stattdessen sollte man *uvicorn* als Worker für **gunicorn** verwenden. *gunicorn* ist ein **WSGI-Server** (*Web Server Gateway Interface*, s. [28]), der Hauptunterschied zu *ASGI* ist, dass asynchrone Funktionen nicht genutzt werden. *uvicorn* lässt sich als Worker verwenden, im Wesentlichen heißt das, dass es mehrere parallel laufende Event Loops geben kann, Details zu *gunicorn* finden sich auf der Webseite [29].

5.3 Der Scraper-Kern

5.3.1 Das *fetchers*-Modul

Das *fetchers*-Modul ist die Implementierung des *request-backend*s und ist für das eigentliche Senden und Empfagen von Requests zuständig. Dabei beinhaltet es im Wesentlichen die folgenden Klassen/Funktionen:

Klasse/Funktion	Bechreibung
FetchResult **FetchResponse**	Wrapper-Klassen für HTTP-Request und Antwort mit den üblichen Attributen (URL, Header, ... bzw. Statuscode, Header, ...)
RequestException	Fehlerklasse für Fehler beim Senden von Requests. Hat die Attribute *results* und *errors*. Beide liefern Iteratoren aus Anfrage-Ergebnis-Tupeln, wobei das Ergebnis `FetchResponse`, `IOError` oder `None` (nicht gesendet) sein kann. *errors* liefert nur fehlerhafte Requests.

Klasse/Funktion	Bechreibung
RequestGenerator	Ein Typ-Alias für einen Generator der via *yield* `FetchRequest`s (Liste oder ein einzelner Request) liefert und via *send* die Antworten darauf empfängt. Via *return* muss ein Ergebnis zurückgeliefert werden.
RequestExecutor	Eine Typ-Definition für ein aufrufbares Objekt (z.B. Funktionen), die einen `RequestGenerator` annehmen und das Ergebnis liefern.
setRequestExecutor(...) **getRequestExecutor()**	Setzt bzw. liefert den verwendeten `RequestExecutor`.
doRequest(...)	Eine Funktion, die einen `RequestGenerator` annimmt, die Requests sendet und das Ergebnis liefert.

Das Modul selbst legt nicht fest, mit welcher Bibliothek die Requests gesendet werden. Mit der Implementierung der `RequestExecutor`-Schnittstelle können eigene Implementierungen ergänzt werden. Standardmäßig gibt es die folgenden zwei Implementierungen:

- **lib_requests** nutzt die *requests*-Bibliothek, welche Requests blockierend und sychron sendet. Ist z.B. gut für Nutzung des Scrapers in CLI's geeignet.
- **lib_aiohttp** nutzt die *aiohttp*-Bibliothek, um Requests asynchron zu senden. Gut für parallele Nutzung, z.B. über die REST-API.

Standardmäßig wird `lib_requests` genutzt, aber die REST-API nutzt `lib_aiohttp`. Listing 5.9 zeigt ein Beispiel für die Nutzung des Moduls.

```
from fetchers import FetchRequest, doRequest
def req_gen(url : str) -> RequestGenerator[str]:
  res = yield FetchRequest(method="GET", url=url)
  return res.body
print(doRequest(req_gen("https://www.example.com")))
```

Listing 5.9: Beispiel für das *fetchers*-Modul

5.3.2 Das *common*-Modul

Dieses Modul enthält diverse Submodule, die vom Rest des Scrapers verwendet werden. Die nachfolgende Tabelle listet alle Submodule mit ihrem entsprechenden Zweck.

Klasse/Funktion	Bechreibung
spec	Definiert abstrakte Datenstrukturen für Templates.
custom_funcs	Definiert eigene zusätzliche JMESPath-Funktionen.
more_types	Hilfsfunktionen für JSON und Definitionen für Type Hints.
misc_utils	Verschiedene nützliche Hilfsfunktionen.
request_generation	Hilft zum Generieren von Requests.
response_processing	Hilft beim Verarbeiten von Antworten.
pagination	Hilft beim Durchführen von *Pagination*.

Das *spec*-System

Das *spec*-Submodul (*spec* als Abkürzung von *specification*) ist hauptsächlich dazu da um die Ideen, die in Abschnitt 4.4 ausgeführt wurden umzusetzen. Dazu gibt es die abstrakte Klasse **SpecBase**, die wie folgt funktioniert:

Kindklassen von `SpecBase` definieren Attribute, die einen zusammengetzten Datentyp bilden (z.B. besteht ein HTTP-Request aus Methode, Header, Pfad, etc.). Die Klasse `SpecBase` fügt die Möglichkeit hinzu von einer bereits existierenden Instanz eine neue Instanz *abzuleiten*. Das heißt, dass die Werte von einigen Attributen überschrieben werden können und alle nicht-überschriebenen Attribute ihren alten Wert behalten. Dabei wird das Objekt, von dem abgeleitet wurde nicht kopiert, sondern referenziert. Finden Änderungen im *Basis*-Objekt statt, so werden diese im abgeleiteten Objekt reflektiert.

Listing 5.10 demonstriert ein Beispiel für ein *RequestSpec*-Objekt, welches Attribute eines HTTP-Requests speichert, anfangs für ein GET-Request an `example.com`.

```
####################################
req = RequestSpec("GET", "example.com", "/")
print(req)  # "GET /" at "example.com"
####################################
req2 = req.derive(host="google.com")
print(req2) # "GET /" at "google.com"
####################################
req.host = "uni-ulm.de"
print(req)  # "GET /" at "uni-ulm.de"
print(req2) # "GET /" at "google.com"
####################################
req.path = "/in/iui-dbis"
print(req)  # "GET /in/iui-dbis" at "uni-ulm.de"
print(req2) # "GET /in/iui-dbis" at "google.com"
####################################
```

Listing 5.10: Beispiel für das *spec*-Konzept

Dann wird ein neuer Request abgeleitet mit dem Host `google.com`. Wird jetzt der Host des Original-Requests geändert, so hat dies keine Konsequenz auf den abgeleiteten Request, da dieser einen neuen Host gesetzt hat. Wird der Pfad geändert, so wirkt sich dies aber auf den abgeleiteten Request aus, da dieser dafür keinen neuen Wert gesetzt hat.

Man kann sich dieses Konzept als eine Art Baumstruktur vorstellen: Leitet man ein neues Objekt ab, erzeugt man an „Kindknoten“. Greift man auf ein Attribut zu, so wird der Pfad entlang der Elternknoten (bis zur Wurzel) abgegangen, bis ein Wert gefunden wird, der nicht `None` ist.

Das *spec*-Modul bietet ein paar nützliche Implementierungen dieses Konzepts an, die in der folgenden Tabelle aufgelistet sind:

Klasse	Bechreibung
SimpleMapSpec **SimpleMutMapSpec**	Implementiert ein assoziatives Array (Python *Dictionaries*, mutierbar und nicht mutierbar). Falls zu einem Schlüssel kein Wert existiert, wird im Basis-Objekt nachgeschaut (bis zur Wurzel). Die eigentlichen Daten werden im Attribut `backingMap` gespeicherrt.

Klasse	Bechreibung
ConcatMapSpec	Ähnlich wie **SimpleMapSpec**, aber hier werden als Werte Arrays von einzelnen skalaren Werten gespeichert. Beim Auslesen eines Schlüssels aus der Map erhält man die einzelnen skalaren Werte zu einem String konkateniert, optional mit einem spezifizierbaren Separator (z.B. ein Komma).
FormatMapSpec	Eine Erweiterung von `ConcatMapSpec`, die die magische Methode `__str__` so implementiert, dass bei einer String-Umwandlung ein spezifizierbarer Format-String mittels der Werte, die in der Map gespeichert sind formatiert wird und das Ergebnis zurückgegeben wird.

```
#####################################
m = SimpleMapSpec(backingMap={"key": "val"})
m = m.derive(backingMap={"key": "test"})
print(m["key"])  # "test"
#####################################
m = ConcatMapSpec(backingMap={"key": [1,2,3]})
m = m.derive(backingMap={"key": [4,5]})
print(m["key"]) # "1,2,3,4,5"
#####################################
m = FormatMapSpec(
  backingMap={"k1": "val", "k2": [1,2,3],
  template="{k1} and {k2}")
print(str(m)) # "val and [1,2,3]"
#####################################
```

Listing 5.11: Demonstration von *MapSpec*-Implementierungen

Request-Generierung

```
def makeRequestGenerator(specGen):
  acc = None # intermediate result
  ex  = None # last exception during sending

  while True:
    try:
      if ex is None: spec = specGen.send(acc)
      else:          spec = specGen.throw(ex)
    except StopIteration as e:
      return e.value
    else:
      ex = None

    if spec is not a RequestSpec (list):
      acc = spec; continue

    try:
      reqs = build request list from specs
      res  = yield reqs
    except RequestException as e:
      ex = e
    else:
      for i in range(len(reqs)):
        acc = reqs[i].callback(acc, res[i])
```

Listing 5.12: Umwandlung *SpecGenerator* zu *RequestGenerator*

Das Submodul *request_generation* definiert eine Klasse **RequestSpec**, die genau die Attribute hat, die im Entwurf in Abbildung 4.4b festgelegt wurden.

Darüberhinaus gibt es eine Typdefinition namens **SpecGenerator**, die im Wesentlichen die Funktionalität des `TemplateGenerator` aus Abschnitt 4.4 abbildet. Ein solcher Generator muss eine Liste von `RequestSpec`-Objekten, ein einzelnes `RequestSpec`-Objekt oder ein *Zwischenergebnis* (Typ frei wählbar) bei einem `send`-Aufruf liefern. Gesendet wird das aktuelle Zwischenergebnis (der aktuelle

Akkumulator-Wert, dieser wird durch Anwendung der Verarbeitungs-Funktion des `RequestSpec`-Objekts erzeugt). Am Ende muss er via `StopIteration` bzw. `return` das finale Ergebnis liefern (darf anderen Typ haben).

Die Funktion **makeRequestGenerator** erzeugt dann aus einem `SpecGenerator` ein `RequestGenerator`. Dabei lässt sich die Logik ungefähr mit dem Code in Listing 5.12 zusammenfassen.

Antwort-Verarbeitung

```
processor = GenericResultProcessor(
  parse      = lambda res: JSON.loads(res.body),
  pattern    = "[*].id",
  accumulate = lambda acc, res: (acc or []) + res
)
```

Listing 5.13: Demonstration der *GenericResultProcessor*-Klasse

Das Submodul *response_processing* implementiert die „Pipeline“, wie sie in Abschnitt 4.3 beschrieben wurde. Die zentrale Klasse ist hier **GenericResultProcessor** (auch eine Subklasse von `SpecBase`). Diese Klasse implementiert die folgenden drei Schritte: *Parsing*, *Transformation* und *Akkumulation*.

Für das Parsing kann man eine Funktion festlegen, die ein `FetchReponse`-Objekt bekommt und JSON-Daten liefern muss. Die Transformation geschieht über einen JMESPath-Ausdruck und die Akkumulation geschieht über eine Funktion, die Akkumulatorwert und die transformierten JSON-Daten annimmt und den neuen Akkumulatorwert liefert. Sämtliche Schritte sind optional, das Standardverhalten ist, dass der Antwort-Körper als JSON geparst wird, keine Transformation durchgeführt wird und der Akkumulatorwert einfach überschrieben wird.

Eine `GenericResultProcessor`-Objekt kann wie eine Funktion aufgerufen werden und daher als Funktion bei `RequestSpec`-Objekten verwendet werden, um die Antwort zu verarbeiten. Listing 5.13 zeigt ein Beispiel für die Verarbeitung einer

Antwort bei der angenommen wird, dass sie ein JSON-Array mit Objekten beinhaltet, die einen Schlüssel `id` haben. Die ID-Liste wird dann an die Akkumulator-Liste angehangen.

Pagination

```
def calcRange(start=0,stop=None,step=1,count=None,size=None)
def fixedPagination(spec,attrName,attrKey,rng)
def tokenPagination(spec,resKey,tokKey,attrName,count=None)
```

Listing 5.14: Pagination-Hilfsfunktionen

Das Submodul *pagination* setzt die zwei Typen von Pagination um, die in Abschnitt 4.5 diskutiert wurden. Die Funktionen, die in Listing 5.14 gezeigt werden helfen bei der Umsetzung dieser zwei Typen von Pagination. Dabei haben sie die folgenden Verwendungszwecke:

- **calcRange** dient dazu, ein iterierbares Objekt für den `rng`-Parameter der `fixedPagination`-Funktion zu erzeugen. Man kann Startseite, Stopseite, Schrittweite, Größe einer Seite und maximale Anzahl an Items angeben. Es lassen sich hiermit auch Offset-Parameter realisieren, indem man einfach die passende Schrittweite wählt.
- **fixedPagination** erzeugt einen `SpecGenerator`, der die Requests für alle Seiten auf einmal liefert. Die *RequestSpec*-Objekte werden aus `spec` abgeleitet. Dabei wird angenommen, dass der zu variierende Parameter in einer Map gespeichert ist. `attrName` ist der Name des Attributes und `attrKey` der Schlüssel für die Map.
- **tokenPagination** ist für token-basierte Pagination. Die Requests werden auch hier aus `spec` erzeugt, es wird erwartet dass die Zwischenergebnisse JSON-Objekte sind. Diese müssen einen Schlüssel für das Token haben (`tokKey`) und einen mit einer Liste für die eigentlichen Items (`resKey`). Das nächste Token wird für das *RequestSpec*-Objekt in einer Map gespeichert. Wie für

fixe Pagination gibt `attrName` den Name des Attributes an und `tokKey` den Schlüssel. Mit `count` kann man noch einen Schwellwert für die maximale Item-Anzahl angeben.

Die tatsächlichen Funktionen haben noch ein paar Parameter mehr für zusätzliche Angaben, aber mit diesen zwei Funktionen kann man Pagination erheblich einfacher realisieren. Um dies mit initialen Requests zu kombinieren, kann man einfach an die erzeugten Generatoren mittels *yield from* delegieren. Beispiele für den Einsatz dieser Funktionen werden in Abschnitt 5.3.3 demonstriert.

5.3.3 Das *playstore*-Modul

Das *playstore*-Modul implementiert die Scraper-Methoden für den Play Store. Die eigentlichen Implementierungen sind in Submodulen für die einzelnen Methoden zu finden, wobei es ein spezielles Submodul `util` gibt, das allgemeine Funktionen wie z.B. für das Parsing zu Verfügung stellt.

Im `util`-Modul findet sich ein Submodul `parse`, das Funktionen für das Parsen der JSON-Daten aus dem Play Store enthält. Die Funktion `parseGPlayHTML` liefert für gegebene Skript-Tag-Schlüssel eine Funktion, die aus einer HTTP-Antwort ein JSON-Objekt parst, dass die JSON-Daten aus den Skript-Tags beinhaltet. Listing 5.15 zeigt die Arbeitsweise.

```
parser = parseGPlayHTML({"ds:5", "ds:8", "ds:3"})
resp   = ... # FetchResponse
json   = parser(resp)
# Resulting JSON object looks like this:
{
  "ds:5": <data from ds:5>,
  "ds:8": <data from ds:8>,
  "ds:3": <data from ds:3>
}
```

Listing 5.15: Demonstration des Skript-Tag-Parsers für den Play Store

Gibt man nur einen Schlüssel an und keine Menge von Schlüsseln, so erhält man direkt die Daten von diesem Schlüssel. Umsetzen lässt sich diese Funktion mittels HTML-Parsing oder regulärer Ausdrücke, die nach die entsprechenden Skript-Tags akzeptieren. Z.B. akzeptiert der folgende reguläre Ausdruck die passenden Skript-Tags:

```
>\s*AF_initDataCallback[\s\S]*?</script>
```

Hat man ein Treffer, so liefern die folgenden beiden regulären Ausdrücke Schlüssel und Daten aus dem Skript-Tag (jeweils in Gruppe Nr. 1):

```
[\"'](ds:.*?)[\"']
data:\s*([\s\S]*?)\}\);\s*</script>
```

Für die Extraktion der Daten aus batchexecute-Requests gibt es die Funktion `parseBatchExec`, die im Wesentlichen nicht mehr macht als der folgende Code:

```
def parseBatchExec(postText):
  data = JSON.loads(postText[5:])
  return JSON.loads(data[0][2])
```

Damit sind alle Hilfsmittel für das Implementieren der Scraper-Methoden des Play Stores vorhanden. Als Beispiel wird hier grob skizziert, wie die Methode `clusterList` implementiert wird.

Die Vorgehensweise zur Implementierung einer Scraper-Methode folgt immer dieser Vorgehensweise:

1. Festlegen der nötigen Requests und wie die Antworten verarbeitet werden (Nutzung von `RequestSpec` und `GenericResultProcessor`). Diese können auch unvollständig sein und Daten später vom Nutzer ergänzt werden.

2. Schreiben der zugehörigen `SpecGenerator`-Funktion, die die vervollständigten `RequestSpec`-Objekte in der zu sendenden Reihenfolge ausgibt. Für Pagination kann das zugehörige Modul verwendet und an den erzeugten Subgenerator delegiert werden.

```
_CLUSTER_LIST_PROCESSOR = GenericResultProcessor(
  # Legt JMESPath-Optionen fest, hier zusätzliche Funktionen
  jpOpts  = jp.Options(custom_functions=EXTENDED_FUNCS),
  pattern = r'[0].{'
    r'token: [3][1]'
    r'topics: [1][?[7]!=null][7][1][*].{'
      r'icon: [1][3][2],'
      r'name: [3],'
      r'id: split([4][4][2], `"?id="`)[-1]'
    r'}[],'
    r'clusters: [1][?[0]!=null].{...}'
  r'}'
)
_CLUSTER_LIST_INIT_SPEC = RequestSpec(
  host     = "play.google.com",
  params   = {"hl": "en", "gl": "us"},
  callback = _CLUSTER_LIST_PROCESSOR.derive(
    parseResponse = parseGPlayHTML("ds:3"),
  )
)
```

Listing 5.16: `clusterList`: initiale Request-Definitionen

3. Eine Funktion definieren, die die `makeRequestGenerator`-Funktion nutzt um einen Request-Generator zu erzeugen und dann mit dem `fetchers`-Modul die Scraper-Methode ausführt.

Für die `clusterList`-Methode gibt es z.B. zwei nötige Requests: Der initiale Request an die HTML-Seite und dann die Pagination-Requests. Also müssen `GenericResultProcessor` und `RequestSpec`-Definitionen für diese Requests angegeben werden.

Listing 5.16 zeigt die Definition des Antwort-Prozessors und des initialen Requests. Man beachte, dass für den Antwort-Prozessor nicht angegeben wurde, wie er die HTTP-Antwort parsen soll, da sich dies zwischen dem initialen und den nachfolgenden Requests unterscheidet (nicht aber die Struktur der enthaltenen Daten). Die Transformation der Play-Store-Daten in das Ergebnisformat findet ausschließlich durch den JMESPath-Ausdruck statt, Änderungen im Format müssen also auch nur hier geändert werden.

```
_CLUSTER_LIST_PAG_SPEC = RequestSpec(
  host    = "play.google.com",
  path    = "/_/PlayStoreUi/data/batchexecute",
  params  = {"hl": "en", "gl": "us", "rpcids": "w3QCWb"},
  method  = "POST",
  headers = {"Content-Type": "application/x-www-form-urlencoded"},
  body    = FormatMapSpec(
    backingMap = {"count": 15},
    template   = "%5B%22w3QCWb%22%2C%22%5B{count}%2C{token}%5D%22%5D"
  ),
  callback = _CLUSTER_LIST_PROCESSOR.derive(
    parseResponse = lambda res: parseBatchExec(res.body),
    accumulate    = RESULT_MERGE
  )
)
```

Listing 5.17: `clusterList`: paginierte Request-Definitionen

Für den paginierten Request liegen die Daten im selben Format vor wie bei den Skript-Tags, daher kann man denselben Antwort-Prozessor verwenden. Die `RequestSpec`-Definition ist in Listing 5.17 gezeigt. Dabei sind die folgenden Dinge zu bemerken:

- Das Parsing läuft anders, da wir hier einen batchexecute-Request haben.
- Da wir Pagination haben, müssen wir die Daten aus einer Seite auch mit den Daten aus der vorigen Seite zusammenfügen (der Akkumulations-Schritt). Konkret heißt das, dass der Schlüssel `token` mit dem Token aus der nächsten Seite aktualisiert werden muss und die `topics`- und `cluster`-Listen mit denen der vorigen Seiten zusammengefügt werden müssen. Dafür definiert das `response_processing`-Modul die Funktion `RESULT_MERGE`, die zwei JSON-Strukturen zusammengefügt. Für skalare Werte (Strings, Zahlen, etc.) wird der neue Wert genommen, Listen werden konkateniert und für Objekte werden die Werte aller Schlüssel rekursiv zusammengefügt (für Schlüssel, die in beiden Objekten existieren).
- Für einen batchexecute-Request ist der entsprechende Request-Körper nötig (s. Anhang A.2). Wir nehmen als Beispiel mal an, dass der Körper `["w3QCWb","[<CNT>,<TOK>]"]` sei (mit max. Item-Anzahl und Token).

Hierfür können wir die Klasse `FormatMapSpec` nutzen, der Format-String wäre einfach `["w3QCWb",[{cnt},{tok}]]`. Es muss allerdings berücksichtigt werden, dass dieser String url-kodiert werden muss! Die Format-Optionen lassen sich dann einfach als Map-Einträge festlegen.

Als nächsten Schritt muss man die `SpecGenerator`-Funktion definieren, die die entsprechenden Requests ausgibt. In unserem Fall also zuerst den initialen Request und dann die paginierten Requests mittels der `tokenPagination`-Funktion (Delegation an den erzeugten Subgenerator). Daten zur Vervollständigung der Requests können einfach per Parameter angenommen werden. Der Code dazu sieht dann ungefähr so aus wie in Listing 5.18.

```
def clusterListGen(urlPath, count=15, lang='en', country='us'):
  # Extrahiert den eigentlichen Pfad und URL-Parameter
  path, params = parse_path(urlPath)
  params["hl"] = lang; params["gl"] = country
  # Gebe initialen Request aus
  res = yield _CLUSTER_LIST_INIT_SPEC.derive(
    params = params,
    path   = path
  )
  # Erzeuge Subgenerator für pagination und delegiere
  res = yield from tokenPagination(
    spec = _CLUSTER_LIST_PAG_SPEC.derive(
      params = {"hl": lang, "gl": country}
    ),
    init    = res,         # initialer Akkumulator
    tokAttr = "body",      # zu modifizierendes Attribut
    tokKey  = "token",     # zu modifizierender Schlüssel
    resKey  = "clusters", # Schlüssel Ergebnis-Liste
    count   = count        # max. Anzahl an Datensätzen
  )
  # Gebe finales Ergebnis aus
  return res
```

Listing 5.18: `clusterList`: Definition des `SpecGenerator`s

Um die Requests dann auch tatsächlich zu senden muss dann nur noch mittels `makeRequestGenerator` der Request-Generator erzeugt werden und dieser dann an die `doRequest`-Funktion übergeben werden, um das Senden durchzuführen. Dies sieht dann z.B. so aus:

```
def clusterList(urlPath, count=30, lang='en', country='us'):
  clGen  = clusterListGen(urlPath, count, lang, country)
  reqGen = makeRequestGenerator(clGen)
  return doRequest(reqGen)
```

5.3.4 Das *appstore*-Modul

Das `appstore`-Modul ist großteils gleich aufgebaut wie das `playstore`-Modul. Hier finden sich im `util`-Modul die Submodule `parse` und `query`, die für das Parsing von App-Store-Antworten (falls nötig) und das Erzeugen von AppHost-Requests zuständig sind. Auf das `parse`-Modul wird jetzt nicht näher eingegangen, aber das `query`-Modul erfordert eine grobe Beschreibung.

