Beitrag Q: Christine Müller

Eine Architektur für die Entwicklung von Umweltinformationssystemen auf Basis von mit Android-Apps aufgenommen Daten

An Architecture for the Development of Environmental Information Systems Based on Data Collected by Android Apps

Christine Müller

[1]*Inforst, mueller@inforst.de*

English Abstract

Environmental Information Systems are often based on field data. Rugged Android devices collect field data connected with geocoordinates in a standardized way. Native Apps allow efficient offline work. Photos and methods of picture analysis can be used, too. Software architects are facing multiple challenges developing apps for these purposes: How to build a backend for the offline android apps and produce customized web views for authorized users? What are the advantages of a REST-Architecture in this context? Which mechanisms can be used for synchronizing the database on server and device? How to handle conflicts during inserting data? This article shows possibilities and challenges at the example of Inforst's Apps for forestry and resource management

Zusammenfassung

Umweltinformationssysteme basieren auf im Feld aufgenommenen Daten. Robuste Android-Smartphones sind dafür geeignet, mit entsprechenden Apps Umweltdaten in standardisierter Form mit Geodaten verknüpft aufzunehmen. Dies muss auch offline möglich sein, weshalb native Apps eingesetzt werden sollten. Dabei können auch Fotos und Techniken der Bildanalyse eingesetzt werden. Im Zusammenhang mit der Entwicklung solcher Anwendungen ergeben sich zahlreiche Herausforderungen für die Software-Architektur: Wie können diese Daten in einer zentralen Datenbank gespeichert und über eine Webanwendung den

berechtigten Nutzern richtig angezeigt werden? Welche Vorteile ergeben sich durch eine REST-Architektur? Welche Mechanismen für die Synchronisation der Daten zwischen Server und Android-Gerät sind sinnvoll? Wie sollen Konflikte beim Einspielen in die Datenbank behandelt werden? Dieser Artikel zeigt Möglichkeiten und Herausforderungen am Beispiel der Apps von Inforst für Forstwirtschaft und Umweltmonitoring.

1 Einleitung

Inforst entwickelt offline lauffähige Apps für die Forstwirtschaft, Ressourcen-Management und Naturschutz. Die App *WaldFliege* ist für die Aufnahme von Holzinformationen im Wald geeignet. Damit können Daten über Holzpolter und Einzelstämme mit GPS-Daten und ggf. Fotos verbunden werden und die Informationen über verschiedene Schnittstellen (z.B. Excel oder .xml-Dateien) weitergegeben werden. Die App *WaldKarte* ergänzt diese App mit einer offline lauffähigen Karte auf der die Lagerorte dargestellt werden. Die App *Kronentranzparenz* ist für den Einsatz im Umweltmonitoring entwickelt und kann bei der jährlich stattfindenden Kronenzustandserhebung unterstützend wirken. Dabei werden der Grünanteil, der Kontrast und die Helligkeit von Baumkronen anhand von Fotos ermittelt. Außerdem entwickelt Inforst Apps für den Einstazbereich Forst und Wald im Auftrag. Im Rahmen dieser Arbeit ist auch die Entwicklung eines Backends für die offline lauffähigen Apps notwendig. Besonderer Fokus liegt hier auf der Interaktion zwischen der App und der Webanwendung. Dabei sind die Zeiten, in denen die Apps offline genutzt werden zu berücksichtigen. Eine weitere Herausforderung ist, möglichst viele Schnittstellen für unterschiedliche Benuzter bereitzustellen, ohne den Entwicklungsaufwand zu hoch zu treiben. Hier ist der Einsatz einer REST-Schnittstelle sinnvoll (Abbildung 1).

