Mobile Computing

Tilman Ginzel Rahel Habacker Jonas Theis

29. September 2015

Scrapp ist eine Android App mit dem Ziel via Push-Notifications mehrere Nutzer über Änderungen von beliebigen Inhalten von Websites zu benachrichtigen. Dafür kann der Nutzer bereits vorhandene Interaktionsfolgen, die festlegen welcher Inhalt von Interesse ist, abonnieren und Zugriffswege für sich konfigurieren. Das regelmäßige Überprüfen auf Änderungen geschieht automatisiert und wird von berechtigten Nutzern geteilt. Bei einer Änderung werden andere Nutzer des gleichen Abonnements benachrichtigt. Der geänderte Inhalt kann dann innerhalb der jeweiligen App heruntergeladen und angezeigt werden.

1 Einleitung

Websites bieten oft keine Möglichkeit Benachrichtigungen zu senden. Dies zwingt den Nutzer, wenn sich spezifische Inhalte ändern, regelmäßig manuell nach Änderungen zu suchen. Dazu ruft er die entsprechende Website wiederholt auf, siehe Abbildung 1. Dieser Vorgang wird „pollen“ genannt. Dabei steigt der (Zeit-)Aufwand durch Navigation auf die gewünschte Seite und erneutes Eingeben spezieller Daten wie zum Beispiel Login- oder Formular-Daten ist notwendig. Kleinteilige Änderungen sind zusätzlich schwer erkennbar. Die Häufigkeit des Pollens steht zudem meist nicht in Relation zu den tatsächlichen Änderungen. Wenn mehrere Nutzer sich für die gleichen Inhalte interessieren, fragen sie oft die gleichen Inhalte beziehungsweise Änderungen ab, was gemeinsam redundant ist.

Die Ideallösung wäre die Webanwendung zu ändern, dies ist jedoch häufig nicht möglich oder nicht gewünscht. Insbesondere bei Anwendungen, die Inhalte wie Metadaten von Unternehmen oder Studierenden von Hochschulen verwalten, ist eine Anpassung keine Möglichkeit.

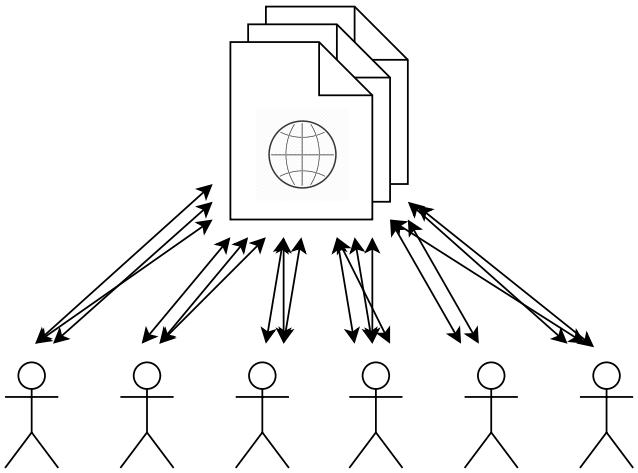

Abbildung 1: Häufiges Anfragen einer Website durch viele Nutzer

Eine Alternative ist automatisiertes Pollen von Websites und Verteilen der Informationen über Änderungen an andere Nutzer. Scrapp realisiert solch eine verteilte Automatisierung des Pollens. Dazu werden die Navigation auf die Seite und die Betrachtung der gewünschten Informationen auf dem Smartphone automatisiert. Durch die lokale Verarbeitung wird die Weitergabe von sensiblen oder persönlichen Daten vermieden. Falls geänderte Informationen vorliegen, werden andere interessierte Nutzer über die Tatsache anonym informiert. Dabei wird nur die Information ob und nicht welche Änderung stattgefunden hat geteilt und allen berechtigten Nutzern zugänglich gemacht, was die Ansicht persönlicher Bereiche ermöglicht.

In Kapitel 2 werden Anforderungen basierend auf alltäglichen Anwendungsbeispielen genannt. Kapitel 3 beschreibt die konzeptionellen Aspekte wie das Datenmodell und die Systemarchitektur. Anschließend wird in den Kapiteln 4 und 5 der Prozess des automatisierten Scrapens im Detail erklärt. Kapitel 6 geht auf die technischen Aspekte bei der Client-Server-Kommunikation ein.

2 Idee

Die Grundidee, das Pollen von Websites zu automatisieren, entsteht aus realen Szenarios. Diese in Abschnit 2.1 vorgestellen Beispiele beruhen auf unterschiedlichen Kontexten beziehungsweise Websites. Trotzdem ist der Benutzer jeweils grundlegend gleich motiviert: Er möchte die Änderung eines spezifischen Inhalts erkennen und abrufen, muss dafür aber oft manuell pollen.

Zur Umsetzung der Idee, den Aufwand des Pollens zu reduzieren, soll der Prozess automatisiert werden. Abschnitt 2.2 analysiert daher die Abfolge einzelner Schritte der Nutzer um

Änderungen zu erkennen und dies mit anderen Nutzern zu teilen. Durch das Teilen muss ein Nutzer seltener pollen, bekommt aber dennoch alle für ihn wichtigen Informationen zeitnah mit.