Das `query`-Modul hilft dabei, AppHost-Requests zu erzeugen. Der Grund dafür ist, dass diese immer den `Authorization`-Header benötigen und daher evtl. ein zusätzlicher Request für das Autorisierungs-Token erforderlich ist. Dafür definiert das Modul die folgenden Methoden:

```
# Erzeugt ein RequestSpec-Objekt für ein Autorisierungs-Token
def buildAppStoreSessionSpec(id,lang,country): ...
# Erzeugt ein "Basis"-RequestSpec-Objekt für AppHost-Requests
def createAppHostSpecBase(params,headers,body,callback): ...
# Vervollständigt einen AppHost-Request mittels Autorisierung
def completeAppHostSpec(spec,rsrcPath,lang,country,authTok)
# Ein Generator, der wenn nötig ein Autorisierungs-Token abfrägt
def appHostSpecGen(id,spec,genFunc,authTok,lang,country,rsrcPath)
```

- **buildAppStoreSessionSpec** erzeugt ein vollständiges `RequestSpec`-Objekt mit dem sich aus einer App-Detail-Seite das Autorisierung-Token abfragen lässt. Sprache und Land sollten irrelevant sein, es wird aber eine App ID benötigt.
- **createAppHostSpecBase** erzeugt ein `RequestSpec`-Objekt für Anfragen an den AppHost-Endpunkt. URL-Parameter, Header, Request-Körper und Verarbeitungsfunktion können angegeben werden. Der Pfad des Requests ist eine `FormatMapSpec`-Instanz, die Map-Einträge sind Pfad-Parameter, von denen es zwei Stück gibt: Einen für die Storefront und einen für den Ressourcen-Pfad. Die Konstanten `PATH_ARG_STOREFRONT` und `PATH_ARG_RSRC` sind die Schlüssel für diese Parameter.
- **completeAppHostSpec** vervollständigt einen AppHost-Request, der mittels `createAppHostSpecBase` erschaffen wurde. Dabei muss das *RequestSpec*-Objekt, der Ressourcen-Pfad, Sprache, Land und das Autorisierungs-Token angegeben werden.
- **appHostSpecGen** ist eine Generator-Funktion, die sich um das Abfragen eines Autorisierungs-Token kümmert, sodass man dies nicht selbst tun muss. Die meisten Parameter sind wie bei `completeAppHostSpec`. Hat man bereits ein Token, so kann man dieses direkt angeben, falls nicht wird durch das Verwenden von `None` ein extra Request für die Abfrage eines Tokens gemacht (dafür wird eine App ID benötigt). Dann wird `completeAppHostSpec` verwendet, um das *RequestSpec*-Objekt zu vervollständigen und `genFunc` mit dem vervollständigten Objekt aufgerufen (daraus können dann z.B. neue Requests abgeleitet werden, die dann bereits autorisiert sind). Es muss ein *SpecGenerator* geliefert werden, an den dann die Ausführung delegiert wird.

Die generelle Vorgehensweise für AppHost-Requests ist großteils analog zum Play Store, es sollten allerdings die Funktionen aus dem *query*-Modul verwendet werden, um die entsprechenden `RequestSpec`-Objekte und `SpecGenerator`-Generatoren zu erzeugen. Z.B. könnte die Scraper-Methode `details_apphost` wie in Listing 5.19 umgesetzt werden.

```
_DETAILS_PROCESSOR = GenericResultProcessor(
  # Parse nicht erforderlich, AppHost liefert bereits JSON
  # Accumulate nicht erforderlich, keine Pagination
  pattern = r"data[0].{...}"
)
_DETAILS_SPEC = createAppHostSpecBase(
  # Keine zusätzlichen URL-Parameter, Header, etc.
  # Sprache+Land werden später festgelegt
  callback = _DETAILS_PROCESSOR
)
def detailsSpecGen(id,lang='en',country='us',authTok=None):
  # Die Generator-Funktion muss einfach nur den vervollstän-
  # digten Request senden, keine Pagination o.Ä. erforderlich
  def genFunc(spec):
    res = yield spec
    return res
  # Erzeuge AppHost-Generator
  return appHostSpecGen(
    id=id, spec=_DETAILS_PROCESSOR.derive(), # "Kopie"
    genFunc=genFunc, authTok=authTok,
    lang=lang, country=country,
    rsrcPath="/apps/" + str(id)
  )
```

Listing 5.19: Beispiel für die Implementierung von `details_apphost`

Da sich AppHost-Requests sehr ähnlich in ihrem Aufbau sind liefert das `lookup`-Modul noch weitere Hilfsmethoden, z.B. für die URL-Parameter *extend* und *include*. Darauf wird hier aber nicht mehr genauer eingegangen. Für Requests, die nicht an den AppHost-Endpunkt gehen ist die Vorgehensweise genau wie beim Play Store.

5.4 Die REST-API

5.4.1 Request-Handler für die Scraper-Methoden

```python
from scraper_core import playstore as pl
# URL-Parameter
_LANG    = Query('en', description="Language (ISO 639-1)")
_COUNTRY = Query('us', description="Country (ISO 3166-1 alpha-2)")
_TERM    = Query(..., description="The search term")
# Request-Handler
@app.get("/playstore/suggest")
async def suggest(
    term    : str = _TERM,
    lang    : str = _LANG,
    country : str = _COUNTRY
  ):
  return await pl.suggest(term, lang=lang, country=country)
```

Listing 5.20: Beispiel-Request-Handler für die Methode `suggest`

Die REST-API wird minimalistisch gehalten, es findet hier nicht wesentlich viel mehr statt, als dass die entsprechenden Request-Handler definiert werden und der Scraper-Kern zur Extraktion genutzt wird. Z.B. zeigt Listing 5.20 wie die Definition des Request-Handlers für die Methode `suggest` des Play Stores aussehen könnte.

In der praktischen Umsetzung findet sich noch eine modularere Aufgliederung (mittels der Klasse `Router`), aber im Wesentlichen ist die Implementierung der API so gestaltet wie in diesem Listing gezeigt. Die Scraper-Methoden des Play Stores finden sich unter dem Pfad `/playstore` und die des App Stores unter dem Pfad `/appstore`. Unter dem Pfad `/docs` findet sich eine interaktive Dokumentation mit der man die Scraper-Methoden ausprobieren kann.

5.4.2 Fehlerfälle

Für die Fehlerbehandlung lässt sich mittels `app.exception_handler(<CLS>)` ein globaler Handler für Exceptions vom Typ `CLS` installieren. Der Scraper-Kern fasst sämtliche potentiellen Fehler unter der Exception **ScrapeException** zusammen (kann sich allerdings via *Exception Chaining* auf unterschiedliche Ursachen beziehen). Auf diese Art und Weise lassen sich Exceptions vom Scraper-Kern abfangen. Dabei werden die folgenden HTTP-Fehlercodes verwendet:

- **404** falls detektiert werden kann, dass die Fehlerursache die Anfrage an eine nicht existierende Ressource war (z.B. ungültige ID's).
- **400** für anderweitige Fehler, die vom Client verursacht wurden. Der Körper der HTTP-Antwort enthält den Fehlertext.
- **500** für interne Fehler (z.B. I/O-Fehler).
- **422** wird von *FastAPI* verwendet, falls die Parameter-Validierung fehlgeschlagen ist.

5.4.3 Request-Caching und Rate-Limiting

Die REST-API nutzt für das `fetchers`-Modul die Bilibothek `aiohttp` zum Senden von Requests. Dies wird kombiniert mit der `aiocache`-Bibliothek, die zum Caching bei asynchroner I/O verwendet wird. Diese Bibliothek bietet einen Python-Dekorator namens `cached_stampede`, der mit einer *Time To Live (TTL)* Daten cachen kann und der sog. *Cache-Stampede*-Effekte vermeidet (s. [2]).

Zur Umsetzung kann man einfach eine Subklasse der `RequestExecutor`-Klasse des `lib_aiohttp`-Moduls erstellen, die die `_make_request`-Methode überschreibt. Dies sieht dann so aus wie in Listing 5.21 gezeigt.

Standardmäßig nutzt die Bibliothek einfach ein Python Dictionary als Cache. Es wird aber auch eine *Redis*-Datenbank und das *memcached*-System unterstützt. Man

```
from fetchers.lib_aiohttp import RequestExecutor
class CachingRequestExecutor(RequestExecutor):
  @cached_stampede(
    # Bestimmt Dauer der Blockade des Funktionsaufrufs um
    # Stampede-Effekte zu vermeiden. Wir wollen denselben
    # Request nicht zweimal senden -> "inf"
    lease = float('inf'),
    # Time To Live in Sekunden (z.B. 1 Stunde)
    ttl = 3600,
    # Erzeugt den Cache-Schlüssel aus den Funktions-
    # Argumenten (muss String sein)
    key_build = lambda f, self, sess, req: cache_str(req)
  )
  async def _make_request(sess, req):
    return await super()._make_request(sess, req)
```

Listing 5.21: Umsetzung von Caching für die REST-API

könnte die Caching-Funktionen auch so ergänzen, dass der Cache in seiner Größe beschränkt ist und z.B. einen LRU-Algorithmus zur *Cache Eviction* verwendet.

Neben dem Request Caching ist es auch sinnvoll, die Request-Rate zu limitieren. *aiohttp* hat diese Funktionalität nicht direkt eingebaut, es lässt sich aber recht einfach ergänzen: Bei *aiohttp* werden alle Requests über die Klasse `ClientSession` gesendet (der `sess`-Parameter in Listing 5.21 ist eine solche Session). Man kann einfach eine Subklasse schreiben, die eine Limitierung implementiert, ein Beispiel dafür ist in [20] demonstriert.

6

Evaluation

In diesem Kapitel wird die Implementierung hinsichtlich der Erfüllung der in Kapitel 2 untersucht.

Die nachfolgende Tabelle zeigt eine Übersicht über den Erfüllungsgrad der einzelnen Anforderungen. Danach wird noch genauer auf einzelne Anforderungen eingegangen.

Anforderung	*Erfüllungsgrad*
FK01	Vollständig erfüllt.
FK02	Vollständig erfüllt.
FK03	Vollständig erfüllt.
FK04	Vollständig erfüllt.
FK05	Teilweise erfüllt, s.u. für Details.
FR01	Vollständig erfüllt.
FR02	Vollständig erfüllt.
FR03	Vollständig erfüllt.
QK01	Teilweise erfüllt, s.u. für Details.
QK02	Vollständig erfüllt, s.u. für Details.
QR01	Vollständig erfüllt, s.u. für Details.
QR02	Vollständig erfüllt, s.u. für Details.

Robustheit gegenüber Store-Blockaden (FK05)

Das vollständige Erfüllen dieser Anforderung ist schwierig: Für IP-Blockaden gibt es keine direkten Maßnahmen zur Umgehung außer ein IP-Wechsel. Sowas ist durchaus machbar, z.B. über IP-Rotationen, aber dies muss außerhalb des Scrapers umgesetzt werden. Darüberhinaus kann man vor allem die Request-Rate limitieren und diese Funktionalität wird bei der REST-API auch unterstützt. Durch die Modularität des `fetchers`-Modul lässt sich dies auch recht einfach umsetzen.

Bzgl. der Verhinderung von CAPTCHA's oder anderer Autorisierungsanforderungen gibt es keine implementierten Gegenmaßnahmen. Zunächst wäre es erforderlich, diesen Zustand erstmal auszulösen, um untersuchen zu können, wie sich die gelieferten Daten des Stores ändern. Dann müsste man entweder einen automatisierten CAPTCHA-Solver implementieren oder einen Menschen den CAPTCHA lösen lassen. Auch hier wird auf proaktive Maßnahmen durch Request-Caching und Rate-Limiting gesetzt.

Robustheit gegenüber Store-Änderungen (QK01)

Der Scraper kann nicht automatisiert auf Store-Änderungen reagieren. Request-Endpunkte, Extraktionsfunktionen, Kommunikationsablauf, etc. müssen alle manuell festgelegt werden. Eine Store-Änderung wird meistens einen Fehler in einer Verarbeituntsfunktion vom Typ `ScrapeException` hervorrufen (kann aber auch ohne Store-Änderungen passieren, z.B. bei ungültigen ID's, I/O-Fehlern, etc.). Würde man einen komplett automatischen Scraper umsetzen wollen, so müssten grob diese folgenden zwei Aspekte unterschieden werden:

1. Automatische Extraktion von Daten: Hat man z.B. ein HTML-Dokument, eine JSON-Struktur o.Ä. gegeben, so sollen daraus automatisch relevante Daten extrahiert werden. Dies ist nicht einfach umzusetzen, da automatisch detektiert werden müsste, welche Daten relevant sind. Bei JSON-Objekten liefern die Schlüssel-Namen meistens auch die Bedeutung eines Datums (z.B. „id“, „title“, etc.). Hat man aber z.B. ein HTML-Dokument, so müsste man aus dem Text in

HTML-Tags, aus Tag-Attributen, etc. eine Semantik ableiten, um zu entscheiden, ob ein Datum relevant ist. Dies ist eine komplett eigene Problemstellung und wird in dieser Arbeit nicht behandelt, daher hat der Scraper dafür auch keinen Automatismus.

2. Neben der automatischen Datenextraktion müsste ein komplett automatisierter Scraper auch in der Lage sein, das Verhalten der gescrapten Webseite nachzubilden. D.h. er müsste in der Lage sein, URLs zu bestimmen, paginierte Requests in der richtigen Reihenfolge zu senden (und diese auch zu initialen Daten zuordnen können), etc. Dies kann über passives Abfragen von Daten kaum realisiert werden, da hier das Verhalten der Webseite nicht analysiert werden kann. Hier wäre z.B. der Einsatz eines automatisierten Browsers nützlich, bei dem sich eine Webseite so verhält wie bei einem menschlichen Nutzer und es lassen sich so auch „High-Level"-Interaktionen wie das Klicken eines Links, Scrollen, etc. umsetzen. So können URLs analysiert werden, Pagination getriggert werden, etc. und aus den abgefangenen HTTP-Antworten Daten extrahiert werden. Hier würde man dann wieder beim ersten Problem stehen und außerdem müsste auch entschieden werden, welche Aktionen (Klicken eines Links, Scrollen, etc.) überhaupt relevant sind und wie sie im Zusammenhang mit empfangenen Daten stehen. Dies ist ebenfalls eine Problemstellung, die in dieser Arbeit nicht untersucht wird.

Anstatt einer automatischen Anpassung des Scrapers an den Store wird hier auf Wartbarkeit gesetzt. Es gibt im Wesentlichen drei Stufen von Anpassungen, die bei einer Store-Änderung notwendig sein können:

1. Änderungen im Datenformat: Hier muss eigentlich nur die Antwort-Verarbeitung, also das `GenericResultProcessor`-Objekt angepasst werden. Meistens muss nur der JMESPath-Ausdruck etwas abgändert werden, manchmal aber auch die Parsing-Funktion (z.B. wenn die regulären Ausdrücke für die Skript-Tags im Play Store nicht mehr passen). Es ist natürlich nötig, dass das neue Datenformat untersucht wird, damit man diese Änderungen umsetzen kann.

Dazu kann man z.B. die Daten aus einer empfangenen HTTP-Antwort vom Scraper in eine Datei speichern lassen und das Datenformat dann untersuchen.

2. Einfache Endpunkt-Änderungen: Falls sich die URL eines Endpunkts ändert oder z.B. irgendwelche zusätzlichen Header erforderlich sind, so kann man einfach die entsprechenden `RequestSpec`-Definitionen anpassen, um diese Änderungen zu reflektieren.

3. Änderungen im Kommunikationsablauf: Dieser Fall liegt vor, wenn sich der komplette Ablauf der Kommunikation mit einem Store für eine bestimmte Scraper-Methode geändert hat (also z.B. ein Autorisierungs-Request ist nötig oder Pagination funktioniert anders). Dann ist es erforderlich, die entsprechenden `SpecGenerator`-Funktionen anzupassen, die diesen Ablauf implementiert.

Sind die Änderungen wirklich gravierend, so kann es auch nötig sein, eine Scraper-Methode wirklich komplett neu zu implementieren. Dabei kann man sich an dem Beispiel im Implementierungs-Kapitel orientieren oder einfach andere bereits implementierte Methoden als Vorlage nehmen. Mit Hilfe des `common`-Moduls sollte eine Person, die die Arbeitsweise des Scrapers und das Konzept von Python-Generatoren verstanden hat in der Lage sein, eine eigene Scraper-Methode zu implementieren. Für die Untersuchung eines Stores bei Änderungen kann man sich an den Untersuchungen der Stores wie hier gezeigt orientieren.

Dokumentation (QK02 u. QR02)

Die Dokumentation der Implementierung ist überwiegend durch Python *Docstrings* realisiert: Dies sind Strings, die als erster Ausdruck in einer Funktion/Klasse als Literal angegeben werden und die Dokumentation beinhalten. Listing 6.1 zeigt ein Beispiel für einen Docstring (Google-Stil):

Python's *PEP 257* (Python Enhancement Proposal) legt gobe Richtlinien für den Dokumentations-Stil von Python-Programmen fest [8]. Es gibt verschiedene Stile,

```python
from typing import Any, Optional
def test(a : str, b : Any, c: Optional[int] = 10) -> bool:
  """
  Does testing stuff.

  Args:
    a: Parameter a.
    b: Parameter b.
    c: Parameter c.

  Returns:
    The test result.
  """
  pass
```

Listing 6.1: Beispiel: Python-Dokumentation im Google-Stil

die diesen Richtlinen folgen (Google Style, Java Style, etc.), in der Implementierung wurde der Google-Stil verwendet. Die Dokumentation lässt sich mittels des Tools *Sphinx* aus dem Code generieren, mehr Details finden sich bei [1].

Darüberhinaus wurden in der Implementierung Python's *Type Hints* verwendet. Dies sind „Annotationen" für Variablentypen, die allerdings zur Laufzeit irrelevant sind. Mittels statischen Typ-Checkern wie *mypy* lässt sich die Typsicherheit des Programms überprüfen, allerdings hat *mypy* selbst noch einige Bugs.

Für die REST-API findet sich zusätzlich eine interaktive Dokumentation unter dem URL-Pfad `/docs`.

Auslieferung (QR01)

Wie in der Anforderung verlangt gibt es ein Docker Image für die REST-API. Das gesamte Projekt wurde mittels Python's *virtualenv*-Mechanismus umgesetzt womit isolierte Projekt-Umgebungen geschaffen werden können. Dependency Management wurde mit Hilfe von *pipenv* durchgeführt. Das Basis-Image basiert auf `uvicorn-gunicon-fastapi`, unter [22] findet sich eine ausführliche Beschreibung sämtlicher Konfigurationsoptionen.

7

Verwandte Scraper

Es gibt einige Projekte, die ebenfalls eine Implementierung eines Scrapers für den Play Store bzw. den App Store zum Ziel hatten/haben. Die meistens diese Projekte sind veraltet und werden nicht mehr gewartet. Im Nachfolgenden werden ein paar solcher Scraper für den Play Store und den App Store präsentiert und kurz auf die Unterschiede zum hier entwickelten Scraper eingegangen.

Play Store

Name	*Sprache*	*Entwickler*	*Methoden*
play-scraper [17]	Python	Daniel Liu	details, collection, developer, suggestions, search, similar, categories
google-play-scraper [24]	PHP	Raúl Rodríguez	getApp(s), getCategories, getCollections, get(Detail)List(Chunk), get(Detail)Search
google-play-scraper [19]	JavaScript (NodeJS)	Facundo Olano	app, list, search, developer, suggest, reviews, similar, permissions, categories

Tabelle 7.1: Verwandte Play-Store-Scraper

Tabelle 7.1 listet drei ähnliche Scraper für den Google Play Store auf. Der letzte dieser drei Scraper geht ähnlich vor wie der entwickelte Scraper: HTML-Skript-Tags für die initialen Daten und batchexecute-Requests für Pagination. Die ersten beiden Scraper sind beide veraltet und funktionieren großteils nicht mehr richtig. Der Grund dafür ist, dass beim Play Store starke Änderungen vorgenommen wurden, insbesondere für Pagination. Diese war damals noch mit erheblich einfacheren POST-Requests bzw. sogar mit URL-Parametern möglich. Beide extrahieren außerdem die Daten aus dem HTML-Dokument mittels CSS-Selektoren anstatt über JSON-Daten.

Neben diesen drei Scrapern gibt es noch einige weitere, die aber nur einen kleinen Teil des Funktionsumfangs anbieten oder sogar nur minimal geänderte Forks dieser drei Scraper sind.

Alle drei Scraper sind allerdings nicht so modular aufgebaut wie der hier entwickelte Scraper. Bei dem hier entwickelten Scraper sind die Module für den Play Store und den App Store komplett isoliert von einander und greifen auf das *common*-Modul für die Umsetzung ihrer Scraper-Methoden zu, das auch eine Ausgangsbasis für die Entwicklung weiterer Module darstellt. Das *common*-Modul stellt die „Bausteine“ für das Bauen einer Scraper-Methode zur Verfügung während bei diesen drei Scrapern die Logik für eine Scraper-Methode weniger aufgetrennt und daher schwerer anzupassen ist.

Außerdem haben alle drei Scraper ihre Request-Backends „fest eingebaut“, wogegen bei dem hier entwickelten Scraper das Request-Backend über das *fetchers*-Modul einfach geändert werden kann, um Funktionalität zu ergänzen.

App Store

Für den App Store gibt es nur ein Projekt, das man wirklich als vollständingen Scraper für den App Store beschreiben kann: *app-store-scraper* [18] von Facundo Olano, der für NodeJS entwickelt wurde. Die verfügbaren Methoden sind *app*, *list*, *search*, *developer*, *suggest*, *similar*, *reviews* und *ratings*.

Dieser Scraper stellt im Prinzip eine Teilmenge des hier entwickelten Scrapers zur Verfügung. Insbesondere nutzt er überhaupt keine AppHost-Requests. App-Details werden ausschließlich mittels der iTunes-API abgefragt, die deutlich weniger Details liefert. Für Rezensionen, ähnliche Apps, etc. werden RSS-Feeds bzw. WebObjects eingesetzt, die teils langsamer sind als AppHost-Requests bzw. auch weniger umfangreich.

Für der Architektur gilt für diesen Scraper dasselbe wie beim Play Store: Der hier entwickelte Scraper ist durch die *common*- und *fetchers*-Module allgemeiner gehalten und kann leichter angepasst werden.

Neben diesem einen Scraper gibt es noch einige Projekte für einzelne Funktionen wie z.B. das Extrahieren von Rezensionen, App-Details, o.Ä. aber keine vollständigen Scraper für den gesamten App Store.

8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurde ein Web-Scraper für die Extraktion von Daten aus dem Google Play Store und dem Apple App Store entwickelt. Dazu wurden zunächst ausführlich die Stores untersucht. Basierend darauf wurde eine Implementierung eines Webscrapers für diese Stores entwickelt. Dabei wurde die Implementierung so entworfen, dass das Anpassen und Ergänzen von Stores möglich einfach ist. Das *fetchers*-Modul ist die Schnittstelle, die für das Senden und Empfangen von HTTP-Requests verantwortlich ist. Es lässt sich einfach anpassen um gewünschtes Netzwerkverhalten zu erreichen. Das *common*-Modul liefert grundlegende Bausteine, um Scraper-Methoden zu entwickeln und ist der Aspekt, der den entwickelten Scraper am meisten von bereits existierenden Scrapern unterscheidet, die die Funktionalität dieses Moduls als Teil der eigentlichen Scraper-Methoden integriert haben. Sowohl die Ergebnisse der Store-Untersuchungen als auch die Implementierung selbst wurden ausführlich dokumentiert und bieten einen guten Anhaltspunkt für potenzielle Erweiterungen und Ergänzungen des Scrapers.

Zusätzlich wurde eine REST-API entwickelt, die die Scraper-Methoden über eine Web-Schnittstelle zur Verfügung stellt. Dies etabliert Plattform- und Programmiersprachen-Unabhängigkeit und erlaubt es den Scraper auch in weitere Anwendungen einzubinden. Eine interaktive Dokumentation demonstriert schnell und effektiv die Nutzung der API.