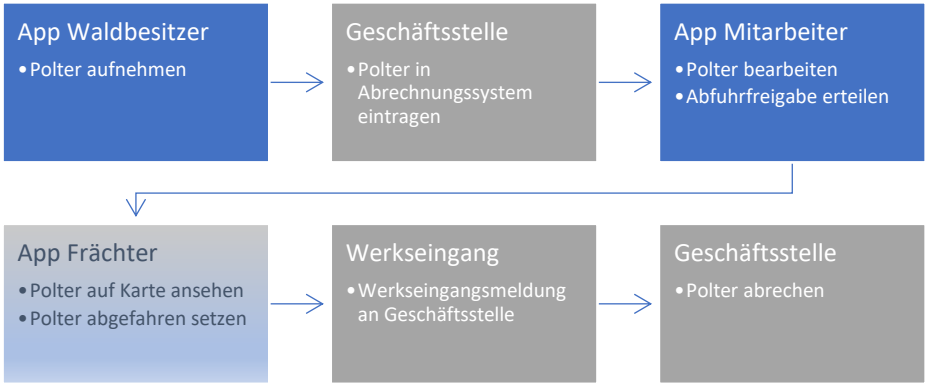

Abbildung 1: Möglicher Geschäftsablauf in einem Forstbetrieb, der Apps von Inforst einsezt. Informationen müssen Anwendungs- und Betriebssystem unabhängig ausgetauscht werden. Interaktionen sollen einfach realisierbar sein.

2 Eine Schichten-Architektur für native Android-Apps

2.1 Persistente Daten innerhalb von Android Apps

An Android-Apps, die im Geschäftskundenbereich eingesetzt werden, werden die gleichen Anforderungen gestellt, wie an Desktop-Anwendungen in diesem Bereich. Daten, die mit Apps aufgenommen werden, müssen dauerhaft gespeichert werden und sollen jederzeit abrufbar sein. Einzelne Datensätze im Cache bereitzustellen, ist für eine arbeitsfähige Anwendung häufig nicht ausreichend. Das bedeutet, dass bei offline arbeitenden Apps eine Datenbank auf dem Gerät vorhanden sein muss. Die SQLite-Datenbank ist eine Datenbank direkt auf dem Android-Gerät. Sie ist offline verfügbar und aufgrund ihrer geringen Größe von nur wenigen hundert kB optimal für den Einsatz auf mobilen Geräten geeignet. Schwierigkeiten ergeben sich bei Änderungen im Datenbank-Schema. Die SQLite-Datenbank erlaubt Änderungen nur über einen eingeschränkten „ALTER TABLE“-Befehl. Das heißt, es können Spalten umbenannt und hinzugefügt werden. Für andere Änderungen mussen Transaktionen vom Entwickler gestaltet werden, zum Beispiel in einem eigenem Upgrade-Manager. Zudem kann die SQLite-Datenbank alle Aktionen nur hintereinander ausführen, was jedoch beim Einsatz innerhalb von Android-Apps selten zu Schwierigkeiten führt. Komplexere Abfragen können in einem eigenen Thread im Hintergrund ausgeführt werden. Eine SQLite-Datenbank wird für eine bestimmte App erstellt und es darf auch nur im Kontext dieser App auf sie zugegriffen werden [Android Community 2018].

2.2 Datenaustausch zwischen Apps

Es gibt jedoch die Möglichkeit, die Daten in der SQLite-Datenbank komplett oder teilweise mit anderen Anwendungen zu teilen. Dies geschieht über Content Provider, die einen gesicherten Zugriff auf die Datenbank ermöglichen. Google stellt solche Content Provider zur Verfügeung, um anderen Apps Zugriff auf z.B. den Kalendar oder die Kontakte des Nutzers zu ermöglichen. Inforst hat mit *WaldFliege* eine App zur Holzaufnahme im Angebot und eine weitere App *WaldKarte*, die eine offline Karte beinhaltet. Sind beide Apps installiert, benutzen sie dieselbe Datenbank. Der Datenaustausch erfolgt hier jedoch nur zwischen diesen beiden Apps. Ist nur *WaldKarte* installiert, speichert *WaldKarte* die Daten in einer eigenen lokalen Datenbank (Abbildung 2).