2.1 Anwendungsbeispiele

Beispiel	Art	Bedingung
QIS	kollaborativ	nein
Blog	kollaborativ	nein
MI-Druckkontigent	exklusiv	ja
Preisänderung	kollaborativ	ja

Tabelle 1: Szenarios

Im 3. Semester des Studiengangs Medieninformatik an der Hochschule RheinMain belegen die Studierenden das Fach Mathe für Informatiker. Im Prüfungsanmeldezeitraum tragen sie sich im Notensystem Qualitätssteigerung der Hochschulverwaltung im Internet durch Selbstbedienung (QIS) für diese Prüfung ein. Nachdem die Prüfung geschrieben wurde, besuchen viele Studierende, die mitgeschrieben haben, das QIS häufig. Sobald die Klausur korrigiert ist, trägt der Professor die Noten ein. Manche Studierende posten dann in öffentlichen Foren, dass die Noten für dieses Fach online sind.

Ein Blogger schreibt auf seiner Website Artikel zu verschiedenen Themen. Nutzer, die sich für bestimmte Themen interessieren, wünschen Benachrichtigungen, wollen aber nicht ihre Kontaktdaten, wie zum Beispiel E-Mail-Adresse hinterlassen. Wenn die Website RSS anbietet kann ein Nutzer die RSS-Feeds abonnieren. Neue Beiträge erscheinen dann in seinem Feedreader, den er trotzdem noch selbst öffnen muss.

Studierende des Studiengangs Medieninformatik der Hochschule Rhein-Main verfügen pro Semester über ein Druckkontingent. Startet ein Studierender einen Druckauftrag, wird dieser möglicherweise nicht ausgeführt. Um die Ursache herauszufinden, besucht er das MI-Portal und loggt sich von dort auf dem Server ein. Es werden seine Druckaufträge, die jeweiligen Kosten und der aktuelle Stand angezeigt. Nun kann er feststellen, ob die Kosten des Druckauftrags sein verbliebenes Druckkontingent übersteigen, und darauf reagieren, indem er das Kontigent auflädt oder den Druckauftrag abbricht. Hier ist die Information für jeden Nutzer individuell, daher ist er nur an der Extraktion von Information interessiert, nicht am Teilen mit anderen.

Der Nutzer findet auf einer Website ein kürzlich eingeführtes Produkt, für welches er sich interessiert. Da es ihm noch zu teuer ist, schaut er regelmäßig auf die Produktseite, um eine Preissenkung zu bemerken. Sinkt der Preis unter den persönlich festgelegten Betrag, kauft er den Artikel.

2.2 Anforderungen

Aus den Szenarios lassen sich folgende Anforderungen extrahieren. Wesentliche Teilaspekte des Prozesses sind das Abonnieren von Änderungen, der Umgang mit Websites, das Feststellen und das Teilen von Änderungen.

Ein Szenario setzt das Interesse eines Nutzers an einer bestimmten Information voraus. Beispielsweise der Blog, bei dem sich Leser für neue Artikel interessieren. Um diese Tatsache abbilden zu können, werden Abonnements benötigt. Ein Abonnement stellt die Verbindung zwischen einem Nutzer und einer Website dar. Dies bedeutet er möchte über bestimmte Änderungen der Website infomiert werden und ist berechtigt diese Information mit Nutzern gleichen Interesses zu teilen, ähnlich dem Mitteilen der neuen Note in Foren.

Die in den Szenarien bisher manuell vorgenommenen Schritte wie Navigation zur Webseite, Suchen des relevanten Abschnitts und mögliches Mitteilen an andere Nutzer sollen zeitlich automatisiert werden. Dazu gehören der dem Ziel entsprechende Umgang mit Websites und die indirekte Kommunikation mit anderen Nutzern über eine zentrale Instanz. Der Umgang mit Websites umfasst die Navigation auf die gewünschte Seite, das Herunterladen und Analysieren, was auch „scrapen“ genannt wird. Zur Festlegung und Parametrisierung dieser Schritte ist eine einheitliche Definition notwendig. Die Definition oder Vorschrift konkretisiert welche Aktionen pro Schritt gegeben sind, wann und wie sie ausgeführt werden und welche Nutzerinformationen gebraucht werden.

Nach dem Herunterladen einer Website ist es notwendig, Teile zu extrahieren, diese auf Änderungen zu überprüfen und mit einfachen Bedingungen auszuwerten, was anstelle der menschlichen Kognition mit Ausdruckssprachen möglich ist. Dabei gibt es für kollaborative Szenarien einen Unterschied, welcher Inhalt auf einer Webseite eine Änderung bedeutet und welcher Inhalt dem Nutzer hinterher angezeigt wird. Im Beispiel ist ein neuer Eintrag in der Notenliste für die entsprechende Prüfung die Änderung, die unter den Studierenden geteilt wird, der Wert (die Note) darin aber das was jeden Studierenden persönlich interessiert.

