Sonderdruck aus JavaSPEKTRUM

Sicher ist sicher

Zielgruppengerecht: Auftrennen von REST-APIs nach Ursprung

Marcos Scholtz

REST-APIs sind heute gang und gäbe – und ein beliebtes Mittel für ihre Absicherung ist OAuth2. Doch wenn unterschiedliche Zielgruppen auf gemeinsame REST-Ressourcen zugreifen, stößt klassisches Berechtigungsmanagement schnell an seine Grenzen. Eine bewährte Alternative ist die Auftrennung der APIs nach Zielgruppe oder „Request-Ursprung". Dies kann Softwarearchitekten helfen, klare Regelwerke für APIs zu entwerfen und in Projekten mehr Ordnung, Klarheit, neue Absicherungsmöglichkeiten und weniger Sicherheitsrisiken einzubringen.

In Zeiten von Microservices und anderen serviceorientierten Architekturen hat sich der OAuth2-Standard bewährt, um die Absicherung solcher verteilten Systeme zu übernehmen. Er bietet viele Vorteile, wie sehr große Sicherheit, User Federation, Identity Brokering, Social Login, Single-Sign On usw.

OAuth2 und JSON-Web-Tokens

OAuth2 bietet je nach Anwendungsfall mehrere Workflows an, um einen User/Client zu autorisieren [OAuth]. Ein Beispiel dafür ist der „Authorization Code Flow", der am meisten verwendet wird (quasi der OAuth2-Standard-Flow).

Die Details können in der Literatur gefunden werden [ACF]. Am Ende aller OAuth2-Flows wird der Resource-Server mit einem signierten Access-Token aufgerufen (ACF 6 in Abb. 1). Dieser prüft die Berechtigungen darin und gewährt Zugriff auf die Ressource.

Als Standard für den Token hat sich JWT etabliert, oder „JSON Web Token" [JWT]. Hier handelt es sich um einen nach RFC 7519 genormten Access-Token im JSON-Format, der alle Informationen für die Authentifikation beinhaltet. Das erlaubt eine zustandslose Kommunikation ohne Session auf dem Server und erklärt den Siegeszug von OAuth2 mit JWT-Tokens bei verteilten System-Architekturen.

In dem Access-Token stehen verschiedene „Claims". Die wichtigsten für die Absicherung sind der „Subject", der den User identifiziert (wem gehört der Token), und die Rollen, die dieser User hat (also was darf er alles tun).

Absicherung von REST-Ressourcen in Java EE

Hat man den User authentifiziert und hat er bereits einen JWT-Access-Token, dann geht es darum, die REST-Endpunkte auf dem „Resource-Server" abzusichern. Nehmen wir also als Beispiel einen einfachen REST-Endpunkt, welcher Kundendaten zur Verfügung