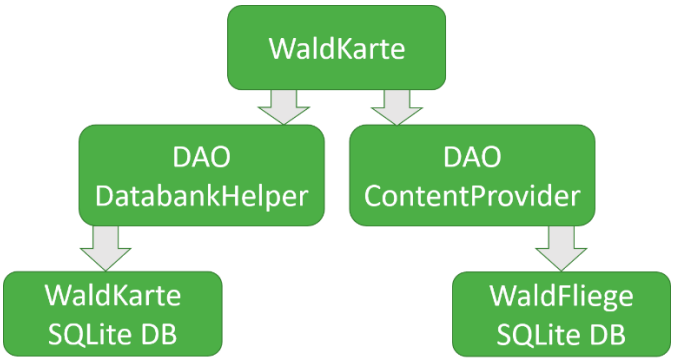

Abbildung 2: Die Apps *WaldFliege* und *WaldKarte*, ein Beispiel für den Einsatz von Content Providern für den Informationsaustausch zwischen zwei verschiedenen Apps

2.3 MVC-Architektur: Voraussetzungen für effiziente Kommunikation mit dem Backend

Bereits bei der Entwicklung der nativen Apps muss die Einbindung in ein mögliches Backend bedacht werden. Dafür müssen die einzelnen Objekte so entworfen werden, dass aus Ihnen einfach Json-Objekte generiert werden können. Diese einfachen Objekte werden von Manager-Klassen (auch Kontrollklassen genannt) verwaltet, die jeweils die Fachlogik für das Objekt implementieren und Berechnungen durchführen. Die Verbindung zur Datenhaltungschicht erfolgt über ein Interface zum SQLite-Databank-Helper bzw. zum Content Provider.

Abbildung 3: Drei-Schicht-Architektur innerhalb der WaldFliege App von Inforst

Im Umgang mit einem persistenter Speicher gibt es vier grundlegenden Operationen, den sogenannten CRUD-Operationen. [HS Augsburg 2018]

- *Create*, Datensatz anlegen,
- *Read* oder *Retrieve*, Datensatz lesen,
- *Update*, Datensatz aktualisieren, und
- *Delete* oder *Destroy*, Datensatz löschen.

Mit Hilfe von Aufrufen der CRUD – Operationen der Datenhaltungschicht sorgt der SQLite-Databenk-Helper oder eben der Content Provider für eine kontrollierte Speicherung in der Datenhaltungsschicht.

Mit dem Room Persistence Library [Google 2018] ist jetzt auch eine den Webservices sehr ähnliche Datenorganisation als fertiges Library auf Android-Geräten nutzbar. Dadurch ist es möglich, die Entwicklungsarbeit von Webanwendung und App anzugleichen.

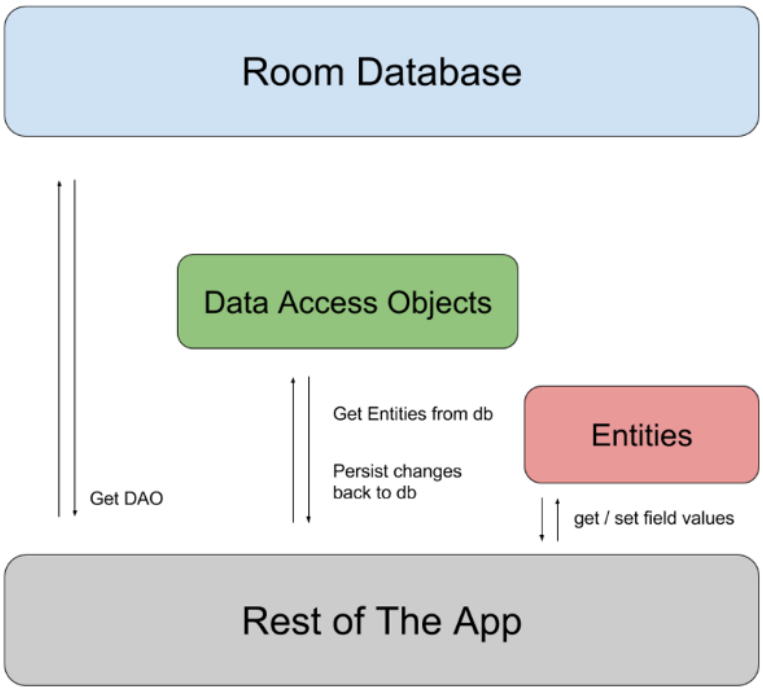

Abbildung 4: Das Room Persistence Library. Quelle: [Google 2018]

