Arbeitsjournal Michael Günter

Datum: 24. März 2015

Tatsächlicher Zeitbedarf	Geplanter Zeitbedarf	Beschreibung der Arbeit	Bemerkungen, Probleme, genutzte Hilfestellungen
3.5h	2h	SQLite-Plugin einbinden und Datenzugriff sicherstellen	Wir hatten vor ngCordova zu verwenden, jedoch konnten wir es damit nicht umsetzen und mussten das Plugin manuell einbinden.
3h	2h	DB-Schema und Stammdaten-Generator realisieren, sowie beispielhafte Stammdaten erfassen	Ich bin auf das Problem gestossen, wie ich mehrere asynchrone Queries aneinanderhängen kann und in der korrekten Reihenfolge ausführen kann.

Reflexion des Arbeitstages

Zu Beginn der Lektion habe ich noch einige kleinere Bugs gefixt. Danach war ich jedoch fast die ganze Zeit mit Elias daran das SQLite-Plugin, das wir für den Datenzugriff verwenden wollen, einzubinden. Im Konzept hatten wir geplant, dass wir das Plugin mit ngCordova einbinden möchten. Jedoch wollte dies nicht recht funktionieren und wir sind dabei auf massive Probleme gestossen. Nachdem wir lange damit rumprobiert haben, entschieden wir uns das Plugin von Hand, ohne ngCordova, einzubinden, was dann nach einigen Versuchen (diese waren etwas kompliziert, weil wir es immer auf einem nativen Device testen mussten) auch klappte. Der Aufwand mit ngCordova wäre schlicht zu gross gewesen. Wir konnten dann sicherstellen, dass bei der Ausführung auf einem mobilen Gerät das SQLite Plugin verwendet wird und ansonsten WebSQL. Wir waren nun in der Lage ein erstes Query auf die Datenbank abzusetzen.

Als dies funktionierte haben wir einen SQLite Service geschrieben, der ein einfaches Query annimmt und die Resultate in JSON formatierte Objekte umformt und asynchron zurückliefert.

Diese Arbeit dauerte, aufgrund der erwähnten Probleme, deutlich länger als erwartet weshalb dann der Morgen auch schon fast zu Ende war und wir nicht mehr, wie geplant, die Dinge mit den Stammdaten machen konnten. Wir haben lediglich begonnen einige Queries für das DB-Schema zu schreiben.

Nun war die Arbeit in der Schule zu Ende und als ich am Abend nachhause kam, arbeitete ich weiter.

Ich habe den DB-Populator bereitgestellt. Dazu musste ich zuerst die Queries für das DB-Schema und für die Beispiel-Stammdaten finalisieren. Danach habe ich den DB-Populator Service geschrieben, welcher grundsätzlich beim ersten Applikationsstart das DB-Schema anhand des Queries und die Stammdaten anhand der Queries generiert und somit die Datenbank betriebsbereit macht.

Leider bin ich dabei auch wieder auf Probleme gestossen. Die SQL Queries die wir alle schön mit Semikolons abgetrennt hatten funktionierten so leider nicht. Das WebSQL API (SQLite-Plugin verwendet auch dasselbe API) akzeptiert leider nicht mehrere Queries, die mit Semikolons getrennt sind. Also musste ich die Queries auftrennen und irgendwie die Methode vom SQLite Service einzeln pro Query aufrufen. Dies erwies sich als sehr schwer weil man die Queries (vorallem die Queries vom DB-Schema) in der richtigen Reihenfolge ausführen muss, damit sie funktionieren. Dass Queries asynchron ausgeführt werden

erschwerte zudem die Sache. Deshalb musste ich eine Chain-Funktion schreiben, die mehrere Queries annimmt und diese in der korrekten Reihenfolge ausführt. Ich sass die meiste Zeit an dieser Funktion und musste lange probieren bis sie richtig funktionierte.

Doch dann funktionierte es endlich und die Datenbank konnte erfolgreich betriebsbereit gemacht werden.

Pendenzen für den nächsten Arbeitstag

Der nächste Schritt ist nun das Laden der Daten, sowie das Implementieren der Queries beim Kombinieren und Aufteilen. Danach folgt das automatische Speichern der Position der Elemente.