8.2 Ausblick

Der hier entwickelte Scraper kann noch in diverser Art und Weise verbessert und erweitert werden. Das *common*-Modul ermöglicht es, Scraper-Methoden aus zur Verfügung gestellten Bausteinen zusammenzusetzen. Anstelle von direkter Nutzung dieser Bausteine in einem Python-Modul könnte man die Spezifikation einer Scraper-Methode auslagern in Spezifikations-Dateien (z.B. im JSON-Format), die es erlauben eine Scraper-Methode ohne Kenntnis der Programmiersprache Python zu definieren. Z.B. könnte so die Definition einer Scraper-Methode aussehen:

```
{
  "name": "details",
  "requests": {
    "init": {
      "host": "play.google.com",
      "path": "/store/apps/details"
      "processor": ...
    }
  },
  "parameters": {
    "appID": {"type": "str"}
  },
  "execute": [
    {"base": "init", "urlParameters": {"id": "$appID"}}
  ]
}
```

Hier legt *name* den Namen der Methode fest, *requests* listet Request-Definitionen auf (können unvollständig sein) und *parameters* definiert Parameter für die Methode. *execute* ist eine Liste von Requests, die gesendet werden sollen, dabei können auch neue Requests abgeleitet werden und dabei die Werte von Methoden-Parametern verwendet werden.

Darüberhinaus könnte man auf Grundlage des Scrapers auch eine Anwendung realisieren, die kontinuierlich Daten aus einem Store extrahiert und diese dann in eine Datenbank einpflegt. So ließe sich z.B. ein Versionsverlauf für App-Details erstellen. Die Anwendung würde dann z.B. eine Liste von beliebten Apps extrahieren und von dort aus dann kaskadierend ähnliche Apps dieser App und dann deren ähnliche Apps, etc. extrahieren. So ließe sich nach und nach eine Datenbank von Apps aus dem Store anlegen, die z.B. für statistische Auswertungen nützlich wäre.

A

Technische Details zum Google Play Store

A.1 URLs zu den HTML-Seiten für statische Daten

Nachfolgend werden für die verschiedenen Funktionen des Play Stores (wie in Abschnitt 3.2.1 definiert) die URL-Pfade angegeben, die zu den HTML-Seiten führen, die die statischen Daten beinhalten. Dazu wird auch der Seiten-Typ angegeben, in Abschnitt A.4 finden sich die zugehörigen Datenformate.

Teilweise ist es nicht möglich, den URL-Pfad direkt anzugeben. Dies ist insbesondere bei einigen Cluster-Übersichten der Fall, da diese nicht systematisch erzeugbare URL-Parameter haben. In diesem Fall wird in zwei Schritten angegegeben, wie man den Pfad erhalten kann.

Bei allen URLs gibt es noch die Möglichkeit, die URL-Parameter **hl** und **gl** anzugeben, die jeweils Sprach- bzw. Länderkürzel angeben (ISO 639-1 bzw. ISO 3166-1 alpha-2).

A.1.1 PL01 - Kategorien auflisten

URL-Pfad:	Irrelevant, muss Präfix `/store/apps` haben.
Seiten-Typ:	Irrelevant
Parameter:	-

A.1.2 PL02 - Cluster-Sammlung/Übersicht

URL-Pfad:	Unterschiedlich
Seiten-Typ:	Cluster-Sammlung/Übersicht
Parameter:	-

A.1.3 PL03 - Apps auflisten

Zunächst wird in Schritt 1 die Cluster-Sammlung extrahiert. Dann wird ein Cluster aus dieser Sammlung selektiert (basierend auf Sammlungs-Typ und evtl. Kategorie) und dann die App-Cards aus diesem extrahiert.

URL-Pfad:	**Schritt 1:** `/store/apps/<coll>` Mit Kategorie (optional): Suffix `/category/<cat>` **Schritt 2**: URL einer Cluster-Übersicht auswählen
Seiten-Typ:	Cluster-Sammlung (Schritt 1)/Cluster-Übersicht (Schritt 2)
Parameter:	• **coll** ist der Sammlungs-Typ. Möglich sind „top“ und „new“ • **cat** ist eine optionale Kategorie-ID, die Ergebnisse auf diese Kategorie einschränkt. S. C.1.4 für eine Liste der Kategorie-IDs.

A.1.4 PL04 - Apps eines Entwicklers auflisten

Es gibt hier zwei URLs: Die erste ist für neue Entwickler-Seiten, welche rein numerische IDs haben und die zweite ist für alte Entwickler-Seiten mit beliebigen Strings als IDs.

URL-Pfad:	`/store/apps/dev?id=<ID>` `/store/apps/developer?id=<ID>`
Seiten-Typ:	Cluster-Übersicht
Parameter:	**ID** ist die Entwickler-ID.

A.1.5 PL05 - Ähnliche Apps

In Schritt 1 wird zunächst über die App-Detail-Seite die URL für den Cluster mit den ähnlichen Apps extrahiert und dann die App-Cards aus diesem extrahiert.

URL-Pfad:	**Schritt 1:** `/store/apps/details?id=<ID>` **Schritt 2**: Cluster-URL aus Detail-Seite
Seiten-Typ:	App-Detail-Seite (Schritt 1)/Cluster-Übersicht (Schritt 2)
Parameter:	**ID** ist die App ID.

A.1.6 PL06 - App-Details

URL-Pfad:	`/store/apps/details?id=<ID>`
Seiten-Typ:	App-Detail-Seite
Parameter:	**ID** ist die App ID.

A.1.7 PL08 - App-Suche

URL-Pfad:	`/store/search?c=apps&q=<q>&price=<p>` `&androidId=<AID>`
Seiten-Typ:	Cluster-Übersicht
Parameter:	• **q** ist der url-kodierte Suchbegriff. • **p** ist optional und filtert Ergebnisse nach dem Preis. Mögliche Werte sind 1 (kostenlos) und 2 (kostenpflichtig). • **AID** ist optional eine ganzzahlige Geräte-ID. Filtert Ergebnisse, sodass nur kompatible Apps geliefert werden. Eine Liste von IDs ist nicht bekannt.

A.1.8 PL10 - Auflisten und Extrahieren von Empfehlungen

Die erste URL führt zu einer Art Cluster-Sammlung mit Listen von „Topic-Cards“ anstatt Cluster-Vorschauen mit App-Cards. Die zweite URL ist für eine konkrete Themenseite.

URL-Pfad:	`/store/apps/editors_choice` (Themen-Liste) `/store/apps/topic?id=<ID>` (Themenseite)
Seiten-Typ:	Cluster-Sammlung/Themenseite
Parameter:	**ID** ist die Themen-ID.

A.2 Eingabe-Format für batchexecute-Requests

Nachfolgend werden für die verschiedenen batchexecute-Requests (s. Abschnitt 3.2.2) die Eingabe-Parameter detailliert erläutert. Hierbei müssen wir die URL-Parameter und die Parameter im Request-Körper betrachten.

A.2.1 URL-Parameter

Name	*Beschreibung*
rpcids	Könnte für *Remote Procedure Call IDs* stehen. Definiert den Request-Typ.
f.sid	Scheint eine Art Sitzungs-ID zu sein, hat aber nicht den gleichen Wert wie der Cookie *SID*, den es ebenfalls gibt. In der HTML-Seite gibt es ein Skript-Tag mit einem Attribut `data-id="_gd"`. Dieses setzt die globale Variable `window.WIZ_global_data` auf ein JSON-Objekt. Der Schlüssel *FdrFJe* enthält den Wert dieses Parameters.

Name	*Beschreibung*
bl	Scheint eine Art Versions-Nummer zu sein, z.B. `20191216.05_p1`. Lässt sich wie auch *f.sid* extrahieren, der Schlüssel ist hier *cfb2h*.
hl	Ein Sprach-Kürzel für die gelieferten Ergebnisse (ISO 639-1).
gl	Ein Länder-Kürzel für die gelieferten Ergebnisse (ISO 3166-1 alpha-2).
soc-app soc-device soc-platform	Unklar, scheinen etwas mit Android-Geräten zu tun zu haben, vllt. zur Filterung von Daten nach Gerät. Diese Parameter können weggelassen werden.
rt	Kann weggelassen werden oder den Wert „c“ haben. In diesem Fall wird die Antwort *chunked* geliefert (s. HTTP/1.1-Header `Transfer-Encoding`, dieser Header existiert in HTTP/2.0 nicht mehr, deshalb dieser URL-Parameter). Der Parameter erschwert das Parsing, daher sollte er Parameter weggelassen werden.

Interessanterweise scheinen diese URL-Parameter allesamt irrelevant für die gelieferten Daten zu sein. Sie können alle weggelassen werden, inklusive `rpcids` und es werden trotzdem die richtigen Daten geliefert. Das liegt daran, dass der Request-Typ ebenfalls im Request-Körper kodiert ist und der URL-Parameter scheinbar ignoriert wird.

A.2.2 Parameter im Request-Körper

Der Request-Körper ist als `application/x-www-form-urlencoded` kodiert, also im selben Format wie URL-Parameter. Zwei Parameter wurden beobachtet:

- **at**: Unklar, was der Zweck von diesem Parameter ist, aber der Wert lässt sich wie beim URL-Parameter **f.sid** erhalten, der Schlüssel ist hier *SNlM0e*.
- **f.req**: Dies ist eine JSON-Struktur, die den eigentlichen Request beinhaltet. Dabei ist dort immer auch der **rpcids**-Parameter enthalten, der den Request-

Typ identifiziert. Wichtig: Die JSON-Struktur wird in einen URL-kodierten String umgewandelt, z.B. wird *[* zu *%5B*.

Der generelle Aufbau des `f.req`-Parameters folgt diesem Schema:

```
[[[
  <rpcids>,<DATA>,null,"generic"
]]]
```

Dabei ist **DATA** die JSON-Struktur, die den eigentlichen Request kodiert, dies aber als String! Z.B. würde die JSON-Struktur `["test"]` als String kodiert `"[\"test\"]"` ergeben (Anführungszeichen müssen escaped werden), mit anschließender URL-Kodierung dann `%22%5B%5C%22test%5C%22%5D%22`.

Nachfolgend werden die JSON-Strukturen für die verschiedenen Request-Typen aufgelistet. Die Antwort ist „fast“ JSON, s. Abschnitt 3.2.2 für die Struktur, die eigentlichen Nutzdaten sind in Abschnitt A.4 angegeben.

A.2.3 Aufbau der Request-JSON-Struktur

w3QCWb (Cluster-Sammlungen)

```
[[
  null,2,null,null,
  [
    [10,[10,<CNT>],null,<TOK>],
    true,null,
    [96,27,4,8,57,30,110,11,16,49,1,3,9,12,104,55,56,51,10,34,31,77],
    [
      null,null,null,
      [[[[7,31],[[1,52,43,112,92,58,69,31,19,96,103]]]]
    ]
  ],
  null,2
]]
```

Variable (Typ)	*Beschreibung*
CNT (int)	Die max. Anzahl an Items, die geliefert werden sollen. Falls der Wert zu groß ist, wird die Antwort zu groß und man erhält einen Fehler. Der Maximalwert ist nicht bekannt, aber bis zu 100 scheint zu funktionieren, der Play Store selbst verwendet 40.
TOK (str)	Das Token, das die nächste Seite für Pagination identifiziert. Die initialen statischen Daten in der HTML-Seite enthalten das erste Token. Ist das Token `null`, so gibt es keine weitere Seite.

qnKhOb (App-Cards)

```
[[
  null,
  [
    [10,[10,<CNT>]],
    true,null,
    [96,27,4,8,57,30,110,79,11,16,49,1,3,9,12,104,55,56,51,10,34,77]
  ],
  null,
  <TOK>
]]
```

Variable (Typ)	*Beschreibung*
CNT (int)	Die max. Anzahl an Items, die geliefert werden sollen. Falls der Wert zu groß ist, wird die Antwort zu groß und man erhält einen Fehler. Der Maximalwert ist nicht bekannt, aber bis zu 100 scheint zu funktionieren, der Play Store selbst verwendet 40.
TOK (str)	Das Token, das die nächste Seite für Pagination identifiziert. Die initialen statischen Daten in der HTML-Seite enthalten das erste Token. Ist das Token `null`, so gibt es keine weitere Seite.

xdSrCf (App-Berechtigungen)

```
[[
  null,
  [<APPID>,7],
  []
]]
```

Variable (Typ)	*Beschreibung*
APPID (str)	Die App ID.

UsvDTd (App-Rezensionen)

```
[
  null,null,
  [
    2,
    <SRT>,
    [<CNT>,null,<TOK>],
    null
    <FILTER>
  ],
  [<APPID>,7]
]
```

Variable (Typ)	*Beschreibung*
SRT (int)	Ganzzahl zwischen 1 und 3, die eine Sortieroption für die gelieferten Rezensionen kodiert. Dabei steht 1 für *relevante Rezensionen*, 2 für *neue Rezensionen* und 3 für *Bewertungen* (d.h. Rezensionen ohne Text, nur Bewertungen). Es kann stattdessen auch `null` verwendet werden, dies entspricht dem Fall, dass der Nutzer keine Sortiertoption ausgewählt hat.

Variable (Typ)	*Beschreibung*
CNT (int)	Die max. Anzahl an Rezensionen, die geliefert werden sollen. Falls der Wert zu groß ist, wird die Antwort zu groß und man erhält einen Fehler. Der Maximalwert ist nicht bekannt, aber bis zu 100 scheint zu funktionieren, der Play Store selbst verwendet 40.
TOK (str)	Das Token, das die nächste Seite für Pagination identifiziert. Hierbei darf auch `null` verwendet werden (im Gegensatz zu den Cluster-Abfragen), in diesem Fall wird die erste Seite der Rezensionen geliefert.
FILTER (Array)	Es gibt zwei mögliche Filteroptionen: Die vom Nutzer abgegebene Bewertung (Ganzzahl zwischen 1 und 5) und ein Geräte-Typ, identifiziert durch einen String. Diese Optionen werden in einen Array im Format `[null,<SCO>,<DEV>]` abgelegt. Dabei ist *SCO* ist die Bewertung und *DEV* das Gerät, beide können `null` sein, falls nicht genutzt. Der Array wird immer auf den ersten nicht-`null`-Wert gekürzt, z.B. `[null,1]`, falls nicht nach dem Geräte-Typ gefiltert wird oder der leere Array, wenn gar kein Filter genutzt wird.
APPID (str)	Die App ID.

UsvDTd (Rezensions-Bearbeitungs-Historie)

```
[
  null,null,
  [4,null,[40]],
  [<APPID>,7],
  <REVID>
]
```

Variable (Typ)	*Beschreibung*
APPID (str)	Die App ID.
REVID (str)	Die Review ID. Diese allein scheint die Rezension nicht zu identifizieren (vllt. nur den Nutzer-Account), daher wird auch die App ID benötigt.

A.3 Abfrage von Suchvorschlägen

Für das Extrahieren von Suchvorschlägen wird kein batchexecute-Request verwendet (auch wenn es einen passenden gibt), sondern es wird ein Request an die folgende URL geschickt:

`https://market.android.com/suggest/SuggRequest?json=1&c=3&q=<TERM>`

Diese URL findet sich häufiger in den vom Play Store gelieferten HTML-Dokumenten, obwohl die Suchleiste selbst sie nicht verwendet. **TERM** ist der url-kodierte Suchbegriff, welcher vervollständigt werden soll. Die anderen beiden Parameter geben an, dass die Antwort als JSON geliefert werden soll und die Ergebnisse Apps sein sollen. Die URL-Parameter **hl** und **gl** wurden auch hier wieder weggelassen und geben Sprach- und Länderkürzel an (genau wie zuvor auch schon). Das Antwortformat findet sich in Abschnitt A.4.15.

A.4 Struktur von Play-Store-Antwort-Daten

Die extrahierten Daten liegen im JSON-Format vor. Beim Play Store sind diese allerdings sehr ungewöhnlich formatiert: Die Struktur wird nur mittels Json-Arrays aufgebaut, es kommen keine JSON-Objekte vor. Das lässt die Struktur sehr verschachtelt und unsauber wirken und es gibt keine Schlüsselwörter, die die Bedeutung eines Datums verraten könnten.

Die nachfolgenden Listings demonstrieren den Aufbau der JSON-Daten. Dabei wird "..." verwendet, um die Auslassung irrelevanter Teile oder eine variable Anzahl an Wiederholungen anzudeuten. Gewinkelte Klammern (z.B. `<TEST>`) sind ein Platzhalter für konkrete Daten.

A.4.1 Struktur einer Cluster-Sammlung/Übersicht

```
[[
  null,
  [<CLUSTER/TOPIC>, <CLUSTER/TOPIC>, ..., <CLUSTER/TOPIC>],
  null, [null, <TOKEN>], null
]]
```

Variable (Typ)	*Beschreibung*
CLUSTER/TOPIC (Array)	Bei Cluster-Sammlungen gibt für jede Cluster-Vorschau und Themenliste einen solchen Wert und bei Cluster-Übersichten nur einen für das zugehörige Cluster. Die enthaltenen Daten beschreiben ein Cluster oder eine Themenliste, der Aufbau ist jeweils in Abschnitt A.4.2 bzw. Abschnitt A.4.3 erläutert.
TOKEN (str)	Das Token zur Abfrage der nächsten Seite von Clustern. Fehlt oder ist `null`, falls es keine weitere Seite gibt.

A.4.2 Struktur eines einzelnen Clusters

```
[
  [
    [<APP>, <APP>, ..., <APP>],
    <TITLE>,
    null,
    [null, null, null, null, [null, null, <PATH>]],
```

```
    true, 2, null,
    [null, <TOKEN>],
    ...
  ],
  ...
]
```

Variable (Typ)	*Beschreibung*
APP (Array)	Eine JSON-Struktur für eine einzelne App-Cards des Clusters. Der Aufbau ist in Abschnitt A.4.4 demonstriert.
TITLE (str)	Der Name des Clusters.
PATH (str)	Der URL-Pfad des Clusters (führt zur Cluster-Übersicht).
TOKEN (str)	Das Token zur Abfrage der nächsten Seite von App-Cards. Fehlt oder ist `null`, falls es keine weitere Seite gibt.

A.4.3 Struktur einer Themenliste

```
[
  null, null, null, [...], null, null, null,
  [
    <TITLE>,
    [<TOPIC>, <TOPIC>, ..., <TOPIC>],
    [null, null, null, null, [null, null, <EC-LINK>]]
  ],
  ...
]
```

Variable (Typ)	*Beschreibung*
TITLE (str)	Der Titel der Themenliste (scheint immer „Neues aus der Redaktion“ zu sein).
EC-LINK (str)	Der URL-Pfad zur Seite der Redaktionsempfehlungen (scheint immer `/store/apps/editors_choice` zu sein).

Variable (Typ)	*Beschreibung*
TOPIC (Array)	Beschreibt eine einzelne „Topic-Card“ (s. Abschnitt A.4.5).

A.4.4 Struktur einer App-Card

```
[
  null,
  [null, [<IMG>, <IMG>, <IMG>], ...],
  <TITLE>, null,
  [
    [<DEV-DATA>],
    [null, [null, [null, <DESC>]]]
  ],
  null,
  [[
    null, null,
    [null, [<SCO-TXT>, <SCO-VAL>]]
  ]],
  [[
    null, null, null,
    [
      null, null,
      <PRICE-DATA>,
      [<APP-ID>, 7]
    ]
  ]],
  ...,
  [<APP-ID>, 7]
]
```

Variable (Typ)	*Beschreibung*
IMG (Array)	Beschreibt ein Icon der App. Es scheint sich hier drei mal um dasselbe Icon in verschiedenen Auflösungen zu handeln. Die Struktur des Arrays, der ein Icon beschreibt ist in Abschnitt A.4.11 gegeben.
DEV-DATA (Array)	Beschreibt Informationen über den Entwickler, s. Abschnitt A.4.10 für die Struktur.
DESC (str)	Die Kurzbeschreibung (mehr eine Art Untertitel) für die App.
SCO-TXT (str) SCO-VAL (float)	Die Durchschnittswertung der App, einmal als formatierter String (eine Nachkommastelle) und einmal als nicht gerundete Gleitkommazahl.
PRICE-DATA (Array)	Beschreibt Preis-Informationen der App, s. Abschnitt A.4.9 für die Struktur.
APP-ID (str)	Die App ID.

A.4.5 Struktur einer Topic-Card

```
[
  null,
  <TOPIC-IMG>,
  [<APP-IMG>, <APP-IMG>, ..., <APP-IMG>],
  <TITLE>,
  [null, null, null, null, [null, null, <PATH>]],
  ...
]
```

Variable (Typ)	*Beschreibung*
TOPIC-IMG (Array)	Beschreibt das Icon des Themas (s. Abschnitt A.4.11 für die Struktur).

Variable (Typ)	*Beschreibung*
APP-IMG (Array)	Icons für die Apps, die zu diesem Thema gehören (Format s. Abschnitt A.4.11)
TITLE	Der Titel des Themas.
PATH	Der URL-Pfad zur Themenseite, aus dem auch die Themen-ID extrahiert werden kann (URL-Parameter *id*).

A.4.6 Struktur der Kategorie-Liste

```
[[
  null,
  [
    [
      2, null, <CAT-STR>,
      [<CAT>, <CAT>, ..., <CAT>]
    ],
    ...
  ],
  ...
]]
```

Variable (Typ)	*Beschreibung*
CAT (Array)	Beschreibt eine einzelne *Kategorie*. Eine Kategorie kann Unterkategorien haben. Dies äußert sich so, dass eine Kategorie-Struktur rekursiv für Unterkategorien aufgebaut ist. Im Dropdown-Menü ist dies so organsisiert, das jede Top-Level-Kategorie eigene Spalte hat, die Überschrift ist für die Top-Level-Kategorie. Nach einer Trennlinie folgen dann die Unterkategorien, man hat also nur eine Tiefe von 2 Ebenen für Kategorien. Die Struktur einer Kategorie im nachfolgenden Listing dargestellt.
CAT-STR (str)	Der übersetzte text „Kategorien“ (Titel des Drop-Down-Menüs).

```
[
  5, [<CAT-LINK>, <CAT-NAME>, ...], <CAT-NAME>,
  [<CAT>, ..., <CAT>]
]
```

Variable (Typ)	*Beschreibung*
CAT-LINK (str)	Der URL-Pfad zu der Kategorie-Übersichtseite. Die letzte Pfadkomponente ist die Kategorie-ID.
CAT-NAME (str)	Der übersetzte Name der Kategorie, es ist unklar, warum dieser zwei mal vorkommt.
CAT (Array)	Eine Unterkategorie, die nach demselben Schema aufgebaut ist. Der so erzeugte Kategorie-Baum hat maximal Tiefe 2 (Top-Level-Kategorien und Unterkategorien).