2.4 Ein Backend für (offline lauffähige) Android Apps

Bei der Entwicklung eines Backends für offline lauffähige Android-Apps ist ein großes Problem die doppelte Datenhaltung bzw. Implementierung der Geschäftslogik. Da die App komplett offline lauffähig sein muss, muss die komplette Datenbank und auch andere Arbeitsgrundlagen wie z.B. Karten auf dem Gerät gespeichert sein. Auch muss die Geschäftslogik auf dem Gerät implementiert sein. Sie muss aber auch in der Web-Anwendung vollständig vorliegen. Aktualisierungen müssen also im Webservice und auf allen Geräten parallel durchgeführt werden.

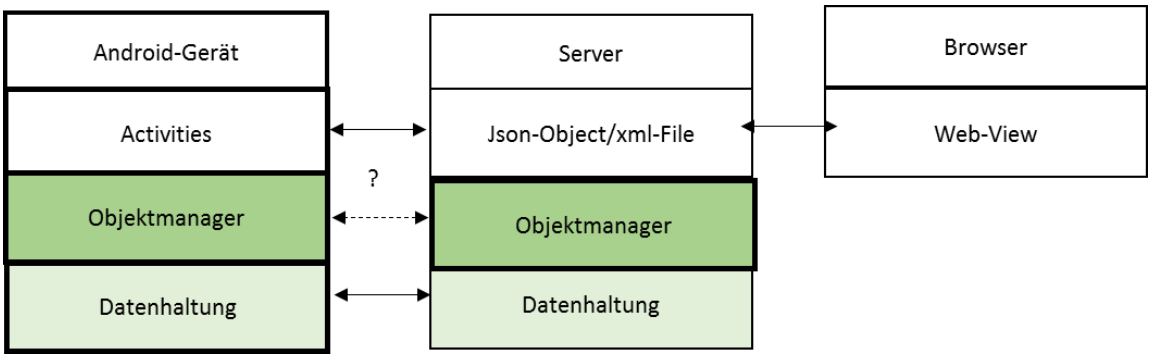

Abbildung 5: Das Problem der doppelten Implementierung der Geschäftslogik

3 REST – Ein Architekturstil

Die meiste Kommunikation über das Internet läuft über das HTTP-Protokoll, ein statusloses Protokoll auf der Anwendungsschicht. REST (Representational state transfer) bezeichnet einen Software-Architekturstil, der die Möglichkeiten Übertragung von Informationen zu eindeutig identifizierbaren Objekten mittels des HTTP-Protokolls optimal nutzt. Dabei folgt REST folgenden Prinzipien:

3.1 Eindeutige Identifikation von Ressourcen

Aus den einzelnenObjekten werden mittels der REST API Ressourcen gebildet und für jede Ressource oder Aggregation von Ressourcen gibt es eine eindeutige URI (universal ressource identifier).

Beispiel für mögliche URIS

- http://www.inforst.de/waldfliege/kundenr0815/hiebBB22/los1/polter17
- http://www.inforst.de/kronentranzparenz/kundenr0816/bilder/20180527
- http://www.inforst.de/waldfliege/kundenr0815/einzelstamm1453

Es ermöglicht auch die Ressourcen in standardisierten Formaten (representations) wie xml, html oder json für verschiedene Anwendungen bereit zu stellen. So lassen sich zum Beispiel einfach Webansichten von Ressourcen erstellen.