Als Beispiel für das Scrapen einer Website dient das Szenario des Notenabrufens aus dem QIS. Die erste Aktion ist das Aufrufen der Startseite. Die zweite Anfrage, bei der ein Studierende sich einloggt, füllt die vorgesehenen Login-Felder mit den Authentifizierungsdaten. Er ist für die folgenden Anfragen dann eingeloggt, sodass der Zugriff auf nur für ihn sichtbare Informationen gelingt. Danach folgen seine Navigationsschritte zur Zielseite und das Anschauen des Inhalts, was durch Herunterladen und Speichern erreicht werden kann. Der Studierende weiß direkt, ob eine neue Zeile in der Notentabelle hinzu gekommen ist, da er den Stand vom letzten Nachsehen kennt. Daher muss der relevante Teil extrahiert und mit der Information aus einem vorherigen Vorgang verglichen werden.

Die in Abbildung 1 dargestellte Redundanz beim manuellen Abrufen von Inhalten erfordert automatisches Benachrichtigen anderer Interessenten über Änderungen. Findet zum Bei-

spiel ein Blog-Leser für ein Thema einen neuen Artikel, sollte er dies den Nutzern nach dem Prinzip „einer für alle“ mitteilen. Sie müssen dann nicht selbst feststellen, dass ein neuer Artikel hinzugekommen ist. Die Frequenz des manuellen Pollens sinkt so für jeden Nutzer erheblich. Dieses Prinzip funktioniert gut, wenn sich mehrere Nutzer für eine Änderung interessieren.

Die Szenarios machen deutlich, dass für das Mitteilen eine Nutzeridentifizierung erforderlich ist. Damit ein Nutzer einem anderen von einem neuen Blog-Artikel erzählen kann und es denjenigen auch interessiert, müssen beide sich kennen und sich für den Blog interessieren oder brauchen einen Vermittler, der beide kennt und die Kontrolle behält was einander mitgeteilt wird.

3 Konzept

Um Interessen an verschiedenen Informationen abbilden zu können, ermöglicht Scrapp es Nutzern, Regeln zu abonnieren. Ein Abonnement ordnet Nutzern eindeutig Regeln zu. Diese sind Vorschriften, die für den Nutzer konfigurierbar sind und dienen dazu Änderungen festzustellen. Sie bestehen aus verschiedenen parametrisierbaren Aktionen, die beim Anwenden der Regel hintereinander ausgeführt werden. Wird eine Regel ausgeführt gibt es ein Ergebnis, was den spezifischen Inhalt enthält. Wie die Daten auf Client und Server abgelegt werden, ist in Abschnitt 3.1 beschrieben.

Als zentrale Instanz zur Datenhaltung und Kommunikation mit und zwischen Clients existiert ein Server, der in Abschnitt 3.2 beschrieben wird. Die Kommunikation besteht aus dem Entgegennehmen von REST-Anfragen der Clients und dem Senden von Notifications an Clients.

Abbildung 2 zeigt den Ablauf und das Prinzip von Scrapp. Dabei sind beliebig viele Websites und Clients sowie ein Server beteiligt. Die App navigiert zu einem regelmäßigen, konfigurierbaren Zeitpunkt oder manuell durch den Nutzer auf einer Website zu der Zielseite und lädt den Inhalt herunter (1). Danach parst sie den Inhalt und entscheidet, ob neue Informationen vorhanden sind (2). Existiert eine Änderung, wird der Server darüber, dass, nicht aber welcher Inhalt geändert wurde, informiert (3). Der Server sagt allen Abonnenten dieser Regel Bescheid (4), woraufhin Sie sich die neuen, möglicherweise sensiblen Inhalte selbst herunterladen können.

3.1 Datenmodell

Abbildung 3 zeigt das ER-Modell von Scrapp. Sowohl Client als auch Server führen jeweils eine eigene Datenbank. Dabei ist der Server maßgebend und der Client aktualisiert Inhalte gegebenenfalls via REST vom Server.

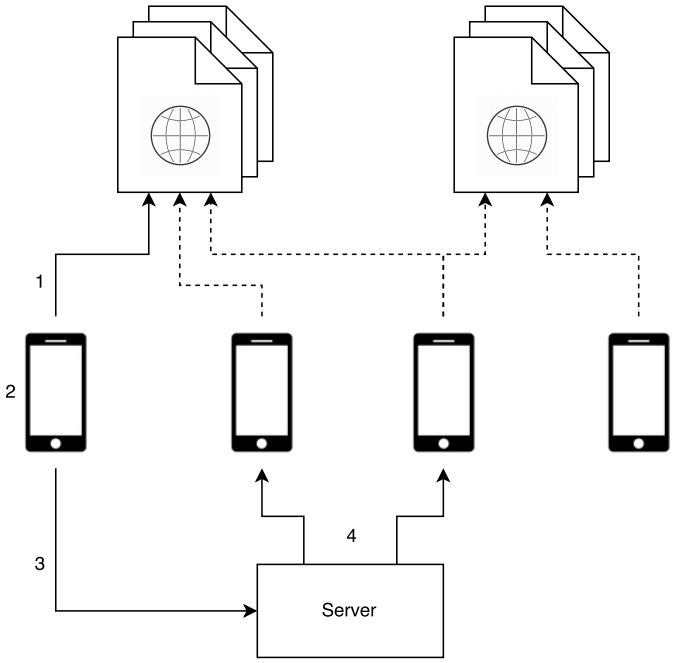

Abbildung 2: Roundtrip

Der Server generiert zu jedem Nutzer eine ID und speichert diese zusammen mit dem Identitäts-Token und dem Google-Cloud-Messaging-Token (GCM). Über das Identitäts-Token kann ein *User* immer eindeutig identifiziert werden. Um den Benutzer benachrichtigen zu können wird das GCM-Token gespeichert.