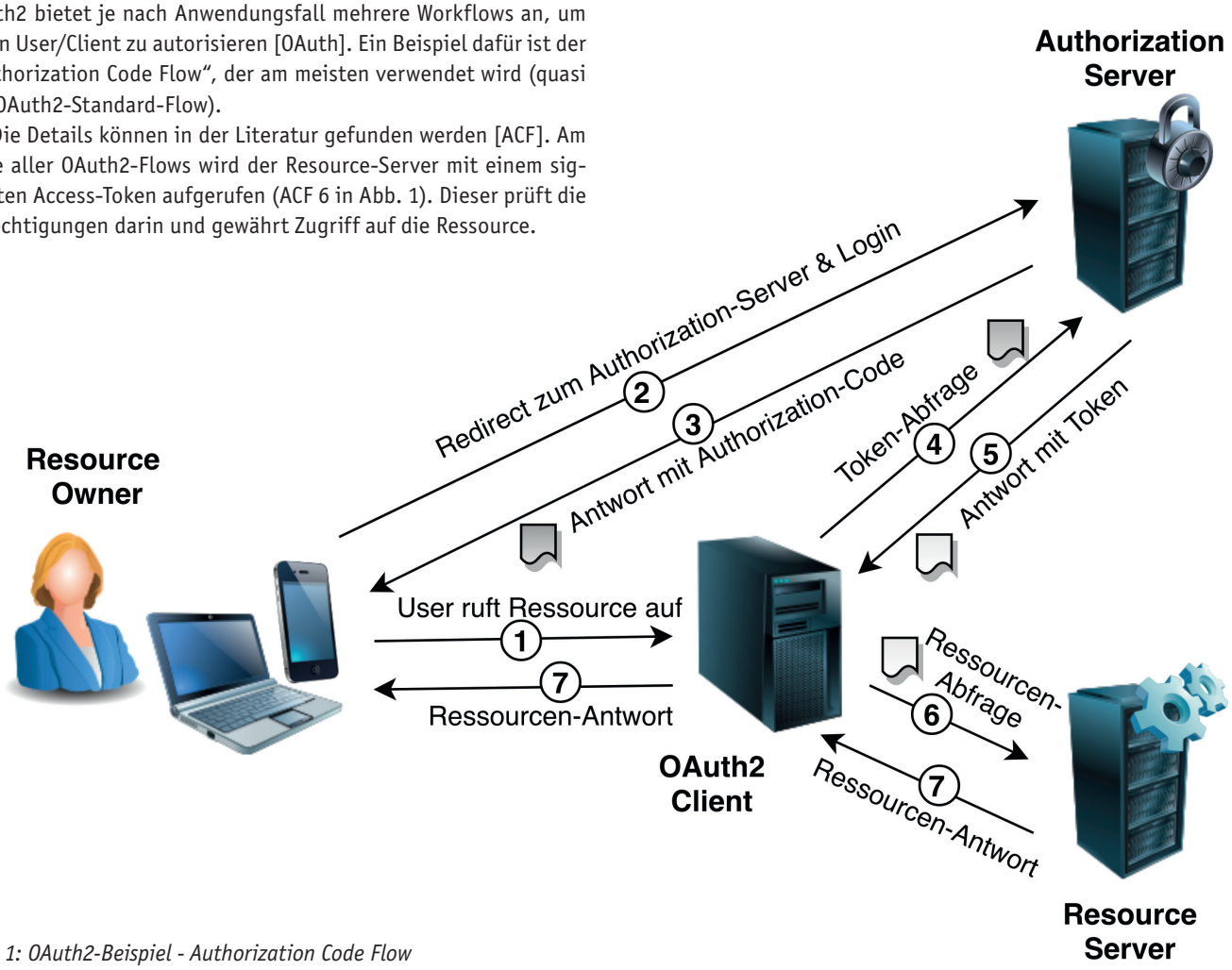

Abb. 1: OAuth2-Beispiel - Authorization Code Flow

Marcos Scholtz ist IT-Berater und Architekt bei der Cofinpro AG mit über 10 Jahren Projekterfahrung überwiegend im Java-Umfeld. Aktuell beschäftigt er sich mit der Einführung von Verticals in der Online-Plattform „MeinInvest“, in der er vor Kurzem die Standardisierung der REST-Schnittstellen realisiert hat.
E-Mail: marcos.scholtz@cofinpro.de

stellt. Nach besten REST-Konventionen [REST] müsste dieser Endpunkt wie folgt aussehen:

```
GET /api/customer/{customerId}
```

Dieser Endpunkt bekommt eine Kundennummer als Parameter und einen Access-Token im Request-Header. Als Erstes muss geprüft werden, ob der User Zugang zu diesem Endpunkt hat. Dazu werden die Rollen benutzt, die dem User zugewiesen sind. Das alleine reicht aber nicht, denn der User könnte jede beliebige Kundennummer im Request liefern. Es muss also zusätzlich geprüft werden, ob die abgefragte Kundennummer auch zum Token passt.

```
@Context
private SecurityContext securityContext;

@GET
@Path("/api/customer/{customerId}")
@RolesAllowed({"customer"}) // (1) Endpunkt-Zugang
public Customer getCustomer(
    @PathParam("customerId") @NotNull String customerId) {
  // (2) Kundennummer aus dem Token lesen
  String tokenCustomerId = securityContext.getUserPrincipal().getName();
  if (!tokenCustomerId.equals(customerId)) {
    throw new ForbiddenException("Only allowed to access own data");
  }
  return customerBean.getCustomer(customerId);
}
```