3.2 Hypermedia/Verlinkung

Die einzelnen Ressourcen werden bei REST sinnvoll miteinander verlinkt, was die Bereitstellung von Schnittstellen erleichtert. So könnte ein Programm eines Drittanbieters vom Link http://inforst.de/waldfliege/kundenr0815/hiebBB22/lose auf die Losliste geleitet werden, in ihr nach allen Poltern mit Länge 2m suchen und diese z.B.

auf den Zustand „abfuhrbereit“ setzen. Dieses Verfahren wird auch als „HATEOAS“ (Hypermedia as the engine of application state) bezeichnet. Allerdings kann sich, bis sich eine flächendende Internetversorgung im Wald ergeben hat, die offline laufende App nicht ausschließlich auf Hypermedia verlassen. Als Schnittstellenanwendung bietet das Verfahren jedoch zahlreiche Vorteile.

3.3 Das HTTP-Protokoll

Das HTTP-Protokoll ist ein statusloses Protokoll und bietet die universellen Methoden GET, POST, PUT, DELETE, HEAD und OPTIONS Objekte/Ressourcen, allerdings müssen nicht immer alle bereitgestellt werden. Die Kommunikation bei REST erfolgt ausschließlich über diese Funktionen, am häufigsten über GET und POST. Damit bietet REST neben eindeutig definierten Ressourcen (siehe 3.1) auch klar definierte Methoden, diese Ressourcen anzusprechen, aufzurufen, zu modifizieren und zu speichern. Dies erleichtert weiter die Anbindung an Software von Drittanbietern. Der Webservice kann die Ressourcen in unterschiedlicher Form bereitstellen, so dass eine firmeneigene App beispielsweise die Ojekte direkt als json-Objekte empfängt, die sie dann intern als Objekte weiter behandelt. Ein Browser erhält dagegen eine HTML-Datei, in dem nur die Werte, die mit dem Nutzer geteilt werden sollen, in ansprechender Form dargestellt werden. Wie die Zugriffsbeschränkungen realisiert werden können, wird in Abschnitt 5 dargestellt.

REST-Schnittstellen erhalten die statuslose Kommunikation über das HTTP-Protokoll. Dadurch ermöglichen sie die parallele Bearbeitung zahlreicher Anfragen auf Serverseite. In einer forstlichen oder für den Naturschutz eingesetzeten Geschäftsanwendung werden solche Szenarien eher nicht auftreten. Was allerdings tatsächlich zum Problem werden kann sind die geringen Bandbreiten bei der Übertragung der Daten im ländlichen Raum. Daher ist eine möglichst voneinander unabhängige Übertragung möglichst kleinteiliger Datenpakete ein Vorteil für diese Anwendungszwecke.

3.4 Vorteile von REST und mögliche Schwierigkeiten

Hier noch einmal zusammenfassend die Vorteile der Verwendung einer REST-Schnittstelle für den beschriebenen Einsatzzweck:

- Die REST-Architektur bietet eine lose Kopplung zwischen den einzelnen Softwaremodulen und erleichtert die Einbindung von Software anderer Anbieter
- Alle Komponenten können miteinander kommunizieren. Objekte sind anhand Ihrer URI von überall identifizierbar.
- Abfragen, Views, Clientanwendungen etc. können wiederverwendet werden.
- Die Webanwendung ist aufgrund der möglichen Kleinteiligkeit der ausgetauschten Objekte gut skalierbar.

Es ergeben sich aufgrund des geplanten Anwendungszweckes aber auch folgende Schwierigkeiten:

- Eine rein auf Hypermedia basierende Geschäftslogik ist nicht möglich, weil die Apps im großen Umfang, für lange Zeiträume und in bedeutenden Geschäftsabläufen offline funktionieren müssen.
- Synchronisierungsmöglichkeiten für die Datenbank und die Geschäftslogik müssen geschaffen werden.
- Die Internetverbindung im ländlichen Raum ist häufig von geringer Bandbreite, was zu Schwierigkeiten führen kann, wenn der Server vor Ort stationiert sein soll.
- Die verwendete Software von Drittanbietern hat derzeit keine Möglichkeiten, REST-Schnittstellen einzubinden. Eventuelle Schnittstellen müssen auch von den Drittanbietern erst noch geschaffen werden.