Eine *Rule* besteht aus einer ID, einem Titel und einer Beschreibung. Weiterhin besteht sie aus mehreren *Actions* und kann von Benutzern abonniert werden.

Die Entität *Subscription* gibt es nur im Schema des Servers. Sie gibt an, welcher Benutzer welche Regel abonniert hat. Wenn ein Benutzer in der App eine Regel nicht mehr abonnieren möchte, wird im Client die Spalte *subscribed* in der Tabelle *Rule* geändert und auf dem Server das *deleted_at*-Datum gesetzt. Da die Historie mitsamt Inhalt – nur auf dem Client vorhanden – sehr groß werden kann, wird diese im gleichen Zuge vom Gerät gelöscht. Bei erneutem Abonnieren kann die Historie von Ergebnissen für den Benutzer wiederhergestellt werden. Desweiteren besteht eine *Subscription* aus *start_time* und *interval*, welche den Startzeitpunkt und die Häufigkeit des automatisierten Scrapens für die abonnierte Regel angibt.

Zu einer *Subscription* gibt es beliebig viele *Results*. Ein *Result* besteht aus einer ID, einem Hash des eigentlichen Inhalts und dem Scrapezeitpunkt. Wichtig hierbei ist, dass auf dem Server nur ein Hash des Inhalts, nicht aber der Inhalt selbst gespeichert wird. So kann ohne den Inhalt zu kennen verglichen werden, ob dieser sich seit dem letzten Scrapezeitpunkt geändert hat. Auf dem Client wird zusätzlich noch der Inhalt gespeichert, um die-

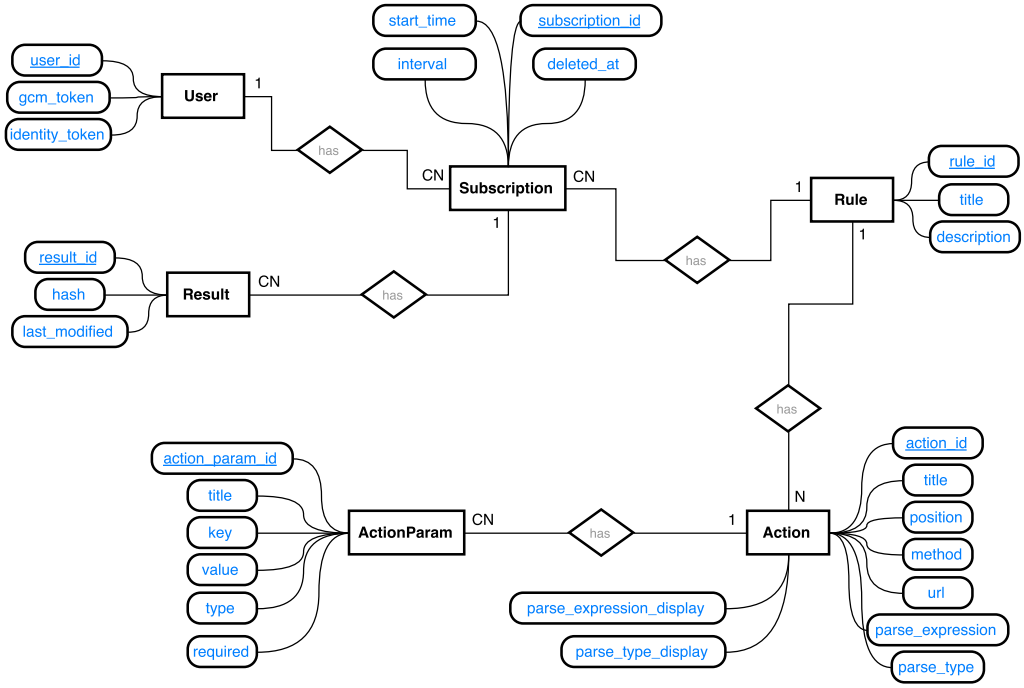

Abbildung 3: Entity-Relationship-Modell

sen anzuzeigen und eine Historie über alle eigenen Scrapevorgänge darzustellen. Dabei wird auf die Vorgabe durch die letzte *Action* der Regel geachtet. Wenn diese eine *parse_expression_display* hat, wird dieser Ausdruck zur Extrahierung der Daten für den Inhalt verwendet und die *parse_expression* für den Hash des Ergebnisses.

Eine *Action* besteht aus einer ID, einem Titel, einer Position innerhalb der Abfolge von *Actions* zu einer *Rule*, einer HTTP-Methode, einer URL und einem optionalen Parse-Ausdruck und Parse-Typ. Außerdem kann die letzte *Action* zusätzlich einen Parse-Ausdruck inklusive Typ speziell für die Anzeige haben. Zu einer *Action* kann es beliebig viele *ActionParams* geben.

Die Entität *ActionParam* besteht aus einer ID, einem optionalen Titel, einem Schlüssel, einem Wert und einem Typ, der die Anzeige des Eingabefeldes in der App bestimmt. Die *ActionParams* werden als Schlüssel-Wert-Paare an einen Request, der aus einer *Action* zusammengebaut wird, angehängt.

