

Security mit Quarkus

Bisher waren unsere REST-Services immer für alle erreichbar, die Zugriff auf den Rechner hatten.

Absichern der Services ist über mehrere Wege möglich.

Folgende Varianten werden hier behandelt:

- Benutzerkonfiguration in Properties-Files
- Benutzerkonfiguration in der DB
- JWT
- OAuth2 / KeyCloak

Ausgangsprojekt aufbauen

- Git-Projekt klonen und alle Branches downloaden:
<https://github.com/aisge/securitydemo.git>
- Datenbank starten
- Security Policy definieren mittels Annotationen:
 - GET-Methoden: `@RolesAllowed("user")`
 - UPDATE-Methoden: `@RolesAllowed("admin")`

Elytron - Properties File (ToDo)

Elytron JDBC (ToDo)

Security mittels JWT

Die Account-bezogenen Services sollen via RBAC mit JWT
abgesichert werden.

Was ist JWT

- JWT = JSON Web Token
- Nach [RFC 7519](#) genormtes Access Token
- Enthält alle wichtigen Informationen über eine Entität, deshalb
 - ist nicht jeweils eine neue Datenbankabfrage notwendig
 - muss die Sitzung nicht am Server gespeichert werden
(stateless möglich)

Aufbau eines JWT

- JWT besteht aus 3 Teilen:
 - HEADER
 - PAYLOAD
 - SIGNATURE
- Jeweils Base64-kodiert und durch einen Punkt getrennt
(HEADER.PAYLOAD.SIGNATURE)

Aufbau eines JWT-Header

Header besteht meist aus zwei Teilen:

- alg

Verwendete Signiermethode (HS256, ES256)

- typ

Ist eigentlich immer JWT

```
{ "alg": "HS256", "typ": "JWT" }
```

Aufbau von JWT - Payload

Enthält die eigentlichen Informationen, die an die Anwendung übermittelt werden sollen. Informationen werden als Key/Value-Paare bereitgestellt. Die Schlüssel werden als **Claims** bezeichnet.

Grundsätzlich werden 3 Arten von Claims unterschieden (siehe nä. Folie). Alle Claims sind optional! So wenige wie nötig integrieren zwecks Performance!

Aufbau von JWT - Payload

3 Arten von Claims:

Aufbau von JWT - Signature

Signatur wird unter Verwendung der Base64-Kodierung des Headers und der Payload mit der angegebenen Signatur-/Verschlüsselungsmethode erstellt.

Dafür ist ein **geheimer Schlüssel** zu verwenden, der nur der Ursprungsanwendung bekannt ist.

- Stellt sicher, dass die Nachricht unterwegs nicht verändert wurde!
- Stellt sicher, dass der Absender der richtige ist (nur der hat privaten Schlüssel)

Link: [JWT-Debugger](#)

SmallRye JWT hinzufügen

Folgende Extension hinzufügen:
SmallRye JWT

```
./mvnw quarkus:add-extension -Dextensions="io.quarkus:quarkus-smallrye-jwt"
```

Konfiguration in `application.properties` ergänzen:

```
mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem  
mp.jwt.verify.issuer=http://htl.at/securitydemo  
  
smallrye.jwt.sign.key-location=privateKey.pem  
smallrye.jwt.new-token.lifespan=60  
smallrye.jwt.new-token.issuer=http://htl.at/securitydemo  
  
quarkus.smallrye-jwt.enabled=true
```

Schlüsselpaar generieren

Wir benötigen ein Schlüsselpaar mit RSA256 als Hash-Algorithmus:

Konfiguration in `application.properties` ergänzen:

```
openssl genrsa -out publicKey.pem  
openssl pkcs8 -topk8 -inform PEM -in publicKey.pem  
    -out privateKey.pem -nocrypt  
openssl rsa -in publicKey.pem -pubout -outform PEM  
    -out publicKey.pem
```

Alternative: [Online-Generator](#)

Neue Resource: AuthResource

Login-Methode prüft UserId und Passwort und generiert ein neues Token, wenn die Credentials korrekt sind:

```
public Response login(JsonObject object) {
    String username = object.getString("username");
    String password = object.getString("password");
    if (username==null || !username.equals(password)) {
        return Response.status(Response.Status.UNAUTHORIZED).build();
    }

    long exp = Instant.now().getEpochSecond() + lifespan;
    Map<String, Object> claims = new HashMap<>();
    claims.put(Claims.upn.name(), username);
    claims.put(Claims.iss.name(), issuer);

    String token = Jwt
        .claims(claims).groups("customer").sign();
    String entity = Json.createObjectBuilder()
        .add("token", token).add("expires_at", exp)
        .build().toString();
    return Response.ok().entity(entity).build();
}
```

Neue Resource: AuthResource

Die Gültigkeitsdauer der Tokens und Issuer könnten von der application.properties injected werden:

```
@Inject  
@ConfigProperty(name="smallrye.jwt.new-token.lifespan")  
long lifespan;  
  
@Inject  
@ConfigProperty(name="mp.jwt.verify.issuer")  
String issuer;
```

Absichern der Resourcen

Auf Klassen- oder Methodenebene kann per Annotation festgelegt werden, welche Rolle(n) benötigt werden, um eine Methode auszuführen:

```
@RolesAllowed("customer")
```

Zusätzlich können Claims direkt Injected werden, um diese dann für Checks heranzuziehen:

```
@Inject  
JsonWebToken jwt;  
  
@Inject  
@Claim(standard = Claims.upn)  
long upn=1;
```

JWT in Angular-Client / Interceptor

Nachdem das Token vom erfolgreichen Login-Aufruf erhalten wurde wird es meist im LocalStorage abgelegt. Dieses Token muss dann bei allen Requests im Header mitgesendet werden, wofür sich ein HttpInterceptor anbietet:

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>,
            next: HttpHandler): Observable<HttpEvent<any>> {
    const idToken = localStorage.getItem('id_token');

    if (idToken) {
      const cloned = req.clone({
        headers: req.headers.set('Authorization', 'Bearer ' + idToken)
      });

      return next.handle(cloned);
    }
    else {
      return next.handle(req);
    }
  }
}
```

JWT in Angular-Client / Interceptor

Zum Aktivieren wird der in der Datei `app.modules.ts` im Abschnitt `providers` angegeben.

```
{  
  provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor,  
  multi: true  
}
```

JWT in Angular-Client / AuthGuard

Zusätzlich kann ein AuthGuard implementiert werden. Darin könnten wir beispielsweise überprüfen, ob der Expires-Zeitstempel schon überschritten wurde. Wenn ja leiten wir auf die Login-Maske um.

Konfig der Routing-Tabelle:

```
{  
  path: 'customer', component: CustomerComponent,  
  canActivate: [AuthGuardService]  
, ...}
```

JWT in Angular-Client / AuthGuard

AuthGuard-Implementierung:

```
@Injectable({
  providedIn: 'root'
})
export class AuthGuardService implements CanActivate {

  constructor(private authService: AuthService,
              private router: Router) { }

  canActivate(): boolean {
    if (!this.authService.isLoggedIn()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }
}
```