4 Umsetzung einer REST-Schnittstelle mit Dropwizard

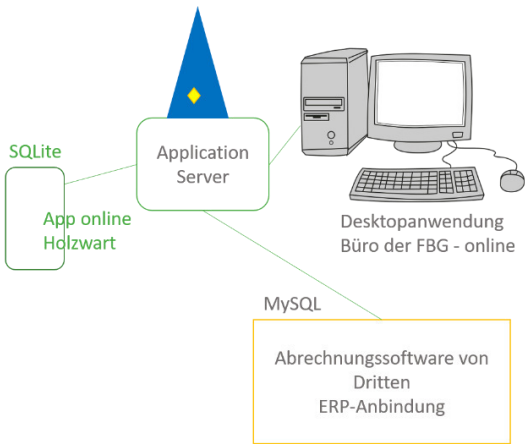

Abbildung 6: Ein Webserver ermöglicht den Datenaustausch zwischen App, Browser und Anwendungen von Drittanwendern

4.1 Das Dropwizard-Framework

Für die Umsetzung von REST-Schnittstellen sind zahlreiche Frameworks (Spring, Grails, Dropwizard) verfügbar. Das bekannteste Framework ist das Spring-Framework. Für die Umsetzung wurde hier jedoch Dropwizard als Framework gewählt, da es unabhängiger im Bezug auf dependency injection ist. Ein Vorteil ist auch das einfache Deployment der Anwendung als fat-jar. Ein Apache-Tomcat-Server ist nicht notwendig und damit auch keine Einbindung in eventuell beim Kunden schon vorhandene Systeme. Die Dropwizard-Anwendung spannt ihren eigenen Jetty-Server auf, der uanbhängig funktioniert und entsprechend leicht gewartet werden kann. Insgesamt bietet Dropwizard eine Sammlung geeigneter Werkzeuge für die Erstellung eines Webservices mit REST-Architektur. Außerdem bietet sie aufgrund der Einbindung von OAuth und der Fähigkeit zur Umsetzung mit HTTPS Werkzeuge für die Entwicklung sicherer Anwendungen, siehe Abschnitt 5.

Als Datenbank auf dem Server wurde eine Postgres-SQL-Datenbank gewählt. Diese kann über den innerhalb von Dropwizard integierten JDBC-Treiber angesprochen werden. Die Interaktion mit der Datenbank erfolgt wieder über DAO-Objekte, die die Ressourcen entsprechend der Geschäftslogik in die Datenbank einfügen oder für den Client bereitstellen. Der Server stellt mehrere GET und POST Befehle für die Clients bereit. Die Ressourcen können im json oder im .xml-Format abgerufen werden. Der in der Forstwirtschaft weit verbreitete Eldat-Standard (KWF 2018) wird gerade von .xml auf json umgestellt, was eine doppelte Bereitstellung erfordert. Innerhalb der Apps ist die Erstellung eines Services für die Kommunikation mit dem Server erforderlich. Dies ist über den Android-HTTPS-Cient möglich.

4.2 Doppelte Datenhaltung und Datenbanksynchronisation

Alle Geräte die offline arbeitsfähig sein müssen, müssen über eigene Datenbanken verfügen. Damit die Objekte datenbankübergreifend identifiziert werden können, müssen sie bei der Erzeugung mit GUIDs (Global unique identifiers) versehen werden. Dies sind eindeutige 16 Byte lange Zahlenkombinationen, die sich nicht wiederholen. Bestehende Objekte ohne GUID müssen ein solches in einem Update erhalten. Auf diese Art können die Objekte über unterschiedliche Geräte hinweg idetentifiziert werden.

