

# OAuth2 und Docker Integrationstests für RPISec

Thonas Herzog

26. Juni 2017

## 1 Einleitung

Dieses Dokument behandelt die Dokumentation der Erweiterung des Projekts *RPISec* (*Raspberry PI Security*) um einen *Auth-Service* (*OAuth2*) und Integrationstests mit *Docker*. Das Projekt *RPISec* ist ein Projekt für die Lehrveranstaltung *Mobile und ubiquitäre Systeme*.

Die bestehende Implementierung beinhaltet die Benutzerverwaltung und die Authentifizierung der Benutzer, was in einen eigenen *Microservice Auth-Service*, der *OAuth2* unterstützen muss, gekapselt werden soll. Für die *Microservices* sollen Integrationstests basierenden auf *Docker* implementiert werden, wobei die Tests auf einem Windows und Linux basierten Entwicklungsrechner ausführbar sein sollen.

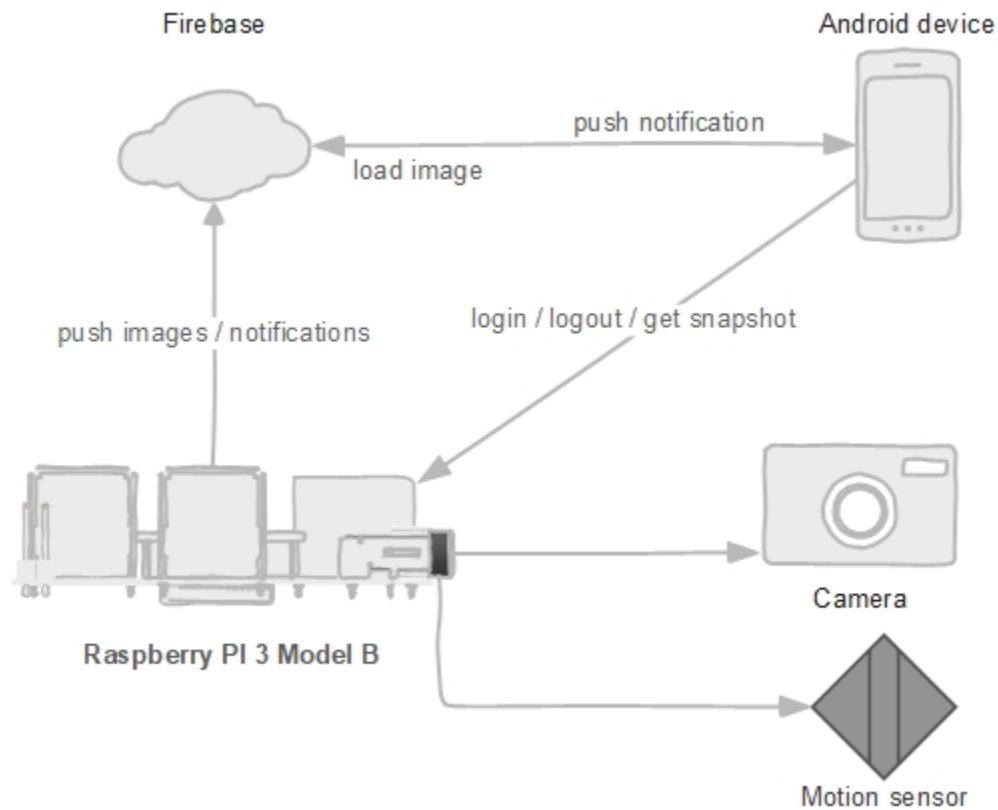


Abbildung 1: RPISec Systemaufbau

## 2 Einrichten

Dieser Abschnitt behandelt das Einrichten des Projekts *RPISec* auf einem Entwicklungsrechner oder *Raspberry PI*. *RPISec* ist auf Zugangsdaten für *GMail*, *Firebase Database* und *Firebase Cloud Messaging (FCM)* angewiesen, die nicht über das Versionierungssystem verwaltet werden und daher nicht in der Projektstruktur enthalten sind. Diese Zugangsdaten müssen lokal bereitgestellt und separat eingebunden werden.