Alle Entitäten haben zusätzlich zu den in Abbildung 3 abgebildeten Eigenschaften noch die beiden *created_at* und *updated_at*, die jeweils das Erstellungsdatum beziehungsweise das Datum der letzten Änderung in koordinierter Weltzeit (UTC) angeben. Dies hat den Vorteil, das der Server-Standort und der Aufenthaltsort des Benutzers keinen Einfluss auf die gespeicherte Zeit haben. Erst bei der Anzeige, wird die Zeit in das aktuelle Locale des Nutzers konvertiert und ist somit immer richtig.

3.2 Systemarchitektur

Scrapp basiert auf einer Client-Server-Architektur, wobei die Android-App die Rolle des Clients übernimmt und mit einem Server kommuniziert. Als zusätzliche Komponente dient der GCM-Server zum Senden von Nachrichten an konkrete Apps. Abbildung 4 zeigt die Kommunikationswege zwischen den drei Komponenten.

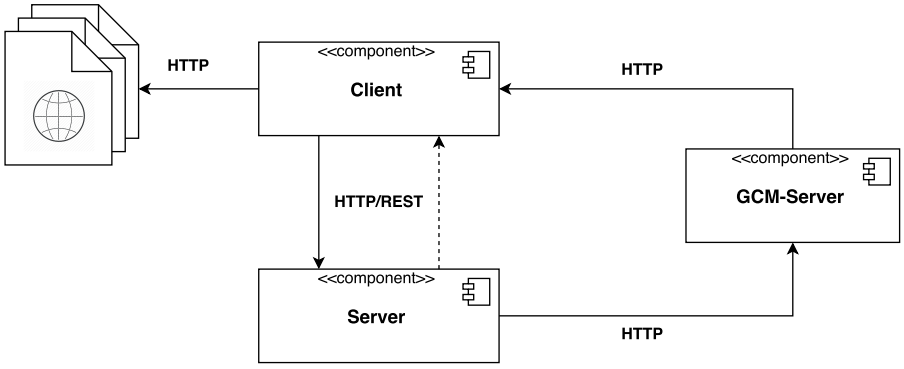

Abbildung 4: Systemarchitektur

Der Client wurde gemäß dem ersten von Virgil Dobjanschi vorgeschlagenen Pattern [3] implementiert. Das Pattern beschreibt wie REST-Anfragen asynchron ausgeführt und nach Erfolg- oder Misserfolg die Ansicht aktualisieren werden kann.

Der in Python implementierte Server ist als zentrale Instanz für die Datenhaltung und -verteilung an die verschiedenen Clients zuständig. Er bedient die Anfragen über die REST-Schnittstelle um beispielsweise gehashte Ergebnisse zu speichern. Über den GCM-Server kann er Push-Notifications an konkrete Clients senden. Dadurch können Änderungen an andere Nutzer verteilt werden. Er wurde mit Flask [1] und Flask-restful [2] entwickelt und als Datenbank wird PostgreSQL [6] verwendet.

4 Scrape-Vorgang

Scrapp traversiert über eine Abfolge von Webpages, die mittels *Actions* zu einer *Rule* vorgegeben sind. Dabei wird ein Benutzer simuliert, der sich durch eine Website klickt, sich beispielsweise einloggt und nach Änderungen oder Neuigkeiten umschaut. Dies geschieht entweder automatisiert nach konfiguriertem Zeitintervall oder wird direkt durch den Benutzer angestoßen. Der Scrape-Vorgang lässt sich in drei – zum Teil wiederholende – Abschnitte aufteilen. Crawlen beschreibt das schrittweise Besuchen von Webpages, Parsen die Ermittlung des Inhalts an einer bestimmten Stelle um danach gegebenenfalls eine Änderung festzustellen. Diese Bereiche werden im Folgenden genauer erläutert.

4.1 Crawlen

Crawling ist der erste Arbeitsschritt im Scrape-Vorgang und beschreibt das schrittweise Besuchen von Webpages. Für jede Aktion wird der Reihe nach geprüft, ob Daten an den Request angehängt werden müssen, also ob Aktionsparameter vorhanden sind, und diese in eine Map gespeichert. Danach wird die HTTP-Methode ermittelt und mittels der Bibliothek jsoup [4] der Request mit entsprechend GET- oder POST-Daten und eventuell vorhandenen Cookies aus den Requests der vorherigen Aktionen abgeschickt. Als nächstes werden vorhandene Cookies für die nächsten Requests gespeichert. Nun beginnt der Prozess des Parsens, welcher im nächsten Abschnitt genauer beschrieben wird. Wenn das Parsen abgeschlossen ist und noch weitere Aktionen vorhanden sind, beginnt das Crawlen wieder von vorne gegebenenfalls mit einer URL, die beim Parsen ermittelt wurde oder durch die Aktion gegeben ist.

Beim Crawlen besonders zu beachten sind Session-Cookies, die abhängig vom *User-Agent* sind und gegebenenfalls Einmal-Links erzeugen. Hierbei ist es wichtig, einem Klickpfad zu folgen und nicht direkt auf eine Unterseite innerhalb des Portals zu springen. Innerhalb von Scrapp gibt es auch die Möglichkeit die Website mittels *WebView* aufzurufen, ohne dass der Benutzer sich bei dieser einloggen muss. Aufgrund der möglichen Abhängigkeit zum *User-Agent* müssen diese zum Crawlen und beim Aufruf der Website exakt gleich sein.

