FarmFinder

Mobile-App zur Auffindung von Bauernhöfen mit Direktverkauf

Robert Stoll

University of Applied Sciences Upper Austria
S1310454032@students.fh-hagenberg.at
Januar 2015

1 Motivation und Problemstellung

Persönlich bin ich der Meinung, dass man vermehrt Lebensmittel direkt bei lokalen Bauern beziehen sollte. Um einerseits dem Konsumenten dies zu vereinfachen und gleichzeitig die Bauern beim Direktverkauf zu unterstützen, möchte ich eine entsprechende Mobile-App entwickeln, welche dem Konsumenten auf einer Karte anzeigt wo es Bauernhöfe gibt, die Direktverkauf anbieten. Dabei sollen die Konsumenten auch die Möglichkeit haben, nach einem bestimmten Produkt zu suchen, um so die Auswahl der Bauernhöfen einzuschränken (siehe Abb. 1).

Nebst der erwähnten App für die Konsumenten, ist auch ein Backend für die Bauern von Nöten, damit diese Ihre Stammdaten sowie Sortiment pflegen können. Die Suche soll gut skalieren, damit in Zukunft auch mehrere Tausend Personen gleichzeitig die Mobile-App benutzen können. Denkbar ist auch, dass die Suche in Zukunft auch über eine Webseite angeboten wird.

2 Design Entscheide

Es wurde entschieden für diesen Prototypen eine Android App zu entwickeln und als Backend auf .NET Technologien sowie Azure zu setzen. Als Suchengine wird der Port von Lucence für .NET verwendet und die zusätzliche Library AzureDirectory soll die Integration von Lucence.NET in Azure erleichtern. Folgende weitere wichtige Designentscheide wurden getroffen:

1. Als Schnittstelle der Suche soll eine REST API dienen mit JSON, JSONP und XML Unterstützung - Web API wird hierfür eingesetzt
2. Backend und Suche sollen getrennt sein, um eine bessere Skalierbarkeit zu erreichen
3. Änderungen im Backend müssen nicht in Echtzeit zur Verfügung stehen.

Implementierungsstechnische Entscheide werden hier nicht aufgeführt und werden im nächsten Abschnitt erwähnt. Vorweg sei genommen, dass ich in Zukunft möglicherweise auf eine alternative Library setzen werde (oder Eigenimplementation) anstelle von AzureDirectory (mehr Information dazu in der Sektion 5 Probleme).

3 Architektur

Grundsätzlich wurde die Architektur auf Skalierbarkeit und Effizienz ausgelegt. Da es sich hierbei um einen Prototypen handelt und die Cloud-Aspekte im Vordergrund standen, wurden einige Aspekte die nicht Cloud spezifisch sind vorerst etwas in den Schatten gestellt. Auf ein paar Punkte wird in 6 Ausblick eingegangen. Eine gute Skalierbarkeit wird unter anderem erreicht, da SearchAPI und Backend getrennt sind. So lässt ich einfach die Anzahl Instanzen für die SearchAPI hinaufschrauben, sollte ein grösserer Suchandrang bestehen. Zum anderen wurde durchgehend