Konfigurationsdatei	Beschreibung
<i>app.properties</i>	Externe Konfigurationsdatei für den <i>App-Service</i>
<i>auth.properties</i>	Externe Konfigurationsdatei für den <i>Auth-Service</i>
<i>app-test.properties</i>	Externe Konfigurationsdatei für die Integrationstests des <i>App-Service</i>
<i>auth-test.properties</i>	Externe Konfigurationsdatei für die Integrationstests des <i>Auth-Service</i>
<i>firebase-account.json</i>	Externe Konfigurationsdatei für die <i>Firebase</i> Authentifizierung

Diese Dateien sind im Verzeichnis */doc/config* der Abgabe 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.

In den folgenden Abschnitten wird beschrieben, wie die Services über die Kommandozeile gebaut und gestartet werden können, wobei die Befehle im Wurzelverzeichnis (*/java*) der Projektstruktur ausgeführt werden müssen.

Bei der Entwicklung wurde auf *Docker* verzichtet, da *Hot-Code-Replacement* und ein schnelleres *Re-Deployment* erwünscht waren.

## 3 Auth-Service

Mit den 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.

```
# on a windows based system the vm options need to wrapped into double quotes "-D..."
.\gradlew :auth:buildFatJar :auth:bootRun
-Dplatform=dev
-Dadmin.email=<admin_email_address>
-Dspring.config.location=<fully_qualified_path_to_auth_properties>
```

Parameter	Werte	Beschreibung
<i>platform</i>	[ <i>dev</i>   <i>prod</i> ]	<b>dev:</b> Profil mit H2 <b>prod:</b> Profil mit PostgreSQL
<i>admin.email</i>	Bsp.: admin@mail.com	Email-Adresse des Admins, der beim Start erstellt wird.
<i>spring.config.location</i>	Bsp.: /auth.properties	Voll qualifizierter Pfad zur Konfigurationsdatei. Der enthaltene Parameter <i>firebase.configFile</i> muss angepasst werden

Während des Start des Service wird ein Administrator Benutzer erstellt (falls noch nicht vorhanden) und es wird eine E-Mail an die angegebene *admin.email* Adresse versendet. Über den in der E-Mail enthaltenen Link kann eine Webseite aufgerufen werden, über die ein Passwort vergeben und dadurch der Zugang aktiviert werden kann.

### 3.0.1 App-Service

Mit den 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.

```
# on a windows based system the vm options need to wrapped into double quotes "-D..."
.\gradlew :app:buildFatJar :app:bootRun
    -Dplatform=dev
    -Dspring.config.location=<fully_qualified_path_to_config_file>
```

Parameter	Werte	Beschreibung
<i>platform</i>	[dev prod]	<b>dev:</b> Profil mit H2 und ohne Sensor <b>prod:</b> Profil mit PostgreSQL und mit Sensor
<i>spring.config.location</i>	Bsp.: /app.properties	Voll qualifizierter Pfad zur Konfigurationsdatei Der enthaltene Parameter <i>firebase.configFile</i> muss angepasst werden

Der *App-Service* benötigt das Verzeichnis *< user.home >/image*, wo die Bilder gespeichert werden.

### 3.0.2 Integrationstests

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

```
# on a windows based system the vm options need to wrapped into double quotes "-D..."
.\gradlew :testsuite/client:clean
    :testsuite/client:prepareDockerInfrastructure
    :testsuite/client:test
    -Dplatform=integrationTest
    -Dapp.config=<fully_qualified_path_to_app_test_properties>
    -Dauth.config=<fully_qualified_path_to_auth_test_properties>
    -Dfirebase.config=<fully_qualified_path_to_firebase_json_file>
```

Parameter	Werte	Beschreibung
<i>platform</i>	<i>integrationTest</i>	Profil für die Integrationstests
<i>app.config</i>	Bsp.: /app.properties	Voll qualifizierter Pfad zur Konfigurationsdatei für den <i>App-Service</i>
<i>auth.config</i>	Bsp.: /auth.properties	Voll qualifizierter Pfad zur Konfigurationsdatei für den <i>Auth-Service</i>
<i>firebase.config</i>	Bsp.: /firebase-account.json	Voll qualifizierter Pfad zur <i>firebase</i> 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 Windows basierten System automatisch gesetzt, wenn sie nicht vorhanden sind. Für Linux basierte Systeme wird in den Standardinstallationsverzeichnissen nach den *Binaries* gesucht, sollten die *Binaries* dort nicht vorhanden sein, so müssen diese Umgebungsvariablen am System gesetzt werden.