A.4.7 Struktur einer Themenseite

```
[
  [<TITLE>, null, <DESC>, <BANNER-IMG>],
  [null, [<TOPIC-APP>, <TOPIC-APP>, ..., <TOPIC-APP>], ...],
  ...
]
```

Variable (Typ)	*Beschreibung*
TITLE (str)	Der Titel des Themas.
DESC (str)	Die Beschreibung des Themas.
BANNER-IMG (Array)	Beschreibt das Banner-Bild des Themas (Struktur s. Abschnitt A.4.11).
TOPIC-APP (Array)	Beschreibt eine App des Themas. Die Struktur ist im nachfolgenden Listing gezeigt.

```
[
  null, null, null, [...], null, null, null, null,
  null, null, null, null, null, null, null,
  [
    [<APP-ID>, 7],
    <TITLE>,
    [
      [...], <DESC>, null,
      [[<REASON>, ..., <REASON>], <LIKE-TXT>]
    ],
    <DEV-DATA>,
    <APP-ICON>,
    [<SCREENSHOT>, <SCREENSHOT>, ..., <SCREENSHOT>],
    [
      [null, null, <YT-ID>, [null, null, <YT-EMBED>], ...],
      <YT-THUMBNAIL>
    ],
    [[
      <CAT-NAME>,
      [null, null, null, null, [null, null, <CAT-LINK>]],
      <CAT-ID>
    ]],
    [
      [[[<PRICE-DATA>]], ...]],
      [[[<PRICE-DATA>]], ...]]
    ],
    [...],
    [<RATINGS-TXT>, <RATINGS-VAL>],
    [<SCO-TXT>, <SCO-VAL>],
    <AGE>,
    [<INSTALLS-TXT>, <INSTALLS-MIN>, <INSTALLS-EXACT>],
    ...
  ]
]
```

Variable (Typ)	*Beschreibung*
APP-ID (str)	Die App ID.
TITLE (str)	Der Titel der App.
DESC (str)	Die Beschreibung der App. Gibt es keine Begründungen für die Empfehlung (s. *REASON*), so wird die Beschreibung angezeigt. Andernfalls nicht und der Wert ist der leere String.
LIKE-TXT (str)	Der übersetzte Text „Was uns daran gefällt“.
REASON (str)	Ein Grund für die Redaktionsempfehlung. Es kann sein, dass es keine Begründungen gibt, dann ist der Array mit den Gründen leer.
DEV-DATA (Array)	Beschreibt Informationen über den Entwickler, s. Abschnitt A.4.10 für die Struktur.
APP-ICON (Array)	Beschreibt ein Icon für die App, die Struktur wird in Abschnitt A.4.11 erläutert.
SCREENSHOT (Array)	Ein Screenshot der App, das Format ist dasselbe wie für das App-Icon.
YT-ID (str) YT-EMBED (str) YT-THUMBNAIL (Array)	Geben jeweils Youtube-Video-ID, Youtube-Einbettungs-URL und Thumbnail für ein Video über die App an. Der Array für das Thumbnail hat denselben Aufbau wie alle Bild-Daten (s. Abschnitt A.4.11).
PRICE-DATA (Array)	Beschreibt den Preis der App, s. Abschnitt A.4.9 für die Struktur. Warum es diese Daten zweimal gibt, ist unklar, denn die Informationen scheinen dieselben zu sein.
RATINGS-TXT (str) RATINGS-VAL (int)	Geben jeweils die Anzahl der App-Bewertungen als formatierter Text und als Ganzzahl an.
SCO-TXT (str) SCO-VAL (float)	Geben jeweils die Durchschnittswertung der App als formatierter Text (eine Nachkommastelle) und als Kommazahl an.
AGE (Array)	Beschreibt die Alters-Einstufung der App (z.B. USK). Der Aufbau ist in Abschnitt A.4.12 gegeben.

Variable (Typ)	*Beschreibung*
INSTALLS-TXT (str) INSTALLS-MIN (int) INSTALLS-EXACT (int)	Geben die Anzahl der Installationen der App an, jeweils als formatierter Text (Tausender-Trennzeichen landesabhängig), abgerundete Ganzzahl und ungerundete Ganzzahl.

A.4.8 Struktur von App-Details

Schlüssel *ds:12*

```
[[
  [...],
  [
    null, [...], [...], null, null, null, null, null,
    null, null, null, [[<APP-ID>], 7]
  ],
  ...
]]
```

Variable (Typ)	*Beschreibung*
APP-ID (str)	Die App ID.

Schlüssel *ds:8*

```
[<SIZE>, <APP-VERSION>, <OS-VERSION>]
```

Variable (Typ)	*Beschreibung*
SIZE (str)	Die Größe der App als menschenlesbarer Text (z.B. „94M“). Kann den Tet „Variiert je nach Gerät“ beinhalten.
APP-VERSION (str)	Die App-Version (z.B. „1.127“).

Variable (Typ)	*Beschreibung*
OS-VERSION (str)	Die minimale Betriebssystem-Version als menschenlesbarer Text (z.B. „4.1 oder höher“). Kann ebenfalls den Text „Variiert je nach Gerät“ beinhalten.

Schlüssel *ds:3*

```
[[
  null, null,
  [[[<PRICE-DATA>], ...]],
  ...
]]
```

Variable (Typ)	*Beschreibung*
PRICE-DATA (Array)	Beschreibt den Preis der App, s. Abschnitt A.4.9.

Schlüssel *ds:6*

```
[
  [
    null, null, null, null, [...], [...],
    [
      [<SCO-TXT>, <SCO-VAL>],
      [
        null,
        [<CNT-TXT-1>, <CNT-VAL-1>],
        [<CNT-TXT-2>, <CNT-VAL-2>],
        [<CNT-TXT-3>, <CNT-VAL-3>],
        [<CNT-TXT-4>, <CNT-VAL-4>],
        [<CNT-TXT-5>, <CNT-VAL-5>]
      ],
      [<RAT-TXT>, <RAT-VAL>],
      [<REV-TXT>, <REV-VAL>]
```

```
    ]
  ],
  ...
]
```

Variable (Typ)	*Beschreibung*
REV-TXT (str) REV-VAL (int)	Geben jeweils die Anzahl der App-Rezensionen als formatierter Text und als Ganzzahl an.
RAT-TXT (str) RAT-VAL (int)	Geben jeweils die Anzahl der App-Bewertungen als formatierter Text und als Ganzzahl an.
SCO-TXT (str) SCO-VAL (float)	Geben jeweils die Durchschnittswertung der App als formatierter Text (eine Nachkommastelle) und als Kommazahl an.
CNT-TXT (str) CNT-VAL (float)	Diese Werte bilden zusammen das Histogramm (in auftretender Reihenfolge für die Bewertungen 1 bis 5). Dabei wird jeweils die Anzahl der Bewertungen als formatierter Text und als Ganzzahl angegeben.

Schlüssel *ds:7*

```
[null, [null, [<CLUSTER>]]]
```

Variable (Typ)	*Beschreibung*
CLUSTER (Array)	Die Cluster-Vorschau für die ähnlichen Apps, s. Abschnitt A.4.2 für die Struktur.

Schlüssel *ds:5*

```
[[
  [<TITLE>],
  [], [], [1], null, [...], null, null, null, null,
  [[null, <DESC>], [null, <SUMMARY>]],
```

```
[
  [<SCREENSHOT>, <SCREENSHOT>, ..., <SCREENSHOT>],
  <APP-ICON>, <APP-BANNER>,
  [
    [null, null, <YT-ID>, [null, null, <YT-EMBED>], ...],
    <YT-THUMBNAIL>
  ],
  <AGE>,
  [
    [<DEV-ID-INTERNAL>, 8],
    <DEV-NAME>,
    [<SUPPORT-MAIL>],
    [
      null, null, null, null, null,
      [null, null, <DEV-WEBSITE>]
    ],
    [<ADDRESS>],
    [
      null, null, null, null,
      [null, null, <DEV-PLAY>
    ]
  ],
  [null, <CHANGELOG>],
  [null, null, <PRIVACY-LINK>],
  [<UPDATE-DATE>, <UPDATE-NANOS>],
  [<INSTALLS-TXT>, <INSTALLS-MIN>, <INSTALLS-EXACT>],
  null, [2],
  [<IN-APP-PURCH>],
  <CAT-DATA>, <ADS-DATA>, <EC-DATA>,
  null, null,
  <WOS-DATA>,
  true, null, false, false,
  <FAM-DATA>,
  null,
  <CAT-ID>,
  null, null, 1, null, null,
  [<ADS-TXT>],
  [...],
```

```
    [<OFFERED-BY>],
    <RELEASE-DATE>
  ],
  ...
]]
```

Variable (Typ)	*Beschreibung*
TITLE (str)	Der Titel der App.
DESC (str)	Die vollständige Beschreibung der App. Kann HTML-*br*-Tags für Zeilenumbrüche beinhalten.
SUMMARY (str)	Eine kurze Zusammenfassung der App.
APP-ICON (Array)	Beschreibt ein Icon der App, s. Abschnitt A.4.11 für die Struktur.
APP-BANNER (Array)	Beschreibt ein Banner-Bild der App, s. Abschnitt A.4.11 für die Struktur.
SCREENSHOT (Array)	Beschreibt ein Screenshot der App, s. Abschnitt A.4.11 für die Struktur.
YT-ID (str) YT-EMBED (str) YT-THUMBNAIL (Array)	Geben jeweils Youtube-Video-ID, Youtube-Einbettungs-URL und Thumbnail für ein Video über die App an. Der Array für das Thumbnail hat denselben Aufbau wie alle Bild-Daten (s. Abschnitt A.4.11).
AGE (Array)	Beschreibt die Alters-Einstufung der App (z.B. USK). Der Aufbau ist in Abschnitt A.4.12 gegeben.
INSTALLS-TXT (str) INSTALLS-MIN (int) INSTALLS-EXACT (int)	Geben die Anzahl der Installationen der App an, jeweils als formatierter Text (Tausender-Trennzeichen landesabhängig), abgerundete Ganzzahl und ungerundete Ganzzahl.
DEV-ID-INTERNAL (str)	Eine interne ID des Entwicklers, der Zweck ist unklar und sie scheint auch sonst keinen Nutzen zu haben.
DEV-PLAY (str)	Der URL-Pfad zur App-Übersichtsseite des Entwicklers. Aus dem URL-Parameter *id* kann die Entwickler-ID extrahiert werden.

Variable (Typ)	*Beschreibung*
DEV-NAME (str) SUPPORT-MAIL (str) DEV-WEBSITE (str) ADDRESS (str)	Geben jeweils Name, Support-Email-Addresse, Website-URL und Impressum des Entwicklers an.
CHANGELOG	Ein Text, der kürzliche Änderungen der App beinhaltet. Kann wie die Beschreibung HTML-*br*-Tags beinhalten.
PRIVACY-LINK	Eine URL, die zu den Datenschutzbestimmungen der App führt.
UPDATE-DATE (int) UPDATE-NANOS (int)	Geben den Zeitpunkt des letzten Updates an, einmal auf Sekunden genau (Unix-Timestamp) und einmal auf Nanosekunden genau (die letzten 6 Stellen scheinen aber immer 0 zu sein, also auf Millisekunden genau).
INSTALLS-TXT (str) INSTALLS-MIN (int) INSTALLS-EXACT (int)	Geben die Anzahl der Installationen der App an, jeweils als formatierter Text (Tausender-Trennzeichen landesabhängig), abgerundete Ganzzahl und ungerundete Ganzzahl.
IN-APP-PURCH (str)	Ein Text, der den Preisbereich von In-App-Käufen beschreibt. Falls es diese nicht gibt, so ist der Wert stattdessen `null`.
CAT-DATA (Array)	Beinhaltet zusätzliche Infos zur Kategorie der App. Die Kategorie-ID ist aber auch durch *CAT-ID* gegeben, sodass dieser Wert nicht relevant ist.
ADS-DATA (Array) EC-DATA (Array)	Liefern Daten über Werbung in der App und Status bzgl. Redaktionsempfehlungen. Der Aufbau ist nicht relevant, nur ob sie nicht `null` sind. Dies ist der Fall, wenn es keine Werbung gibt bzw. die App nicht von der Redaktion empfohlen ist.
WOS-DATA (Array) FAM-DATA (Array)	Wie *ADS-DATA* und *EC-DATA* aber bzgl. Unterstützung von *WearOS* und der *Familienbibliothek*.
CAT-ID (str)	Die ID der Kategorie, zu der die App gehört.

Variable (Typ)	*Beschreibung*
ADS-TXT (str)	Ist `null`, wenn die App keine Werbung beinhaltet, ansonsten der sprach-angepasste Text „Enthält Werbung“.
OFFERED-BY (str)	Gibt den Namen des Unternehmens an, das die App anbietet. Scheint immer „Google Commerce Ltd.“ zu sein.
RELEASE-DATE (str)	Das Veröffentlichungsdatum der App (Format sprach-abhängig).

A.4.9 Struktur von Preis-Informationen

```
[
  null,
  [<PRICE>, <PRICE-ORIG>],
  null, [...], null, 2,
  [
    null, null, null, null, null,
    [null, null, <PURCH-LINK>]
  ],
  null, [...], null, null, null, null, null,
  [[<OFFER-END>], <END-TXT>],
  null, []
]
```

Variable (Typ)	*Beschreibung*
PRICE (Array) PRICE-ORIG (Array)	Geben jeweils den aktuellen Preis und den Originalpreis an (nur, falls die App im Angebot ist, sonst existiert der entsprechende Index gar nicht). Ein Preis-Array hat drei Einträge in dieser Reihenfolge: Den Preis als Ganzzahl multipliziert mit 1 000 000 (z.B. 1790000 für 1,79), den Preis als formatierten Text (Format landes-abhängig, z.B. “$1,.79”) und die Währungs-ID (ISO 4217).

Variable (Typ)	*Beschreibung*
PURCH-LINK (str)	Eine URL, die direkt zum Kaufen/Installieren-Dialog führt.
OFFER-END (int) END-TXT (str)	Ein Unix-Timestamp für das Ende des Angebots und ein Text, der die Restdauer beschreibt (z.B. „noch 1 Tag verbleibend“). Falls kein Angebot vorliegt, so ist statt dem umschließenden Array der Wert `null` vorzufinden.

A.4.10 Struktur von Entwickler-Informationen

```
[
  <DEV-NAME>,
  [
    null, null, null, null,
    [null, null, <DEV-PATH>]
  ],
  ...
]
```

Variable (Typ)	*Beschreibung*
DEV-NAME (str)	Der Name des Entwicklers.
DEV-PATH (str)	Der URL-Pfad zur App-Übersichtsseite des Entwicklers. Aus ihm kann die Entwickler-ID extrahiert werden (URL-Parameter `id`).

A.4.11 Struktur von Bild-Referenzen

```
[
  null, 2,
  [<WIDTH>, <HEIGHT>],
  [null, null, <IMG-URL>]
  ...
```

```
]
```

Variable (Typ)	*Beschreibung*
WIDTH (int) HEIGHT (int)	Die Dimensionen des Bilds. Manchmal ist statt dem Array auch `null` zu finden, dann fehlen die Dimensionen.
IMG-URL (str)	Die URL zu dem Bild.

A.4.12 Struktur von Alters-Einstufungen

```
[
  <AGE-TXT>,
  <AGE-ICON>,
  [null, <AGE-REASONS>],
  [null, <INTERACTIVE-ELEMS>],
  ...
]
```

Variable (Typ)	*Beschreibung*
AGE-TXT (str)	Der Text der Alterseinstufung (z.B. „USK ab 6 Jahren“).
AGE-ICON (Array)	Beschreibt das Icon für die Alterseinstufung (z.B. die USK-Icons), für das Format s. Abschnitt A.4.11.
AGE-REASONS (str)	Ein Text, der Gründe (komma-getrennt) für die Einstufung liefert (z.B. „Abstrakte Gewalt“).
INTERACTIVE-ELEMS (str)	Ein Text, der die relevanten Aspekte der Nutzerunteraktion beschreibt (z.B. „Onlinekäufe“).

A.4.13 Struktur von App-Berechtigungen

```
[
```

```
  [<PERM-GRP>, <PERM-GRP>, ..., <PERM-GRP>],
  [<PERM-GRP>],
  [
    [null, <PERM-TEXT>],
    ...,
    [null, <PERM-TEXT>]
  ]
]
```

Variable (Typ)	*Beschreibung*
PERM-GRP (Array)	Eine Berechtigungsgruppe, diese umfasst Daten für eine Gruppe von Berechtigungen, s.u. für die Struktur. Der erste Array beinhaltet Gruppen zu normalen Berechtigungen und der zweite nur eine Gruppe mit sonstigen Berechtigungen.
PERM-TEXT (str)	Diese Werte sind Beschreibungen jeweils einer einzelnen Berechtigung, in diesem Fall für ungruppierte Berechtigungen (gehören nicht zu einer Berechtigungsgruppe). Der äußere Array kann auch komplett fehlen, dann gibt es keine ungruppierten Berechtigungen.

```
[
  <GRP-NAME>,
  [null, 2, null, [null, null, <GRP-ICON>]],
  [
    [null, <PERM-TEXT>],
    ...,
    [null, <PERM-TEXT>]
  ]
]
```

Variable (Typ)	*Beschreibung*
GRP-NAME (str)	Der Name der Berechtigungsgruppe.

Variable (Typ)	*Beschreibung*
GRP-ICON (Array)	Beschreibt das Icon der Gruppe, s. Abschnitt A.4.11 für die Struktur.
PERM-TEXT (str)	Diese Werte sind Beschreibungen jeweils einer einzelnen Berechtigung der Gruppe.

A.4.14 Struktur von App-Rezensionen

```
[
  [<REVIEW>, <REVIEW>, ..., <REVIEW>],
  [null, <TOKEN>]
]
```

Variable (Typ)	*Beschreibung*
REVIEW (Array)	Beschreibt eine einzelne Rezension/Bewertung.
TOKEN (str)	Das Token zur Abfrage der nächsten Seite von Rezensionen, `null` falls es keine weiteren gibt.

```
[
  <REVIEW-ID>,
  [<USER-NAME>, <USER-ICON>],
  <SCORE>, null,
  <REVIEW-TXT>, [<TIME>, <NANOS>], <LIKES>
  [<DEV-NAME>, <RESPONSE-TXT>, [<TIME>, <NANOS>]],
  null, [...],
  <VERSION>,
  null,
  [[
    [<CRIT>, [<RATING>]],
    ...,
    [<CRIT>, [<RATING>]],
  ]],
```

```
  null,
  <EDIT-DATA>
]
```

Variable (Typ)	*Beschreibung*
REVIEW-ID (str)	Die ID der Rezension.
USER-NAME (str)	Der Nutzername.
USER-ICON (Array)	Beschreibt das Avatar-Icon des Nutzers, s. Abschnitt A.4.11 für die Struktur.
REVIEW-TXT (str)	Der eigentliche Text der Rezension, `null` falls nur eine Bewertung abgegeben wurde.
LIKES (int)	Die Anzahl an „Daumen hoch“ für diese Rezension.
DEV-NAME (str) RESPONSE-TXT (str)	Name und Antworttext des Entwicklers auf diese Rezension. Falls es keine Antwort gibt, so fehlt der umschließende Array und es ist dort stattdessen der Wert `null` zu finden.
TIME (int) NANOS (int)	Der Zeitpunkt der Rezension als Unix-Timestamp, einmal auf die Sekunde genau und einmal auf die Nanosekunde genau (wobei die letzten 6 Stellen immer 0 zu sein scheinen, also eher Millisekunden).
VERSION (str)	Die Version der App, für die die Rezension geschrieben wurde.
CRIT (str) RATING (int)	Angaben zu Bewertungen des Nutzers bzgl. der App. Je nach Kategorie bietet der Play Store beim Abgeben einer Rezension die Möglichkeit, eine App nach bestimmten Kriterien mit einer Auswahl möglicher Antworten zu bewerten. Z.B. gibt es Fragen wie „Bietet diese App nützliche Benachrichtigungen“ mit den Antwortmöglichkeiten „Nein“, „Weiß nicht“ und „Ja“. *CRIT* ist die ID eines Kriteriums (z.B. *vaf_never_display_disruptive_ads*) und *RATING* die Antwort als Zahl kodiert.

Variable (Typ)	*Beschreibung*
EDIT-DATA (Array)	Beschreibt Optionen für das Options-Menü einer Rezension, spezifisch ob die Rezension bearbeitet wurde. Wenn dies nicht der Fall ist, ist hier einfach nur der Wert `null` vorzufinden.

A.4.15 Struktur von Suchvorschlägen

```
[
  {"t": "q", "s": <TERM>},
  ...,
  {"t": "q", "s": <TERM>}
]
```

Variable (Typ)	*Beschreibung*
TERM (str)	Ein vervollständigter Suchbegriff.

A.5 Konstanten

A.5.1 Sammlungs-IDs

Es gibt die folgenden Typen von Sammlungen:

TOP_FREE	**TOP_PAID**	**TOP_GROSSING**
TRENDING	**TOP_FREE_GAMES**	**TOP_PAID_GAMES**
TOP_GROSSING_GAMES	**NEW_FREE**	**NEW_PAID**
NEW_FREE_GAMES	**NEW_PAID_GAMES**	

Es gibt zwei Gruppen von Sammlungen: *Top*-Sammlungen und *New*-Sammlungen. Der URL-Parameter **coll** wie in Abschnitt A.1.3 beschrieben kann auch nur die Werte

`new` und `top` annehmen. Die hier genannten Sammlungs-Typen entsprechen den Typen von Clustern, die man auf solchen Seiten auffinden kann. Je nach Anzahl der Cluster auf einer Sammlungs-Seite finden sich unterschiedliche Typen von Clustern in unterschiedlichen Reihenfolgen. Die gezeigten Typen hängen aber nur von der Anzahl der Cluster und *top* oder *new* ab: Es gibt die folgenden Typen von Sammlungen:

N	*Typ*	*Reihenfolge*
2	top	**TOP_FREE**, **TOP_PAID**
3	top	**TOP_FREE**, **TOP_PAID**, **TOP_GROSSING**
4	top	**TOP_FREE**, **TOP_GROSSING**, **TRENDING**, **TOP_PAID**
6	top	**TOP_FREE**, **TOP_PAID**, **TOP_GROSSING**, **TOP_FREE_GAMES**, **TOP_PAID_GAMES**, **TOP_GROSSING_GAMES**
2	new	**NEW_FREE**, **NEW_PAID**
4	new	**NEW_FREE**, **NEW_PAID**, **NEW_FREE_GAMES**, **NEW_PAID_GAMES**

A.5.2 Kategorie-IDs

Es gibt die folgenden Kategorie-IDs:

APPLICATION	**ANDROID_WEAR**	**ART_AND_DESIGN**
AUTO_AND_VEHICLES	**BEAUTY**	**BOOKS_AND_REFERENCE**
BUSINESS	**COMICS**	**COMMUNICATION**
DATING	**EDUCATION**	**ENTERTAINMENT**
EVENTS	**FINANCE**	**FOOD_AND_DRINK**
HEALTH_AND_FITNESS	**HOUSE_AND_HOME**	**LIBRARIES_AND_DEMO**
LIFESTYLE	**MAPS_AND_NAVIGATION**	**MEDICAL**
MUSIC_AND_AUDIO	**NEWS_AND_MAGAZINES**	**PARENTING**
PERSONALIZATION	**PHOTOGRAPHY**	**PRODUCTIVITY**
SHOPPING	**SOCIAL**	**SPORTS**
TOOLS	**TRAVEL_AND_LOCAL**	**VIDEO_PLAYERS**
WEATHER	**GAME**	**GAME_ACTION**
GAME_ADVENTURE	**GAME_ARCADE**	**GAME_BOARD**

GAME_CARD	**GAME_CASINO**	**GAME_CASUAL**
GAME_EDUCATIONAL	**GAME_MUSIC**	**GAME_PUZZLE**
GAME_RACING	**GAME_ROLE_PLAYING**	**GAME_SIMULATION**
GAME_SPORTS	**GAME_STRATEGY**	**GAME_TRIVIA**
GAME_WORD	**FAMILY**	**FAMILY_ACTION**
FAMILY_BRAINGAMES	**FAMILY_CREATE**	**FAMILY_EDUCATION**
FAMILY_MUSICVIDEO	**FAMILY_PRETEND**	

Es ist außerdem anzumerken, dass für *FAMILY*-Kategorien der URL-Parameter `age` angegeben werden kann, der die Apps auf eine Altersklasse einschränkt. Mögliche Werte sind **AGE_RANGE1** bis **AGE_RANGE3**, die für die Altersklassen „bis 5 Jahre“, „6 bis 8 Jahre“ und „ab 8 Jahren“ stehen.

B

Technische Details zum Apple App Store

B.1 Parameter und Datenformat für AppHost-Requests

B.1.1 URL-Parameter

Dieser Abschnitt ist eine ausführliche Beschreibungen der möglichen URL-Parameter für AppHost-Requests:

- **Der URL-Parameter *platform***
  Dieser URL-Parameter ist für alle AppHost-Requests erforderlich und scheint die Plattform anzugeben, für die die Daten geliefert werden sollen. Der einzig bekannte und relevante Wert ist **web**.
- **Der URL-Parameter *additionalPlatforms***
  Dieser URL-Parameter ist optional und bei allen AppHost-Requests möglich. Der Wert ist eine komma-getrennte Liste von Geräten, die beim Liefern von Apps berücksichtigt werden sollen (d.h. es werden Apps geliefert, die auf diesen Plattformen verfügbar sind). Bekannte Werte sind **appletv**, **ipad**, **iphone** und **mac**.
- **Der URL-Parameter *l***
  Dieser URL-Parameter ist optional und bei allen AppHost-Requests möglich.