Speziell HTML-Weiterleitungen stellen eine weitere Herausforderung beim Crawlen dar. Diese müssen programmatisch erkannt und weiterverfolgt werden, da die Webpage nur einen HTTP-Statuscode 200 sendet. Daher kann nicht anhand dem Statuscode entschieden werden, ob eine Weiterleitung oder ein erfolgreicher Request vorliegt.

Ein allgemeines Problem beim automatisierten Verarbeiten von Websites ist die Fehlkonfiguration einiger Webserver. Diese geben im Fehlerfall nicht immer einen richtigen HTTP-Statuscode zurück, sondern verstecken den Fehler innerhalb der Seite. So kann es unter Umständen extrem schwierig sein Fehler zu erkennen oder auf diese zu reagieren.

4.2 Parsen

Das Parsen beginnt direkt nachdem eine Response des Webservers der Website empfangen wurde. Mit Hilfe von jsoup [4] wird das HTML-Dokument in eine XHTML valide Form umgewandelt. Auf diesem Document-Object können nun beliebige CSS-Selektor-Ausdrücke mittels jsoup angewendet werden. Da das Dokument in einer XHTML validen Form ist, können auch XPath-Ausdrücke darauf angewendet werden. Dies geschieht mittels Javas eigener XPath Komponente aus dem Paket *javax.xml.xpath*. Bei beiden Parsevorgängen wird ein String ausgegeben, welcher möglicherweise weiterhin HTML-Schnipsel enthält.

Häufig wird beim Parsevorgang nur eine URL aus einem Link extrahiert, um mit diesem auf die nächste Seite im Klickpfad zu gelangen.

4.3 Feststellen von Änderungen

Nach dem letzten Abrufen der Website-Inhalte im Client wird zuerst geprüft ob Inhalt geparst werden konnte und gegebenenfalls dem Benutzer eine Fehlermeldung angezeigt. Wenn alles glatt gelaufen ist, wird das gehashte Ergebnis an den Server geschickt und der geparste Inhalt lokal in die Datenbank gespeichert. Auf dem Server findet dann die eigentliche Überprüfung auf Neuheit statt. Um neue Änderungen feststellen zu können, existiert auf dem Server eine Relation, die für jede Regel das aktuellste Ergebnis mit einem eindeutigen Hash speichert. Gibt es für eine Regel noch kein aktuellstes Ergebnis, ist ein am Server ankommendes Ergebnis auf jeden Fall das Neuste. Dann werden alle Abonnenten direkt via Push-Notification benachrichtigt. Existiert bereits ein Ergebnis, wird zuerst anhand des Hash-Wertes überprüft, ob das Ergebnis aktueller ist.

5 Scheduling des Pollens

Der Nutzer kann in Scrapp auf zwei verschiedene Arten den Scrape-Vorgang starten. Entweder er löst ihn manuell in der App aus, oder konfiguriert das regelmäßige Pollen. Letzteres wird in diesem Kapitel geschildert und ist der letzte Schritt zur Automatisierung des Scrape-Prozesses.

Automatisches Pollen bedeutet, dass das Scrapen für eine Regel nicht durch den Nutzer aktiv initiiert, sondern automatisch von Scrapp gestartet wird. Dabei legt ein zu konfigurierendes Intervall die Regelmäßigkeit, also in welcher Häufigkeit gescraped wird, fest. Der Startzeitpunkt ist ebenfalls konfigurierbar. Diese Funktionalität bewirkt, dass der Nutzer nicht mehr manuell nach Änderungen suchen muss, sondern diese immer automatisch mitbekommt. Die Zeit zwischen dem Auftreten und Feststellen einer Änderung ist abhängig von der Nutzerzahl, welche die entsprechende Regel regelmäßig scrapen und wie häufig der eigene Client automatisch scraped.

Die Einstellungen hierfür sind für jede Regel individuell und unabhängig von anderen Regeln. Auf der Konfigurationsseite ist es möglich den Startzeitpunkt und das Intervall festzulegen. Dabei kann der Nutzer zwischen Minuten, Stunden, Tagen oder Wochen wählen und die Anzahl einstellen. Beim Abonnieren einer Regel ist die Startzeit defaultmäßig der Zeitpunkt, an dem die Regel vom Server abgerufen wurde, und die Anzahl der Minuten 0.

Das automatische Scrapen ist mithilfe des *AlarmManagers* von Android umgesetzt. Damit hat Scrapp Zugriff auf die Alarm-Dienste des Systems. Abonniert oder konfiguriert ein Nutzer eine Regel mit Startzeitpunkt und Intervall, wird für diese Regel ein neuer Alarm

mit diesen Parametern erzeugt. Das System löst dann ab dem Startzeitpunkt im Abstand der eingestellten Zeit einen Alarm aus. Scrapp erfährt per Broadcast davon und startet den nächsten Scrape-Vorgang. Wird dabei eine Änderung festgestellt, bekommt er nur in diesem Fall eine Push-Nachricht.