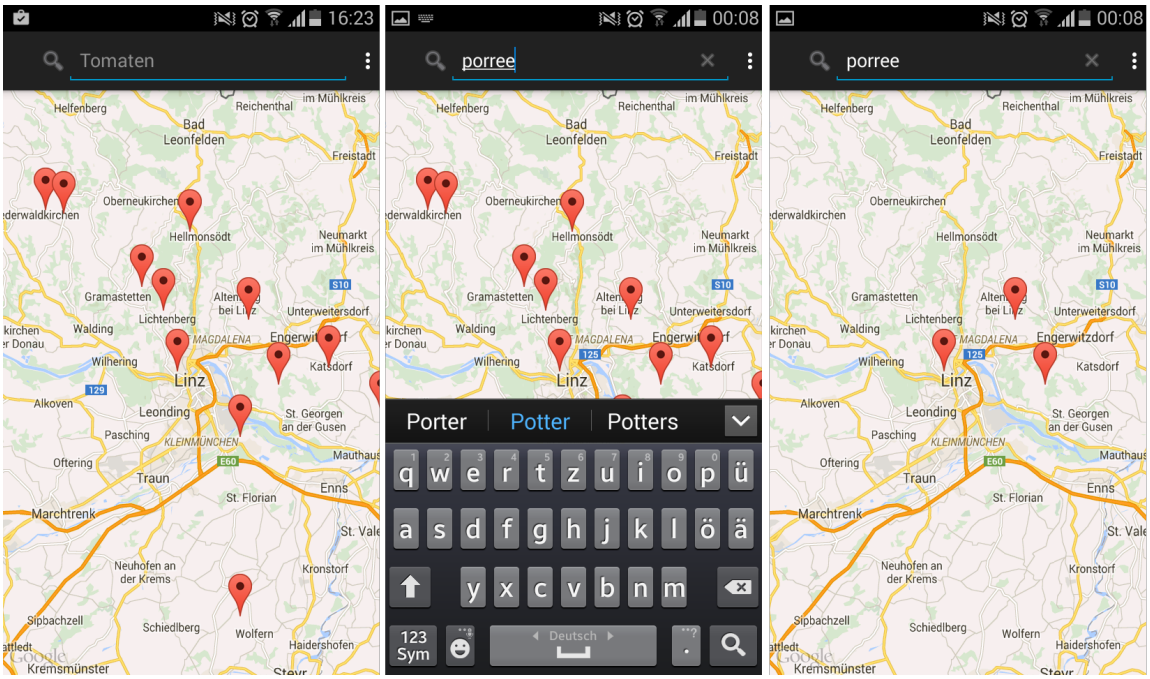

Abb. 1: *Screenshots des Prototypen der Mobilen-App*

auf ein asynchrones Pattern gesetzt. In Abb. 2 ist eine Grobübersicht der verschiedenen Subsysteme ersichtlich und wie sie miteinander interagieren.

Als Schnittstelle nach aussen dienen die beiden Web-Rollen. Einerseits das Backend als Webseite und andererseits die SearchAPI als REST API. In Abb. 3 ist der Ablauf beim Hinzufügen eines neuen Produkts durch einen Bauern ersichtlich. Dabei gilt zu beachten, dass Consumer A das neu hinzugefügte Produkt im Suchresultat noch nicht sehen kann und es erst im Suchergebniss des Consumer B enthalten sein könnte. Im folgenden wird kurz auf jedes einzelne Subsystem eingegangen – dadurch sollte auch der Ablauf in Abb. 3 klarer werden.

3.1 Backend

Das Backend dient den Bauern zur Pflege der Stammdaten sowie zur Verwaltung ihres Sortiments. Änderungen werden dem IndexUpdatingWorker per Queue mitgeteilt. Konkret wird ein Update-IndexDto auf die IndexUpdatingQueue gelegt, welches die Information enthält, welcher Bauernhof Änderungen erfahren hat (Id der Farm) und wie (Create, Update oder Delete) – dabei werden Änderungen am Sortiment ebenfalls als Änderung am Bauernhof gehandelt, da im invertierten Index von Lucene nicht jedes einzelne Produkt erfasst wird, sondern der Bauernhof als ein Element bzw. Dokument.

Das Backend enthält momentan eine kurze Erklärung über den Prototypen (Startseite), eine Dokumentationsseite der REST-API, sowie eine Seite, die es erlaubt Suchanfragen an die API zu stellen. All diese Seiten würden in Zukunft vom Backend entfernt.