Er gibt die Sprache und das Land der gelieferten Daten an. Der Wert muss ein Sprach-Kürzel (ISO 639-1) und ein Länder-Kürzel (ISO 3166-1 alpha-2) beinhalten, beide durch ein Bindestrich getrennt, z.B. **en-us** oder **de-de**.

- **Der URL-Parameter *ids***
  Dieser URL-Parameter ist nur relevant, wenn man mehrere IDs zu einem Ressourcen-Typ abfragen möchte (s. Abschnitt 3.3.2) und ist dann auch erforderlich. Der Wert muss eine komma-getrennte Liste von ID's für den Ressourcen-Typ sein, z.B. **164654,4235684** (Dies sind keine echten gültigen IDs).

- **Der URL-Parameter *offset***
  Dieser URL-Parameter ist optional und nur bei Abfragen für eine relative Ressource wichtig, die über mehrere Seiten verteilt ist. Dann gibt dieser Parameter den Versatz an, d.h. beim wievielten Datensatz angefangen werden soll. Es scheinen immer maximal 10 Ergebnisse geliefert zu werden, es ist nicht bekannt, ob man die Seitengröße angeben kann.

  Achtung: Der Versatz scheint sich nicht immer um die Anzahl der empfangenen Items zu erhöhen. Z.B. kann es vorkommen, dass man bei `offset=0` zehn Items empfängt aber der Versatz danach nicht `offset=10` ist, sondern z.B. `offset=12` sein muss. Den korrekten Versatz für die nächste Seite kann man aus dem `next`-Schlüssel der Antwort-Daten erhalten. Berücksichtigt man diese Sprünge nicht, so kann man Duplikate erhalten, z.B. kann der Datensatz bei Offset 10 derselbe sein wie bei Offset 11. Warum Duplikate überhaupt geliefert werden, ist unklar.

- **Der URL-Parameter *extend***
  Dieser URL-Parameter ist nur relevant, wenn man Details zu mehreren Apps abfragen möchte (Ressourcen-Typ *apps*). Der Wert muss eine komma-getrennte Liste gültiger Optionen sein. Jede Option wird den Effekt haben, dass ein Schlüssel mit demselben Namen wie die Option in den Item-Attributen hinzugefügt wird, für eine Beschreibung der Optionen und der gelieferten Daten, s. die Ressource *apps* im Abschnitt B.1.2.

- **Der URL-Parameter *include***
  Dieser URL-Parameter erlaubt es, bestimmte relative Ressourcen in die Antwort-Daten miteinzubinden, ohne dass extra Requests für diese gemacht werden müssen. Die Daten finden sich dann im *relationships*-Objekt eines Items (s. Tabelle 3.9).
  Es ist zu beachten, dass nur der Parameter *l* für Sprache und Land rekursiv angewandt wird und dass Items für relative Ressourcen vom selben Typ keine Attribute beinhalten (d.h. z.B. für ähnliche Apps werden *extend*-Optionen ignoriert und generell keine Attribute geliefert, nur ID's)!
  Außerdem scheint die Option *genres* immer angewandt zu werden, egal ob sie angegeben wird oder nicht. *developer* wird ebenfalls immer geliefert, aber wenn nicht als Option angegeben, dann nur unvollständig (ohne Attribute).

B.1.2 Format der Antwort-Daten

Die nachfolgenden Abschnitte werden abhängig vom Ressourcen-Typ die Attribute eines Items auflisten.

Attribute für die Ressource *user-reviews*

Schlüssel	*Beschreibung*
title	Der Titel der Rezension.
rating	Die Bewertung (Ganzzahl 1-5).
isEdited	Wurde die Rezension bearbeitet?
review	Der eigentliche Text der Rezension.
userName	Der Benutzername.

Attribute für die Ressource *in-apps*

Schlüssel	*Beschreibung*
url	URL zu diesem In-App-Kauf (oder zu der App).
isMerchandiseEnabled	Boolescher Wert, genaue Bedeutung unklar.
isFamilyShareable	Boolescher Wert: Kann der In-App-Kauf über die Familienbibliothek geteilt werden?
offerName	ID des In-App-Kaufs als String
releaseDate	Veröffentlichungsdatum als landes-abhängiger String.
name	Name des In-App-Kaufs.
isMerchandiseVisibleByDefault	Boolescher Wert, genaue Bedeutung unklar.
isSubscription	Boolescher Wert: Ist der In-App-Kauf ein Abonnement?
description	Ein JSON-Objekt mit mehreren Beschreibungen. Jeder Schlüssel hat als Wert eine Beschreibung. Bis jetzt wurde nur der Schlüssel `standard` beobachtet.
offers	Ein JSON-Array, das die Preise für den Kauf liefert. Das Array enthält JSON-Objekte mit den folgenden Schlüsseln: • **buyParams**: String mit URL-Parametern für eine Kauf-URL. • **type**: Ein String, der den Kauf-Typ angibt (z.B. „get“, „update“ oder „buy“). • **priceFormatted**: Der nutzerfreundlich formatierte Preis als String (z.B. „$ 1.79“). • **price**: Der Preis als Kommazahl. • **currencyCode**: Die Währungs-ID als String.

Attribute für die Ressource *genres*

Schlüssel	*Beschreibung*
url	Die URL zur Übersichtsseite für dieses Genre.
name	Der übersetzte Name dieses Genres.
parentName	Der übersetzte Name des Ober-Genres.
parentId	Die ID (Ganzzahl) des Ober-Genres.

Attribute für die Ressource *developer*

Schlüssel	*Beschreibung*
url	Die URL zur Übersichtsseite für diesen Entwickler.
name	Der Name des Entwicklers.
editorialArtwork	Wie der Schlüssel `artwork` der Ressource *apps*.

Allgemeine Attribute für die Ressource *apps*

Dies sind die allgemeinen Attribute für die Ressource *apps*, d.h. wie bei allen anderen Ressourcen auch unter dem Item-Schlüssel `attributes`.

Schlüssel	*Beschreibung*
familyShareEnabledDate	Soll wohl das Datum sein, an dem die Familienbibliothek für die App aktiviert wurde (ISO 8601). Dieses scheint aber inkorrekt zu sein (immer dasselbe Datum, Jahr 0001), daher kann hiermit nur geprüft werden, ob die Familienbibliothek verfügbar ist.
url	Die URL zu der App in der Web-Oberfläche.
reviewsRestricted	Boolescher Wert, Bedeutung unklar.
supportsOcelot	Boolescher Wert, Bedeutung unklar.

Schlüssel	*Beschreibung*
deviceFamilies	Ein JSON-Array mit Geräte-Bezeichnern für untersützte Geräte-Typen (bekannt sind „iphone“, „ipad“, „ipod“, „mac“ und „appletv“).
chartPositions	Ein JSON-Objekt, wobei jeder Schlüssel für ein Rating-System steht und als Wert Daten über die Position der App enthält. Das einzige beobachtete System ist `appStore`, dieses hat als Wert ein JSON-Objekt mit den folgenden Schlüsseln: • **genre**: Die Genre-ID in dem die App platziert wird. • **genreName**: Der übersetzte Name des Genres. • **chart**: Ein String mit einem Bezeichner für das Wertungssystem, z.B. „top-free“. • **position**: Eine Ganzzahl mit der Position der App.
isPreorder	Boolescher Wert: Ist diese App schon veröffentlicht?
hasEula	Boolescher Wert: Hat diese App eine EULA?

Schlüssel	*Beschreibung*
userRating	Ein JSON-Objekt mit Daten bzgl. der Nutzer-Wertung der App. Dabei gibt es die folgenden Schlüssel: • **value**: Die Durchschnittswertung als Kommazahl. • **ratingCount**: Die Anzahl an Nutzerwertungen. • **ratingCountList**: Eine Liste mit fünf Indizes, wobei jeder Wert die Anzahl an Bewertungen für die Bewertung angibt, die durch seinen Index+1 definiert ist (Index 0: Bwewertung mit 1 Stern, Index 1: 2 Sterne, etc.)
offers	Ein JSON-Array, dass die Preise für den Kauf liefert. Der Array enthält JSON-Strukturen mit den folgenden Schlüsseln: • **buyParams**: String mit URL-Parametern für eine Kauf-URL. • **type**: Ein String, der den Kauf-Typ angibt (z.B. „install“ oder „buy“). • **priceFormatted**: Der nutzerfreundlich formatierte Preis als String (z.B. „$ 1.79“). • **price**: Der Preis als Kommazahl. • **currencyCode**: Die Währungs-ID als String.

Schlüssel	*Beschreibung*
contentRatingsBySystem	Ein JSON-Objekt, die Schlüssel sind Bezeichner für verschiedene Bewertungssysteme bzgl. Alterseinschränkungen. Der einzige bekannte Schlüssel ist `appsApple`, der Wert ist ein JSON-Objekt mit den folgenden Schlüsseln: • **name**: Der Name der Wertung (üblicherweise die Alterseinschränkung als Text, z.B. „9+“). • **value**: Bedeutung unbekannt, für „9+„ steht hier der Wert 200. • **rank**: Vermutlich eine Stufe innerhalb des Wertungssystem, für „9+“ ist der Wert z.B. 2. • **advisories**: Ein JSON-Array mit Strings für die Gründe der Einstufung.
name	Der Titel der App als String.
sellerLabel	Der übersetzte Text „Anbieter“
supportsArcade	Boolescher Wert, ob die App Apple Arcade untersützt.
isFirstPartyHideableApp	Boolescher Wert, Bedeutung unklar.
eula **Erfordert extend-Option**	Die EULA der App, vermutlich fehlt dieser Schlüssel, falls keine existiert.
usesLocationBackgroundMode	Boolescher Wert, vermutlich ob die App im Hintergrund Standortdaten verwendet.
fileSizeByDevice **Erfordert extend-Option**	Ein JSON-Objekt, das Geräte-Bezeichnern eine Ganzzahl zuordnet, die für die Größe der App in Bytes steht. Es gibt den Schlüssel „universal“, der vermutlich geräte-unabhängig ist.

Schlüssel	*Beschreibung*
platformAttributes	Ein JSON-Objekt, die Schlüsssel sind Plattform-Bezeichner und die Werte sind JSON-Objekte mit plattform-spezifischen Attributen. Nur ein Plattform-Bezeichner ist bekannt: `ios`. Der nächste Abschnitt listet die Schlüssel für diese Plattform auf.

iOS-Attribute für die Ressource *apps*

Dies sind Attribute für Ressource *apps*, die spezifisch für die Plattform *iOS* sind (s. `platformAttributes` im vorigen Abschnitt).

Schlüssel	*Beschreibung*
requires32bit	Boolescher Wert, vermutlich ob 32Bit-Kompatibilität erforderlich ist.
artwork	Eine JSON-Struktur für das Icon der App. Die relevanten Schlüssel sind: • **width**: Eine Ganzzahl für die Breite des Bilds. • **height**: Eine Ganzzahl für die Höhe des Bilds. • **url**: Die URL für das Bild. Achtung: Diese URL enthält ist nicht komplett, sie muss noch formatiert werden[1].
isDeliveredInIOSAppForZulu	Boolescher Wert, Bedeutung unklar.
hasMessagesExtension	Boolescher Wert, genaue Bedeutung unklar.

[1] Die letzte Pfadkomponente folgt dem Schema `<IMG-NAME>{w}x{h}{c}.{f}`, wobei **IMG-NAME** der Bild-Name ist. Die geschweiften Klammern sind Formatier-Optionen und müssen durch konkrete Werte ersetzt werden: *w* ist die Breite und *h* die Höhe. Das Bild wird dann passend skaliert. Man kann auch den Wert `0w` für die Höhe verwenden, dann wird diese der Breite angepasst (Seitenverhältnis wird beibehalten). Wofür *c* steht ist unklar, *f* ist das Format, möglich sind z.B. *jpg* oder *png*

Schlüssel	*Beschreibung*
versionHistory **Erfordert extend-Option**	Ein JSON-Array aus JSON-Objekten von denen jedes die folgenden Schlüssel hat: • **versionDisplay**: Ein String mit der Versionsnummer. • **releaseNotes**: Ein kurzer Text, der die Änderungen der Version beschreibt. • **releaseDate**: Veröffentlichungsdatum der Version, Format ist land-abhängig.
hasPrivacyPolicyText	Boolescher Wert, genaue Bedeutung unklar. Vermutlich ob die App einen Text zur Datenschutzerklärung hat.
supportedLocales	Ein JSON-Array mit unterstützen Sprachen der App. Jeder Eintrag ist ein JSON-Objekt mit den folgenden Schlüsseln: • **name**: Der übersetzte Name der Sprache. • **tag**: Das Sprach-Kürzel (ISO 639-1).
videoPreviewsByType **Erfordert extend-Option**	Ein JSON-Objekt, jeder Schlüssel ist der Bezeichner für ein Geräte-Typ und hat als Wert ein JSON-Array bei dem jeder Eintrag ein JSON-Objekt ist, das ein Video für die App und die Plattform beschreibt. Die Schlüssel sind die folgenden: • **previewFrame**: Ein JSON-Objekt für das Video-Thumbnail. Das Format ist wie bei `artwork`. • **video**: Eine URL zu einer m3u8-Video-Playlist für das zugeordnete Video.
privacyPolicyURL **Erfordert extend-Option**	Eine URL zu der Datenschutzerklärung der App.
supportsGameController	Boolescher Wert, ob Spiele-Controller unterstützt werden.

Schlüssel	*Beschreibung*
releaseDate	Das Veröffentlichungsdatum der App als String. Das Format ist vom Land abhängig.
hasSafariExtension	Boolescher Wert, genaue Bedeutung unklar. Vermutlich bzieht sich „Safari“ auf Apple's Web-Browser.
isStandaloneForZulu	Boolescher Wert, genaue Bedeutung unklar.
externalVersionId	Ganzzahlige ID, Bedeutung unklar.
screenshotsByType **Erfordert extend-Option**	Ein JSON-Objekt, dessen Schlüssel Geräte-Bezeichner sind und dessen Werte JSON-Arrays sind, bei denen jeder Eintrag ein JSON-Objekt ist, welches einen Screenshot für das Gerät beschreibt. Der Aufbau ist wei bei `artwork`.
supportURLForLanguage **Erfordert extend-Option**	Eine URL zu einer Support-Seite für die App.
websiteUrl **Erfordert extend-Option**	Eine URL zu der Webseite des Entwicklers.
editorialArtwork	Ein JSON-Objekt. Die Schlüssel sind Bezeichner für bestimte Typen von Artwork, die Werte sind JSON-Objekte, die ein Bild beschreiben. Der Aufbau ist wie bei `artwork`. Als Bezeichner sind bekannt: *storeFlowcase*, *subscriptionHero*, *bannerUber*, *brandLogo* und *emailFeature*.
requirementsString	Ein String, der Anforderungen an das Gerät angibt (z.B. „Erfordert iOS 9.0 oder höher“)
subtitle	Ein kurz zusammenfassender Untertitel für die App.
isDeliveredInIOS AppForWatchOS	Boolescher Wert, genau Bedeutung unklar.
bundleId	Ein String mit der sog. *Bundle ID* der App.
hasInAppPurchases	Boolescher Wert, ob die App In-App-Käufe hat.

Schlüssel	*Beschreibung*
isAppleWatchSupported	Boolescher Wert, ob Apple Watch unterstützt wird.
isStandaloneForWatchOS	Boolescher Wert, ob die App nur für WatchOS ist.
supportsPassbook	Boolescher Wert, ob die App Apple Passbook untersützt.
copyright	Ein String mit einen Copyright-Bezeichner für die App.
requiresGameController	Boolescher Wert, ob die App einen Spiele-Controller benötigt.
isStandaloneWith-CompanionForZulu	Boolescher Wert, genaue Bedeutung unklar.
isHiddenFromSpringboard	Boolescher Wert, ob die App vom Springboard (der iOS-„Desktop“) versteckt ist.
isGameCenterEnabled	Boolescher Wert, ob die App Apple's Game Center unterstützt.
minimumOSVersion	Eine Kommazahl mit der minimalen iOS-Version.
hasFamilyShareable-InAppPurchases	Boolescher Wert, ob die App In-App-Käufe hat, die mit der Familie teilbar sind.
editorialNotes	Ein JSON-Objekt mit den Schlüsseln **short** und **tagline**, die Notizen der Redaktion angeben.
is32bitOnly	Boolescher Wert, ob diese App nur auf 32Bit-Systemen läuft.
isStandaloneWith-CompanionForWatchOS	Boolescher Wert, genaue Bedeutung unklar.
seller	Der Name des Anbieters der App.
isSiriSupported	Boolescher Wert, ob Apple's Spracherkennung Siri untersützt wird.
languageList	Ein JSON-Array mit Strings, die die unterstützten Sprachen der App angeben (übersetzt).

Schlüssel	*Beschreibung*
messagesScreenshots **Erfordert extend-Option**	Ein JSON-Objekt, das vermutlich Screenshots für App-Benachrichtigungen beinhaltet. Bis jetzt nur leere Objekte beoachtet, daher keine Aussage über Format möglich.
description **Erfordert extend-Option**	Wie `description` der Ressource *in-apps*.
requiredCapabilities	Ein String mit Voraussetzungen an Gerät und Plattform (komma-getrennte Werte), z.B. „armv7“.
offers	Beschreibt den Preis der App (oder Angebote dafür). Die Struktur ist dieselbe wie bei `offers` der Ressource *in-apps*.

Es gibt noch einige weitere *extend*-Optionen, deren Wirkungsweise allerdings nicht klar ist, da bis jetzt bei keiner getesteten App Änderungen zu sehen waren. Diese Optionen sind **developerInfo**, **editorialVideo**, **privacyPolicyText** und **promotionalText**.

B.1.3 Extraktion des Autorisierungs-Token

Wie in Abschnitt 3.3.2 erläutert wird für AppHost-Requests ein Autorisierungs-Token benötigt, welches aus einer App-Detail-Seite extrahiert werden kann.

Die URL für eine App-Detail-Seite ist allgemein:

`https://apps.apple.com/<GL>/app/id<ID>`

Hierbei ist **GL** das Länder-Kürzel und **ID** eine App ID. In dem empfangenen HTML-Dokument gibt es ein *meta*-Tag mit Attribut `name="web-experience-app/config/environment"`, welches ein weiteres Attribut namens `content` hat. Der Wert dieses Attributs ist ein url-kodierter JSON-String. Parst man diesen, so erhält man ein JSON-Objekt mit einigen Konfigurationsdaten.

Relevant ist der Schlüssel `MEDIA_API`. Dies ist wieder ein JSON-Objekt. Der Schlüssel `token` ist das gesuchte Token. Das Token scheint generell zumindest eine Zeit lang immer dasselbe zu sein, unabhängig von App ID und Sprache, daher bietet sich hier Caching an.

Achtung: Das HTML-*meta*-Tag wird aus dem DOM durch Skripte entfernt, daher wird man es im Browser-Inspektor mit aktiviertem JavaScript nicht finden können.

B.2 App-Details mittels Storefront-Header

Die URL, um Details zu einer App abzufragen ist

`https://apps.apple.com/<GL>/app/id<ID>`.

Fügt man dem HTTP-Request den Header `X-Apple-Store-Front:<SID>,32` hinzu (wobei **SID** eine Storefront ID ist), dann hat die gelieferte HTML-Seite ein etwas anderes Format. Es gibt ein Skript-Tag, das einer globalen Variable `its.serverData` ein JSON-Objekt mit nützlichen Daten zuweist:

```
<script type="text/javascript" charset="utf-8">
its.serverData = {
  "storePlatformData": {
    "product-dv-product": {
      "results": {
        <APPID>: <APP-DATA>
      }
    }
  },
  "pageData": {
    "softwarePageData": <SP-DATA>
  }
```

Sowohl **APP-DATA** als auch **SP-DATA** sind JSON-Objekte, die relevante Daten beinhalten. Viele von diesen Daten sind großteils gleich aufgebaut wie bei App-Details über AppHost-Requests.

Die folgenden Schlüssel finden sich im **APP-DATA**-Objekt:

Schlüssel	*Beschreibung*
artwork	Ein Array von JSON-Objekten, die die Schlüssel **width**, **height** und **url** haben und so ein Icon für die App in unterschiedlichen Auflösungen beschreiben.
artistName	Der Name des Entwickelers
familyShareEnabledDate	Wie bei AppHost-Requests
hasMessagesExtension	Wie bei AppHost-Requests
url	Wie bei AppHost-Requests
softwareInfo	Ein JSON-Objekt mit diesen Schlüsseln: • **seller** • **requirementsString** • **eulaUrl** • **supportUrl** • **websiteUrl** • **privacyPolicyUrl** Alle Schlüssel entsprechenden denen bei AppHost-Requests (`supportUrl` entspricht `supportURLForLanguage`).
deviceFamilies	Wie bei AppHost-Requests.
genreNames	Wie `genres` bei AppHost-Requests.
isPreorder	Wie bei AppHost-Requests.
id	Die App ID
releaseDate	Wie bei AppHost-Requests.

Schlüssel	*Beschreibung*
userRating	Ein JSON-Objekt mit den Schlüsseln **value**, **ratingCount**, **valueCurrentVersion** und **ratingCountCurrentVersion**, die jeweils die Durschschnitts-Nutzerwertung und Anzahl von Nutzerwertungen für die aktuelle Version und die App insgesamt angeben.
contentRatingsBySystem	Wie bei AppHost-Requests.
name	Wie bei AppHost-Requests.
uber	Ein JSON-Objekt mit einem relevanten Schlüssel **masterArt**. Der Wert hat denselben Aufbau wie der Schlüssel `artwork` und das zugeordnete Bild scheint eine Art Banner-Bild zu sein.
artistUrl	Die URL zur App-Übersichtsseite des Entwicklers.
screenshotsByType	Ein JSON-Objekt, die Schlüssel sind Bezeichner für Geräte-Typen und die Werte sind JSON-Arrays, bei dem jeder Eintrag für einen Screenshot steht. Ein Eintrag hat denselben Aufbau wie der Wert von `artwork`.
subtitle	Wie bei AppHost-Requests.
bundleId	Wie bei AppHost-Requests.
isFirstPartyHideableApp	Wie bei AppHost-Requests.
hasInAppPurchases	Wie bei AppHost-Requests.
isAppleWatchSupported	Wie bei AppHost-Requests.
supportsPassbook	Wie bei AppHost-Requests.
copyright	Wie bei AppHost-Requests.
subtitle	Wie bei AppHost-Requests.

Schlüssel	*Beschreibung*
videoPreviewByType	Wie bei AppHost-Requests, aber der `previewFrame`-Schlüssel ist wie `artwork` hier aufgebaut und nicht wie bei AppHost-Requests.
usesLocationBackgroundMode	Wie bei AppHost-Requests.
requiresGameController	Wie bei AppHost-Requests.
isHiddenFromSpringBoard	Wie bei AppHost-Requests.
artistId	Die ID des Entwicklers.
fileSizeByDevice	Wie bei AppHost-Requests.
genres	Ein Array aus JSON-Objekten, die jeweils die Schlüssel **id**, **name** und **url** (alles Strings) beinhalten und somit ID, übersetzten Namen und URL zur Genre-Seite angeben.
minimumOSVersion	Wie bei AppHost-Requests (einziger Unterschied: hier ist das „s“ groß geschrieben!).
isGameControllerSupported	Wie bei AppHost-Requests.
is32bitOnly	Wie bei AppHost-Requests.
isSiriSupported	Wie bei AppHost-Requests.
description	Wie bei AppHost-Requests.
requiredCapabilities	Wie bei AppHost-Requests.
offers	Wie bei AppHost-Requests.

Die folgenden Schlüssel finden sich im **SP-DATA**-Objekt:

Schlüssel	*Beschreibung*
versionHistory	Wie bei AppHost-Requests.
customersAlsoBoughtApps	Ein JSON-Array von ID’s (als String) von ähnlichen Apps.
moreByThisDeveloper	Ein JSON-Array von ID’s (als String) von anderen Apps des Entwicklers.

Schlüssel	*Beschreibung*
addOns	Gibt Details zu In-App-Käufen an. Großteils der gleiche Aufbau wie `offers`, aber der Schlüssel **type** fehlt. Stattdessen gibt es den Schlüssel **offerType**, der als Wert ein JSON-Objekt mit dem Schlüssel **offerType** hat, welcher dieselbe Bedeutung wie **type** hat.