Einige Überlegungen kostet das Format beziehungsweise die Einstellung des Intervalls. Dabei ist es wichtig flexible Intervallzeiten anzubieten und gleichzeitig den Nutzer nicht mit zu vielen Einstellungsmöglichkeiten zu verwirren. Scrapp deckt diesbezügliche mögliche Intervalle von einer Minute bis einem Jahr ab. Nach einem Neustart des Smartphones werden alle Alarme des Systems gelöscht. Daher reagiert Scrapp darauf und stellt nach dem Neustart alle Alarme für abonnierte Regeln wieder her. Die Präzision der Alarme ist nicht immer kontrollierbar und daher manchmal geringer, sodass Alarme gelegentlich im falschen Intervall ausgelöst werden. Ein Beispiel ist das Festlegen eines Startzeitpunktes in der Vergangenheit. In diesem Fall wird der erste Alarm direkt ausgelöst, auch wenn anhand des Intervalls an diesem Zeitpunkt kein Alarm vorgesehen ist.

6 Datenkommunikation

Das folgende Kapitel beschreibt die Kommunikation zwischen Client und Server. Hierbei werden beispielhafte Abläufe genannt sowie die REST-Schnittstelle beschrieben.

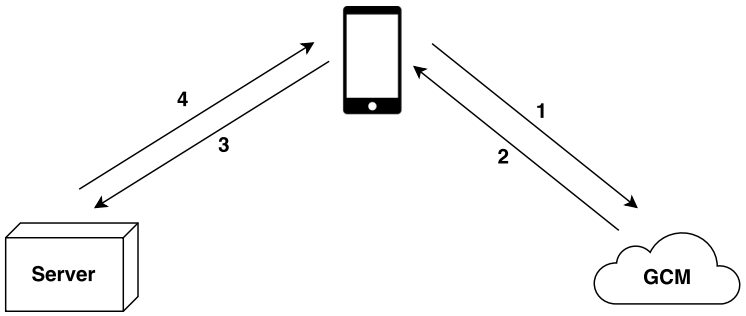

Abbildung 5: Ablauf der Nutzer-Registrierung

Damit der Server mit konkreten Clients kommunizieren kann, muss er diese eindeutig identifizieren können. Ebenso muss der Client sich bei jeder Anfrage authentifizieren können. Abbildung 5 zeigt den Registrierungsvorgang, welcher beim ersten App-Start im Hintergrund durchgeführt wird. Zu Beginn fragt der Client beim GCM-Server nach einem eindeutigen GCM-Token (1, 2). Dieses wird dann an den Server gesendet (3) und zusammen mit einem dort generierten Identitäts-Token gespeichert. Dieses Identitäts-Token wird nun zurück an den Client gesendet (4), welches zukünftig als Authentifizierung bei jeder REST-Anfrage an den Server als Header-Parameter mitgesendet wird.

Abbildung 6 zeigt den Ablauf für den Vorgang, alle Regeln vom Server abzufragen. Es wird

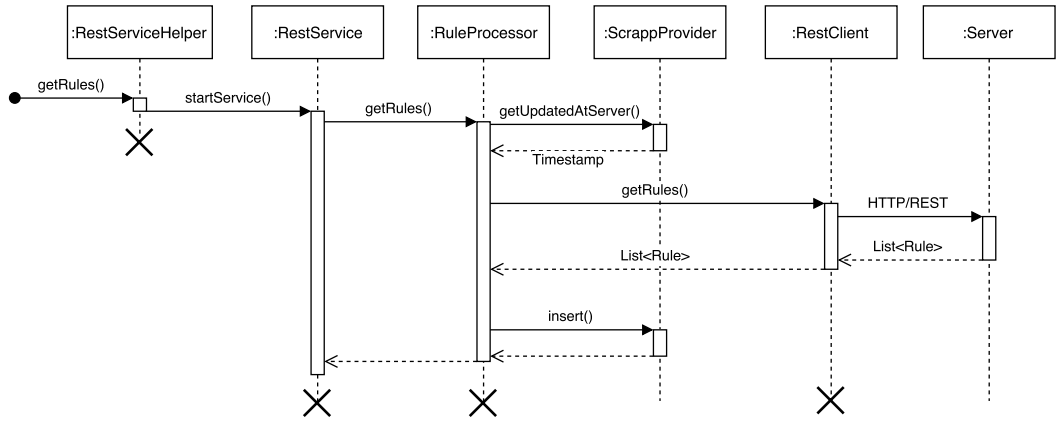

Abbildung 6: Sequenzdiagramm für den Abruf von allen Regeln

die Methode getRules() vom *RestServiceHelper* aufgerufen, welcher einen IntentService startet, sodass der Vorgang in einem Background-Thread durchgeführt wird. Im *RestService* wird der *RuleProcessor* erzeugt und auch hier wird getRules() aufgerufen. Bevor die REST-Anfrage ausgeführt wird, wird aus der Datenbank der neueste *updated_at_server*-Timestamp geladen. Dieser wird als Header-Parameter mitgesendet, sodass der Server nur neuere Regeln selektieren kann. Der RuleProcessor stößt dann die eigentliche REST-Anfrage über den *RestClient* an, welcher eine Liste von Rules zurückgibt. Diese wird nun über den *ContentProvider* in die Datenbank eingefügt. Alle registrierten *ContentObserver* werden über die neuen Regeln notifiziert, sodass zum Beispiel ein Loader in einer Activity die Ansicht aktualisieren kann. Der in Abbildung 6 gezeigte Ablauf kann auch auf die restlichen Requests übertragen werden.