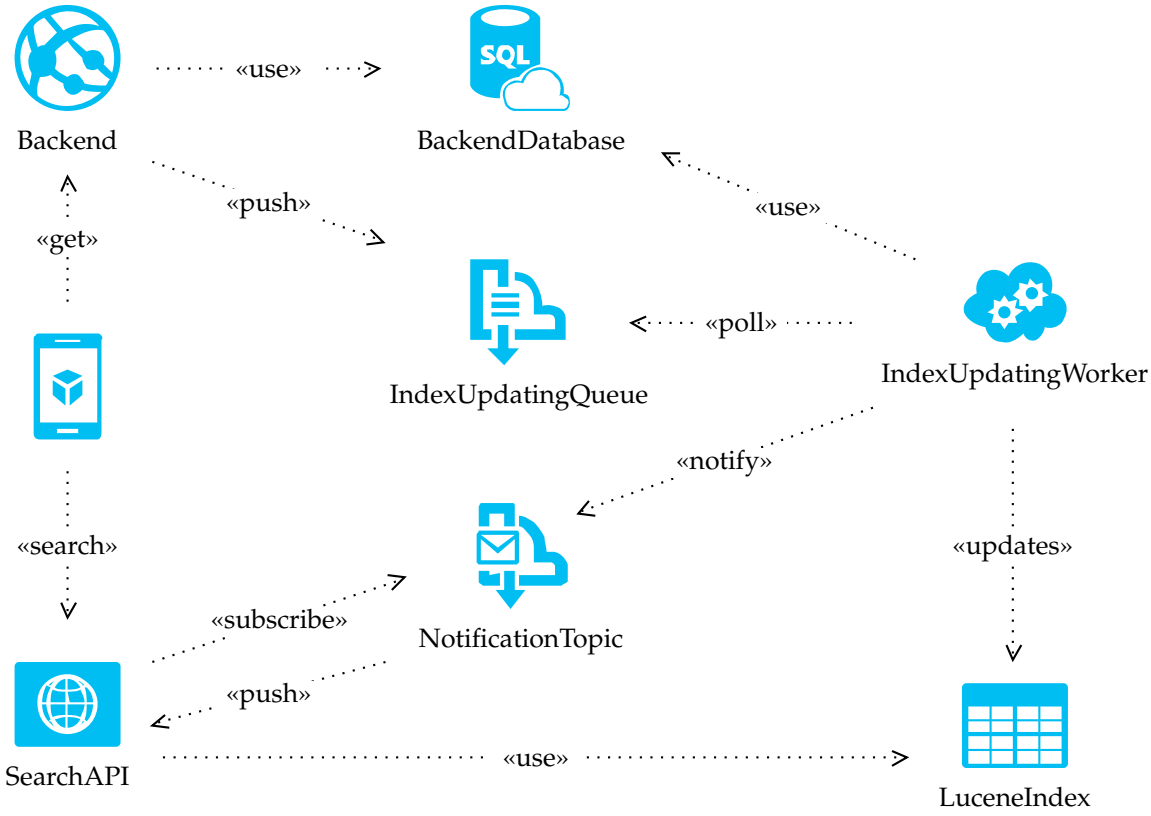

Abb. 2: *Grobübersicht der einzelnen Subsysteme*

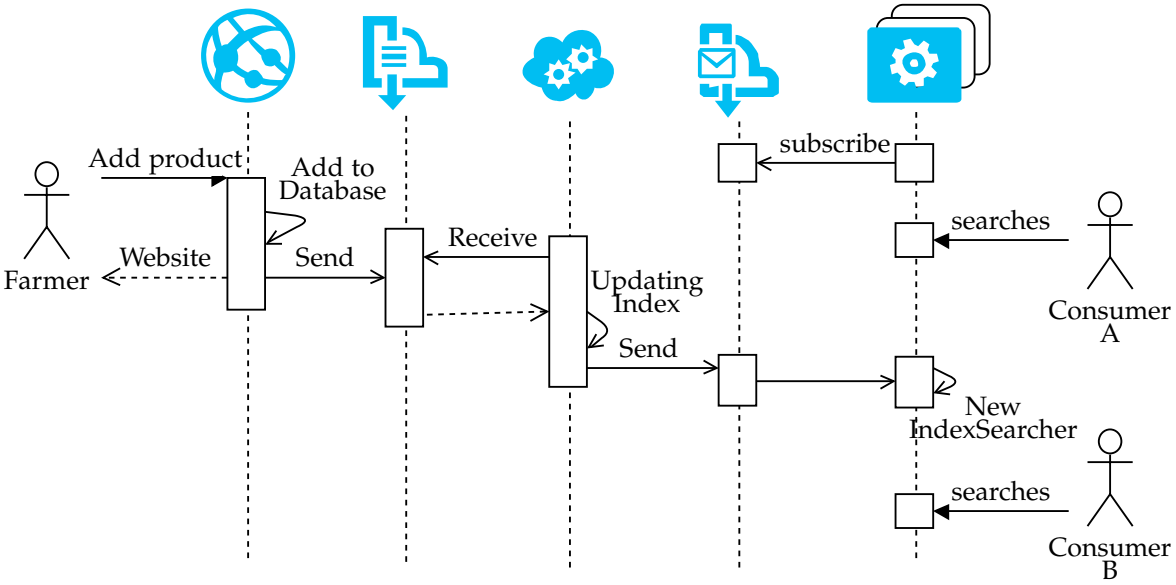

Abb. 3: *Ablauf wenn ein neues Produkt hinzugefügt wird*

3.2 SearchAPI

Die SearchAPI besitzt einzig und alleine die nötigen Controller für die REST-API (momentan ein einziger Controller, namentlich der SearchController). Routing-Definitionen wurden gänzlich gelöscht, um einzelne Request möglichst schlank zu halten (ohne Routing-Lookup wird es schneller). Hier wäre denkbar, dass man anstelle von Web API auf Komponenten von ServiceStack zurückgreift und auf OWIN aufbaut. Dies sollte eine weitere Effizienzsteigerung ermöglichen.

Die SearchAPI betreibt einen Hintergrundprozess, welcher sich bei der NotificationTopic registriert hat und erfährt, wenn der Lucene-Index aktualisiert wurde. Tritt dies ein, so wird der Index im lokalen Cache verworfen und der Lucene-Index wird erneut geladen. Eine weitere Massnahme wurde ergriffen, um eine Flut von Änderungen nicht in x-fachem Laden des Indexes enden zu lassen. So wird angenommen bzw. die verwendete Heuristik baut auf der Annahme auf, dass wenn eine neue Änderung eintrifft und die letzte nicht älter als eine Minute her ist, noch weitere Änderungen folgen werden. Daher wird in einer solchen Situation ein Timer eingestellt, welcher nach einer weiteren Minute den Index aktualisiert. In der Zwischenzeit werden alle Benachrichtigungen betreffend einer Änderung am Index ignoriert (die Logik ist in ThrottlingCall implementiert).

3.3 IndexUpdatingWorker