Listing 1: Einfach gesicherter REST-Endpunkt

Die meisten OAuth2-Provider bieten eine Integration mit Java EE an. Dabei werden die Daten in dem Access-Token einfach in Java EE-Klassen übersetzt, wie den „java.security.Principal“, sowie die Rollen aus dem Token verwendet. So kann man die normale JAX-RS-Sicherung von REST-Endpunkten verwenden (s. Listing 1):

1. Die Annotation `@RolesAllowed` ist Teil der Spezifikation JSR 250 [JSR250, SecAn] und berechtigt in dem Fall nur User mit der Rolle „customer“, diesen Endpunkt aufzurufen. Diese Code-nahe Syntax ist viel einfacher und weniger fehleranfällig als das Standard-Mapping der Rollen in der „web.xml“.
2. Der „subject“ vom Token (die Kundennummer) wurde vom OAuth2-Adapter im „Principal“ kopiert und kann verwendet werden, um die angegebene Kundennummer zu validieren.

Neue Zielgruppe: der Kundenberater

Erweitern wir das Beispiel um eine neue Zielgruppe: den Kundenberater. Angenommen, wir bieten zusätzlich zum Kundenportal eine interne Anwendung an, in der die Kundenberater deren Kunden verwalten können. Dies bedeutet konkret, dass ein Kundenberater sich in die Anwendung einloggen kann und seinen eigenen Token bekommt. In diesem Token ist jetzt eine andere Rolle vorhanden, zum Beispiel „adviser“, und seine eigene Berater-Nummer als „subject“. Jeder Kundenberater darf nur seine eigenen Kunden verwalten, was die Autorisierung entsprechend komplizierter macht (s. Listing 2).

```
@Context
private SecurityContext securityContext;

@GET
@Path("/api/customer/{customerId}")
@RolesAllowed({"customer", "adviser"}) // Zugang fuer zwei Rollen
public Customer getCustomer(
    @PathParam("customerId") @NotNull String customerId) {
  // Subject vom Token lesen, ist das eine Kundennr oder Beraternr?
  String tokenUserId = securityContext.getUserPrincipal().getName();
  if (securityContext.isUserInRole("customer")) {
    if (!tokenUserId.equals(customerId)) {
      throw new ForbiddenException("Only allowed to access own data");
    }
  } else if (securityContext.isUserInRole("adviser")) {
    if (!customerBean.customerPertainsToAdviser(
        customerId, tokenUserId)) {
      throw new ForbiddenException(
        "Only allowed to access own customers");
    }
  }
  return customerBean.getCustomer(customerId);
}
```

Listing 2: Gesicherter Endpunkt mit zwei Zielgruppen

Der Endpunkt erlaubt jetzt den Zugang für zwei Rollen. Aber die Autorisierung ist Rollen-abhängig, was zu den Weichen im Code führt. Das Ganze wird immer unübersichtlicher und fehleranfälliger, je mehr Zielgruppen wir dazunehmen. Meistens ist hier noch mehr Logik enthalten, zum Beispiel fürs Logging. Man könnte natürlich den Code etwas organisieren, die Autorisierung in getrennte Methoden nach Rolle verlagern, ausgeklügelte eigene Annotationen erfinden usw. Die Komplexität wird aber nur verlagert.

Getrennte APIs nach Request-Ursprung

Anhand dieses Problems bietet sich die Alternative an, Endpunkte pro Zielgruppe zu bauen. Das ist zwar nicht mehr wirklich REST-konform, denn man hat am Ende verschiedene Endpunkte für dieselbe Ressource, bietet aber einige Vorteile.