B.3 Nutzung der Lookup-API

Die URL unter die die Lookup-API erreichbar ist, ist `https://itunes.apple.com/<METHOD>`. Es gibt zwei verschiedenen Methoden, die genutzt werden können: `search` zur Suche nach Apps und `lookup`, um Daten zu einer bestimmten ID nachzuschlagen.

Zunächst werden die relevanten URL-Parameter erläutert. Erforderliche Parameter sind **fett** markiert. Für eine volle Liste, siehe [14].

Parameter	*Methode*	*Beschreibung*
term	search	Der url-kodierte Suchterm.
id	lookup	Die ID, die nachgeschlagen werden soll. Dies muss nicht unbedingt eine App ID sein, es funktioniert z.B. auch eine Entwickler-ID, mit der man dann alle Apps eines Entwicklers bekommen kann!
bundleId	lookup	Alternative zu `id`, stattdessen die sog. Bundle ID.
country	beide	Das Länder-Kürzel (ISO 3166-1 alpha-2) für den zu durchsuchenden Store.
lang	beide	Die Sprache für die Ergebnisse. Muss Sprach- und Länderkürzel (ISO 639-1 bzw. ISO 3166-1 alpha-2), getrennt durch einen Unterstrich enthalten, z.B. „en_us“.
media	beide	Der Medien-Typ. Muss für Apps **software** sein.

Parameter	*Methode*	*Beschreibung*
entity	beide	Der Typ der gelieferten Daten. Muss für Apps **software**, **iPadSoftware** oder **macSoftware** sein.
attribute	search	Erlaubt Suche nach bestimmtem Attribut. Kann weggelassen werden, um „alle“ Attribute zu durchsuchen.
limit	search	Erlaubt, die Anzahl der Ergebnisse zu limitieren. Der Maximalwert ist 200.
version	beide	Bestimmt Format der Egebnis-Daten. Möglich sind 1 und 2. Standard ist 2.

Die gelieferte Antwort ist immer ein JSON-Objekt mit zwei Schlüsseln: Der erste Schlüssel ist **resultCount**, der die Anzahl der Ergebnisse angibt und der zweite ist **results**, ein JSON-Array bei der jeder Eintrag ein Ergebnis-Item (JSON-Objekt) ist. Der Aufbau hängt vom `media`- und `entity`-Parameter ab, sind diese beide *software*, so erhält man die folgenden Schlüssel im Ergebnis:

Schlüssel	*Beschreibung*
wrapperType	Definiert den Ergebnis-Typ. Für Apps benötigen wir „software“.
trackId	Die ID der App.
bundleId	Die Bundle ID der App.
trackViewUrl	Die URL zur App-Detail-Seite der App.
trackName	Der Titel der App.
description	Die Beschreibung der App.
artworkUrl512 artworkUrl100 artworkUrl60	URLs zum Icon der App, in den Auflösungen 512x512, 100x100 und 60x60 Pixel.
screenshotUrls ipadScreenshotUrls appletvScreenshotUrls	Jeweils JSON-Arrays mit URLs zu Screenshots der App von iPhone, iPad und AppleTV.

Schlüssel	*Beschreibung*
fileSizeBytes	Die Größe der App als String (der Wert ist eine Zahl, muss aber geparsed werden).
version	Die aktuelle Version der App als String.
minimumOsVersion	Die minimale Betriebssystem-Version für die App als String.
price	Der Preis der App als Kommazahl.
formattedPrice	Der Preis als formatierter Text.
currency	Die Währungs-ID als String.
averageUserRating	Die durchschnittliche Nutzer-Wertung.
averageUserRatingForCurrentVersion.	Die durchschnittliche Nutzer-Wertung für die aktuelle Version.
userRatingCount	Die Anzahl an Nutzer-Wertungen für die App.
userRatingCountForCurrentVersion	Die Anzahl an Nutzer-Wertungen für die aktuelle Version der App.
contentAdvisoryRating	Der Text für die Alters-Einstufung.
advisories	Ein JSON-Array mit Gründen (Strings) für die Einstufung.
artistId	Die ID des Entwicklers.
artistName	Der Name des Entwicklers.
sellerUrl	Die Webseite des Entwicklers.
sellerName	Der Unternehmens-Name des Entwicklers (Verkäufer).
releaseNotes	Der Changelog (String) für die letzte Aktualisierung der App.
currentVersionReleaseDate	Datum der Veröffentlichunge der aktuellen Version der App (Schlüssel kann evtl. fehlen).

Schlüssel	*Beschreibung*
releaseDate	Das Datum der originalen Veröffentlichung der App (Format sprachabhängig).
primaryGenreId	Die ID des Haupt-Genres, zu dem die App gehört (Ganzzahl).
primaryGenreName	Der Name des Haupt-Genres, zu dem die App gehört (String).
genreIds	Ein Array von Genre-IDs für Genres, zu der die App gehört.
genres	Ein Array von Genre-Namen für Genres, zu der die App gehört.
languageCodesISO2A	Ein JSON-Array mit Sprach-Codes für Sprachen, die die App unterstützt.
supportedDevices	Ein JSON-Array mit Strings, die Bezeichner für untersützte Geräte sind.

Zusätzliche Informationen liefert die offzielle Dokumentation [14].

B.4 Apps eines Entwicklers aus der Web-Oberfläche

Bei AppHost-Requests kann man nur mittels der relativen Ressource *developer-other-apps* andere Apps eines Entwicklers erhalten, man benötigt dazu aber bereits eine App ID. Die Web-Oberfläche hat jedoch auch Entwickler-Seiten, bei denen man alle Apps eines Entwicklers sehen kann. Apple setzt wie bereits erwähnt EmberJS [27] ein, für serverseitiges Rendering wird Fastboot [34] verwendet. Fastboot hat ein Konzept namens *Shoebox* mit dem man den Zustand der Anwendung vom Server an den Client weitergeben kann, z.B. um dort noch mehr Rendering durchzuführen. Shoebox-Daten werden in Skript-Tags also rohe JSON-Objekte abgelegt. Dies sieht z.B. so aus:

```
<script type="fastboot/shoebox" id="...">
  {"key": "value", "key2": "value2", ...}
</script>
```

Das `type`-Attribut hat immer den Wert `fastboot/shoebox` und das `id`-Attribut identifiziert die Shoebox. Eine Entwickler-Seite erreicht man unter der URL

`https://apps.apple.com/<GL>/developer/id<ID>`.

Hierbei ist **GL** wieder ein Länder-Kürzel und **ID** ist die Entwickler-ID. Das HTML-Dokument enthält verschiedene Shoebox-Skript-Tags und das, das wir suchen hat die ID `shoebox-ember-data-store`. Die darin enthaltenen Daten liefern zwar keine App-Details, aber die IDs der Apps des Entwicklers lassen sich auslesen.

Das enthaltene JSON-Objekt hat einen Schlüssel `data`, welcher ein JSON-Objekt mit einem Schlüssel `relationships` liefert. Der Wert ist ein JSON-Objekt, das als Schlüssel Bezeichner für diverse Plattform/Geräte-Typen hat. Zugeordnet sind JSON-Arrays, deren Einträge Objekte mit einem Schlüssel `id` sind, der die App ID als String speichert. Z.B. sieht dies so aus:

```
{
  "data": {
    "relationships": {
      "iPhoneApps": [
        {"type": "lockup/app", "id": "..."},
        ...,
        {"type": "lockup/app", "id": "..."}
      ],
      "macApps": ...
    },
    ...
  },
  ...
}
```

So lassen sich die IDs der Apps extrahieren, die App-Details kann man dann mit den anderen Methoden extrahieren.

B.5 Datenerhalt über Legacy WebObjects

Abschnitt 3.3.2 geht kurz auf den Aufbau von Requests an WebObject-URLs ein. Nachfolgend werden einige konkrete URLs aufgeführt, welche Parameter diese erfordern und welche Daten diese liefern.

B.5.1 Suche nach Apps

Host	search.itunes.apple.com
Pfad	/WebObjects/MZStore.woa/wa/search
URL-Parameter	• **term**: Der url-kodierte Suchbegriff. • **clientApplication**: Muss **Software** sein. • **media**: Muss **software** sein.
Storefront-Header	`<SID>,24 t:native`

Als Antwort erhalten wir ein JSON-Objekt mit einer Menge verschiedener Daten. Relevant ist der Schlüssel **bubbles**. Dieser hat als Wert ein JSON-Array nur einem Eintrag, ein JSON-Objekt mit den Schlüssel **results**. Der zugeordnete Array enthält JSON-Objekte mit einem Schlüssel **id** für die ID eines Suchergebnisses (als String). Der Schlüssel **entity** sollte den Wert *software* haben, sonst ist die ID keine App ID. Die Struktur sieht wie folgt aus:

```
{
  "bubbles": [
    {
      "results": [
```

```
        {"id": <ID>, "entity": <ENT>, ...}
        ...,
        {"id": <ID>, "entity": <ENT>, ...}
      ]
    }
  ]
}
```

Es gibt hier übrigens keine *Pagination*, die ID-Liste kann aber recht lang sein.

B.5.2 Vervollständigung eines Suchbegriffs

Host	search.itunes.apple.com
Pfad	/WebObjects/MZSearchHints.woa/wa/hints
URL-Parameter	• **q**: Der url-kodierte Suchbegriff. • **media**: Muss **software** sein.
Storefront-Header	`<SID>,24`

Als Antwort erhält man diesmal kein JSON, sondern XML. Es scheint zumindest über den Storefront-Header keinen Möglichkeit zu geben, das zu ändern. Das XML liegt in Apple's *plist*-Format (*property list*) vor, die DTD ist in `http://www.apple.com/DTDs/PropertyList-1.0.dtd` gegeben. Im Wesentlichen bildet es JSON-Typen auf die folgenden XML-Tags ab:

- **string**, **real**, **integer** und ein paar weitere Tags entsprechen den primitiven JSON-Werten und dürfen nur ein Wert des entsprechenden Typs beinhalten.
- **array** ist ein Tag bei dem jeder Kindknoten als Eintrag eines Arrays betrachtet wird (wie ein JSON-Array).
- **dict**-Tags entsprechen ungefähr JSON-Objekten. Es muss aus Paaren von einem **key**-Tag gefolgt und einem weiteren Tag bestehen. Das zweite Tag ist der Wert zu dem Schlüssel.

Das erhaltene XML entspricht einem JSON-Objekt, das einen Schlüssel **hints** hat, dessen Wert ein Array aus Objekten ist, wobei jedes Objekt einen Schlüssel **term** hat, der den vervollständigten Suchbegriff beinhaltet. Das sieht z.B. so aus:

```
<dict>
  <key>title</key><string>Suggestions</string>
  ...
  <key>hints</key><array>
    <dict>
      <key>term</key><string>TERM-1</string>
      <key>url</key><string>URL-1</string>
    </dict>
    ...
    <dict>
      <key>term</key><string>TERM-N</string>
      <key>url</key><string>URL-N</string>
    </dict>
  </array>
</dict>
```

B.6 Daten aus RSS-Feeds

B.6.1 RSS-Feed für Rezensionen

Der RSS-Feed für Rezensionen zu einer App hat die folgende URL:

`https://itunes.apple.com/<GL>/rss/customerreviews/page=<PG>/id=<ID>/sortby=<SRT>/json`

Die Parameter sind hierbei:

GL	Das Länder-Kürzel
PG	Die Seiten-Nummer (min. 1, max. 10, 50 Items pro Seite)
ID	Die App ID
SRT	Eine Sortier-Option. Die `sortby`-Pfadkomponente kann auch weggelassen werden, um die Standard-Sortierung zu verwenden. Die möglichen Werte sind *mostRecent* und *mostHelpful*.

Als Antwort erhält man ein JSON-Objekt mit einem Schlüssel **feed**, dessen Wert ein Objekt mit einem Schlüssel **entry** ist, der wiederum ein JSON-Array mit den eigentlichen Rezensionen beinhaltet. Jede Rezension ist ein JSON-Objekt mit den folgenden Schlüsseln:

Schlüssel	*Beschreibung*
author	Ein JSON-Objekt mit den Schlüsseln **uri** und **name**, die jeweils eine URL zum Profil des Nutzers und seinen Namen angeben.
im:version	Die Version der App, für die die Rezension verfasst wurde.
im:rating	Die abgegebene Bewertung für die App.
id	Die ID der Rezension.
title	Der Titel der Rezension.
content	Der Text der Rezension.
voteSum	Nicht ganz klar, vielleicht die Anzahl an „Daumen hoch“ der Rezension.
voteCount	Unterschied zu *voteSum* nicht ganz klar, vielleicht die Anzahl an „Daumen hoch“ und „Daumen runter“.

Es ist anzumerken, dass die Werte bei diesen Schlüsseln immer JSON-Objekte mit einem Schlüssel **label** sind, welcher den eigentlichen Wert als String beinhaltet. Z.B. erhält man den Nutzernamen aus dem Schlüssel `author`, dann `name` und dann `label`. ID’s und andere numerische Daten müssen entsprechend in Ganzzahlen geparsed werden.

B.6.2 Der alte RSS-Feed für App-Listen

Der alte (Legacy) RSS-Feed für Rezensionen zu einer App hat die folgende URL:

`http://ax.itunes.apple.com/WebObjects/MZStoreServices.woa/ws/RSS/<COLL>/<CAT>/limit=<LIM>/json?s=<SID>`

Die Parameter sind hierbei:

SID	Die Storefront-ID wie bei WebObject-Anfragen.
COLL	Der Bezeichner der abgefragten Sammlung.
CAT	Die Kategorie-ID. Achtung: Dies scheint nicht korrekt zu funktionieren, die Pfadkomponente hierfür kann einfach weggelassen werden.
LIM	Eine Ganzzahl als Begrenzer für die Anzahl abgefragter Items. Die zugehörige Pfadkomponente kann auch weggelassen werden für eine Standard-Anzahl.

Als Antwort erhält man ein JSON-Objekt mit einem Schlüssel **feed**, dessen Wert ein Objekt mit einem Schlüssel **entry** ist, der wiederum ein JSON-Array mit den eigentlichen Apps beinhaltet. Jede App ist ein JSON-Objekt mit den folgenden Schlüsseln:

Schlüssel	*Beschreibung*
im:name	Ein JSON-Objekt, das im Schlüssel **label** den Namen der App speichert.
im:image	Ein JSON-Array, welches aus JSON-Objekten besteht. Jedes Objekt hat einen Schlüssel **label** (die URL für ein Icon der App) und einen Schlüssel **attributes**, ein JSON-Objekt mit Bild-Attributen. Bekannt ist nur das Attribut **height**, die Höhe des Bilds (als String abgespeichert!).
im:summary	Ein JSON-Objekt, dass im Schlüssel **label** eine Kurzebeschreibung der App speichert.

Schlüssel	*Beschreibung*
im:price	Ein JSON-Objekt mit einem Schlüssel **attributes**. Der Wert ist ein JSON-Objekt mit den Schlüsseln **amount** und **currency**, die den Preis und die Währungs-ID als Strings angeben (der Preis kann in eine Kommazahl geparsed werden).
rights	Ein JSON-Objekt, das im Schlüssel **label** einen String bzgl. Copyright speichert.
title	Ein JSON-Objekt, das im Schlüssel **label** den Namen der App speichert.
link	Ein JSON-Objekt oder ein Array von JSON-Objekten. Der Schlüssel **attributes** liefert ein Objekt, in dem Attribute gespeichert sind. Eines der Objekte hat ein Attribut **rel** mit dem Wert *alternate*. Dieses Objekt hat auch ein Attribut **href**, welcher den Link zur App-Detail-Seite der App liefert.
im:artist	Ein JSON-Objekt, das im Schlüssel **label** den Namen des Entwicklers hat. Der Schlüssel **attributes** ist ein JSON-Objekt mit einem Schlüssel **href**, der eine URL zur App-Übersichtsseite liefert.
id	Ein JSON-Objekt mit einem verschachtelten Objekt im Schlüssel **attributes**. Dieses hat einen Schlüssel **im:id** mit der App ID (als String) und **im:bundleId** mit der Bundle ID der App.
category	Ein JSON-Objekt mit einem verschachtelten Objekt im Schlüssel **attributes**. Dieses hat einen Schlüssel **im:id** mit der Genre ID der App (als String) und **label** mit dem übersetzten Genre-Namen für das Haupt-Genre der App.
im:releaseDate	Ein JSON-Objekt, das im Schlüssel **label** das Veröffentlichungsdatum der App (ISO 8601) speichert.

B.6.3 Der neue RSS-Feed für App-Listen

Der neue RSS-Feed für Rezensionen zu einer App hat die folgende URL:

`https://rss.itunes.apple.com/api/v1/<GL>/ios-apps/<COLL>/<CAT>/<LIM>/explicit.json`

Die Parameter sind hierbei:

GL	Das Länder-Kürzel.
COLL	Der Bezeichner der abgefragten Sammlung.
CAT	Die Kategorie-ID oder der spezielle Wert „all“ für alle Kategorien. Achtung: nur „all“ scheint hier zu funktioneren.
LIM	Eine Ganzzahl als Begrenzer für die Anzahl abgefragter Items.

Als Antwort erhält man ein JSON-Objekt mit einem Schlüssel **feed** mit einem Schlüssel **results**, der als Wert einen JSON-Array mit den eigentlichen Apps beinhaltet. Jede App ist ein JSON-Objekt mit den folgenden Schlüsseln:

Schlüssel	*Beschreibung*
name	Der Titel der App.
artworkUrl100	Eine URL zu einem 100x100 Pixel großen Icon der App.
copyright	Ein Copyright-String für die App.
id	Die ID der App als String.
url	Die URL zu der App-Detail-Seite für die App.
artistId	Die ID des Entwicklers als String.
artistName	Der Name des Entwicklers.
releaseDate	Das Veröffentlichungsdatum der App (Format ist landesabhängig).
genres	Ein JSON-Array aus JSON-Objekten, die die Genres der App beschreiben. Jedes Objekt hat die Schlüssel **genreId** und **name** für ID (als String) und übersetztem Namen des Genres.

B.7 Konstanten

B.7.1 Sammlungs-IDs für alte RSS-Feeds

topmacapps	**topfreemacapps**
topgrossingmacapps	**toppaidmacapps**
newapplications	**newfreeapplications**
newpaidapplications	**topfreeapplications**
topfreeipadapplications	**topgrossingapplications**
topgrossingipadapplications	**toppaidapplications**
toppaidipadapplications	

B.7.2 Sammlungs-IDs für neue RSS-Feeds

new-apps-we-love	**new-games-we-love**	**top-free**
top-free-ipad	**top-grossing**	**top-grossing-ipad**
top-paid		

B.7.3 Kategorie-IDs

Name	*ID*	*Name*	*ID*
BOOKS	6018	**MAGAZINES_FOOD**	13012
BUSINESS	6000	**MAGAZINES_CRAFTS**	13013
CATALOGS	6022	**MAGAZINES_ELECTRONICS**	13014
EDUCATION	6017	**MAGAZINES_ENTERTAINMENT**	13015
ENTERTAINMENT	6016	**MAGAZINES_FASHION**	13002
FINANCE	6015	**MAGAZINES_HEALTH**	13017
FOOD_AND_DRINK	6023	**MAGAZINES_HISTORY**	13018
GAMES	6014	**MAGAZINES_HOME**	13003
GAMES_ACTION	7001	**MAGAZINES_LITERARY**	13019
GAMES_ADVENTURE	7002	**MAGAZINES_MEN**	13020

Name	*ID*	*Name*	*ID*
GAMES_ARCADE	7003	**MAGAZINES_MOVIES_AND_MUSIC**	13021
GAMES_BOARD	7004	**MAGAZINES_POLITICS**	13001
GAMES_CARD	7005	**MAGAZINES_OUTDOORS**	13004
GAMES_CASINO	7006	**MAGAZINES_FAMILY**	13023
GAMES_DICE	7007	**MAGAZINES_PETS**	13024
GAMES_EDUCATIONAL	7008	**MAGAZINES_PROFESSIONAL**	13025
GAMES_FAMILY	7009	**MAGAZINES_REGIONAL**	13026
GAMES_MUSIC	7011	**MAGAZINES_SCIENCE**	13027
GAMES_PUZZLE	7012	**MAGAZINES_SPORTS**	13005
GAMES_RACING	7013	**MAGAZINES_TEENS**	13028
GAMES_ROLE_PLAYING	7014	**MAGAZINES_TRAVEL**	13029
GAMES_SIMULATION	7015	**MAGAZINES_WOMEN**	13030
GAMES_SPORTS	7016	**MEDICAL**	6020
GAMES_STRATEGY	7017	**MUSIC**	6011
GAMES_TRIVIA	7018	**NAVIGATION**	6010
GAMES_WORD	7019	**NEWS**	6009
HEALTH_AND_FITNESS	6013	**PHOTO_AND_VIDEO**	6008
LIFESTYLE	6012	**PRODUCTIVITY**	6007
MAGAZINES_AND_NEWSPAPERS	6021	**REFERENCE**	6006
MAGAZINES_ARTS	13007	**SHOPPING**	6024
MAGAZINES_AUTOMOTIVE	13006	**SOCIAL_NETWORKING**	6005
MAGAZINES_WEDDINGS	13008	**SPORTS**	6004
MAGAZINES_BUSINESS	13009	**TRAVEL**	6003
MAGAZINES_CHILDREN	13010	**UTILITIES**	6002
MAGAZINES_COMPUTER	13011	**WEATHER**	6001

B.7.4 Storefront-IDs

Die ID's lassen sich aus Apple's *Advanced Partner Linking* entnehmen [9].

dz	143563	**ag**	143540	**ao**	143564	**ai**	143538	**ar**	143505
am	143524	**au**	143460	**at**	143445	**az**	143568	**bh**	143559

bb	143541	**by**	143565	**be**	143446	**bz**	143555	**bm**	143542
bo	143556	**bw**	143525	**br**	143503	**bs**	143539	**vg**	143543
bn	143560	**bg**	143526	**ca**	143455	**ky**	143544	**kz**	143517
cl	143483	**cn**	143465	**co**	143501	**cr**	143495	**hr**	143494
cy	143557	**cz**	143489	**dk**	143458	**dm**	143545	**ec**	143509
eg	143516	**sv**	143506	**ee**	143518	**fi**	143447	**fr**	143442
de	143443	**do**	143508	**gb**	143444	**gh**	143573	**gr**	143448
gd	143546	**gt**	143504	**gy**	143553	**hn**	143510	**hk**	143463
hu	143482	**is**	143558	**in**	143467	**id**	143476	**ie**	143449
il	143491	**it**	143450	**jm**	143511	**jp**	143462	**jo**	143528
ke	143529	**kn**	143548	**kr**	143466	**kw**	143493	**lv**	143519
lb	143497	**lc**	143549	**lt**	143520	**lu**	143451	**md**	143523
mo	143515	**mk**	143530	**mg**	143531	**my**	143473	**ml**	143532
mt	143521	**mu**	143533	**mx**	143468	**ms**	143547	**np**	143484
nl	143452	**nz**	143461	**ni**	143512	**ne**	143534	**ng**	143561
no	143457	**om**	143562	**pk**	143477	**pa**	143485	**py**	143513
pe	143507	**ph**	143474	**pl**	143478	**pt**	143453	**qa**	143498
ro	143487	**ru**	143469	**sa**	143479	**sn**	143535	**sg**	143464
sk	143496	**si**	143499	**za**	143472	**es**	143454	**lk**	143486
sr	143554	**se**	143456	**ch**	143459	**tc**	143552	**tw**	143470
tz	143572	**th**	143475	**tn**	143536	**tr**	143480	**tt**	143551
ug	143537	**ua**	143492	**ae**	143481	**us**	143441	**uy**	143514
uz	143566	**vc**	143550	**ve**	143502	**vn**	143471	**ye**	143571

C

Scraper-Methoden

In diesem Kapitel werden die Scraper-Methoden definiert, wie es in Anforderung FK02 verlangt wird. Dabei werden separat für den Play Store und den App Store für jede Methode die Parameter und das Format des Ergebnisses festgelegt.