Diese Worker Rolle dient zur Erstellung und Aktualisierung des Lucene Indexes. Die Wartung des Indexes wurde inkrementell gestaltet. Das heisst, die Erstellung des gesamten Indexes ist nur einmal nötig[1] und danach werden Updates, die quasi über die IndexUpdatingQueue rein flattern, gepflegt – sprich ein inkrementeller Ansatz wurde gewählt. Wie auch bei der SearchAPI wird eine Flut von Änderung mittels Throttlingcall gedrosselt.

Sollte der IndexUpdatingWorker aus irgend einem Grund aussteigen, so führt er alle nötigen Updates beim Starten der Worker Rolle aus (wiederum ein inkrementeller Ansatz). Theoretisch würde es genügen, wenn der IndexUpdatingWorker die verbliebenen Änderungen auf der Queue nach dem Start abarbeiten würde. Wieso aber ein Abgleich mit der Datenbank nötig war wird in 5.1 AzureDirectory erläutert.

Änderungen am Index werden per NotificationTopic an alle SearchAPI-Instanzen mitgeteilt.

3.4 IndexUpdatingQueue

Die IndexUpdatingQueue dient als Middleware für die asynchrone Kommunikation zwischen Backend und IndexUpdatingWorker.

3.5 NotificationTopic

Diese Topic dient als Middleware für die asynchrone Kommunikation zwischen IndexUpdating-Worker und den SearchAPI-Instanzen. Dabei besitzt der IndexUpdatingWorker nur das Recht zu schreiben und die SearchAPI-Instanzen nur das Recht zu lesen.

3.6 BackendDatabase

Die Datenbank dient zur Persistierung der Stammdaten sowie des Sortiments der jeweiligen Bauern und Bäuerinnen.

[1] Tatsächlich musste ich eine Fehlerbehandlung einbauen, die den Index neu erstellt, wenn der Index zuvor in einen inkonsistenten Zustand geraten ist - hier zeigen sich erste Schwächen der AzureDirectory-Library.