Als Best Practice hat sich bewährt, ein Präfix für jede Zielgruppe oder „Request-Ursprung“ zu definieren. So ist an der URL klar erkennbar, wer die Zielgruppe einer Programmierschnittstelle ist, und man eröffnet weitere Möglichkeiten zur redundanten Absicherung der Endpunkte nach der URL. Das können wir als Regel definieren:

- *Regel 1: APIs sollen ein Präfix entsprechend des Ursprungs in der URL haben.*

Nehmen wir in unserem Beispiel für die Kunden-APIs das Standard-Präfix „api“ und für die Berater-APIs das Präfix „adviser“, sehen die beiden REST-Endpunkte so aus:

```
GET /api/customer
GET /adviser/customer/{customerId}
```

Ein erster wichtiger Punkt: Im Kunden-API ist keine Kundennummer mehr als Parameter vorhanden, denn man kann immer davon ausgehen, dass diese Programmierschnittstelle einen Token bekommt, in dem die Kundennummer vorhanden ist. Da man hier keine Validierung mehr braucht, werden Fehler viel unwahrscheinlicher. Ein Verbot solcher IDs in Requests (sei es im Pfad, Query-Parameter oder Request-Body) können wir auch als generische Regel aufnehmen:

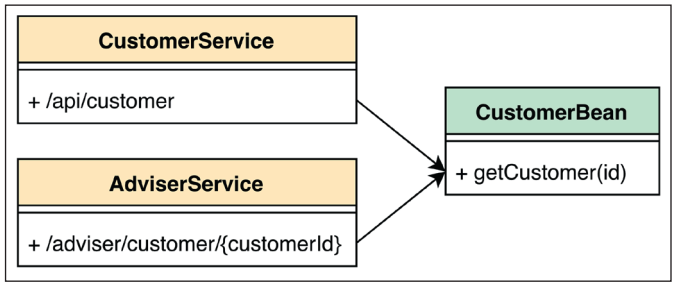

Abb. 2: Trennung von REST-API und Business-Logik

- *Regel 2: In den Token vorhandene IDs sind in den Requests nicht erlaubt.*

Um Code-Duplizierung zu vermeiden, sollte der Entwickler aber bei so einer Architektur auf keinen Fall die Business-Logik in dem Endpunkt selbst einbauen. Die Endpunkte sollen nur Authentifizierung und Autorisierung regeln und Business-Logik aufrufen, die woanders ist, zum Beispiel in einer injizierten Java-Bean (s. Abb. 2). Die Autorisierungslogik kann so für jede Zielgruppe getrennt gelöst werden und für alle Endpunkte der Gruppe wiederverwendet werden. Das schafft Ordnung und Klarheit.

Die Trennung der URLs mit Präfixen bietet auch einen anderen entscheidenden Vorteil: Es wird möglich, eine bestimmte API-Gruppe zusätzlich redundant zu sichern, zum Beispiel durch Firewall-Regeln oder IP-Sperren. Im Beispiel könnte man eine IP-Sperre einrichten und für die „/adviser"-URLs und Zugriff nur aus einer bestimmten IP-Range erlauben, in der die Kundenberater arbeiten. Das führt zur nächsten Regel:

- *Regel 3: APIs mit Zugangssperren je Zielgruppe redundant absichern.*

Aber wie sieht es aus, wenn der aufgerufene Service andere Daten braucht, die von einem anderen Service verwaltet werden? Also konkret, der Service muss einen anderen Service per REST aufrufen. Im Grunde ist der Request-Ursprung derselbe, zum Beispiel der Kunde, und wir sind im Besitz von seinem Token. Also reichen wir den Token einfach weiter. Der Token steckt im „Authorization"-Header im Request, also liest man den Header aus (z. B. mittels der `@HeaderParam`-JAX-RS-Annotation), um diesen im REST-Client wieder als Header einzufügen. Für den aufgerufenen Service sieht es so aus, als ob der Kunde selbst den Aufruf gemacht hat.

