Raspberry PI Security Application

Thonas Herzog, Philipp Wurm

15. Juni 2017

1 Einleitung

Dieses Dokument behandelt die Dokumentation des Projekts für die Lehrveranstaltung *Service Engineering*, das eine Erweiterung des Projekts *RPISec* für die Lehrveranstaltung *Mobile und ubiquitäre Systeme* ist. Das Projekt *RPISec* soll um einen OAuth2-Authentifizierungsservice erweitert werden und es sollen Integrationstests implementiert werden, welche die *Microservice*-Infrastruktur mit *Docker* testen.

2 Entwicklungsrechner einrichten

Dieser Abschnitt behandelt das Einrichten des Projekts *RPISec* auf einem Entwicklungsrechner. Es werden Zugangsdaten für *GMail*, *Firebase* und *Firebase Cloud Messasging* benötigt, die nicht in der Projektstruktur enthalten sind und extern verwaltet werden und daher eingebunden werden müssen.

Konfigurationsdatei	**Beschreibung**
app.properties	Die externe Konfigurationsdatei für den Entwicklungsbetrieb für den *App-Service*
auth.properties	Die externe Konfigurationsdatei für den Entwicklungsbetrieb für den *Auth-Service*
app-test.properties	Die externe Konfigurationsdatei für die Integrationstests für den *App-Service*
auth-test.properties	Die externe Konfigurationsdatei für die Integrationstests für den *Auth-Service*
firebase-account.json	Die externe Konfigurationsdatei für die *Firebase* Authentifizierung

Diese Dateien sind im Verzeichnis $< doc\_location > /config$ enthalten, wobei die Datei *firebase-account.json* und die *GMail* Zugangsdaten ab 15.07.2016 18:00 nicht mehr gültig sein werden, da ab diesem Datum die Zugänge geschlossen werden.

2.0.1 Auth-Service

Mit folgenden *Gradle* Befehl kann der *Auth-Service* über die Kommandozeile gestartet werden. Um den Service in einer IDE zu starten kann eine *Run Configuration* spezifisch für die IDE eingerichtet werden, die alle notwendigen *Gradle* Kommandos und VM-Options definiert.

```
gradle buildFatJar bootRun
       -Dplatform=dev
       -Dadmin.email=<admin_email_address>
       -Dspring.config.location=<fully_qualified_path_to_config_file>
```

Parameter	Werte	Beschreibung
platform	*dev*	**dev**: Profil für die Entwicklung
admin.email	Bsp.: admin@mail.com	Die Email-Adresse des Admins, der beim Start des Service erstellt wird.
spring.config.location	Bsp.: /config.properties	Der voll qualifiziert Pfad zur externen Konfigurationsdatei

2.0.2 App-Service

Mit folgenden *Gradle* Befehl kann der *App-Service* über die Kommandozeile gestartet werden. Um den Service in einer IDE zu starten kann eine *Run Configuration* spezifisch für die IDE eingerichtet werden, die alle notwendigen *Gradle* Kommandos und VM-Options definiert.

```
gradle buildFatJar bootRun
           -Dplatform=dev
           -Dspring.config.location=<fully_qualified_path_to_config_file>
```

Parameter	Werte	Beschreibung
platform	*dev*	**dev**: Profil für die Entwicklung
spring.config.location	Bsp.: /config.properties	Der voll qualifiziert Pfad zur externen Konfigurationsdatei

2.0.3 Integrationstests

Mit folgenden *Gradle* Befehl können die Integrationstest über die Kommandozeile ausgeführt werden. Es muss sichergestellt werden, dass *Docker* gestartet wurde und dass der Benutzer alle nötigen Rechte für *Docker* hat. Auf einer *Windows* Maschine muss sichergestellt werden, dass das Laufwerk wo die Quelltexte liegen als für *Docker* freigegeben wurde.

```
gradle cleanState clean perpareDockerInfrastructure test
       -Dplatform=integrationTest
       -Dapp.config=<fully_qualified_path_to_app_config_file>
       -Dauth.config=<fully_qualified_path_to_app_config_file>
       -DfirebaseConfig=<fully_qualified_path_to_firebase_json_file>
```

Parameter	Werte	Beschreibung
platform	*integrationTest*	**dev**: Profil für die Integrationstests
app.config	Bsp.: /app.properties	Der voll qualifiziert Pfad zur externen Konfigurationsdatei für den *App-Service*
auth.config	Bsp.: /auth.properties	Der voll qualifiziert Pfad zur externen Konfigurationsdatei für den *Auth-Service*
firebaseConfig	Bsp.: /app.properties	Der voll qualifiziert Pfad zur *firebase account* JSON Datei

In der Datei *Gradle Build*-Datei *java/testsuite/client/build.gradle* werden die beiden Umgebungsvariablen *DOCKER_COMPOSE_LOCATION* und *DOCKER_COMPOSE_LOCATION* auf einem Windowssystem automatisch gesetzt, wenn sie nicht vorhanden sind.