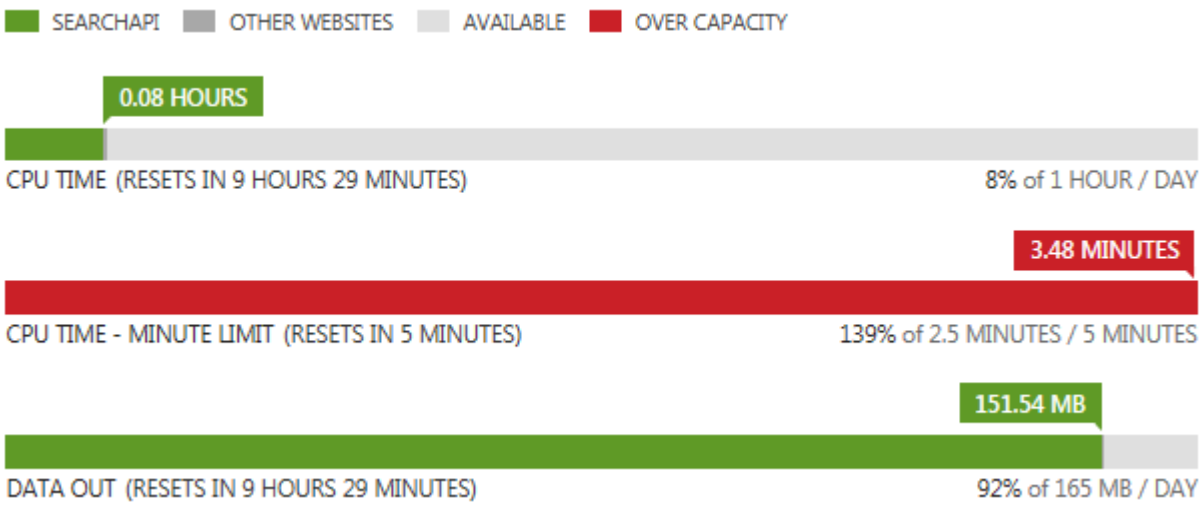

Abb. 4: *CPU-Limiten nach ca. 2 Minuten Laufzeit des BigLoad-Tests*

3.7 LuceneIndex

Der Lucene-Index wird in einem Blob-Storage persistiert und von den jeweiligen SearchAPI-Instanzen dann jeweils gelesen und lokal gecached.

4 Lasttests

Es wurden 3 Lasttests aufgesetzt (befinden sich ebenfalls in der Visual Studio Solution) und durchgeführt. Der erste Test (BigLoad) wurde viel zu optimistisch angesetzt und hatte zum Ziel eine Last von 10'000 Benutzern zu simulieren und musste ziemlich schnell abgebrochen werden (nach ca. 2 Minuten) bzw. wurde ziemlich schnell unterbrochen aufgrund der Limits in Azure (siehe Abb. 4). Da gleichzeitig auch das DATA OUT Limit überschritten wurde, mussten anschliessende Tests vertagt werden.

Zeit	Benutzer-Load	Seiten/Sekunden	Total Anzahl Seiten	Antwortzeit in Sekunden
00:15	450	62.67	940	0.17
00:30	800	129.93	2889	0.20
00:45	1200	202.87	5932	0.29
01:00	1550	192.87	8825	0.72
01:15	1950	238.2	12398	2.53
01:30	2300	95.93	13837	2.90
01:45	2700	197.87	16805	6.53

Tabelle 1: *Load-Test Kennzahlen*

Was sich aus dem BigLoad-Test dennoch herauslesen liess, ist, dass selbst bei 1000 eingetragenen Bauern und 1200 Benutzern die gleichzeitig intensiv die Mobile-App verwenden – aus meiner Sicht eine realistische Zahl eines kurzzeitigen Picks von weniger als 1 Minute – die Antwortzeit der SearchAPI im Schnitt bei 0.29s lag, eine gute Zeit meiner Meinung nach. Tabelle 1 zeigt im

Detail was innerhalb der ersten paar Minuten des BigLoad-Tests für Kennzahlen hervorgingen.