Abbildung 3 zeigt das Beispiel, dass der Kunde seine eigenen Daten aufruft, inklusive der Summe aller seiner Rechnungen. Der Kunde ruft den ersten Service nur mit seinem Token auf, ohne Kundennummer im Request (die vom Token wird verwendet). Zusätzlich ruft der Service einen zweiten Service auf und übergibt den Token des Kunden. Beide Endpunkte in dieser Aufrufkette verhalten sich gleich und sind „Kunden-Endpunkte", auch wenn der zweite nicht direkt vom Kunden aufgerufen wird. Deswegen reden wir vom *Request-Ursprung* als Kriterium, um die APIs aufzuteilen. In diesem Fall ist der Kunde der Ursprung.

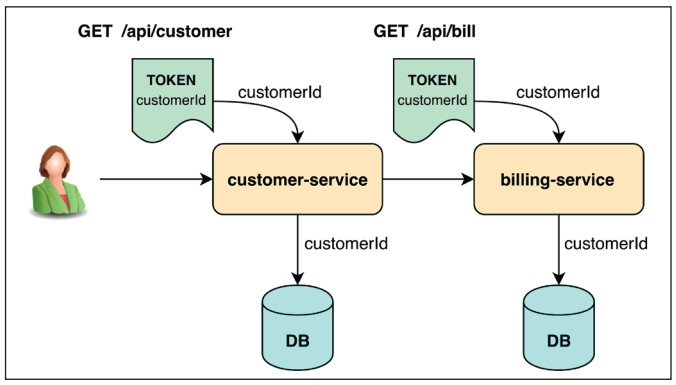

Abb. 3: Aufruf über zwei Services hinweg

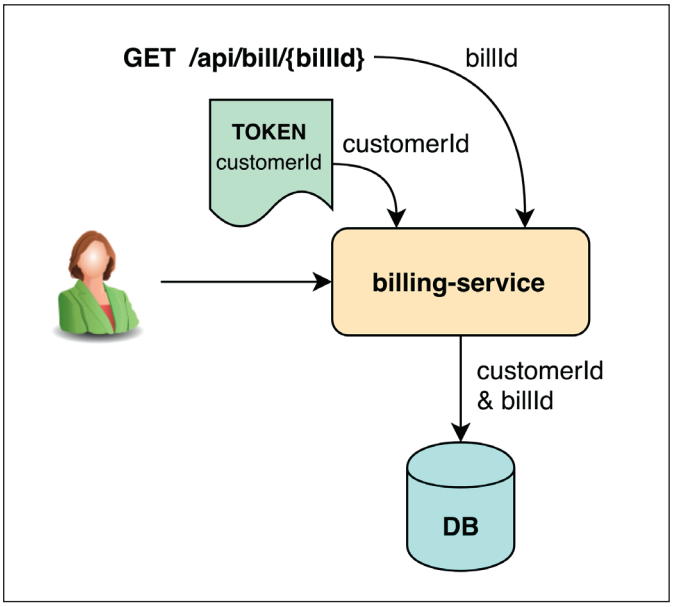

Abb. 4: Zusammenführen von IDs aus Request und Token

Dateneigentum sicherstellen

Für manche Ressourcen wird aber keine ID im Token vorhanden sein. Beispielsweise möchte ein Kunde die Details zu einer seiner Rechnungen laden. Der Endpunkt könnte so aussehen:

```
GET /api/bill/{billId}
```

Die Rechnungsnummer muss im Request stehen (im Beispiel als Pfad-Parameter), denn diese ist nicht im Token. Nun sollte der Entwickler sicherstellen, dass die Rechnung auch zum Kunden gehört. Sollte diese Prüfung vergessen werden, könnte der Kunde beliebige Rechnungen, unter anderem auch von anderen Kunden, aufrufen. Die Lösung ist, die Kundennummer aus dem Token zu verwenden und gegen die Rechnung zu validieren. Oder gar die Kundennummer bis zur Datenbankabfrage mitzunehmen, sodass nur die Rechnung mit der richtigen Rechnungsnummer und der richtigen Kundennummer geliefert wird (s. Abb. 4).