3 OAuth2 Authentifizierungsservice

Dieser Abschnitt behandelt die Dokumentation des OAuth2-Authentifizierungsservice, der für die Authentifizierung der mobilen *Clients* für den bestehenden Applikationsservice sowie die Benutzerverwaltung verantwortlich ist. Der OAuth2-Authentifizierungsservice wurde mit *Spring Boot* implementiert, wobei *Spring Boot* schon alle benötigten Funktionalitäten für OAuth2 bereitstellt und die Applikation nur mehr konfiguriert werden muss.

Spring Boot stellt ein Datenbankschema für OAuth2 zur Verfügung, was angewendet wurde und die OAuth2 *Clients*, *Tokens* usw. werden über JDBC in einer Datenbank verwaltet.

Neben diesen Datenbankschema wurden auch Benutzertabellen angelegt, die über JPA verwaltet werden und keine strikten Beziehungen zu OAuth2 Tabellen haben, jedoch halten die Benutzer die *Id* des generierten Oauth2-Clients, damit sichergestellt werden kann, das bei Anfragen nur *Client Credentials* akzeptiert werden, die auch dem Benutzer zugewiesen sind, was so in OAuth2 nicht vorgesehen ist.

3.1 OAuth2 Authentifizierung

Nachdem am Authentifizierungsservice mobile *Clients* authentifiziert werden, wird für diese *Clients* der OAuth2-*Password Flow* verwendet. Da es vermieden werden soll diesen mobilen *Clients* OAuth2-*Client Credentials* mit auszuliefern, wird bei jedem *Login* eines mobilen *Clients* ein neuer *OAuth2 Client* für diesen mobilen *Client* angelegt und gegebenenfalls ein bereits existierender gelöscht, damit werden bei jedem Login neue *Client Credentials* für die mobilen *Clients* generiert.

OAuth2 würde jede gültigen Zugangsdaten für jeden existierenden OAuth2-*Client* akzeptieren, was so geändert wurde, dass nur Zugangsdaten von Benutzer akzeptiert werden, die dem OAuth2-*Clients* zugewiesen sind. Dadurch sind die OAuth2-*Clients* Benutzer direkt zugeordnet und können nicht von anderen Benutzern verwendet werden, was meiner Meinung nach die Sicherheit erhöht.

3.2 Benutzerverwaltung

Dieser Abschnitt der Dokumentation behandelt die Benutzerverwaltung, die Teil des Authentifizierungsservice ist. Da es sich um eine Sicherheitsanwendung handelt wollen wir keine Benutzerverwaltung von anderen Services nutzen, sondern wollen die Benutzer selbst verwalten, was im Authentifizierungsservice implementiert wurde. Es wurden JPA-*Entitäten User* und *ClientDevice* implementiert, wobei *ClientDevice* die Referenz auf den erstellten OAuth2-*Clients* für den Benutzer hält.