Beim zweiten Lasttest (MiddleLoad) war das Ziel einer Last von 1'000 Benutzern stand zu halten, was für gute 8 Minuten auch ohne Probleme ging, danach war wiederum das Limit der CPU-Zeit überschritten. Insgesamt konnten in diesem Fall etwas mehr als 10'000 Suchanfragen bewältigt werden. Interessanter ist der letzte Lasttest (SmallLoad), welcher über 8 Minuten eine ansteigende Last von 10 Benutzern auf 500 Benutzer simulierte und 500 Benutzer während weiteren 4 Minuten standhalten konnte – danach war das DATA OUT Limit wiederum überschritten. Im Schnitt Betrug hier die Antwortzeit 0.13 Sekunden und insgesamt wurden ca. 25'000. Suchanfragen bewältigt.

Auffallend bei allen Tests war, dass der erste Request bis zu 10 Sekunden dauern kann. Ein äusserst unangenehmer Nebeneffekt, der sich wohl daraus ergibt, dass die Ressourcen von Azure runtergefahren werden, wenn über längere Zeit keine Suchanfrage stattgefunden hat. Dies wäre für ein produktives System höchst problematisch, da ein Benutzer der Mobilen-App wohl sehr schnell vergrault würde. Hier wäre zu überlegen, ob man einen Hintergrundprozess laufen lassen will, welcher jede Stunde eine Abfrage tätigt, um dieses Problem zu umgehen.

Fazit aus den Laststests: für wirklich grosse Kapazitäten ist die gewählte Subscription logischerweise nicht geeignet. Eine grundlegende Fehlüberlegung beim Test-Setup war allerdings, dass es in der Praxis wohl eher ungewöhnlich ist, dass Benutzer hunderte von Suchanfragen nacheinander stellen – Think Time wurde zwar berücksichtigt, dennoch würden Benutzer wohl nicht mehr als 10 Suchanfragen nacheinander stellen. Somit entsprach die eigentliche Last wohl weit mehr als der angepeilten Benutzerlast, was selbstverständlich positiv ist. Wie oben beschrieben, wurden im letzten Lasttest insgesamt etwas mehr als 25'000 Suchanfragen bedient. Bei einem geschätzten Suchverhalten von 5 Suchanfragen pro Benutzer täglich, könnte man mit der gewählten Subscription 5'000 Benutzer bedienen ohne dabei die Limits zu übersteigen, was mehr als genügend wäre.

5 Probleme

Während dem Projekt bin ich auch über einige Stolpersteine gefallen und ein paar möchte ich hier kurz erläutern.

5.1 AzureDirectory

Wie bereits in der Sektion 2 Design Entscheide erwähnt, habe ich negative Erfahrungen mit der Library AzureDirectory gemacht. Einerseits ist die Dokumentation veraltet und man weiss nach einiger Zeit nicht mehr wirklich, welcher Information man aus der Dokumentation vertrauen darf (gerade bezüglich Best-Practices) und welche sich als falsch oder nicht ideal entpuppt haben. So schreib beispielsweise Czernicki in einem Eintrag auf Stackoverflow "Keep in mind, (I mentioned this before in posts) AzureDirectory in my opinion is not 'Enterprise' or 'serious production' ready...".[2]

Zum anderen musste ich extra Fehlerbehandlungen einführen, da das locking bzw. den Lock wieder freigeben nicht einwandfrei funktioniert hat (lock wird teils erst nach 30-60 Sekunden freigegeben). Dies ist insbesondere dann problematisch, wenn der IndexUpdatingWorker abstürzt

[2] Link zum Kommentar: http://stackoverflow.com/a/18500738

und nach einem Wiederanstarten dann gleich nochmals abstürzen würde aufgrund einer unerwarteten LockObtainFailedException. Da es sich gezeigt hat, dass in solchen Fällen meist auch der Index nicht mehr synchron mit der Datenbank war und ausstehende Änderungen in der Queue teils nicht alle Unterschiede abgedeckt haben, wurde entschieden, dass der IndexUpdatingWorker sich nach dem Start mit der Datenbank abgleicht (selbstständig quasi das Delta berechnet) und zwar unabhängig davon, ob in der Queue theoretisch alle nötigen Änderungen stehen (was zu diesem Zeitpunkt noch nicht bekannt ist).