Um Fehler in diesem Zusammenhang zu vermeiden, können wir die Verwendung der ID aus dem Token zur Pflicht machen:

- *Regel 4: Es muss immer mindestens eine ID aus dem Token zur Validierung verwendet beziehungsweise mit dem Request kombiniert werden.*

Diese vier Regeln gelten natürlich für alle APIs. Im Beispiel des Kundenberater-API ist es die Beraternummer, die niemals in Requests enthalten sein darf und die immer aus dem Token verwendet werden muss. So stellt man sicher, dass jeder Berater nur auf seine eigenen Kunden und Daten zugreifen kann (s. Abb. 5).

Wie immer gibt es Ausnahmen. Ein Administrator-API zum Beispiel erlaubt Zugriff auf alle Daten und benötigt keine Validierung gegen eine ID aus dem Token.

Sonderfall: die Service API

Es gibt Fälle, in denen ein REST-Aufruf keinen User als Ursprung hat. Betrachten wir zum Beispiel einen nächtlichen Batch-Job in unserem Kunden-Service, der an alle Kunden mit offenen Rechnun-

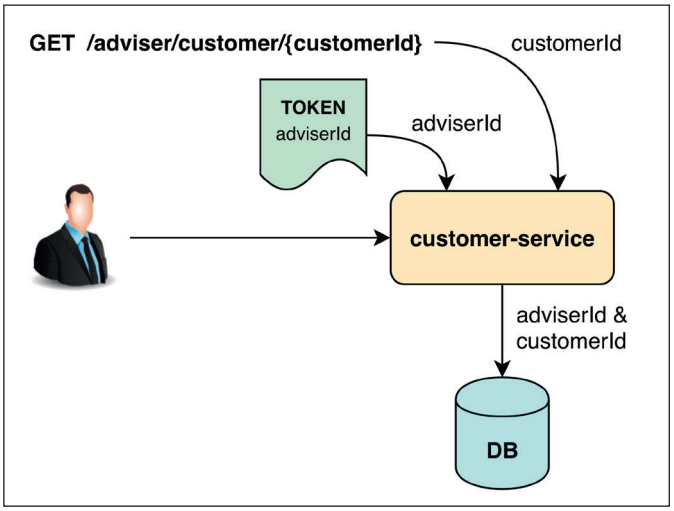

Abb. 5: Kombinierung der Beraternummer aus dem Token

gen eine E-Mail verschickt. Um die Rechnungen per REST abzufragen, braucht der Service einen Token. Das bedeutet konkret, dass sich der Service selbst einloggen (mit einem technischen User), einen Token holen und mit diesem Token seine Aufrufe tätigen muss. Der Request-Ursprung ist jetzt der Service selbst, also definieren wir eine neue API-Gruppe, die Service-APIs, mit dem Präfix „service". Endpunkte in dieser Gruppe erlauben nur den Zugriff für bestimmte technische Rollen, die nur Services selbst besitzen.

Der Login selbst (s. Abb. 6) gestaltet sich viel einfacher, denn OAuth2 bietet dafür den „Client Credentials Flow" an. Der Service beantragt beim Autorisierungsserver selbst einen Token, mit Angabe seiner Zugangsdaten (clientId und secret) und bekommt in der Antwort seinen eigenen Access-Token. Es muss noch an weitere Aspekte gedacht werden, wie Logout oder Token-Refresh, aber wie bei allen API-Gruppen können solche Probleme einmalig gelöst und für alle APIs der Gruppe verwendet werden.