3.2.1 *Client* Login

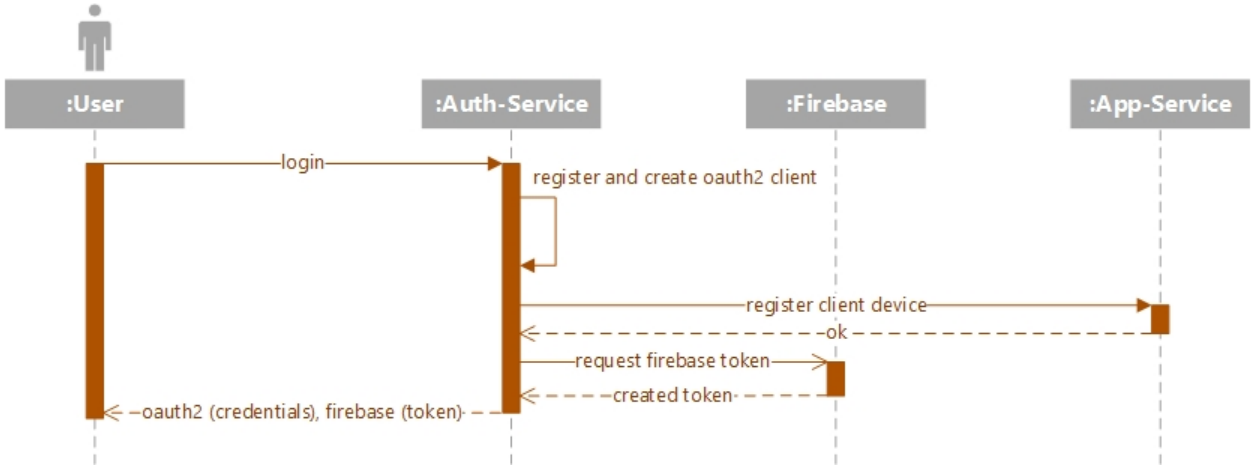

Abbildung 1: Sequenzdiagramm des Logins über einen mobilen *Client*

Die Abbildung 1 zeigt den Ablauf des Logins eines Benutzers über einen mobilen *Client*. Der Benutzer wird mit seinen Zugangsdaten via REST am Authentifizierungsservice authentifiziert und es wird ein OAuth2-*Client* für das verwendete Endgerät erstellt, wobei die *Client*-Anwendung einen eindeutigen Schlüssel für jedes Endgerät erzeugen muss. Dieser registrierte *Client* wird an den Applikationsservice via REST übermittelt. Anschließend wird für den *Client* auf *Firebase* ein Token erstellt, mit dem sicher der *Client* auf *Firebase* authentifizieren kann. Als Antwort wird dem *Client* folgendes JSON-Resultat übermittelt.

Quelltext 1: JSON-Antwort an den *Client*

```
{
  "created": "06.06.2017 07:19:21",
  "token": "eyJhbGciOiJSUzI1NiJ9.eyJhdWQiOiJodHRwczovL2lkZW50....",
  "clientId": "3a97bafe-8d2e-4f2e-bd01-14715bc17435",
  "clientSecret": "11c2216c-9d2d-4926-9892-94f8371ff5f4"
}
```

Das Registrieren des Clients am Applikationsservice erfolgt über Basic Authentifizierung geschützte Schnittstelle, die nur für einen Systembenutzer nutzbar ist, der dem Authentifizierungsservice bekannt ist.

3.2.2 *Client Firebase Cloud Messaging Token (FCM)* Registrierung

Die Abbildung 2 zeigt den Ablauf der Registrierung des FCM-Tokens am Applikationsservice über den Authentifizierungsservice. Die Registrierung über den Authentifizierungsservice wurde gewählt, da dieser Service die Benutzer und deren Endgeräte verwaltet. Das Registrieren des FCM-Tokens am Applikationsservice erfolgt über Basic Authentifizierung geschützte Schnittstelle, die nur für einen Systembenutzer nutzbar ist, der dem Authentifizierungsservice bekannt ist.

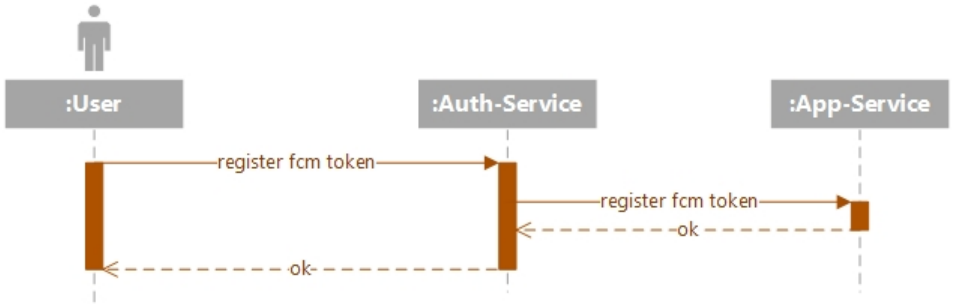

Abbildung 2: Sequenzdiagramm der Registrierung des FCM-Token

Als Resultat wird bei dieser Schnittstelle nur der Http Statuscode 200 zurückgeliefert.

3.3 *Swagger Client* Generierung

Dieser Abschnitt behandelt die Generierung der *Client-API* mit *Swagger*. Das *OpenSource*-Projekt *SpringFox* stellt eine Integration von *Swagger* für *Spring MVC* zur Verfügung mit der aus *RestController Swagger*-Definitionen erstellt werden können. Des Weiteren wird die *Swagger-UI* mitgeliefert, mit der die implementierten REST-Schnittstellen getestet werden können.

Aus den generierten *Swagger*-Definition der *Microservice* spezifischen REST-Schnittstellen wurden *gradle* Projekte generiert, welche die *Client* Implementierungen enthalten. Die generierten Projekte wurden in das Wurzelprojekt *java* aufgenommen, in dem alle Projekte des Projekts *RPISec* enthalten sind.