Es stellt sich die Frage, wann die Datenbank synchronisiert werden sollte. Aus Nutzersicht ist eine möglichst automatisierte Sychronistation wünschenswert. Die App erledigt diese Arbeit im Hintergrund, ohne das sich der Anwender darum kümmern muss. Dabei muss für den Nutzer der App ersichtlich sein, wann die App vollständig mit der zentralen Datenbank synchronisiert ist. Dies kann zum Beispiel durch eine andere Farbe innerhalb der Liste angezeigt werden.

Besonders kritisch sind das Einfügen und das Löschen von Objekten in der zentralen Datenbank. Wenn bei dem Einfügen ein Konflikt auftritt (was durch klar definierte Geschäftsabläufe selten vorkommen sollte), muss der Nutzer dies erkennen und steuernd eingreifen können. Notwendig ist der Entwurf klarer Zustandsdiagramme für die einzelnen Objekte. Festgelegt werden muss, von welchem Zustand das Objekt auf Veranlassung welches Nutzers in welchen anderen Zustand wechseln kann. So kann ein Holzpolter folgende Zustände durchlaufen: (1) bereitgestellt – (2) freigegeben – (3) abfuhrbereit – (4) abegfahren – (5) Werkseingang vorhanden – (6) Waldbeitzer abgerechnet. Zustand (1) kann vom Waldbesitzer oder dessen forstlichem Dienstleister gemeldet werden. Zustand (2) vom Außendienstmitarbeiter des forstlichen Zusammenschlusses, Zustände (3), (5) und (6) von der Geschäftsstelle und Zustand (4) vom Frachtunternehmen. Bestimmte Methoden, z.B. „Menge ändern“ dürfen ab einem bestimmten Zustand (hier 2) nicht mehr angewandt werden. Durch die Zustände kann der Nutzer kontrollieren, wann welche Daten an wen übermittelt werden.

5 Sicherheitsaspekte

Sobald die Daten die lokalen Geräte verlassen und über das Internet bereitgestellt werden, erhöhen sich die Anforderungen an die Sicherheit der Anwendung um ein Vielfaches. Möglicherweise kann man bei einer Geschäftsanwendung in einem Nischenbereich zuerst kein hohes Risiko für einen Angriff erkennen. Es handelt sich bei den übermittelten Daten jedoch zum Teil über persönliche Daten der Nutzer. Zudem könnten Informationen über die Lagerorte von (Wert-)Holzbeständen durchaus genutzt werden. Davon abgesehen erweitert der Zugang zu jedem Server die Möglichkeiten krimineller Organisationen. Deshalb gilt es, wichtige Aspekte der Internetsicherheit zu berücksichtigen.

5.1 Vertraulichkeit

Es muss sichergestellt werden, dass derjenige, der Daten schickt oder anfordert, wirklich ein berechtiger Nutzer des Webdienstes ist. Zur Absicherung bietet das Dropwizard-Framework das Instrument OAuth. Dies stellt eine Authentifizierung des Nutzers über Nutzername und Passwort (z. B. bei Browseranwendungen) oder auch über ein Token (zum Beispiel bei Verwendung der App) sicher. So kann der Server sicherstellen, dass die Anfragen von einem berechtigen Nutzer stammen. OAuth ermöglicht die Anlage verschiedener „Rollen“ für Nutzer. So kann ein einfaches Mitglied einer forstlichen Vereinigung nur auf seine Daten in der Datenbank zugreifen, ein Mitarbeiter jedoch auf die Daten mehrerer Mitglieder. Erfolgt die Kommunikation über HTTPS kann ein App-Nutzer sicherstellen, das er die Daten an den richtigen Webservice sendet und sie nicht während der Übermittlung verändert werden. Dazu ist die Integration eines SSL-Zertifikats in die App notwendig.

5.2 Verfügbarkeit

Wie schon beschrieben kann die Verfügbarkeit eines Servers, der im ländlichen Raum in Deutschland stationiert ist, nicht immer garantiert werden. Aber auch ein in einem Rechenzentrum stationierter Server kann z.B. durch ständige Anfragen von Schadprogrammen lahmgelegt werden. Dies kann innerhalb des Webservices durch HealthChecks und Response-Regeln sowie die oben beschreibenen Zugangsbeschränkung vermieden werden.