## 4 *Auth-Service*

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

*Spring Boot* stellt ein Datenbankschema für *OAuth2* zur Verfügung, welches die *OAuth2 -Clients*, *-Tokens* usw. 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, dass bei Anfragen an den Service nur *Client-Credentials* von *Clients* akzeptiert werden, die auch dem Benutzer zugewiesen sind, was so in *OAuth2* nicht vorgesehen ist.

Um die *OAuth2-Clients* auf Benutzer einzuschränken wurden einige Klassen von *Spring Boot* angepasst bzw. implementiert, damit dieses Verhalten unterstützt wird, was in der Klasse *SecurityConfiguration* im Projekt *auth* eingesehen werden kann.

### 4.1 *OAuth2 Authentifizierung*

Nachdem am *Auth-Service* mobile *Clients* authentifiziert werden, wird für diese *Clients* der *OAuth2-Password-Flow* angewendet. Da es vermieden werden soll, mit der *Client*-Applikation *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.

### 4.2 *Benutzerverwaltung*

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

#### 4.2.1 *Client Login*

Die Abbildung 2 zeigt den Ablauf des Logins eines Benutzers über einen mobilen *Client*. Der Benutzer wird mit seinen Zugangsdaten via REST am *Auth-Service* authentifiziert und es wird ein *OAuth2-Client* für den verwendeten mobilen *Client* erstellt, wobei die *Client*-Anwendung einen eindeutigen Schlüssel für jedes Endgerät erzeugen muss. Dieser registrierte *Client* wird an den *App-Service* via REST übermittelt. Anschließend wird für den mobilen *Client* auf *Firebase* ein Token erstellt, mit dem sich der mobile *Client* auf *Firebase* authentifizieren kann. Als Antwort wird dem mobilen *Client* folgendes JSON-Resultat übermittelt.



```
1 {
2   "created": "06.06.2017 07:19:21",
3   "token": "eyJhbGciOiJSUzI1NiJ9.eyJhdWQiOiJodHRwczovL2lkZW50Lm9yZy5kb291L14715bc17435",
4   "clientId": "3a97baf8-8d2e-4f2e-bd01-14715bc17435",
5   "clientSecret": "11c2216c-9d2d-4926-9892-94f8371ff5f4"
6 }
```

## 5 Client Firebase Cloud Messaging (FCM) Token Registrierung

```
sequenceDiagram
    actor User
    participant User as :User
    participant Auth-Service as :Auth-Service
    participant App-Service as :App-Service
    User->>Auth-Service: register fcm token
    activate Auth-Service
    Auth-Service->>App-Service: register fcm token
    activate App-Service
    App-Service-->>Auth-Service: ok
    deactivate App-Service
    Auth-Service-->>User: ok
    deactivate Auth-Service
```

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

## 5.1 *Swagger Clients*

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

Aus den generierten *Swagger*-Definitionen der REST-Schnittstellen wurden *Gradle* Projekte generiert, welche die implementierten *Clients* enthalten. Die generierten Projekte wurden in das Wurzelprojekt *java* mitaufgenommen, in dem sich alle Projekte des Projekts *RPISec* befinden.

Für die Generierung wurden die beiden Skripte *generate-clients.bat* und *update-clients.bat* implementiert, wobei das Skript *generate-clients.bat* die Projekte und *Clients* generiert und das Skript *update-clients.bat* nur die *Clients* generiert. Damit die *Client* Implementierungen generiert werden können, müssen die Services gestartet sein, da die *Swagger*-JSON-Definitionen von *SpringFox* nur beim Start der Anwendung generiert werden und nicht bei dessen *Build*.

Die *Swagger-UI* kann unter folgenden Link erreicht werden `< BASE_URL >/swagger-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 >`.