Regelwerk und Integration in der CI-Pipeline

Ein großer Vorteil der Trennung der APIs nach Ursprung ist, dass die APIs für einen bestimmten Ursprung immer gleich abgesichert werden. Die generischen Regeln, die wir identifiziert haben, können pro API-Gruppe sehr genau und detailliert festgelegt werden. Werden diese Regeln überall befolgt, gibt es kaum eine Möglichkeit, eine Sicherheitslücke einzubauen. Allen Beteiligten im Projekt ist klar, wie die API-Endpunkte gesichert werden sollen, und es entstehen keine Sonderlösungen je Team oder Entwickler. Zusätzlich können anfällige APIs (mit sehr offenen Berechtigungen) durch extra Redundanzen gesichert werden.

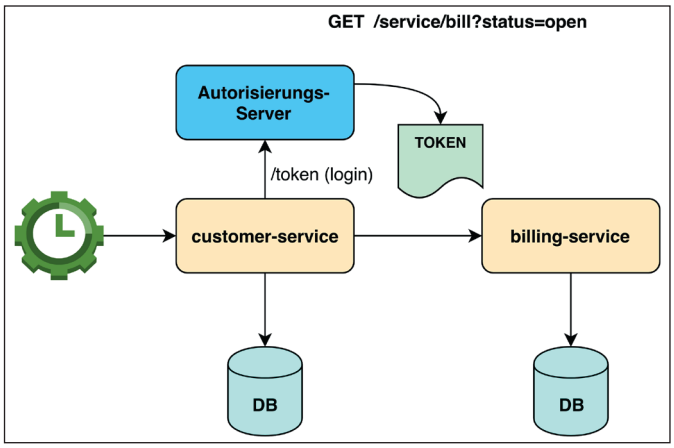

Abb. 6: Service-Login innerhalb eines Batch-Jobs

Also besteht der erste Schritt darin, ein Regelwerk für die API-Absicherung zu definieren, und dieses im Projekt für alle beteiligten Teams und Entwickler festzulegen. Tabelle 1 zeigt ein Beispiel für ein solches Regelwerk.

Das Regelwerk bringt Standardisierung für die APIs, aber nur solange die Regeln eingehalten werden. Durch wiederverwendbaren Code oder Annotationen (zum Beispiel in einer internen Bibliothek) kann man den Entwicklern die Arbeit ersparen und ihnen dazu verhelfen, den Standard zu verwenden. Zum Beispiel kann man durch einen JAX-RS-Request-Filter die entsprechende ID aus dem Token holen und sie per CDI-Producer injizierbar machen.

Eine absolute Garantie, dass keiner eine Rolle vergessen oder doch einen Fehler machen könnte, ist das nicht. Die Regeln können wir aber auch durch statische Code-Analyse-Werkzeuge prüfen lassen, wie zum Beispiel SonarQube. SonarQube ist sehr einfach mit eigenen Regeln zu erweitern [Sonar]. Sinnvolle Prüfungen könnten sein:

- Ob das URL-Präfix zur API-Gruppe passt.
- Ob kein anderes „Präfix-reserviertes" Wort in der URL auftaucht.
- Ob die JSR-250-Annotation vorhanden ist und die Rollen zur API-Gruppe passen.
- Ob die verbotenen Token-IDs in den Requests auftauchen (Pfad, Query-Param, Body-Inhalt).
- Ob die erforderlichen Token-IDs zum Abgleich in der REST-Methode aus dem Token geholt werden.

Das alles setzt voraus, dass SonarQube auch erkennen kann, zu welcher API-Gruppe ein REST-Endpunkt gehört, was man durch eine Markierung an den REST-Endpunkten (Annotationen) oder durch eine Konvention der Package-Namen realisieren kann. Daraus ergibt sich eine sechste Prüfung:

- Ob die REST-Klasse in der richtigen Package-Struktur ist oder korrekt annotiert ist.