In einer solchen Situation wäre ist es durchaus möglich, dass in der Queue eine Änderung steht, die dann beim Abgleich bereits durchgeführt wurde. Zu Beginn wurde angedacht, dass ein Timestamp im DTO mitgeführt wird, mit welchem sich vergleichen lässt, ob das aktuelle Update dem Erwarteten entspricht. Jedoch wurde schnell festgestellt, dass Zeitabgleich über mehrere Instanzen hinweg nicht sinnvoll ist, da eine Zeitsynchronisation nicht gegeben ist. Daher wurde der Entity Farm ein weiteres Feld hinzugefügt (IndexDateTime), welches einfach erlaubt zu überprüfen, ob die aktuellste Version bereits im Index vorhanden ist – in einem solchen Fall darf das Update ignoriert werden und die SearchAPI-Instanzen müssen nicht den Cache invalidieren.

5.2 Suche über Webseite

Das Backend bietet momentan zum Testen der SearchAPI ein Webinterface an. Da in Zukunft womöglich nicht nur die Mobile-App als User-Interface dienen soll, konnte hier auch bereits ein erster Prototyp implementiert werden. Dabei stiess ich auch auf ein durch Sicherheitsrichtlinien gegebenes Problem. Webbrowser erlauben es nicht, JSON von einer anderen Domain zu parsen[3]. Um das Problem zu umgehen, wurde hier auf JSONP zurückgegriffen.

6 Ausblick

Da sich FarmFinder momentan erst in einem Prototypenstadium befindet, gibt es noch einige Punkte, die verbessert werden könnten:

- Data-Layer ist momentan noch fix im Backend enthalten und IndexUpdatingWorker referenziert die entsprechenden Klassen. Dies könnte noch als eigenständiges Projekt herausgelöst werden. Hierbei muss beachtet werden, dass Code-First Migrationen nicht mehr so einfach deployed werden können und zusätzlicher Aufwand nötig wäre.
- die Klasse ThrottlingCall, welche in der SearchAPI implementiert ist, wird momentan vom IndexUpdatingWoker einfach per Referenz ins Projekt eingebunden. Hier könnte man sich überlegen, ob man ein Utility-Projekt anlegen möchte – bleibt es allerdings bei einer einzigen Klasse, so wäre ein solchen Ansatz ziemlich over-engineered.
- Fehlerbehandlung wenn Index während dem Betrieb nicht Aktualisiert werden kann ist noch ausstehend (z.B. wenn Blob-Storage gerade nicht erreichbar ist).

Aber auch noch einige Anforderungen an FarmFinder wurden im Zuge des Moduls Cloud-Computing noch nicht umgesetzt:

- Lokalisierung (momentan wird Linz fix als Ort angenommen)
- Authentifizierung und Autorisierung für das Backend.

[3] Diese Sicherheitsmassnahme soll es erschweren Cross-Site-Scripting Attacken durchzuführen.

- Expertensuche für Mobile-App
- Lizenzierungsmodell nötig oder Finanzierung über Werbung/Sponsoring?

Weiter gilt es noch abzuschätzen, ob die Programmierung auf kostengünstigere Komponenten setzen sollte. Beispielsweise könnte eine normale Queue anstelle der ServiceBus-Queue verwendet werden, welche wesentlich günstiger wäre, dafür einige Features nicht unterstützt (wie beispielsweise dass man die Queue nicht pollen muss). Man könnte sogar noch weiter gehen und ganz auf die Queue verzichten, indem der IndexUpdatingWorker periodisch (z.B. jede Minute) selbstständig ein Delta rechnet und die nötigen Updates vornimmt. Genauso wäre es möglich auf die NotificationTopic zu verzichten. SearchAPI-Instanzen könnten ebenfalls periodisch nach neuen Änderungen am Index Ausschau halten (IndexDateTime wäre nicht älter als 1 Minute) und selbstständig den Cache invalidieren. Dies scheint mir tatsächlich die elegantere Variante, zumal Änderungen nicht in Echtzeit übernommen werden müssen und so doch einiges an Geld gespart werden könnte. Im Rahmen dieses Moduls habe ich mich aber bewusst für Queue und Topic entschieden, damit ich den Umgang mit diesen ebenso testen konnte.

Zu guter Letzt gilt es noch genaustens zu überprüfen, ob Windows Azure kostenmässig andere Dienste übertrifft oder ob der Anbieter gewechselt werden sollte.