Sämtliche Ergebnisse liegen immer in Form von JSON vor. Wird nichts anderes explizit vermerkt, so ist das Ergebnis ein JSON-Objekt und eine Tabelle gibt die Schlüssel des Ergebnis-Objekts an. Für JSON-Objekte gilt generell, dass ein Schlüssel immer den Wert `null` haben kann (kann z.B. der Fall sein bei Daten, die nur manchmal vorhanden sind). Dieser Umstand wird nicht explizit vermerkt!

Es ist anzumerken, dass die folgenden zwei Parameter bei allen Methoden immer möglich sind und daher in den Parameter-Angaben weggelassen werden:

- **lang**: Optional ein Sprach-Kürzel (ISO 639-1), standardmäßig „en“. Bestimmt vor allem die Sprache der gelieferten Daten.
- **country**: Optional ein Länder-Kürzel (ISO 3166-1 alpha-2), welches Dinge wie Verfügbarkeit, Datumsformat, Währungsschreibweise, etc. festlegt

Für die REST-API gilt, dass die Parameter als URL-Parameter übergeben werden. Für Parameter, die mehrere Werte in Form eines Arrays annehmen darf der URL-Parameter mit unterschiedlichen Werten wiederholt werden. Die Methoden für die REST-API sind unter dem Pfad `/<STORE>/<METHOD>` zu erreichen, wobei **STORE** entweder *playstore* oder *appstore* sein muss und **METHOD** die entsprechende Methode. Im Verzeichnis `/docs` findet sich eine interaktive Dokumentation.

Die Fehlertypen werden hier allgemein definiert, da sie bei allen Methoden im Wesentlichen gleich sind. Es gibt grob die folgenden drei Fehlertypen:

1. **Illegale Parameter**: Dies ist z.B. der Fall bei ungültigen ID's, ungültigen Sprach/Land-Kombinationen, Werte außerhalb gültiger Wertebereiche, etc. Teilweise ist dies im Voraus erkennbar und teilweise nicht.
2. **I/O-Fehler**: Falls es Probleme gibt, die Verbindung mit einem Store herzustellen (generell I/O-Fehler). Für die REST-API erhält man dann HTTP-Fehlercode **500**.
3. **Store-Änderungen**: Fehler, die durch Store-Änderungen hervorgerufen werden. Dies kann nicht wirklich detektiert werden, außer dass die Datenverarbeitung nicht mehr korrekt funktioniert, was aber auch bei ungültigen Eingaben (z.B. ungültigen App ID's) der Fall ist. Daher wird dies meistens als der erste Fehler-Typ detektiert.

Der Scraper-Kern wirft bei einem Fehler eine Exception vom Typ **ScrapeException**. Die REST-API liefert einen der folgenden HTTP-Fehler: 404 wenn eine Ressource nicht existiert (sofern detektierbar), 400 für client-seitige Fehler, 422 für Validierungsfehler (durch FastAPI) und 500 für interne Fehler (z.B. I/O-Fehler). In jedem Fall wird der Körper der HTTP-Antwort einen Fehlertext als String beinhalten.

C.1 Google Play Store

Für den Play Store wird häufiger der Parameter **maxCount** oder **maxApps** zu sehen sein. Dieser Parameter wird verwendet, um die Anzahl gelieferter Daten zu beschränken. Es ist anzumerken, dass dieses Maximum nur als Schwellwert verwendet wird, d.h. wenn der Wert überschritten wird, werden keine weiteren Daten abgerufen. Die Ergebnis-Liste wird aber nicht gekürzt, kann also immer noch länger sein! Der Grund dafür ist unter Anderem, dass Tokens für weitere Datenseiten dann nicht mehr den korrekten Versatz darstellen würden.

C.1.1 Methode: *clusterList*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
path	str	Nein	Der Play-Store-Pfad unter dem die Cluster-Sammlung zu finden ist.
clusterKey	str	Ja (ds:3)	Der Skript-Tag-Schlüssel für das Skript-Tag, das die Daten der Cluster-Sammlung beinhaltet.
maxCount	int > 0	Ja (15)	Die maximale Anzahl an Cluster-Sammlungen, die extrahiert werden sollen.

Ergebnis

Name	*Typ*	*Beschreibung*
token	str	Das Token, mit dem die nächste Seite extrahiert werden kann.
topics	Array	Daten von Topic-Cards auf der Seite. Enthalten sind JSON-Objekte mit den Schlüsseln: • **icon**: Eine URL zum Icon der Topic-Card. • **name**: Der Name des Themas. • **id**: Die ID des Themas.
clusters	Array	Jeder Eintrag ist ein JSON-Objekt mit den Daten eines Clusters, s. C.1.2 für das Format.

C.1.2 Methode: *cluster*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
path	str	Nein	Der Play-Store-Pfad unter dem die Cluster-Übersicht zu finden ist.
maxCount	int > 0	Ja (30)	Die maximale Anzahl an Apps, die extrahiert werden sollen.

Ergebnis

Name	*Typ*	*Beschreibung*
token	str	Das Token, mit dem die nächste Seite von App-Cards extrahiert werden kann.
name	str	Der Name des Clusters (kann `null` oder leer sein).
clusterUrl	str	Die vollständige URL, unter der die Cluster-Übersicht aufgerufen werden kann.
apps	Array	Jeder Eintrag ist ein JSON-Objekt mit Informationen über eine App.

Das Objekt für eine App-Card enthält die folgenden Daten:

Name	*Typ*	*Beschreibung*
icons	Array	Ein Array mit Strings für die URLs zu allen Icons der App.
title	str	Der Name der App.
developer	Objekt	Ein JSON-Objekt mit den Schlüsseln **name** und **id** (beides Strings), die Name und ID des Entwicklers angeben.
description	str	Eine Kurzbeschreibung der App.

Name	*Typ*	*Beschreibung*
score	Objekt	Ein JSON-Objekt mit den Schlüsseln **text** und **value**, die die Durchschnittswertung der App als formatierten String und als Kommazahl angeben.
price	Objekt	Ein JSON-Objekt mit diesen Schlüsseln: • **price**: Der Preis der App als Ganzzahl, multiplziert mit 1 000 000 (z.B. 1790000 für 1,79). Ist einfach 0 für kostenlose Apps. • **currency**: Die Währungs-ID (ISO 4217). • **text**: Der formatierte Preis der App (Format landesabhängig).
originalPrice	Objekt	Entweder `null` oder ein JSON-Objekt mit der gleichen Struktur wie `price`. Gibt den Original-Preis an, falls die App gerade im Angebot ist.
purchaseLink	str	Eine URL, die direkt zu einem Kaufen/Instalieren-Dialog für die App führt.
offerEnd	Objekt	Ein JSON-Objekt mit den Schlüsseln **timestamp** und **text**. Ersterer ist ein Unix-Timestamp für das Ende des Angebots und letzterer ein sprach-abhängiger Text (z.B. „Angebot gilt noch 1 Tag“).
id	str	Die App ID.

C.1.3 Methode: *listApps*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
coll	Sammlungs-ID	Nein	Die Sammlung, aus der Apps extrahiert werden sollen. S. Abschnitt A.5.1 für eine Liste.

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
category	Kategorie-ID	Ja (`null`)	Optional eine Kategorie, auf die die Extraktion eingeschränkt werden soll. S. Abschnitt A.5.2 für eine Liste.
preview	bool	Ja (False)	Wenn ja, dann wird nur die Cluster-Vorschau für die Sammlung extrahiert, andernfalls die Apps aus der kompletten Cluster-Übersicht.
maxApps	int > 0	Ja (30)	Die maximale Anzahl an Apps, die extrahiert werden sollen (wird nur berücksichtig, wenn `preview == False` ist).
age	Alters-ID	Ja (`null`)	Nur für die Kategorie *FAMILY* gültig. Schränkt die Kategorie auf einen Alters-Bereich ein. S. Abschnitt A.5.2 für eine Liste.

Ergebnis

Das Ergebnis hat dasselbe Format wie bei `cluster` (s. Abschnitt C.1.2).

C.1.4 Methode: *categories*

Parameter

Keine zusätzlichen Parameter abseits der Standard-Parameter.

Ergebnis

Das Ergebnis ist ein JSON-Array, welches JSON-Objekte beinhaltet, die zwei Schlüssel haben. Der Schlüssel **translatedName** liefert den übersetzten Namen der Kategorie und **id** liefert die Kategorie-ID.

C.1.5 Methode: *developer*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
devID	str/int	Nein	Die Entwickler-ID.
maxApps	int > 0	Ja (30)	Die maximale Anzahl an Apps, die extrahiert werden sollen (wird nur berücksichtig, wenn `preview == False` ist).

Ergebnis

Das Ergebnis hat dasseble Format wie bei `cluster` (s. Abschnitt C.1.2).

C.1.6 Methode: *similar*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
appID	str	Nein	Die App ID.
preview	bool	Ja (False)	Wenn ja, dann werden App-Cards aus der Cluster-Übersicht extrahiert, andernfalls nur aus der Cluster-Vorschau.

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
maxApps	int > 0	Ja (30)	Die maximale Anzahl an Apps, die extrahiert werden sollen (wird nur berücksichtig, wenn `preview == False` ist).

Ergebnis

Das Ergebnis hat dasselbe Format wie bei `cluster` (s. Abschnitt C.1.2).

C.1.7 Methode: *details*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
appID	str	Nein	Die App ID.

Ergebnis

Name	*Typ*	*Beschreibung*
size	str	Beschreibt die Größe der App in einer menschenfreundlichen Form (z.B. „42M“).
appVersion	str	Die aktuelle Version der App.
osVersion	str	Die minimal erforderliche OS-Version.
price	Objekt	Genau wie bei **cluster**.
originalPrice	Objekt	Genau wie bei **cluster**.
purchaseLink	str	Genau wie bei **cluster**.
offerEnd	str	Genau wie bei **cluster**.
score	Objekt	Enthält zwei Schlüssel: **text** ist die als String nutzerfreundlich formatierte Durchschnittswertung der App und **value** ist die Durchschnittswertung als Kommazahl.

Name	*Typ*	*Beschreibung*
ratings	int	Die Anzahl der Nutzerwertungen.
reviews	int	Die Anzahl der Rezensionen.
histogram	Array	Gibt die Anzahl an Nutzerwertungen mit einer bestimmten Bewertung an. Die Array-Einträge sind die Anzahlen und die Indizes (+1) die zugehörigen Bewertungen (1-5) Z.B. liefert Index 2 die Anzahl der Wertungen mit 3 „Sternen“.
id	str	Die ID der App.
title	str	Der Titel der App.
description	str	Die vollständige Beschreibung der App.
summary	str	Die Kurzbeschreibung (Untertitel) der App.
screenshots	Array	Ein Array mit URLs zu Screenshots der App.
icon	str	Eine URL zu dem Icon der App.
banner	str	Eine URL zu dem Banner-Bild der App.
video	Objekt	Ein Objekt mit den Schlüsseln **ytVideoID**, **ytEmbedLink** und **ytThumbnail**, die jeweils die YouTube-Video-ID, YouTube-Einbettungslink und eine URL zu dem Thumbnail für das Video zu der App beschreiben. Falls kein Video existiert, ist bei dem Schlüssel statt dem Objekt der Wert `null` vorzufinden.

Name	*Typ*	*Beschreibung*
contentRating	Objekt	Beschreibt die Alters-Einstufung der App. Die Schlüssel sind: • **ratingText**: Der Einstufungs-Text (z.B. „USK ab 6 Jahren“). • **ratingIcon**: Eine URL zum Einstufungs-Icon. • **ratingReason**: Ein String, der die Gründe für die Einstufung bechreibt (z.B. „Abstrakte Gewalt“). • **interactiveElements**: Ein String, der die Möglichkeiten der Interaktion mit der App beschreibt (z.B. „Onlinekäufe“).
developer	Objekt	Liefert Daten über den Entwickler. DIe Schlüssel sind: • **id**: Die ID des Entwicklers. • **internalID**: Eine Art interne ID, genauer Zweck nicht klar. • **name**: Der Name des Entwicklers. • **support**: Eine Mail-Addresse für Support-Anfragen. • **website**: Eine URL zu der Webseite des Entwicklers. • **address**: Das Impressum des Entwicklers.
installs	Objekt	Ein Objekt mit den Schlüsseln **text**, **minInstalls** und **exactInstalls**, das die Anzahl der Installationen der App als nutzer-freundlicher Text, abgerundete Ganzzahl und ungerundete Ganzzahl angibt.
hasIAP	bool	Gibt an, ob die App In-App-Käufe hat.
IAPDetails	str	Beschreibt den Preisbereich der In-App-Käufe.
hasAds	bool	Gibt an, ob die App Werbung hat.
editorsChoice	bool	Gibt an, ob die App von der Redaktion empfohlen ist.
wearOS	bool	Gibt an, ob die App Google’s WearOS unterstützt.
familyLibrary	bool	Gibt an, ob die App die Android-Familienbibliothek unterstützt.

Name	*Typ*	*Beschreibung*
categoryID	str	Die ID der Kategorie, zu der die App gehört.
categoryName	str	Der Name der Kategorie, zu der die App gehört.
offeredBy	str	Ein String, der den Unternehmensnamen des App-Anbieters beschreibt (scheint immer „Google Commerce Ltd.“ zu sein).
releaseDate	str	Das Datum, an dem die App veröffentlich wurde. Das Format ist sprach-abhängig.
similar	Objekt	Die Cluster-Vorschau zu den ähnlichen Apps dieser App. Das Format ist wie bei **cluster** (s. Abschnitt C.1.2) definiert.

C.1.8 Methode: *reviews*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
appID	str	Nein	Die App ID.
maxCount	int > 0	Ja (15)	Die maximale Anzahl an Cluster-Sammlungen, die extrahiert werden sollen.
sort	int	Ja (`null`)	Eine Sortieroption. Kann weggelassen werden für die Standard-Sortierung. Möglich sind relevante Rezensionen (1), neue Rezensionen (2) oder nur Bewertungen (3).
score	int (1-5)	Ja (`null`)	Ein Filter für die Bewertung der Rezensionen.
device	str	Ja (`null`)	Eine Geräte-ID, um die Ergebnisse für dieses Gerät zu filtern. Nicht empfohlen, da Format der Geräte-IDs nicht bekannt.

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
token	str	Ja (`null`)	Das Token für die Seite mit der angefangen werden soll.

Ergebnis

Das Ergebnis ist ein JSON-Objekt mit einem Schlüssel **token** mit dem Token für die nächste Seite und einem Schlüssel **reviews**, ein Array mit den Rezensionen der App. Jede Rezension ist ein JSON-Objekt mit den folgenden Schlüsseln:

Name	*Typ*	*Beschreibung*
id	str	Die Review-ID.
userName	str	Der Nutzer-Name.
userIcon	str	Eine URL zu dem Icon des Nutzers.
score	int (1-5)	Die vom Nutzer angegebene Bewertung.
text	str	Der Text der Rezension. Kann `null` sein, falls es keinen gibt.
time	Objekt	Der Zeitpunkt der Rezension. Hat die Schlüssel **unixTime** (ein Unix-Timestamp auf die Sekunde genau) und **nanoTime** (Timestamp auf die Nanosekunde genau, auch wenn die letzten sechs Stellen anscheinend immer 0 sind).
thumbsUp	int	Die Anzahl an „Daumen hoch“.
devResponse	Objekt	`null`, falls der Entwickler nicht geantwortet hat. Andernfalls ein Objekt mit den Schlüsseln **name** (Name des Entwicklers), **text** (Antwort-Text) und **time** (Antwortzeit, wieder ein Objekt mit *unixTime* und *nanoTime*).
appVersion	str	Die Version der App für die die Rezension geschrieben wurde.

Name	*Typ*	*Beschreibung*
criteria	Objekt	Je nach Kategorie bietet der Play Store beim Abgeben einer Rezension die Möglichkeit, eine App nach bestimmten Kriterien mit einer Auswahl möglicher Antworten zu bewerten. Z.B. gibt es Fragen wie „Bietet diese App nützliche Benachrichtigungen“ mit den Antwortmöglichkeiten „Nein“, „Weiß nicht“ und „Ja“. Dieses Objekt ordnet Kriteriums-ID's entsprechende Antworten zu (diese sind immer als Zahl kodiert bzw. `null`, falls der Nutzer keine Antwort abgegeben hat).
isEdited	bool	Gibt an, ob die Rezension bearbeitet wurde.

C.1.9 Methode: *reviewHistory*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
appID	str	Nein	Die App ID.
reviewID	str	Nein	Die Rezensions-ID.

Ergebnis

Das Ergebnis hat exakt dasselbe Format wie für **reviews** (s. Abschnit C.1.8).

C.1.10 Methode: *search*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
term	str	Nein	Der Suchbegriff.

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
price	int	Ja (`null`)	Ein Filter für kostenlos (1)/kostenpflichtig (2) oder alle Apps (0).
androidId	str	Ja (`null`)	Eine Art ID, die es erlaubt nach Apps kompatibel zu einem Gerät zu filtern. Aufbau oder eine Liste solcher IDs ist unbekannt, daher ist die Nutzung nicht empfohlen.
maxApps	int > 0	Ja (30)	Die maximale Anzahl an Apps, die extrahiert werden sollen.

Ergebnis

Das Ergebnis hat exakt dasselbe Format wie für **cluster** (s. Abschnit C.1.2).

C.1.11 Methode: *suggest*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
term	str	Nein	Der Suchbegriff.

Ergebnis

Als Ergebnis wird ein Array aus Strings zurückgegeben, die die vervollständigten Sucbegriffe darstellen.

C.1.12 Methode: *permissions*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
appID	str	Nein	Die App ID.

Ergebnis

As Ergebnis wird ein JSON-Array zurückgegeben, die Einträge sind JSON-Objekte mit den folgenden Schlüsseln:

Name	*Typ*	*Beschreibung*
name	str	Name der Berechtigungs-Gruppe.
icon	str	Die URL zum Icon der Berechtigungs-Grupe.
descriptions	Array	Ein Array mit Beschreibungen (Strings) zu den einzelnen Berechtigungen.

C.1.13 Methode: *topic*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	str	Nein	Die Themen-ID.

Ergebnis

Das Ergebnis ist ein JSON-Objekt mit den Schlüsseln **name**, **description** und **banner**, die jeweils Namen, Beschreibung und die URL zum Banner-Bild eines Themas angeben. Darüberhinaus gibt es den Schlüssel **apps**, ein JSON-Array bei dem jeder Eintrag die Daten für eine empfohlene App beinhaltet. Die Struktur einer App ist wie folgt:

Name	*Typ*	*Beschreibung*
price	Objekt	Genau wie bei **cluster**.

Name	*Typ*	*Beschreibung*
originalPrice	Objekt	Genau wie bei **cluster**.
purchaseLink	str	Genau wie bei **cluster**.
offerEnd	Objekt	Genau wie bei **cluster**.
score	Objekt	Genau wie bei **details**.
ratings	int	Genau wie bei **details**.
id	str	Genau wie bei **details**.
title	str	Genau wie bei **details**.
description	String	Wie bei **details** die vollständige Beschreibung der App. Es kann aber sein, dass dies der leere String ist, s. dazu den Schlüssel `ecReasons`
screenshots	Array	Genau wie bei **details**.
icon	str	Genau wie bei **details**.
video	Objekt	Genau wie bei **details**.
contentRating	Objekt	Genau wie bei **details**.
developer	Objekt	Genau wie bei **cluster**.
installs	Objekt	Genau wie bei **details**.
categoryID	str	Genau wie bei **details**.
categoryName	str	Genau wie bei **details**.
ecReasons	Array	Ein Array von Strings, die angeben, warum diese App von der Redaktion empfohlen wird. Der Array kann leer sein. In diesem Fall ist die App-Beschreibung via `description` gegeben, andernfalls ist die Beschreibung der leere String.

C.2 Apple App Store

C.2.1 Methode: *details_api*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int/Array	Nein	Eine einzelne App ID oder ein Array von App ID's (Ganzzahlen).

Ergebnis

Wurde nur eine einzelne App ID angegeben, so erhält man das Ergebnis für diese App. Hat man einen Array von App ID's übergeben, so erhält man einen Array von Ergebnissen. Ein Ergebnis ist ein JSON-Objekt mit den folgenden Schlüsseln:

Name	*Typ*	*Beschreibung*
id	int	Die App ID.
bundleId	str	Die Bundle ID.
url	str	Die URL zur App-Detail-Seite.
title	str	Der Titel der App.
description	str	Die Beschreibung der App.
icon	str	Eine URL zu einem App-Icon.
allIcons	Array	Ein Array mit Icons, die durch JSON-Objekte beschrieben werden, die die Schlüssel width, width und url haben und jeweils Breite, Höhe (beide gazzahlig) und URL eines Icons beschreiben.
screenshots	Array	Ein Array mit URL's (Strings) für Screenshots von der App.

Name	*Typ*	*Beschreibung*
screenshotsByType	Objekt	Ein JSON-Objekt, bei dem die Schlüssel Bezeichner für einen Geräte-Typ sind, die Werte sind Arrays für Screenshots der App. Die Einträge dieser Arrays sind genau so aufgebaut wie bei `allIcons`.
size	int	Die Größe der App in Bytes.
appVersion	str	Die aktuelle Version der App.
osVersion	str	Die minimale OS-Version für die App.
price	Objekt	Ein JSON-Objekt mit diesen Schlüsseln: • **price**: Der Preis der App als Ganzzahl, 0 falls sie kostenlos ist. • **currency**: Die Währungs-ID (ISO 4217). • **text**: Der als String formatierte (landesabhängige) Preis der App.
score	Objekt	Ein JSON-Objekt mit den Schlüsseln **text** und **value**, die die Durchschnittswertung der App als formatierter String und als Kommazahl angeben.
currentScore	Objekt	Wie `score`, aber nur für die aktuelle Version der App.
ratings	int	Gibt die Anzahl der Nutzerwertungen an.
currentRatings	int	Wie `ratings`, aber nur für die aktuelle Version der App.
contentRating	Objekt	Ein JSON-Objekt mit den Schlüsseln **ratingText** (String) und **ratingReasons** (Array aus Strings), die jeweils eine Alters-Einstufung und eine Liste mit Gründen für diese Einstufung beinhalten.

Name	*Typ*	*Beschreibung*
developer	Objekt	Liefert Daten über den Entwickler. DIe Schlüssel sind: • **id**: Die ID des Entwicklers. • **name**: Der Name des Entwicklers. • **website**: Eine URL zu der Webseite des Entwicklers. • **seller**: Der Name des App-Anbieters (Unternehmensname).
changelog	str	Beschreibt die Änderungen der App in der letzten Version.
lastUpdate	str	Das Datum der letzten Aktualisierung (Format landes-abhängig).
releaseDate	str	Das Veröffentlichungsdatum (Format landes-abhängig).
categoryID	str	Die ID der Hauptkategorie der App.
categoryName	str	Der Name der Hauptkategorie der App.
allCategories	Array	Ein Array, der alle Kategorien der App beinhaltet. Jeder Eintrag ist ein JSON-Objekt mit den Schlüsseln **id** und **name** für ID und Name der Kategorie.
languages	Array	Ein Array, der die Sprach-Kürzel für alle unterstützten Sprachen der App auflistet.
supportedDevices	Array	Ein Array, der aus Strings besteht, die jeweils einen unterstützten Geräte-Typ bezeichnen.

C.2.2 Methode: *details_itunes*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Eine einzelne App ID.

Ergebnis

Das Ergebnis ist ein JSON-Objekt mit den folgenden Schlüsseln:

Name	*Typ*	*Beschreibung*
id	int	Wie bei **details_api**.
bundleId	str	Wie bei **details_api**.
url	str	Wie bei **details_api**.
title	str	Wie bei **details_api**.
description	str	Wie bei **details_api**.
summary	str	Eine Kurzbeschreibung (Untertitel der App).
copyright	str	Ein Copyright-String für die App.
eula	str	Eine WebObject-URL zu der EULA für die URL.
isPreorder	bool	Gibt an, ob die App schon veröffentlicht wurde.
familyLibrary	bool	Gibt an, ob die App die Familienbibliothek unterstützt.
hasMessagesExtension	bool	Genaue Bedeutung ist unklar.
isFirstPartyHideableApp	bool	Genaue Bedeutung ist unklar.
gameCenter	bool	Gibt an, ob die App Apple's Game Center unterstützt.
appleWatch	bool	Gibt an, ob die App die Apple Watch unterstützt.