Für die Generierung wurden die beiden Skripte *generate-clients.bat* und *update-clients.bat* implementiert, wobei das Skript *generate-clients.bat* die Projekte und *Client* Implementierungen generiert und das Skript *update-clients.bat* nur die *Client* Implementierungen generiert. Damit die *Client* Implementierungen generiert werden können, müssen die Services gestartet sein.

Die *Swagger-UI* kann unter folgenden Link erreicht werden $< BASE\_URL >/swagger\text{-}ui.html$, wobei die $BASE\_URL$ der Pfad ist, unter dem der *Microservice* erreicht werden kann. Die $BASE\_URL$ hat das Format $< PROTOCOL >://< HOST >:< PORT >/< CONTEXT\_ROOT >/$.

4 Integrationstests

Dieser Abschnitt behandelt die Integrationstests für die implementierten *Microservices* in einer Docker Infrastruktur.

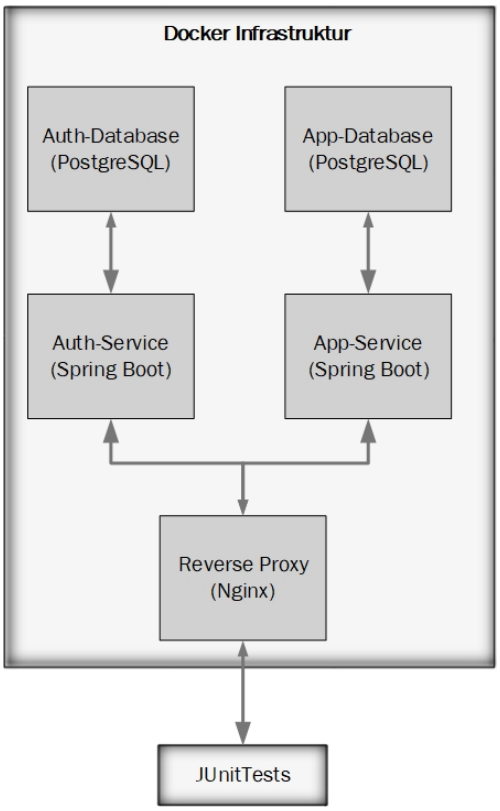

Abbildung 3: Aufbau der Integrationstests mit Docker Infrastruktur

Die Abbildung 3 zeigt den Aufbau der Docker Infrastruktur und die Verbindung zu den implementierten JUnit-Tests. Die Tests wurden in einer Suite zusammengefasst, wobei diese Suite eine *Junit ClasRule* definiert, welche die Docker Infrastruktur via *Docker Compose* vor der Ausführung der *Suite* erstellt und startet und nach er Ausführung der *Suite* die Infrastruktur stoppt und die *Docker Container* entfernt. Die gesamte Infrastruktur ist aufgebaut, wie sie auch im produktiven Betrieb genutzt wird, jedoch konnten die *Dockerfiles* nicht wiederverwendet werden, da dort ARM basierende *Base Images* verwendet werden und die Tests auf einer x86 Architektur ausgeführt werden. Sie sind aber gleich nur das die *Dockerfiles* der Tests von *Base Images* ableiten, die auf einer x86 Architektur aufbauen.

Da ein eigenes *Base Image* verwendet wird muss dieses *Image* bevor die Tests ausgeführt werden mit folgenden Befehl erstellt werden. Dazu muss vorherig in das Verzeichnis *testsuite/client/src/main/resources/docker* gewechselt werden.

```
docker build -t rpisec-test-base base/.
```

Als Implementierung der *JUnit ClassRule* wird die Bibliothek *docker-compose-rule-junit4*[1] von *Palantir* verwendet, die es erlaubt über einen *Builder* die *ClassRule* zu konfigurieren und zu erstellen. Jedoch hat sich gezeigt, dass wenn während der Ausführung eine Ausnahme ausgelöst wird, die *Docker Infrastruktur* nicht richtig runter gefahren wird und dadurch *Docker Container* Namenskonflikte auftreten, die ein erneutes Starten der Tests verhindern.

[1] `https://github.com/palantir/docker-compose-rule`

Die Tests nutzen die in Abschnitt 3.3 beschriebenen generierten *Swagger Clients*, um die Kommunikation der *Clients* mit den *Microservices* und der Kommunikation der *Microservices* untereinander zu testen. Es sei aber angemerkt das es sich hier um reine *Blackbox* Tests handelt, die nur aus der Sicht der *Clients* testen.