	Kunden-API	Berater-API	Admin-API	Service-API
Ursprung	Kunde	Kundenberater	Administrator	Services/ Skripte
Präfix	/api	/adviser	/admin	/service
Datenzugriff	nur *eigene* Daten	nur Daten *eigener* Kunden	alle Daten	alle Daten
Rollen	customer	adviser, sales	admin	technische Rollen
ID Handling	Kundennummer (customerId) verboten in Requests, erforderlich aus dem Token zur Validierung	Beraternummer (adviserId) verboten in Requests, erforderlich aus dem Token zur Validierung	alle Ids in Requests erlaubt, keine Id aus dem Token zur Validierung	alle Ids in Requests erlaubt, keine Id aus dem Token zur Validierung
Redundanz	keine	IP-Sperre (nur bekannte IPs haben Zugriff)	nur aus dem internen Netzwerk erreichbar	nur aus dem internen Netzwerk erreichbar

Tabelle 1: Beispiel eines Regelwerks für die API-Absicherung

In einem großen Projekt mit vielen REST-Schnittstellen ist es schon erstaunlich, wie viele Fehler plötzlich entdeckt werden, nachdem solche Regel-Überprüfungen erstellt und aktiviert worden sind. Ist SonarQube in der CI-Pipeline integriert, kann man kein Projekt mehr überhaupt bauen, solange gegen eine dieser Regeln verstoßen wird. Die CI-Pipeline kann für Sonderfälle und kritische APIs, die nicht durch das Regelwerk allein geprüft werden können, zusätzlich mit automatisierten Schnittstellen-Tests erweitert werden, zum Beispiel mittels „Rest Assured" [RestAs].

Fazit

Die Auftrennung der REST-APIs nach Request-Ursprung ist am Anfang mit Mehraufwand und Kosten verbunden. Man muss teilweise für jede Ressource mehrere, verschiedene APIs erstellen, warten, und testen.

Die vielen Endpunkte zu warten, ist aber deutlich günstiger, denn diese sind alle weitaus simpler und folgen einem Standard. Hat man zusätzlich alles bis zur CI-Integration erreicht, kann man relativ zuversichtlich sein, dass es keine Sicherheitslücken in den eigenen REST-APIs mehr gibt und dass keine neuen Lücken eingebaut werden können. Eine 100-prozentige Sicherheit kann es selbstverständlich nie geben, aber die Erfahrung hat gezeigt, dass die Standardisierung alleine und das gemeinsame Verständnis vom Aufbau der APIs die meisten Fehlerquellen eliminiert und die Überprüfung der Regeln durch statische Code-Analysen die restlichen „Aufmerksamkeitsfehler" verhindert. Die Möglichkeit der redundanten Sperrung von bestimmten Endpunkten (Firewall) erhöht die Sicherheit noch einmal wesentlich. Also übersteigt am Ende der Nutzen bei Weitem die Kosten.

Ein starkes Sicherheits-Augenmerk bei der Konzeption von REST-APIs bleibt aber wie gehabt erforderlich, denn nicht alles kann durch diese Standardisierung aufgegriffen werden, wie sensible Daten in Response-Payloads usw. Die häufigsten Fehlerquellen können aber durch diese Methode eliminiert werden.

Links

[ACF] Authorization Code Flow, https://www.oauth.com/oauth2-servers/server-side-apps/authorization-code/

[JSR250] Java Specification Request 250: Common Annotations for the Java™ Platform, https://jcp.org/en/jsr/detail?id=250

[JWT] JSON Web Token, https://jwt.io/introduction/

[KC] Keycloak, https://www.keycloak.org/documentation.html

[OAuth] Request of Comments 6749: The OAuth2 2.0 Authorization Framework, https://tools.ietf.org/html/rfc6749

[PKCE] Request of Comments 7636: Proof Key for Code Exchange by OAuth Public Clients, https://tools.ietf.org/html/rfc7636

[REST] REST Resource Naming Guide, https://restfulapi.net/resource-naming/

[RestAs] Rest Assured, http://rest-assured.io/

[SecAn] Securing JAX-RS resources using Annotations, https://www.ibm.com/support/knowledgecenter/SS7JFU_8.5.5/com.ibm.websphere.express.iseries.doc/ae/twbs_jaxrs_impl_securejaxrs_annotations.html

[Sonar] Writing Custom Java Rules 101, SonarQube, https://docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101