Name	*Typ*	*Beschreibung*
passbook	bool	Gibt an, ob die App Apple Passbook unterstützt.
usesLocation	bool	Gibt an, ob die App im Hintergrund Standortdaten nutzt.
gameControllerRequired	bool	Gibt an, ob die App einen Spiele-Controller benötigt.
gameControllerSupported	bool	Gibt an, ob die App einen Spiele-Controller unterstützt.
hiddenFromSpringboard	bool	Gibt an, ob die App auf dem Springboard zu sehen ist.
is32BitOnly	bool	Gibt an, ob die App nur auf 32Bit-Systemen läuft.
siriSupported	bool	Gibt an, ob die App Apple's Siri unterstützt.
requiredCapabilities	String	Beschreibt die Voraussetzungen für diese App als Leerzeichen-getrennter String (z.B. „armv7 opengles“).
supportedDevices	Array	Wie bei **details_api**.
icon	str	Wie bei **details_api**.
allIcons	Array	Wie bei **details_api**.
screenshots	Array	Wie bei **details_api**.
screenshotsByType	Objekt	Wie bei **details_api**.
video	Objekt	Ein JSON-Objekt mit zwei Schlüsseln **thumb** und **video**, die die URL's zu einem Video-Thumbnail bzw. zu einer m3u8-Playlist für das Video beschreiben.

Name	*Typ*	*Beschreibung*
videoByType	Objekt	Ein JSON-Objekt, dessen Schlüssel Geräte-Bezeichner sind. Die Werte sind JSON-Objekte, die so aufgebaut sind wie der Schlüssel `video`, allerdings ist der Wert bei **thumb** nicht nur eine URL, sondern ein Array von mehreren Thumbnails verschiedener Auflösungen. Das Format des Arrays ist wie bei `screenshotsByType`.
banner	str	Die URL zum Banner-Bild für die App. Das Format ist wie bei `icon`.
allBanners	Array	Ein Array vom Banner-Bild in unterschiedlichen Auflösungen. Das Format ist wie bei `allIcons`.
size	int	Wie bei **details_api**.
sizeByDevice	Objekt	Ordnet Geräte-Bezeichnern die Größe der App in Bytes zu (ganzzahlig).
osVersion	int	Wie bei **details_api**.
privacyUrl	str	Die URL zur Datenschutzerklärung der App.
releaseDate	str	Wie bei **details_api**.
hasIAP	bool	Gibt an, ob die App In-App-Käufe hat.
requirementsString	str	Ein String, der Anforderungen an Geräte für die App beschreibt.
price	Objekt	Wie bei **details_api**, aber der Schlüssel `currency` fehlt.

Name	*Typ*	*Beschreibung*
offers	Array	Eine Array für alle Angebote der App. Enthalten sind JSON-Objekte mit denselben Schlüsseln wie bei `price` aber noch mit zwei zusätzlichen Schlüsseln: **buyParams** ist ein String von Kauf-Optionen formatiert wie URL-Parameter und **type** gibt den Typ des Angebots an (z.B. „get“, „buy„ oder „update“)
score	Objekt	Wie bei **details_api**.
currentScore	Objekt	Wie bei **details_api**.
ratings	int	Wie bei **details_api**.
currentRatings	int	Wie bei **details_api**.
contentRating	Objekt	Wie bei **details_api**.
developer	Objekt	Wie bei **details_api**, aber mit einem zusätzlichen Schlüssel **support**, der eine URL für den Support angibt.
categoryID	str	Wie bei **details_api**.
categoryName	str	Wie bei **details_api**.
allCategories	Array	Wie bei **details_api**.
appVersion	str	Wie bei **details_api**.
changelog	str	Wie bei **details_api**.
lastUpdate	str	Wie bei **details_api**.
versionHistory	Array	Ein Array, der die Versions-Geschichte der App beschreibt. Die Einträge sind JSON-Objekte mit den Schlüsseln **version** (Versions-Nummer), **notes** (Release-Notizen für die Version) und **date** (Veröffentlichungsdatum, Format sprach-abhängig).

Name	*Typ*	*Beschreibung*
IAPDetails	Array	Ein Array mit Daten zu In-App-Käufen. Jeder Eintrag ist ein JSON-Objekt mit den Schlüsseln **name** (Name des In-App-Kaufs) und **price** (beschreibt den Preis der App als lesbarer Text formatiert).
similar	Array	Ein Array von App ID's (Ganzzahlen) für ähnliche Apps.
moreByThisDev	Array	Ein Array von App ID's (Ganzzahlen) für andere Apps des Entwicklers.

C.2.3 Methode: *details_apphost*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
ids	int/Array	Nein	Eine einzelne App ID oder ein Array von App ID's.
options	Array	Ja	Ein Array von Optionen, die das Ergebnis beeinflussen. Eine Option ist eine Konstante im Programm. Die möglichen Optionen sind **website**, **support**, **devInfo**, **desc**, **eula**, **size**, **privacy**, **screens**, **video**, **version**, **similar**, **devApps**, **iap**.
authTok	str	Ja (`null`)	Das Autorisierungs-Token. Falls keines angegeben wird, wird ein extra Request gemacht, um das Token zu erhalten.

Ergebnis

Wenn nur eine ID gegeben wurde, dann ist das Ergebnis ein JSON-Objekt mit den App-Daten, ansonsten ein JSON-Array aus JSON-Objekten mit den Daten für die Apps. Solch ein JSON-Objekt ist wie folgt aufgebaut:

Name	*Typ*	*Option*	*Beschreibung*
developer	Objekt	-	Die Schlüssel **id** und **seller** geben die ID als Ganzzahl und den Anbieter der App als String an.
id	int	-	Wie bei **details_itunes**.
bundleId	int	-	Wie bei **details_itunes**.
url	str	-	Wie bei **details_itunes**.
title	str	-	Wie bei **details_itunes**.
summary	str	-	Wie bei **details_itunes**.
copyright	str	-	Wie bei **details_itunes**.
isPreorder	bool	-	Wie bei **details_itunes**.
familyLibrary	bool	-	Wie bei **details_itunes**.
hasMessagesExtension	bool	-	Wie bei **details_itunes**.
hasSafariExtension	bool	-	Genaue Bedeutung unklar (vermutl. Bezug auf den Safari Web Browser).
isFirstPartyHideableApp	bool	-	Wie bei **details_itunes**.
standaloneCompanionWOS	bool	-	Genaue Bedetunng unklar.
standaloneCompanionZulu	bool	-	Genaue Bedeutung unklar.
deliveredForWatchOS	bool	-	Genaue Bedeutung unklar.
deliveredForZulu	bool	-	Genaue Bedeutung unklar.
gameCenter	bool	-	Wie bei **details_itunes**.
appleWatch	bool	-	Wie bei **details_itunes**.
passbook	bool	-	Wie bei **details_itunes**.
usesLocation	bool	-	Wie bei **details_itunes**.

Name	*Typ*	*Option*	*Beschreibung*
gameControllerRequired	bool	-	Wie bei **details_itunes**.
gameControllerSupported	bool	-	Wie bei **details_itunes**.
hiddenFromSpringboard	bool	-	Wie bei **details_itunes**.
is32BitOnly	bool	-	Wie bei **details_itunes**.
requires32Bit	bool	-	Unterschied zu `is32BitOnly` unklar.
siriSupported	bool	-	Wie bei **details_itunes**.
supportsOcelot	bool	-	Genaue Bedeutung unklar.
firstVersionIAP	str	-	Vermutl. erste App-Version mit In-App-Käufen.
reviewsRestricted	bool	-	Genaue Bedeutung unklar.
requiredCapabilities	str	-	Wie bei **details_itunes**.
supportedDevices	Array	-	Wie bei **details_itunes**.
icon	str	-	Wie bei **details_itunes**.
allIcons	Array	-	Wie bei **details_itunes**, aber die URLs sind unformatiert[1].
releaseDate	str	-	Wie bei **details_itunes**.
hasIAP	bool	-	Wie bei **details_itunes**.
requirementsString	str	-	Wie bei **details_itunes**.
price	Objekt	-	Wie bei **details_api**.
offers	Array	-	Wie bei **detals_itunes**, aber der Schlüssel **currency** ist wie beim **details_api** auch dabei.
score	Objekt	-	Wie bei **details_itunes**.
ratings	int	-	Wie bei **details_itunes**.

Name	*Typ*	*Option*	*Beschreibung*
histogram	Array	-	Ein Array wobei der Index+1 für eine Wert zwischen 1 und 5 steht und der zugehörige Eintrag im Array die Anzahl der Bewertungen mit diesem Wert angibt.
contentRating	Objekt	-	Wie bei **details_itunes**.
categoryID	str	-	Wie bei **details_itunes**.
categoryName	str	-	Wie bei **details_itunes**.
allCategories	Array	-	Wie bei **details_itunes**, aber mit zwei zusätzlichen Schlüsseln **parentId** und **parentName**, die ID und Name der Oberkategorie angeben.
developer	Objekt	website	Der Schlüssel **website** gibt die URL der Webseite des Entwicklers an.
developer	Objekt	support	Der Schlüssel **support** gibt die URL der Support-Seite des Enticklers an.
developer	Objekt	devInfo	Der Schlüssel **name** ist der Name des Entwicklers und **logo** ein JSON-Objekt mit den Schlüsseln *width*, *height* und *url*, das das Logo-Bild des Entwicklers beschreibt.
description	str	desc	Wie bei **details_itunes**.
eula	str	eula	Der EULA-Text (kann sehr lang sein).
size	int	size	Wie bei **details_itunes**.

Name	*Typ*	*Option*	*Beschreibung*
sizeByDevice	Objekt	size	Wie bei **details_itunes**.
privacyUrl	str	privacy	Wie bei **details_itunes**.
screenshots	Array	screens	Wie bei **details_itunes**.
screenshotsByType	Objekt	screens	Wie bei **details_itunes**, aber die URLs sind unformatiert[1].
video	Objekt	video	Wie bei **details_itunes**.
videoByType	Objekt	video	Wie bei **details_itunes**, aber die Thumbnail-URLs sind unformatiert[1].
appVersion	str	version	Wie bei **details_itunes**.
changelog	str	version	Wie bei **details_itunes**.
lastUpdate	str	version	Wie bei **details_itunes**.
versionHistory	Array	version	Wie bei **details_itunes**.
similar	Array	similar	Wie bei **details_itunes**.
moreByThisDev	Array	devApps	Wie bei **details_itunes**.
IAPDetails	Array	iap	Ein Array von JSON-Objekten, die In-App-Käufe beschreiben.

Ein JSON-Objekt für In-App-Käufe (s. `IAPDetails`-Schlüssel) ist wie folgt aufgebaut:

Name	*Typ*	*Beschreibung*
id	int	Die ID des In-App-Kaufs.
merchEnabled	bool	Genaue Bedetung unklar.
merchVisibleByDefault	bool	Genaue Bedeutung unklar.
isSubscription	bool	Gibt an, ob es sich um ein Aboennement handelt.
offerID	str	Eine alternative ID (ähnlich wie die Bundle ID).
releaseDate	str	Veröffentlichungsdatum des In-App-Kaufs (Format landes-abhängig).

Name	*Typ*	*Beschreibung*
name	str	Der Name des In-App-Kaufs.
description	str	Eine Beschreibung des In-App-Kaufs.
price	Objekt	Wie bei **details_api**.
offers	Array	Wie bei **details_itunes**, aber der Schlüssel **currency** ist wie bei **details_api** auch dabei.

C.2.4 Methode: *developer_api*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die Entwickler-ID.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_api** (s. Abschnitt C.2.1).

C.2.5 Methode: *developer_apphost*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die Entwickler-ID.
offset	int >= 0	Ja (0)	Der Versatz in die App ID's des Entwicklers.
count	int > 0	Ja (30)	Die max. Anzahl der zu extrahierenden Apps.
options	Array	Ja	Wie bei **details_apphost**.
authTok	Array	Ja	Wie bei **details_apphost**.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_apphost** (s. Abschnitt C.2.3).

C.2.6 Methode: *listApps_old*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
coll	str	Nein	Eine Konstante für eine Sammlung. Eine Liste findet sich in Abschnitt B.7.1,
count	int > 0	Ja (30)	Die maximale Anzahl an Apps (max. 100!)
catID	int	Ja (`null`)	Eine Kategorie-ID. Scheint nicht zu funktionieren, daher nicht empfohlen. Eine Liste findet sich in Abschnitt B.7.3.

Ergebnis

Name	*Typ*	*Beschreibung*
id	int	Die ID der App.
bundleId	str	Die Bundle ID der App.
url	str	Die URL zur App-Detail-Seite.
title	str	Der Name der App.
icon	str	Die URL zum Icon der App.
allIcons	Array	Ein Array von Icon-URLs für die App.
description	str	Die Beschreibung der App.
price	Objekt	Wie bei **details_api**.
copyright	str	Ein copyright-String.
developer	Objekt	Die Schlüssel sind **id** und **name** für die ID des Entwicklers (int) und den Namen des Entwicklers (str).

Name	*Typ*	*Beschreibung*
categoryID	int	Die ID der Hauptkategorie der App.
categoryName	str	Der Name der Hauptkategorie der App.
releaseDate	str	Das Veröffentlichungsdatum der App (Format landesabhängig).

C.2.7 Methode: *listApps_new*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
coll	str	Nein	Eine Konstante für eine Sammlung. Eine Liste findet sich in Abschnitt B.7.2,
count	int > 0	Ja (30)	Die maximale Anzahl an Apps (max. 100!)
catID	int	Ja (`null`)	Eine Kategorie-ID. Scheint nicht zu funktionieren, daher nicht empfohlen. Eine Liste findet sich in Abschnitt B.7.3.

Ergebnis

Name	*Typ*	*Beschreibung*
id	int	Die ID der App.
url	str	Die URL zur App-Detail-Seite.
title	str	Der Name der App.
icon	str	Die URL zum Icon der App.
copyright	str	Ein copyright-String.
developer	Objekt	Die Schlüssel sind **id** und **name** für die ID des Entwicklers (int) und den Namen des Entwicklers (str).
categoryID	int	Die ID der Hauptkategorie der App.
categoryName	str	Der Name der Hauptkategorie der App.

Name	*Typ*	*Beschreibung*
allCategories	Array	Ein Array mit allen Kategorien der App. Die Einträge sind JSON-Objekte mit den Schlüsseln **id** und **name** für ID und Name der Kategorie.
releaseDate	str	Das Veröffentlichungsdatum der App (Format landesabhängig).

C.2.8 Methode: *reviews_rss*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die App ID
offset	int >= 0	Ja (0)	Der Versatz in die Rezensions-Liste.
count	int > 0	Ja (30)	Die maximale Anzahl an Rezensionen (max. 500-offset!)
sort	str	Ja (`null`)	Eine Sortier-Option. Möglich sind *mostRecent* und *mostHelpful*.

Ergebnis

Name	*Typ*	*Beschreibung*
userName	str	Der Nutzername.
userUrl	str	Eine URL zum Profil des Nutzers.
appVersion	str	Die Version der App, für die die Rezension geschrieben wurde.
score	int (1-5)	Die abgegebene Bewertung der App.
title	str	Der Titel der Rezension.
id	int	Die ID der Rezension.
text	str	Der Text der Rezension.

Name	*Typ*	*Beschreibung*
voteSum	int	Vermutlich die Anzahl an „Daumen hoch“ dieser Rezension.
voteCount	int	Vermutlich die Anzahl an „Daumen hoch“ und „Daumen runter“ dieser Rezension.

C.2.9 Methode: *reviews_apphost*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die App ID.
offset	int >= 0	Ja (0)	Der Versatz in die Rezensions-Liste.
count	int > 0	Ja (30)	Die maximale Anzahl an Rezensionen.
authTok	str	Ja (`null`)	Wie bei **details_apphost**.

Ergebnis

Name	*Typ*	*Beschreibung*
userName	str	Der Nutzername.
score	int (1-5)	Die abgegebene Bewertung der App.
title	str	Der Titel der Rezension.
text	str	Der Text der Rezension.
time	str	Das Datum der Rezension (Format landes-abhängig).
isEdited	bool	Gibt an, ob die Rezension bearbeitet wurde.

C.2.10 Methode: *search_api*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
term	str	Nein	Der Suchbegriff.
count	int > 0	Ja (30)	Die maximale Anzahl an Apps.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_api** (s. Abschnitt C.2.1).

C.2.11 Methode: *search_apphost*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
term	str	Nein	Der Suchbegriff.
offset	int >= 0	Ja (0)	Der Versatz in die App-Liste.
count	int > 0	Ja (30)	Die maximale Anzahl an Apps.
options	Array	Ja	Wie bei **details_apphost**.
authTok	str	Ja	Wie bei **details_apphost**.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_apphost** (s. Abschnitt C.2.3).

C.2.12 Methode: *similar_api*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die App ID.
offset	int >= 0	Ja (0)	Der Versatz in die App-Liste.
count	int > 0	Ja (30)	Die maximale Anzahl an Apps.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_api** (s. Abschnitt C.2.1).

C.2.13 Methode: *similar_apphost*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
id	int	Nein	Die App ID.
offset	int >= 0	Ja (0)	Der Versatz in die App-Liste.
count	int > 0	Ja (30)	Die maximale Anzahl an Apps.
options	Array	Ja	Wie bei **details_apphost**.
authTok	Array	Ja	Wie bei **details_apphost**.

Ergebnis

Ein JSON-Array von App-Details für die Apps des Entwicklers. Das Format ist das von **details_apphost** (s. Abschnitt C.2.3).

C.2.14 Methode: *suggest*

Parameter

Name	*Typ*	*Optional (Wert)*	*Beschreibung*
term	str	Nein	Der Suchbegriff.

Ergebnis

Ein JSON-Array dessen Einträge Strings sind, die die vervollständigten Suchbegriffe beinhalten.

Tabellenverzeichnis

Abbildungsverzeichnis

Literatur

[1] Georg Bradl und Contributors. *Sphinx - Python Document Generator*. 2020. URL: https://www.sphinx-doc.org/en/master/ (besucht am 19. 07. 2020).

[2] Wikipedia Contributors. *Cache stampede*. 24. Mai 2020. URL: https://en.wikipedia.org/wiki/Cache_stampede (besucht am 19. 07. 2020).

[3] Mozilla Foundation. *HTTP Authorization Header*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization (besucht am 11. 07. 2020).

[4] Python Software Foundation. *Data Model*. 19. Juli 2020. URL: https://docs.python.org/3/reference/datamodel.html (besucht am 19. 07. 2020).

[5] Python Software Foundation. *General Python FAQ*. 14. Juli 2020. URL: https://docs.python.org/3/faq/general.html (besucht am 14. 07. 2020).

[6] Python Software Foundation. *Python Built-in Functions*. 19. Juli 2020. URL: https://docs.python.org/3/library/functions.html (besucht am 19. 07. 2020).

[7] Python Software Foundation. *Python asyncio*. 19. Juli 2020. URL: https://docs.python.org/3/library/asyncio.html (besucht am 19. 07. 2020).

[8] David Goodger. *PEP 257 – Docstring conventions*. 29. Mai 2001. URL: https://www.python.org/dev/peps/pep-0257/ (besucht am 19. 07. 2020).

[9] Apple Inc. *Advanced Partner Linking*. URL: https://affiliate.itunes.apple.com/resources/documentation/linking-to-the-itunes-music-store/ (besucht am 03. 07. 2020).

[10] Apple Inc. *App Store*. 3. Juli 2020. URL: https://apps.apple.com/de/genre/ios/id36 (besucht am 03. 07. 2020).

[11] Apple Inc. *Apple Performance Partners Program*. URL: https://affiliate.itunes.apple.com/resources/ (besucht am 11. 07. 2020).

[12] Apple Inc. *Apple RSS Feeds*. URL: https://www.apple.com/rss/ (besucht am 11. 07. 2020).

[13] Apple Inc. *RSS Feed Generator*. URL: https://rss.itunes.apple.com/de-de (besucht am 11. 07. 2020).

[14] Apple Inc. *iTunes Search API*. URL: https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/ (besucht am 11. 07. 2020).

[15] Google Inc. *Play Store*. 3. Juli 2020. URL: https://play.google.com/store/apps/ (besucht am 03. 07. 2020).

[16] M. Jones, D. Hardt und Microsoft. *RFC 6750*. 1. Okt. 2012. URL: https://tools.ietf.org/html/rfc6750 (besucht am 11. 07. 2020).

[17] Daniel Liu. *play-scraper*. 15. Sep. 2019. URL: https://github.com/danieliu/play-scraper (besucht am 19. 07. 2020).

[18] Facundo Olano. *app-store-scraper*. 15. Juni 2001. URL: https://github.com/facundoolano/app-store-scraper (besucht am 19. 07. 2020).

[19] Facundo Olano. *google-play-scraper*. 16. Juni 2020. URL: https://github.com/facundoolano/google-play-scraper (besucht am 19. 07. 2020).

[20] Quentin Pradet. *How do you rate limit calls with aiohttp?* 16. März 2020. URL: https://quentin.pradet.me/blog/how-do-you-rate-limit-calls-with-aiohttp (besucht am 19. 07. 2020).

[21] L. Rabe. *Anzahl der verfügbaren Apps in den Top App-Stores im 2. Quartal 2019*. 4. März 2020. URL: https://de.statista.com/statistik/daten/studie/208599/umfrage/anzahl-der-apps-in-den-top-app-stores/ (besucht am 23. 06. 2020).

[22] Sebastián Ramírez. *uvicorn-gunicorn-fastapi*. 6. Juni 2001. URL: https://github.com/tiangolo/uvicorn-gunicorn-fastapi-docker (besucht am 19. 07. 2020).

[23] Sebastián Ramírez und Contributors. *FastAPI*. 19. Juli 2020. URL: https://github.com/tiangolo/fastapi (besucht am 19. 07. 2020).

[24] Raúl Rodríguez. *google-play-scraper*. 16. Juni 2019. URL: https://github.com/raulr/google-play-scraper (besucht am 19. 07. 2020).

[25] James Saryerwinnie. *JMESPath Specification*. 2015. URL: https://jmespath.org/specification.html (besucht am 19. 07. 2020).

[26] ASGI Team. *Asynchronous Server Gateway Interface*. 2018. URL: https://asgi.readthedocs.io/en/latest/ (besucht am 19. 07. 2020).

[27] EmberJS Team. *Ember.js - A framework for ambitious web developers*. URL: https://emberjs.com/ (besucht am 11. 07. 2020).

[28] WSGI Team. *Web Server Gateway Interface*. 25. Sep. 2011. URL: https://wsgi.readthedocs.io/en/latest/what.html (besucht am 19. 07. 2020).

[29] gunicorn Team. *gunicorn - Python WSGI HTTP server for Unix*. 8. Juli 2020. URL: https://github.com/benoitc/gunicorn (besucht am 19. 07. 2020).

[30] uvicorn Team. *uvicorn - A lightning-fast ASGI Server*. 17. Juli 2020. URL: https://github.com/encode/uvicorn (besucht am 19. 07. 2020).

[31] uvicorn Team. *uvicorn - Deplyoment*. URL: https://www.uvicorn.org/deployment/ (besucht am 19. 07. 2020).

[32] F. Tenzer. *Statistiken zu Smartphones*. 1. Apr. 2020. URL: https://de.statista.com/themen/581/smartphones/ (besucht am 23. 06. 2020).

[33] F. Tenzer. *Statistiken zur Smartphone-Nutzung in Deutschland*. 7. Mai 2020. URL: https://de.statista.com/themen/6137/smartphone-nutzung-in-deutschland/ (besucht am 23. 06. 2020).

[34] Fastboot contributors. *Fastboot - progressive enhancement for ambitious web apps*. URL: https://ember-fastboot.com/ (besucht am 11. 07. 2020).

Name: Manuel Schmid Matrikelnummer: 946951

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Manuel Schmid