Tabelle 2 zeigt alle REST-Endpunkte. Bei jeder Anfrage wird ein Header-Parameter *Identity-Token* für die Authentifizierung mitgesendet. Bei GET-Anfragen wird zusätzlich auch *Updated-At-Server* gesetzt, welcher auf dem Server zur Selektierung von neueren Inhalten genutzt werden kann.

URL	**Method**	**Response**	**Beschreibung**
/users	POST	201	Erstellt einen Nutzer
/rules	POST	201	Erstellt eine neue Regel
/rules	GET	200	Ruft alle Rule-Objekte ab
/rules/{rule_id}	GET	200	Ruft ein Rule-Objekt mit Actions ab
/rules/{rule_id}/subscription	POST	201	Abonniert eine Regel
/rules/{rule_id}/subscription	PUT	204	Ändert die Zeiteinstellung
/rules/{rule_id}/subscription	DELETE	204	Deabonniert eine Regel
/rules/{rule_id}/result	POST	201	Erstellt ein Ergebnis

Tabelle 2: REST-Endpunkte

7 Fazit & Ausblick

Auch wenn unsere Lösung des in Kapitel 1 vorgestellten Problems, dass Benutzer oft und manuell selbst pollen müssen, nicht die Ideallösung ist, sehen wir Scrapp als eine gute Alternative. Die Automatisierung des Pollens vermindert tatsächlich den Aufwand Änderungen zu erfassen und hilft interessante Websites nicht aus dem Auge zu verlieren.

Das Scrapen an sich funktionierte mit unserer Modellierung sehr schnell und effektiv. Scrapp kann so gut wie alle Websites ohne Javascript nahezu problemlos scrapen. Bei Websites, beispielsweise Single-Page-Anwendungen die Javascript nutzen, müsste der Code erst ausgeführt werden, damit das Ergebnis zum Parsen verwendet werden kann. Ein Headless-Browser wie Selenium [5] führt den Code aus und stellt das Ergebnis zur Verfügung, wodurch dieses Problem gelöst wird. Allerdings gibt es nach wie vor Probleme bei der Feststellung ob ein Request erfolgreich oder nicht erfolgreich war.

Das Pattern von Virgil Dobjanschi erwies sich sowohl aus programmatischer als auch aus Nutzersicht als sinnvoll. Beim Programmieren wurde das Problem der Asynchronität in der Android-Entwicklung stark vereinfacht. Für den Nutzer können jederzeit (Teil-)Änderungen kommuniziert werden.

Scrapp funktioniert generisch für verschiedene Arten von Änderungen in unterschiedlichen Websites. Dieser Vorteil bringt auch ein paar Schwierigkeiten mit sich. Generell ist es problematisch so generisch Daten darzustellen. Es ist beispielsweise immer ungewiss welchen spezifischen Inhalt ein Benutzer analysiert und dargestellt haben möchte, sodass darauf kaum eingegangen werden kann. Gleichzeitig wäre das Erstellen der Regeln für jegliche Arten von Änderungen und Websites in der App besonders wegen der Parse-Ausdrücke und der Navigation kompliziert und mühselig. Daher bieten wir ein Webtool an, in dem eine Regel per JSON-String auf dem Server erstellt werden kann. Um die Problematik der Regelerstellung zu lösen, empfiehlt sich ein grafisches Webtool, bei dem der Benutzer sich durch eine Website bis zur Zielseite klickt. Idealerweise werden dabei alle Nutzereingaben verfolgt und direkt als Parameter zur jeweiligen Action gespeichert.

Das Scheduling kann mit Vorschlägen von regelmäßigen Scrape-Zeiten beziehungsweise Intervallen erweitert werden. Der Server weiß welcher Client wann regelmäßig scraped. Basierend darauf könnte er berechnen, welches neue Intervall am effizientesten mit Hinblick auf die gleichmäßige zeitliche Verteilung der Scrape-Vorgänge aller abonnierten Clients für eine Regel ist. Dieses Intervall könnte einem Nutzer vorgeschlagen werden.

Weiterhin könnte Scrapp als Bibliothek ausgebaut werden, sodass andere Apps die Scrape-Funktionalität als Grundlage nutzen könnten. Dies wäre für auf einen Bereich spezialisierte Apps, zum Beispiel eine QIS-App, von Interesse.

Online-Quellen

[1] Flask (A Python Microframework). http://flask.pocoo.org/. [letzter Zugriff: 18. Juli 2015].

[2] Flask-RESTful. https://flask-restful.readthedocs.org/en/0.3.3/. [letzter Zugriff: 18. Juli 2015].

[3] Google I/O 2010 - Android REST client applications. https://www.youtube.com/watch?v=xHXn3Kg2IQE. [letzter Zugriff: 18. Juli 2015].

[4] jsoup Java HTML Parser, with best of DOM, CSS, and jquery. http://jsoup.org/. [letzter Zugriff: 18. Juli 2015].

[5] Selenium automates browsers. http://www.seleniumhq.org/. [letzter Zugriff: 13. August 2015].

[6] The world's most advanced open source database. http://www.postgresql.org/. [letzter Zugriff: 13. August 2015].