5.3 Verbindlichkeit

Für die Sicherheit und den störungsfreien Ablauf eines Webservices ist es notwendig, nachvollziehen zu können, welches Gerät wann in welcher Form auf den Server zugergriffen hat. Dropwizard bietet hier mit Metrics ein detailliertes Log-Tool, das neben zahlreichen Analysemöglichkeiten auch die Sicherheit des Systems erhöht, indem es Angriffe auf das System nachvollziehbar macht. Schwierig stellt sich die Kontrolle der Inhalte bei Anwendungen dar, die für eine breite Öffentlichkeit entwickelt und bereitgestellt werden. Ein Beispiel sind Portale für das Posten von Beobachtungen im Naturschutz oder Anwendungen im Bereich Umweltbildung. Die Mögllichkeit Bilder und freien Text zu senden, birgt immer auch die Gefahr, unerwünschte Bilder und unerwünschte Texte ungewollt zu veröffentlichen. Selbst wenn der Verursacher

ermittelt werden kann, kann dies einen großen Schaden anrichten. Bisher ist dagegen nur manuelle Kontrolle vor der Veröffentlichung möglich.

6 Weiterführende Themen - Wartbarkeit

Wie oben beschrieben hat man bei der Verwendung von offline lauffähigen Apps das Problem der doppelten Implementierung der Geschäftslogik auf dem Server und den Android-Geräten. Wie aber soll nun vorgegangen werden, wenn sich die Geschäftslogik ändert? Die Änderungen müssen auf allen Geräten (möglichst zeitgleich) konform durchgeführt werden. Inwieweit eine Regelmaschine hier mögliche Lösungen bieten, muss gesondert untersucht werden. Ein anderer Ansatz ist hier ein Refactoring der App-Anwendungen, das diese stärker dem Prinzip der dependecy injection unterwirft. Geschäftsregeln können als Bestandteil der Konfigurationen für die einzelnen Objekte, leichter einheitlich auf Server und Client verwendet werden und leichter ausgetauscht werden. Möglicherweise ist dies in Kombination mit der Einbindung des oben beschriebenen Room-Frameworks möglich. Ein gleichzeitiges Update der Apps auf allen Mobilgeräten ist durch eine Mobile Enterprise Lösung möglich, mit deren Hilfe die Mobilgeräte eines Unternehmens zentral verwaltet werden können.

7 Literaturverzeichnis

Android Community (2018): Application Sandbox. https://source.android.com/security/app-sandbox (aufgerufen am 21.08.2018).

Dewanto, Dr. Lofi (2018): Autorisierungsdienste mit OAuth https://www.heise.de/developer/artikel/Szenarien-und-Spezifikationen-845431.html (aufgerufen am 21.08.2018).

Google Developers (2018): Room Persistence Library. https://developer.android.com/topic/libraries/architecture/room und https://developer.android.com/training/data-storage/room/ (aufgerufen am 21.08.2018).

HS Augsburg (2018): CRUD. https://glossar.hs-augsburg.de/CRUD (aufgerufen am 21.08.2018).

KWF (2018) http://www.kwf-online.org/logistik/datenschnittstellen/eldat.html und https://www.eldatstandard.de/ (aufgerufen am 21.08.2018).

Starke, Gernot (2011): *Effektive Software Architekturen*. München: Carl Hanser Verlag. ISBN: 978-3-446-45207-7

Takai, D. (2017). Architektur für Websysteme: Serviceorientierte Architektur, Microservices, domänengetriebener Entwurf (Hanser eLibrary). https://www.hanser-elibrary.com/isbn/9783446450561 (aufgerufen am 30.08.2018)

Tilkov, Stefan, et al. (2015): *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. eBook Central, http://ebookcentral.proquest.com/lib/htwk/detail.action?docID=2046366.

Created from htwk on 2018-03-21 04:44:15.

Van Steen, M. und Tanenbaum, S. (2017): *Distributed Systems*. Maarten van Steen.