Nachdem die *OAuth2*-Authentifizierung über die *Swagger-UI* nicht funktioniert hat, wird für alle REST-Schnittstellen auch eine *Postman-Collection* zur Verfügung gestellt, mit der die REST-Schnittstellen im *Chrome Browser* getestet werden können. Die JSON-Datei, welche die *Postman-Collection* enthält ist im Verzeichnis */doc/config* enthalten.

## 6 *Docker* unterstützte Integrationstests

Dieser Abschnitt behandelt die Integrationstests für die implementierten *Microservices* in einer *Docker* Infrastruktur. Die Abbildung 4 zeigt den Aufbau der *Docker* Infrastruktur und die Verbindung zu den implementierten *JUnit*-Tests. Die Tests wurden in einer *TestSuite* zusammengefasst, wobei diese Suite eine *JUnit ClassRule* definiert, welche die *Docker* Infrastruktur via *Docker-Compose* vor der Ausführung der *TestSuite* erstellt und startet und nach der Ausführung der *TestSuite* die Infrastruktur stoppt und die *Docker Container* entfernt.

Die verwendete *JUnit ClassRule* wird von der Bibliothek *docker-compose-rule-junit4*<sup>1</sup> von *Palantir* zur Verfügung gestellt, die es erlaubt über einen *Builder* die *ClassRule* zu konfigurieren und zu erstellen. Jedoch hat sich gezeigt, dass wenn während des Startens der *Docker* Infrastruktur 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. Mit dem folgenden Befehl kann die *Docker* Infrastruktur bereinigt werden.

```
docker rm -f rpisec-test-auth-db
rpisec-test-app-db
rpisec-test-auth
rpisec-test-app
rpisec-test-nginx
```

Die Tests nutzen die im Abschnitt 5.1 beschriebenen generierten *Swagger Clients*, um die Kommunikation der *Clients* mit den *Auth-Service* zu testen. Es sei aber angemerkt dass es sich hier um reine *Blackbox* Tests handelt, die nur aus der Sicht der *Clients* testen.

---

<sup>1</sup><https://github.com/palantir/docker-compose-rule>

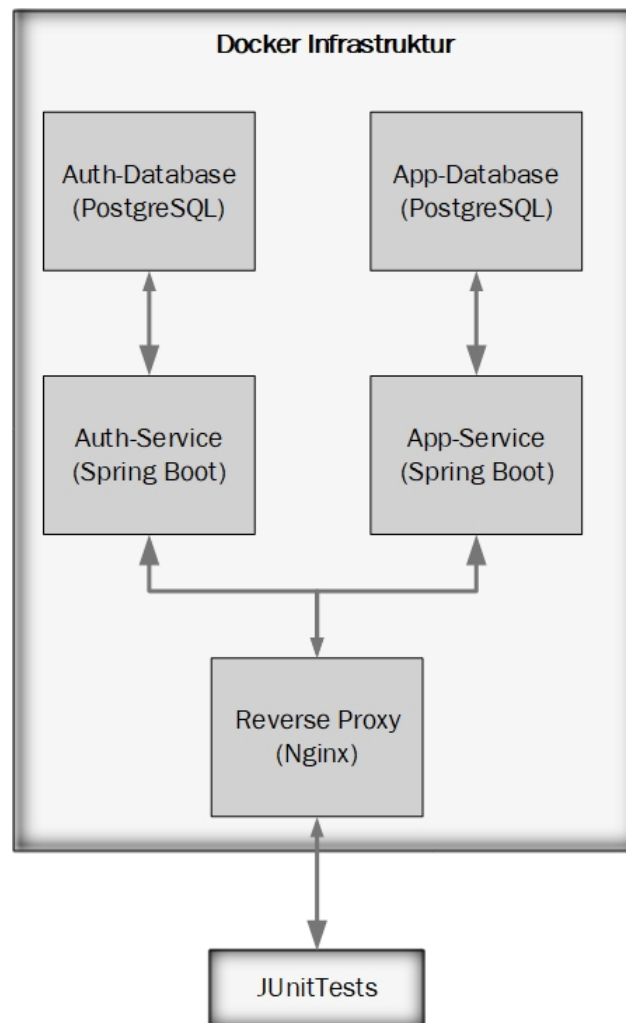


Abbildung 4: Aufbau der Integrationstests mit Docker Infrastruktur